طراحی الگوریتم

فهرست مطالب

فصل اول : مرتبه الگوریتم ها و تحلیل الگوریتم های بازگشتی

غالباً یک مسئله را با استفاده از چند تکنیک متفاوت می توان حل کرد ولی فقط یکی از آنها به الگوریتمی منجر می‌شود که بسیار سریع‌تر از بقیه است. می خواهیم روش های مختلف حل مسائل را بررسی کنیم تا بتوانیم یک راه حل بهینه برای حل مسائل پیدا کنیم. سرعت کامپیوتر هر چه بالاتر باشد و قیمت حافظه هر چقدر که کاهش یابد، کارایی همواره باید مدنظر باشد. حال با مقایسه دو الگوریتم جست و جوی ترتیبی و دودویی برای یک مسئله، اهمیت این موضوع را نشان می دهیم.

الگوریتم جست و جوی ترتیبی

می خواهیم ببینیم که آیا کلید x در آرایه S[1..n] با n کلید قرار دارد؟ اگر در آرایه وجود داشت، موقعیت آن (p) را برگرداند و اگر وجود نداشت، صفر برگرداند. در این جستجو، عنصر x را با عناصر آرایه از ابتدا به سمت انتها مقایسه می کنیم. اگر به عنصر مورد نظر برسیم، از حلقه خارج شده و شماره خانه آرایه که x در آن قرار داشت را بر می گردانیم. اگر جستجو تا انتهای آرایه انجام شود و x پیدا نشود، در این حالت متغیر حلقه از تعداد عناصر آرایه بیشتر شده است و صفر را بر می گردانیم.

```
int   seqsearch (int  n, const keytype  S[ ], keytype   x){
     index  p = 1;
     while (p <= n   &&   S[p] != x)
              p++;
     if (p > n)   p=0;
          return  p;
}
```

تذکر: نوع داده index برای متغیر صحیحی است که به عنوان اندیس به کار می‌رود.

تذکر : اگر نخواهیم روالی مقادیر را از طریق آرایه برگرداند، آن آرایه را با واژه const معرفی می‌کنیم.

الگوریتم جست و جوی دودویی

با فرض این که به دنبال x هستیم، الگوریتم ابتدا، x را با عنصر میانی آرایه مقایسه می‌کند. اگر مساوی بود، الگوریتم به پایان می‌رسد. اگر x کوچک‌تر از عنصر میانی بود، باید در نیمه نخست آرایه باشد (اگر وجود داشته باشد) و الگوریتم جست و جو در نیمه نخست آرایه تکرار می‌گردد (یعنی x با عنصر میانی نیمه اول آرایه مقایسه می‌شود. اگر مساوی بود، الگوریتم به پایان می‌رسد و الی آخر). اگر x بزرگ‌تر از عنصر میانی آرایه بود، جست و جو در نیمه دوم آرایه تکرار می‌شد. این رویه چندین بار تکرار می‌گردد تا x پیدا شود یا معلوم گردد که x در آرایه وجود ندارد.

مثال: برای پیدا کردن عدد ۹ در آرایه مرتب زیر با روش جستجوی دودویی، به چند مقایسه نیاز است؟

1	2	3	4	5	6	7	8	9
5	9	12	20	35	50	82	88	97

ابتدا عدد 9 با عنصر وسط آرایه یعنی 35 مقایسه می شود و چون از آن کوچکتر است مقایسه به طور بازگشتی در زیر آرایه x[1..4] انجام می گیرد، یعنی با عنصر وسط این آرایه مقایسه می شود که با آن برابر است. بنابراین با دو مقایسه به نتیجه می رسیم.

الگوریتم جستجوی دودویی

در الگوریتم زیر تعیین می کنیم که آیا x در آرایه مرتب n کلیدی S[1..n] وجود دارد یا خیر. اگر وجود داشت موقعیت x در S یعنی p و اگر وجود نداشت صفر را بر می گرداند.

```
void   binsearch (int  n ,const  keytype  S[ ], keytype  x, index&  p){
        index   low, high, mid;
        low=1;  high = n; p = 0;
        while (low <= high   &&  p == 0){
             mid = [(low + high)/2];
             if (x == S[mid])   p = mid;
            else   if (x < S[mid])   high = mid–1;
                   else     low = mid+1;
        }
}
```

تذکر: اگر در آخر نوع داده، علامت & قرار دهیم یعنی پارامتر حاوی مقداری است که توسط الگوریتم بازگردانده می‌شود. (از علامت & برای آرایه استفاده نمی کنیم.)

مقایسه کار انجام شده توسط جست وجوی دودویی و جست وجوی ترتیبی

جست و جوی ترتیبی، n مقایسه انجام می‌دهد تا تعیین کند آیا x در آرایه‌ای به اندازه n وجود دارد یا خیر.

تعداد مقایسه‌های انجام شده توسط جست و جوی دودویی در یک آرایه مرتب n عنصری برابر $\lg n + 1$ می باشد.

مثال: در یک آرایه مرتب 32 عنصری ، وقتی x بزرگتر از تمام عناصر موجود در آرایه باشد، الگوریتم جستجوی دودویی 6 مقایسه انجام می‌دهد. $(\lg 32 + 1 = 6)$. ترتیب شماره عناصر مقایسه شده عبارتند از: 32 , 31 , 30 , 28 , 24 , 16 .

تذکر: در تحلیل الگوریتم‌ها به جای $\log_2$ از نماد خلاصه lg استفاده می کنیم.

مثال: هنگامی که آرایه حاوی 4 میلیارد عنصر باشد، جست و جوی دودویی تنها به 33 مقایسه و جست و جوی ترتیبی، چهار میلیارد مقایسه نیاز دارد. حتی اگر کامپیوتر قادر به کامل کردن یک بار گذر از حلقه while در عرض یک نانوثانیه باشد، جست و جوی ترتیبی 4 ثانیه زمان می‌برد تا عدم وجود x را در آرایه اعلان کند، حال آن که جست و جوی دودویی تقریباً بلافاصله به نتیجه می‌رسد.

تذکر: جست و جوی ترتیبی هنوز هم در مقیاس‌های زمانی قابل تحمل برای انسان، عمل می کند. حال به یک الگوریتم نامناسب می‌پردازیم که کار را در زمانی قابل تحمل به انجام نمی‌رساند.

هدف محاسبه جمله nاُم از دنباله فیبوناچی است. جمله اول این دنباله برابر 0 و جمله دوم برابر 1 است. جمله های بعـدی از جمـع دو جمله قبلی بدست می آید. بنابراین آن را می توان به طریق بازگشتی زیر تعریف کرد:

$$f_n = f_{n-1} + f_{n-2} \qquad \text{برای } n \geq 2$$

$$f_0 = 0$$

$$f_1 = 1$$

با محاسبه چند جمله اول داریم:

$$f_2 = f_1 + f_0 = 1 + 0 = 1$$

$$f_3 = f_2 + f_1 = 1 + 1 = 2$$

$$f_4 = f_3 + f_2 = 2 + 1 = 3$$

$$f_5 = f_4 + f_3 = 3 + 2 = 5$$

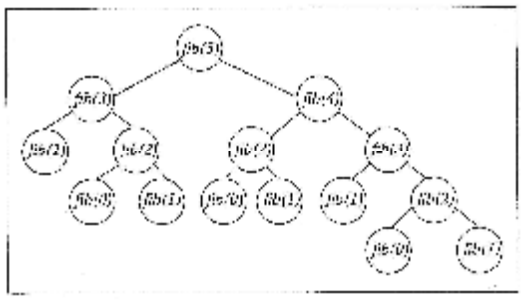
تذکر: دنباله فیبوناچی کاربردهای گوناگونی در علم کامپیوتر و ریاضیات دارد.

الگوریتم بازگشتی برای محاسبه جمله nاُم فیبوناچی

```
int  fib (int  n){
    if (n <= 1)  return  n;
   else      return   fib(n – 1) + fib(n – 2);
}
```

تذکر: کارایی این الگوریتم بسیار کم است.

شکل زیر درخت بازگشتی متناظر با این الگوریتم را برای محاسبه fib(5) نشان می‌دهد. جهت بـه دسـت آوردن fib(5) در سـطح فوقانی، به fib(4) و fib(3) نیاز داریم. برای محاسبه fib(3) به fib(2) و fib(1) و به همـین ترتیـب الـی آخـر. بنـابراین چـون مقادیر، بارها و بارها محاسبه می‌شوند ، این الگوریتم کارایی ندارد. برای مثال، fib(2) سه بار محاسبه می‌شود.



این درخت نشان می‌دهد که الگوریتم برای تعیین fib(n) به ازای $0 \leq n \leq 6$، تعداد جملات زیر را محاسبه می‌کند:

n	0	1	2	3	4	5	6
تعداد جملات محاسبه شده	1	1	3	5	9	15	25

حال آن که تعداد جملات fib(6)، حاصل جمع گره‌ها در درخت‌هایی است که ریشه آنها fib(5) و fib(4)، به علاوه یک گره در ریشه است.

در مورد هفت مقدار اول، با هر بار افزایش n به میزان 2 واحد، تعداد جملات در درخت به بیش از دو برابر افزایش می‌یابد. مثلا برای n=4 تعداد جملات درخت برابر با 9 و برای n=6 تعداد جملات درخت برابر با 25 است.

T(n) را تعداد جملات در درخت بازگشتی مربوط به n در نظر می‌گیریم. اگر با افزودن 2 به n ، تعداد جملات درخت بیش از 2 برابر شود، پس برای n که توان مثبتی از 2 است داریم:

$$T(n) > 2 \times T(n-2) > 2 \times 2 \times T(n-4) > \ldots > 2 \times 2 \times 2 \times \ldots \times T(n-n) > 2^{n/2} T(0)$$

چون T(0)=1 است، $T(n) > 2^{n/2}$ است.

تعداد جملات محاسبه شده توسط الگوریتم بازگشتی بالا ، برای تعیین جمله nاُم فیبوناچی، بزرگتر از $2^{n/2}$ است.

الگوریتم تکرار برای محاسبه جمله nاُم دنباله فیبوناچی

در الگوریتم تکرار هنگام محاسبه یک مقدار، آن را در آرایه‌ای ذخیره می کنیم تا هرگاه در آینده به آن نیاز بود، لازم نباشد دوباره محاسبه شود.

```
int  fib(int  n){
    index   i;
    int    f[0..n];
    f[0] = 0;
    if (n > 0) {
       f[1] = 1;
       for (i = 2; i <= n; i++)
             f[i] = f[i–1] + f[i–2];
     }
    return   f[n];
}
```

الگوریتم تکرار برای تعیین fib(n)، هر یک از (n+1) جمله نخست را فقط یک بار محاسبه می‌کند. پس برای تعیین جمله nاُم، (n+1) جمله را محاسبه می‌کند. الگوریتم بازگشتی بیش از $2^{n/2}$ جمله را برای تعیین جمله nاُم محاسبه می‌کرد.

با فرض اینکه کامپیوتر مورد استفاده هر جمله را در یک نانوثانیه انجام دهد، هنگامی که n برابر با 80 باشد، الگوریتم بازگشتی حداقل 18 دقیقه زمان لازم دارد و هنگامی که n برابر با 120 باشد، 36 سال زمان لازم است که از حوصله عمر انسان خارج است.

nاُم فیبوناچی را تقریباً بی‌درنگ محاسبه می‌کند.

تذکر: روش «تقسیم و حل» برای برخی مسائل، به الگوریتم‌هایی بسیار کارآمد منجر می‌شود(مانند جست و جوی دودویی)، ولی برای مسائل دیگر(مانند محاسبه جمله n ام دنباله فیبوناچی) به الگوریتم‌هایی با کارایی بسیار کم می‌انجامد. الگوریتم کارآمدی که برای محاسبه جمله nاُم فیبوناچی طرح کردیم مثالی از روش برنامه‌نویسی پویا است. بنابراین انتخاب بهترین روش بسیار مهم است.

تحلیل پیچیدگی زمانی

در تحلیل کارایی الگوریتم‌ها برحسب زمان، تعداد چرخه‌های واقعی CPU را تعیین نمی‌کنیم، زیرا به کامپیوتر خاصی که الگوریتم روی آن اجرا می‌شود، بستگی دارد. همچنین تک‌تک دستورهای اجرا شده را شمارش نمی کنیم، زیرا تعداد دستورها به نوع زبان برنامه‌نویسی که الگوریتم توسط آن پیاده‌سازی می‌شود و نحوه نوشتن برنامه، بستگی دارد. در تحلیل کارایی الگوریتم ها، به معیارهای نیاز داریم که مستقل از کامپیوتر، زبان برنامه‌نویسی و برنامه‌نویس باشد. به طور کلی، زمان اجرای یک الگوریتم با افزایش اندازه ورودی زیاد می‌شود و زمان اجرا با تعداد دفعاتی که یک عمل اصلی انجام می‌شود به عنوان تابعی از اندازه ورودی، تحلیل می شود. پس از تعیین اندازه ورودی، چند دستور یا گروه دستورها را انتخاب می‌کنیم، به طوری که کل کار انجام شده توسط الگوریتم، تقریباً متناسب با تعداد دفعاتی باشد که توسط این دستور انجام می‌شوند. این دستور یا گروه دستورها را عمل اصلی در الگوریتم می‌نامند. برای مثال در جستجوی دودویی، دستور مقایسه را عمل اصلی در نظر می گیریم.

تحلیل پیچیدگی زمانی برای حالت معمول

تحلیل پیچیدگی زمانی یک الگوریتم مشخص می‌کند که عمل اصلی، به ازای هر مقدار از اندازه ورودی چند بار اجرا می‌شود. در برخی از موارد، تعداد دفعاتی که اجرا می‌شود، نه تنها به اندازه ورودی بلکه به مقادیر ورودی نیز بستگی دارد. مثلا در الگوریتم جست و جوی ترتیبی، اگر x در نخستین عنصر آرایه باشد، عمل اصلی یک بار و اگر x در آرایه نباشد، n، بار انجام می شود. در چنین مواردی، T(n) به عنوان تعداد دفعاتی تعریف می‌شود که الگوریتم عمل اصلی را برای یک نمونه از n انجام می‌دهد. T(n) را پیچیدگی زمانی الگوریتم در حالت معمول می‌گویند و تعیین T(n) را تحلیل پیچیدگی زمانی برای حالت معمول می‌نامند.

مثال: برای الگوریتم محاسبه مجموع عناصر آرایه که در زیر آورده شده است، T(n) را مشخص کنید.

تابع sum ، تمام اعداد موجود در آرایه a[1..n] را با هم جمع می کنید.

```
number   sum (int  n, const  number  a[ ]){
        index      i; number  s=0;
        for (i = 1; i <= n; i + +)
             s = s + a[i];
        return   s;
}
```

تذکر: از نوع داده number هنگامی استفاده می‌کنیم که صحیح یا اعشاری بودن اعداد برای الگوریتم اهمیتی نداشته باشد.

عنصر به s) همواره n بار اجرا می‌شود و داریم: $T(n) = n$

تکنیک های تحلیل دیگر

بعضی از الگوریتم ها فاقد پیچیدگی زمانی برای حالت معمول می باشند.به طور نمونه عمل اصلی در الگوریتم جست وجوی ترتیبی، به ازای همه نمونه‌های n به تعداد یکسانی اجرا نمی‌شود. برای تحلیل چنین الگوریتم‌هایی از تکنیکهای تحلیل زیر می توان استفاده کرد:

1- $W(n)$: در نظر گرفتن، حداکثر تعداد دفعاتی که عمل اصلی اجرا می‌گردد.

2- $A(n)$: در نظر گرفتن، میانگین تعداد دفعاتی که عمل اصلی اجرا می‌گردد.

3- $B(n)$: در نظر گرفتن، حداقل تعداد دفعاتی که عمل اصلی اجرا می‌گردد.

تعیین $B(n)$ را تحلیل پیچیدگی زمانی در بهترین حالت و تعیین $W(n)$ را تحلیل پیچیدگی زمانی در بدترین حالت می نامند.

تذکر: اگر $T(n)$ وجود داشته باشد، (مانند الگوریتم sum)، آنگاه : $T(n)=W(n)$ و $T(n) = A(n)$ و $T(n) = B(n)$

تحلیل پیچیدگی زمانی برای الگوریتم جست و جوی ترتیبی

۱- در بدترین حالت

در بدترین حالت برای الگوریتم جست و جوی ترتیبی، عمل اصلی حداکثر n دفعه اجرا می‌شود که این در حالتی است که x آخرین عنصر آرایه باشد یا x در آرایه نباشد. بنابراین: $W(n) = n$

۲- در بهترین حالت

چون $n \geq 1$ است، حداقل یک بار گذر از حلقه باید وجود داشته باشد. اگر $x=s[1]$ باشد، n هر چه که باشد، یک بار گذر از حلقه خواهیم داشت. بنابراین داریم: $B(n)=1$.

۳- در حالت میانگین

اگر x در آرایه باشد، فرض می‌کنیم احتمال وجود آن در هر یک از خانه‌های آرایه، یکسان است و عناصر آرایه همگی متمایز هستند.

به ازای $1 \leq k \leq n$، احتمال آن که x در خانه kاُم از آرایه باشد، $\frac{1}{n}$ است. اگر x در خانه kاُم از آرایه باشد، تعداد دفعاتی که عمل اصلی انجام می‌شود تا x پیدا شود، k است. بنابراین پیچیدگی زمانی در حالت میانگین عبارت است از:

$$A(n) = \sum_{k=1}^{n}(k \times \frac{1}{n}) = \frac{1}{n} \times \sum_{k=1}^{n} k = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

به طور میانگین حدود نیمی از آرایه جست و جو می‌شود.

برای تحلیل این حالت، احتمال وجود x را در آرایه p در نظر می‌گیریم. احتمال آن که x در خانه kاُم باشد، p/n است و احتمال آن که در آرایه نباشد p – 1 است. اگر x در خانه kاُم باشد، k بار از حلقه عبور خواهیم کرد و اگر x در آرایه نباشد، n بار عبور از حلقه خواهیم داشت. بنابراین پیچیدگی زمانی میانگین عبارت است:

$$A(n) = \sum_{k=1}^{n}(k \times \frac{p}{n}) + n(1-p) = \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p) = n(1-\frac{p}{2}) + \frac{p}{2}$$

اگر p = 1 باشد، همانند حالت قبل، A(n) = (n + 1)/2 است.

اگر $p = \frac{1}{2}$ باشد، A(n) =3n/4 + 1/4 است. این بدان معناست که حدود 3/4 آرایه به طور میانگین جست و جو می‌شود.

تذکر: معمولاً تحلیل حالت میانگین از تحلیل بدترین حالت دشوارتر است. برای الگوریتم‌هایی که فاقد پیچیدگی زمانی در حالت معمول هستند، تحلیل‌های بدترین حالت و حالت میانگین را بیش از تحلیل بهترین حالت انجام می‌دهیم. تحلیل حالت میانگین از آن جهت ارزشمند است که به ما می‌گوید الگوریتم هنگامی که چندین بار روی چندین ورودی متفاوت اجرا می‌شود، چقدر زمان می‌برد. این موضوع، برای مثال، در مورد الگوریتم مرتب‌سازیی که مکرراً برای مرتب کردن همه ورودی‌های ممکن به کار می‌رود، مفید است. اگر به طور میانگین، زمان مرتب‌سازی خوب باشد، یک مرتب‌سازی نسبتاً آهسته نیز قابل تحمل خواهد بود. در الگوریتم مرتب‌سازی سریع تحلیل حالت میانگین کفایت نمی‌کند و تحلیل بدترین حالت مفیدتر است، زیرا مرز بالایی زمان صرف شده توسط الگوریتم را ارائه می‌دهد. همچنین تحلیل بهترین حالت ارزش چندانی ندارد.

توابع پیچیدگی

اگر برای یک الگوریتم خاص به پیچیدگی زمانی اشاره نشود، معمولاً از نمادهای استاندارد توابع نظیر f(n) و g(n) برای نشان دادن توابع پیچیدگی استفاده می شود. تابعی مانند f(n)=n و یا f(n)=lgn ، مثال‌هایی از توابع پیچیدگی هستند، زیرا همه آنها اعداد صحیح مثبت را به اعداد حقیقی غیرمنفی نگاشت می‌کنند.

مرتبه الگوریتم

الگوریتم‌هایی با پیچیدگی زمانی از قبیل n و 100n را الگوریتم‌های زمانی خطی می‌گویند، زیرا پیچیدگی زمانی آنها با اندازه ورودی n رابطه خطی دارد، ولی الگوریتم‌هایی با پیچیدگی زمانی n^2 و $0.01n^2$ را الگوریتم‌های زمانی درجه دوم می‌گویند، زیرا پیچیدگی زمانی آنها نسبت به اندازه ورودی n یک تابع درجه دوم است. هر الگوریتم زمانی خطی در نهایت کارایی بیشتری از الگوریتم زمانی درجه دوم دارد.

مجموعه کامل توابع پیچیدگی را که با توابع درجه دوم محض قابل دسته‌بندی باشند، $\theta(n^2)$ می‌گویند. اگر تابعی عضو مجموعه $\theta(n^2)$ باشد، می‌گوییم که تابع از مرتبه n^2 است. هنگامی که پیچیدگی زمانی الگوریتمی به $\theta(n^2)$ تعلق داشته باشد، الگوریتم را الگوریتم زمانی درجه دوم می‌گویند. مانند مرتب‌سازی تعویضی. برخی از گروه‌های پیچیدگی متداول عبارتند از:

$\theta(\lg n)$ $\theta(n)$ $\theta(n\lg n)$ $\theta(n^2)$ $\theta(n^3)$ $\theta(2^n)$

مثال: تابع $g(n) = 3n^3 + 6n^2 + 20$ از چه مرتبه ای است؟

حل: می توانیم جملاتی از مرتبه پایین را حذف کنیم. یعنی $g(n)$ از مرتبه n^2 است، یعنی $g(n) \in \theta(n^2)$.

مثال: تابع $T(n) = n(n-1)/2$ از چه مرتبه ای است؟

حل: چون داریم:

$$\frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

با حذف جمله $n/2$ که از مرتبه پایین‌تر است، ثابت می‌شود که $T(n) \in \theta(n^2)$ است.

در جدول زیر زمان اجرای الگوریتم‌هایی که پیچیدگی زمانی آنها توسط این توابع نشان داده شده، مشخص شده است. فرض می‌کنیم که پردازش عمل اصلی برای هر الگوریتم 1 نانوثانیه (10^{-9} ثانیه) به طول می‌انجامد.

n	$f(n) = n\lg n$	$f(n) = n$	$f(n) = n\lg n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.33µs	0.01 µs	0.033 µs	0.1 µs	1 µs	1 µs
100	0.007 µs	0.10 µs	0.664 µs	10 µs	1 ms	سال 4×10^{13}
10^9	0.030 µs	1 s	29.9 s	سال 31,7		

با توجه به جدول متوجه می شوید که حتی الگوریتم زمانی درجه دوم نیز برای پردازش نمونه‌ای با اندازه ورودی یک میلیارد به 31/7 سال زمان نیاز خواهد داشت. الگوریتم $\theta(n \lg n)$ برای پردازش چنین نمونه‌ای فقط 29/9 ثانیه زمان خواهد برد.

تذکر: با آگاهی از پیچیدگی زمانی، اطلاعات بیشتری به دست می‌آید. برای مثال، الگوریتم‌های فرضی با پیچیدگی زمانی $100n$ و $0.01n^2$ را در نظر بگیرید. اگر پردازش عملیات اصلی و اجرای دستورات سربار را در هر دو الگوریتم در نظر بگیریم، در آن صورت، الگوریتم زمانی درجه دوم، برای نمونه‌های کوچک‌تر از 10,000 کارآمدتر است. اگر کاربرد مورد نظر هرگز نیازی به نمونه‌های بزرگ‌تر از 10000 نداشته باشد، الگوریتم زمانی درجه دوم را باید پیاده‌سازی کرد. ولی اگر فقط بدانیم که پیچیدگی زمانی به ترتیب $\theta(n)$ و $\theta(n^2)$ است، نتیجه بالا را نمی توان گرفت. بنابراین، گاهی فقط مرتبه را تعیین می‌کنیم.

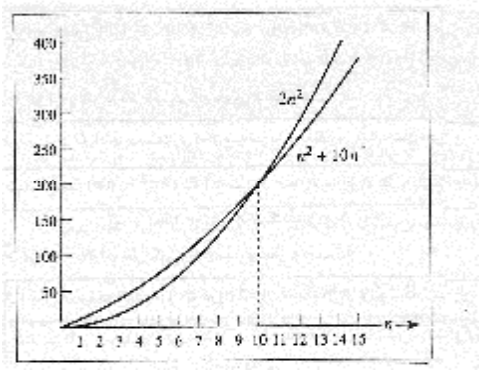
آشنایی بیشتر با مرتبه الگوریتم‌ها

در این مبحث، مفاهیمی چون O ، Ω و θ را تعریف می کنیم.

تعریف اُی بزرگ :

برای یک تابع پیچیدگی مفروض $f(n)$، $O(f(n))$ مجموعه‌ای از توابع پیچیدگی $g(n)$ است که برای آنها هم یک ثابت حقیقی مثبت c و یک عدد صحیح غیرمنفی N وجود دارد به قسمی که به ازای همه $n \geq N$ داریم: $g(n) \leq c \times f(n)$

اگر $g(n) \in O(f(n))$ باشد، می‌گوییم $g(n)$، اُی بزرگ $f(n)$ است.



گرچه n^2+10n در آغاز، بالای $2n^2$ بود،ولی به ازای $n \geq 10$، داریم: $n^2+10n \leq 2n^2$. یعنی در تعریف O می توانیم $c=2$ و $N=10$ را در نظر بگیریم.**n**

تذکر: مقدار c و N منحصر بفرد نمی باشند. به طور نمونه در مثال قبل می توان $c = 11$ و $N = 1$ را در نظر گرفت. چون به ازای $n \geq 1$ داریم: $n^2+10n \leq 11n^2$.**n**

مثال: نشان دهید : $5n^2 \in O(n^2)$

حل: چون به ازای $n \geq 0$ داریم: $5n^2 \leq 5n^2$، می توان $c = 5$ و $N = 0$ را در نظر گرفت. ■

مثال: مشان دهید که : $\frac{n(n-1)}{2} \in O(n^2)$.

حل: زیرا، به ازای $n \geq 0$ داریم: $\frac{n(n-1)}{2} \leq \frac{n^2}{2}$ می توان $c=\frac{1}{2}, N=0$ را در نظر گرفت. ■

مثال: نشان دهید: $n^2 \in O(n^2+10n)$

حل: از آن جا که به ازای $n \geq 0$ داریم: $n^2 \leq 1 \times (n^2+10n)$. می توان $c = 1$ و $N = 10$ را در نظر گرفت. ■

مثال: نشان دهید : $n \in O(n^2)$

حل: از آن جا که به ازای $n \geq 1$ داریم: $n \leq 1 \times n^2$. می توان $c = 1$ و $N = 1$ را در نظر گرفت. ■

> لازم نیست یک تابع پیچیدگی دارای جمله درجه دوم باشد تا در $O(n^2)$ باشد. فقط کافی است که در نهایت، روی نمودار، در پایین تابع درجه دوم محض قرار گیرد.

تعریف امگا

برای تابع پیچیدگی مفروض $f(n)$, $\Omega(f(n))$ مجموعه ای از توابع پیچیدگی $g(n)$ است که برای آن ها یک عدد ثابت حقیقی مثبت و c یک عدد صحیح غیر منفی N وجود دارد به قسمتی که به ازای همه $n \geq N$ داریم:

$g(n) \geq c \times f(n)$

اگر $g(n) \in \Omega f(n)$ باشد، می گوییم $g(n)$ امگای $f(n)$ است.

حل: چون به ازای $n \geq 0$ داریم: $5n^2 \geq 1 \times n^2$. می توانیم $C = 1$, $N = 0$ را در نظر بگیریم. ■

مثال: نشان دهید که : $n^2 + 10n \in \Omega(n^2)$

حل: از آن جا که به ازای $n \geq 0$ داریم: $n^2 + 10n \geq n^2$. می توانیم $C = 1$, $N = 0$ را در نظر بگیریم . ■

مثال: نشان دهید که: $T(n) = \frac{n(n-1)}{2} \in \Omega(n^2)$

حل: به ازای $n \geq 2$ داریم: $n - 1 \geq \frac{n}{2}$ ، بنابراین داریم: $\frac{n(n-1)}{2} \geq \frac{n}{2} \times \frac{n}{2} = \frac{1}{4}n^2$ یعنی می توان $C = \frac{1}{4}$, $N = 2$ را در نظر گرفت. ■

تذکر: ثابت های N,C که به ازای آن ها شرایط تعریف Ω برقرار می شود، منحصر بفرد نیستند.

اگر تابعی $\Omega(n^2)$ باشد، در نهایت تابع در نمودار، در بالای یک تابع درجه دوم محض قرار می گیرد.

مثال: نشان دهید که : $n^3 \in \Omega(n^2)$

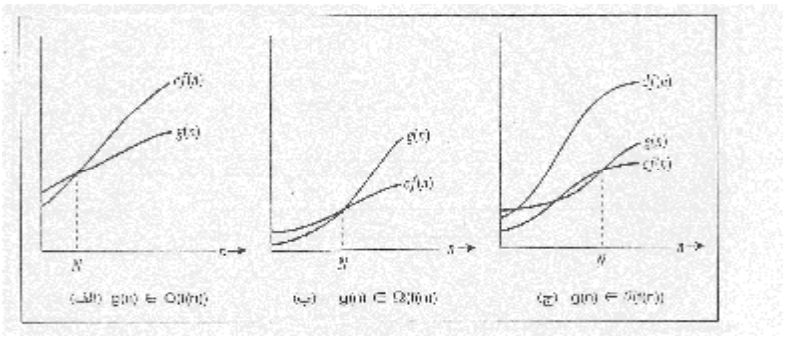
حل: اگر $n \geq 1$ باشد، داریم: $n^3 \geq 1 \times n^2$. بنابراین $C = 1$, $N = 1$. ■

تذکر: اگر تابعی هم در $O(n^2)$ و هم در $\Omega(n^2)$ باشد، در نهایت روی نمودار در پایین تابع درجه دوم محض قرار می‌گیرد.

تعریف:

برای یک تابع پیچیدگی مفروض f(n)، داریم: $q(f(n)) = O(f(n))$ **I** $\Omega(f(n))$. یعنی $q(f(n))$ مجموعه‌ای از توابع پیچیدگی g(n) است که برای آن‌ها ثابت‌های حقیقی مثبت c و d و عدد صحیح غیر منفی N وجود دارد به قسمی که: $c \times f(n) \leq g(n) \leq d \times f(n)$. اگر $g(n) \in q(f(n))$ باشد، می‌گوییم g(n) از مرتبهٔ f(n) است.

شکل زیر هر سه نماد را نشان می دهد:

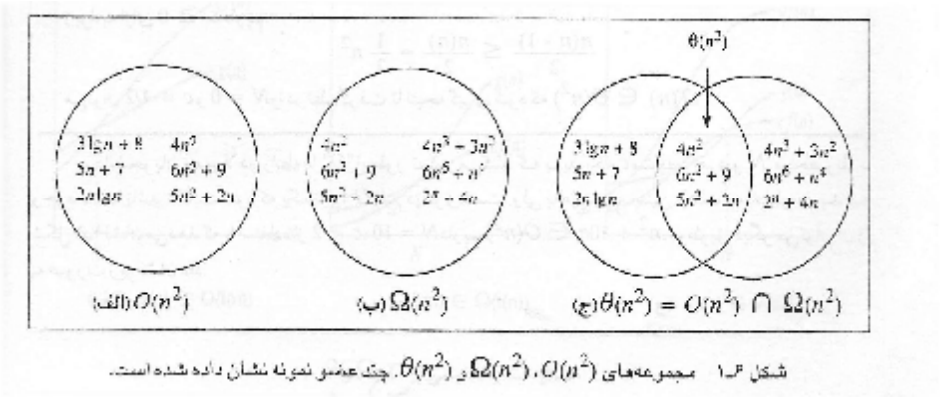


مثال: اگر $T(n) = \frac{n^2}{2} - \frac{n}{2}$ **، ثابت کنید که** $T(n) \in \theta(n^2)$.

حل: قبلا دیدیم که $T(n)$ هم در $O(n^2)$ و هم در $\Omega(n^2)$ وجود دارد. بنابراین در $\theta(n^2)$ نیز وجود دارد:

$$T(n) \in O(n^2) \cap \Omega(n^2) = q(n^2)$$

شکل زیر مجموعه‌های $O(n^2)$، $\Omega(n^2)$ و $\theta(n^2)$ و چند عضو نمونه را نشان می دهد:



شکل ۳-۱ مجموعه‌های $O(n^2)$، $\Omega(n^2)$ و $\theta(n^2)$. چند عضو نمونه نشان داده شده است.

شکل نشان می‌دهد که $\theta(n^2)$ اشتراک $O(n^2)$ و $\Omega(n^2)$ است.

در شکل تابع 5n+7 در $\Omega(n^2)$ نیست و تابع $4n^3 + 3n^2$ در $O(n^2)$ وجود ندارد. بنابراین هیچ یک از دو تابع در $\theta(n^2)$ نیستند.

$g(n) \in q(f(n))$ اگر و فقط اگر $f(n) \in q(g(n))$

برای مثال: $n^2 + 10n \in q(n^2)$, $n^2 \in q(n^2 + 10n)$. یعنی q توابع پیچیدگی را به دو مجموعه متمایز تقسیم می‌کند. این مجموعه‌ها را دسته‌های پیچیدگی می‌نامند. هر تابع موجود در یک دستهٔ مفروض می‌تواند نمایندهٔ آن دسته باشد. برای سهولت، هر دسته را معمولاً با ساده‌ترین عضو آن نشان می‌دهیم. دستهٔ پیچیدگی فوق با $q(n^2)$ نشان داده می‌شود.

تعریف اُی کوچک

برای یک تابع پیچیدگی f(n) مفروض، o(f(n)) عبارت از مجموعه کلیه توابع پیچیدگی g(n) است که این شرط را برآورده می‌سازند: به ازای هر ثابت حقیقی مثبت c، یک عدد صحیح غیر منفی N وجود دارد به قسمی که به ازای $n \geq N$ داریم: $g(n) \leq c \times f(n)$. اگر $g(n) \in o(f(n))$ باشد، می‌گوییم g(n) ای کوچک f(n) است.

مثال: نشان دهید که: $n \in o(n^2)$.

حل: با فرض c>0 ، باید یک N پیدا کنیم به قسمی که برای $n \geq N$ داشته باشیم: $n \leq cn^2$. اگر طرفین این نامعادله را بر cn تقسیم کنیم، داریم: $n \leq \frac{1}{c}$. بنابراین کافی است، هر $N \geq \frac{1}{c}$ را انتخاب کنیم. مقدار N به ثابت c بستگی دارد. برای مثال، اگر c=0,00001 باشد، باید N را حداقل مساوی 100000 در نظر بگیریم. یعنی به ازای $n \geq 100000$ داریم: $n \leq 0.00001n^2$ ■

O(f(n)) هست ولی در $\Omega(f(n))$ نیست.

تذکر: پیچیدگی زمانی برخی الگوریتم‌ها، با n زیاد نمی‌شود. دسته پیچیدگی حاوی چنین توابعی را می‌توان با هر ثابتی نشان داد که برای سهولت آن را با $q(1)$ نشان می‌دهیم.

برای مثال، پیچیدگی زمانی در بهترین حالت، یعنی B(n) برای الگوریتم جستجوی ترتیبی به ازای هر مقدار n برابر با 1 است.

ویژگی های مهم مرتبه

چند ویژگی مهم مرتبه در زیر آورده شده است.

1 - $g(n)\in O(f(n))$ اگر و فقط اگر $f(n)\in\Omega(g(n))$.

2 - $g(n)\in q(f(n))$ اگر و فقط اگر $f(n)\in q(g(n))$.

3 - اگر b>1 و a>1، در آن صورت: $\log_a n\in q(\log_b n)$.

یعنی همهٔ توابع پیچیدگی لگاریتمی در یک دستهٔ پیچیدگی قرار دارند. این دسته را با $q(1g\ \ n)$ نشان می‌دهیم.

4 - اگر b>a>0 در آن صورت: $a^n\in o(b^n)$.

یعنی همهٔ توابع پیچیدگی نمایی در یک دسته پیچیدگی قرار دارند.

5 - به ازای همهٔ مقادیر a>0 داریم: $a^n\in o(n!)$. (یعنی n! از هر تابع پیچیدگی نمایی بدتر است.)

6 - ترتیب دسته‌های پیچیدگی زیر را در نظر بگیرید:

$$q(1g\ \ n)\quad q(n)\quad q(n\,1g\ n)\quad q(n^2)\quad q(n^i)\quad q(n^k)\quad q(a^n)\quad q(b^n)\quad q(n!)$$

که در آن k>i>2 و b>a>1 است. اگر تابع پیچیدگی g(n) در دسته‌ای واقع در طرف چپ دستهٔ حاوی f(n) باشد، در آن صورت: $g(n)\in o(f(n))$

7 - اگر $g(n)\in O(f(n)),\ d>0,\ c\geq 0$ و $h(n)\in q(f(n))$ باشد، در آن صورت:

$$c\times g(n)+d\times h(n)\in q(f(n))$$

8- $f(n)+O(f(n))=\theta(f(n))$

9- $g(n)=\Omega(f(n))\Rightarrow g(n)=\Omega(O(f(n)))$

10- $f(n)+g(n)=O(\max\{f(n),g(n)\})$

مثال: ویژگی 6 بیان می‌کند که هر تابع لگاریتمی در نهایت بهتر از هر تابع چند جمله‌ای، هر تابع چند جمله‌ای در نهایت بهتر از هر تابع نمایی و هر تابع نمایی در نهایت بهتر از هر تابع فاکتوریل می‌شود. برای

$\lg n \in o(n), \qquad n^{10} \in o(2^n) \qquad , \qquad 2^n \in o(n!)$

مثال: ویژگی ۳ بیان می‌کند کـه همـهٔ توابـع پیچیـدگی لگـاریتمی در یـک دسـته پیچیـدگی قـرار دارنـد. بـرای نمونه داریم: $q(\log_4 n) = q(\lg n)$. ■

مثال: با استفاده مکرر از ویژگی‌های 6 و 7 داریم:

$5n + 3\lg n + 10n \lg n + 7n^2 \in q(n^2)$

استفاده از حد برای تعیین مرتبه

مرتبه را می‌توان با استفاده از حد تعیین کرد.

قضیه :

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \begin{cases} c & g(n) \in q(f(n)) \quad c > 0 \\ 0 & g(n) \in o(f(n)) \\ \infty & f(n) \in o(g(n)) \end{cases}$$

مثال: نشان دهید که : $\frac{n^2}{2} \in o(n^3)$

حل: با توجه به قضیه داریم:

$$\lim_{n\to\infty} \frac{n^2/2}{n^3} = \lim_{n\to\infty} \frac{1}{2n} = 0$$

مثال: نشان دهید به ازای $b>a>0$ داریم: $a^n \in o(b^n)$

حل: با توجه به قضیه داریم:

$$\lim_{n\to\infty} \frac{a^n}{b^n} = \lim_{n\to\infty} \left(\frac{a}{b}\right)^n = 0$$

این حد صفر است زیرا $0 < \frac{a}{b} < 1$: ■

قاعده هوپیتال:

اگـر f(x) و g(x) هـر دو مشـتق پـذیر باشـند و اگـر $\lim_{n\to\infty} f(x) = \lim_{n\to\infty} g(x) = \infty$، در آن صـورت هرگـاه حـد سـمت راست وجود داشته باشد، داریم: $\lim_{n\to\infty} \frac{f(x)}{g(x)} = \lim_{n\to\infty} \frac{f'(x)}{g'(x)}$

مثال: نشان دهید که $\lg n \in o(n)$.

$$\lim_{n\to\infty} \frac{\lg x}{x} = \lim_{n\to\infty} \frac{d(\lg x)/dx}{dx/dx} = \lim_{n\to\infty} \frac{1/(x \ln 2)}{1} = 0$$

$$\lim_{x\to\infty}\frac{\log x}{\log x}=\lim_{x\to\infty}\frac{d(\log_a x)/dx}{d(\log_b x)/dx}=\frac{1/(x1na)}{1/(x1nb)}=\frac{\ln b}{\ln a}>0$$

فاکتوریل

تقریب stirling برای فاکتوریل n برابر است با:

$$n!=\sqrt{2\pi n}\left(\frac{n}{e}\right)^n(1+\theta(\frac{1}{n}))$$

که e مبنای لگاریتم طبیعی است. با توجه به این تقریب داریم:

$$n!=o(n^n)\quad,\quad n!=\omega(2^n)\quad,\quad \lg(n!)=\theta(n\lg n)$$

تذکر: می توان برای هر یک از نمادهای w,o,q,Ω,O به ترتیب از علائم $>,<,=,\geq,\leq$ استفاده کرد.

تذکر: عبارت $f(n)\in q(n^2)$ و $f(n)=q(n^2)$ هر دو یک معنا دارد.

تحلیل الگوریتم های بازگشتی

تحلیل الگوریتم بازگشتی به سادگی الگوریتم های تکراری نیست. معادله بازگشتی باید حل شود تا پیچیدگی زمانی مشخص گردد. تکنیک هایی را برای حل چنین معادلات و استفاده از این راه حل ها در تحلیل الگوریتم های بازگشتی بررسی خواهیم کرد.

بررسی کارایی الگوریتم فاکتوریل

```
int  fact (int n){
   if (n==0)   return  1;
   else    return   n*fact(n-1);
}
```

برای بررسی کارایی این الگوریتم مشخص می کنیم که این تابع برای هر مقدار n چند بار دستور ضرب را انجام می دهد. برای یک n معین تعداد ضرب هایی که انحام می شود برابر است با تعداد ضرب های انجام شده در فراخوانی (n-1) به اضافه عمل ضرب n در fact(n-1) . بنابراین اگر تعداد ضرب های انجام شده برای یک مقدار معین n را با T(n) نمایش دهیم، داریم:

$$T(n)=T(n-1)+1$$

$$T(0)=0$$

چنین معادله ای را معادله بازگشتی می گویند. در این الگوریتم وقتی n=0 باشد هیچ ضربی صورت نمی گیرد. لذا $T(0)=0$ شرط اولیه است. برای حل این معادله چند مقدار اولیه می دهیم تا به عملکرد آن پی ببریم:

$$T(1)=T(0)+1=0+1=1$$

$$T(2)=T(1)+1=1+1=2$$

$$T(3)=T(2)+1=2+1=3$$

محاسبه فاکتوریل عدد 3 ، سه عمل ضرب انجام می شود. ■

مثال: بازگشتی زیر را حل کنید.(n مضربی از 2 است)

$T(n)=T(n-2)+1 \ , \ T(0)=0$

حل: چند مقدار اولیه عبارتند از:

$T(2)=T(0)+1=0+1=1$

$T(4)=T(2)+1=1+1=2$

$T(6)=T(4)+1=2+1=3$

نتیجه می شود که: $T(n)=\frac{n}{2}$.

مثال: بازگشتی $T(n)=T(\frac{n}{2})+1$ را حل کنید.($T(1)=0$ و n توانی از 2 است و $n>1$)

حل: چند مقدار اولیه عبارتند از:

$T(2)=T(1)+1=0+1=1$ $T(4)=T(2)+1=1+1=2$ $T(8)=T(4)+1=2+1=3$

نتیجه می شود که: $T(n)=2lg^n$ ■.

مثال: بازگشتی زیر را حل کنید.(n توانی از 3 است و $n>1$)

$T(n)=T(\frac{n}{3})+1 \ , \ T(1)=0$

حل: چند مقدار اولیه عبارتند از:

$T(3)=T(1)+1=0+1=1$ $T(9)=T(3)+1=1+1=2$

نتیجه می شود که: $T(n)=\log_3^n$ ■.

حل معادله بازگشتی $T(n)=T(\frac{n}{b})+1$ برابر است با : $\log_b^n$. (n توانی از b است و $T(1)=0$)

مثال: بازگشتی $T(n)=7T(\frac{n}{2})$ را حل کنید. ($T(1)=1$ و n توانی از 2 است و $n>1$)

حل: چند مقدار اولیه عبارتند از:

$T(2)=7T(1)=7$, $T(4)=7T(2)=7^2$, $T(8)=7T(4)=7^3$

نتیجه می شود که: $T(n)=7^{\lg n}$ ■.

مثال: بازگشتی $T(n)=2T(\frac{n}{3})$ را حل کنید. ($T(1)=1$ و n توانی از 3 است و $n>1$)

حل: چند مقدار اولیه عبارتند از:

$T(3)=2T(1)=2$, $T(9)=2T(3)=2^2$, $T(27)=2T(9)=2^3$

نتیجه می شود که: $T(n) = 2\log$.

حل معادله بازگشتی $T(n) = aT(\frac{n}{b})$ برابر $a^{\log_b^n}$ می باشد. (n توانی از b است و $T(1)=1$ و $a>1$)

حل معادله بازگشتی با جایگزینی

گاهی بازگشتی را می توان با جایگزینی حل کرد. مثال های زیر این روش را تشریح می کنند.

مثال: بازگشتی زیر را حل کنید.

$$T(n) = nT(n-1)$$

$$T(1) = 1$$

حل: هر معادله را به صورت زیر در معادله قبلی جایگزین می کنیم:

$$\begin{aligned} T(n) &= nT(n-1) \\ &= n[(n-1)T(n-2)] \\ &= n(n-1)[(n-2)T(n-3)] \\ &= \dots \\ &= n(n-1)(n-2)(n-3)\dots T(1) \\ &= n(n-1)(n-2)\dots 1 = n! \end{aligned}$$

مثال: بازگشتی زیر را حل کنید.

$$T(n) = T(n-1) + n$$

$$T(1) = 1$$

حل: هر معادله را به صورت زیر در معادله قبلی جایگزین می کنیم:

$$\begin{aligned} T(n) &= T(n-1)+n \\ &= T(n-2)+n-1+n \\ &= T(n-3)+n-2+n-1+n \\ &\cdot \\ &\cdot \\ &= T(1)+2+\dots+n-2+n-1+n \\ &=1+2+\dots+n-2+n-1+n \\ &= \sum_{i=1}^{n} i = \frac{n(n+1)}{2}. \end{aligned}$$

مثال: بازگشتی زیر را حل کنید.

$$T(n) = 2T(n-1) + 1$$

$$T(1) = 1$$

$$T(n) = 2T(n-1) + 1$$
$$= 2(2T(n-2)+1)+1 = 2^2 T(n-2) + 2 + 1$$
$$= 2^3 T(n-3) + 2^2 + 2 + 1$$
$$= \ldots$$
$$= 2^{n-1} T(1) + 2^{n-2} + 2^{n-3} + \ldots + 2 + 1$$
$$= 2^{n-1} + 2^{n-2} + 2^{n-3} \ldots + 2 + 1 \qquad \text{تصاعد هندسی}$$
$$= \frac{1 \times (2^n - 1)}{2-1} = 2^n - 1$$

مثال: بازگشتی زیر را حل کنید.

$$f(n) = n^2 + nf(n-1)$$
$$f(1) = a$$

حل:

$$f(n) = n^2 + nf(n-1)$$
$$= n^2 + n[(n-1)^2 + (n-1)f(n-2)]$$
$$= n^2 + n(n-1)^2 + n(n-1)f(n-2)$$
$$= n^2 + n(n-1)^2 + n(n-1)[(n-2)^2 + (n-2)f(n-3)]$$
$$= n^2 + n(n-1)^2 + n(n-1)(n-2)^2 + n(n-1)(n-2)f(n-3)$$
$$= n^2 + n(n-1)^2 + n(n-1)(n-2)^2 + \ldots + kn!$$

بنابراین $f(n) = \theta(n!)$.

حل معادله بازگشتی با استفاده از قضیه اصلی

می توان بعضی از معادله های بازگشتی را به کمک قضیه زیر که به نام قضیه اصلی شناخته می شود، حل کرد.

قضیه اصلی: فرض کنید تابع پیچیدگی $T(n)$ سرانجام غیر نزولی است و موارد زیر را بر آورده می کند:(n توانی از b است و $n>1$)

$$T(n) = aT(\frac{n}{b}) + cn^k$$
$$T(1) = d$$

که در آن $k \geq 0$ و $b \geq 2$ مقادیر صحیح a,c,d ثوابتی اند که $a > 0, c > 0, d \geq 0$ است آنگاه داریم:

$$T(n) \in \begin{cases} \theta(n^k) & a < b^k \\ \theta(n^k \lg n) & a = b^k \\ \theta(n^{\lg_b^a}) & a > b^k \end{cases}$$

تذکر: اگر در حکم بازگشتی بالا به جای = از =< استفاده شود، آن گاه در نتیجه از O به جای θ باید استفاده کرد.

تذکر: اگر در حکم بازگشتی بالا به جای = از >= استفاده شود، آن گاه در نتیجه از Ω به جای θ باید استفاده کرد.

$$T(n) = 8T(\frac{n}{4}) + 5n^2$$

$$T(1) = 3$$

حل: با توجه به قضیه اصلی، داریم a=8,b=4,k=2 و در نتیجه $a < b^k$، بنابراین $T(n) \in \theta(n^2)$. ■

مثال: بازگشتی $T(n) = 9T(\frac{n}{3}) + 5n$ را حل کنید. ($T(1) = 7$ و n توانی از 3 است و n>1)

حل: با توجه به قضیه اصلی، داریم a=9,b=3,k=1 و در نتیجه $a > b^k$، بنابراین: $T(n) \in \theta(n^{\log_3^9}) = \theta(n^2)$ ■

مثال: بازگشتی $T(n) = 8T(\frac{n}{2}) + 5n^3$ را حل کنید. ($T(64) = 200$ n توانی از 2 است و n>1)

حل: با توجه به قضیه اصلی، داریم a=8,b=2,k=3 و در نتیجه $a = b^k$، بنابراین: $T(n) \in \theta(n^3 \lg n)$ ■

اگر در قضیه قبل به جای T(1)=d از T(s)=d استفاده شود، که در آن s ثابتی به توان b است، در این صورت نتیجه قضیه اصلی برقرار است.

مثال: بازگشتی $T(n) = T(\frac{2n}{3}) + 1$ را حل کنید. (n توانی از 3 است و n>1)

حل: با توجه به قضیه اصلی، داریم $a = 1, b = \frac{3}{2}, k = 0$ و در نتیجه $a = b^k$، بنابراین

$$T(n) = \theta(n^0 \lg n) = \theta(\lg n)$$ ■

تعمیم قضیه اصلی:

در قضیه اصلی اگر به جای n^k تابع دیگری مانند f(n) باشد، یعنی داشته باشیم :

$$T(n) = aT(\frac{n}{b}) + cf(n)$$

آنگاه :

$T(n) = O(n^{\log_b^a})$	$n^{\log_b^a} > O(f(n))$
$T(n) = O(f(n))$	$n^{\lg_b^a} < O(f(n))$
$T(n) = \lg n \times O(f(n))$	$a = b$

مثال: بازگشتی $T(n) = 4T(\frac{n}{2}) + \log n$ را حل کنید.

$$n^{\log_2^4} = n^2 > \log n \Rightarrow T(n) = O(n^2)$$

مثال: بازگشتی $T(n) = 3T(\frac{n}{4}) + n \lg n$ را حل کنید.

حل:

$$n^{\lg_4^3} = n^{0.79} < n \lg n \Rightarrow T(n) = O(n \lg n)$$

مثال: بازگشتی $T(n) = 2T(\frac{n}{2}) + \lg n!$ **را حل کنید.**

حل:

$$a = b = 2 \Rightarrow T(n) = \lg n \times \lg n!$$

$$\lg n \times \lg n! < \lg n \times \lg n^n = \lg n \times n \times \lg n \Rightarrow T(n) = O(n \lg^2 n)$$

مثال: بازگشتی زیر را حل کنید.

$$T(n) = 8T(\frac{n}{9}) + n \lg n$$

حل: با توجه به صورت تعمیم یافته قضیه اصلی جواب رابطه بازگشتی ، چون $n^{\lg_9^8} < n \lg n$ ، بنابراین جواب $\theta(n \lg n)$ می باشد.

حل معادله بازگشتی با تغییر نام

مثال: بازگشتی $T(2^m) = 2T(2^{m/2}) + m$ را حل کنید.

حل: با تغییر نام $T(2^m) = S(m)$ داریم: $S(m) = 2T(\frac{m}{2}) + m$ که با توجه به قضیه اصلی داریم: $\theta(m \lg m)$. ■

مثال: بازگشتی $T(n) = 2T(\sqrt{n}) + \lg n$ **را حل کنید.**

حل: با تغییر نام $n = 2^m$ داریم: $T(2^m) = 2T(2^{m/2}) + m$ و با تغییر نام $T(2^m) = S(m)$ داریم:

$$S(m) = 2T(\frac{m}{2}) + m$$

با توجه به قضیه اصلی داریم: $\theta(m \lg m)$ و با جایگذاری $n = 2^m$ یعنی $m = \lg n$ داریم: $\theta(\lg n . \lg \lg n)$ ■

مثال: بازگشتی $T(n) = 4T(\sqrt{n}) + 1$ را حل کنید.

حل: با تغییر نام $n = 2^m$ داریم: $T(2^m) = 4T(2^{m/2}) + 1$ و با تغییر نام $T(2^m) = S(m)$ داریم: $S(m) = 4S(\frac{m}{2}) + 1$

که با توجه به قضیه اصلی داریم: $S(m) = \theta(m^{\lg_2^4}) = \theta(m^2)$.

از آنجا که $n = 2^m$ یعنی $m = \lg n$ داریم: $T(n) = \theta(\lg n)^2$ ■

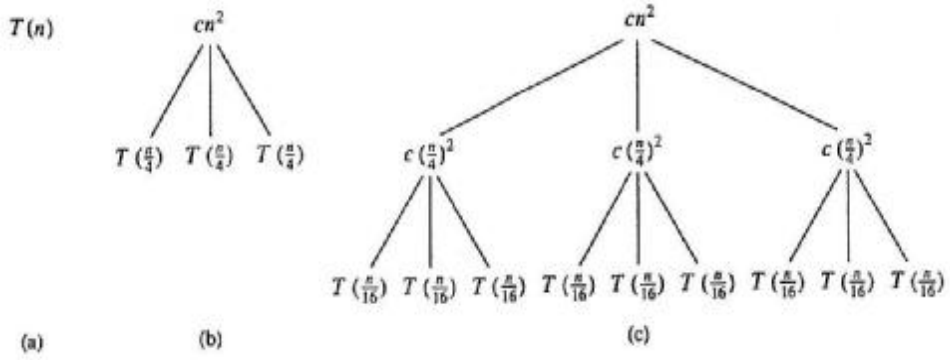
تذکر: حل روابط بازگشتی همگن در پیوست آورده شده است.

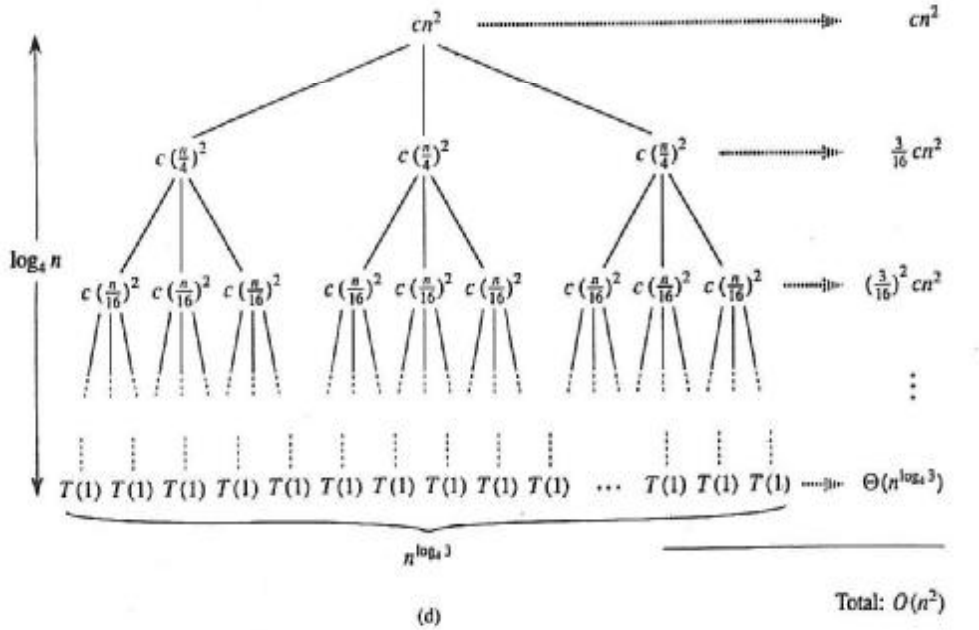
درخت بازگشت

در یک درخت بازگشت، هر گره بیانگر هزینه یک زیر مسئله در مجموعه فراخوانی های تابع بازگشتی را نشان می دهد. هزینه های داخل هر سطح درخت را جمع می کنیم تا مجموع هزینه های هر سطح را بدست آوریم، و سپس تمام هزینه های هر سطح را جمع می کنیم تا هزینه کل همه سطوح رابطه بازگشتی را تعیین کنیم. درخت های بازگشت بویژه زمانیکه رابطه بازگشتی، زمان اجرای یک الگوریتم تقسیم و حل را توصیف می کند مفید هستند.

درخت بازگشت تولید شده توسط $T(n) = aT(\frac{n}{b}) + f(n)$ یک درخت a تایی کامل با $n^{\log_b}$ برگ و ارتفاع $\log_b^n$ است.

مثال: در زیر درخت بازگشت برای رابطه بازگشتی $T(n) = 3T(\frac{n}{4}) + cn^2$ ترسیم شده است. درخت کاملا گسترش یافته در قسمت(d) دارای ارتفاع $\log_4^n$ است. این درخت $\log_4^n + 1$ سطح دارد. با توجه به درخت بازگشت به حدس $T(n) = O(n^2)$ برای رابطه بازگشتی داده شده می رسیم.





۱- کدام یک از گزاره های زیر غلط است؟

1) $f(n)+O(f(n))=\theta(f(n))$
2) $g(n)=\Omega(f(n)) \Rightarrow g(n)=\Omega(O(f(n)))$
3) $f(n)+g(n)=O(\max\{f(n),g(n)\})$
4) $g(n)\neq O(f(n)) \Rightarrow g(n)=\Omega(f(n))$

۲- توابع $h(n)=\lg^2 n, g(n)=\lg^{\lg n} n, f(n)=4^{\lg n}$ را در نظر بگیرید. کدام یک از گزاره های زیر صحیح است؟

1) $f(n)\in O(g(n)), f(n)\in \Omega(h(n))$
2) $f(n)\in \Theta(h(n)), g(n)\in \Omega(f(n))$
3) $g(n)\in \Omega(h(n)), h(n)\in \Omega(f(n))$
4) $h(n)\in O(g(n)), f(n)\in \Theta(g(n))$

۳- کدام یک از عبارات زیر درست‌اند؟

$e^{c\sqrt{n}}=O(e^{\sqrt{n}}).I \qquad c\geq 1$

$n^2=O(n\log n).II$

$n=O(n\log n).III$

1) II فقط
2) III فقط
3) I و II
4) I و III

۴- کدام یک از عبارت‌های زیر نادرست است؟

1) $O(n!)=O(n^n)$
2) $O(10^6)<O(n)<O(n.\log_2 n)$
3) هر الگوریتم از مرتبه $\theta(n)$، از مرتبه $o(n^2)$ نیز هست.
4) حذف عنصر آخر در یک لیست زنجیره‌ای یک طرفه، با داشتن اشاره‌گرهای first و last از مرتبه $o(1)$ است.

۵- کدام یک از موارد زیر نادرست است؟

1) $6*2^n+n^2=\theta(n^2)$
2) $6*2^n+n^2=\theta(2^n)$
3) $6*2^n+n^2=\Omega(2^n)$
4) $6*2^n+n^2=\Omega(n^2)$

۶- کدام یک از تساویهای زیر درست است؟

1) $n^2\log n=\theta(n^2)$
2) $6n^3/(\log n+1)=O(n^3)$
3) $n^2/\log n=\theta(n^2)$
4) $n^3 2^n+6n^2 3^n=O(n^3 2^n)$

نزدیک تر است؟

$$T(n,k) = T(\frac{n}{2},k) + T(n,\frac{k}{4}) + kn$$

$$T(*,1) = T(1,*) = a$$

1) $\log_4^k$ 2) $\log_4^{nk}$ 3) $\log_2^n + \log_4^k$ 4) $\max\{\log_2^n, \log_4^k\}$

۸- تعداد ستاره هایی که توسط الگوریتم زیر با فراخوانی A(n) چاپ می شود، چقدر است؟

```
A(int  i){
  if(i>۰) {
      A(i/۲);
      A(i/۲);
    }
  Print("*");
}
```

1) n 2) $2^{\log_2^n}$ 3) 2^n 4) $2^{\lfloor \log_2^n \rfloor + 1}$

۹- پیچیدگی زمانی الگوریتم زیر از چه درجه ای است؟

```
int   f (int  n, int  m){
   if (n<=1  ||  m<=1)   return  1;
    else    return   n+m*f(n-1,m/2);
}
```

1) n 2) $\log m$ 3) $\min(n , \log m)$ 4) $\max(n , \log m)$

۱۰- کدام یک از موارد زیر در مورد T(n) درست است؟

$$\begin{cases} T(n) = T(n-1) + \frac{1}{n}, n > 1 \\ T(1) = 1 \end{cases}$$

1) $T(n) \in O(1)$ 2) $T(n) \in O(n)$

3) $T(n) \in O(\log n)$ 4) $T(n) \in O(Lnn)$

F(x,0)= F(x+1،0) + F(x+1،1) , if x<n

F(x,1)=2F(x+1،0) + F(x+1،1) , if x<n

F(n,0)=1 , F(n,1)=0

اگر از این رابطه بخواهیم مقدار F(۱،۱) را به صورت کارا حساب کنیم، چند بار عمل " جمع" (همان + در رابطه هـای فوق) را باید انجام دهیم؟

1) $O(2^n)$ 2) $O(n^2)$ 3) $O(n)$ 4) $O(2^{n-1})$

۱۲- اگر T(n) تعداد ستاره های چاپ شده توسط Mystery(n) باشد، کدام یک از عبارت های زیر، رابطه بازگشـتی T(n) را به درستی نشان می دهند؟

```
void  mystery(int n) {
  if (n>=۲) {
      mystery (n-۱);
      print "***";
      mystery (n-۲);
      print "****";
      mystery (n-۱);
   }
}
```

1) $T(n)=1+ T(n-3)+2+ T(n-4)+1$ 2) $T(n)=5\ T(n-1)+4\ T(n-2)$

3) $T(n)=3\ T(n-1)+4\ T(n-2)$ 4) $T(n)=2\ T(n-1)+ T(n-2)+7$

۱۳- در مورد رابطه بازگشتی زیر کدام گزینه صحیح است؟

$$T(n) = 4T(\sqrt{n}) + 1$$
$$T(2) = 1$$

1) $T(n) = \frac{1}{3}(\lg n)^2 - \frac{4}{3}$ 2) $T(n) = \frac{4}{3}(4)^n - \frac{1}{3}$

3) $T(n) = \frac{4}{3}(\lg n)^2 - \frac{1}{3}$ 4) $T(n) = \frac{4}{3}(n^2) - \frac{1}{3}$

۱۴- جواب تابع بازگشتی $T(n) = 100T(\frac{n}{99}) + \lg(n!)$ **کدام است؟**

1) $\theta(n^2)$ 2) $\theta(n^{\lg_{99}^{100}})$ 3) $\theta(n \lg n)$ 4) $\theta(n \lg^2 n)$

۱۵- تابع $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n$ **را در نظر بگیرید. کدام یک از روابط زیر برای T(n) درست است؟**

1) $O(\lg\lg n)$ 2) $O(\lg n.\lg\lg n)$ 3) $O(n.\lg\lg n)$ 4) $O(n.\lg n.\lg n)$

۱۶- جواب رابطه بازگشتی $T(n) = 81T(\frac{n}{9}) + n\lg n$، کدام یک از گزینه های زیر است؟

1) $\theta(\lg n)$ 2) $\theta(n)$ 3) $\theta(n^2 \lg n)$ 4) $\theta(n \lg n)$

۱۷- **f(n)** زمان اجرای الگوریتمی مطابق رابطه بازگشتی زیر محاسبه شده است. کدام گزینه صحیح است؟

$$f(n) = n^2 + nf(n-1)$$
$$f(1) = a$$

1) $f(n) = \theta(2^n)$ 2) $f(n) = \theta(n!)$ 3) $f(n) = \theta(2^{n!})$ 4) $f(n) = \theta(n!)^2$

۱۸- مرتبه زمانی الگوریتمی با تابع زمانی زیر، برابر کدام گزینه است؟

$$T(n) = 3T(n-1) + 4T(n-2)$$
$$T(0) = 0, T(1) = 1$$

1) n^2 2) 4^n 3) $2^n \lg n$ 4) $n^4 \lg n$

۱۹- فرض کنید زمان اجرای الگوریتم روی **n** عنصر ورودی برابر **T(n)** می باشد، که به صورت زیر تعریف می شـود. در این صورت زمان اجرای الگوریتم فوق **O(n)** کدام است؟

$$T(n) = T(n-1) + n - 1$$
$$T(1) = 1$$

1) $\frac{n^2}{2} + \frac{n}{2}$ 2) $n^2 + \frac{n}{2}$ 3) $\frac{n^2}{2} - \frac{n}{2}$ 4) $n \log n - \frac{n}{2}$

۲۰- مرتبه اجرایی الگوریتم زیر کدام است؟

```
int  f (int n){
  if (n==۱)   return ۱;
  else   return f(n-۱)+f(n-۱)+۱;
}
```

1) $O(n)$ 2) $O(2^n)$ 3) $O(n \log n)$ 4) $O(2^{\frac{n}{2}})$

۲۱- رابطه بازگشتی زیر را در نظر بگیرید. در این صورت **T(n)** کدام است؟

$$T(n) = \begin{cases} 1 & if\ (n=1) \\ 2T(n-1)+1 & if\ (n>1) \end{cases}$$

1) $2^n + 2$ 2) $n \log_2^n + 1$ 3) $n^2 \log_2^n$ 4) $2^n - 1$

پیچیدگی زمانی الگوریتم فرض شود، کدام درست است؟

```
int   fibo(int  n){
    if (n<=۱)   return  n;
   else   return  fibo(n-۱)+ fibo(n-۲);
}
```

(1 $T(n) > 2^{n/2}$ (2 $T(n) > n^2$ (3 $T(n) = 2T(n-2)+1$ (4 $T(n) = 2T(n-1)+1$

۲۳- تابع بازگشتی زیر را درنظر بگیرید. پیچیدگی زمانی تابع فوق کدام است؟ (n>۱ و n توانی از ۲ است.)

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1$$
$$T(1) = 0$$

(1 $o(\lg n)$ (2 $O(n^2)$ (3 $o(n \lg n)$ (4 $o(2 \lg n)$

۲۴- الگوریتم زیر را برای جستجوی x در آرایه A شامل n عنصر درنظر بگیرد. پیچیدگی زمانی این الگوریتم چیست؟ (فرض کنیم در ابتدا down=۱ , top =n , x ورودی باشد)

```
int   search (int  down, int  top){
  int  t;
  if (down > top)  return ۰;
 else{
       t= (down + top) div ۲;
        if (x== A[t])    return   t;
        else  if (x < A[t])   return search (down , t-۱);
              else    return  search (t+۱ , top);
       }
}
```

(1 $o(\lg_{10}^{n})$ (2 $o(n \lg n)$ (3 $O(\frac{n}{2})$ (4 $o(\lg n)$

پاسخ تشریحی

4-1) هر سه گزاره زیر صحیح است:

الف) $f(n)+O(f(n))=\theta(f(n))$

ب) $g(n)=\Omega(f(n)) \Rightarrow g(n)=\Omega(O(f(n)))$

ج) $f(n)+g(n)=O(\max\{f(n),g(n)\})$

1-2) تابع $f(n)$ را می توان به صورت زیر نوشت:

$$4^{\log n}=2^{\log^{n^2}}=n^2$$

حال از هر سه تابع ، لگاریتم می گیریم:

$$f(n)=2\log n \qquad g(n)=\log n.\log(\log n) \qquad h(n)=2\log(\log n)$$

با مقایسه این توابع مشخص است که $f<g$ و بنابراین داریم: $f\in O(g)$. همچنین چون $f>h$ ، بنابراین داریم: $f\in\Omega(h)$ بنابراین گزینه یک صحیح است.

2-3) در صورتی که به جای علامت O از علامت کوچکتر استفاده شود، متوجه می شوید که فقط مورد سوم صحیح است. (می دانیم که $n<n\log n<n^2$)

4-4) گزینه های 1و2و3 درست می باشند. گزینه 4 نادرست است چون برای حذف عنصر آخر باید آدرس گره ما قبل آخر را داشت که در این حالت نیاز به پیمایش لیست می باشد.

1-5) کافی است برای بررسی صحت گزینه ها، به جای θ از علامت = و به جای Ω از علامت $\geq$ استفاده شود. همچنین به جای عبارت $6*2^n+n^2$ از عبارت 2^n استفاده کنید.

2-6) با قرار دادن علامت $\leq$ به جای نماد O ، درستی گزینه 2 مشخص است.

3-7) به طور نمونه اگر درخت $T(8,4)$ را رسم کنیم، متوجه می شویم که عمق درخت برابر 4 است. به عبارتی برابر است با:

$$\log_2^n+\log_4^k$$

نحوه محاسبه $T(8,4)$:

$$T(8,4)=T(4,4)+T(8,1)+8\times4 \qquad T(4,4)=T(2,4)+T(4,1)+4\times4 \qquad T(2,4)=T(1,4)+T(2,1)+2\times4$$

4-8) تعداد اجرای دستور داخل حلقه ها به ازای چند مقدار n در جدول زیر آورده شده است:

N	1	2	3	4	5	6	7	8	9
تعداد تکرار	3	7	7	15	15	15	15	24	24

با نگاه به این جدول، مشخص است که جواب $2^{\lfloor\log_2^n\rfloor+1}$ است.

3-9) تابع f به فرم $f(n-1,m/2)$ خود را فراخوانی می کند. بنابراین اگر n-1 زودتر به 1 برسد، از مرتبه $O(n)$ و اگر $m/2$ زودتر به 1 برسد، از مرتبه $O(\log m)$ می باشد. در نتیجه تابع از مرتبه $\min(n,\log m)$ می باشد.

4-10) با فرض $n=3$ داریم:

$$T(3) = T(2) + \frac{1}{3} = 1 + \frac{1}{2} + \frac{1}{3}$$

$$T(2) = T(1) + \frac{1}{2} = 1 + \frac{1}{2}$$

بنابراین مشخص است که T(n) سری $T(n) = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = \sum_{i=1}^{n} \frac{1}{i}$ بوده و داریم:

$$\sum_{i=1}^{n} \frac{1}{i} < \int_{i=1}^{n} \frac{1}{i} di = Lni\Big|_{i=1}^{n} = Ln(n)$$

3-11) تعداد دفعاتی که عمل جمع برای محاسبه F(1،1) انجام می شود برابر است با: $2 \times (n-2) + 1 = O(n)$

به طور نمونه به ازای n=5 برای محاسبه F(1،1) به 7 عمل جمع نیاز است:

$$F(1,1) = 2F(2,0) + F(2,1)$$
$$F(2,0) = F(3,0) + F(3,1)$$
$$F(2,1) = 2F(3,0) + F(3,1)$$
$$F(3,1) = 2F(4,0) + F(4,1)$$
$$F(3,0) = F(4,0) + F(4,1)$$
$$F(4,0) = F(5,0) + F(5,1)$$
$$F(4,1) = 2F(5,0) + F(5,1)$$

روش دوم: می توان از جدول زیر نیز برای محاسبه F(1،1) به ازای n=5 استفاده کرد. ابتدا خانه(5،0) را 1 و خانه (5،1) را 0 قرار می دهیم. سپس خانه (4،0) با جمع دو خانه پایین آن مشخص می شود و خانه (4،1) نیز از مجموع دو برابر خانه (5،0) با خانه (5،1) بدست می آید. این روال را ادامه داده تا خانه (1،1) مشخص شود. مشاهده می کنید که به 7 عمل جمع نیاز بود.

	0	1
1		2*7+10=17
2	3+4=7	2*3+4=10
3	1+2=3	2*1+2=4
4	1+0=1	2*1+0=2
5	1	0

4-12) در هر بار اجرای تابع، دو بار به ازای n-1 و یکبار به ازای n-2 به صورت بازگشتی صدا زده می شود و 7 کارکتر ستاره نیز چاپ می کند. بنابراین رابطه بازگشتی آن به صورت زیر است:

$$T(n)=2T(n-1)+T(n-2)+7$$

3-13) با تغییر نام $n = 2^m$ داریم:

$$T(2^m) = 4T(2^{m/2}) + 1$$

$$S(m) = 4S(\frac{m}{2}) + 1$$

که با توجه به قضیه اصلی : $S(m) = \theta(m^{\lg_2^4}) = \theta(m^2)$ و از آنجا که $n = 2^m$ یعنی $m = \lg n$ داریم:

$T(n) = \theta(\lg n)^2$.

با توجه به گزینه های تست، مشخص است که جواب گزینه 1 یا 3 است:

$$T(n) = c_1(\lg n)^2 + c_2$$

که با توجه به شرط مرزی $T(2)=1$ داریم:

$$1 = c_1(\lg 2)^2 + c_2 \Rightarrow c_1 + c_2 = 1$$

بنابراین جواب گزینه 3 است، چون : $\frac{4}{3} + (-\frac{1}{3}) = 1$

2-14) با توجه به اینکه $\lg(n!) = \theta(n \lg n)$ ، رابطه داده شده را به صورت زیر می نویسیم:

$$T(n) = 100T(\frac{n}{99}) + \lg(n!)$$

با توجه به صورت تعمیم یافته قضیه اصلی، $n^{\lg_{99}^{100}} > n \lg n$ بنابراین : $t(n) = \theta(n^{\lg_{99}^{100}})$.

تذکر: $n^{\lg_{99}^{100}} = n^{1.0022}$ و $n \lg n = n^{1.002}$.

2-15) در رابطه $T(n) = 2T(\sqrt{n}) + \lg n$ با تغییر نام $n = 2^m$ داریم: $T(2^m) = 2T(2^{m/2}) + m$

و با تغییر نام $T(2^m) = S(m)$ داریم: $S(m) = 2T(\frac{m}{2}) + m$

با توجه به قضیه اصلی داریم: $\theta(m \lg m)$ و با جایگذاری $n = 2^m$ یعنی $m = \lg n$ داریم: $\theta(\lg n . \lg \lg n)$

4-16) با توجه به صورت تعمیم یافته قضیه اصلی جواب رابطه بازگشتی داده شده را به دست می آوریم:

$$T(n) = 8T(\frac{n}{9}) + n \lg n$$

چون $n^{\lg_9^8} < n \lg n$ ، بنابراین جواب $\theta(n \lg n)$ می باشد.

2-17) با استفاده از روش جایگذاری با تکرار، رابطه بازگشتی را حل می کنیم:

$$\begin{aligned}
f(n) &= n^2 + nf(n-1) \\
&= n^2 + n[(n-1)^2 + (n-1)f(n-2)] \\
&= n^2 + n(n-1)^2 + n(n-1)f(n-2) \\
&= n^2 + n(n-1)^2 + n(n-1)[(n-2)^2 + (n-2)f(n-3)] \\
&= n^2 + n(n-1)^2 + n(n-1)(n-2)^2 + n(n-1)(n-2)f(n-3) \\
&= n^2 + n(n-1)^2 + n(n-1)(n-2)^2 + \ldots + kn!
\end{aligned}$$

بنابراین $f(n) = \theta(n!)$.

$T(n) = 3T(n-1) + 4T(n-2) \Rightarrow r^2 - 3r - 4 = 0$

ریشه های این معادله برابر 4 و 1- است، بنابراین جواب عمومی آن به صورت زیر است:

$T(n) = c_1 4^n + c_2 (-1)^n$

بنابراین $T(n) = \theta(4^n)$.

می توان با توجه به شرایط مرزی داده شده ضرایب را پیدا کنیم. (در این تست لازم نیست.)

3_19)

$T(n) = (n-1) + T(n-1)$

$T(n) = (n-1) + (n-2) + T(n-2)$

.....

$T(n) = (n-1) + (n-2) + (n-3) + ... + 1 = \frac{n(n-1)}{2}$

2-20) با توجه به اینکه مرتبه اجرایی تابع t(n)=at(n-b)+1 برابر $o(a^{\frac{n}{b}})$ می باشد، جواب 2^n است.

4-21) رابطه بازگشتی زیر، همان رابطه برج هانوی است که در متن درس بررسی شده است و $T(n) = 2^n - 1$.

$$T(n) = \begin{cases} 1 & if\ (n=1) \\ 2T(n-1)+1 & if\ (n>1) \end{cases}$$

1-22) مرتبه اجرای الگوریتم محاسبه جمله n ام سری فیبوناچی برابر $o(2^{\frac{n}{2}})$ می‌باشد.

3-23) با توجه به قضیه اصلی، چون $a = b^k$ داریم: $T(n) \in \theta(n \lg n)$

4-24) الگوریتم داده شده همان جستجوی دودویی می باشد.

فصل دوم . روش تقسیم و حل

در روش تقسیم و حل (divide-and-conquer) مسائل به چندین زیر مسئله که مشابه مسئله اصلی هستند اما اندازه کوچکتری دارند، می شکنند و آن زیر مسئله ها به طور بازگشتی حل می شوند و سپس این حل ها را جهت ایجاد برای مسئله اصلی ترکیب می کنند. روش تقسیم و حل یک روش بالا به پائین (top_down) است. یعنی حل یک نمونه سطح بالا از مسئله با رفتن به پائین و بدست آوردن حل نمونه های کوچکتر حاصل می شود.

تذکر: هنگام طراحی الگوریتم های تقسیم و حل معمولاً آن را به صورت یک روال بازگشتی می نویسیم.

روش تقسیم و حل

راهبرد طراحی تقسیم و حل شامل مراحل زیر است:

1- تقسیم نمونه ای از یک مسئله به یک یا چند نمونه کوچکتر

2- حل نمونه های کوچکتر(اگر نمونه های کوچکتر به قدر کافی کوچک نبودند، از بازگشت استفاده می کنیم.)

3- در صورت نیاز حل نمونه های کوچکتر را ترکیب کرده تا حل نمونه اولیه بدست آید.

جستجوی دودویی

نسخه تکراری جستجوی دودویی در فصل قبل بررسی شد. در این جا یک نسخه بازگشتی را ارائه می دهیم. این الگوریتم روش بالا به پائین را که در تقسیم و حل به کار می رود نشان می دهد. مراحل جستجوی دودویی عبارت است از:

اگر x برابر عنصر میانی بود تمام. در غیر این صورت:

1- آرایه را به دو زیر آرایه تقسیم کن که هر یک حدوداً نصف آرایه اولیه اند. اگر x کوچکتر از عنصر میانی بود، زیر آرایه چپ و اگر بزرگتر از عنصر میانی بود زیر آرایه راست را انتخاب کن.

2- با تعیین اینکه x در آن زیر آرایه است آن را حل کن. در غیر اینصورت اگر زیر آرایه به قدر کافی کوچک نیست برای حل از بازگشتی استفاده کن.

3- حل مسئله آرایه را از حل مسئله زیر آرایه بدست آور.

الگوریتم جست و جوی دودویی(بازگشتی)

الگوریتم تعیین می کند که آیا x در آرایه مرتب شده S[1..n] به اندازه n وجود دارد یا خیر. موقعیت x در آرایه توسط تابع برگردانده می شود.(اگر x در S نباشد، صفر برگردانده می شود)

```
index  bsearch(index  low , index  high){
   index   mid;
   if (low>high)   return  0;
  else{
      mid = ⌊(low + high) / 2⌋;
      if (x==S[mid])  return  mid;
      else if (x < S[mid])  return  bsearch(low,mid-1);
           else  return  bsearch(mid+1, high);
      }
}
```

در مورد الگوریتم های بازگشتی، تعداد رکوردهای فعالیت که در پشته قرار گرفته اند به عمق فراخوانی های بازگشتی بستگی دارد. برای جستجوی دودویی، پشته به عمقی می رسد که در بدترین حالت برابر $\lg n + 1$ می باشد.

الگوریتم تکراری سریع تر از نسخه بازگشتی است زیرا نیازی به نگهداری پشته ندارد.

تحلیل پیچیدگی زمانی در بدترین حالت برای الگوریتم جستجوی دودویی(بازگشتی)

عمل اصلی را مقایسه x با s[mid] در نظر می گیریم و اندازه ورودی، تعداد عناصر آرایه (n) می باشد. با فرض اینکه n توانی از 2 باشد، تحلیل را انجام می دهیم. یک راه برای آن که بدترین حالت بتواند رخ دهد هنگامی است که x بزرگتر از همه عناصر آرایه باشد، آنگاه هر فراخوانی بازگشتی باعث کاهش طول آرایه دقیقاً به نصف میزان اولیه می شود. برای مثال اگر n=16 باشد در آن صورت mid=8 است. چون x از همه عناصر آرایه بزرگتر است، هشت عنصر بالایی به عنوان ورودی فراخوانی بازگشتی اول محسوب می شود. در مرحله بعد، 4 عنصر بالایی، ورودی فراخوانی بازگشتی دوم محسوب می شود و به همین ترتیب تا آخر.

بنابراسن دستور بازگشتی آن به صورت $W(n) = W(\frac{n}{2}) + 1$ می باشد که $W(1) = 1$. جواب این رابطه $W(n) = \lg n + 1$ می باشد.

تذکر: اگر n به توانی از 2 محدود نباشد، داریم: $W(n) = \lfloor \lg n \rfloor + 1 \in \theta(\lg n)$

تذکر: جستجوی دودویی فاقد پیچیدگی زمانی در حالت معمول است.

مرتب سازی ادغامی (Merge Sort)

مرتب سازی ادغامی راه خوبی برای نشان دادن روش تقسیم و حل است. در این الگوریتم، نمونه ای از یک مسئله به نمونه های کوچکتر تقسیم می شود. دو آرایه جدید (نمونه های کوچکتر) در واقع از روی آرایه ورودی (نمونه اولیه) ایجاد می گردند. ادغام دو طرفه (tow-way merging) به معنای ترکیب دو آرایه مرتب شده در یک آرایه مرتب است. با به کارگیری مکرر روال ادغام می توان آرایه را مرتب کرد.

برای آرایه n عنصری مرتب سازی ادغامی شامل مراحل زیر می شود (برای سهولت n توانی از 2 باشد):

1- تقسیم آرایه به دو زیر آرایه ، هر یک با n/2 عنصر

2- حل هر زیر آرایه با مرتب سازی آن. اگر زیر آرایه به قدر کافی کوچک نبود برای آن از بازگشتی استفاده می کنیم.

3- ترکیب حل های زیرآرایه ها از طریق ادغام آنها در یک آرایه مرتب.

مثال: جهت مرتب سازی آرایه 8 عنصری می توان آن را به دو زیر آرایه تبدیل کرد که تعداد عناصر هر کدام 4 است. سپس دو زیر آرایه را مرتب کرده آنها را ادغام می کنیم تا یک آرایه مرتب شده ایجاد گردد. البته برای مرتب سازی آرایه 4 عنصری، آن را به همان شیوه به دو زیر آرایه 2 عنصری تقسیم می کنیم و آنها را مرتب سازی و ادغام می کنیم. که البته برای مرتب سازی آرایه 2 عنصری، آن را به دو زیر آرایه 1 عنصری تقسیم می کینم و آنها را که مرتب هستند را با هم ادغام می کنیم.

```
void   mergesort (int  n , keytype S[ ]){
      const  int   h = [n/2] , m = n – h ;
      keytype   U[1..h],  V[1..m];
      if (n > 1){
         copy  S[1..h]  to U[1..h];
         copy  S[h +1..n]  to V[1..m];
         mergesort(h, U);
         mergesort(m, V);
         merge (h, m, U, V, S);
        }}
```

در الگوریتم بالا، نیمه اول آرایه S را در آرایه U و نیمه دوم را در آرایه V کپی می شوند و هر یک از این دو آرایه به صورت بازگشتی مرتب شده و در نهایت این دو آرایه در آرایه S توسط روال merge ادغام می شوند.

الگوریتم ادغام

```
void   merge (int  h, int  m, const keytype U[ ], const keytype  V[ ],keytype  S[ ]){
      index   i=1, j=1, k=1;
     while (i<= h  &&  j<= m)
     {
        if (U[i] < V[j])  { S[k] = U[i];  i++; }
        else {S[k] = V[j];  j++;  }
        k++;
      }
     if  (i > h) copy  V[j..m]  to  S[k..h+m];
     else   copy  U[i....h]  to  S[k...h+m];
}
```

تحلیل پیچیدگی زمانی در بدترین حالت برای الگوریتم ادغام

عمل اصلی را مقایسه U[i] با V[j] در نظر می گیریم. اندازه ورودی تعداد عناصر موجود در هر یک از دو آرایه ورودی یعنی h و m می باشند. بدترین حالت، هنگام خروج از حلقه رخ می دهد، زیرا یکی از اندیس ها، مثلاً i ، به نقطه خروج خود یعنی h رسیده است. حال آن که اندیس دیگری یعنی j به m-1 یعنی یک واحد کمتر از نقطه خروج خود رسیده است. برای مثال این حالت هنگامی رخ می دهدکه m-1 عنصر اول آرایه V ، در S قرار داده شده، بعد از آنها همه h عنصر موجود در U درS قرار گیرند که در آن زمان، خروج از حلقه صورت می پذیرد، زیرا i و h مساوی اند. بنابراین داریم: W(h,m)=h+m-1

عمل اصلی، مقایسه ای است که در ادغام صورت می پذیرد. تعداد کل مقایسه ها عبارت است از تعداد مقایسه هایی که در فراخوانی بازگشتی mergsort با U به عنوان ورودی انجام می شود، به علاوه تعداد مقایسه هایی که در فراخوانی بازگشتی mergsort با V به عنوان ورودی انجام می شود، به علاوه تعداد مقایسه ها در فراخوانی سطح بالای merge . بنابراین:

$$W(n) = W(h) + W(m) + h + m - 1$$

W(h) : زمان لازم برای مرتب سازی U W(m) : زمان لازم برای مرتب سازی V h+m-1 : زمان لازم برای ادغام

اگر n توانی از 2 باشد، آنگاه h و m برابر $\frac{n}{2}$ می باشد و داریم:

$$W(n) = W(\frac{n}{2}) + W(\frac{n}{2}) + n - 1 = 2W(\frac{n}{2}) + n - 1$$

اگر اندازه ورودی 1 باشد، شرط پایانی برآورده می شود و اد غامی انجام نمی گیرد. بنابراین W(1)=0 است.

دستور بازگشتی :

$$W(n) = 2w(\frac{n}{2}) + n - 1$$
$$W(1) = 0$$

با حل این دستور بازگشتی داریم:

$$W(n) = n\lg n - (n-1) \quad \in \theta(n\lg n)$$

تذکر: اگر n توانی از 2 نباشد، داریم:

$$W(n) = W(\left\lfloor \frac{n}{2} \right\rfloor) + W(\left\lceil \frac{n}{2} \right\rceil) + n - 1 \quad \in \theta(n\lg n)$$

نکته: در مرتب سازی درجا (in_place sort) ، از فضایی بیشتر از آن چه که مورد نیاز نگهداری ورودی است، استفاده نمی شود.

تذکر: الگوریتم مرتب سازی ادغامی، مرتب سازی درجا نیست. زیرا از آرایه های U و V به علاوه آرایه ورودی S استفاده می کند. تعداد کل عناصر آرایه های اضافی تولید شده، حدوداً n(1+1/2+1/4+…)=2n می شود.

مرتب سازی سریع (Quick sort)

در مرتب سازی سریع، آرایه به دو بخش تقسیم شده و سپس هر یک از بخش ها به صورت بازگشتی مرتب می شوند. در این روش یک عنصر به عنوان عنصر محور انتخاب می شود و عناصر کوچکتر از محور در یک آرایه و عناصر بزرگتر از محور در آرایه دیگری قرار می گیرند. می گوئیم آرایه افراز می شود. عنصر محوری می تواند هر عنصری باشد که برای سهولت اولین عنصر را در نظر می گیریم.

پس از افراز ترتیب عناصر در زیر آرایه ها مشخص نیست و به نحوه پیاده سازی افراز بستگی دارد. ترتیب آنها از چگونگی افراز آرایه ها ناشی می شود. همه عناصر کوچکتر از عنصر محوری در طرف چپ آن و همه عناصر بزرگتر در طرف راست آن واقع هستند. سپس مرتب سازی سریع به طور بازگشتی فراخوانی می شود تا هر یک از دو آرایه را مرتب کند. آنها نیز افراز می شوند و این روال آنقدر ادامه می یابد تا به آرایه ای با یک عنصر برسیم که ذاتاً مرتب است.

```
void  quicksort (index low, index high){
    index pivotpoint;
    if (high > low){
      partition (low, high, pivotpoint);
      quicksort (low, pivotpoint-1);
      quicksort (pivotpoint+1, high);
    }
}
```

الگوریتم افراز آرایه

```
void partition (index low, index high , index& pivotpoint){
    index i, j; keytype pivotitem;
    pivotitem = S[low];
    j = low;
    for (i = low+1 ; i <= high ; i++)
        if (S[i] < pivotitem)
            { j++;
             exchange S[i] and S[j]; }
    pivotpoint = j;
    exchange S[low] and S[pivotpoint] ;
}
```

در این روال، هر گاه معلوم شود عنصری کوچکتر از عنصر محوری است به طرف چپ آرایه حرکت داده می شود.

تحلیل پیچپیدگی زمانی در حالت معمول برای الگوریتم افراز

عمل اصلی، مقایسه s[i] با pivotitem است و اندازه ورودی، تعداد عناصر موجود در زیر آرایه یعنی، $n=high - low +1$ می باشد. چون هر یک از عناصر به جز اولی مقایسه شده اند داریم: $T(n) = n - 1$

تحلیل پیچیدگی در بدترین حالت برای الگوریتم مرتب سازی سریع

عمل اصلی، مقایسه S[i] با pivotitem در روال partition می باشد و اندازه ورودی تعداد عناصر موجود در آرایه S یعنی n است. بدترین حالت هنگامی رخ می دهد که آرایه از قبل به ترتیب غیر نزولی مرتب شده باشد. چون در این حالت هیچ عنصری کوچکتر از نخستین عنصر آرایه که همان عنصر محوری است نخواهد بود. بنابراین هنگامی که partition در بالاترین سطح فراخوانی شود هیچ عنصری در طرف چپ عنصر محوری قرار نمی گیرد و مقدار pivotpoint که توسط partition تعیین می شود یک است. به طور مشابه در هر بار فراخوانی بازگشتی pivotpoint مقدارlow را دریافت می کند. بنابراین آرایه به طور مکرر به یک زیر آرایه خالی در طرف چپ و زیر آرایه با یک عضو کمتر در طرف راست افراز می شود. بنابراین داریم:

$T(n)=T(0) + T(n-1) + n–1$

$T(0)$: زمان لازم برای مرتب سازی زیرآرایه طرف چپ

n-1 : زمان لازم برای افراز

تذکر: علت اینکه از نماد $T(n)$ استفاده می کنیم این است که می خواهیم پیچیدگی در حالت معمول را برای طبقه ای از نمونه ها تعیین کنیم که از قبل به ترتیب غیر نزولی مرتب شده اند.

دستور بازگشتی:

$T(n)= T(n-1) + n - 1 \qquad n>0$

$T(0)=0$

حل این دستور بازگشتی به صورت $T(n) = \frac{n(n-1)}{2}$ است.

پیچیدگی زمانی در بدترین حالت با رابطه زیر مشخص می شود:

$$W(n) = \frac{n(n-1)}{2} \in \theta(n^2)$$

بدترین حالت زمانی رخ می دهد که آرایه از قبل مرتب شده باشد زیرا همواره نخستین عنصر را به عنوان عنصر محوری انتخاب می کنیم بنابراین اگر به نوعی بدانیم که آرایه تقریباً مرتب است انتخاب آن به عنوان عنصر محوری خوب نیست. اگر از این روش استفاده کنیم بدترین حالت در هنگامی که آرایه مرتب باشد رخ نمی دهد، ولی پیچیدگی زمانی در بدترین حالت هنوز هم $n(n-1)/2$ خواهد بود.

سئوال:

در بدترین حالت الگوریتم مرتب سازی سریع از مرتب سازی تعویضی سریعتر نیست، پس چرا آن را مرتب سازی سریع نامیده اند؟

جواب: به علت رفتار آن در حالت میانگین.

تحلیل پیچیدگی زمانی در حالت میانگین برای الگوریتم مرتب سازی سریع

فرض می کنیم احتمال آنکه pivotpoint بازگردانده شده توسط partition ، هر یک از اعداد 1 تا n باشد یکسان است. هنگامی که هر یک از ترتیب های ممکن به تعداد دفعاتی مساوی مرتب شوند در این مورد پیچیدگی زمانی در حالت میانگین با دستور بازگشتی زیر مشخص می شود:

$$A(n) = \sum_{p=1}^{n} \frac{1}{n}[A(p-1) + A(n-p)] + n - 1$$

$\frac{1}{n}$: احتمال آن که محل محور (pivotpoint) برابر p باشد.

عبارت داخل [...] : زمان میانگین برای مرتب سازی آرایه ها هنگامی که محل محور برابر p است.

n-1 : زمان لازم برای افراز

با انجام عملیاتی روی رابطه بالا و قرار دادن $a_n = \frac{A(n)}{n+1}$، دستور بازگشتی زیر را داریم:

$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} \qquad n > 0$

$a_0 = 0$

که با حل آن داریم: $a_n \approx 2\ln n$ و بنابراین $A(n) \in \theta(n \lg n)$.

الگوریتم ضرب ماتریس های استراسن

الگوریتم ضرب دو ماتریس $n \times n$ طبق تعریف ضرب ماتریس ها در زیر آورده شده است. این الگوریتم آرایه‌های دوبعدی $A[1..n][1..n]$ و $B[1..n][1..n]$ را ضرب کرده و حاصل را در آرایه $C[1..n][1..n]$ قرار می دهد.

```
void  matrixmult( int  n , const  number A[ ] [ ] , B [ ] [ ] , number C [ ] [ ] ){
        index   i, j, k;
        for (i = 1; i <= n; i++)
          for (j = 1; j <= n; j++) {
              C[i][j] = 0;
              for (k=1 ; k <= n; k++)
                  C[i][j] = C[i][j] + A[i][k] * B[k][j];
            }
}
```

تحلیل پیچیدگی زمانی در حالت معمول برای الگوریتم بالا

عمل اصلی، دستور ضرب در داخلی‌ترین حلقه for است و اندازه ورودی تعداد سطرها و ستون‌ها یعنی n می باشد. همواره n بار گذر از حلقه for i وجود دارد که در هر گذر همواره n بار گذر از حلقه for j وجود دارد و در هر گذر از حلقه for j همواره n بار گذر از حلقه for k وجود دارد. چون عمل اصلی در داخل حلقه for k قرار دارد، خواهیم داشت: $T(n) = n \times n \times n = n^3$

روش استراسن برای ضرب دو ماتریس

در روش معمولی ضرب دو ماتریس که در بالا آورده شد، پیچیدگی زمانی ضرب و همچنین جمع در $\theta(n^3)$ هستند. استراسن الگوریتمی را ارائه داد که پیچیدگی آن، چه از لحاظ ضرب و چه از لحاظ جمع و تفریق بهتر از پیچیدگی درجه سوم است. در روش استراسن تعداد ضرب ها برابر $n^{2.81}$ و تعداد جمع و تفریق ها برابر $6n^{2.81} - 6n^2$ می باشد.

مثال: فرض کنید C حاصل ضرب دو ماتریس $A_{2\times2}$ و $B_{2\times2}$ باشد. یعنی :

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$m_1 = (a_{11} + a_{22})(b_{11} + b_{22})$$
$$m_2 = (a_{21} + a_{22})b_{11}$$
$$m_3 = a_{11}(b_{12} - b_{22})$$
$$m_4 = a_{22}(b_{21} - b_{11})$$
$$m_5 = (a_{11} + a_{12})b_{22}$$
$$m_6 = (a_{21} - a_{11})(b_{11} + b_{12})$$
$$m_7 = (a_{12} - a_{22})(b_{21} + b_{22})$$

حاصل ضرب C به صورت زیر می باشد:

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

تذکر: استراسن به طور نمونه برای محاسبه c_{12} که برابر $a_{11}b_{12} + a_{12}b_{22}$ می باشد، یک جمله $a_{11}b_{22}$ اضافه و کم کرد:

$$a_{11}b_{12} + a_{12}b_{22} + a_{11}b_{22} - a_{11}b_{22}$$

که می توان آن را به صورت $a_{11}(b_{12} - b_{22}) + (a_{11} + a_{12})b_{22}$ نوشت و جمله اول را m_3 و جمله دوم را m_5 نامید.

برای ضرب دو ماتریس2×2 ، روش عادی به 8 عمل ضرب و 4 عمل جمع نیاز دارد و روش استراسن به 7 عمـل ضـرب و 18 عمل جمع و تفریق نیاز دارد. پس یک عمل ضرب صرفه جویی می شود که چندان جالب نیست. در واقع روش استراسـن در مـورد ضرب ماتریس های 2×2 ارزش چندانی ندارد.

مثال: دو ماتریس زیر را به روش استراسن در هم ضرب کنید.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix} \qquad B = \begin{bmatrix} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{bmatrix}$$

حل: شکل زیر نحوه افراز را نشان می دهد:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix} \times \begin{bmatrix} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{bmatrix}$$

محاسبات به طریق زیر می باشد:

$$M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$
$$\left(\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \right) \times \left(\begin{bmatrix} 8 & 9 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix} \right) = \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix}$$

هنگامی که ماتریس ها به قدر کافی کوچک باشند، آن ها را به شیوه استاندارد در هم ضرب می کنیم.(در این مثال n=2)

$$M_1 = \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix} = \begin{bmatrix} 3 \times 17 + 5 \times 7 & 3 \times 10 + 5 \times 9 \\ 11 \times 17 + 13 \times 7 & 11 \times 10 + 13 \times 9 \end{bmatrix} = \begin{bmatrix} 86 & 75 \\ 278 & 227 \end{bmatrix}$$

دست می آید. ■

الگوریتم استراسن

```
void strassen (int n , n × n_matrix A ,B , n × n_matrix& C){
    if (n <= threshold)
        compute C = A × B using the standard algorithm;
    else {
        partition A into four submatrices A11 , A12 , A21 , A22 ;
        partition B into four submatrices B11 , B12 , B21 , B22 ;
        compute C = A × B using Strassen's Method ;
       }
}
```

تذکر: در الگوریتم بالا، مقدارthreshold نقطه ای است که در آن احساس می شود استفاده از الگوریتم ضرب استاندارد بهتر از فراخوانی بازگشتی روال Strassem باشد.

تحلیل پیچیدگی زمانی تعداد ضرب ها در الگوریتم استراسن در حالت معمول

عمل اصلی، یک ضرب ساده و اندازه ورودی تعداد سطرها و ستون ها در ماتریس ها یعنی n است. عمل ضرب تا رسیدن به دو ماتریس1×1 ادامه می یابد و در آن جا دو عدد موجود در ماتریس ها در هم ضرب می شوند. مقدار آستانه واقعی تاثیری بر مرتبه الگوریتم ندارد. هنگامی که دو ماتریس n×n با $n > 1$ داشته باشیم، الگوریتم دقیقا هفت بار فراخوانی می شود و هر بار یک ماتریس (n/2)×(n/2) ارسال می شود و هیچ ضربی در بالاترین سطح انجام نمی گیرد.

دستور بازگشتی :

$T(n) = 7\ T(\frac{n}{2})$

$T(1)=1$

با فرض n توانی از 2 باشد. حل این دستور بازگشتی برابر است با:

$T(n) = n^{\lg 7} \approx n^{2.81} \in \theta(n^{2.81})$

تحلیل پیچیدگی زمانی تعداد جمع ها و تفریق های الگوریتم استراسن در حالت معمول

عمل اصلی، یک جمع یا تفریق ساده است. هنگامی که n=1 باشد، هیچ جمع و تفریقی صورت نمی پذیرد. هنگامی که دو ماتریس n×n با $n > 1$ داریم، الگوریتم دقیقا هفت بار فراخوانی می شود. هر بار یک ماتریس (n/2)×(n/2) ارسال می شود و 18 عمل جمع و تفریق ماتریسی روی ماتریس های(n/2)×(n/2) انجام می شود.

$$T(n) = 7T(\frac{n}{2}) + 18(\frac{n}{2})^2$$

$$T(1) = 0$$

حل این دستور بازگشتی:

$$T(n) = 6n^{\lg 7} - 6n^2 \approx 6n^{2.81} - 6n^2 \in \theta(n^{2.81})$$

تذکر: اگر n توانی از 2 نباشد، تعداد کافی از سطرها و ستون های صفر به ماتریس های اولیه اضافه کرده تا ابعاد توانی از 2 شوند و یا در فراخوانی بازگشتی، هرگاه تعداد سطرها و ستون ها فرد شد، می توان فقط یک سطر و ستون اضافی صفر، اضافه کرد.

الگوریتم استراسن از لحاظ تعداد ضرب های انجام شده همواره کارایی بیشتری دارد و از لحاظ تعداد جمع و تفریق ها به ازای مقادیر بزرگ n ، کارایی آن بیشتر است.

عملیات محاسباتی روی اعداد صحیح بزرگ

اگر بخواهیم اعمال محاسباتی روی اعداد صحیحی انجام دهیم که بزرگ تر از حدی هستند که سخت افزار کامپیوتر قادر به نمایش آن ها است، از روش تقسیم و حل استفاده می کنیم. یک راه برای نمایش دادن یک عدد صحیح، استفاده از آرایه ای از اعداد صحیح است، که در هر خانه آرایه یک رقم را نگهداری می کند. برای مثال نحوه ذخیره عدد صحیح 8452396 به صورت زیر است:

1	2	3	4	5	6	7
6	9	3	2	5	4	8

برای نگهداری اعداد مثبت و منفی، بالاترین محل آرایه را برای علامت عدد رزرو کنیم. برای نشان دادن عدد مثبت از صفر و برای نشان دادن عدد منفی از یک استفاده می کنیم.

ضرب اعداد صحیح بزرگ

دو عدد صحیح بزرگ را می توان با یک الگوریتم استاندارد از درجه 2 ، در یکدیگر ضرب کرد. این الگوریتم مبتنی بر استفاده از روش تقسیم و حل برای تقسیم یک عدد صحیح n رقمی به دو عدد صحیح با حدود n/2 رقم است. اگر n تعداد ارقام عدد صحیح u باشد، آن را به دو عدد صحیح یکی x با $\lceil n/2 \rceil$ رقم و دیگری y با $\lfloor n/2 \rfloor$ رقم تبدیل می کنیم. به صورت: $u = x \times 10^m + y$ که $m = \lfloor n/2 \rfloor$.

مثال: می توان عدد 12345 را به دو عدد 123 و 45 تقسیم کرد: $12345 = 123 \times 10^2 + 45$

اگر دو عدد صحیح n رقمی داشته باشیم:

$$u = x \times 10^m + y \quad , \quad v = w \times 10^m + z$$

حاصل ضرب آن ها طبق رابطه زیر به دست می آید :

$$uv = (x \times 10^m + y)(w \times 10^m + z) = xw \times 10^{2m} + (xz + wy) \times 10^m + yz$$

می توانیم v و u را با انجام 4 عمل ضرب روی اعداد صحیح و با حدود نیمی از ارقام و اجرای عملیات زمان خطی، در هم ضرب کنیم .

$$567{,}832 \times 9{,}423{,}723 = (567 \times 10^3 + 832)(9423 \times 10^3 + 723)$$

$$= 567 \times 9{,}423 \times 10^6 + (567 \times 723 + 9423 \times 832) \times 10^3 + 832 \times 723$$

سپس این اعداد صحیح کوچک تر را می توان به طور بازگشتی، با تقسیم آن ها به اعداد صحیح کوچک تر، در هم ضرب کرد. این فرایند تقسیم آنقدر ادامه می یابد تا به یک مقدار آستانه برسیم که در آن زمان، عمل ضرب به طریق استاندارد (با استفاده از سخت افزار کامپیوتر) انجام می شود.

تذکر: اگر تعداد ارقام صحیح یکی نباشد، کافی است که از $m=\lfloor n/2 \rfloor$ برای تبدیل هر دو آن ها استفاده کنیم، که در آن n تعداد ارقام موجود در عدد بزرگ تر است.

الگوریتم ضرب اعداد صحیح بزرگ

```
large-integer prod(large-integer u , large-integer v){
  large-integer x,y,z,w; int n,m;
  n=maximum(number of digits in u , number of gigits in v)
  if(u==0 || v==0) return 0;
  else if (n<= threshold)    return u×v obtained in the usual way;
  else{
       m=⌊n/2⌋;
       x = u divide 10^m ; y = u rem 10^m ; w = v divide 10^m ; z = v rem 10^m ;
       return prod(x,w) ×10^2m + (prod(x,z)+ prod(w,y))×10^m + prod(y,z);
   }
}
```

تذکر: توابع divide , rem و × نماینگر توابع خطی زمانی هستند که باید آن ها را بنویسیم.

تحلیل پیچیدگی زمانی در بدترین حالت برای الگوریتم ضرب اعداد صحیح بزرگ

عمل اصلی در الگوریتم بالا، دستکاری یک رقم دهدهی در یک عدد صحیح بزرگ در هنگام جمع کردن، تفریق کردن، یا انجام اعمال $\times 10^m$ ، $\text{rem } 10^m$ و $\text{divide } 10^m$ می باشد. هریک از سه فراخوانی اخیر، عمل اصلی را m بار انجام می دهند. اندازه ورودی، تعداد ارقام هر یک از دو عدد صحیح می باشد. بدترین حالت، زمانی است که هر دو عدد صحیح هیچ رقمی مساوی صفر نداشته باشند، زیرا بازگشتی فقط هنگامی به پایان می رسد که به حد آستانه ای برسیم. اگر n توانی از 2 باشد، در این صورت ، w,y,x و z همگی دقیقا n/2 رقم خواهند داشت. یعنی اندازه ورودی هر یک از چهار فراخوانی بازگشتی به تابع prod، برابر n/2 است. چون m=n/2 است، عملیات زمانی خطی جمع،تفریق، $\times 10^m$، $\text{rem } 10^m$ و $\text{divide } 10^m$ همگی دارای پیچیدگی زمانی خطی نسبت به n می باشند. همه عملیات زمانی خطی را در یک جمله cn گروه بندی کنیم که در آن c یک ثابت مثبت است :

$$W(n) = 4W(\frac{n}{2}) + cn \qquad n > s \qquad \Rightarrow \theta(n^{\lg_2^4}) = \theta(n^2)$$
$$W(s) = 0$$

مقدار واقعی s که در آن، نمونه دیگر تقسیم نمی شود، کوچک تر یا مساوی مقدار آستانه و توانی از 2 است، زیرا همه ورودی ها در این حالت توانی از 2 هستند.

تذکر: می توان روش ضرب دو عدد صحیح بزرگ را بهبود بخشید که در این حالت داریم: $W(n) \in \theta(n^{\lg_2^3}) \approx \theta(n^{1.58})$. این بهبود در پیوست کتاب آورده شده است.

کجا نمی توان از روش تقسیم و حل استفاده کرد؟

در صورت امکان باید در موارد زیر از روش تقسیم و حل پرهیز کرد:

1- نمونه ای با اندازه n به دو یا چند نمونه تقسیم می شود که اندازه آن ها نیز تقریبا n است.

2- نمونه ای با اندازه n تقریبا به n نمونه با اندازه n/c تقسیم می شود که c یک مقدار ثابت است.

افراز در حالت اول به یک الگوریتم زمانی نمایی و در حالت دوم به یک الگوریتم $\theta(n^{\lg n})$ منجر می شود. الگوریتم جمله n ام فیبوناچی(بازگشتی) یک الگوریتم تقسیم و حل است که نمونه را تقسیم کرده جمله nام را از روی دو نمونه (n-1) ام و (n-2) ام محاسبه می کند. تعداد جملات محاسبه شده توسط این الگوریتم نسبت به n نمایی است، حال آنکه تعداد جملات محاسبه شده توسط الگوریتم جمله n ام فیبوناچی(تکراری) نسبت به n خطی است.گاهی مسئله ای نیاز به حالت نمایی دارد و دلیلی ندارد که از روش تقسیم و حل پرهیز کرد، مانند مسئله برج های هانوی که ترتیب حرکات، که از الگوریتم استاندارد تقسیم و حل به دست می آید، نسبت به n نمایی بوده و با توجه به محدودیت های مسئله در نحوه حرکت دادن دیسک ها دارای بیشترین کارایی است.

مسئله برجهای هانوی

در این مسئله سه میله وجود دارد، که بر روی میله اول n مهره قرار دارد. هدف انتقال n مهره به میله سوم است که برای این کار از میله دوم کمک گرفته می شود. اندازه مهره ها در میله اول از پایین به بالا کاهش می یابد. در انتقال مهره ها هرگز مهره بزرگتر بر روی مهره کوچکتر نباید قرار بگیرد و در هر بار فقط امکان انتقال یک مهره وجود دارد که از بالای میله انتخاب می شود. شرط توقف این مسئله را وقتی در نظرمی گیریم که فقط یک مهره وجود دارد که در این حالت آن مهره را از میله اول به میله سوم منتقل می کنیم.

در صورت وجود بیش از یک مهره ، مسئله را به صورت زیر حل می کنیم:

1) انتقال n-1 مهره از میله مبدا به میله کمکی.

2) انتقال مهره nام از میله مبدا به میله مقصد.

3) انتقال n-1 مهره از میله کمکی به میله مقصد.

الگوریتم انتقال n مهره از BEG به END به کمک END

```
tower (n , BEG, AUX, END){
   if  (n==1)     BEG → END;
  else{
```

```
        BEG → END;
        tower (n–1, AUX, BEG, END).
    }
}
```

رابطه بازگشتی الگوریتم بالا :

$$T(n) = T(n-1) + 1 + T(n-1)$$
$$T(1) = 1$$

بنابراین رابطه بازگشتی الگوریتم برجهای هانوی، برابر $T(n) = 2T(n-1) + 1$ می باشد. جواب این معادله برابر است با:

$$T(n) = 2^n - 1$$

بنابراین مرتبه اجرایی این الگوریتم برابر $o(2^n)$ می باشد.

مثال: انتقال 3 مهره از میله A به میله C به کمک میله B .

(برای سادگی به جایtower از T استفاده می کنیم.)

$$T(3,A,B,C) = \begin{cases} T(2,A,C,B) = \begin{cases} T(1,A,B,C) = A \to C & (1) \\ A \to B & (2) \\ T(1,C,A,B) = C \to B & (3) \end{cases} \\ A \to C \quad (4) \\ T(2,B,A,C) = \begin{cases} T(1,B,C,A) = B \to A & (5) \\ B \to C & (6) \\ T(1,A,B,C) = A \to C & (7) \end{cases} \end{cases}$$

هفت حرکت لازم برای انتقال 3 مهره از میله A به B به کمک شماره در شکل بالا مشخص شده است.

مجموعه تست

۱- کم رشد ترین حد بالای زمان اجرای الگوریتم استراسن(strassen) برای ضرب دو ماتریس مربعی n در n کدام است؟

1) 8 2) 7 3) 6 4) 5

۲- چنانچه در الگوریتم quicksort ، الگوریتم بخش بندی(partition) زمان ثابت C نیاز داشته باشد، زمان اجرای مرتب سازی سریع در حالت تصادفی(داده های تصادفی) چیست؟

1) $\theta(n^2)$ 2) $\theta(n)$ 3) $\theta(\lg n)$ 4) $\theta(n \lg n)$

۳- فرمول $T(n) = 2T(\frac{n}{2}) + n - 1$ **، دقیقاً زمان Merge Sort را در کدام حالت مشخص می‌کند؟**

1) بدترین حالت 2) بهترین حالت 3) در همه حالت 4) بهترین و میانگین

۴- مرتبه زمانی مرتب سازی یک آرایه با n عنصر با روش Merge sort ، در بدترین حالت کدام است؟

1) n 2) n^2 3) $n \lg n$ 4) $n^2 \lg n$

۵- در ضرب ماتریس ها به روش استراسن(Strassen) اگر مساله کوچک ضرب ماتریس های 2×2 **باشد، برای ضرب دو ماتریس** 8×8 **چند ضرب عددی صورت می پذیرد؟**

1) 57 2) 343 3) 392 4) 512

۶- اگر روشی برای ضرب ماتریس های با ۶ عمل ضرب یا تقسیم و ۲۸ عمل جمع یا تفریق داشته باشیم و این روش را با استفاده از تقسیم و غلبه برای ضرب ماتریس های به کار ببریم(روش استراسن)، مرتبه ضرب ماتریسی حاصل برابر است با.....

1) $\theta(n^2)$ 2) $\theta(n^2 \lg n)$ 3) $\theta(n^2 \lg_6^{28})$ 4) $\theta(n^{\lg_2^6})$

۷- دستور حذف شده برنامه جستجوی دودویی زیر کدام است؟

```
Procedure Binsearch(a:elementarray;x:element;  var  left,right,j : integer)
  var  mid  : integer;
begin
  if (left<=rigth)  then
    begin
       mid:=(left+right) div ۲;
      case   compare(x,a[mid])   of
          '>' : Binsearch(a,x,mid+۱,right,j);
          '<' : دستور حذف شده
          '=' : j:=mid;
      end;
```

```
end;
```

1) Binsearch(a,x,mid-1,right,j); 2) Binsearch(a,x,mid-1,left,j);

3) Binsearch(a,x,left,mid-1,j); 4) Binsearch(a,x,left,mid,right);

۸- اگر الگوریتم جستجوی دودویی را برای جستجوی عناصر آرایه A[۱..۸]=[۵،۱۰،۱۵،۲۰،۲۵،۳۰،۳۵،۴۰] به کار ببریم، میانگین تعداد مقایسه ها برای جستجوی موفق تقریبا کدام است؟

1) 2,2 2) 2,4 3) 2,6 4) 2,8

۹- رویه partition در الگوریتم QuickSort به صورت زیر است. اگر تمام درایه های A[p..r] دارای مقدار یکسانی باشند، مقداری که رویه فوق بر می گرداند چقدر است؟

```
function  partition(p,r : integer): integer;
  var   x,i,j: integer;
begin
   x:=A[p];  i:=p-۱;  j:=r+۱;
   while  TRUE  do
      begin
        repeat  j:=j-۱   until  A[j]<=x;
        repeat  i:=i+۱ until  A[i]>=x;
        if  i<j  then  swap(A[i],A[j]) else  return (j);
      end
end;
```

1) p 2) r 3) $\left\lceil \frac{p+r}{2} \right\rceil$ 4) $\left\lfloor \frac{p+r}{2} \right\rfloor$

۱۰- اگر در الگوریتم مرتب سازی سریع به ترتیب صعودی، عنصر لولا (pivot) را همان عنصر اول لیست بگیریم و با استفاده از آن یک بار لیست مرتب نزولی و یکبار دیگر مرتب صعودی را مرتب کنیم، گزینه صحیح برای مرتبه تعداد عملیات اصلی(مقایسه و جا به جایی) را در این دو حالت انتخاب کنید.

1) هر دو حالت از $O(n \lg n)$

2) هر دو حالت از $O(n^2)$

3) برای لیست صعودی $O(n)$ و برای لیست نزولی $O(n^2)$

4) برای لیست صعودی $O(n \lg n)$ و برای لیست نزولی $O(n^2)$

۱۱- آرایه n عنصری A را در نظر بگیرید، فرض کنید $n = 2^k$ باشد. الگوریتم Mergesort بر روی آرایه A در بدترین حالت چند مقایسه میان عناصر آرایه انجام می دهد؟

1) $n \lg n - n + 1$ 2) $n \lg n + n - 1$ 3) $n \lg n - 2n - 1$ 4) $n \lg n - n - 1$

مرتب شده واحد، شامل همه رکوردها به دست آوریم. در هر ادغام رکوردهای فایل های ورودی ممکن است چند بار از یک فایل خوانده و در یک فایل دیگر نوشته شوند. به هر کدام از این نوشتن و خواندن یک جابه جـایی مـی گـوییم. حداقل تعداد کل این جا به جایی ها برای ادغام همه فایل ها چقدر است؟

1) 195 2) 200 3) 185 4) 215

۱۳- تابع زیر را در نظر می گیریم. فرض کنید که **T(n)** تعداد سطرهایی است که در اثر اجرای این تـابع روی صـفحه، نمایش داده خواهند شد. و داریم $n \leq 1$ و **T(n)=۱**. کدام یک از تعاریف بازگشتی زیر صحیح است؟

```
Function concow(n)
  if n<=۱ then
     writeln('STOP')
     return ۱;
  else
    for i=۱ to n do
       writeln(concow(n div ۲))
    return n
```

1) $T(n) = \sum_{i=1}^{n} T(\left\lfloor \frac{i}{2} \right\rfloor)$ 2) $T(n) = nT(\left\lfloor \frac{n}{2} \right\rfloor) + n$

3) $T(n) = nT(\left\lfloor \frac{n}{2} \right\rfloor)$ 4) $T(n) = T(\left\lfloor \frac{n}{2} \right\rfloor) + n$

۱۴- کدام یک از گزینه های زیر در مورد الگوریتم **Quick Sort** درست است؟

(فرضیات: عنصر اول محور است، آرایه از قبل مرتب است)

1) زمان اجرای الگوریتم $O(n)$ است.
2) زمان اجرای الگوریتم $O(n^2)$ است.
3) زمان اجرای الگوریتم $O(n \lg n)$ است.
4) متوسط زمان اجرای الگوریتم $O(n \lg n)$ است.

۱۵- با فرض وجود تعداد **n** دیسک در برج هانوی مشخص نمائید تعداد جابجائی های لازم برای انتقـال دیسـک هـای فوق از میله **i**ام به میله **j**ام معادل کدام رابطه بازگشتی است؟

1) T(n)=2T(n-1)+1 2) T(n)=3T(n-1)+1
3) T(n)=2T(n-2)+n-2 4) T(n)=3T(n-2)+2

۱۶- کدام رابطه بازگشتی **T(n)** را برای الگوریتم بازگشتی جستجوی دودویی نشان می دهد؟

1) $T(n) = 2T(n-2) + 1$ 2) $T(n) = T(\frac{n}{2}) + 1$

3) $T(n) = \frac{1}{2}T(n-2) + 1$ 4) $T(n) = 2T(n-1) + 1$

اگر **w(n)** پیچیدگی زمانی این الگوریتم باشد، کدام یک از روابط زیر صحیح می باشد؟ (**n>۱** و **n** توانی از **۲** است)

1) $w(n) = 2w(n-1) + \frac{n-1}{2}$
2) $w(n) = 2w(n-1) + \frac{n}{2}$
3) $w(n) = 2w(\frac{n}{2}) + n - 1$
4) $w(n) = w(n-1) + w(n-2) + n - 1$

۱۸- الگوریتم مرتب سازی ادغامی یک آرایه n عنصری را با چه زمانی مرتب می نماید؟

1) $O(n^2)$
2) $O(2^n)$
3) $O(n \lg n)$
4) $O(n^3)$

۱۹- مرتبه زمانی الگوریتم بازگشتی زیر چیست؟(فراخوانی به صورت akgor(۱,n) می باشد.)

```
procedure algor(i,j)
  if i=j then return
  k := ⌊(i + j) / 2⌋
  call algor(i,k);
  call algor(k+۱,j)
end algor
```

1) $\theta(\lg n)$
2) $\theta(n)$
3) $\theta(n \lg n)$
4) $\theta(n^2)$

۲۰- در الگوریتم Mergesort اگر به جای آنکه هر بار لیست به دو قسمت مساوی تقسیم شود به چهار قسمت مساول تقسیم گردد و در مرحله ترکیب این چهار لیست در یکدیگر ادغام شوند، پیچیدگی زمانی الگوریتم چه خواهد شد؟

1) $\theta(n^{3/4})$
2) $\theta(n \lg n)$
3) $\theta(n^2)$
4) $\theta(n^2 \lg_4^n)$

۲۱- اگر دو ماتریس 4×4 با روش ضرب استراسن در یکدیگر ضرب شوند، برای ضرب این دو ماتریس چند ضرب عددی صورت می گیرد؟

1) 28
2) 49
3) 7
4) 11

```
Procedure   MergeSort(low,high,A)
begin
  if   low < high  then
    begin
       mid=⌊(low + high) / 2⌋;
       mergesort(Low,mid,A);
       mergesort(Low,mid,A);
       mergesort(Low,mid,A);
   end
end
```

رویه Merge در رویه فوق دو لیست مرتب شده A[low..mid] و A[mid+۱..high] را ادغام می کند. تعداد صداهای انجام گرفته به رویه فوق برای مرتب کردن لیست زیر چیست؟

310 , 285 , 179 , 625 , 351 , 423 , 861 , 254 , 450 , 520

1) 23　　2) 28　　3) 19　　4) 20

۲۳- الگوریتم Quick Sort

1) همواره $O(n^2)$ است.　　2) فقط در بدترین حالت یعنی زمانی که داده ه ا مرتب باشند $O(n^2)$ است.

3) همواره $O(n \lg n)$ است.　　4) سریعترین روش مرتب کردن است.

۲۴- الگوریتم Merge Sort از نظر زمان محاسبه دارای

1) $O(n \lg n)$ است.　　2) $O(n)$ است.　　3) $O(n^2)$ است.　　4) هیچکدام

۲۵- الگوریتم QuickSort یک رشته n تایی را در حالت متوسط با چه سرعتی مرتب می کند؟

1) $O(n \lg n)$　　2) $O(n)$　　3) $O(n^2)$　　4) $O(\lg n)$

پاسخ تشریحی

2-1) در روش استراسن برای ضرب دو ماتریس $n \times n$ نیاز به 7 ضرب ماتریس های $\frac{n}{2} \times \frac{n}{2}$ می باشد.

2-2)

3-3) فرمول $T(n) = 2T(\frac{n}{2}) + n - 1$ ، دقیقاً زمان Merge Sort را در همه حالات (بدترین، بهترین و میانگین) مشخص می‌کند.

3-4) مرتبه زمانی مرتب سازی یک آرایه با n عنصر با روش Merge sort ، در بدترین حالت $q(n \log n)$ است.

3-5) مشابه مسئله استراسن است که در متن کتاب توضیح داده شد ولی با این تفاوت که در نهایت ماتریس ها 2×2 است (نه 1×1). بنابراین وقتی ماتریس ها 2×2 باشند از روش ضرب معمولی استفاده می کنیم که نیاز به 8 عمل ضرب دارد. رابطه بازگشتی به صورت زیر می باشد:

$$T(n) = 7T(\frac{n}{2})$$
$$T(2) = 8$$

با توجه به این رابطه داریم:

$$T(4) = 7T(2) = 7 \times 8 = 56$$
$$T(8) = 7T(4) = 7 \times 56 = 392$$

4-6) اگر عمل اصلی را تعداد ضرب یا تقسیم در نظر بگیریم، داریم:

$$T(n) = 6T(\frac{n}{2}) \Rightarrow T(n) = n^{\log_2^6}$$

3-7) در جستجوی دودویی، در صورتی که عنصر مورد جستجو یعنی x از عنصر وسط آرایه یعنی a[mid] کوچکتر باشد، جستجو در نیمه پایینی آرایه ادامه می یابد. اندیس بالای آرایه نیمه پایین برابر mid-1 است.

3-8) تعداد مقایسه ها برای پیدا کردن هر یک از عناصر در زیر آن نوشته شده است:

1	2	3	4	5	6	7	8
5	10	15	20	25	30	35	40
3	2	3	1	3	2	3	4

بنابراین میانگین برابر است با:

$$\frac{3+2+3+1+3+2+3+4}{8} = \frac{21}{8} \approx 2.6$$

4-9) مقداری که تابع بر می گرداند برابر $\left\lfloor \frac{p+r}{2} \right\rfloor$ می باشد. به طور نمونه مقداری که برای آرایه زیر برگردانده می شود برابر $\left\lfloor \frac{3+8}{2} \right\rfloor = 5$ می باشد: (x=8,p=3,r=8)

8	8	8	8	8	8

2-10) مرتبه مرتب سازی سریع برای آرایه مرتب برابر $O(n^2)$ است و صعودی یا نزولی بودن فرقی نمی کند.

1-11) در مرتب سازی ادغامی اگر n توانی از 2 باشد، آنگاه آرایه به دو قسمت برابر $\frac{n}{2}$ می باشد و داریم:

$$W(n) = W(\frac{n}{2}) + W(\frac{n}{2}) + n - 1 = 2W(\frac{n}{2}) + n - 1$$

با حل این دستور بازگشتی داریم:

$$W(n) = n\lg n - (n-1) \quad \in \theta(n\lg n)$$

1-12) کافی است با اعداد داده شده درخت هافمن بسازیم. طول مسیر وزن داده شده برابر است با:

$$5\times 3 + 10\times 3 + 20\times 2 + 25\times 2 + 30\times 2 = 195$$

2-13) بدون در نظر گرفتن حلقه for جواب $T(n) = T(\left\lfloor \frac{n}{2} \right\rfloor) + 1$ است. بنابراین با در نظر گرفتن این حلقه داریم:

$$T() = n(T(\left\lfloor \frac{n}{2} \right\rfloor) + 1)$$

2-14) چون آرایه مرتب باشد، زمان اجرای الگوریتم $O(n^2)$ است.

1-15) در مسئله برج هانوی، مراحل زیر انجام می شود:

الف) انتقال n-1 مهره از میله مبدا به میله کمکی.

ب) انتقال مهره nام از میله مبدا به میله مقصد.

ج) انتقال n-1 مهره از میله کمکی به میله مقصد.

رابطه کلی به صورت زیر نوشته می شود:

$$T(n)=T(n-1)+1+T(n-1)=2T(n-1)+1$$

2-16) در هر مرحله یک مقایسه انجام شده و مقایسه یا در نیمه پایین و یا در نیمه بالا انجام می گیرد، یعنی تعداد عناصر نصف می شوند.

3-17) برای آرایه n عنصری مرتب سازی ادغامی شامل مراحل زیر می شود:

1- تقسیم آرایه به دو زیر آرایه ، هر یک با n/2 عنصر

2- حل هر زیر آرایه با مرتب سازی آن. اگر زیر آرایه به قدر کافی کوچک نبود برای آن از بازگشتی استفاده می کنیم.

3- ترکیب حل های زیرآرایه ها از طریق ادغام آنها در یک آرایه مرتب. (با حداکثر n-1 مقایسه)

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + (n-1)$$

3-18) مرتبه اجرایی Mergesort همواره $O(n\lg n)$ است.

$$T(n) = 2T(\frac{n}{2})$$

که با توجه به مطالب فصل اول ، حل آن برابر است با : $\theta(n)$

2-20) رابطه بازگشتی چنین الگوریتمی برابر است با: $T(n) = 4T(\frac{n}{2}) + cn$ که جواب آن برابر است با: $\theta(n \lg n)$

4-21) رابطه بازگشتی تعداد ضرب ها در روش استراسن برابر است با:

$$T(n) = 7T(\frac{n}{2})$$
$$T(1) = 1$$

بنابراین داریم:

$$T(2) = 7T(1) = 7$$
$$T(4) = 7T(2) = 7 \times 7 = 49$$

3_22)

310 , 285 , 179 , 625 , 351 , 423 , 861 , 254 , 450 , 520

ابتدا آرایه 10 عنصری به دو آرایه 5 عنصری تقسیم می شود (یکبار فراخوانی)

310 , 285 , 179 , 625 , 351 , 423 , 861 , 254 , 450 , 520

هر یک از آرایه های 5 عنصری به دو آرایه 2و 3 عنصری تقسیم می شود: (دو بار فراخوانی)

310 , 285 , 179 , 625 , 351 , 423 , 861 , 254 , 450 , 520

آرایه های 2 عنصری به دو آرایه 1عنصری و آرایه های 3 عنصری به دو آرایه 1و2 عنصری تقسیم می شوند. (چهار بار فراخوانی)

310 , 285 , 179, 625 , 351 , 423 , 861 , 254 , 450،520

آرایه های 2 عنصری به دو آرایه 1عنصری تقسیم می شوند. (دو بار فراخوانی)

310 , 285 , 179 , 625 , 351 , 423 , 861 , 254 , 450 , 520

سپس برای هر یک از 10 آرایه تک عنصری، نیز یک بار رویه فراخوانی می شود. بنابراین در نهایت 19=10+9 مرتبه رویه صدا زده می شود.

2-23) الگوریتم مرتب سازی سریع، فقط در بدترین حالت یعنی زمانی که داده ها مرتب باشند $O(n^2)$ است.

1-24) رابطه بازگشتی مرتب سازی ادغام برابر است با:

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + (n-1)$$

که حل این رابطه برابر است با: $O(n \lg n)$.

1-25) الگوریتم QuickSort یک رشته n تایی را در حالت متوسط با سرعت $O(n \lg n)$ مرتب می کند.

فصل سوم : برنامه نویسی پویا

روش تقسیم و حل نمونه ای از مسئله را با تقسیم آن به نمونه های کوچکتر و سپس حل این نمونه های کوچک تر حل می کند که یک روش بالا به پایین است و در مسائلی نظیر مرتب سازی ادغامی جواب می دهد که در آن ها نمونه های کوچک تر با هم ارتباطی ندارند ولی در مسائلی چون محاسبه جمله n ام دنباله فیبوناچی جواب نمی دهد، چون نمونه های کوچک تر با هم ارتباط دارند. برای مثال، برای محاسبه جمله پنجم فیبوناچی بایستی جمله های سوم و چهارم فیبوناچی را محاسبه می کردیم. چون الگوریتم تقسیم و حل این دو جمله را مستقل از هم تقسیم می کند، جمله دوم فیبوناچی را بیش از یک بار محاسبه می کند.

برنامه نویسی پویا (dynamic programming) از این لحاظ که نمونه را به نمونه های کوچکتر تقسیم می کند مشابه روش تقسیم و حل است. ولی در این روش ابتدا نمونه های کوچکتر را حل کرده و نتایج را ذخیره می کنیم و بعداً هرگاه به یکی از آن ها نیاز پیدا شد به جای محاسبه دوباره کافی است آن را بازیابی کنیم. در برنامه نویسی پویا از آرایه‌ای (جدولی) استفاده می شود که در آن یک حل بنا می شود. برنامه نویسی پویا یک روش پایین به بالا است.

مراحل بسط یک الگوریتم برنامه نویسی پویا به شرح زیر است:

1- ارائه یک ویژگی بازگشتی برای حل نمونه ای مسئله

2- حل نمونه ای از مسئله به شیوه پایین به بالا با حل نمونه های کوچکتر

هر مسئله بهینه سازی را نمی توان با استفاده از برنامه نویسی پویا حل کرد. اصل بهینگی باید در مسئله صدق کند.

تعریف: گفته می شود اصل بهینگی در یک مسئله صدق می کند اگر یک حل بهینه برای نمونه ای مسئله، همواره حاوی حل بهینه برای همه زیرنمونه ها باشد.

مثال: الگوریتم کارآمد برای محاسبه جمله n ام فیبوناچی مثالی از برنامه نویسی پویا است. این الگوریتم جمله n ام فیبوناچی را با n+1 جمله در آرایه f که از صفر تا n اندیس گذاری شده است، تعیین می کند.

تذکر: اصل بهینگی در همه مسائل بهینه سازی صدق نمی کند.

برنامه نویسی پویا از این لحاظ که به یک ویژگی بازگشتی برای تقسیم نمونه ای به نمونه های کوچک تر نیاز دارد، به روش تقسیم و حل شباهت دارد. اما در برنامه نویسی پویا از ویژگی بازگشتی برای حل تکراری نمونه ها به ترتیب و با شروع از نمونه کوچک‌تر، استفاده می شود. به این ترتیب هر نمونه کوچک تر فقط یک بار حل می شود.

ضریب دو جمله ای

برای محاسبه ضریب جمله k+1 ام از بسط $(a+b)^n$، از رابطه زیر استفاده می کنیم:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \qquad 0 \le k \le n$$

به طور نمونه ضریب جمله سوم بسط $(a+b)^3$ برابر است با:

$$\binom{3}{2} = \frac{3!}{2!\times(3-2)!} = 3$$

(یادآوری: $(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$)

مثال: طریقه بدست آوردن از بدست آوردن را بدست آورید.

$$\binom{8}{5}=\frac{8!}{5!\times(8-5)!}=\frac{8\times7\times6\times5!}{5!\times3!}=\frac{8\times7\times6}{3!}=56$$

برای مقادیری از n و k که بزرگ هستند، نمی توان ضریب دو جمله ای را مستقیماً از این تعریف محاسبه کنیم زیرا !n بزرگ است. در مبحث آنالیز درس آمار و احتمال رابطه زیر را مشاهده کرده ایم:

$$\binom{n}{k}=\begin{cases}\binom{n-1}{k-1}+\binom{n-1}{k} & 0<k<n\\ 1 & k=0 \quad or \quad k=n\end{cases}$$

با استفاده از این ویژگی بازگشتی می توان نیاز به محاسبه !n یا !k را برطرف کرد. پس به الگوریتم تقسیم و حل می رسیم.

الگوریتم : محاسبه ضریب دو جمله ای با استفاده از تقسیم و حل

هدف محاسبه ضریب دو جمله ای $\binom{n}{k}$ می باشد که n وk اعداد صحیح و مثبت می باشند و $k \le n$ است.

```
int   bin(int n , int  k){
  if (k==0  ||  n==k)
       return  1;
  else
      return  bin(n-1,k-1) + bin(n-1,k)
}
```

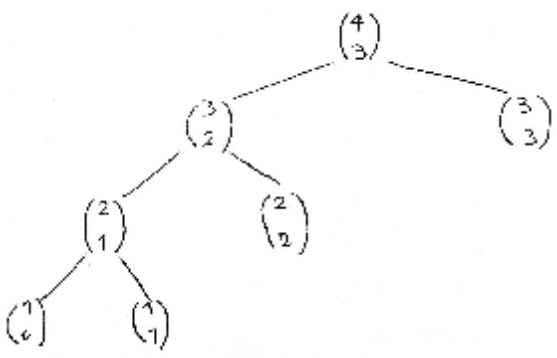
تعداد جملاتی که الگوریتم bin برای تعیین $\binom{n}{k}$ محاسبه می کند برابر $2\binom{n}{k}-1$ می باشد.

مثال: برای محاسبه $\binom{4}{3}$ چند جمله محاسبه می شود.

از آنجا که $\binom{4}{3}=4$ است، بنابراین طبق نکته قبل، تعداد محاسباتی که برای تعیین آن لازم است برابر است با:

$$2\times4-1=7$$

روش دوم: درخت بازگشتی را رسم کرده و تعداد گره های آن را می شماریم:



بازگشتی این الگوریتم کارایی بسیار کمی دارد، چون در هربار فراخوانی بازگشتی، نمونه ها چندین بار حل می شوند. برای مثال، bin(n-1,k-1) و bin(n-1,k) هر دو نیاز به نتیجه bin(n-2,k-1) دارند و این نمونه در هر فراخوانی بازگشتی به طور جداگانه محاسبه می شود.

روش تقسیم و حل مادامی که نمونه ای به دو نمونه کوچکتر تقسیم شود که تقریباً به بزرگی نمونه اولیه هستند، کارایی ندارد.

مراحل تولید یک الگوریتم برنامه نویسی پویا برای این مسئله

1- یک ویژگی بازگشتی ایجاد می کنیم. اگر بر حسب B بنویسیم، خواهیم داشت:

$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & 0 < j < i \\ 1 & j = 0 \text{ یا } j = i \end{cases}$$

2- نمونه ای از مسئله را به شیوه پایین به بالا با محاسبه سطرهای B به ترتیب و با شروع از سطر اول حل می کنیم. آخرین مقداری که محاسبه می شود B[n][k] همان $\binom{n}{k}$ است.

مثال: مقدار $\binom{4}{2} = B[4][2]$ را محاسبه کنید.

حل: ماتریس B[0..2][0..4] را در نظر می گیریم. ستون اول (شماره صفر) این ماتریس را 1 قرار می دهیم. همچنین در خانه هایی که شماره سطر و ستون آنها برابر است را نیز 1 قرار می دهیم. در حال حاضر ماتریس به شکل زیر است:

	0	1	2
0	1		
1	1	1	
2	1		1
3	1		
4	1		

سپس خانه های بعدی ماتریس را به کمک رابطه زیر ، سطر به سطر پر می کنیم:

B[i][j]=B[i-1][j-1]+B[i-1][j] i>j>0

B[2][1]=B[1][0]+B[1][1]=1+1=2

B[3][1]=B[2][0]+B[2][1]=1+2=3

B[3][2]=B[2][1]+B[2][2]=2+1=3

B[4][1]=B[3][0]+B[3][1]=1+3=4

B[4][2]=B[3][1]+B[3][2]=3+3=6

خانه های دیگر این ماتریس را سطر به سطر محاسبه می کنیم تا به خانه B[4][2] برسیم که جواب نهایی است. جدول نهایی به صورت زیر است:

0	1		
1	1	1	
2	1	2	1
3	1	3	3
4	1	4	6

در هر تکرار، مقادیر مورد نیاز برای آن تکرار قبلاً محاسبه و ذخیره شده اند. ■

الگوریتم: محاسبه ضریب دو جمله ای با استفاده از برنامه نویسی پویا

```
int  bin(int  n , int  k){
    index  i,j;
    int      B[0..n][0..k];
    for (i=0 ; i<= n ; i++)
        for (j=0 ; j<= minimum(i,k) ;  j++)
            if (j==0 || j==i)
                B[i][j]=1;
            else   B[i][j]=B[i-1][j-1]+B[i-1][j];
    return  B[n][k];
}
```

مثال: تعداد گذرهای انجام شده از حلقه j ، برای محاسبه bin(۵،۳) به صورت زیر است:

i	0	1	2	3	4	5
تعداد گذر	1	2	3	4	4	4

■

بنابراین در حالت کلی، تعداد گذرهای انجام شده از حلقه j در جدول زیر آورده شده است:

i	0	1	2	3	...	k	k+1	...	n
تعداد گذرها	1	2	3	4	...	k+1	k+1	...	k+1

با توجه به این جدول، پس تعداد کل گذرها عبارت است از:

$$1 + 2 + 3 + 4 + \ldots + k + (k+1) + (k+1) + \ldots + (k+1)$$

$$= \frac{k(k+1)}{2} + (n-k+1)(k+1) = \frac{(2n-k+2)(k+1)}{2} \in O(nk)$$

مثال: تعداد گذرهای انجام شده از حلقه j ، برای محاسبه bin(6،2) را بدست آورید.

حل: در رابطه بالا به جای k ، مقدار 2 و به جای n ، مقدار 6 را قرار می دهیم: $\frac{(2\times 6-2+2)\times(2+1)}{2} = \frac{12\times 3}{2} = 18$

i	0	1	2	3	4	5	6
تعداد گذر	1	2	3	3	3	3	3

الگوریتم فلوید برای یافتن کوتاه ترین مسیر

در این مسئله هدف یافتن کوتاه ترین مسیر از رأسی به رئوس دیگر است. کوتاه ترین مسیر باید مسیری ساده باشد. یک کاربرد متداول کوتاه ترین مسیر، تعیین کوتاه ترین مسیر میان دو شهر است. مسئله کوتاه ترین مسیر یک مسئله بهینه سازی است. برای هر نمونه از یک مسئله بهینه سازی ممکن است بیش از یک حل وجود داشته باشد. حل نمونه حلی است که دارای مقدار بهینه است. بسته به نوع مسئله، مقدار بهینه یا حداقل است یا حداکثر. در مورد مسئله کوتاه ترین مسیر، مقدار بهینه حداقل طول است. چون ممکن است بیش از یک کوتاه ترین مسیر از رأسی به رأس دیگر وجود داشته باشد، مسئله ما یافتن هر یک از این کوتاه ترین مسیرهاست.

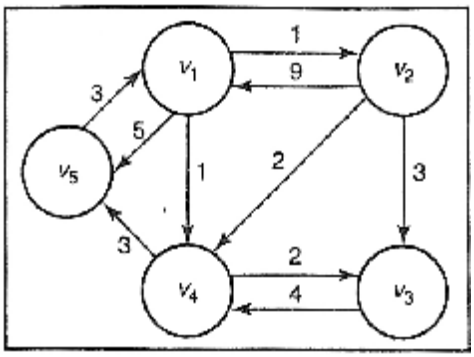
یک الگوریتم برای این مسئله، تعیین طول همه مسیرها، برای هر رأس، از آن رأس به هر یک از رئوس دیگر است. اما زمان این الگوریتم بدتر از زمان نمایی است. چون اگر از هر رأس به رئوس دیگر یک یال وجود داشته باشد، در این صورت، زیر مجموعه ای از همه مسیرها، عبارت از مجموعه ای خواهد بود که از رأس نخست شروع می شوند و به رأسی دیگر ختم می شوند و از همه رئوس دیگر عبور می کنند چون رأس دوم در چنین مسیری می تواند هر یک از **n - 2** رأس باشد، رأس سوم در چنین مسیری می تواند هر یک از **n - 3** رأس باشد،...، و رأس دومی به آخری روی چنین مسیری فقط می تواند یک رأس باشد، تعداد کل مسیرها از یک رأس که از همه رئوس دیگر بگذرد، عبارت است از:

$$(n-2)(n-3)\ldots 1 = (n-2)!$$

که بدتر از حالت نمایی است. در بسیاری از مسائل بهینه سازی با همین وضعیت مواجه هستیم. یعنی، الگوریتمی که همه حالت های ممکن را در نظر بگیرد زمان آن نمایی یا بدتر است.

با استفاده از برنامه نویسی پویا، یک الگوریتم زمانی درجه سوم برای مسئله کوتاه ترین مسیر ایجاد می کنیم. نخست الگوریتمی طرح می کنیم که فقط طول کوتاه ترین مسیرها را تعیین کند. سپس آن را طوری اصلاح می کنیم که کوتاه ترین مسیر را نیز ایجاد کند.

پیچیدگی زمانی الگوریتم فلوید در بدترین حالت برابر است با $\theta(n^3)$



حل: قطر اصلی ماتریس همجواری صفر است. در این ماتریس، W[i][j] وزن یال میان رأس i ام و رأس j ام است. اگر یال مستقیمی بین دو گره نباشد، در خانه مربوط به آن بی نهایت(∞) قرار می دهیم.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & \infty & 1 & 5 \\ 9 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix} \end{array}$$

مثال: با توجه به ماتریس همجواری مثال قبل، کوتاهترین مسیر را بدست آورید.

حل: گراف داده شده دارای پنج گره است. بنابراین ماتریس D[1..5][1..5] را در نظر می گیریم. ابتدا تمام خانه های این ماتریس را مانند ماتریس W در نظر می گیریم و به آن $D^{(0)}$ می گوییم. سپس به کمک رابطه زیر، $D^{(1)}, D^{(2)}, D^{(3)}, D^{(4)}, D^{(5)}$ ها را محاسبه می کنیم. $D^{(5)}$ جواب است.

$$D^{(k)}[i][j] = \min(D^{(k-1)}[i][j] \ , \ D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$$

به طور نمونه برای پیدا کردن کوتاهترین مسیر بین گره 2 و 5 مراحل زیر طی می شود:

$$D^{(1)}[2][5] = \min(D^{(0)}[2][5], D^{(0)}[2][1] + D^{(0)}[1][5]) = \min(\infty, 9+5) = 14$$

$$D^{(2)}[2][5] = \min(D^{(1)}[2][5], D^{(1)}[2][2] + D^{(1)}[2][5]) = \min(14, 0+14) = 14$$

$$D^{(3)}[2][5] = \min(D^{(2)}[2][5], D^{(2)}[2][3] + D^{(2)}[3][5]) = \min(14, 3+\infty) = 14$$

$$D^{(4)}[2][5] = \min(D^{(3)}[2][5], D^{(3)}[2][4] + D^{(3)}[4][5]) = \min(14, 2+3) = 5$$

$$D^{(5)}[2][5] = \min(D^{(4)}[2][5], D^{(4)}[2][5] + D^{(4)}[5][5]) = \min(5, 5+0) = 5$$

بنابراین طول کوتاه ترین مسیر از v2 به v5 برابر 5 است. برای این کار کافی است از v2 به v4 رفته و سپس از v4 به v5 برویم.

روابط زیر برقرار است:

$$D^{(k)}[i][k] = D^{(k-1)}[i][k] \quad , \quad D^{(k)}[k][j] = D^{(k-1)}[k][j]$$

با توجه به این نکته، لازم به محاسبه $D^{(2)}[2][5]$ و یا $D^{(5)}[2][5]$ نبود، چون:

$$D^{(2)}[2][5] = D^{(1)}[2][5]$$

$$D^{(5)}[2][5] = D^{(4)}[2][5]$$

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & 1 & 4 \\ 8 & 0 & 3 & 2 & 5 \\ 10 & 11 & 0 & 4 & 7 \\ 6 & 7 & 2 & 0 & 3 \\ 3 & 4 & 6 & 4 & 0 \end{bmatrix} \end{array}$$

الگوریتم فلوید

هدف محاسبه کوتاه ترین مسیر از هر رأس در یک گراف موزون به رئوس دیگر است. (وزن ها اعداد غیرمنفی هستند). ورودی ها، ماتریس همجواری گراف و تعداد رئوس موجود در گراف(n) است. خروجی این الگوریتم، یک آرایه دو بعدی D[1..n][1.n]، است که D[i][j] وزن کوتاه ترین مسیر میان راس i ام و راس j ام است.(تذکر: محاسبات را فقط به کمک یک آرایه به نام D انجام می دهیم.)

```
void  floyd (int  n , const  number  W[ ] [ ] , number  D[ ] [ ] )
{
  index  i,j,k;
  D=W;
  for (k=1;k<=n;k++)
     for (i=1;i<=n ; i++)
        for (j=1;j<=n;j++)
            D[i][j] = minimum(D[i][j],D[i][k]+D[k][J]);
}
```

تحلیل پیچیدگی زمانی در بدترین حالت برای الگوریتم فلوید

عمل اصلی، دستورهای موجود در حلقه for می باشد، و سه حلقه تو در تو داریم و از هر حلقه n بار گذر داریم، پس خواهیم داشت:

$$\mathbf{T(n) = n \cdot n \cdot n = n^3 \in q(n^3)}$$

تذکر: در پیوست این کتاب، الگوریتم فلوید (الگوریتم دوم) آورده شده است، که کوتاه ترین مسیرها نیز ایجاد می شود.

ضرب زنجیره‌ای ماتریس‌ها

تعداد حالات ممکن برای پرانتز گذاری در ضرب n ماتریس با استفاده از سری کاتالان محاسبه می شود. سری کاتالان به صورت زیر می باشد:

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i)$$

$$T(1) = 1$$

که حل این سری برابر است با: $\frac{1}{n}\binom{n-1}{}$

مثال: تعداد حالات ممکن برای پرانتز گذاری در ضرب ۴ ماتریس چقدرا است؟

حل: اگر در فرمول به جای n ، عدد 4 قرار دهیم جواب 5 خواهد بود. این پنج حالت در زیر آورده شده است:

(AB)(CD) ((A(BC))D) (A((BC)D)) (((AB)C)D) (A(B(CD)))

■

برای ضرب یک ماتریس با ابعاد $i \times j$ در یک ماتریس با ابعاد $j \times k$ به $i \times j \times k$ عمل ضرب نیاز است.

مثال: بهترین حالت ضرب ماتریس های زیر را مشخص کنید.

$$\underset{20\times 2}{A} \times \underset{2\times 30}{B} \times \underset{30\times 12}{C} \times \underset{12\times 8}{D}$$

ابعاد هر ماتریس در زیر آن مشخص شده است.

$A(B(CD))$ $30\times12\times8+2\times30\times8+20\times2\times8=3{,}680$

$(AB)(CD)$ $20\times2\times30+30\times12\times8+20\times30\times8=8{,}880$

$A((BC)D)$ $2\times30\times12+2\times12\times8+20\times2\times8=1{,}232$

$((AB)C)D$ $20\times2\times30+20\times30\times12+20\times12\times8=10{,}320$

$(A(BC))D$ $2\times30\times12+20\times2\times12+20\times12\times8=3{,}120$

ترتیب سوم برای ضرب این چهار ماتریس، ترتیب بهینه است، چون به کمترین تعداد ضرب نیاز است.■

اصل بهینگی در این مسئله صدق می‌کند. یعنی ترتیب بهینه برای ضرب n ماتریس، در برگیرندهٔ ترتیب بهینه برای ضرب هر زیر مجموعه‌ای از این n ماتریس است. برای مثال، اگر ترتیب بهینه برای ضرب شش ماتریس چنین باشد: $A_1((((A_2A_3)A_4)A_5)A_6)$ در این صورت: $(A_2A_3)A_4$ باید ترتیب بهینه برای ضرب ماتریس‌های A_2 تا A_4 باشد. این بدان معناست که می توان از برنامه‌نویسی پویا برای به دست آوردن حل استفاده کنیم.

ویژگی بازگشتی برای ضرب n ماتریس $(1 \le i \le j \le n)$:

$$M[i][j] = \min_{i\le k\le j-1} imum(M[i][k] + M[k+1][j] + d_{i-1}d_kd_j) \qquad i < j$$

$$M[i][j] = \mathbf{0} \qquad i=j$$

زمان الگوریتم تقسیم و حل، بر اساس این ویژگی، نمایی است. با استفاده از برنامه‌نویسی پویا برای محاسبه مقادیر $M[i][j]$ الگوریتمی با کارایی بیشتر طراحی می‌کنیم و از شبکه‌ای مشابه مثلث خیام استفاده خواهیم کرد.

نحوه انجام محاسبات : نخست، همهٔ عناصر قطر اصلی را صفر می‌کنیم، سپس همهٔ عناصر قطری را که درست در بالای آن قرار دارد و آن را قطر 1 می‌نامیم، محاسبه می‌کنیم، سپس همهٔ عناصر قطر 2 را محاسبه می‌کنیم و .... به این شیوه ادامه می‌دهیم تا تنها عنصر موجود در M[1][n] که جواب نهایی است، محاسبه شود.

$$\underset{20\times 2}{A} \times \underset{2\times 30}{B} \times \underset{30\times 12}{C} \times \underset{12\times 8}{D}$$

حل: ابتدا ماتریس M[1..4][1..4] با قطر صفر به صورت زیر ایجاد می کنیم:

	1	2	3	4
1	0			
2		0		
3			0	
4				0

سپس عناصر قطر 1و2و3 را به صورت زیر محاسبه می کنیم.

محاسبه عناصر قطر اول:

$$M[1][2] = M[1][1] + M[2][2] + d_0 d_1 d_2 = 0 + 0 + 20\times 2\times 30 = 120$$

$$M[2][2] = M[2][2] + M[3][3] + d_1 d_2 d_3 = 0 + 0 + 2\times 30\times 12 = 720$$

$$M[3][4] = M[3][3] + M[4][4] + d_2 d_3 d_4 = 0 + 0 + 30\times 12\times 8 = 2880$$

محاسبه عناصر قطر دوم:

$$M[1][3] = \begin{cases} k=1 \Rightarrow M[1][1] + M[2][3] + d_0 d_1 d_3 = 0 + 720 + 20\times 2\times 12 = 1200 \\ k=2 \Rightarrow M[1][2] + M[3][3] + d_0 d_2 d_3 = 120 + 0 + 20\times 30\times 12 = 7320 \end{cases}$$

بنابراین M[1][3] برابر 1200 است. (عدد کوچکتر بین 1200 و 7320)

$$M[2][4] = \begin{cases} k=2 \Rightarrow M[2][2] + M[3][4] + d_1 d_2 d_4 = 0 + 2880 + 2\times 30\times 8 = 3360 \\ k=3 \Rightarrow M[2][3] + M[4][4] + d_1 d_3 d_4 = 720 + 0 + 2\times 12\times 8 = 912 \end{cases}$$

بنابراین M[2][4] برابر 912 است.

محاسبه عنصر قطر سوم:

$$M[1][4] = \begin{cases} k=1 \Rightarrow M[1][1] + M[2][4] + d_0 d_1 d_4 = 0 + 912 + 20\times 2\times 8 = 1232 \\ k=2 \Rightarrow M[1][2] + M[3][4] + d_0 d_2 d_4 = 120 + 2880 + 20\times 30\times 8 = 7800 \\ k=3 \Rightarrow M[1][3] + M[4][4] + d_0 d_3 d_4 = 1200 + 0 + 20\times 12\times 8 = 3120 \end{cases}$$

بنابراین M[1][4] برابر 1232 است.

ماتریس M به صورت زیر می باشد:

	1	2	3	4
1	0	120	1200	1232
2		0	720	912
3			0	2880
4				0

برای مشخص شدن نحوه ضرب ها از یک ماتریس به نام P[1..n-1][1..n] استفاده می کنیم و در هر خانه های آن مقـدار k ای را که به ازای آن مقدار حداقل مشخص شد را قرار می دهیم. این ماتریس به صورت زیر می باشد:

	1	2	3	4
1		1	1	1
2			2	3
3				3

به طور نمونه خانه P[1][4] برابر 1 است، چون به ازای k=1 مقدار حداقل در محاسبه M[1][4] حاصل شد. و یا P[2][4] برابر 3 است، چون به ازای k=3 مقدار حداقل برای محاسبه M[2][4] حاصل شد.

تعیین نحوه ضرب به کمک ماتریس P:(ماتریس اول=A،ماتریس دوم=B،ماتریس سوم=C وماتریس چهارم=D)

مقدار $P[i][j]$ مشخص کننده نقطه‌ای است که در آن ماتریس‌های i تا j به ترتیبی بهینه برای ضرب شدن، از هم جدا می‌شوند.

چون خانه P[1][4] برابر 1 است، بنابراین پرانتز اول بعد از ماتریس اول قرار داده می شود (A)(BCD) . حال چون نحـوه ضـرب سه ماتریس BCD مشخص نیست، به P[2][4] نگاه کرده و چون برابر 3 است، یعنی بعد از مـاتریس سـوم یعنـی C بایـد پرانتـز گذاشت:

(A((BC)D)

مثال: فرض کنید شش ماتریس زیر را داریم:

$$\underset{5\times 2}{A_1} \times \underset{2\times 3}{A_2} \times \underset{3\times 4}{A_3} \times \underset{4\times 6}{A_4} \times \underset{6\times 7}{A_5} \times \underset{7\times 8}{A_6}$$

$$d_0\ d_1 \quad d_1\ d_2 \quad d_2\ d_3 \quad d_3\ d_4 \quad d_4\ d_5 \quad d_5\ d_6$$

محاسبه قطر ۱:

$$M[1][2] = \min_{1\le k\le 1} imum(M[1][k] + M[k+1][2] + d_0 d_k d_2) = M[1][1] + M[2][2] + d_0 d_1 d_2$$

$$= 0 + 0 + 5\times 2\times 3 = 30$$

مقادیر M[2][3] ، M[3][4] ، M[4][5] و M[5][6] به همین شیوه محاسبه می‌شوند.

محاسبه قطر ۲:

$$M[1][3] = \min_{1\le k\le 2} imum(M[1][k] + M[k+1][3] + d_0 d_k d_3)$$

$$= \min imum(M[1][1] + M[2][3] + d_0 d_1 d_3, M[1][2] + M[3][3] + d_0 d_2 d_3)$$

$$\min imum(0 + 24 + 5\times 2\times 4, 30 + 0 + 5\times 3\times 4) = 64$$

مقادیر $M[2][4]$، $M[3][5]$ و $M[4][6]$ نیز به همان شیوه محاسبه می شوند.

$$M[1][4] = \min_{1\le k\le 3} imum(M[1][k] + M[k+1][4] + d_0 d_k d_4)$$

$$= \min imum(M[1][1] + M[2][4] + d_0 d_1 d_4, M[1][2] + M[3][4] + d_0 d_2 d_4,\ M[1][3] + M[4][4] + d_0 d_3 d_4)$$

$$= \min imum(\mathbf{0} + 72 + 5\times 2\times 6{,}30 + 72 + 5\times 3\times 6{,}64 + \mathbf{0} + 5\times 4\times 6) = 132$$

مقادیر دیگر این قطر به همان شیوه محاشبه می شوند.

عناصر قطر 4 و 5 نیز به همین شیوه محاسبه می شوند. عنصر M[1][6] حل نمونهٔ مورد نظر ما است. یعنی حداقل تعداد ضرب-های ساده و مقدار آن عبارت است از: $M[1][6] = 348$. آرایه M در زیر نشان داده شده است:

	1	2	3	4	5	6
1	0	30	64	132	226	348
2		0	24	72	156	268
3			0	72	198	366
4				0	168	392
5					0	336
6						0

نحوه به دست آوردن پرانتز گذاری:

آرایه P، برای این مثال در شکل زیر نشان داده شده است:

	1	2	3	4	5	6
1		1	1	1	1	1
2			2	3	4	5
3				3	4	5
4					4	5
5						5

چون P[1][6] برابر 1 است، داریم : $A_1(A_2A_3A_4A_5A_6)$.

چون P[2][6] برابر 5 است، داریم: $A_1((A_2A_3A_4A_5)A_6)$.

چون P[2][5] برابر 4 است، داریم: $A_1(((A_2A_3A_4)A_5)A_6)$

چون P[2][4] برابر 3 است، داریم: $A_1((((A_2A_3)A_4)A_5)A_6)$

بنابراین ترتیب بهینه ضرب 6 ماتریس که منجر به کمترین تعداد ضرب می شود برابر است با: $A_1((((A_2A_3)A_4)A_5)A_6)$ ■

الگوریتم حداقل ضرب‌ها

می خواهیم حداقل تعداد ضرب‌های اصلی مورد نیاز برای ضرب n ماتریس و ترتیبی که حداقل تعداد را به دست می‌دهد را مشخص کنیم. ورودی‌ها الگوریتم، تعداد ماتریس‌ها(n) و آرایه d[0..n] است که ابعاد ماتریس ها در آن ذخیره شده اند. این الگوریتم، یک آرایه دو بُعدی P[1..n-1،1..n] که ترتیب بهینه را از آن می توان به دست آورد، را به عنوان خروجی بر می گرداند.

```
int   minmult (int  n, const int  d[ ] , index P[ ][ ] ){
      index   i, j, k,a,t; int  M [1..n][1..n];
```

```
    M[a][a]=0;
for (t =1 ;  t <= n–1 ;  t++)
    for (i=1 ;  i<= n–t  ;  i++){
        j= i+t;
        M[i][j] =  min imum  (M[i][k] + M[ t+ 1][j] + d [i-1] * d[k] * d[j]);
                   i≤k≤ j–1
        P[i][j] = a value of k that gave the minimum;
    }
return   M[1][n];   }
```

تحلیل پیچیدگی زمانی حالت معمول برای الگوریتم حداقل ضرب‌ها

عمل اصل، مقایسه‌ای را که برای آزمون حداقل انجام می‌شود، می باشد. سه حلقهٔ تودرتو داریم. به ازای مقادیر معلـوم t و i، تعـداد گذرها از حلقهٔ k عبارت است از: $j - l - i + 1$ و چون j=i+t است، این رابطه برابر است با: $i + t - 1 - i + 1 = t$. بـه ازای مقدار معلومی از t، تعداد گذرها از حلقهٔ for i برای n-t است. چون t از یک تا $n-1$ تغییر می‌کند، تعداد کل دفعاتی کـه عمـل اصلی انجام می‌شود، برابر $\sum_{t=1}^{n-1} t(n-t)$ بوده که این عبارت برابر است با: $\frac{n(n-1)(n+1)}{6} \in q(n^3)$

روش دوم:

$$\sum_{t=1}^{n-1}\sum_{i=1}^{n-t}\sum_{k=i}^{j-1}1=\sum_{t=1}^{n-1}\sum_{i=1}^{n-t}(j-1-i+1)=\sum_{t=1}^{n-1}\sum_{i=1}^{n-t}(i+t-1-i+1)=\sum_{t=1}^{n-1}\sum_{i=1}^{n-t}t=\sum_{t=1}^{n-1}t(n-t)=\frac{n(n-1)(n+1)}{6}$$

یاد آوری از کتاب ساختمان داده های مهندس شیرافکن: $\sum_{i=1}^{n-1} i(n-i)=\frac{n(n-1)(n+1)}{6}$

مثال: تعداد دفعاتی که دستور Minimum اجرا می شود(یعنی تعداد دفعاتی که k مقـدار مـی گیـرد)، در ضـرب ۴ ماتریس چقدر است؟

حل: با توجه به فرمول داریم:

$$\frac{4(4-1)(4+1)}{6}=10$$

روش دوم : می توان تعداد تکرار دستور داخل سه حلقه را مشخص کرد:

t	i	j	k
1	1	2	1
	2	3	2
	3	4	3
2	1	3	1،2
	2	4	2،3
3	1	4	1،2،3

همان طور که در جدول بالا مشاهده می کنید، تعداد دفعاتی که k مقدار می گیرد برابر 10 می باشد.

می خواهیم ترتیب بهینه برای ضرب n ماتریس را چاپ کنیم. این الگوریتم از آرایه P که توسط الگوریتم قبل پیدا شد، استفاده می کند. $P[i][j]$ نقطه‌ای است که در آن ماتریس‌های i تا j به ترتیبی بهینه برای ضرب آن ماتریس‌ها جدا می‌شوند.

```
void   order (index   i , index   j) {
     if (i == j)  cout < < "A" < < i;
     else {
            k = P[i][j];
            cout < < "(";
            order(i, k);
            order(k+1, j);
            cout <  ")";
          }
}
```

برای الگوریتم order داریم: $T(n) \in q(n)$

خروجی الگوریتم order برای ضرب 6 ماتریس چند مثال قبل، به صورت زیر می باشد:

$A_1((((A_2A_3)A_4)A_5)A_6)$.

قضیه: یک الگوریتم غیر هوشمند برای ضرب زنجیره ای ماتریس ها عبارت است از در نظر گرفتن همهٔ ترتیب‌های ممکن و پیدا کردن حداقل ضرب‌های انجام شده است. این الگوریتم حداقل به صورت نمایی است.

اثبات: فرض کنید T(n) تعداد ترتیب‌های متفاوت برای ضرب n ماتریس $A_1, A_2, ..., A_n$ در یکدیگر باشد. یک زیر مجموعه از این ترتیب‌ها شامل ترتیب‌هایی می‌شود که در آن‌ها A_1 آخرین ماتریسی است که ضرب می‌شود. تعداد ترتیب‌های متفاوت در این زیر مجموعه، T(n-1) می‌شود، زیرا این تعداد ترتیب‌های متفاوتی است که با آن می توان A_2 تا A_n را در هم ضرب نمود. یک زیر مجموعه دیگر، عبارت از همهٔ ترتیب‌هایی است که در آن A_n آخرین ماتریسی است که ضرب می‌شود. بدیهی است تعداد ترتیب‌های متفاوت در این زیر مجموعه نیز T(n-1) است. بنابراین داریم:

$T(n) \geq 2T(n-1)$

چون فقط یک راه برای ضرب دو ماتریس وجود دارد، T(2)=1 است. با حل این دستور بازگشتی داریم:

$T(n) \geq 2^{n-2}$

درخت‌های جست‌وجوی دودویی بهینه

درخت جست‌وجوی دودویی(BST)، یک درخت دودویی از عناصر (کلید) است که از یک مجموعه مرتب حاصل می‌شود، به قسمی که:

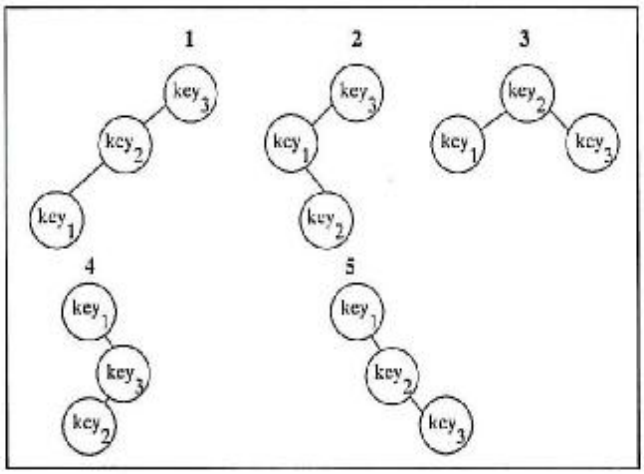
1- هر گره حاوی یک کلید است.

2- کلیدهای موجود در زیر درخت چپ یک گره مفروض، کوچک‌تر از کلید آن گره هستند.

هدف ما سازمان‌دهی کلیدها در یک BST است به قسمی که زمان میانگین برای تعیین مکان کلیدها به حداقل برسد. درختی کـه به این شیوه سازمان‌دهی می‌شود درخت بهینه نام دارد. فرض می‌کنیم $key_1, key_2, key_3, ..., key_n$، کلیدهای مرتب شده، و p_i احتمال مساوی بودن کلید key_i با کلید مورد جست‌وجو باشد. اگر c_i تعداد مقایسـه‌هـای مـورد نیـاز بـرای یـافتن key_i در یـک درخت مفروض باشد، زمان جست‌وجوی میانگین برای آن درخت، به صورت $\sum_{i=1}^{n} c_i p_i$ است و می‌خواهیم آن را به حداقل برسانیم.

مثال: فرض کنید key_1, key_2, key_3 کلیدهای مرتب شده باشند. احتمال مساوی بودن کلید key_1 با کلید مورد جستجو برابـر 0,7 و احتمال مساوی بودن کلید key_2 با کلید مورد جستجو برابر 0,2 ، احتمال مساوی بودن کلید key_3 با کلید مورد جسـتجو برابر 0,1 می باشد. زمان های جستجوی میانگین را مشخص کنید. ($P_1 = 0.7, P_2 = 0.2, P_3 = 0.1$)

حل: با 3 کلید داده شده می توان 5 درخت متفاوت به صورت زیر ساخت.



1. 3(0,7) + 2(0,2) + 1(0,1) = 2,6
2. 2(0,7) + 3(0,2) + 1(0,1) = 2,1
3. 2(0,7) + 1(0,2) + 2(0,1) = 1,8
4. 1(0,7) + 3(0,2) + 2(0,1) = 1,5
5. 1(0,7) + 2(0,2) + 3(0,1) = 1,4

مشخص است که حالت آخر بهترین است.چون کمترین زمان جستجو را دارد.

به طور کلی، یک BST بهینه را نمی توان با در نظر گرفتن همهٔ BST ها یافت، زیرا تعـداد چنـین درخـت‌هـایی، حـداقل رابطه نمایی با n دارد.

هر زیر درخت بهینه‌ای از یک درخت بهینه، برای کلیدهای موجود در آن زیر درخت بهینه است. بنابراین، اصل بهینگی برقرار است.

اگر فقط همه درخت‌های جست‌وجو با عمق $n-1$ را بیابیم، تعداد درخت‌ها، عددی نمایی خواهد بود. در یک BST با عمق $n-1$، تنها گره در هر یک از $n-1$ سطح غیر از ریشه می تواند در طرف راست یا چپ والد (parent) خود باشد. یعنی در هر یک از آن سطوح دو امکان وجود دارد یعنی تعداد درخت‌های جست‌وجوی مختلف با عمق $n-1$ برابر 2^{n-1} است.

در الگوریتم تعیین یک BST بهینه از رابطه زیر استفاده می کنیم:

$$A[i][j] = \min_{i \le k \le j} imum(A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} p_m \qquad i < j$$

$$A[i][i] = p_i \quad , \quad A[i][i-1] = 0 \quad , \quad A[j+1][i] = 0$$

با محاسبهٔ ترتیبی مقادیر روی هر قطر کار را ادامه می‌دهیم.

مثال: با توجه به رابطه گفته شده، حداقل زمان میانگین جست‌وجو را مشخص کنید.

حل: ماتریسی با ابعاد A[1..n+1][0..n] در نظر می گیریم. قطر اصلی آن را صفر قرار داده و احتمال ها را در قطر یک قرار می دهیم. حال عناصر آرایه را به ترتیب قطری به کمک رابطه بدست می آوریم.

محاسبه A[۱][۲]:

$$k=1 \Rightarrow A[1][2] = A[1][0] + A[2][2] + P_1 + P_2 = 0 + 0.2 + 0.7 + 0.2 = 1.1$$
$$k=2 \Rightarrow A[1][2] = A[1][1] + A[3][2] + P_1 + P_2 = 0.7 + 0 + 0.7 + 0.2 = 1.6$$

و کمترین مقدار یعنی 1,1 را در خانه A[1][2] قرار می دهیم.

محاسبه A[۲][۳]:

$$k=2 \Rightarrow A[2][3] = A[2][1] + A[3][3] + P_2 + P_3 = 0 + 0.1 + 0.2 + 0.1 = 0.4$$
$$k=3 \Rightarrow A[2][3] = A[2][2] + A[4][3] + P_2 + P_3 = 0.2 + 0 + 0.2 + 0.1 = 0.5$$

و کمترین مقدار یعنی 0,4 را در خانه A[2][3] قرار می دهیم.

محاسبه A[۱][۳]:

$$k=1 \Rightarrow A[1][3] = A[1][0] + A[2][3] + P_1 + P_2 + P_3 = 0 + 0.4 + 0.7 + 0.2 + 0.1 = 1.4$$
$$k=2 \Rightarrow A[1][3] = A[1][1] + A[3][3] + P_1 + P_2 + P_3 = 0.7 + 0.1 + 0.7 + 0.2 + 0.1 = 1.8$$
$$k=3 \Rightarrow A[1][3] = A[1][2] + A[4][3] + P_1 + P_2 + P_3 = 1.1 + 0 + 0.7 + 0.2 + 0.1 = 2.1$$

و کمترین مقدار یعنی 1,4 را در خانه A[1][3] قرار می دهیم.

ماتریس پر شده به صورت زیر است. جواب نهایی در خانه A[1][3] قرار دارد:

	0	1	2	3
1	0	0,7	1,1	1,4
2		0	0,2	0,4
3			0	0.1
4				0

می دهیم که به ازای آن مقدار حداقل پیدا شده بود. به طور نمونه چون مقدار A[1][2] به ازای k=1 بدست آمده بود، در R[1][2] مقدار 1 را قرار می دهیم. و یا چون مقدار A[2][3] به ازای k=2 بدست آمده بود در R[2][3] مقدار 2 را قرار می دهیم. در نهایت چون مقدار A[1][3] به ازای k=1 بدست آمده بود، در R[1][3] مقدار 1 را قرار می دهیم. ماتریس R در زیر نشان داده شده است:

	0	1	2	3
1	0	1	1	1
2		0	2	2
3			0	3
4				0

با توجه به ماتریس R از آنجا که R[1][3] برابر 1 می باشد، مشخص است که بین سه گره، key1 ریشه اصلی است. حال با نگاه به R[2][3] که برابر 2 است، مشخص می شود که بین key2 و key3 ، key3 ریشه است. درخت نهایی، همان شکل 5 در مثال قبل است. که ریشه key1 بود و key2 فرزند راست آن و key3 فرزند راست key2 بود.

الگوریتم درخت جست‌وجوی دودویی بهینه

می خواهیم یک BST بهینه برای مجموعه‌ای از کلیدها(هر یک با احتمالی مشخص) را تعیین کنیم. ورودی‌ها، تعداد کلیدها (n) و آرایه‌ای از اعداد حقیقی P[1..n] است.(P[i] احتمال جست‌وجوی کلید i اُم). خروجی الگوریتم متغیر minavg است، که مقدار زمان جست‌وجوی میانگین برای یک BST بهینه در نهایت در آن ذخیره می شود. همچنین یک خروجی دیگر به نام R[1..n+1][0..n] داریم که که از روی آن می توان یک درخت بهینه ایجاد کرد. $R[i][j]$، اندیس کلید در ریشه یک درخت بهینهٔ حاوی کلیدهای i اُم تا j اُم است.

```
void optsearchtree (int n, const float p[ ] ,float& minavg , index R[ ][ ]) {
    index i, j, k, t; float A[1..n+1][0..n];
    for (i=1 ; i <= n ; i++){ A[i][i–1] = 0; A[i][i] = p[i]; R[i][i] = i; R[i][i–1] = 0; }
   A[n+1][n]= 0;
   R[n+1][n] = 0;
   for (t = 1; t <= n–1 ; t ++)
        for (i =1; i <= n–t ; i++){
            j = i + t;
            A[i][j] = min imum (A[i][k – 1] + A[ k+ 1][j] )+ ∑_{m=i} p_m;
            R[i][j] = a value of k that gave the minimum;
         }
    minavg= A[1][n];
}
```

عمل اصلی را دستورات اجرا شده به ازای هر مقدار از k در نظر می گیریم. این دستورات شامل یک مقایسه برای آزمون حداقل است. اندازه ورودی تعداد کلیدها یعنی n می باشد. سه حلقه تودرتو داریم :

$$\sum_{t=1}^{n-1}\sum_{i=1}^{n-t}\sum_{k=i}^{j}1=\sum_{t=1}^{n-1}\sum_{i=1}^{n-t}(j-i+1)=\sum_{t=1}^{n-1}\sum_{i=1}^{n-t}(i+t-i+1)=\sum_{t=1}^{n-1}\sum_{i=1}^{n-t}(t+1)=\sum_{t=1}^{n-1}(n-t)(t+1)$$

حل نهایی برابر است با:

$$T(n)=\frac{n(n-1)(n+4)}{6}\in q(n^3)$$

یاد آوری از کتاب ساختمان داده های مهندس شیرافکن :

$$\sum_{i=1}^{n-1}(i+1)(n-i)=\frac{n(n-1)(n+4)}{6}$$

مثال: تعداد دفعاتی که عمل اصلی(مقدار دهی به k) در ایجاد یک BST بهینه با 3 کلید انجام می گیرد را مشخص کنید.

حل: می توان در رابطه به دست آمده به جای n مقدار 3 را قرار دهیم: $\frac{3\times2\times7}{6}=7$

روش دوم : الگوریتم را ردیابی کنیم:

t	i	j	k
1	1	2	1،2
	2	3	2،3
2	1	3	1،2،3

بنابراین مشخص است که k ، هفت مرتبه مقدار دهی شده است.

الگوریتم زیر یک درخت دودویی از آرایه R ایجاد می‌کند. R حاوی اندیس کلیدهای انتخاب شده برای ریشه، در هر مرحله، است.

الگوریتم ساخت BST بهینه

مسئله: ساخت یک درخت جست‌وجوی دودویی بهینه.

ورودی: n، تعداد کلیدها؛ آرایه key حاوی n کلید مرتب و آرایه R که توسط الگوریتم قبلی ایجاد شده است. $R[i][j]$ اندیس کلید در ریشهٔ یک درخت بهینهٔ حاوی کلیدهای i اُم تا j اُم است.

خروجی‌ها: اشاره‌گر tree به درخت جست‌وجوی دودویی بهینه حاوی n کلید.

```
node_pointer  tree (indedx  i, j){
    index  k;  node_pointer  p;
    k = R[i][j];
    if (k == 0)  return  NULL;
   else {
        p = new  nodetype;
        p -> key = key[k];
```

```
            p -> right = tree (k+1, j);
            return  p;
          }
}
```

دستور p = new nodetype یک گره جدید را گرفته آدرس آن را در p قرار می‌دهد.

مثال: با توجه به مقادیر زیر برای آرایه Key و $p_1 = \frac{3}{8}$, $p_2 = \frac{3}{8}$, $p_3 = \frac{1}{8}$, $p_4 = \frac{1}{8}$ ، آرایه‌های A و R را مشخص کنید.

1	2	3	4
2	5	6	9

حل:

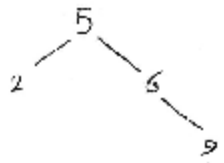
	0	1	2	3	4
1	0	$\frac{3}{8}$	$\frac{9}{8}$	$\frac{11}{8}$	$\frac{7}{4}$
2		0	$\frac{3}{8}$	$\frac{5}{8}$	1
3			0	$\frac{1}{8}$	$\frac{3}{8}$
4				0	$\frac{1}{8}$
5					0

A

	0	1	2	3	4
1	0	1	1	2	2
2		0	2	2	2
3			0	3	3
4				0	4
5					0

R

درخت ایجاد شده توسط الگوریتم، در شکل زیر نشان داده شده است. حداقل زمان میانگین جست‌وجو برابر $\frac{7}{4}$ است.

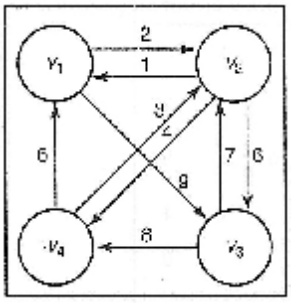


تذکر: مقدار $R[1][2]$ می تواند یک یا دو باشد. چون هر کدام از این اندیس‌ها می تواند اندیس ریشه در درخت بهینه‌ای باشد که حاوی دو کلید نخست است. بنابراین، هر یک از این دو اندیس، حداقل مقدار $A[1][2]$ را در دو الگوریتم قبل مشخص می‌کنند و این بدان معناست که هر یک از آن‌ها را می توان برای $R[1][2]$ برگزید.

مسئله فروشنده دوره گرد، یافتن یک تور بهینه در گرافی موزون و جهت دار است. یک (مدار هامیلتون) در یک گراف جهت دار، عبارت است از مسیری از یک راس به خودش که از هر کدام از روس دیگر دقیقا یک بار عبور می کند. یک تور بهینه، چنین مسیری با طول حداقل است.

فرض کنید فروشنده‌ای در نظر دارد به یک سفر فروشندگی برود که شامل 20 شهر می‌شود. هر شهر توسط جاده‌ای به چند شهر دیگر متصل است. برای به حداقل رساندن زمان سفر می‌خواهیم کوتاه‌ترین مسیر را با شروع از موطن فروشنده، با یک بار عبور از همهٔ شهرها و بازگشت به موطن تعیین کنیم. این مسئله را تعیین کوتاه‌ترین مسیر برای مسئله فروشنده دوره‌گرد می‌گویند. نمونه-ای از این مسئله را می‌توان با یک گراف موزون نشان داد که در آن هر رأس نشانگر یک شهر است. فرض می‌کنیم که اوزان مقادیر غیر منفی‌اند.

مثال: تورهای گراف زیر با شروع از v۱ را مشخص کنید.



حل: سه تور وجود دارد. که آخرین تور بهینه است.

$[v1, v2, v3, v4, v1] = 22$
$[v1, v3, v2, v4, v1] = 26$
$[v1, v3, v4, v2, v1] = 21$

نکته: اگر از هر راس به رئوس دیگر یک یال وجود داشته باشد، تعداد کل تورها برابر $(n-1)!$ می باشد.

سوال: آیا برنامه نویسی پویا را می توان برای مسئله فروشنده دوره گرد به کار برد؟

جواب: بله- چون اصل بهینگی برقرار است. اگر vk اولین راس پس از v1 روی هر تور بهینه باشد، زیر مسیر آن تور از vk به v1 باید کوتاهترین مسیر از vk به v1 باشد که از هر کدام از رئوس دیگر دقیقا یک بار عبور می کند.

با استفاده از تساوی زیر یک الگوریتم برنامه نویسی پویا برای مسئله فروشنده دوره گرد طرح می کنیم.

$$D[i][A] = \min_{j:vj\in A} imum(W[i][j] + D[j][A - \{v_j\}]$$

$$D[i][\phi] = W[i][1]$$

ماتریس D به صورت D[1..n][subset of V-{v1}] می باشد.

مثال: یک تور بهینه برای گرافی که ماتریس همجواری آن به صورت زیر است، تعیین کنید.(همان گراف مثال قبل)

	1	2	3	4
1	0	2	9	∞
1	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

حل: ماتریس D به صورت $D[1..4][subset\ \ of\{v_2, v_3, v_4\}]$ می باشد. که زیر مجموعه های $\{v_2, v_3, v_4\}$برابر است با:

$$\{\{\phi\},\{v_2\},\{v_3\},\{v_4\},\{v_2, v_3\},\{v_2, v_4\},\{v_3, v_4\},\{v_2, v_3, v_4\}\}$$

بنابراین ماتریس دارای 4 سطر و 8 ستون است. ابتدا طبق رابطه $D[i][\phi] = W[i][1]$ ، ستون اول را پر می کنیم:

$$D[2][\phi] = W[2][1] = 1 \quad , \quad D[3][\phi] = W[3][1] = \infty \quad , \quad D[4][\phi] = W[4][1] = 6$$

	$\{\phi\}$	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_2, v_3\}$	$\{v_2, v_4\}$	$\{v_3, v_4\}$	$\{v_2, v_3, v_4\}$
1	-	-	-	-	-	-	-	
2	1	-			-	-		
3	∞		-		-		-	
4	6			-		-	-	

حال مجموعه های یک عنصری را تعیین می کنیم. (خانه های خاکستری در شکل بالا)

در مجموعه های تک عنصری چون $A = v_j$ بنابراین $A - \{v_j\} = \phi$ و داریم:

$$D[i][A] = \min_{vj\in A} imum(W[i][j] + D[j][A - \{v_j\}] = W[i][j] + D[j][\phi]$$

به عبارتی :

$$D[i][\{v_j\}] = W[i][j] + D[j][\phi]$$

با توجه به این رابطه داریم:

$$D[2][\{v_3] = W[2][3] + D[3][\phi] = 6 + \infty = \infty$$

$$D[2][\{v_4\}] = W[2][4] + D[4][\phi] = 4 + 6 = 10$$

$$D[3][\{v_2\}] = W[3][2] + D[2][\phi] = 7 + 1 = 8$$

$$D[3][\{v_4\}] = W[3][4] + D[4][\phi] = 8 + 6 = 14$$

$$D[4][\{v_3\}] = W[4][3] + D[3][\phi] = \infty + \infty = \infty$$

در این لحظه ماتریس به صورت زیر است:

	$\{\phi\}$	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_2,v_3\}$	$\{v_2,v_4\}$	$\{v_3,v_4\}$	$\{v_2,v_3,v_4\}$
1	-	-	-	-	-	-	-	
2	1	-	∞	10	-	-		
3	∞	8	-	14	-		-	
4	6	4	∞	-		-	-	

سپس مجموعه های دو عنصری را تعیین می کنیم،یعنی خانه های خاکستری در شکل بالا:

$$D[2][\{v_3,v_4\}] = \begin{cases} W[2][3] + D[3][\{v_4\}] = 6 + 14 = 20 \\ W[2][4] + D[4][\{v_3\}] = 4 + \infty = \infty \end{cases}$$

بنابراین $D[2][\{v_3,v_4\}] = 20$ است.

$$D[3][\{v_2,v_4\}] = \begin{cases} W[3][2] + D[2][\{v_4\}] = 7 + 10 = 17 \\ W[3][4] + D[4][\{v_2\}] = 8 + 4 = 12 \end{cases}$$

بنابراین $D[3][\{v_2,v_4\}] = 17$ است.

$$D[4][\{v_2,v_3\}] = \begin{cases} W[4][2] + D[2][\{v_3\}] = 3 + \infty = \infty \\ W[4][3] + D[3][\{v_2\}] = \infty + 8 = \infty \end{cases}$$

و در نهایت خانه $D[1][\{v_2,v_3,v_4\}]$ را که جواب نهایی است را پیدا می کنیم:

$$D[1][\{v_2,v_3,v_4\}] = \begin{cases} W[1][2] + D[2][\{v_1,v_3\}] = 2 + 20 = 22 \\ W[1][3] + D[3][\{v_2,v_4\}] = 9 + 12 = 21 \\ W[1][4] + D[4][\{v_2,v_3\}] = \infty + \infty = \infty \end{cases}$$

بنابراین جواب تور بهینه برابر 21 است.

	$\{\phi\}$	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_2,v_3\}$	$\{v_2,v_4\}$	$\{v_3,v_4\}$	$\{v_2,v_3,v_4\}$
1	-	-	-	-	-	-	-	۲۱
2	1	-	∞	10	-	-	20	
3	∞	8	-	14	-	12	-	
4	6	4	∞	-	∞	-	-	

حال برای اینکه یک تور بهینه را بدست آوریم از ماتریس P استفاده می کنیم. ابعاد این ماتریس به صورت ماتریس D است و در هر خانه ماتریس P متناظر با خانه پر شده در ماتریس D ، مقدار j ای که به ازای آن مقدار حداقل بدست آمد، قرار داده می شود. به طور نمونه عدد 21 به ازای v_3 حاصل شد، بنابراین در خانه متناظر آن در p عدد 3 را قرار می دهیم. یا عدد 12 در ماتریس D به ازای v_4 حاصل شد، بنابراین عدد 4 را در خانه متناظر آن در ماتریس P قرار می دهیم:

1								۳
2				4			3	
3		2		4		4		
4		2						

نحوه استفاده از این ماتریس برای تعیین تور بهینه:

ابتدا به آخرین خانه پر شده نگاه می کنیم:

$$P[1][\{v_2, v_3, v_4\}] = 3$$

چون مقدار آن 3 است، یعنی ابتدا از v_1 به v_3 باید رفت. حال برای این که مشخص شود بعد از v_3 به کدام گره باید رفت، باید به خانه $P[3][\{v_2, v_4\}]$ نگاه کنیم. چون محتوای این خانه برابر 4 است ، باید v_3 از به v_4 رفت. در نهایت هم با توجه به اینکه $P[4][\{v_2\}] = 2$ باید به گره v_2 رفت. بنابراین تور بهینه عبارت است از :

$$[v_1, v_3, v_4, v_2, v_1]$$

تحلیل الگوریتم فروشنده دوره گرد در حالت معمول

به ازای هر مجموعه A حاوی k راس، باید n-1-k رای در نظر گرفته شود و به ازای هر یک از این رئوس، عمل اصلی k مرتبه اجرا می شود. چون تعداد زیر مجموعه های A از حاوی k راس، برابر با است، تعداد کل دفعاتی که عمل اصلی انجام می شود، عبارت است از:

$$T(n) = \sum_{k=1}^{n-2} (n-1-k) \times k \times \binom{n-1}{k}$$

که این مقدار برابر است با:

$$T(n) = (n-1)(n-2)2^{(n-3)} \in \theta(n^2 2^n)$$

تذکر: استفاده از این الگوریتم وقتی کارایی دارد که n کوچک باشد. با فرض این که پردازش دستور اصلی در الگوریتم، 1 میکرو ثانیه زمان ببرد و n=20 باشد، این الگوریتم به 45 ثانیه زمان نیاز دارد تا تور بهینه را مشخص کند:

$$(20-1)(20-2)2^{(20-3)}\mu s = 45s$$

اگر n=60 باشد، این الگوریتم به زمان خیلی زیادی نیاز دارد.

پیچیدگی حافظه ای برای الگوریتم فروشنده دوره گرد برابر است با:

$$M(n) = 2 \times n2^{n-1} = n2^n \quad \in \theta(n2^n)$$

۱- در یک راهرو **n** تابلو پشت سرهم برای نصب پوستر آماده شده است(تابلوهای b_1 تا b_n). طبق مقررات، یک پوستر نباید در دو تابلوی پشت سرهم و در یک تابلو نباید بیش از یک عدد از یک پوستر نصب شود. برای هر تابلوی یک " ضریب دید" w_i تعیین شده که نشان دهنده ی میزان دید آن تابلو است(هر چه عدد بزرگ تر باشد به این معنی است که پوستر این تابلو بیش تر از بقیه دیده می شود)(بدیهی است که پوسترها نباید هم دیگر را بپوشانند. اما همیشه جا برای نصب پوستر در هر تابلو هست). با داشتن **W** ها برای همه ی تابلوها، می خواهیم یک پوستر را در تعدادی از این تابلوها نصب کنیم که مجموع ضریب دید آن بیشینه شود. می توان نشان داد که مسئله راه حل پویا دارد. برای این کار فرض کنید **f(i)** مجموع ضریب دید(وزن) تابلوهای با شماره ۱ تا **i** است که بر روی آنها پوستر نصب می شود. در آن صورت کدام یک از رابطه های بازگشتی زیر درست است؟ (بدیهی است که $f(0)=0 , f(1)=W_1$)

1) $f(i)=\max\{f(i-1), w_i + f(i-2)\}$
2) $f(i)=\max\{f(i-1), w_1 + f(i-2)\}$
3) $f(i)=\max\{f(i-2), w_1 + f(i-1)\}$
4) $f(i)=\max\{f(i-1) + w_1, f(i-2)\}$

۲- برای سه ماتریس $N_1(m\times n)$، $N_2(n\times p)$ و $N_3(p\times q)$، اگر بخواهیم تعداد ضربهای $N_1\times(N_2\times N_3)$ با $(N_1\times N_2)\times N_3$ یکسان باشد، باید:

1) فقط $m=n=p=q$ باشد.
2) $m=n$ یا $p=q$
3) $\frac{1}{m}+\frac{1}{n}=\frac{1}{p}+\frac{1}{q}$
4) $\frac{1}{m}-\frac{1}{n}=\frac{1}{q}-\frac{1}{p}$

۳- الگوریتم زیر هزینه ضرب بهینه **n** تا ماتریس $M_1\times M_2\times\ldots\times M_n$ را که ابعاد M_i برابر $d_{i-1}\times d_i$ است را مشخص می کند. مجموع تعداد خواندن درایه های **C** در این الگوریتم چندتاست؟

```
for i:=۱ to n do
   c[i,j]=۰
for t:=۲ to n do
  for i:=۱ to n-t+۱ do
   {
      j:=i+t-۱;
      c[i,j]=min(c[i,k] + c[k+۱,j]+ d_{i-1} * d_k * d_j )
          i < k ≤ j
   }
```

1) $\sum_{t=1}^{n}(2t-1)(n-t)$
2) $\sum_{t=2}^{n}(2t-1)(n-t)$
3) $\sum_{t=2}^{n}2(t-1)(n-t+1)$
4) $\sum_{t=2}^{n}2(t-1)(n-t)$

برای محاسبه حاصلضرب آنهاست؟

1) $(A\times B)\times(C\times D)$ 2) $(A\times(B\times C))\times D$

3) $A\times((B\times C)\times D)$ 4) $((A\times B)\times C)\times D$

۵- اگر $A_{13\times5}$ و $B_{5\times89}$ و $C_{89\times3}$ و $D_{3\times34}$ باشد، حداقل تعداد عمل ضرب لازم برای محاسبه ABCD چقدر است؟

1) 2586 2) 4055 3) 2856 4) 5420

۶- فرمول بهینه سازی بازگشتی زیر برای کدام مسئله می باشد و بهترین روش نوشتن الگوریتم آن کدام است؟

$$M[i][j]=\min_{i\le k\le j-1} imum(M[i][k]+M[k+1][j]+d_{i-1}d_kd_j) \qquad i<j \;\; M[i][j]=0 \quad \text{if} \quad i=j$$

1) مسئله ضرب دو ماتریس Dynamic programming

2) مسئله پرانتز گذاری ضرب n ماتریس Divide and conquer

3) مسئله بهینه سازی درخت جستجوی باینری Dynamic programming

4) مسئله پرانتز گذاری ضرب n ماتریس Dynamic programming

۷- فرض کنید T(n) برابر تعداد پرانتزبندی های مختلف برای ضرب n ماتریس در هم باشد. در این صورت T(۱)=T(۲)=۱ و

1) $T(n)=\sum_{i=1}^{n} T(i)\times T(n-i)$ 2) $T(n)=\sum_{i=1}^{n-1} T(i)T(n-i)$

3) $T(n)=\sum_{i=1}^{n} T(i)T(n-i+1)$ 4) $T(n)=\sum_{i=1}^{n-1} T(i)\times T(n-i+1)$

۸- در ضرب سه ماتریس زیر، تحت چه شرایطی ضرب $(N_1N_2)N_3$ سریعتر از ضرب $N_1(N_2N_3)$ است؟

ماتریس	ابعاد
N_1	$w\times x$
N_2	$x\times y$
N_3	$y\times z$

1) $x>y$ 2) $\frac{1}{x}+\frac{1}{z}<\frac{1}{w}+\frac{1}{y}$

3) $\frac{1}{w}+\frac{1}{x}<\frac{1}{y}+\frac{1}{z}$ 4) $w+x<y+z$

```
function   fibo( n : integer): integer;
  var  f , f۱, f۲ , i : integer;
begin
   f۱:=۱;
   f۲:=۱;
   for  i:=۱  to n  do
     begin
        f:=f۱+f۲;
        f۱:=f۲;
        f۲:=f;
     end;
   fibo:=f;
end;
```

1) الگوریتم این برنامه از رده برنامه ریزی پویا و مرتبه آن خطی است.

2) الگوریتم این برنامه از رده تقسیم و غلبه و مرتبه آن خطی است.

3) الگوریتم این برنامه از رده برنامه ریزی پویا و مرتبه آن بیش از خطی است.

4) الگوریتم این برنامه از رده تقسیم و غلبه و مرتبه آن بیش از خطی است.

۱۰- الگوریتم زیر $\binom{n}{m}$ را برای اعداد صحیح $n \geq m$ محاسبه می‌کند، عمل + چندبار اجرا می‌شود؟

```
function   c (n,m : integer) : integer;
begin
     if ( m=n) or (m=۰)  then
        c:=۱
     else
        c :=c (n-۱,m) + c (n-۱,m-۱)
end;
```

1) $\binom{n}{m}$ 2) mn 3) $\binom{n}{m}-1$ 4) $n(n-m)$

۱۱- الگوریتم فلوید کوتاهترین مسیر بین همه زوج نقطه ها را در گراف جهت دار و وزن دار G محاسبه می کنـد. اگـر یال های با وزن منفی هم داشته باشیم، کدام یک از گزینه های زیر بهترین توصیف این الگوریتم است؟

1) با این الگوریتم می توان وجود یا عدم وجود دور منفی را تشخیص داد.

2) الگوریتم برای یال های منفی، ولی بدون دور منفی، ممکن است در دور بیفتد.

3) الگوریتم برای یال های منفی، ولی بدون دور منفی، متوقف می شود ولی درست کار می کند.

4) الگوریتم با یال های منفی ولی بدون دور منفی، متوقف می شود ولی جواب آن همیشه غلط است.

ترین مسیر ساده برای گراف بدون وزن و جهت دار G را در نظر می گیریم. برای حل این مسئله:

1) می توان از روش برنامه ریزی پویا استفاده کرد چون اصل بهینگی برقرار است.

2) با وجودی که اصل بهینگی برقرار است نمی توان از روش برنامه ریزی پویا استفاده کرد.

3) نمی توان از روش برنامه ریزی پویا استفاده کرد چون اصل بهینگی برقرار نیست.

4) با وجودی که اصل بهینگی برقرار نیست می توان از روش برنامه ریزی پویا استفاده کرد.

۱۳- در الگوریتم فلوید برای پیدا کردن طول کوتاهترین مسیرها بین هر جفت گره از یک گراف جهت دار n گره ای از روش پویا استفاده می شود و در هر خانه [i,j] از ماتریسی به نام D طول کوتاه ترین مسیرها بین گره های j تا i ثبت می شود. همچنین ماتریسی به نام P تشکیل می شود. در این ماتریس مقدار P[i,j] برابر گره ای است که در مسیر بهینه از i به j باید از آن عبور کرد. فرض کنیم که برای گرافی با ۴ گره ماتریس P به شکل زیر باشد. کدام یک از گزینه های زیر مسیر بهینه برای رفتن از گره ۱ به گره ۳ را مشخص می کند؟

$$P = \begin{bmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

1) 1 4 3 2) 1 2 3 3) 1 4 2 3 4) 1 2 4 3

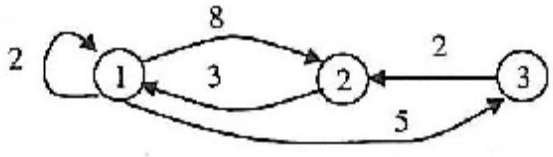
۱۴- می خواهیم برای ماتریس های $M1_{(10\times20)}$ و $M2_{(20\times50)}$ و $M3_{(50\times1)}$ و $M4_{(1\times100)}$ ترکیب بهینه پرانتزبندی پیدا نمائیم تا تعداد ضرب های کل جهت محاسبه عبارت ذیل حداقل گردد. این ترکیب بهینه عبارت است از:

$$M = M_1 \times M_2 \times M_3 \times M_4$$

1) $(M_1 \times (M_2 \times M_3)) \times M_4$ 2) $((M_1 \times M_2) \times M_3) \times M_4$

3) $M_1 \times ((M_2 \times M_3) \times M_4$ 4) هیچکدام

۱۵- با استفاده از الگوریتم فلوید کوتاهترین مسیر بین گره های ذیل عبارتند از:



1

	1	2	3
1	0	5	7
2	3	0	8
3	5	10	0

2

	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

3

	1	2	3
1	2	7	5
2	3	0	8
3	5	2	0

4

	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0

	V1	V2	V3	V4	V5
V1	0	14	4	10	20
V2	14	0	7	8	7
V3	4	5	0	7	16
V4	11	7	9	0	2
V5	18	7	17	4	0

1) 30 2) 31 3) 43 4) 34

۱۷- کدامیک از عبارات صحیح است؟

1) برای حل هر مساله می توان یک الگوریتم با استفاده از روش برنامه ریزی پویا طراحی کرد.

2) روش حریصانه همیشه یک راه حل بهینه را بدست می دهد.

3) روش تقسیم و غلبه یک روش بالا به پایین است در صورتیکه روش برنامه ریزی پویا یک روش پایین به بالا می باشد.

4) برای اینکه روش تقسیم و غلبه برای یک مساله مورد استفاده قرار گیرد اصل بهینگی باید برقرار باشد.

پاسخ تشریحی

1-1) اگر دو تابلو داشته باشیم آنگاه آن را یا می توان در تابلوی اول نصب کرد یا در تابلوی دوم. (طبق صورت تست، یک پوستر را نباید در دو تابلوی پشت سر هم نصب کرد). بنابراین f(2) برابر است با حداکثر مقدار بین w_1 و w_2 . اگر در گزینه ها به i مقدار 2 بدهیم، داریم:

1) f(2)=max{f(1) , w2+f(0)}=max{w1 , w2+0}=max{w1,w2}

2) f(2)=max{f(1) , w1+f(0)}=max{w1 , w1+w1}=max{w1،2w1}

3) f(2)=max{f(0) , w1+f(1)}=max{0 , w1+w1}=max{0،2w1}

4) f(2)=max{f(1)+w1 , f(0)}=max{w1+w1 , 0}=max{2w1،0}

بنابراین گزینه 1 درست است.

تذکر: دقت کنید که گزینه های 3 و4 یکسان هستند.

روش دوم: دو حالت وجود دارد که بین آنها باید بیشترین مقدار در نظر گرفته شود:

الف- پوستر در خانه i ام نصب شود. در این حالت زاویه دید آن یعنی w_i به علاوه f(i-2) باید در نظر گرفته شود (چون یک پوستر را نباید در دو تابلوی پشت سرهم نصب کرد)

ب- پوستر در خانه i ام نصب نشود. در این حالت در خانه قبلی نصب شده است و f(i-1) را در نظر می گیریم.

بنابراین داریم: $f(i) = \max\{f(i-1), w_i + f(i-2)\}$

2-4) تعداد ضربهای $N_1\times(N_2\times N_3)$ برابر npq+mnq و تعداد ضربهای $(N_1\times N_2)\times N_3$ برابر mnp+mpq می باشد و داریم :

$$npq + mnq = mnp + mpq$$

با تقسیم طرفین رابطه به mnpq داریم : $\frac{1}{m}+\frac{1}{p}=\frac{1}{q}+\frac{1}{n}$ که می توان آن را به صورت مقابل نیز نوشت :

$$\frac{1}{m}-\frac{1}{n}=\frac{1}{q}-\frac{1}{p}$$

3-3) سه حلقهٔ تودرتو داریم. به ازای مقادیر معلوم t و i، تعداد گذرها از حلقهٔ k (حلقه دستور min با شمارنده $i < k \le j$) عبارت است از: $j - i$ و چون $j = i + t - 1$ است، این رابطه برابر است با: $i + t - 1 - i = t-1$

تعداد گذرها از حلقهٔ for i برای n-t+1 است. چون t از دو تا n تغییر می‌کند، تعداد کل دفعاتی که عمل اصلی(تعداد خواندن های درایه های C) انجام می‌شود، عبارت است از:

$$\sum_{t=2}^{n} 2(t-1)(n-t+1)$$

تذکر: عدد 2 در فرمول بالا به این علت است که 2 بار c خوانده می شود، یکی c[i,k] و یکی c[k+1,j].

3-4) در ضرب چند ماتریس در یکدیگر باید ضربها به نحوی انجام شود که تعداد آنها حداقل باشد. بنابراین ابتدا دو ماتریسی را

ماتریس را در ماتریس D ضرب کرده و در نهایت ماتریس A را در ماتریسBCD ضرب می کنیم.

3-5) دو ماتریس را با هم طوری ضرب می‌کنیم که بعد بزرگتری حذف شود. مثلاً با ضرب A و B یک ماتریس 13×89 بدست می‌آید که بعد 5 حذف شده ولی با ضربB وC یک ماتریس 5×3 بدست می‌آید که بعد 89 حذف شده است. بنابراین ابتدا B وC را در هم ضرب می‌کنیم. نحوه ضرب به صورت زیر است:

$$A_{13\times5}(BC)_{5\times3}D_{3\times34} => (ABC)_{13\times3}D_{3\times34} => (ABCD)_{13\times34}$$

نحوه محاسبه تعداد ضربها : برای ضرب دو ماتریس $m\times n$ و $n\times p$ نیاز به $m\times n\times p$ ضرب داریم. بنابراین تعداد ضربهای مورد نیاز عبارتند از: الف- ضرب B و C : $5\times89\times3$ ب- ضرب A در BC: $13\times5\times3$ ج- ضرب ABC در D: $13\times3\times34$

که در مجموع برابر است با :

$$5\times89\times3+13\times5\times3+13\times3\times34=2856$$

4-6) فرمول داده شده برای مسئله پرانتز گذاری ضرب n ماتریس می باشد. زمان الگوریتم تقسیم و حل، بر اساس این ویژگی، نمایی است. با استفاده از برنامه‌نویسی پویا برای محاسبه مقادیر M[i][j] الگوریتمی با کارایی بیشتر طراحی می‌کنیم و از شبکه‌ای مشابه مثلث خیام استفاده خواهیم کرد.

7-2) تعداد حالات مختلف ضرب n ماتریس از سری کاتالان قابل محاسبه است.

$$T(n)=\sum_{i=1}^{n-1}T(i)T(n-i)$$

به طور نمونه حالات مختلف ضرب 4 ماتریس برابر است با:

$$T(4)=\sum_{i=1}^{3}T(i)T(4-i)$$

$$T(4)=T(1).T(3)+T(2).T(2)+T(3).T(1)$$

جمله اول $A\times(B\times C\times D)$ و جمله دوم $(A\times B)\times(C\times D)$ و جمله سوم $(A\times B\times C)\times(D)$ را نشان می دهد.

8-2) تعداد ضربهای مورد نیاز برای محاسبه $(N_1N_2)N_3$ برابر wxy+wyz و تعداد ضربهای مورد نیاز برای محاسبه $N_1(N_2N_3)$ برابر xyz+wxz می باشد و چون در صورت مسئله گفته حالت اول از حالت دوم سریعتر باشد، داریم:

$$wxy+wyz < xyz+wxz$$

حال برای اینکه به فرم یکی از گزینه ها در بی آید، طرفین را بر wxyz تقسیم می کنیم: $\frac{1}{z}+\frac{1}{x}<\frac{1}{w}+\frac{1}{y}$

9-1) الگوریتم داده شده به روش پویا نوشته شده و مرتبه آن خطی است.

10-3) برای محاسبه ترکیب $\binom{n}{m}$ ، نیاز به $\binom{n}{m}-1$عمل جمع می‌باشد. به طور مثال برای محاسبه مقدار $\binom{3}{2}$ (که حاصل آن برابر 3 است) نیاز به 2 عمل جمع است.

$$C(3,2)=C(2,2)+C(2,1)$$

$$C(2,1)=C(1,1)+C(1,0)$$

3-12) اصل بهینگی در مسئله طولانی ترین مسیر برقرار نمی باشد، بنابراین نمی توان از روش برنامه ریزی پویا استفاده کرد.

4-13) برای رفتن از گره 1 به گره 3 به ماتریس P نگاه کرده، مشخص است که P[1][3]=4 ، یعنی برای رفتن از گره 1 به 3 باید از گره 4 گذشت. با توجه به اینکه P[1][4]=2 ، برای رفتن از گره 1 به 4 باید از 2 عبور کرد. بنابراین مسیر به صورت زیر است:

1-14) دو ماتریس را با هم طوری ضرب می‌کنیم که بعد بزرگتری حذف شود. بنابراین ابتدا دو ماتریس M2 و M3 را در هم ضرب کرده و سپس ماتریس M1 را در ماتریس بدست آمده ضرب می کنیم و در نهایت ماتریس حاصل را در ماتریس M4 ضرب می کنیم.

2-15) کوتاهترین مسیر بین گره های گراف داده شده برابر است با:

	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

به طور نمونه کوتاهترین مسیر از گره 1 به خودش برابر 0 است. یا کوتاهترین مسیر از گره 1 به گره 2 برابر 7 است (از گره 3 به عنوان واسط عبور می کنیم)

2-16) تور بهینه به صورت زیر می باشد:

$$v1 \xrightarrow{4} v3 \xrightarrow{5} v2 \xrightarrow{7} v5 \xrightarrow{4} v4 \xrightarrow{11} v1$$

بنابراین طول مسیر برابر است با:

4+5+7+4+11=31

3-17) روش تقسیم و غلبه یک روش بالا به پایین است در صورتیکه روش برنامه ریزی پویا یک روش پایین به بالا می باشد.

فصل چهارم : روش حریصانه

الگوریتم حریصانه با انجام یک سری انتخاب، که در جای خود بهینه است، عمل می کند. امید این است که یک حل بهینه سرتاسری یافت شود، ولی همواره چنین نیست. برای یک الگوریتم مفروض باید تعین کرد که آیا حل همواره بهینه است یا خیر. واژه "حریصانه" غالباً برای حل مسائل بهینه سازی بکار می روند. در روش حریصانه، تقسیم به نمونه های کوچکتر صورت نمی پذیرد.

مسئله خرد کردن پول

یکی از مثال هایی که موضوع حریصانه را می توان به کمک آن روشن کرد، مسئله دادن بقیه پول می باشد. فروشنده یک فروشگاه غالباً برای دادن بقیه پول به خریدار دچار مشکل می شود. مشتریان معمولاً مایل نیستند مقداری زیادی پول خرد بگیرند. بنابراین، هدف وی نه تنها دادن بقیه پول به میزان صحیح، بلکه انجام این کار با حداقل تعداد سکه ممکن است.

راه حل: در آغاز فروشنده بزرگترین سکه از لحاظ ارزش را پیدا می کند، یعنی ملاک وی ارزش سکه است این را در الگوریتم حریصانه، روال انتخاب می نامند. سپس باید ببیند که آیا با افزودن این سکه به بقیه پول، جمع کل آنها از چیزی که باید باشد بیشتر می شود یا خیر. این بررسی را امکان سنجی (feasibility check) می نامند. اگر با افزودن این سکه، بقیه پول از میزان لازم بیشتر نشود، این سکه به مجموعه افزوده می شود سپس تحقیق می کند تا ببیند که آیا مقدار بقیه پول با میزان لازم برابر شده است یا خیر. این موضوع را بررسی راه حل (solution check) می گویند. اگر برابر نبودند، با استفاده از روال انتخاب یک سکه دیگر انتخاب می کند و فرایند تکرار می شود او چندین بار این کار را انجام می دهد که مقدار بقیه پول با میزان لازم برابر شود یا اینکه دیگر سکه ای برایش باقی نماند که در این حالت قادر به بازگرداندن مقدار بقیه پول نیست.

این الگوریتم را حریصانه می نامیم، زیرا روال انتخاب صرفاً شامل گرفتن حریصانه بزرگترین سکه بعدی، بدون اندیشیدن به عواقب چنین انتخابی است. هنگامی که سکه ای پذیرفته شد، بطور دائمی در حل گنجانده می شود. هنگامی که سکه ای رد شد بطور دائمی از حل مطرود می گردد.

این روال آیا به حل بهینه منجر می شود؟ یعنی در این مسئله هنگامی که حل آن امکان پذیر است، آیا حل فراهم آمده توسط الگوریتم، حاوی حداقل تعداد سکه های مورد نیاز برای دادن بقیه پول است یا خیر؟

مثال: اگر سکه های آمریکایی (یک سنتی، پنج سنتی، ده سنتی، بیست و پنج سنتی و نیم دلاری) باشند و اگر از هرکدام حداقل یک عدد موجود باشد، الگوریتم حریصانه همواره در صورت وجود حل بهینه را برمی گرداند.

به طور نمونه برای برگرداندن 36 سنت به مشتری، مراحل زیر انجام می شود:

1- دادن یک سکه 25 سنتی به مشتری

2- دادن یک سکه 10 سنتی

3- دادن یک سکه 10 سنتی دیگر (ولی چون از 36 سنت بیشتر می شود، این سکه پس گرفته می شود.)

4- دادن یک سکه 5 سنتی (ولی چون از 36 سنت بیشتر می شود، این سکه پس گرفته می شود.)

5- دادن یک سکه 1 سنتی

بنابراین در کل یک سکه 25 سنتی، یک سکه 10 سنتی و یک سکه 1 سنتی به مشتری داده شد.

مثال: اگر یک سکه 12 سنتی را جزء سکه های آمریکایی در نظر بگیریم، الگوریتم حریصانه همواره به حل بهینه نمی انجامد. به طور نمونه برای برگرداندن 16 سنت به مشتری، مراحل زیر انجام می شود:

1- دادن یک سکه 12 سنتی به مشتری

3- دادن یک سکه 5 سنتی (ولی چون از 16 سنت بیشتر می شود، این سکه پس گرفته می شود.)

4- دادن چهار سکه 1 سنتی

حل حریصانه حاوی پنج سکه می شود، ولی حل بهینه (یک سکه ده سنتی، پنج سنتی و یک سنتی) فقط سه سکه است.

الگوریتم حریصانه، حل بهینه را تضمین نمی کند.

نحوه کار الگوریتم حریصانه

الگوریتم حریصانه، کار را با یک مجموعه تهی آغاز کرده به ترتیب عناصری به مجموعه اضافه می کند تا این مجموعه حلی برای نمونه ای از یک مسئله را نشان دهد. هر دو تکرار شامل مولفه های زیر است:

روال انتخاب : عنصر بعدی را که باید به مجموعه اضافه شود، انتخاب می کند. انتخاب طبق یک ملاک حریصانه اجرا می شود که یک شرط بهینه را در همان برهه برآورده می سازد.

بررسی امکان سنجی: تعیین می کند که آیا مجموعه جدید برای رسیدن به حل، عملی است یا خیر.

بررسی راه حل: تعیین می کند که آیا مجموعه جدید، حل نمونه را ارائه می کند یا خیر.

در این فصل، ابتدا مسائلی را مورد بحث قرار خواهیم داد که الگوریتم حریصانه همواره برای آنها منجر به حل بهینه می شود. سپس مسائلی را مورد بحث قرار می دهیم که الگوریتم حریصانه مناسب آن نیست. در همان بخش روش حریصانه را با برنامه نویسی پویا مقایسه می کنیم تا روشن شود چه زمانی،کدام الگوریتم کاربرد دارد. الگوریتم هایی که بررسی می کنیم عبارتند از:

1- پریم و کروسکال 2- دیکسترا 3- زمان بندی 4- کد هافمن 5- کوله پشتی

درخت های پوشای کمینه

درخت پوشا برای گراف G یک زیر گراف متصل است که حاوی همه رئوس موجود در G بوده و یک درخت باشد یعنی چرخه نداشته باشد. درخت پوشا با وزن کمینه را درخت پوشای کمینه می گویند.

فرض کنید بخواهید چند شهر معین را با جاده به هم وصل کنید، به قسمی که بتوان از هر شهر به شهر دیگر رفت. می خواهیم این کار را با حداقل مقدار جاده کشی انجام دهیم. حال الگوریتمی بسط می دهیم که این مسئله و مسائل مشابه را حل کند. مسئله حذف یال هایی از یک گراف بدون جهت، موزون و متصل است تا زیر گرافی تشکیل شود که همه رئوس متصل باقی بمانند و حاصل جمع اوزان یالهای باقیمانده،کمینه باشد.

تذکر: یک زیر گراف با حداقل وزن باید یک درخت باشد، زیرا اگر زیر گرافی درخت نباشد، حاوی چرخه خواهد بود و می توانیم با حذف یک یال از چرخه ،گراف متصلی با وزن کمتر به دست آوریم.

تذکر: یک زیر گراف متصل با وزن کمینه باید یک درخت پوشا باشد، ولی هر درخت پوشا دارای وزن کمینه نیست.

هر گراف می تواند بیش از یک درخت پوشای کمینه داشته باشد.

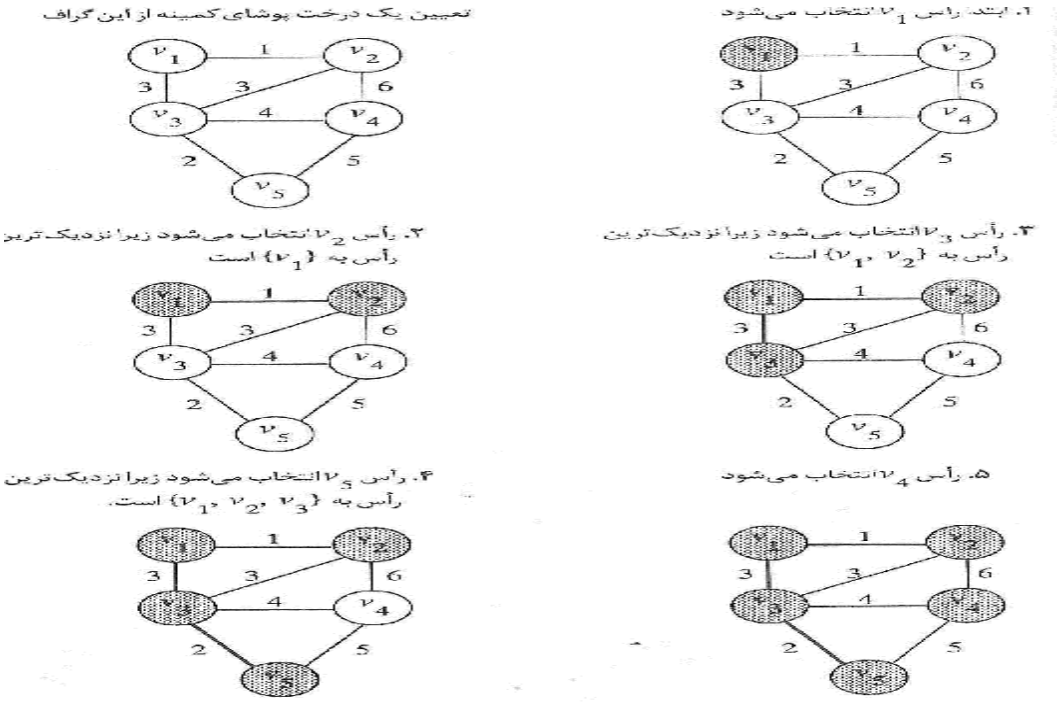
یافتن درخت پوشای کمینه که همه درخت های پوشا را در نظر داشته باشد در بدترین حالت، از حالت نمایی بدتر است.

پوشا (F) زیر مجموعه E است. درخت پوشا را با T=(V,F) نشان می دهیم. مسئله، یافتن زیر مجموعه F از E است به طوری که T یک درخت پوشای کمینه برای G باشد.

برای به دست آوردن درخت پوشای کمینه، دو الگوریتم حریصانه متفاوت به نام های پریم و کروسکال را بررسی می کنیم. هریک از این الگوریتم ها از یک ویژگی بهینه محلی استفاده می کند. از آنجا که تضمینی وجود ندارد که یک الگوریتم حریصانه مفروض همواره حل بهینه را بدهد، ثابت می کنیم که الگوریتم های کروسکال و پریم همواره درخت های پوشای کمینه را ایجاد می کنند.

الگوریتم پریم

الگوریتم پریم با زیر مجموعه ای از یال های F و زیر مجموعه ای از رئوس Y آغاز می شود، زیرمجموعه Y حاوی یک رأس دلخواه است. به عنوان مقدار اولیه،{V1}را به Y می دهیم. نزدیکترین رأس به Y، رأسی در V-Y است که توسط یالی با وزن کمینه به رأسی در Y متصل است. رأسی که از همه به Y نزدیک تر است، به Y و یال آن به F افزوده می شود. اتصال ها به طور دلخواه شکسته می شوند. در این مورد V2 به Yو (V1,V2) به F افزوده می شود. این فرآیند افزودن نزدیکترین رئوس، چندین بار تکرار می شود تا Y=V بشود. روال انتخاب و بررسی امکان سنجی به همراه یکدیگر اجرا می شوند زیرا رأس جدیدی از V-Y تضمین می کند که چرخه ای ایجاد نگردد. شکل زیر الگوریتم پریم را نشان می دهد.



در پیاده سازی گراف را به کمک ماتریس همجواری که در زیر آورده شده است، نمایش می دهیم:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{bmatrix} 0 & 1 & 3 & \infty & \infty \\ 1 & 0 & 3 & 6 & \infty \\ 3 & 3 & 0 & 4 & 2 \\ \infty & 6 & 4 & 0 & 5 \\ \infty & \infty & 2 & 5 & 0 \end{bmatrix}$$

همچنین در پیاده سازی الگوریتم به دو آرایه nearest و distance نیاز داریم. اندیس نزدیکترین راس در Y به v_i در nearest[i] ذخیره می شود و وزن یال میان v_i و راسی که توسط nearst[i] مشخص شده است، در distance[i] ذخیره می می شود. $(2 \le i \le n)$

چون در آغاز $Y=\{v1\}$ است، به nearest[i] مقدار اولیه یک داده می شود و به distance[i] مقدار اولیه ای مساوی وزن یال میان v1 وv2 داده می شود. به موازات افزوده شدن رئوس به Y، این دو آرایه چنان بهنگام می شوند تا به رأس جدیدی در Y که نزدیکترین راس به هر یک از رأس های خارج از Y است مراجعه کنند. برای که آن که تعیین کنیم کدام رأس باید به Y افزوده شود، در هر دو تکرار، اندیسی را محاسبه می‌کنیم که برای آن، distance[i] از همه کوچکتر است. این اندیس را vnear می نامیم. رأسی که اندیس آن vnear باشد، با نسبت دادن عدد -1 به distance[vnear] به Y افزوده می شود.

تذکر: الگوریتم پریم در پیوست آورده شده است.

با توجه به الگوریتم پریم، در حلقه repeat دو حلقه وجود دارد که هریک(n-1) بار تکرار می شود. اجرای دستورات داخل هر یک از آن ها را به عنوان یک بار اجرای عمل اصلی در نظر می گیریم. اندازه ورودی تعداد رئوس یعنی n است. چون حلقه repeat، به تعداد (n-1) بار تکرار می شود، پیچیدگی زمانی عبارت است از:

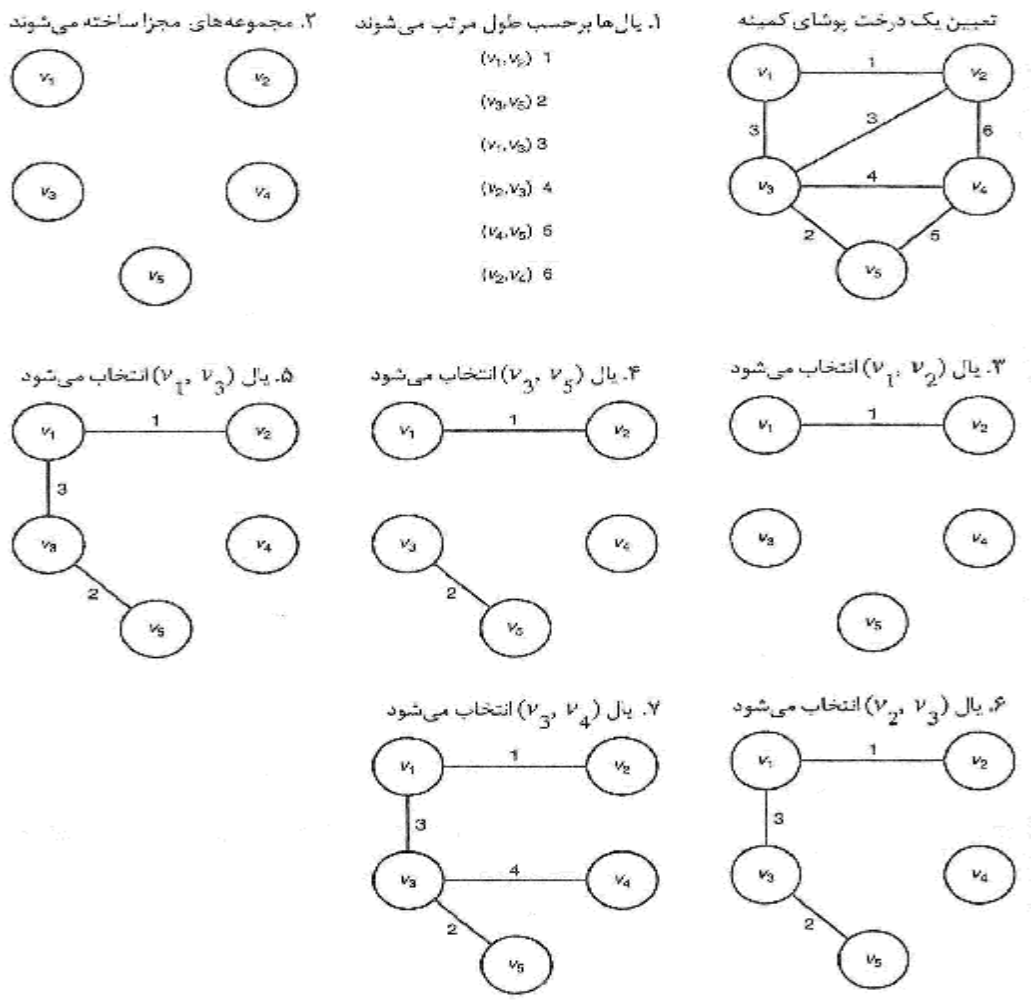
$$T(n) = 2(n-1)(n-1) \in \theta(n^2)$$

تذکر: گرچه الگوریتم حریصانه را غالباً آسان تر از الگوریتم برنامه نویسی پویا می توان نوشت، معمولاً تعیین اینکه آیا الگوریتم حریصانه همواره حل بهینه را ارائه می کند، دشوار است. برای یک الگوریتم برنامه نویسی پویا فقط کافی بود نشان دهیم اصل بهینگی برقرار است. برای یک الگوریتم حریصانه معمولاً به اثبات رسمی نیاز داریم.

قضیه : الگوریتم پریم همواره یک درخت پوشای کمینه تولید می کند.

الگوریتم کروسکال

الگوریتم کروسکال با ایجاد زیر مجموعه های مستقلی از V آغاز می شود که برای هر رأس یک زیر مجموعه در نظر گرفته می شود و هر زیر مجموعه حاوی فقط همان رأس است. سپس یال ها را طبق ترتیب غیر نزولی وزن وارسی می کند (اتصال ها به طور دلخواه شکسته می شوند). اگر یالی دو رأس را در مجموعه های غیر مستقل به هم متصل کند،آن یال اضافه می شود و دو زیر مجموعه در هم ادغام می شوند.



تذکر: الگوریتم کراسکال در پیوست آورده شده است.

با توجه به الگوریتم کراسکال، عمل اصلی، یک دستور مقایسه است و اندازه ورودی تعداد رئوس (n) و تعداد یال ها(m) می باشد. این الگوریتم از 3 قسمت تشکیل شده است:

1- مرتب سازی یالها از مرتبه $\theta(m \lg m)$.

2- مقداردهی n مجموعه مجزا از مرتبه $\theta(n)$.

3- دستکاری مجموعه متمایز در حلقه while از مرتبه $\theta(m \lg m)$.

چون $m \geq (n-1)$ است، مرتب سازی و دستکاری مجموعه های متمایز بر زمان مقدار دهی اولیه غلبه می کند، یعنی :

$$W(m,n) \in \theta(m \lg m)$$

در بدترین حالت، هر رأس را می توان به هریک از رئوس دیگر متصل کرد، یعنی:

$$m = \frac{n(n-1)}{2} \in \theta(n^2)$$

بنابراین به جای m از n^2 استفاده می کنیم و داریم:

$W(m,n) \in \theta(n^2 \lg n)$ بنابراین در بدترین حالت داریم:

- در یک گراف متصل رابطه $(n-1) \le m \le \frac{n(n-1)}{2}$ برقرار است.
- گراف بسیار متراکم : گرافی که تعداد یال های آن(m) نزدیک به $n-1$ باشد.
- گراف بسیار متصل: گرافی که تعداد یالهای آن نزدیک به $\frac{n(n-1)}{2}$ باشد.
- پیچیدگی زمانی در پریم برابر $\theta(n^2)$و در کروسکال برابر $\theta(m \lg m)$ است.
- مرتبه کروسکال در گراف بسیار متراکم، $\theta(n \lg n)$ودر گراف بسیار متصل، $\theta(n^2 \lg n)$ است.
- برای گراف بسیار متراکم، کروسکال و برای گراف بسیار متصل، پریم سریعتر است.

قضیه : الگوریتم کروسکال همواره یک درخت پوشای کمینه ایجاد می کند.

الگوریتم دیکسترا (کوتاه ترین مسیر تک مبدأ)

در فصل قبل، الگوریتمی به نام فلوید از مرتبه $q(n^3)$ بررسی شد که در آن کوتاهترین مسیر از هر راس به همه رئوس دیگر در یک گراف موزون مشخص می شد.اگر فقط بخواهیم کوتاه ترین مسیر از یک راس به بقیه راس های دیگر را مشخص کنیم، الگوریتم فلوید طاقت فرسا است. دیکسترا ، الگوریتمی از مرتبه $q(n^2)$ ارائه داد که از یک روش حریصانه استفاده می کند و به آن مسئله کوتاه ترین مسیر تک مبدا می گویند.

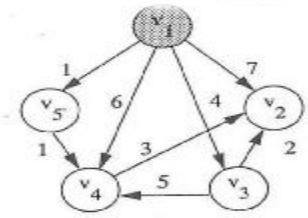
فرض می کنیم که از رأس مورد نظر به هر یک از رئوس دیگر، مسیری وجود دارد. در غیر این صورت با قدری اصلاح می توان مشکل را حل کرد. الگوریتم دیکسترا مشابه الگوریتم پریم است. برای مقدار دهی اولیه به مجموعه Y، رأسی را در آن قرار می دهیم که کوتاه ترین مسیرهای آن باید تعیین شود. برای تمرکز می گوییم آن رأس V1 است. به مجموعه F مقدار اولیه تهی می دهیم. نخست یک رأس را انتخاب می کنیم که از همه بهV1 نزدیکتر است و آن را بهY و<v1,v> را به F اضافه می کنیم. (منظور از<v1,v> یال میان v1وv است.) سپس مسیرهایی از v1 به رئوس موجود در V-Y را چک می کنیم که فقط رئوس موجود در Y را به عنوان رئوس واسطه مجاز می شمارند. یکی از مسیرها کوتاه ترین مسیر است. رأسی که در انتهای چنین مسیری باشد به Yو یالی (روی مسیری) که آن رأس را در بر می گیرد، به F افزوده می شود. این روال چندان ادامه می یابد که Y با مجموعه همه رئوس (V) برابر شود. در این نقطه F حاوی یال های موجود در کوتاه ترین مسیر است.

با توجه به تحلیل الگوریتم پریم، برای الگوریتم دیکسترا نتیجه می گیریم که :

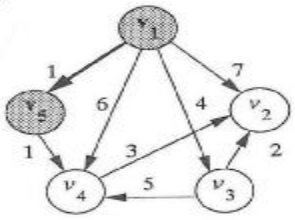
$$T(n) = 2(n-1)^2 \in \theta(n^2)$$

قضیه: الگوریتم دیکسترا همواره کوتاه ترین مسیر را ایجاد می کند.

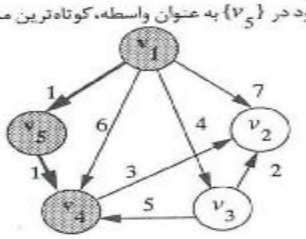
تذکر: الگوریتم پریم و دیکسترا را می توان با استفاده از یک heap یا heap فیبوناچی پیاده سازی کرد. پیاده سازی heap به $\theta(m \lg n)$ و پیاده سازی heap فیبوناچی به $\theta(m + n \lg n)$ تعلق دارد.



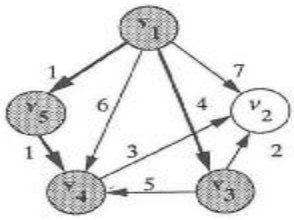
۱. رأس v_5 به این دلیل انتخاب می‌شود که نزدیک‌ترین رأس به v_1 است.



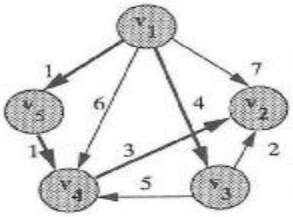
۲. رأس v_4 به این دلیل انتخاب می‌شود که با استفاده از رأس‌های موجود در $\{v_5\}$ به عنوان واسطه، کوتاه‌ترین مسیر را از v_1 دارد.



۳. رأس v_3 به این دلیل انتخاب می‌شود که با استفاده از رأس‌های موجود در $\{v_4, v_5\}$ به عنوان واسطه، کوتاه‌ترین مسیر را از v_1 دارد.



۴. کوتاه‌ترین مسیر از v_1 به v_2، $[v_1, v_5, v_4, v_2]$ است.



زمان بندی

مسائل زمان بندی را به دو گروه "زمان بندی ساده" و "زمان بندی با مهلت معین" تقسیم بندی می کنیم. ابتدا با ذکر یک مثال زمان بندی ساده را توضیح می دهیم.

مثال: فرض کنید زمان های خدمات برای سه کار به قرار $t_1 = 5, t_2 = 10, t_3 = 4$ است. (واحد هـای زمـانی واقعـی در مسـئله اهمیتی ندارند). اگر این کارها را به ترتیب 1، 2 و 3 زمان بندی کنیم، زمان های صرف شده در سیستم برای سه کار بـه ایـن قـرار است:

کار	زمان در سیستم	مجموع
1	5 (زمان ارائه خدمات)	5
2	5 (انتظار برای انجام کار 1) + 10 (زمان ارائه خدمات)	15
3	5 (انتظار برای انجام کار 1) + 10 (انتظار برای انجام کار 2) + 4 (زمان ارائه خدمات)	19

زمان کل در سیستم برای این زمان بندی عبارت است از: $5+15+19=39$

زمان بندی های ممکن:

$5+(5+10)+(5+10+4)=39$	$[1,\ 2,\ 3]$
$5+(5+4)+(5+4+10)=33$	$[1,\ 3,\ 2]$
$10+(10+5)+(10+5+4)=44$	$[2,\ 1,\ 3]$
$10+(10+4)+(10+4+5)=43$	$[2,\ 3,\ 1]$
$4+(4+5)+(4+5+10)=32$	$[3,\ 1,\ 2]$
$4+(4+10)+(4+10+5)=37$	$[3,\ 2,\ 1]$

با توجه به جدول بالا، زمان بندی $[3,\ 1,\ 2]$ با زمان کل 32 بهینه است.

تذکر: بنابراین یک زمان بندی بهینه هنگامی بدست می آید که کاری با کوچک ترین زمان ارائه خدمات (کار 3) اول از همه برنامه ریزی شود. سپس کاری که از لحاظ کوچکی زمان ارائه خدمات در مقام دوم قرار دارد (کار 1) قرار میگیرد و سرانجام نوبت به کاری می رسد که دارای بیشترین زمان ارائه خدمات است (کار 2)، بدیهی است که این زمان بندی از این رو بهینه به نظر می رسد که کوتاه ترین کارها را در آغاز برنامه قرار می دهد.

تذکر: زمان الگوریتمی که همه زمان بندی های ممکن را در نظر می گیرد، به صورت فاکتوریل است.

الگوریتم زمان بندی تنها کاری که انجام می دهد مرتب سازی کارها بر حسب زمان ارائه خدمات است. بنابراین پیچیدگی زمانی آن برابر است با : $W(n)\in q(n\ \lg\ n)$

قضیه : تنها زمان بندیی که زمان کل در سیستم را کمینه سازی می کند، زمان بندیی است که در آن کارها بر حسب افزایش زمان ارائه خدمات مرتب می شوند.

زمان بندی با مهلت معین

در این مسئله زمان بندی، هر کاری در یک واحد زمانی به پایان می رسد و دارای یک مهلت و سود معین است. اگر هر کار پیش از مهلت معین یا در آن مدت انجام شود، سود مورد نظر به دست می آید. هدف، زمان بندی کارها به نحوه ای است که سود بیشینه به دست آید. لازم نیست همه کارها زمان بندی شوند.

تذکر: نیازی نیست هر زمان بندیی را که در آن یک کار پس از مهلت مقرر آن برنامه ریزی شده است، در نظر بگیریم.

مثال: فرض کنید کارها، مهلت ها و سودهای زیر را داریم:

کار	مهلت	سود
1	2	30
2	1	35
3	2	25
4	1	40

زمان بندی ها و سودهای ممکن عبارت است از:

زمان بندی	سود کل
[1, 3]	30 + 25 = 55
[2, 1]	35 + 30 = 65
[2, 3]	35 + 25 = 60
[3, 1]	25 + 30 = 55
[4, 1]	40 + 30 = 70
[4, 3]	40 + 25 = 65

کاری که دارای مهلت 1 است، نمی تواند به عنوان کار دوم زمان بندی شود. همانطور که در جدول بالا مشاهده می کنید کارهای 2 و4 به عنوان کار دوم زمان بندی نشده اند. زمان بندی [1,3] یعنی کار1 پیش از مهلت خود و کار 3 در زمان مهلت خود آغاز می گردد.

زمان بندی [1,2] غیر ممکن است، زیرا کار 1 ابتدا در زمان 1 آغاز شده یک واحد زمانی لازم دارد تا به پایان برسد و بعد کار 2 در زمان 2 آغاز می شود. ولی مهلت برای کار 2، زمان 1 است. زمان بندی های غیر ممکن ذکر نشده اند. زمان بندی [4,1] با سود کل 70 بهینه است.

در این مثال، کاری که بیشترین سود را دارد (کار 4) در زمان بندی بهینه گنجانده شده است ولی کاری که از لحاظ سود دومین مقام را دارد (کار 2) در زمان بندی گنجانده نشده است. چون مهلت هر دو کار برابر با یک است، نمی توان هر دو را در زمان بندی گنجاند و آن که دارای سود بیشتری است گنجانده می شود. کار دیگری که در زمان بندی در نظر گرفته شده است، کار 1 است زیرا سود آن بیشتر از کار 3 است. این نشان می دهد که یک روش حریصانه منطقی برای حل این مسئله، عبارت از مرتب سازی غیر نزولی کارها بر اساس سود آن ها و سپس وارسی هر یک از کارها به ترتیب و در صورت امکان، افزودن آن به زمان بندی است.

■

تعریف: ترتیبی را ترتیب امکان پذیر می گویند که همه کارها در آن به ترتیب در مهلت مقرر خود آغاز شوند. در مثال قبل، [4,1] یک ترتیب امکان پذیر است ولی [1,4] ترتیب امکان پذیری نیست.

یک مجموعه از کارها را مجموعه امکان پذیر می گوییم اگر حداقل یک ترتیب امکان پذیر برای کارهای مجموعه موجود باشد. در مثال قبل، مجموعه {1,4} یک مجموعه امکان پذیر است زیرا ترتیب زمان بندی [4,1] امکان پذیر است، حال آنکه {2,4} یک مجموعه امکان پذیر نیست، زیرا هیچ ترتیب زمان بندی اجازه نمی دهد که هر دو کار در مهلت های خود آغاز شوند.

هدف ما یافتن یک ترتیب امکان پذیر با سود کل بیشینه است، چنین ترتیبی را ترتیب بهینه و مجموعه کارها در ترتیب را مجموعه بهینه کارها می نامند.

کار	مهلت	سود
1	3	40
2	1	35
3	1	30
4	3	25
5	1	20
6	3	15
7	2	10

1- S برابر با f قرار داده می شود.

2- S برابر با {1} قرار داده می شود چون ترتیب [1] امکان پذیر است.

3- S برابر با {1,2}قرار داده می شود چون ترتیب [2,1] امکان پذیر است.

4- {1,2,3}رد می شود زیرا هیچ ترتیب امکان پذیری برای این مجموعه وجود ندارد.

5- S برابر {1,2,4}قرار داده می شود چون ترتیب [2,1,4] امکان پذیر است.

6- {1,2,4,5}رد می شود زیرا هیچ ترتیب امکان پذیری برای این مجموعه وجود ندارد.

7- {1,2,4,6} رد می شود زیرا هیچ ترتیب امکان پذیری برای این مجموعه وجود ندارد.

8- {1,2,4,7}رد می شود زیرا هیچ ترتیب امکان پذیری برای این مجموعه وجود ندارد.

مقدار نهایی S، برابر با {1,2,4} است و یک ترتیب امکان پذیر برای این مجموعه [2,1,4] است. چون کارهـای 1 و 4 هـر دو دارای مهلت 3 هستند.

تذکر: می توان به جای [2,1,4] از ترتیب امکان پذیر [2,4,1] استفاده کرد.

فرض کنید S مجموعه ای از کارها باشد. در این صورت S امکان پذیر خواهد بود اگر و فقط اگر ترتیب حاصل از مرتب شـدن کارهای S بر اساس مهلت های غیر نزولی، امکان پذیر باشد.

مثال: در مثال قبل آیا مجموعه {1,2,4,7}، امکان پذیر است؟

حل: ما فقط باید امکان پذیر بودن ترتیب زیر را چک کنیم :

$$[2,\ 7,\ 1,\ 4]$$
$$\uparrow\ \uparrow\ \uparrow\ \uparrow$$
$$1\quad 2\quad 3\quad 3$$

مهلت هر کار در زیر آن ذکر شده است. چون کار 4 توسـط مهلـتش زمـان بنـدی نشـده اسـت(مهلـت آن 3 اسـت و نمـی توانـد چهارمین کار باشد)، این ترتیب امکان پذیر نیست و طبق لم این مجموعه امکان پذیر نیست.

هدف تعیین زمان بندی با سود کل بیشینه است، با این فرض که هر کاری دارای سود است و فقط وقتی قابل حصول است که آن کار در مهلت مقررش انجام شود. ورودی هاتعداد کارها (n) و آرایه ای از اعداد صحیح به نام deadline[1..n] که $deadline[i]$ مهلت مقرر برای کار $[i]$اُم است. این آرایه به ترتیب غیر نزولی و بر اساس سود مرتبط با هر کار مرتب سازی شده است. خروجی، یک ترتیب بهینه J برای کارها است.

```
void   schedule (int  n , const int   deadline[ ] , sequence_of_integer &   j ){
    index   i; sequence_of_integer   K;
    J =  [1];
    for (i = 2 ;  i <= n ; i++ ) {
        K = J  with  i  added according to nondecreasing values of deadline[i];
        if  ( K  is  feasible)
            J = K;
     }
}
```

تذکر: در الگوریتم فرض شده که کارها قبلاً بر اساس سودها به صورت غیر نزولی مرتب و بعد به الگوریتم تحویل داده شده اند. چون سودها فقط برای مرتب سازی کارها مورد نیاز هستند، به عنوان پارامتر الگوریتم ذکر نشده اند.

تحلیل پیچیدگی زمانی در بدترین حالت برای الگوریتم زمان بندی با مهلت معین

عمل اصلی در این الگوریتم، دستور مقایسه است. کارها در زمان مرتب شده و به الگوریتم تحویل داده می شوند. در هر دور تکرار حلقه، حداکثر (i-1) مقایسه برای اضافه کردن کار i ام به k انجام می شود و حداکثر برای بررسی امکان پذیر بودن K به i مقایسه نیاز است. بنابراین در بدترین حالت داریم:

$$\sum_{i=2}^{n}[(i-1)+i] = n^2 - 1 \in \theta(n^2)$$

و چون $\theta(n^2)$ به زمان مرتب سازی غلبه می کند، داریم:

$$W(n) \in \theta(n^2)$$

قضیه: الگوریتم زمان بندی با مهلت معین، همواره یک مجموعه بهینه ارائه می کند.

مثال: فرض کنید کارهای زیر داده شده اند:

کار	1	2	3	4	5	6	7
مهلت	3	1	1	3	1	3	2

الگوریتم این عملیات را انجام می دهد :

1. J مساوی با [1] قرار داده می شود.
2. K مساوی با [2،1] قرار داده شده مشخص می شود که امکان پذیر است.

3. K مساوی با [2،3،1] قرار داده شده رد می شود زیرا امکان پذیر نیست.
4. K مساوی با [2،1،4] قرار داده شده مشخص می شود که امکان پذیر است.

   J مساوی با [2،1،4] قرار داده می شود زیرا K امکان پذیر است.
5. K مساوی با [2،5،1،4] قرار داده شده رد می شود زیرا امکان پذیر نیست.
6. K مساوی با [2،1،6،4] قرار داده شده رد می شود زیرا امکان پذیر نیست.
7. K مساوی با [2،7،1،4] قرار داده شده رد می شود زیرا امکان پذیر نیست.

مقدار نهایی J برابر با [2،1،4] است.

مرتبه زمانی الگوریتم زمان بندی با مهلت معین(در بدترین حالت) برابر است با: $W(n) \in q(n^2)$

قضیه : الگوریتم زمان بندی با مهلت معین همواره یک مجموعه بهینه تولید می کند.

کد هافمن

روش متداول نمایش فایل، کد دودویی است که هر کاراکتر با رشته دودویی منحصر به فردی نمایش داده می شود. کد دودویی می تواند دارای طول ثابت یا متغیر باشد. در طول ثابت، طول کد تمام کاراکترها یکسان است.

مثال: فایلی با محتویات ababcbbbc را در نظر بگیرید. این فایل را به دو صورت طول ثابت و طول متغیر کد گذاری کنید.

الف- در کد گذاری با طول ثابت، به هر یک از کاراکترهای استفاده شده در فایل یک طول ثابت می دهیم. به طور نمونه :

a:00 , b:01 , c:11

ب- در کد گذاری با طول متغیر به کاراکتری که بیشتر از همه تکرار شده یعنی b کد 0 را اختصاص می دهیم. به این ترتیب، کد a نمی تواند 00 باشد، زیرا در این صورت نمی توانیم یک a را از دو b تشخیص دهیم. علاوه بر این، نمی خواهیم a را به صورت 01 کد گذاری کنیم، زیرا وقتی به یک مقدار 0 رسیدیم، نمی توانیم تشخیص دهیم که b است یا بخشی از کد مربوط به a است.

در نهایت این کاراکترها را به صورت زیر کد گذاری می کنیم: a : 10 , b : 0

, c : 11

فایل در روش اول به صورت 000100011101010111 و در روش دوم به صورت 1001001100011 کدگذاری می شود. که به ترتیب از 18 بیت و 13 بیت استفاده شد . ■

برای یک فایل معین، مسئله کد دودویی بهینه، یافتن کد دودویی برای کاراکترهای فایل است تا فایل را با حداقل تعداد بیت ها نمایش دهد.

کد پیشوندی

کد پیشوندی، نوع خاصی از کد با طول متغیر است. در کد پیشوندی، هیچ کد واژه مربوط به یک کاراکتر ، آغاز کد واژه کاراکتر دیگر را نمی سازد. به عنوان مثال، اگر 01 کد حرف ''a'' باشد، آن گاه 011 نمی تواند کد واژه حرف ''b'' باشد. در کد پیشوندی لازم نیست هنگام تجزیه فایل پیشگویی صورت گیرد. برای تجزیه فایل، از اولین بیت سمت چپ فایل و ریشه درخت شروع می

وقتی به برگ رسیدیم، کاراکتر را در برگ می یابیم. سپس به ریشه برمی گردیم و روال قبلی را با شروع از بیت بعدی موجود در دنباله بیت ها ادامه می دهیم.

مثال: مجموعه کاراکترهای $\{a,b,c,d,e,f\}$ را در نظر بگیرید. تعداد دفعاتی که هر یک از این کاراکترها در فایل ظاهر می شوند، عبارت است از:

a:16 , b: 5 , c: 12 , d: 17 , e: 10 , f: 25

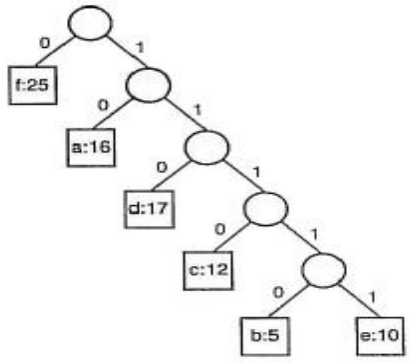
یک کد گذاری در زیر نشان داده شده است:

a=10 , b=11110 , c=1110 , d=110 , e=11111 , f=0

تعداد بیت ها برای این کد گذاری:

16(2)+5(5)+12(4)+17(3)+10(5)+25(1)=231

درخت دودویی معادل کد:



در صورتی که از کد گذاری با طول ثابت استفاده می کردیم، کد هر یک از کاراکترها 3 حرفی بود و تعداد بیت های لازم برابر بود با:

$$(16+5+12+17+10+25)\times 3 = 255$$

تذکر: تعداد بیت های لازم اگر از کد گذاری هافمن استفاده شود، از هر دو روش گفته شده بهتر است.■

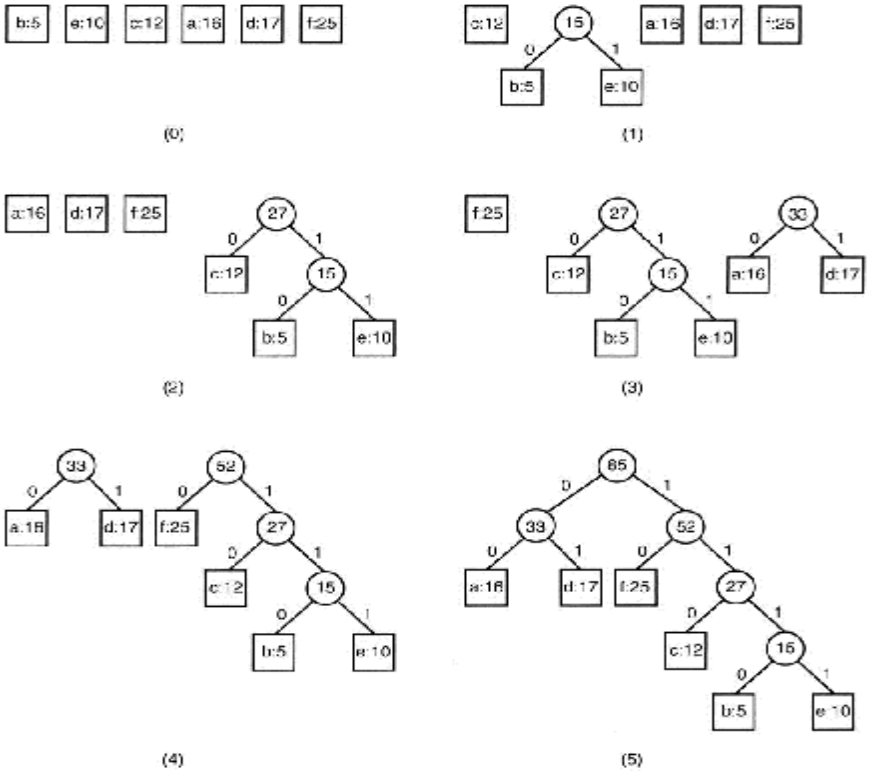
تعداد بیت های لازم برای کد گذاری فایلی که توسط درخت دودویی T برای کد خاصی ایجاد می شود، به صورت زیر تعیین می گردد : $bits(T) = \sum_{i=1}^{n} frequency(v_i)depth(v_i)$ ، که در آن $\{v_1, v_2, ..., v_n\}$ مجموعه کاراکترهای موجود در فایل، $frequency(v_i)$ تعداد دفعات حضور v_i در فایل، و $depth(v_i)$ عمق v_i در T است.

الگوریتم هافمن

هافمن یک الگوریتم حریصانه را طراحی کرد که از طریق ساخت یک درخت دودویی متناظر با یک کد بهینه، یک کد بهینه دودویی را تولید می کند. کدی که توسط این الگوریتم ایجاد می شود کد هافمن نام دارد.

مثال: مجموعه کاراکترهای $\{a,b,c,d,e,f\}$ را در نظر بگیرید که تعداد فراوانی هر کاراکتر در فایل، در زیر آمده است. کد هر کاراکتر به روش هافمن را بدست آورید؟

حل: در شکل زیر مجموعه ای از درخت ها را که در هر گذر از حلقه for i توسط الگوریتم ساخته می شوند، نشان داده شده است. مجموعه(0) ، قبل از ورود به حلقه را نشان می دهد. (مقدار ذخیره شده در هر گره، مقدار فیلد frequency در گره است.)



درخت نهایی مانند درخت شماره 5 است ولی در داخل گره های مشخص شده با دایره، عددی قرار نمی دهیم.

- درخت دودویی متناظر با یک کد پیشوندی دودویی بهینه، کامل است. یعنی، هر گره غیر برگ حاوی دو فرزند است.
- الگوریتم هافمن یک کد دودویی بهینه را تولید می کند.

تذکر: الگوریتم هافمن در پیوست آورده شده است.

- الگوریتم کد هافمن در زمان $\theta(n \lg n)$ اجرا می شود.

مسئله کوله پشتی

همه الگوریتم های حریصانه حل بهینه را تولید نمی کنند. برای آن که نشان داده شود یک الگوریتم حریصانه همواره حل بهینه ای به دست می دهد باید آن را اثبات کرد. در مورد برنامه نویسی پویا، کافی است تعیین کنیم که آیا اصل بهینگی برقرار است یا خیر. برای روشن تر شدن اختلاف میان دو روش پویا و حریصانه، دو مسئله کوله پشتی صفر و یک و مسئله کوله پشتی کسری را ارائه می دهیم. مشاهده خواهیم کرد که مسئله کوله پشتی کسری را به کمک یک الگوریتم حریصانه می توان حل کرد ولی این الگوریتم قادر نیست مسئله کوله پشتی صفر و یک را حل کند و مسئله کوله پشتی صفر و یک را به کمک برنامه نویسی پویا حل می کنیم.

دزدی با کوله پشتی وارد یک جواهرفروشی می شود. اگر وزن کل قطعات از یک حد بیشینه W فراتر رود، کوله پشتی پاره خواهد شد. هر قطعه دارای ارزش و وزن معینی خواهد بود. هدف تعیین حداکثر ارزش قطعات است در حالی که وزن کل آن ها از حد معین W فراتر نرود. این مسئله را مسئله کوله پشتی صفر و یک می گویند. فرض کنید n قطعه وجود داشته باشد به طوری که:

$S = \{item_1, item_2, item_3, ..., item_n\}$

ورن $item_i$ را با W_i و ارزش آن را با P_i نشان می دهیم. (w_i ، P_i و W همگی اعدادی صحیح هستند.)

باید زیر مجموعه A از S تعیین شود به قسمی که : $\sum P_i$ نسبت به $\sum W_i \le W$ بیشینه شود. قطعات را به ترتیب افزایش ارزش آن ها به ازای واحد وزن مرتب می کنیم و آن ها را به ترتیب انتخاب می کنیم. هر قطعه در صورتی در کوله گذاشته می شود که وزن کل فراتر از W نرود.

مثال: فرض کنید سه قطعه با ارزش و وزن مشخص شده در جدول زیر داریم. ظرفیت کوله پشتی(W) نیز 30 پوند است. نحوه انتخاب قطعات چگونه است؟

قطعه	ارزش(دلار)	وزن(پوند)
1	50	5
2	60	10
3	140	20

حل: ارزش قطعات به ازای واحد وزن آن ها به صورت زیر است:

$$item_1 : \frac{\$50}{5} = \$10 \quad , \quad item_2 : \frac{\$60}{10} = \$6 \quad , \quad item_3 : \frac{\$140}{20} = \$7$$

بنابراین ابتدا $item_1$ و سپس $item_3$ انتخاب می شود که سود کل 190 را می دهد، حال آن که حل بهینه، انتخاب $item_2$ و $item_3$ است که 200 سود را در بر دارد. مشکل این جاست که پس از آن که $item_1$ و $item_3$ انتخاب شدند، 5 پوند از ظرفیت باقی می ماند، ولی این اتلاف به این دلیل است که $item_2$ ، مقدار 10 پوند وزن دارد. این الگوریتم حریصانه قادر به حل مسئله کوله پشتی صفر و یک نیست.

کوله پشتی کسری

در این مسئله دزد می تواند هر کسری از قطعه را بردارد. می توان قطعات در مسئله کوله پشتی صفر و یک را به صورت شمش های طلا و نقره و در مسئله کوله پشتی کسری، به صورت کیسه های حاوی خاک طلا و نقره در نظر گرفت.

مثال: با توجه به قطعات زیر، اگر راهبرد حریصانه انتخاب قطعات با بزرگ ترین ارزش به ازای واحد وزن باشد، نحوه انتخاب قطعات چگونه است؟

قطعه	ارزش(دلار)	وزن(پوند)
1	50	5
2	60	10
3	140	20

قطعه 2 پر می شود. این کسر از تقسیم ورن باقی مانده به وزن قطعه بدست می آید. سود کلی که حاصل می شود:

$$50+140+\frac{5}{10}\times 60=220$$

الگوریتم حریصانه در مسئله کوله پشتی کسری مثل کوله پشتی صفر و یک، ظرفیت را هدر نمی دهد و همواره حل بهینه را به دست می دهد.

کوله پشتی صفر و یک (روش پویا)

اگر بتوانیم نشان دهیم که اصل بهینگی برقرار است، می توانیم مسئله کوله پشتی صفر و یک را با استفاده از برنامه نویسی پویا حل کنیم. فرض کنید A یک زیر مجموعه بهینه از n قطعه باشد. دو حالت وجود دارد: یا A حاوی $item_n$ هست یا نیست. اگر A حاوی $item_n$ نباشد، A با زیر مجموعه ای بهینه از (n-1) قطعه نخست برابر است. اگر A حاوی $item_n$ باشد، سود کل حاصل از قطعات موجود در A برابر P_n به علاوه سود بهینه است. وقتی که قطعات را بتوان از (n-1) قطعه نخست انتخاب کرد (وزن کل آن ها از $W-w_n$ بیشتر نشود). بنابراین، اصل بهینگی برقرار است.

اگر به ازای $i > 0$ و $w > 0$، p[i][w] سود بهینه حاصل از انتخاب i قطعه اول و تحت این محدودیت باشد که وزن کل نباید از w تجاوز کند:

$$P[i][w]=\begin{cases}\max imum(P[i-1][w],P_i+P[i-1][w-w_i]) & w_i\le w\\ P[i-1][w] & w_i>w\end{cases}$$

سود بشینه برابر با p[n][w] است. می توانیم این مقدار را با استفاده از آرایه دو بعدی P [0..n][0..w] تعیین کنیم. مقادیر موجود در سطرها و ستون های آرایه را به ترتیب با استفاده از رابطه قبلی برای p[i][w] محاسبه می کنیم. مقادیر p[0][w] و p[i][0] را مساوی صفر قرار می دهیم.

تعداد عناصری که محاسبه می شود عبارت است از: $n\ \ W\in q(nW)$

نسخه بهتر الگوریتم

این واقعیت که عبارت قبلی برای تعداد عناصر آرایه محاسبه شده نسبت به n خطی است، ممکن است باعث شود که به خطا تصور کنیم آن الگوریتم برای همه نمونه هایی که حاوی n قطعه هستند، کارایی دارد، ولی چنین نیست. برای مثال، اگر $W=n!$ باشد، تعداد عناصر محاسبه شده، $q(n\times n!)$ است. اگر n = 20 و $W=20!$ باشد، این الگوریتم روی یک کامپیوتر مدرن هزاران سال به طول خواهد انجامید.

تذکر: هنگامی که W در مقایسه با n بسیار بزرگ باشد، این الگوریتم از الگوریتم غیرهوشمند که همه زیر مجموعه ها را در نظر می گرفت بدتر است.

این الگوریتم را می توان چنان بهبود بخشید که در بدترین حالت تعداد عناصر محاسبه شده به $\theta(2^n)$ تعلق داشته باشد. با این بهبود، الگوریتم هیچگاه از روش غیر هوشمند بدتر نمی شود و غالباً بسیار بهتر است.

شود. بنابراین، تنها عناصر مورد نیاز در سطر (n-1) اُم، آن هایی هستند که برای محاسبه p[n][w] به کار می روند. چون:

$$P[n][w]=\begin{cases}\max imum(P[n-1][w], P_n + P[n-1][w-w_n]) & w_n \le W \\ P[n-1][W] & w_n > W\end{cases}$$

تنها عناصر مورد نیاز در سطر (n-1) اُم عبارت است از: $P[n-1][W-w_n]$ و $P[n-1][W]$.

از n به طرف عقب ادامه می دهیم تا تعیین کنیم کدام عناصر مورد نیازند. یعنی، پس از آن که تعیین کردیم کدام عناصر در سطر i اُم مورد نیازند، با استفاده از این واقعیت که : P[i][w] از روی P[i-1][w] و p[i-1][w-w_i] محاسبه می شود، تعیین می کنیم کدام عناصر در سطر (i-1) اُم مورد نیاز هستند. هنگامی که به n = 1 یا $w \le 0$ رسیدیم، توقف می کنیم. پس از تعیین این که کدام عناصر مورد نیاز هستند، محاسبات را با شروع از سطر نخست انجام می دهیم.

مثال: فرض کنید قطعات زیر و w = 30 را داشته باشیم. سود بهینه را بدست آورید.

قطعه	ارزش(دلار)	وزن(پوند)
1	50	5
2	60	10
3	140	20

حل: نخست باید تعیین کنیم در هر سطر کدام عناصر باید محاسبه شود.

تعیین عناصر مورد نیاز در سطر 3 : P[3][W]=P[3][30]

تعیین عناصر مورد نیاز در سطر 2 : برای محاسبه P[3][30] به موارد زیر نیاز داریم:

P[3-1][30]=P[2][30] , P[3-1][30-w_3]=P[2][10]

تعیین عناصر مورد نیاز در سطر 1 : برای محاسبه P[2][30] به موارد زیر نیاز داریم:

P[2-1][30]=P[1][30] , P[2-1][30-w_2]=P[1][20]

برای محاسبه P[2][10] به موارد زیر نیاز داریم:

P[2-1][10]=P[1][10] , P[2-1][10-w_2]=P[1][0]

حال محاسبات را انجام می دهیم.

محاسبه سطر 1:

$$P[1][w]=\begin{cases}\max imum(P[0][w], \$50 + P[0][w-5]) & w_1 = 5 \le w \\ P[0][W] & w_1 = 5 > W\end{cases}$$

$$=\begin{cases}\$50 & w_1 = 5 \le w \\ \$50 & w_1 = 5 > w\end{cases}$$

بنابراین: P[1][10]= 0 , P[1][10]= 50 , P[1][20]= 50 , P[1][30]= 50

$$P[2][10] = \begin{cases} \max imum(P[1][10], \$60 + P[1][0]) & w_2 = 10 \le 10 \\ P[1][0] & w_2 = 10 > 10 \end{cases}$$

$$= \$60$$

$$P[2][30] = \begin{cases} \max imum(P[1][30], \$60 + p[1][20]) & w_2 = 10 \le 30 \\ P[1][30] & w_2 = 10 > 30 \end{cases}$$

$$= \$60 + \$50 = \$110$$

محاسبه سطر 3:

$$P[3][30] = \begin{cases} \max imum(P[2][30], \$140 + P[2][10]) & w_3 = 20 \le 30 \\ P[2][30] & w_3 = 20 > 30 \end{cases}$$

$$= \$140 + \$60 = \$200$$

تذکر: این نسخه از الگوریتم فقط هفت عنصر را محاسبه می کند، حال آن که نسخه اولیه 3×30 یعنی 90 عنصر را محاسبه می کرد.

- تعداد عناصری که توسط الگوریتم برنامه نویسی پویا برای مسئله کوله پشتی صفر و یک محاسبه می شود در بدترین حالت به $O(\min imum(2^n, nW))$ تعلق داشت.
- تا کنون کسی برای مسئله کوله پشتی صفر و یک الگوریتمی نیافته است که بدترین حالت آن بهتر از زمان نمایی باشد و در عین حال هنوز کسی عدم امکان آن را نیز اثبات نکرده است. (مانند مسئله فروشنده دوره گرد)

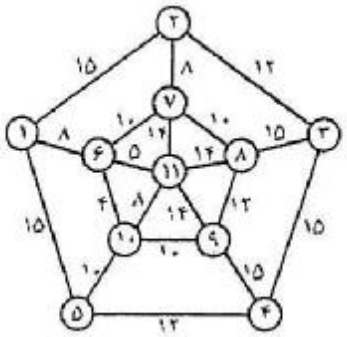
مجموعه تست

۱- مساله کوله پشتی کسری(fractional) زیر را که در آن گنجایش کوله ۱۰۰ است در نظر بگیرید. بیشترین سودی که حاصل می شود چقدر است؟

اشیاء	07	06	05	04	03	02	01
ارزش	1	5	2	3	3	2	7
وزن	20	30	70	60	50	40	10

1) 14,5 2) 16 3) 15,5 4) 15

۲- اگر از الگوریتم Kruskal برای یافتن درخت پوشای کمینه (MST) گراف زیر استفاده کنیم، یال (۶،۱۱) در چه مرحله ای انتخاب می شود؟



1) مرحله 1 2) مرحله 4 3) مرحله 3 4) مرحله 2

۳- در الگوریتم فشرده سازی هافمن، اگر برای یافتن دو نویسه با کمترین فراوانی از جست و جوی خطی به جای هرم(heap) استفاده شود، زمان اجرای آن چه خواهد بود؟(n تعداد نویسه هایی است که قرار است کد گذاری شوند)

1) $\theta(n)$ 2) $\theta(n^2)$ 3) $\theta(n^2 \lg n)$ 4) $\theta(n \lg n)$

۴- چند تا از گزاره های زیر درست هستند؟

الف- اگر در یک گراف وزن دار بدون جهت، یال e=(u,v) در درخت فراگیر کمینه باشد، حتما دو راس وجود دارد که کوتاهترین مسیر بین آن دو از e می گذرد.

ب- مسئله ی یافتن همه ی کوتاه ترین مسیرها از یک راس به بقیه ی راس ها را در یک گراف وزن دار بدون جهت و هم بند با مجموعه ی یال های E را می توان در $O(|E|)$ و نه در $O(|E|+|V|)$ است.

ج- در یک گراف وزن دار و ساده بدون جهت با وزن های نامساوی، یال با سبک ترین وزن و یال دومین سبک ترین وزن هر دو در درخت فراگیر کمینه هستند.

1) صفر 2) 1 3) 2 4) 3

چند تا از گزاره های زیر درست اند؟

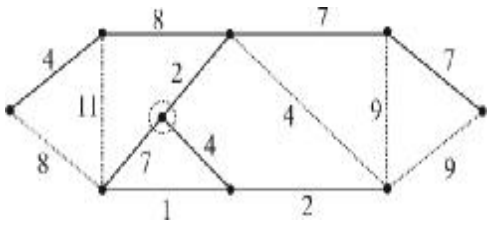
الف- برش کمینه ی (S,T) در هر دو گراف یکی است.

ب- درخت فراگیر کمینه ی هر دو گراف یکی است.

ج- کوتاه ترین مسیر بین دو راس مشخص در دو گراف شامل یال های یکسان هستند.

1) 1 2) صفر 3) 2 4) 3

۶- در گراف روبرو لبه‌های پررنگ، درخت پوشای حداقل را با استفاده از روش prim نمایش می‌دهد. گره‌ای که با دایره مشخص شده نقطه شروع است. در این حالت آخرین گره ای که به درخت پوشا متصل شده است، دارای چه وزنی است؟



1) 8 2) 7 3) 4 4) 2

۷- در مسئله knapsack ۰-۱ با پنج شییء به شرح زیر، ظرفیت کیسه برابر ۱۳ کیلو گرم می باشد. مقدار سود ماکزیمم چند دلار چیست؟ (p_i سود و w_i وزن شیء i ام)

i	1	2	3	4	5
p_i	35	30	20	12	3
w_i	7	5	2	3	1

1) 65 2) 68 3) 70 4) 80

۸- اگر قرار باشد ۶ سخنرانی زیر با شروع و پایان مشخص را طوری برنامه ریزی کنیم که بیشترین تعداد سخنرانی در یک سالن قابل ارائه باشد، بیشترین تعداد سخنرانی های ممکن کدام است؟

شماره سخنرانی	۱	۲	۳	۴	۵	۶
شروع	۲	۱	۳	۳	۴	۶
پایان	۹	۲	۴	۵	۷	۸

1) 2 2) 5 3) 4 4) 3

۹- اگر رشته abcabbaccaabdffe را با روش کدینگ هافمن کد کنیم، طول کد چند بیت خواهد شد؟

1) 38 2) 36 3) 34 4) هیچکدام

1) $n-2$ 2) $n-1$ 3) $\frac{n}{2}$ 4) $\log n$

۱۱- جمله زیر را اگر به روش هافمن فشرده سازی کنیم، اندازه ی فشرده شده چند بیت خواهد بود؟

(∪فاصله خالی است.)

This ∪ is ∪ the ∪ set ∪ that ∪ has ∪ set

1) 60 2) 68 3) 76 4) 92

۱۲- متنی شامل ۷۰۰۰ حرف از حروف a و b و c و d و e و f با تفاوت تکرار a=۱۰۰۰ ، b=۱۲۰۰، c=۸۰۰، d=۱۵۰۰ ، e=۱۸۰۰ و f=۷۰۰ موجود است. چنانچه کدی بهینه برای حروف بالا انتخاب نمائیم، تعداد کل بیتهای لازم برای تبدیل متن مذکور به مجموعه ای از بیتها چقدر است؟

1) 14600 2) 17700 3) 24300 4) 35200

۱۳- رشته متنی (abaabacadcade) را در نظر بگیرید. کد هافمن حاصل برای هر یک از نویسه ها برابر است با.....

1) a=1,b=01,c=001,d=0001,e=0000
2) a=0,b=100,c=101,d=110,e=100
3) a=000,b=001,c=010,d=011,e=100
4) a=00,b=01,c=10,d=11,e=100

۱۴- در فشرده سازی با استفاده از کد هافمن اگر x و y دو کاراکتری باشند که کمترین تکرار را داشته باشند در آن صورت کد بهنه برای این دو نویسه(کاراکتر)

1) دارای پیشوند یکسان می باشند، فقط در آخرین لیست آنها متفاوت است.
2) دارای پسوند یکسان می باشند فقط در اولین لیست متقاوت است.
3) دارای پیشوند یکسان نمی باشند و فقط کد آخر آنها برابر است.
4) دارای پیشوند یکسان نمی باشند و فقط کد اول آنها با هم برابر است.

۱۵- یک گراف همبند G=(V,E) که وزن یالهای آن ۱ است را در نظر بگیرید. می خواهیم درخت فراگیر کمینه (MST) این گراف را پیدا کنیم:

1) این کار را می توان در $O(|E|\log|E|)$ انجام داد.
2) این کار را می توان در $O(|V||E|)$ انجام داد.
3) این کار را می توان در $O(|V|\log|E|)$ انجام داد.
4) این کار را می توان در $O(|E|)$ انجام داد.

درست است؟

1) در روش Prim در هر مرحله هم بندی درخت رعایت مـی شـود و زمـان مصـرفی الگـوریتم Prim و الگـوریتم Kruskal یکسان است.

2) در روش Prim در هر مرحله هم بندی درخت رعایت می شود، اما در روش Kruskal در مرحله پایانی، این امر اتفـاق مـی افتد، اما زمان مصرفی روش Prim بهتر است.

3) در روش Kruskal ، در هر مرحله هم بندی درخت رعایت می شود، اما در روش Prim در مرحله پایانی این امر اتفاق مـی افتد، اما زمان مصرفی روش Prim بهتر است.

4) در روش Kruskal، در هر مرحله هم بندی درخت رعایت نمی شود، اما در پایان الگوریتم درخت هم بند اسـت. امـا زمـان مصرفی آن از روش Prim بهتر است.

۱۷- درخت پوشای مینیمم برای گراف وزن دار G، در چه صورت یکتا می باشد؟

1) در صورتی که گراف G یک درخت باشد.

2) در صورتی که در گراف G هیچ دو یالی دارای وزن یکسان نباشند.

3) در صورتی که گراف G غیر جهتدار و مرتبط باشد.

4) موارد 1و2

۱۸- فرض کنیم گراف G=(V,E) یک گراف بی‌جهت و M یک درخت فراگیر مینیمم بـرای G باشـد، کـدام عبـارت درست است؟

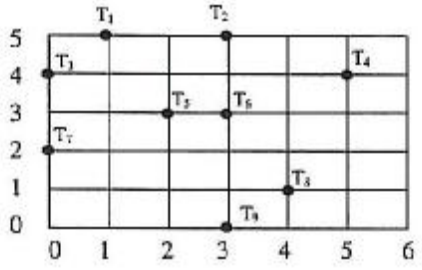
1) مسیر M بین هر جفت رأس V_1 وV_2 کوتاه‌ترین مسیر است.

2) مسیر M بین هر جفت رأس V_1 وV_2 کوتاه‌ترین مسیر نمی‌باشد.

3) تمام مسیرهای موجود در M کوتاه‌ترین مسیر می‌باشند.

4) بین تمام رئوس در M مسیری وجود ندارد.

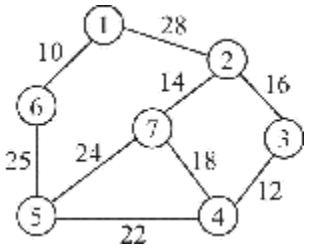
گردد. فرض کنید ۵ ولت به یکی از ترمینال ها وصل است. برای اینکه کمترین سیم بنـدی در مـدار بـه کـار رود، بگوئید حداقل چقدر سیم لازم دارد؟(فاصله هر سطر و ستون را یک سانتی متر در نظر بگیرید.)

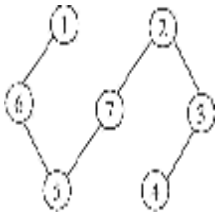


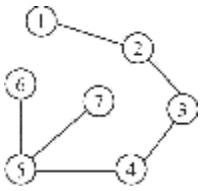
1) $8+2\sqrt{2}+2\sqrt{5}$ 2) $6+2\sqrt{2}+2\sqrt{5}$

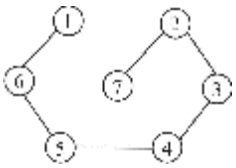
3) $5+2\sqrt{2}+3\sqrt{5}$ 4) $7+2\sqrt{2}+2\sqrt{5}$

۲۰- درخت پوشا با حداقل هزینه گراف زیر کدام است؟



1) 

2) 

3) 

4) هیچکدام

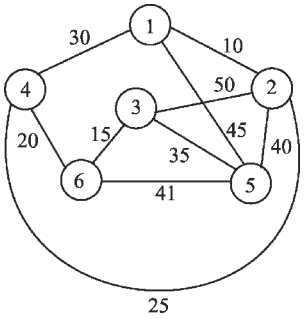
۲۱- گراف همبندی بدون جهت با n گره و n-۲ لبه داریم. کدامیک از الگوریتم های زیر برای تولید درخـت پوشـا بـا حداقل هزینه بر روی گراف مناسب تر است؟

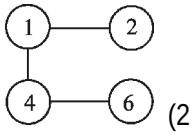
1) prim 2) kruskal 3) sollin 4) هیچکدام

۲۲- برای یافتن درخت پوشای حداقل (Minimum Spanning tree) یک گراف خلوت، کدامیک از الگوریتم های زیر مناسب تر است؟

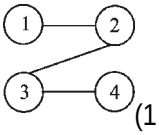
1) Floyd 2) Prim 3) Kruskal 4) Dijkstra

درخت حاصله در انتهای مرحله سوم این الگوریتم را به ما می‌دهد؟

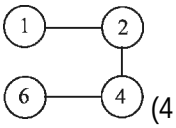




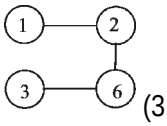
(2



(1



(4

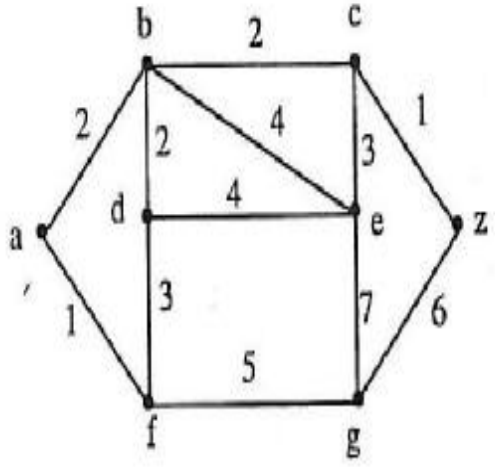


(3

۲۴- شش کار به شرح ذیل داریم. g_i نشان دهنده سود حاصل از اجرای کار i ام است اگر و فقط اگر بعد از زمان d_i انجام نشود. فرض کنید هر کار در واحد زمان انجام می شود. حداکثر سود حاصل از اجرا چقدر است؟

i	1	2	3	4	5	6
G_i	20	15	10	7	5	3
D_i	3	1	1	3	1	3

1) 42 2) 45 3) 37 4) 32



18 (4 12 (3 3 (2 5 (1

پاسخ تشریحی

3-1) ابتدا نسبت وزن به ارزش را محاسبه می کنیم:

اشیاء	07	06	05	04	03	02	01
وزن / ارزش	20	6	35	20	16,6	20	1,4

سپس بر اساس مقدار صعودی این نسبت، شی ء های 1 و 6 و3 را به طور کامل انتخاب می کنیم . در این حالت 90 کیلو از 100 کیلو ظرفیت کوله پشتی پر می شود و ده کیلو باقی مانده را با کسری از شیء 4 باید پر کنیم. این کسر از تقسیم وزن باقی مانده به وزن قطعه بدست می آید یعنی $\frac{10}{40}$

بنابرین سود کل برابر است با:

$$7+5+3+\frac{10}{40}\times 20=15.5$$

4-2) طبق الگوریتم کراسکال، در مرحله اول کوتاهترین یال یعنی (6،10) و در مرحله دوم یال (6،11) انتخاب می شود.

2_3)

3_4)

3_5)

3-6) با دنبال کردن الگوریتم پریم ، به سادگی مشخص می شود که آخرین گره ای که به مجموعه گره های انتخابی اضافه خواهد شد، گره با وزن 4 می باشد.

3-7) برای کسب سود ماکزیمم، قطعات 1و3و4و5 را انتخاب می کنیم. وزن کل این قطعات برابر است با:

7+2+3+1=13

و سود این قطعات برابر است با:

35+20+12+3=70

4-8) ابتدا فعالیت ها را بر اساس زمان پایان مرتب می کنیم:

i	1	2	3	4	5	6
S[i]	1	3	3	4	6	2
F[i]	2	4	5	7	8	9

سپس کارهایی را انتخاب کرده که شروع آنها از پایان کار قبلی انتخاب شده، کمتر نباشد. بنابراین کارهای 1و2و4 انتخاب می شوند.

1-9) در رشته داده شده تعداد تکرار هر یک از حروف برابر است با: a=5 , b=4 , c=3 , d=1 , f=2 , e=1

با رسم درخت هافمن، مشخص می شود که 2،c,b,a بیتی و e,d ، 4بیتی می باشند. بنابراین تعداد کل بیت های مورد نیاز برابر است با: $5\times 2+4\times 2+3\times 2+1\times 4+2\times 3+1\times 4=38$

ریشه فاصله دارد. به طور مثال برای $n=3$ ، بزرگترین طول کد برابر 2 می باشد.

3-11) تعداد تکرار هر یک از حروف برابر است با:

$t=6$, $h=4$, $i=2$, $s=5$, $e=3$, $\cup=6$

بعد از رسم درخت هافمن، کد هر یک از کاراکترها به صورت زیر مشخص می شود:

$t=00$, $\cup=01$, $i=1000$, $a=1001$, $e=101$, $h=110$, $s=111$

2-12) با رسم درخت هافمن کد هر یک از حروف مشخص می شود:

$a=000$, $b=001$, $c=110$, $d=10$, $e=01$, $f=111$

در نتیجه f,c,b,a ، 3 بیتی و e,d ، 2 بیتی هستند. بنابراین تعداد کل بیتها برابر است با:

$$1000\times3+1200\times3+800\times3+1500\times2+1800\times2+700\times3=17700$$

2-13) تعداد تکرار هر یک از کاراکترها برابر است با: $a=6,b=2,c=2,d=2,e=1$ با رسم درخت هافمن کد هر یک از حروف مشخص می شود: $a=0$, $b=100$, $c=101$, $d=110$, $e=111$

1-14) در فشرده سازی با استفاده از کد هافمن اگر x وy دو کاراکتری باشند که کمترین تکرار را داشته باشند در آن صورت کد بهینه برای این دو کاراکتر دارای پیشوند یکسان می باشند، فقط در آخرین لیست آنها متفاوت است. به طور نمونه اگر تعداد تکرار هر یک از حروف برابر باشد با: $a=1,b=1,c=10,d=30$ ، آنگاه با رسم درخت هافمن مشخص می شود که کد a برابر 000 و کد b برابر 001 است. یعنی a و b که کمترین تعداد تکرار را دارند، کد آنها دارای پیشوند یکسان 00 است.

4_15)

2-16) در روش Prim در هر مرحله هم بندی درخت رعایت می شود، اما در روش Kruskal در مرحله پایانی، این امر اتفاق می‌افتد.

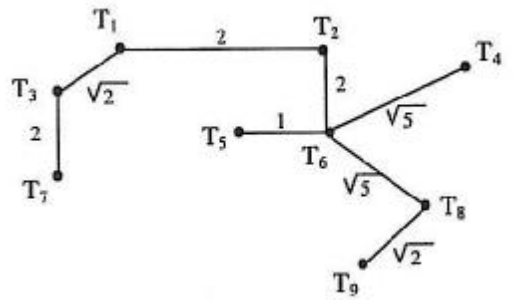
4-17) درخت پوشای مینیمم برای گراف وزن دار G ، در دو حالت یکتا می باشد:

الف- گراف یک درخت باشد.(حلقه نداشته باشد)

ب- هیچ دو یالی در گراف، دارای وزن یکسانی نباشند.

2-18) دلیلی ندارد که مسیر بین دو گره در درخت پوشا ، کوتاهترین مسیر بین آن دو گره باشد.

4-19) این مساله مانند درخت پوشای حداقل است که به کمک روش کراسکال آن را حل می کنیم. یالهای انتخابی به صوت زیر است:

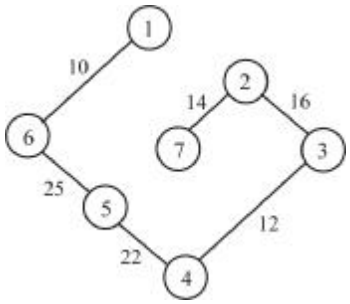


$$1+(\sqrt{2}+\sqrt{2})+(2+2+2)+(\sqrt{5}+\sqrt{5})=7+2\sqrt{2}+2\sqrt{5}$$

3-20) اگر از روش کراسکال در مورد گراف استفاده شود، ترتیب اضافه شدن یالها برابر است با:

10،12،14،16،22،25

و درخت پوشای با حداقل هزینه زیر حاصل می شود:



4-21) گراف همبند بدون جهت با n گره حداقل $n-1$ یال دارد و نمی‌توان گراف همبندی با $n-2$ یال رسم کرد.

3-22) در گراف خلوت، تعداد یالها کم می‌باشد و چون الگوریتم کراسکال براساس یالها کار می‌کند، مناسب تراست.

4-23) در الگوریتم پریم از گره شروع به سمتی حرکت می کنیم که کمترین مسیر را داشته باشد. سپس از مجموعه گره های حاصل، به گره‌ای می رویم که کمترین مسیر را داشته باشد و به همین روال ادامه می دهیم. البته باید توجه کرد که حلقه ای ایجاد نگردد.

در گراف داده شده 3 مرحله اول به صورت زیر است :

مرحله اول : اتصال گره 1 به گره 2 . مرحله دوم : اتصال گره 2 به گره 4 . مرحله سوم : اتصال گره 4 به گره 6 .

1-24) با توجه به مهلت داده شده کارهای 1و2و4 انتخاب می شوند. بنابراین سود حاصل برابر است با: 20+15+7=42

1-25) کوتاهترین مسیر بین دو راس a و z برابر است با: $a \to b \to c \to z$ که طول آن برابر است با:2+2+1=5

فصل پنجم : روش عقبگرد

از تکنیک عقبگرد برای حل مسائلی استفاده می‌شود که در آن‌ها دنباله‌ای از اشیا از یک مجموعه مشخص انتخاب می‌شود، به طوری که این دنباله، ملاکی را در بر می‌گیرد.

اگر سعی کنید راه خود را از "هزار توی شمشادی" معروف بیابید، راهی ندارید جز دنبال کردن مسیری بی‌امید تا این که به یک بن‌بست برسید. در این حالت، برمی‌گردید و راهی دیگر را امتحان می‌کنید. فکر کنید اگر تابلویی وجود داشته باشد که به شما بگوید مسیری که پیش گرفته‌اید به بن‌بست منتهی می‌شود، چقدر کارها آسان‌تر می‌شود. اگر این تابلو در ابتدای راه قرار داده شود، زمان صرفه‌جویی شده می تواند چشمگیر باشد، زیرا همهٔ دو راهی‌های بعد از آن را دیگر لازم نیست در نظر بگیریم. بدین ترتیب از چندین بن‌بست پرهیز می‌شود. در الگوریتم‌های عقبگردی این تابلوها وجود دارند.

روش عقبگرد برای مسائلی نظیر کوله‌پشتی صفر و یک بسیار مفید است. یک الگوریتم برنامه‌نویسی پویا برای این مسئله یافتیم که در بدترین حالت، نمایی است. یک راه برای حل مسئله کوله‌پشتی صفر و یک، این است که واقعاً همهٔ زیر مجموعه‌ها تولید شوند، ولی این درست مثل حرکت در هزار توی بدون تابلو و رسیدن به بن‌بست است. چون 2^n زیر مجموعه وجود دارد و این روش فقط وقتی عملی است که n کوچک باشد، ولی اگر هنگام تولید این زیر مجموعه‌ها بتوانیم علائمی بیابیم که به ما بگویند بسیاری از این زیر مجموعه‌ها نیاز به تولید شدن ندارند، می توان از صرف وقت زیاد پرهیز کنیم، و این چیزی است که الگوریتم عقبگردی انجام می‌دهد.

الگوریتم‌های عقبگردی برای مسائلی از قبیل کوله‌پشتی صفر و یک در بدترین حالت باز هم نمایی (یا حتی بدتر) هستند. این الگوریتم‌ها از آن‌رو مفید هستند که برای بسیاری از نمونه‌های بزرگ مؤثرند نه برای همهٔ نمونه‌ها.

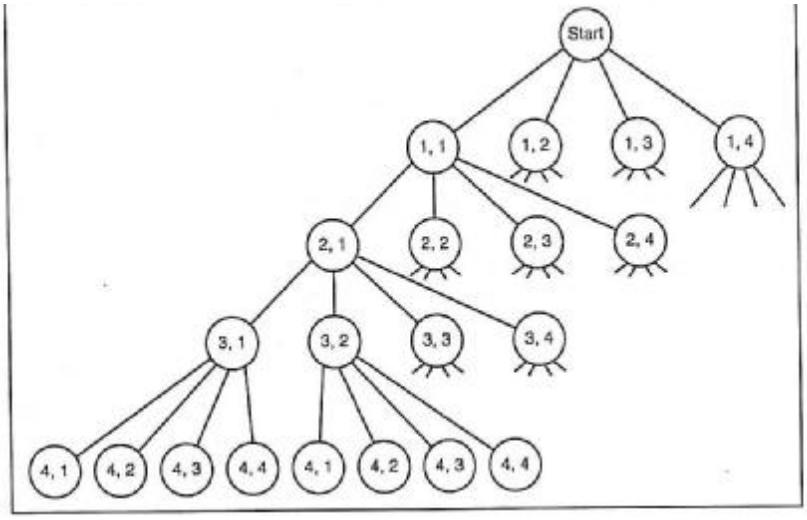
مسئله n وزیر

هدف چیدن n مهره وزیر در یک صفحهٔ شطرنج $n \times n$ است، به طوری که هیچ دو وزیری یکدیگر را گارد ندهند، یعنی هیچ دو مهره‌ای نباید در یک سطر، ستون یا قطر یکسان باشند.

در این مسئله، دنباله عبارت از n موقعیتی است که در آن‌ها وزیرها قرار داده می‌شوند، مجموعه برای هر انتخاب، عبارت از n^2 موقعیت روی صفحه شطرنج است. مسئله n وزیر، شکل کلی نمونه‌ای است که در آن $n = 8$ است (صفحه شطرنج استاندارد). برای کوتاه کردن بحث، با استفاده از نمونه $n = 4$ مسئله را بررسی می کنیم.

مسئله را به این صورت ساده می کنیم که هیچ دو وزیری نباید در یک ردیف قرار بگیرند. با نسبت دادن هر وزیر به یک ردیف و چک کردن این که کدام ترکیب از ستون‌ها به حل می‌انجامد، مسئله را حل می کنیم. چون هر وزیر را می توان در یکی از چهار ستون قرار داد، تعداد حل‌های کاندیدا، $4 \times 4 \times 4 \times 4 = 256$ است.

حل‌های کاندیدا را می توان با ساختن درختی ایجاد کنیم که در آن انتخاب‌های ستون برای نخستین وزیر (وزیر ردیف اول) درگره‌های سطح 1 از درخت، انتخاب‌های ستون برای دومین وزیر (وزیر ردیف دوم) در گره‌های سطح 2، نگهداری می‌شوند و غیره. یک حل کاندید، مسیری از ریشه به یک برگ است. این درخت را درخت فضای حالت (state space tree) می‌گویند که بخش کوچکی از آن در شکل زیر نشان داده شده است:



کل درخت 256 برگ دارد که هر برگ به یک حل کاندیدا مربوط می‌شود. زوج مرتب $< i, j >$ به این معنی است که وزیر در ردیف iأم و ستون jأم قرار دارد.

برای تعیین حل‌ها، همهٔ حل‌های کاندیدا را به ترتیب با شروع از مسیری که در منتها الیه سمت چپ قرار دارد بررسی می‌کنیم (هر مسیر از ریشه به برگ). چند مسیر اول به صورت زیر چک می‌شوند:

$[< 1, 1 >, < 2, 1 >, < 3, 1 >, < 4, 1 >]$

$[< 1, 1 >, < 2, 1 >, < 3, 1 >, < 4, 2 >]$

$[< 1, 1 >, < 2, 1 >, < 3, 1 >, < 4, 3 >]$

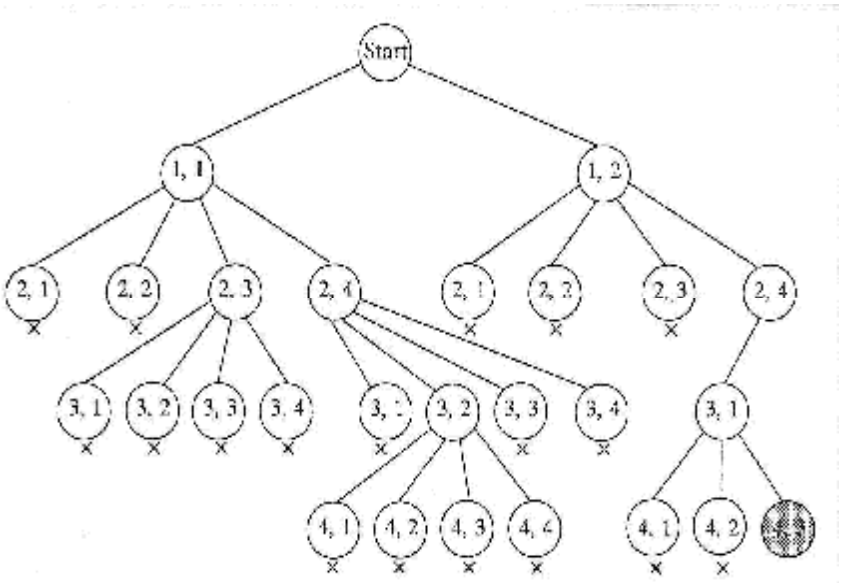
$[< 1, 1 >, < 2, 1 >, < 3, 1 >, < 4, 4 >]$

$[< 1, 1 >, < 2, 1 >, < 3, 2 >, < 4, 1 >]$

ترتیب ملاقات گره‌ها بر طبق جست‌وجوی عمقی است که در آن فرزندان یک گره از چپ به راست مورد ملاقات قرار می‌گیرند. جست‌و جوی عمقی سادهٔ یک درخت فضای حالت، از هیچ علامتی بهره نمی‌برد.

برای کارآمد تر کردن جست‌وجو،می توان از علائمی استفاده کرد. برای مثال، هیچ دو وزیری نمی توانند در یک ستون باشند. بنابراین، شاخهٔ ای از گره $< 2,1 >$ منشعب نمی کنیم و علامتی می زنیم.(چون از قبل وزیر 1 را در ستون 1 قرار داده‌ایم، نمی توانیم وزیر 2 را در آن قرار دهیم). این علامت به ما می‌گوید که این گره به بن‌بست منتهی می‌گردد. به طور مشابه، هیچ دو وزیری نمی توانند در یک قطر باشند و علامتی برای $< 2, 2 >$ می زنیم.

در روال عقبگرد، پس از تعیین این که گرهی منجر به بن‌بست شود، به گره والد باز می‌گردیم (عقبگرد می‌کنیم) و جست‌وجو را در فرزند دیگری از گره ادامه می‌دهیم. یک گره را غیر امید بخش می‌گوییم اگر هنگام ملاقات گره معلوم شود که احتمالاً منجر به حل نمی‌شود. عقبگرد شامل انجام دادن جست‌وجوی عمقی در درخت فضای حالت، تعیین امید بخش بودن گره و در صورت امیدبخش نبودن گره، منجر به بازگشت به گره والد می‌شود. این فرایند را هرس کردن درخت فضای حالت می گویند.



گرهی که امیدبخش نباشد با علامت × نشان داده می‌شود. گره ای که در آن نخستین راه‌حل یافت می‌شود را به صورت رنگی نشان داده ایم.

(الف) $<1,1>$ امیدبخش است.

(ب) $<2,1>$ امیدبخش نیست. {چون وزیر 1 در ستون 1 است}

$<2,2>$ امیدبخش نیست. {چون وزیر 1 در قطر چپی است}

$<2,3>$ امیدبخش است.

(ج) $<3,1>$ امیدبخش نیست. {چون وزیر 1 در ستون 1 است}

$<3,2>$ امیدبخش نیست. {چون وزیر 2 در ستون راستی است}

$<3,3>$ امیدبخش نیست. {چون وزیر 2 در ستون 3 است}

$<3,4>$ امیدبخش نیست. {چون وزیر 2 در قطر چپی است}

(د) عقبگرد به $<1,1>$.

$<2,4>$ امیدبخش است.

(ه) $<3,1>$ امیدبخش نیست. { چون وزیر 1 در ستون 1 است}

$<3,2>$ امیدبخش است.

(و) $<4,1>$ امیدبخش نیست. {چون وزیر 1 در ستون 1 است}

$<4,2>$ امیدبخش نیست. {چون وزیر 3 در ستون 2 است}

$<4,3>$ امیدبخش نیست. {چون وزیر 3 در قطر چپی است}

$<4,4>$ امیدبخش نیست. {چون وزیر 2 در ستون 4 است}

(ز) عقبگرد به $<2,4>$.

< 3, 4 > امیدبخش نیست. {چون وزیر 2 در ستون 4 است}

(ح) عقبگرد به ریشه.

< 1, 2 > امیدبخش است.

(ط) < 2, 1 > امیدبخش نیست. {چون وزیر 1 در قطر راستی است}

< 2, 2 > امیدبخش نیست. {چون وزیر 1 در ستون 2 است}

< 2, 3 >امیدبخش نیست. {چون وزیز 1 در قطر چپی است}

< 2, 4 > امیدبخش است.

(ی) < 3, 1 > امیدبخش است.

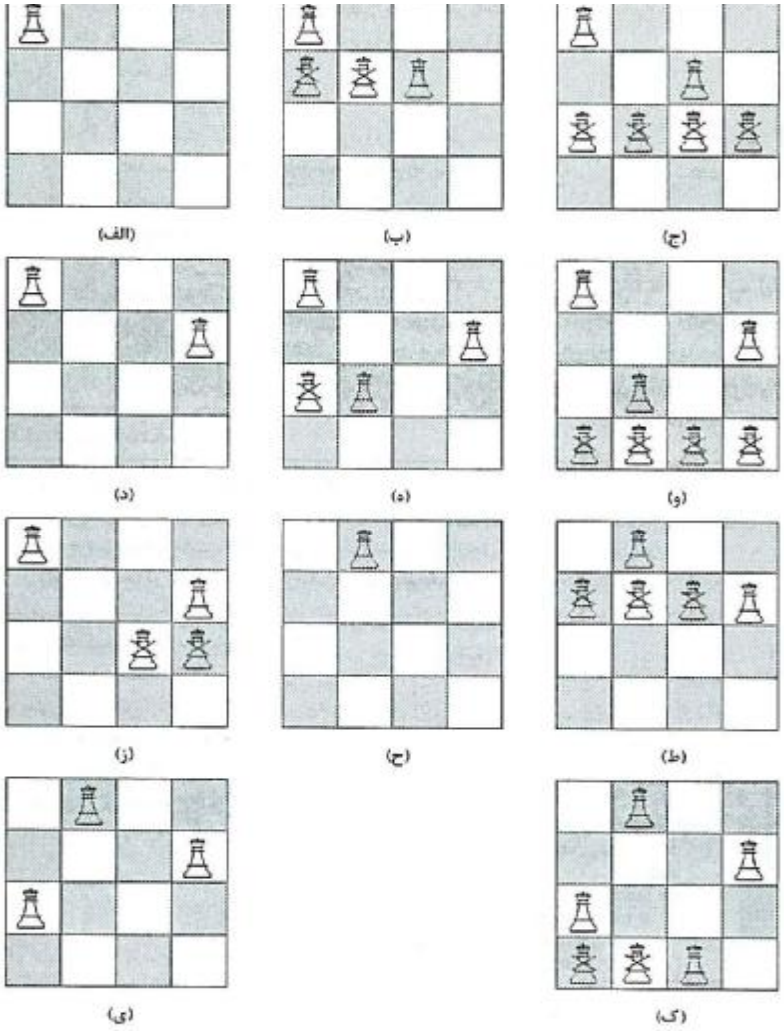
(ک) < 4, 1 > امیدبخش نیست. {چون وزیر 3 در ستون 1 است}

< 4, 2 > امیدبخش نیست. {چون وزیر 1 در ستون 2 است}

< 4, 3 > امیدبخش است.

در این نقطه، نخستین حل پیدا شده است. الگوریتم عقبگرد، پیش از یافتن یک حل، 27 گره را چک می‌کند. بدون عقبگرد، جست‌وجوی عمقی در درخت فضای حالت مستلزم چک کردن 155 گره پیش از رسیدن به همان حل است.

شکل زیر موقعیت های واقعی صفحه شطرنج برای حل مسئله 4 وزیر را نشان می دهد:



تذکر: الگوریتم عقبگرد در واقع نیاز به ایجاد درخت ندارد. بلکه فقط کافی است مقادیر موجود در شاخه‌ای کـه مـورد بررسـی قـرار می‌گیرد، در دسترس باشد. درخت فضای حالت به طور ضمنی در الگوریتم وجود دارد، زیرا واقعاً ساخته نمی‌شود.

اُم i ستونی باشد که در آن وزیر ردیف col(i)تابع امید بخش باید چک کند که آیا دو وزیر در یک ستون یا قطر هستد یا خیر. اگر اُم هـر دو در یـک سـتون هسـتند بایـد ببینـیم کـه آیـا: kقرار می‌گیرد، در آن صورت، برای این که ببینیم آیا با وزیر در ردیـف اُم را در راستای یکی از قطرهایش گـارد بدهـد، در ایـن iاُم، وزیری در ردیف k . بنابراین اگر وزیر در ردیف $col(i) = col(k)$ صورت:

$\mathrm{col(i) - col(k) = k - i}$ یا $col(i) - col(k) = i - k$

تذکر: الگوریتم n وزیر در پیوست کتاب آورده شده است.

مسئله حاصل جمع زیر مجموعه‌ها

در مسئله کوله‌پشتی صفر و یک، هدف، بیشینه‌سازی ارزش کل قطعات است با این شرط که وزن کل از W تجاوز نکند. با فرض اینکه ارزش این قطعات در هر واحد وزن، یکسان باشند،حل بهینه برای دزد صرفاً مجموعه‌ای از قطعات است که وزن کل آن‌ها بیشینه باشد و فقط این محدودیت وجود دارد که وزن کل آن‌ها از W فراتر نرود. ممکن است در آغاز، دزد سعی کند تعیین کند که آیا مجموعه‌ای وجود دارد که وزن کل آن برابر W باشد یا خیر، زیرا این حالت از همه بهتر است. مسئله تعیین چنین مجموعه‌هایی را مسئله حاصل جمع زیر مجموعه‌ها می‌گویند.

مشخصاً، در مسئله حاصل جمع زیر مجموعه‌ها، n عدد صحیح مثبت، w_i (اوزان) و یک عدد صحیح مثبت W وجود دارد. هدف، یافتن همهٔ زیر مجموعه‌هایی از این اعداد صحیح است که حاصل جمع آن‌ها برابر W است. معمولاً مسائل را طوری بیان می‌کنیم که همهٔ حل‌ها را بیابیم. ولی برای هدفی که دزد دارد، تنها یافتن یک حل کافی است.

مثال: به ازای n=5 وW=21 ، حل های مسئله حاصل جمع زیرمجموعه ها را مشخص کنید.

$w_1 = 5 , w_2 = 6$ و $w_3 = 10$ و $w_4 = 11$ و $w_5 = 16$

حل:حل‌ها عبارتند از: $\{w_1, w_2, w_3\}, \{w_1, w_5\}$ و $\{w_3, w_4\}$چون جمع وزن هر یک از آنها برابر 21 است.

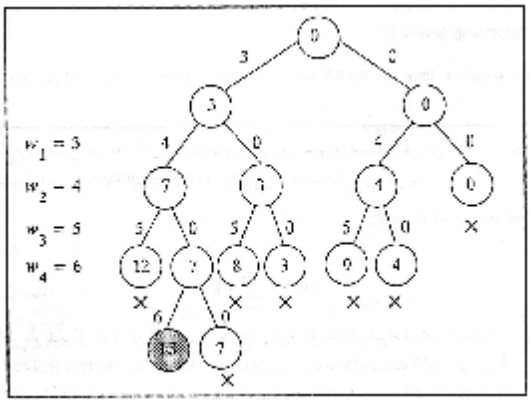
استفاده از درخت فضای حالت

برای حل نمونه هایی با n بزرگ، می توان از درخت فضای حالت استفاده کرد. در این درخت از ریشه به طرف چپ رفته تا عنصری در مجموعه گنجانده شود و به طرف راست رفته تا آن را از مجموعه طرد کرد. هنگامی که بگنجانیم، وزن را روی یال می نویسیم و اگر نگنجانیم، صفر می نویسیم.

مثال: در صورت استفاده از عقبگرد برای n=4 و w=13 ، درخت فضای حالت را نشان دهید.

$w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6$

در هر گره وزن کل قطعات گنجانده شده تا آن نقطه نگهداری می شود. تنها حل در گره سایه دار است و برابر $\{w_1, w_2, w_4\}$است.



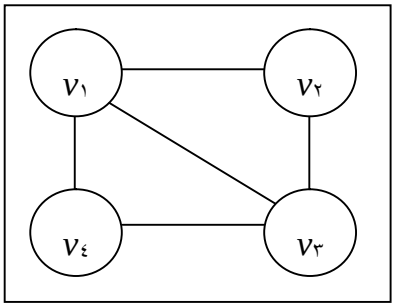
تعداد گره‌هایی از درخت فضای حالت که توسط الگوریتم حاصل جمع زیر مجموعه ها جستوجو می‌شوند، عبارت است از:

$1 + 2 + 2^2 + ... + 2^n = 2^{n+1} - 1$

تذکر: الگورتیم عقبگرد برای مسئله حاصل جمع زیر مجموعه‌ها در پیوست کتاب آورده شده است.

هدف در مسئله رنگ آمیزی m (m-Coloring)، پیدا کردن همهٔ راه های ممکن برای رنگ آمیزی یک گراف بدون جهت، با استفاده از حداکثر m رنگ متفاوت است، به طوری که هیچ دو رأس مجاوری هم رنگ نباشند.

مثال: گراف شکل زیر را در نظر بگیرید.



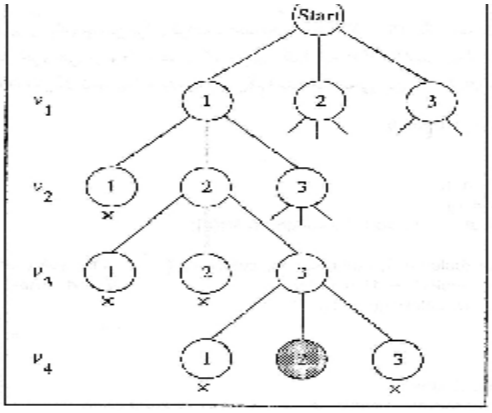
در گراف بالا، حلی برای مسئله رنگ آمیزی 2، وجود ندارد چون با حداکثر 2 رنگ متفاوت، راهی برای رنگ آمیزی رئوس وجود ندارد به قسمی که رئوس مجاور هم رنگ نباشند. اما برای مسئله رنگ آمیزی 3، می توان حلی به صورت مقابل ارائه داد: راس v1 را با رنگ 1، راس v2 را با رنگ 2 ، راس v3 را با رنگ 3 و راس v4 را با رنگ 2 رنگ آمیزی کرد. در کل، شش حل برای مسئله رنگ آمیزی 3 برای این گراف وجود دارد. ولی، این شش حل فقط در شیوهٔ جا به جایی رنگ ها با هم تفاوت دارند. برای مثال یک حل دیگر عبارت است از: رنگ آمیزی v1 با رنگ 2، v2 و v3 با رنگ 1 و v 4 با رنگ 3.

درخت فضای حالت برای مسئله رنگ آمیزی m

در این درخت، هر رنگ ممکن برای رأسv1، در سطح 1 امتحان می شود، هر رنگ ممکن برای رأس v2 در سطح 2 امتحان می شود و به همین ترتیب تا این که هر رنگ ممکن برای رأس vn در سطح n امتحان گردد. هر مسیر از ریشه به برگ یک حل کاندیدا به شمار می رود. با تعیین این که آیا هیچ یک از دو رأس مجاور هم رنگ هستند، در مورد درستی حل کاندیدا تحقیق می کنیم.

تذکر: در بحث زیر، گره به درخت فضای حالت و رأس، به گرافی که رنگ آمیزی می شود، مربوط است.

چون هر گره در صورتی غیر امیدبخش است که یک رأس مجاور به رأسی که در گره رنگ آمیزی می شود قبلاً با همان رنگ، رنگ آمیزی شده باشد می توان در این مسئله عقبگرد کرد. شکل زیر بخشی از درخت فضای حالت هرس شده را نشان می دهد که از به کارگیری راهبرد عقبگرد در مورد رنگ آمیزی گراف مثال قبل با 3 رنگ به دست می آید. عدد درون گره عبارت از شماره رنگ به کار رفته روی رأسی است که در آن گره رنگ می شود. نخستین حل در گره سایه دار یافت می شود. گره های غیر امید بخش با علامت × مشخص شده اند. پس از آن که v1 با رنگ 1 رنگ آمیزی شد، انتخاب رنگ 1 برای v2 غیر امیدبخش است زیرا v1 مجاور v2 است. به طور مشابه، پس از آن که v1، v2 و v3 به ترتیب با رنگ های 1، 2 و 3 رنگ آمیزی شدند، انتخاب رنگ 1 برای v4 غیر امیدبخش می شود، زیرا v1 مجاور v4 است.



تعداد گره ها در درخت فضای حالت برای این الگوریتم برابر است با: $1+m+m^2+...+m^n=\frac{m^{n+1}-1}{m-1}$

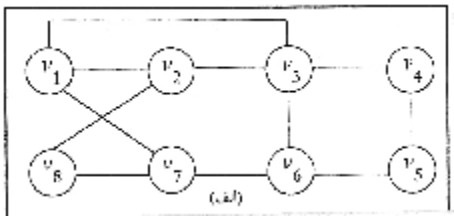
تذکر: به ازای $m \geq 3$، هیچ کس تاکنون الگوریتمی طرح نکرده است که در بدترین حالت مؤثر باشد.

تذکر: الگوریتم عقبگرد برای مسئله رنگ آمیزی m در پیوست کتاب آورده شده است.

مسئله مدارهای هامیلتونی

با داشتن یک گراف متصل و بدون جهت، مدار هامیلتونی (تور)، مسیری است که از یک رأس شروع می شود، هر یک از رئوس گراف را دقیقاً یک بار ملاقات می کند و در رأس شروع، به پایان می رسد.

مثال : گراف زیر حاوی مدار هامیلتونی [v1,v2,v8,v7,v6,v5,v4,v3,v1] می باشد:



درخت فضای حالت

رأس شروع در سطح صفر از درخت قرار داده می شود، (رأس صفرم در مسیر)، در سطح 1، همهٔ رئوس غیر از رأس شروع را به عنوان نخستین رأس پس از رأس شروع در نظر می گیریم. در سطح 2، هر یک از همین رئوس را به عنوان رأس دوم در نظر می گیریم و الی آخر. سرانجام، در سطح $(n-1)$ اُم هر یک از این رئوس را به عنوان رأس $(n-1)$ اُم در نظر می گیریم. با در نظر گرفتن ملاحظات زیر می توانیم در این درخت فضای حالت، عقبگرد کنیم:

1- رأس i اُم از مسیر باید مجاور به رأس $(n-1)$ اُم از مسیر باشد.

2- راس $(n-1)$ اُم باید مجاور به رأس صفرم (رأس شروع) باشد.

3- رأس i اُم نمی تواند یکی از $(i-1)$ رأس باشد.

تذکر: الگوریتم عقبگرد برای مسئله مدارهای هاملیتونی در پیوست کتاب آورده شده است.

- تعداد کره ها در درخت فضای حالت برای این الگوریتم عبارت است از.

$$1+(n-1)+(n-1)^2+\ldots+(n-1)^{n-1}=\frac{(n-1)^n-1}{n-2}$$

این تساوی بسیار بدتر از حالت نمایی است.

مقایسه الگوریتم برنامه نویسی پویا و الگوریتم عقبگرد برای مسئله کوله پشتی صفر و یک

تعداد عناصری که توسط الگوریتم برنامه نویسی پویا برای مسئله کوله پشتی صفر و یک محاسبه می شود در بدترین حالت به $O(\min imum(2^n, nW))$ تعلق داشت. در بدترین حالت، الگوریتم عقبگرد $\theta(2^n)$ گره را چک می کند. به خاطر حد اضافی nW، ممکن است به نظر برسد که الگوریتم برنامه نویسی پویا ارجحیت دارد. ولی در الگوریتم عقبگرد، بدترین حالت، دید خوبی از میزان کارِ صرفه جویی شده، به دست نمی دهد. با در نظر گرفتن ملاحظات فراوان، تحلیل نظری کارایی نسبی در این الگوریتم دشوار است. در چنین مواردی، الگوریتم ها را می توان با اجرای آن ها روی چندین نمونه و دیدن این که کدام الگوریتم معمولاً بهتر عمل می کند، مقایسه نمود. هورویتز این کار را انجام داد و معلوم کرد که الگوریتم عقبگرد معمولاً کارایی بیشتری نسبت به الگوریتم برنامه نویسی پویا دارد.

مجموعه تست

۱- در مسئله کوله پشتی صفر ویک با داشتن n شیئی و ظرفیت W برای کوله پشتی کدام گزینه نادرست است؟

1) کاراترین الگوریتم برای این مسئله، الگوریتم حریصانه با $\theta(n)$ زمان مورد نیاز است.

2) الگوریتم برنامه ریزی پویا در بدترین حالت برای این مسئله $\theta(n.w)$ زمان نیاز دارد.

3) الگوریتم شاخه و قید در بدترین حالت برای این مسئله $\theta(2^n)$ زمان نیاز دارد.

4) الگوریتم برگشت به عقب در بدترین حالت برای این مسئله $\theta(2^n)$ زمان نیاز دارد.

۲- فرض کنید (i,j) و (k,L) دو عنصر در یک صفحه 8×8 باشند (مانند صفحه شطرنج)، کدام گزینه هم‌قطر بودن آنها را تعیین می‌کند؟(تهدید دو مهره فیل)

1) $(i=k)or(j=L)$

2) $(i-k=j-L)or(i-k=L-j)$

3) $i-k=j-L$

4) $(i-K=j-L)or(i-k=L+8-J)$

۳- در ابتدای برنامه روبرو می‌خواهیم درایه‌های یک ماتریس را به طور شطرنجی با صفر و یک پر کنیم. چه دستوری باید در محل نقطه چین نوشته شود؟

```
main( ){
  int    t[۸][۸];
  int    j;
 for ( j=۰ ; j<۶۴ ; j++)
       .....
}
```

1) `t[j/8][j%8]=(j+(j/8))%2`

2) `t[j/8][j%8]=(j%2)`

3) `t[j%8][j/8]=(j+(j/8)%2`

4) `t[j%8][j/8]=(j%2)`

های آنها برابر m است را پیدا و چاپ کند. آرایه x سراسری است و x[i]=۱ یعنی w[i] انتخاب شـده و x[i]=۰ یعنـی w[i] انتخاب نشده. متغیر m نیز سراسری است. الگوریتم به صورت sumofsub($\sum_{i=1}^{n} w[i],0,1$) از بیرون فراخـوانی خواهد شد؟

```
void   sumofsub(s,r,k){
  x[k]=۱;
  if (s+w[k]==m)
     print(x[۱],x[۲],...,x[k]);
   else  if (s+w[k]+w[k+۱] <= m )
           sumofsub(s+w[k] , r-w[k] , k+۱);
   if (( s+r+w[k] >=m ) && (s+w[k+۱] <=m ))
    {
      x[k]=۰;
      sumofsub(s,r-w[k],k+۱);
   }
}
```

1) الگوریتم فقط یک جواب را پیدا می کند.

2) الگوریتم همه جواب ها را به درستی پیدا کرده و چاپ می کند.

3) الگوریتم شرط توقف ندارد و در حلقه نامتناهی فراخوانی بازگشتی خواهد افتاد.

4) الگوریتم یک جواب درست و چندین جواب نادرست پیدا خواهد کرد چرا که آرایه x برای جواب های بعدی مقدار دهی اولیه نمی شود.

پاسخ تشریحی

1-1) کوله پشتی صفر و یک توسط روش حریصانه قابل حل نمی باشد. (کوله پشتی کسری توسط روش حریصانه در زمان $\theta(n \lg n)$ حل می شود.)

2-2) در صورتیکه (i,j) و (k,L) هم‌قطر باشند ، رابطه $i-j=k-L$ یا رابطه $i+j=k+L$ برقرار است. مانند (2،3) با (4،5) و یا (8،4) با(5،7) و یا (2،2) با(7،7) . توجه کنید که رابطه ها را می توان به صورت $i-k=j-L$ و $i-k=L-j$ نیز بیان کرد.

1-3) شماره سطرها به کمک رابطه $j/8$ و شماره ستون ها به کمک رابطه $j\%8$ بدست می‌آید. (j از 0 تا 63 تغییر می‌کند.) مثلاً خانه یازدهم (یعنی$j=10$)، در سطر شماره 1 و ستون شماره 2 واقع است. بنابراین جواب گزینه 1 یا 2 می‌باشد. از طرفی در گزینه 2 عبارت $(J\%2)$، باعث می‌شود که خانه‌ها به صورت ستونی یک در میان با صفر و یک پر شوند و نه به صورت شطرنجی، بنابراین جواب گزینه یک می‌باشد.

2-4) الگوریتم داده شده، همان الگوریتم عقبگرد برای مساله حاصل جمع زیر مجموعه ها می باشد.

فصل ششم: پیچیدگی محاسباتی (مرتب سازی و جست و جو)

پیچیدگی محاسباتی، عبارت از مطالعه تمام الگوریتم‌های امکان پذیر برای حل یک مسئله مفروض است. در تحلیل پیچیدگی محاسباتی کوشش می کنیم تا حد پایینی کارایی همه الگوریتم ها را برای یک مسئله مفروض به دست آوریم.

هدف ما برای یک مسئله مفروض، تعیین حد پایینی $\Omega(f(n))$ و بسط یک الگوریتم $\theta(f(n))$ برای مسئله است. پس از انجام این کار، می دانیم که به جز بهبود بخشیدن به ثابت، نمی‌توانیم الگوریتم را به طریق دیگر بهبود بخشیم.

فرض کنید کسی بخواهد یک میلیارد عنصر را تقریباً بلافاصله در یک برنامه کاربردی شبکه ای مرتب سازی کند. ممکن است او ساعت ها یا حتی سال ها وقت صرف طراحی یک الگوریتم زمانی خطی یا بهتر از آن کند. ممکن است پس از صرف یک عمر تلاش در این راه دریابد بسط چنین الگوریتمی امکان پذیر نیست. دو راه برای پرداختن به این مسئله وجود دارد:

1- کوشش برای بسط الگوریتمی با کارایی بیشتر

2- کوشش برای اثبات آن که بسط چنین الگوریتمی امکان پذیر نیست. اگر چنین چیزی اثبات شود، دیگر باید از به دست آوردن الگوریتمی سریع تر دست کشید.

مثال: مسئله ضرب ماتریس ها به الگوریتمی نیاز دارد که پیچیدگی زمانی آن به $\Omega(n^2)$ تعلق دارد. به عبارتی حد پایینی مسئله ضرب ماتریس‌ها به $\Omega(n^2)$ تعلق دارد. این بدان معنا نیست که ایجاد یک الگوریتم $\theta(n^2)$ برای ضرب ماتریس‌ها حتماً امکان پذیر است، بلکه فقط بدان معناست که ایجاد الگوریتمی بهتر از $\theta(n^2)$ امکان پذیر نیست. زیرا بهترین الگوریتم ما $\theta(n^{2.38})$ و حد پایینی ما $\Omega(n^2)$ است. هنوز ادامه تحقیق روی مسئله ارزش دارد. از یک طرف، می‌توانیم برای یافتن یک الگوریتم موثرتر با استفاده از روش شناسی طراحی الگوریتم‌ها کوشش کنیم و از طرف دیگر می توانیم با استفاده از تحلیل پیچیدگی محاسباتی یک حد پایینی بزرگ تر به دست آوریم. شاید روزی بتوانیم الگوریتمی بسط دهیم که بهتر از $\theta(n^{2.38})$ باشد یا شاید روزی اثبات کنیم که حد پایینی آن بزرگ تر از $\theta(n^2)$ است.

تحلیل پیچیدگی محاسباتی را با مطالعه مسئله مرتب سازی معرفی می‌کنیم. طبقه‌ای از الگوریتم‌های مرتب سازی که به اندازه حد پایینی خوب هستند، شامل همه الگوریتم‌هایی می‌شوند که مرتب سازی را فقط با مقایسه کلیدهای انجام می‌دهند. اگر رکوردها در یک ترتیب دلخواه آرایش یافته باشند، عمل مرتب سازی عبارت از بازآرایی آنها است به قسمی که براساس مقادیر کلیدها مرتب شده باشند. در الگوریتم‌های ما، کلیدها در یک آرایه نگهداری می‌شوند و ما به فیلدهای غیرکلیدی رجوع نمی‌کنیم، ولی فرض می‌شود که این فیلدها به همراه کلید بازآرایی می‌شوند. الگوریتمهایی که مرتب سازی را فقط از طریق مقایسه کلیدها انجام می دهند، می‌توانند دو کلید را مقایسه کنند تا معلوم شود کدام بزرگ تر است و می‌توانند کلیدها را کپی کنند، ولی نمی‌توانند عملیات دیگری روی آن‌ها انجام دهند. الگوریتم‌های مرتب سازیی که تا کنون به آن‌ها برخورده‌ایم در این طبقه قرار می‌گیرند.

در این فصل :

1- الگوریتم‌هایی را مورد بحث قرار خواهیم داد که فقط با مقایسه کلیدها مرتب سازی می‌کنند. به طور مشخص مرتب سازی درجی و مرتب سازی انتخابی را مورد بحث قرار می‌دهیم که دو مورد از الگوریتم های مرتب سازی درجه دوم با بیشترین کارایی هستند.

2- نشان می‌دهیم مادامی که خود را به شرایط مرتب سازی درجی و مرتب سازی انتخابی محدود کنیم، نمی‌توانیم زمان درجه دوم را بهبود بخشیم.

تذکر: مرتب سازی مبنا براساس مقایسه کلیدها مرتب سازی نمی‌کنند.

الگوریتم ها را برحسب تعداد مقایسه های کلیدها و تعداد انتساب‌های رکوردها تحلیل مـی‌کنـیم. بـرای مثـال، در الگـوریتم مرتـب سازی تعویضی، تعویض s[j] , s[i] را می توان به صورت مقابل پیاده سازی نمود:

```
temp=S[i];  S[i]=S[j];   S[j]=temp;
```

یعنی به ازای یک بار تعویض، سه بار انتساب رکوردها انجام می‌شود. تعداد انتساب های رکوردها را تحلیل می‌کنـیم، زیـرا هنگـامی که رکوردها بزرگ هستند، زمان صرف شده برای انتساب یک رکورد کاملاً چشمگیر اسـت. همچنـین مقـدار فضـای اضـافی را کـه الگوریتم‌ها علاوه بر فضای لازم جهت نگهداری ورودی نیاز دارند، تحلیل می کنیم.

> هنگامی که فضای اضافی یک مقدار ثابت باشد، یعنی با n ، تعداد کلیدهایی که قرار است مرتب سازی شوند، افزایش نیابـد، الگوریتم را مرتب سازی درجا (inplace) می‌گویند.

تذکر: فرض می‌کنیم که همواره مرتب سازی به ترتیب غیرنزولی است.

مرتب سازی درجی

الگوریتم مرتب سازی درجی الگوریتمی است که مرتب سازی را با درج رکوردها در یک آرایه مرتب شده موجود مرتـب سـازی مـی کند. فرض کنید کلیدهای موجود در i-1 محل نخست از آرایه نگهداری می شوند و x مقدار کلید موجود در محـل iام باشـد. x را به ترتیب با کلیدهای موجود در محل (i-1) ام، در محل (i-2) ام و غیره مقایسه می کنیم تا این که کلیدی کوچـک تـر از x پیـدا شود. اگر j محلی باشد که این کلید در آن قرار دارد، کلیدهای موجود در محل های (j+1) تا (j-1) را بـه محـل هـای (j+2) تـا i منتقل می کنیم و x را در محل (j+1) ام درج می کنیم. این فرآیند را به ازای i=2 تا i=n تکرار می کنیم.

الگوریتم مرتب سازی درجی

می خواهیم n کلید موجود در آرایه s[1..n] را به ترتیب نزولی مرتب کنیم.

```
void    insertionsort (int  n , keytype s[ ] ){
    index       i,j;  keytype   x;
    for (i=2 ; i<=n ; i++){
        x=s[i];
        j=i-1;
        while(  j>0  &&  s[i]>x ) {
            s[j+1]=s[j];
            j--;
          }
        s[j+1]=x;
    }
}
```

عمل اصلی، مقایسه s[j] با x می باشد و اندازه ورودی، تعداد کلیدهایی که باید مرتب شوند یعنی n است. برای یک i مفروض، مقایسه s[j] با x اغلب هنگامی انجام می شود که j برابر با صفر می شود و حلقه while خاتمه می یابد. با این فرض که شرط دوم در یک عبارت &&، هنگام نادرست بودن شرط اول ارزیابی نمی شود، هنگامی که j برابر صفر باشد، مقایسه s[j] با x صورت نمی پذیرد. پس این مقایسه به ازای یک i مفروض حداکثر (i-1) مرتبه انجام می شود. چون مقدار i از 2 تا n تغییر می کند، تعداد کل مقایسه حداکثر عبارت است از: $\sum_{i=2}^{n}(i-1)=\frac{n(n-1)}{2}$. اگر کلیدها از آغاز به ترتیب غیر صعودی مرتب شده باشند، این حد به دست می آید، یعنی: $W(n)=\frac{n(n-1)}{2}$

تحلیل پیچیدگی زمانی تعداد مقایسه های کلیدها در الگوریتم مرتب سازی درجی در حالت میانگین

برای یک i مفروض، i محل وجود دارد که x را می توان در آنها درج کرد. یعنی x می تواند در محل iام بماند، به محل (i-1) برود، به محل (i-2) برود و غیره. چون پیش از این x را بررسی نکرده ایم یا آن را در الگوریتم به کار نبرده ایم، دلیلی ندارد که فرض کنیم احتمال درج آن در یک محل بیش از احتمال درج آن در سایر محل ها است. بنابراین، به هر یک از i محل نخست احتمال یکسان نسبت داده می شود. این بدان معناست که هر محل دارای یک احتمال 1/i است. هنگامی که x در نخستین محل درج شود، تعداد مقایسه ها (i-1) است، زیرا شرط نخست در عبارت کنترل کننده حلقه while در صورتی که j=0 باشد، نادرست است، یعنی شرط دوم ارزیابی نمی شود.

تذکر: تنها فضایی که با n افزایش می یابد، اندازه آرایه ورودی s است. پس، الگوریتم مرتب سازی درجی، یک مرتب سازی درجا است و فضای اضافی به $\theta(1)$ تعلق دارد.

مقایسه مرتب سازی درجی با مرتب سازی تعویضی

الگوریتم	مقایسه کلید	انتساب رکوردها	استفاده از فضای اضافی
تعویضی	$T(n)=\frac{n^2}{2}$	$W(n)=\frac{3n^2}{2}$	درجا
		$A(n)=\frac{3n^2}{4}$	
درجی	$W(n)=\frac{n^2}{2}$	$W(n)=\frac{n^2}{2}$	درجا
	$A(n)=\frac{n^2}{4}$	$A(n)=\frac{n^2}{4}$	
انتخابی	$T(n)=\frac{n^2}{2}$	$T(n)=3n$	درجا

تذکر: مرتب سازی انتخابی همان مرتب سازی تعویضی با قدری اصلاح است.

1- مرتب سازی درجی بر حسب مقایسه های کلیدها همواره حداقل به خوبی مرتب سازی تعویضی و به طـور میـانگین بهتـر از آن عمـل مـی کند.

2- بر حسب انتساب های رکوردها، مرتب سازی درجی هم در بدترین حالت و هم در حالت میانگین بهتر عمل می کند. چـون هـر دو مرتب سازی درجا هستند.

3- مرتب سازی درجی الگوریتم بهتری است.

الگوریتم مرتب سازی انتخابی

```
void  selectionsort (int n,  keytype  S[ ]){
      index   i, j, smallest;
      for (i=1; i<=n-1; i++) {
         smallest=i;
         for (j=i+1;j<=n;j++)
              if (s[j] < s[smallest])
                   smallest = j;
         exchange  s[i]  and  s[smallest];
       }
}
```

این الگوریتم دارای همان پیچیدگی زمانی مرتب سازی تعویضی بر حسب مقایسه کلیدها است. ولی، انتساب کلیـدها دارای تفـاوت چشمگیری است. به جای آن که همانند مرتب سازی تعویضی، با هر بار کوچک تر بودن s[j] از s[i]، این دو با هم تعـویض شـوند، در مرتب سازی انتخابی، صرفاً اندیس کوچک ترین کلید فعلی در میان کلیدهای موجود در محل های iام یا nام را مد نظر قرار می دهیم. پس از تعیین آن رکورد، آن را با رکورد موجود در محل iام تعویض می کنیم. به ایـن ترتیـب، کوچـک تـرین کلیـد پـس از نخستین گذر از حلقه for i تعداد تعویض های انجام شده دقیقا به (n-1) می رسد. چون سه انتساب به ازای هر تعویض مورد نیـاز است، پیچیدگی زمانی تعداد انتساب رکوردها توسط مرتب سازی انتخابی در حالت معمول عبارت است از: $T(n)=3(n-1)$

تعداد انتساب های رکوردها برای مرتب سازی تعویضی در حالت میانگین، حدود $3n^2/4$ بود. مرتب سازی تعویضـی گـاه بهتـر از مرتـب سازی انتخابی عمل می کند. برای مثال، اگر رکوردها از قبل مرتب باشند، مرتب سازی تعویضی انتساب رکوردها را انجام نمی دهد.

بر حسب مقایسه کلیدها، مرتب سازی درجی همواره حداقل به خوبی مرتب سازی انتخابی و به طور میانگین بهتر از آن عمـل مـی کند. ولی پیچیدگی زمانی مرتب سازی انتخابی بر حسب انتسابات رکوردها خطی است، در حالی کـه بـرای مرتـب سـازی درجـی، درجه دوم است. بنابراین اگر n بزرگ باشد و رکوردها هم بزرگ باشند (به طوری که زمـان انتسـاب آنهـا چشـمگیر باشـد) مرتـب سازی انتخابی باید بهتر عمل کند.(هنگامی که n بزرگ باشد زمان خطی بسیار سریع تر از زمان درجه دوم است.)

هر الگوریتم مرتب سازی که رکوردها را به ترتیب انتخاب کند و آن ها را در موقعیت مناسب خود قرار دهـد، مرتـب سـازی انتخابی نام دارد. این بدان معناست که الگوریتم تعویضی نیز یک الگوریتم انتخابی است.

در عمل، هیچ کدام از این الگوریتم ها برای مقادیر بزرگ n مناسب نیستند، زیرا زمان همه آنها در بدترین حالت درجه دوم است و مادامی که خود را به الگوریتم هایی از این طبقه محدود کنیم، تا آنجا که به مقایسه کلیدها مربوط می شود، بهبود بخشیدن به زمان درجه دوم غیر ممکن است.

حدود پایین برای الگوریتم هایی که در هر مقایسه یک وارونگی را حذف می کنند

مرتب سازی درجی پس از هر بار مقایسه، یا کاری انجام نمی دهد یا کلید موجود در محل iام را به محل (i+1) ام منتقل می کند. با منتقل کردن کلید موجود در محل jام به یک محل بالاتر، این واقعیت را تصحیح کرده ایم که x باید پیش از آن کلید بیاید. ولی، این تنها کاری است که انجام داده ایم. نشان می دهیم همه الگوریتم های مرتب سازی که فقط از طریق مقایسه کلیدها مرتب سازی را انجام می دهند و چنین مقدار محدودی از باز آرایی را پس از هر بار مقایسه انجام می دهند، حداقل به زمان درجه دوم نیاز دارند. فرض کنیم که کلیدها صرفاً اعداد مثبت و صحیح n ، ... ، 2، 1 هستند، زیرا می توانیم کوچک ترین کلید را با 1، دومی را با 2 و ... جایگزین کنیم.

مثال: ورودی الفبایی [R, C, D] را داریم. می توانیم 1 را به C، 2 را به D و 3 را به R ربط دهیم و ورودی معادل آن، {3،1،2} را به دست آوریم. هر الگوریتمی که این اعداد صحیح را فقط با مقایسه کلیدها مرتب کند برای مرتب سازی سه کاراکتر هم به همان تعداد مقایسه نیاز دارد. یک جایگشت از n عدد صحیح مثبت را می توان به عنوان ترتیبی از آن اعداد صحیح در نظر گرفت. n! ترتیب متفاوت از این اعداد وجود دارد:

[3،2،1] [3،1،2] [2،3،1] [2،1،3] [1،3،2] [1،2،3]

یعنی n! ورودی متفاوت (برای یک الگوریتم مرتب سازی) حاوی n کلید متمایز وجود دارد. این شش جایگشت، ورودی های متفاوتی به اندازه 3 هسند. جایگشت را با [k1,k2,...,kn] نشان می دهیم. یعنی ki یک عدد صحیح در موقعیت i است. برای مثال، برای جایگشت [3،1،2] داریم: $k_1 = 3, k_2 = 1, k_3 = 2$

وارونگی

وارونگی در یک جایگشت، زوج (k_i, k_j) است به طوری که i<j و $k_i > k_j$ باشد.

مثال: جایگشت [3،2،4،1،6،5] حاوی وارونگی های (3،2) ، (3،1) ، (2،1) ، (4،1) و (6،5) است.

یک جایگشت، وارونگی نخواهد داشت اگر و فقط اگر دارای ترتیب مرتب [1،2،...،n] باشد. یعنی عمل مرتب سازی n کلید متمایز عبارت از حذف همه وارونگی ها در یک جایگشت است.

قضیه: هر الگوریتمی که n کلید متمایز را فقط از طریق مقایسه کلیدها انجام دهد و پس از هر بار مقایسه، حداکثر یک وارونگی را حذف کند، باید در بدترین حالت حداقل n(n-1)/2 مقایسه و به طور میانگین حداقل n(n-1)/4 مقایسه روی کلیدها انجام دهد.

1)/4 است. چون فرض کردیم که الگوریتم حداکثر یک وارونگی را پس از هر مقایسه انجام می دهد، به طور میانگین باید حداقل این تعداد مقایسه را برای حذف همه وارونگی ها و در نتیجه مرتب سازی ورودی انجام دهیم.

مرتب سازی درجی حداکثر وارونگی شامل s[j] و x را پس از هر مقایسه حذف می کند و در نتیجه می توان قضیه بالا را در مورد آن به کار برد.

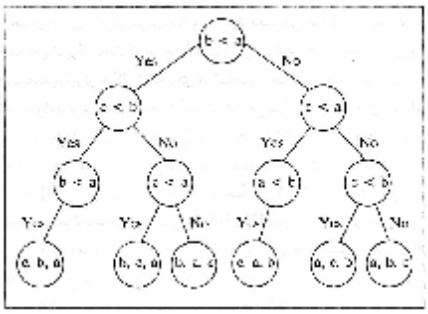
الگوریتم مرتب سازی تعویضی

```
void   exchangesort (int n, keytype s[ ]){
        index  i,j;
        for (i=1 ; i<=n-1 ; i++)
          for ( j=i+1 ; j<=n ; j++)
              if (s[j]<s[i])
                  exchange s[i] and s[j];
}
```

فرض کنید که در حال حاضر آرایه s حاوی جایگشت [2،4،3،1] است و داریم 2 را با 1 مقایسه می کنیم. پس از مقایسه، 2 با 1 تعویض می شود و در نتیجه وارونگی های (2،1) ، (4،1) و (3،1) حذف می شوند. ولی وارونگی های (4،2) و (3،2) اضافه شده اند و کاهش خالص در وارونگی فقط یکی است. این مثال، نتیجه ای کلی را نشان می دهد که مرتب سازی تعویضی همواره دارای کاهش خالص حداکثر یک وارونگی پس از هر مقایسه است.

از آنجا که پیچیدگی زمانی مرتب سازی درجی در بدترین حالت، n(n-1)/2 و در حالت میانگین، حدود $n^2/4$ است، آنقدر خوب هست که بتوانیم امیدوار باشیم با الگوریتم هایی کار کنیم که مرتب سازی را فقط با مقایسه کلیدها انجام داده پس از هر مقایسه حداکثر یک وارونگی را حذف می کند.

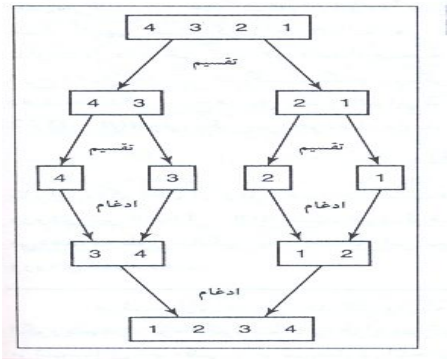
درخت تصمیم گیری متناظر با مرتب سازی تعویضی در هنگام مرتب سازی سه کلید :



مرتب سازی ادغامی

مرتب سازی ادغامی را در گذشته معرفی کردیم. می خواهیم نشان دهیم که گاهی این مرتب سازی پس از هر مقایسه بیش از یک وارونگی را حذف می کند. الگوریتم هایی که حداکثر یک وارونگی پس از هر مقایسه انجام می دهد، حداقل n(n-1)/2 مقایسه را در هنگامی که ورودی به ترتیب عکس باشد، انجام می دهند.

پس از آن که 3 و 1 مقایسه شدند، 1 در نخستین محل آرایه قرار می گیرد و در نتیجه وارونگی های (3،1) و (4،1) حذف می شوند. پس از آنکه 3 و 2 مقایسه شدند، 2 در دومین محل آرایه قرار می گیرد و در نتیجه وارونگی های (3،2) و (4،2) حذف می شوند.



پیچیدگی زمانی تعداد مقایسه های کلیدها در مرتب سازی ادغامی در بدترین حالت، وقتی که n توانی از 2 است به صورت زیر است و به طور کلی به $\theta(n \lg n)$ تعلق دارد: $W(n) = n \lg n - (n-1)$

تذکر : با توسعه یک الگوریتم مرتب سازی که پس از هر مقایسه بیش از یک وارونگی را حذف می کند، کارایی به طرز چشمگیری افزایش یافته است. الگوریتم های $\theta(n \lg n)$ می توانند از عهده ورودی های بسیار بزرگ برآیند.

پیچیدگی زمانی تعداد مقایسه های رکوردها در مرتب سازی ادغامی در حالت معمول تقریباً برابر است با $T(n) \approx 2n \lg n$

تحلیل استفاده از فضای اضافی برای مرتب سازی ادغامی

حتی نسخه بهبود یافته مرتب سازی ادغامی نیاز به یک آرایه اضافی کامل به اندازه n دارد. به علاوه، زمانی که الگوریتم، در حال مرتب سازی نخستین زیر آرایه است. مقادیر mid , mid+1 , low و high باید در پشته رکوردهای فعالیت نگهداری شوند. از آنجا که آرایه همواره از وسط تقسیم می شود، این پشته تا عمق $\lceil \lg 2 \rceil$ رشد می کند. فضای لازم برای آرایه اضافی رکوردها غالب است و این بدان معناست که در حالت معمول، استفاده از فضای اضافی به $\theta(n)$ رکورد تعلق دارد.

تذکر: روشهای بهبود مرتب سازی ادغامی در پیوست آورده شده است.

مرتب سازی سریع

پیچیدگی زمانی این الگوریتم در بدترین حالت از درجه دوم است، ولی پیچیدگی زمانی تعداد مقایسه های کلیدها در حالت میانگین عبارت است از: $A(n) \approx 1{,}38(n+1) \lg n$ که چندان بدتر از مرتب سازی ادغامی نیست. مزیت مرتب سازی سریع نسبت به ادغامی این است که نیازی به آرایه اضافی ندارد. ولی هنوز هم یک مرتب سازی درجا نیست زیرا در حالی که الگوریتم، زیر آرایه نخست را مرتب سازی می کند، نخستین و آخرین اندیس زیر آرایه دیگر باید در پشته رکوردهای فعالیت نگهداری شوند. در مرتب سازی سریع بر خلاف مرتب سازی ادغامی، تضمینی وجود ندارد که آرایه همواره از وسط تقسیم شود. در بدترین حالت، ممکن است partition مکرراً آرایه را به زیر آرایه تهی در سمت چپ (یا راست) و زیرآرایه ای با یک عنصر کمتر در سمت راست

در بدترین حالت به $\theta(n)$ تعلق دارد. می توان مرتب سازی سریع را چنان اصلاح کرد که استفاده از فضای اضافی حداکثر در حدود $\lg n$ باشد.

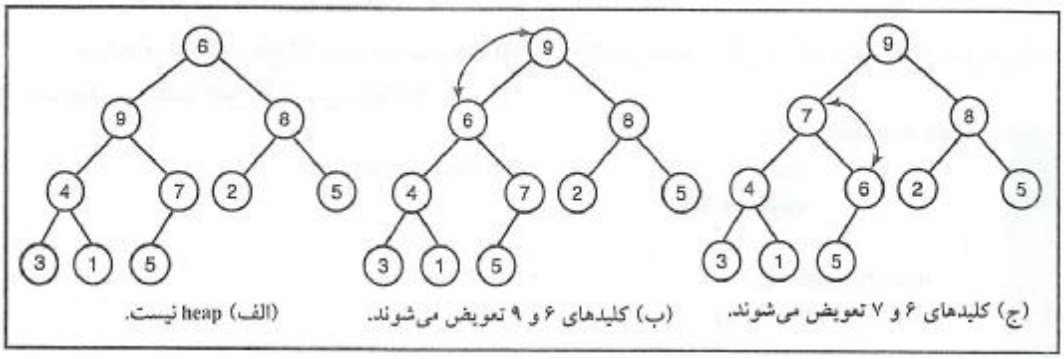
تعداد میانگین تعویض های انجام شده توسط مرتب سازی سریع حدود $0.69(n+1)\lg n$ است. به فرض آن که هر تعویض مستلزم سه انتساب باشد، پیچیدگی زمانی تعداد انتساب های رکوردهای انجام شده توسط مرتب سازی سریع حدوداً عبارت است از:

$$A(n) \approx 2.07(n+1)\lg n$$

تذکر: روش های کاهش استفاده از فضای اضافی در مرتب سازی سریع در پیوست آورده شده است.

مرتب سازی heap

در درخت heap ، مقدار هر گره بزرگتر یا مساوی مقدار فرزندانش است. اگر به طور مکرر کلید نگهداری شده در ریشه را حذف کنیم و در عین حال ویژگی heap را حفظ کنیم، کلیدها به ترتیب غیر صعودی حذف خواهند شد. اگر برای حذف آن ها، آن ها را در آرایه ای با شروع از محل n ام قرار دهیم و به طرف پایین یعنی محل اول پیش برویم، به ترتیبی غیر نزولی در آرایه مرتب خواهند شد. پس از حذف کلیدی که در ریشه قرار دارد، می توانیم با قرار دادن کلید واقع در گره پایینی به جای کلید واقع در ریشه، حذف گره پایینی و فراخوانی روال siftdown ، ویژگی heap را حفظ کنیم. روال siftdown ، کلیدی که هم اکنون در ریشه قرار دارد را به پایین heap می لغزاند تا ویژگی heap حفظ شود(منظور از گره پایینی برگی است که در منتها الیه سمت راست قرار دارد). این جابه جایی با مقایسه کلید واقع در ریشه با بزرگ ترین کلید فرزندان ریشه صورت می پذیرد. اگر ریشه بزرگ تر باشد، کلیدها تعویض می‌شوند. این فرآیند تا پایین درخت تکرار می شود تا کلید واقع در یک گره کوچکتر از بزرگ ترین فرزندان آن گره نباشد. در شکل الف، چون مقدار گره ریشه از فرزندش کوچکتر است، ویژگی heap ندارد. با تعویض جای کلیدهای 6 و 9 شکل ب حاصل می شود و سپس کلیدهای 7 و 6 تعویض می شوند و شکل ج حاصل می شود. این درخت ویژگی heap را دارد.



مرتب سازی heap ، یک الگوریتم $\theta(n\lg n)$ و درجا است.

تذکر: الگوریتم siftdown در پیوست آورده شده است.

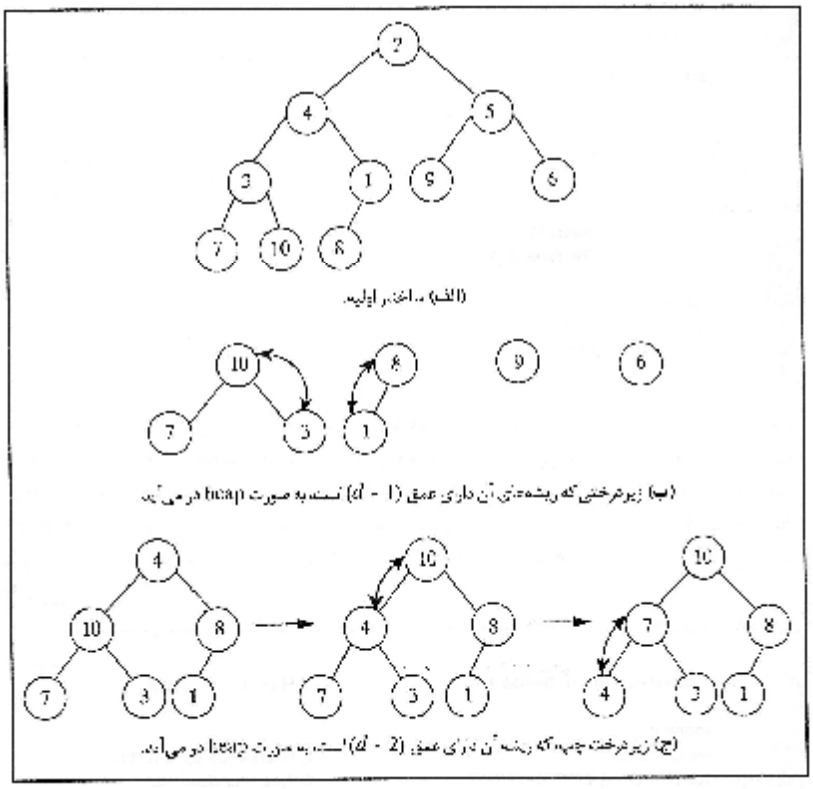
دارای ویژگی heap نیست.

می‌توانیم درخت را با فراخوانی‌های مکرر siftdown با اجرای عملیات زیر، به یک heap تبدیل کنیم:

1- تبدیل همه زیر درخت‌هایی که ریشه‌های آن دارای عمق (d-1) است، به heap .

2- تبدیل همه زیر درخت‌هایی که ریشه آن‌ها دارای عمق (d-2) است به heap.

3- تبدیل کل درخت (تنها زیر درختی که ریشه آن دارای عمق 0 است) به heap.



الگوریتم مرتب سازیheap

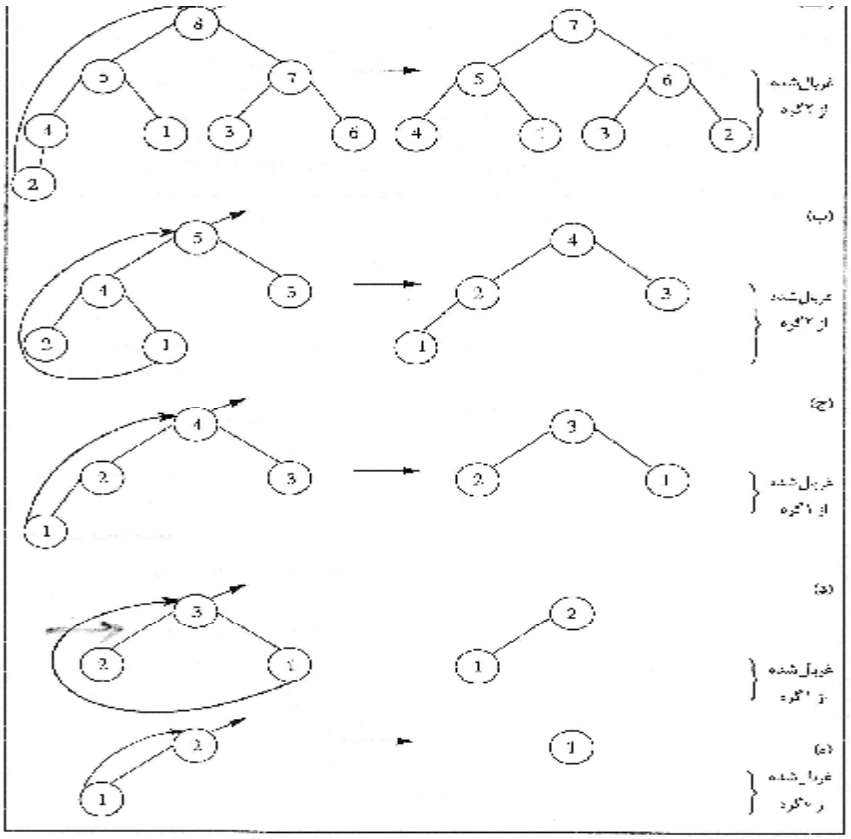
کلیدها از قبل در یک درخت دودویی اساساً کامل H آرایش یافته اند:

```
void heapsort ( int   n, Heap  H, Keytype  S[ ]){
   makeheap(n,H);
   removekeys(n, H, S);
}
```

تذکر: الگوریتم های Makeheap و removekeys در پیوست کتاب آورده شده است.

الگوریتم مرتب سازی heap ، یک مرتب سازی درجا نیست چون برایheap به فضای اضافی نیاز داریم. البته مـی تـوانheap را با استفاده از یک آرایه پیاده سازی کرد. همان آرایه ای که ورودی را نگهداری می‌کند می‌تواند برای پیاده سازی heap نیز بـه کـار رود و هرگز یک محل از آرایه به طور همزمان برای بیش از یک منظور لازم نیست.

شکل زیر نحوه حذف کلیدها از یک heap با هشت کلید را نشان می دهد:



پیاده سازی مرتب سازی heap

می‌توان یک درخت دودویی اساساً کامل را با نگهداری ریشه در نخستین محل از یک آرایه، فرزند چپ آن در دومین محل و فرزند راست آن در سومین محل، فرزندان چپ و راست از فرزند چپ را در چهارمین و پنجمین محل و... در آن آرایه نگهداری کرد.

تذکر: اندیس فرزند چپ یک گره، دو برابر اندیس آن گره و اندیس فرزند راست یک گره یکی بیشتر از دو برابر اندیس آن گره است.

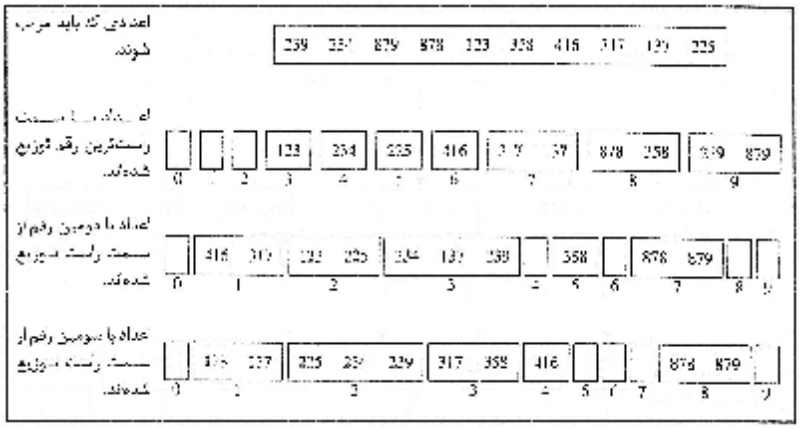
در شبه کد سطح بالای مرتب سازی heap ، لازم بود که کلیدها از آغاز در یک درخت دودویی اساساً کامل قرار گیرند. اگر کلیدها را در یک آرایه به ترتیبی دلخواه قرار دهیم، در یک درخت دودویی اساساً کامل سازماندهی خواهند شد. پس از آن که درخت دودویی اساساً کامل به heap تبدیل شد، کلیدها با شروع از محل nام آرایه و رفتن به طرف پایین، یعنی محل یکم، از آرایه حذف می‌شوند. چون این کلیدها به ترتیب در آرایه خروجی قرارداده می‌شوند، می‌توانیم از H.S به عنوان آرایه خروجی استفاده کنیم، بدون این که امکان نوشته شدن کلیدها روی یکدیگر وجود داشته باشد. این الگوریتم درجا است. این الگوریتم به صورت زیر است:

```
void   heapsort ( int  n , heap&  H){
       makeheap ( n , H);
       removekeys (n , H , H.S);
}
```

ورودی این تابع، آرایه‌ای از n کلید که در پیاده سازی heap در آرایه H نگهداری می‌شوند و خروجی، کلیدهای مرتب شده به صورت غیر نزولی در آرایه H.S می باشد.

در مرتب سازی heap داریم: $W(n) \in \theta(n \lg n)$ و $A(n) \approx 2n \lg n$

هر الگوریتم مرتب سازی که از طریق مقایسه کلیدها عمل می کند، نمی تواند از $\theta(n \lg n)$ بهتر شود. در روش مرتب سازی مبنایی، در مبنای 10 ، ده گروه از 0 تا 9 در نظر می گیریم. ارقام را از راست به چپ بازرسی کرده و هر کلید را در گروه متناظر با رقمی که در حال حاضر بازرسی می شود، قرار می دهیم. بعد از پایان گذر اول، اعداد بر حسب اولین رقم سمت راست مرتب شده اند. این روال در شکل زیر نشان داده شده است.



با هر بار گذر، اگر دو کلید باید در یک گروه قرار گیرند، کلیدی که از گروه واقع در منتها الیه سمت چپ در گذر قبلی می آید، در طرف چپ کلید دیگر قرار می گیرد. به عنوان مثال بعد از اولین گذر، کلید 416 در گروه واقع در چپ کلید 317 قرار دارد. بنابراین، هنگامی که در دومین گذر، هر دو آن ها در گروه نخست قرار بگیرند، کلید 416 در طرف چپ کلید 317 قرار می گیرد. ولی در گذر سوم، کلید 416 به طرف راست کلید 317 می رود، چون در گروه چهارم قرار داده می شود، سپس کلید 317 در گروه سوم قرار داده می شود.

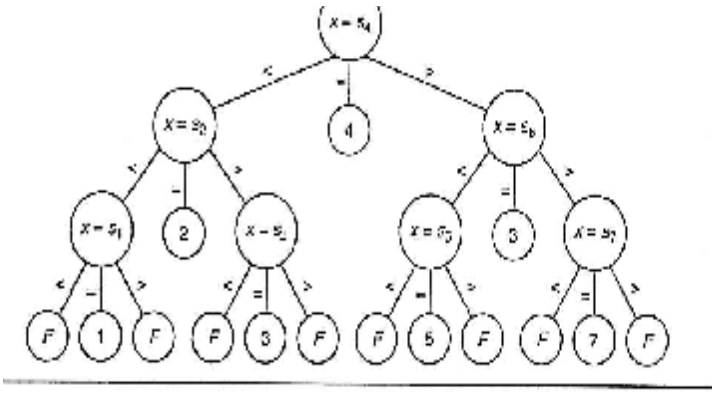
در مرتب سازی مبنایی داریم: $T(d,n) = d \times (n+10) \in \theta(d \times n)$

(که n ، تعداد اعداد صحیح و d، تعداد ارقام دهدهی در هر عدد صحیح است.)

پیچیدگی محاسباتی(جست و جو)

مسئله جست‌وجوی یک کلید را می‌توان به این صورت توصیف کرد: آرایه S حاوی n کلید و کلید x مفروضند؛ اندیس i را بیابید به طوری که $x=S[i]$ باشد؛ اگر x با هیچ یک از کلیدها مساوی نبود، شکست را گزارش کنید. جست‌وجوی دودویی برای حل این مسئله، هنگامی که آرایه مرتب باشد، بسیار کارآمد است و پیچیدگی زمانی آن در بدترین حالت، $\lfloor \lg n \rfloor + 1$ است.

تا هنگامی که خود را به الگوریتم‌هایی محدود کنیم که جست‌وجو را فقط از طریق مقایسه کلیدها انجام می‌دهند، بهبودی امکان‌پذیر نیست. می‌توانیم هر الگوریم قطعی که کلید x را در آرایه n کلیدی جست‌وجو می‌کند، با یک درخت تصمیم‌گیری مرتبط کنیم. شکل زیر یک درخت تصمیم‌گیری متناظر با جست‌وجوی دودویی در میان هفت کلید را نشان می‌دهد.



شکل ۸-۱ درخت تصمیم‌گیری متناظر با جست‌وجوی دودویی هنگامی‌که هفت کلید وجود دارد.

در این درخت، هر گره بزرگ، مقایسه‌ای از یک عنصر آرایه با کلید x را نشان می‌دهد و هر گره کوچک (برگ) حاوی نتیجهٔ گزارش شده است. هنگامی که x در آرایه باشد، اندیس عنصری را که مساوی کلید x باشد و هنگامی که x در آرایه نباشد، F را بـه نشـانه شکست گزارش می‌کنیم.

هر برگ از درخت تصمیم‌گیری برای جست‌وجوی کلید x در میان n کلید، نقطه‌ای را نشان می‌دهـد کـه در آن، الگـوریتم متوقـف شده اندیس i را گزارش می‌کند، به طوری که $x = s_i$ باشد، یا شکست را گزارش می‌کند. هر گره داخلی نشانگر یک مقایسه اسـت. درخت تصمیم‌گیری برای جست‌وجوی کلید x در میان n کلید را معتبر می‌گویند اگر به ازای هر نتیجهٔ ممکن، مسیری از ریشه به یک برگ وجود داشته باشد که آن نتیجه را گزارش کند. یعنی به ازای هـر $1 \le i \le n$، مسـیرهایی بـرای $x = s_i$ موجـود باشـد و همچنین مسیری باشد که به شکست منجر شود.

- هر الگوریتم قطعی که کلید x را در میان n کلید متمایز، فقط با مقایسه کلیدها جست‌وجو می‌کند، باید در بـدترین حالـت، حداقل $\lfloor \lg n \rfloor + 1$ مقایسه کلیدها را انجام دهد.
- درخت حاوی گره‌های مقایسه در درخت تصمیم‌گیری معتبر و هرس شدهٔ متنـاظر بـا جسـت‌وجـوی دودویـی، یـک درخـت دودویی تقریباً کامل است.

قضیه: از میان الگوریتم‌های قطعی که کلید x را فقط با مقایسه کلیدها در میان n کلید جست‌وجو می‌کنند، اگر فرض کنیم کـه x در آرایه است و همهٔ محل‌های آرایه به یک میزان محتمل باشند، جست‌وجوی دودویی در کارایی حالت میانگین خود بهینـه اسـت. بنابراین، هر الگوریتمی از این نوع، باید حداقل $\lfloor \lg n \rfloor - 1$ مقایسه کلیدها را انجام دهد.

الگوریتم یافتن بزرگ ترین و کوچک ترین کلیدها

استفاده از این الگوریتم بهتر از یافتن بزرگترین و کوچک ترین کلیدها به طور مستقل است، زیرا بـرای برخـی ورودی هـا، مقایسـه S[i] با large به ازای همه مقادیر i انجام نمی شود. بنابراین، کارایی الگوریتم در حالت معمول را بهبود بخشیده ایـم. ولـی هرگـاه S[1] کوچک ترین کلید باشد، آن مقایسه برای همه مقادیر i انجام می شود. بنابراین، تعداد مقایسه های کلیدها در بدترین حالـت عبارت می شود از: W(n)=2(n-1)، که دقیقا برابر با تعداد مقایسه های انجام شده در حالتی است که کوچک ترین و بزرگ ترین

تر را بیابیم. این کار می توان با حدود n/2 مقایسه و بزرگ ترین همه کلیدهای بزرگتر را با n/2 مقایسه دیگر یافت. به این ترتیب بزرگ ترین و کوچک ترین کلید ها را با حدود 3n/2 مقایسه به دست می آوریم.

نمی توان این کارایی را بهبود بخشید. از آنجا که درخت تصمیم گیری برای مسئله انتخاب، خوب عمل نمی کند،(یک نتیجه ممکن است در بیش از یک برگ لحاظ شود)، برای نشان دادن این موضوع از درخت تصمیم گیری نمی توان استفاده کرد.

قضیه: هر الگوریتمی قطعی که بتواند کوچک ترین و بزرگ ترین کلید ها را در میان n کلید از ورودی ممکن، تنها با مقایسه کلید ها بیابد، باید در بدترین حالت حداقل تعداد مقایسه روی کلید ها انجام می دهد برابر است با:

اگر n زوج باشد : $\frac{3n}{2}-2$ و اگر n فرد باشد : $\frac{3n}{2}-\frac{3}{2}$

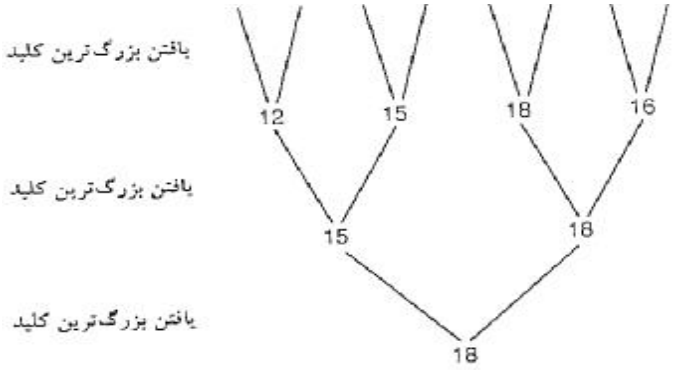
یافتن بزرگ ترین کلید دوم

برای یافتن بزرگ ترین کلید دوم، می توان از الگوریتم "یافتن بزرگ ترین کلید" برای یافتن بزرگ ترین کلید با n-1 مقایسه، حذف آن کلید و سپس استفاده دوباره از این الگوریتم برای یافتن بزرگ ترین کلید باقیمانده با n-2 مقایسه استفاده کرد. بنابراین، می توان بزرگ ترین کلید دوم را با 2n-3 مقایسه بیابیم. بسیاری از مقایسه های انجام شده در هنگام یافتن بزرگ ترین کلید را می توان برای حذف کلیدها از رقابت برای یافتن دومین کلید بزرگ به کار گرفت. یعنی، هر کلیدی که به کلیدی غیر از بزرگترین کلید ببازد، نمی تواند بزرگ ترین کلید دوم باشد. روش تورنمنت از همین واقعیت استفاده می کند.

برای سهولت، فرض کنید اعداد متمایز بوده و n توانی از 2 است. همانند تورنمنت، کلیدها را جفت جفت مقایسه می کنیم تا در نهایت یک دور باقی بماند. اگر هشت کلید داشته باشیم، در دور اول، چهار مقایسه، دور دوم، دو مقایسه و دور آخر یک مقایسه خواهیم داشت. برنده دور آخر، بزرگترین کلید است.

تذکر: روش تورنمنت مستقیما هنگامی کاربرد دارد که n توانی از 2 باشد. در غیر اینصورت می توان تعداد عناصر کافی اضافه کرد تا اندازه آرایه توانی از 2 شود. برای مثال اگر اندازه آرایه 53 باشد، 11 عنصر، هر یک به مقدار $-\infty$ به انتهای آرایه اضافه می کنیم تا آرایه درست 64 عنصر داشته باشد.

گرچه برنده در دور آخر، بزرگ ترین کلید است، بازنده در این دور الزاما بزرگ ترین کلید دوم نیست. در شکل زیر، بزرگ ترین کلید دوم (16) در دور دوم به بزرگ ترین کلید (18) می بازد و حذف می شود. این یکی از مشکلات تورنمنت های واقعی است، زیرا آن دو تیمی که بهترین هستند الزاما در بازی نهایی رو در روی هم قرار نمی گیرند.



برای یافتن بزرگ ترین کلید دوم، می توانیم همه کلیدهایی را که به بزرگ ترین کلید باخته اند، مد نظر قرار داد و سپس بزرگ ترین آن ها را یافت. ولی وقتی که از قبل نمی دانیم بزرگ ترین کلید کدام است، چطور می توانیم آن کلیدها را مدنظر داشته باشیم؟ این کار را می توان با نگهداری لیست های پیوندی کلیدها، یکی به ازای هر کلید، انجام داد. پس از آن که کلیدی یک مقایسه را باخت، به لیست پیوندی کلید برنده اضافه می شود.

اگر n توانی از 2 باشد، در دور نخست n/2 مقایسه، دور دوم $n/2^2$ ... و دور آخر $n/2^{\lg n}$ مقایسه خواهیم داشت. تعداد کل مقایسه‌های لازم برای یافتن بزرگ‌ترین کلید دوم عبارت است از:

$$T(n) = n - 1 + \lg n - 1 = n + \lg n - 2$$

برای هر n مفروض داریم:

$$T(n) = n + \lceil \lg n \rceil - 2$$

تذکر: این کارایی بهتر از دو بار استفاده از الگوریتم برای یافتن بزرگترین کلید دوم استفاده شود. (در آن شیوه $2n - 3$ مقایسه لازم بود.)

قضیه : هر الگوریتم قطعی که بتواند بزرگ‌ترین کلید دوم را در هر ورودی ممکن، تنها با مقایسه کلیدها بیابد، باید در بدترین حالت، حداقل تعداد مقایسه‌هایی که روی کلید انجام دهد برابر است با : $n + \lceil \lg n \rceil - 2$.

در بدترین حالت، حداقل n–1 مقایسه برای یافتن بزرگ‌ترین کلید و حداکثر $n + \lceil \lg n \rceil - 2$ مقایسه برای یافتن بزرگ‌ترین کلید دوم لازم است. هر الگوریتمی که بزرگ‌ترین کلید دوم را بیابد، باید بزرگ‌ترین کلید را هم بیابد، زیرا برای دانستن آن که کلیدی بزرگ‌ترین کلید دوم است، باید بدانیم که یک مقایسه را باخته است. آن باخت هم باید به بزرگ‌ترین کلید باشد. بنابراین، یافتن بزرگ‌ترین کلید دوم، دشوارتر است.

یافتن کوچک‌ترین کلید k ام

برای یافتن کوچک‌ترین کلید k ام، می توان کلیدها را مرتب کرده و سپس کلید k ام را برگرداند. این کار در زمان $\theta(n \lg n)$ انجام پذیر است. اگر از الگوریتم زیر استفاده کنیم، به تعداد مقایسه های کمتری نیاز داریم و در حالت میانگین یک تابع خطی است، یعنی: $A(n) \in \theta(n)$.

تذکر: الگوریتم یافتن کوچکترین و بزرگترین کلیدها با جفت کردن کلیدها در پیوست آورده شده است.

تذکر: الگوریتم انتخاب در پیوست آورده شده است.

حد پایین برای یافتن k امین کوچک‌ترین کلید در مجموعه n کلیدی (برای $k>1$) به صورت زیر است:

$$n + (k-1)\left\lceil \lg\left(\frac{n}{k-1}\right) \right\rceil - k$$

مجموعه تست

۱- دنباله ای از اعداد صحیح به تعداد n وجود دارد. اگر برخی از این اعداد تکراری باشند و در کل k عدد غیر تکراری وجود داشته باشد، آنگاه با چه مرتبه زمانی مناسبی می توان این دنباله را مرتب کرد؟

1) $O(n \log n)$ 2) $O(n \log k)$ 3) $O(k \log n)$ 4) $O(k \log k)$

۲ - در الگوریتم مرتب سازی آرایه ی A با n عنصر فرض کنید b>۱ یک عدد ثابت است. همچنین فرض کنید که هزینه ی مقایسه ی دو عنصر A[i] و A[j]، یا تعویض آن ها ، اگر $|i-j| \le b$ برابر صفر(خیلی کم) و در غیر اینصورت برابر ۱ (خیلی زیاد) است. توجه کنید که با این فرض، هزینه ی مرتب سازی درجی، حبابی برابر O(۱) می شود. چون فقط عناصر مجاور را مقایسه و تعویض می کنند. با این فرض هزینه ی مرتب سازی ادغامی A (merge Sort) در بدترین حالت چقدر است؟ (بدیهی است که اگر T(n) زمان اجرا باشد داریم: n<b و T(n)=۱)

1) $O(n \lg(n/b))$ 2) $O(n/b \lg(n/b))$ 3) $O(n \lg n)$ 4) $O(n \lg n)$

۳- می دانیم که هزینه ی الگوریتم مرتب سازی درجی(Insertion Sort) برای مرتب سازی یک آرایه ی A با n عنصر متناسب با تعداد وارونگی(inversion) های عناصر آن آرایه است. زوج (i,j) را یک عدد وارونگی می گوییم اگر i<j و A[i]>A[j] . با فرض احتمال این که یک زوج اندیس دلخواه از A یک وارونگی باشد برابر $\frac{1}{2}$ است، میانگین تعداد وارونگی های یک آرایه ی A با عناصر متمایز چقدر است؟

1) $\frac{n^2-n}{2}$ 2) $\frac{n^2}{2}$ 3) $\frac{n^2}{4}$ 4) $\frac{n^2-n}{4}$

۴- یک الگوریتم مرتب سازی بر مبنای درخت تصمیم گیری، حداقل دارای چه تعداد مقایسه خواهد بود؟ فرض کنید

$(a = \log_2^{n!}, b = n \log n, c = \log n =, d = n)$

1) a 2) a,b 3) a,c 4) b,d

۵- فرض کنید یک درخت تصمیم گیری دودویی برای ادغام دو لیست مرتب زیر داریم. در درخت، گره های میانی نشان دهنده مقایسه و گره های برگ نشان دهنده ترتیب لیست مرتب نهایی است. حد پایین تعداد گره های برگ این درخت چند است؟

$list1 : x_1, x_2$
$list2 : y_1, y_2, y_3, y_4, y_5$

1) 32 2) 21 3) 15 4) 10

۶- کدام گزینه تعداد برگ های یک درخت تصمیم گیری (Decision tree) برای مرتب سازی پنج عنصر را نشان می‌دهد؟

1) 64 2) 120 3) 128 4) 720

برابر **n** است. متوسط زمان اجرای این الگوریتم چقدر است؟

1) $\theta(n^2)$ 2) $\theta(n)$ 3) $\theta(n \log n)$ 4) $\theta^2(n^2 \log n)$

۸- کدام یک از الگوریتم های مرتب سازی زیر در الگوریتم Radix sort برای بالا بردن سرعت استفاده می شود؟

1) Quick sort 2) heap sort 3) Insertion sort 4) Merge sort

۹- برای مرتب سازی آرایه ای با ۲۰۰۰ عضو، از الگوریتم Randomized-Quicksort استفاده شده است. اگر call stack برنامه در شروع الگوریتم خالی باشد و برای هر فراخوانی تابع تنها ۴ بایت آدرس برگشت در call stack قرار گیرد، در طی این فراخوانی، حداکثر طول اشغال شده call-stack به طور متوسط چند بایت خواهد بود؟

1) 4 2) 8 3) 44 4) 88

۱۰- تابع بازگشتی زیر را درنظر بگیرید. کدام یک از الگوریتم های زیر را می توان طوری تغییر داد که براساس این فرمول کار کند و پیچیدگی زمان آن چه خواهد بود؟

$$\begin{cases} T(1) = O(1) \\ T(n) = 3T(\frac{n}{3}) + O(n) \end{cases}$$

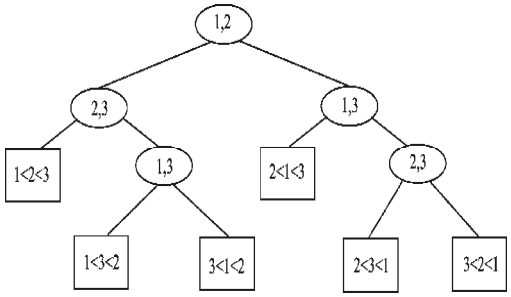
1) heap sort , $O(n \lg n)$ 2) Quick sort , $O(n \lg n)$

3) Insertion sort , $O(n^2)$ 4) Quick sort , $O(n^2)$

۱۱- در یک آرایه A به اندازه n، اگر i<j و A[i] > A[j] می‌گوییم که زوج(i,j) یک "زوج- معکوس" در A است. بیشترین تعداد "زوج- معکوس‌ها" در یک آرایه n عضوی چند تاست؟

1) $\frac{n(n-1)}{2}$ 2) n^2 3) $n^2 - n$ 4) $\frac{n^2}{4}$

مقایسه، استفاده می‌شود. برای مثال درخت روبرو، برای مرتب کردن یک فایل با سه عضو، استفاده می‌شود. کدام یک از گزاره‌های ذیل، در مورد این نوع درخت‌ها درست است؟



1) عمق این نوع درخت‌ها، حداقل $\log_2(n!)$ است و در گروه $o(n.\log n)$ قرار می‌گیرد.

2) عمق این نوع درخت‌ها، حداکثر $\log_2(n!)$ است و در گروه $o(n.\log n)$ قرار می‌گیرد.

3) عمق این نوع درخت‌ها، حداکثر $\log_2(n!)$ است و حداقل $o(n.\log n)$ مقایسه در آنها انجام می‌شود.

4) عمق این نوع درخت‌ها، حداقل $\log_2(n!)$ است و حداکثر $o(n.\log n)$ مقایسه در آنها انجام می‌شود.

۱۳- در گونه جدیدی از مرتب سازی سریع، برای انتخاب محور از میان n عنصر، عنصر اول آرایه را انتخاب می کنیم و با الگوریتم مرتب سازی درجی آن ها را مرتب می کنیم. محور، عنصر میانه این تعداد عنصر مرتب است. بقیه الگوریتم مانند مرتب سازی سریع عمل می کند. کدام یک از رابطه های باگشتی زیر، زمان اجرای این الگوریتم را در بدترین حالت نشان می دهد؟

1) $T(n) \le T(n-\sqrt{n}) + O(\sqrt{n})$

2) $T(n) \le T(2\sqrt{n}) + T(n-2\sqrt{n}) + O(n)$

3) $T(n) \le T(\sqrt{n}) + T(n-\sqrt{n}) + O(n)$

4) $T(n) \le T(2\sqrt{n}) + T(n-2\sqrt{n}) + O(\sqrt{n})$

۱۴- در الگوریتم مرتب سازی حبابی (یا تبادلی) فرض می کنیم T(n) تعداد دستورالعمل مقایسه باشد. در این صورت T(n) کدام است؟

1) $T(n) = \frac{n(n+1)}{2}$

2) $T(n) = 2n-1$

3) $T(n) = \frac{n(n-1)}{2}$

4) $T(n) = \frac{n^2}{2} - 1$

۱۵- اگر در مرتب سازی سریع، زیر لیست های کوچکتر در ابتدا مرتب شوند، پشته بازگشتی دارای چه عمقی خواهد بود؟

1) $o(n)$

2) $o(n^2)$

3) $o(\log n)$

4) $o(1)$

پیچیدگی محاسباتی : مسئله جستجو

۱۶- بهترین الگوریتم برای جستجوی مقداری مثل z در یک ماتریسی به ابعاد $n \times n$ که در راستای سطرها و ستون ها به صورت غیر نزولی مرتب شده است، دارای چه هزینه زمانی می باشد؟

۱) $O(n^2)$ ۲) $o(n)$ ۳) $O(n^{\frac{1}{2}})$ ۴) $O(n\log_2^n)$

۱۷- الگوریتم پیدا کردن دو عنصر بیشینه و کمینه در یک آرایه با N عنصر به صورت زیر است. حداکثر و حداقل تعداد مقایسه های دو عنصر از آرایه (سطر۴ و ۶) به ترتیب کدام است؟

```
MINMAX (A)
۱-  min ← ۱
۲-  max ← ۱
۳-  for i ← ۲ to N do
۴-        if A[i] < A[min]  then
۵-              min ← i
۶-         else if A[i] > A[max]  then
۷-                    max ← i
```

۱) 2n و n ۲) هر دو n-1 ۳) هر دو n ۴) 2(n-1) و n-1

۱۸- دومین کوچک ترین عنصر بین n عنصر را با چند مقایسه می توان بدست آورد؟

۱) $n+\lceil\log n\rceil-1$ ۲) $n+\lceil\log n\rceil-2$

۳) $n+\lfloor\log n\rfloor-1$ ۴) $n+\lfloor\log n\rfloor-2$

۱۹- در الگوریتم پیدا کردن k امین عنصر، ابتدا همه عناصر را به دسته های ۵ تایی(به جز احتمالا یک دسته) تقسیم می کنیم، میانه هر دسته را به دست می آوریم و سپس میانه میانه ها را به صورت بازگشتی پیدا می کنیم. این عنصر را به عنوان محور انتخاب می کنیم و عمل Partition را بر روی آرایه عناصر انجام می دهیم. پس از آن همین الگوریتم را به صورت بازگشتی(و برای یک k دیگر) بر روی یکی از بخش ها اجراء می کنیم تا عنصر مورد نظر پیدا شود. زمان اجرای این الگوریتم توسط کدام یک از رابطه های بازگشتی زیر بیان می شود؟

۱) $T(n)=T(\lceil\frac{n}{5}\rceil)+T(\frac{3n}{10}-6)+O(n)$ ۲) $T(n)=T(\lfloor\frac{n}{5}\rfloor)+T(\frac{3n}{10}-6)+O(n)$

۳) $T(n)=T(\lceil\frac{n}{5}\rceil)+T(\frac{7n}{10}+6)+O(n)$ ۴) $T(n)=T(\lfloor\frac{n}{5}\rfloor)+T(\frac{7n}{10}+6)+O(n)$

۲۰- می نیمم و ماکزیمم اعداد ذخیره شده در آرایه A[۱..n] با چند مقایسه بین این اعداد به دست می آید؟ (فرض کنید n=۲k+۱ و k عدد طبیعی است)

۱) $3k+1$ ۲) $\frac{3k}{2}-1$ ۳) $3k$ ۴) $\frac{3k}{2}-\frac{3}{2}$

۲۱- مینیمم و ماکزیمم اعداد ذخیره شده در آرایه A[۱..n] با چند مقایسه بین این اعداد بدست می آید؟ (آزاد ۸۸)

1) $3k-2$ 2) $\frac{3k}{2}-2$ 3) $\frac{3k}{2}+1$ 4) $\frac{3}{2}(k-1)$

۲۲- تعداد مقایسه های لازم برای مشخص نمودن مینیمم و ماکزیمم عناصر ذخیره شده در یک آرایه یک بعدی شامل n عنصر را T(n) فرض می کنیم. در این صورت رابطه T(n) کدام است (فرض کنید T(۱)=۰ , T(۲)=۱) ؟

1) $T(n)=\frac{3T(n-2)}{2}+1$ 2) $T(n)=T(n-1)+T(n-2)+2$

3) $T(n)=2T(n-2)+1$ 4) $T(n)=T(n-2)+3$

پاسخ تشریحی

2-1) دنباله ای از اعداد صحیح بـه تعـداد n وجـود دارد. اگـر k عـدد غیـر تکـراری وجـود داشـته باشـد، آنگـاه بـا مرتبـه زمـانی $O(n\log k)$ می توان این دنباله را مرتب کرد.

1_2)

4-3) اگر آرایه نزولی باشد، بیشترین تعداد وارونگی را داریم که برابر $\frac{n(n-1)}{2}$ می باشد. بنابراین میانگین تعداد وارونگـی هـا بـا احتمال $\frac{1}{2}$ برابر است با: $\frac{n(n-1)}{4}$.

2-4) یک درخت تصمیم گیری برای مرتب سازی n عنصر، دارای ارتفاع $\log n!$ می باشد که $n!$ تعـداد بـرگ هـای ایـن درخـت است.

تذکر: $\log n!$ از نظر پیچیدگی با $n\log n$ برابر است.

2-5) حالتهای ممکن عبارتند از:

الف- قرار دادن $x1,x2$ به صورت پشت سرهم $(x1x2)$ در میان پنج y (6 حالت وجود دارد)

ب- قرار دادن $x1$و$x2$ به طوری که پشت سرهم نباشند، در میان y ها (15 حالت وجود دارد)

$$\binom{6}{2}=\frac{6!}{2!\times 4!}=15$$

بنابراین در کل 6+15 یعنی 21 حالت ممکن است.

2-6) تعداد برگ های یک درخت تصمیم گیری برای مرتب سازی n عنصر برابر $n!$ است.

2-7) مرتبه اجرایی الگوریتم Radix sort روی n عدد r رقمی برابر $q(n\times r)$ می باشد و چون در صورت تست گفته شده فاصله محدود به $[0,n^2-1]$ می باشد، بنابراین تعداد ارقام اعداد محدود می باشد و زمان اجرا $\theta(n)$است.

3-8) از الگوریتم مرتب سازی درجی در الگوریتم Radix sort برای بالا بردن سرعت استفاده می شود.

3-9) در Randomized-Quicksort ، محور به صورت تصادفی و با احتمال یکسان یکی از عناصر انتخاب مـی شـود و بقیـه ی الگوریتم مانند Quick Sort عادی است. اگر داده ها در هر بار به طور مساوی تقسیم شوند، حداکثر عمق درخت بازگشـتی $\log n$ می باشد که با فرض $n=2000$ برابر 11 می باشد و چون هر آدرس 4 بایتی است، فضای پشته برابر است با : $11\times 4=44$

تذکر: در بدترین حالت، داده ها به دو دسته با طول $n-1$ و 0 تقسیم می شوند (همه در یک طرف محور قرار می گیرنـد)، و فضـای پشته در این حالت $o(n)$ می باشد.

2-10) می توان الگوریتم مرتب سازی سریع را به گونه ای تغییر داد که بر اساس فرمول زیر کار کند:

$$\begin{cases}T(1)=O(1)\\ T(n)=3T(\frac{n}{3})+O(n)\end{cases}$$

پیچیدگی این تابع با توجه به قضیه اصلی که در فصل اول داده شد، برابر است با: $\theta(n\lg n)$

1	2	3
8	5	2

توجه شود که بیشترین تعداد زوج معکوس وقتی رخ می‌دهد که آرایه به طور نزولی مرتب باشد. واضح است که تعداد(n-1) زوج معکوس با عنصر اول می‌توان درست کرد، تعداد (n-2) زوج معکوس با عنصر دوم و در نهایت یک زوج معکوس با عنصر یکی به آخر. بنابراین در مجموع تعداد زوج معکوس ها برابر است با:

$$(n-1)+(n-2)+.....+2+1=\frac{n(n-1)}{2}$$

1-12) عمق درخت تصمیم گیری، حداقل برابر $\log_2(n!)$ می باشد. همچنین اگر به رابطه $n!$ توجه شود، مشخص خواهد شد که: $\log n! = o(n.\log n)$

3-13) اگر میانه $2\sqrt{n}+1$ عنصر را به عنوان عنصر افراز در نظر بگیریم، بعد از فراخوانی الگوریتم partition ، حداقل $\sqrt{n}$ عنصر قبل و $\sqrt{n}$ عنصر بعد از عنصر افراز قرار می گیرند. بنابراین برای مرتب سازی n عنصر، $\sqrt{n}$ عنصر قبل و $n-\sqrt{n}$ عنصر بعد از محور قرار می گیرد. بنابراین رابطه بازگشتی به صورت زیر است:

$$T(n) \le T(\sqrt{n}) + T(n-\sqrt{n}) + O(n)$$

$O(n)$ ، مرتبه زمانی مرتب کردن $2\sqrt{n}+1$ عنصر به کمک مرتب سازی درجی است.

3-14) تعداد مقایسه ها در روش مرتب سازی حبابی برابر $\frac{n(n-1)}{2}$ می باشد.

1-15) اگر در روش Quick Sort ، داده ها هر بار به طور مساوی تقسیم شوند، حداکثر عمق درخت بازگشتی logn بوده و به فضای پشته $o(\log n)$ نیاز خواهد بود. اما در بدترین حالت یعنی وقتی داده ها به قسمتهای به طول n-1 و صفر تقسیم شده باشد، فضای پشته $o(n)$ خواهد بود.

2-16) بهترین الگوریتم برای جستجوی مقداری مثل z در یک ماتریسی به ابعاد $n \times n$ که در راستای سطرها و ستون ها به صورت غیر نزولی مرتب شده است، دارای هزینه زمانی $o(n)$ می باشد.

4-17) در بهترین حالت شرط A[i]<A[min] همواره برقرار بوده و دستور else بررسی نمی‌شود و مقایسه کمتری خواهیم داشت. این حالت وقتی رخ می‌دهد که آرایه به صورت نزولی مرتب است. در این حالت حلقه for به تعداد n-1 مرتبه اجرا می‌شود. در بدترین حالت شرط A[i]<A[min] هیچگاه برقرار نمی باشد و همواره دستور else بررسی می شود. این حالت وقتی رخ می دهد که عنصر Min در خانه اول قرار دارد. در این وضعیت تعداد مقایسه ها دو برابر حالت اول یعنی 2(n-1) می باشد.

2-18) پیدا کردن k امین کوچکترین عنصر در میان n عنصر به $n+(k-1)\left\lceil \log(\frac{n}{k-1}) \right\rceil - k$ مقایسه نیاز دارد. با قرار دادن k=2 داریم: $n+\lceil \lg n \rceil - 2$.

4-19) مراحل الگوریتم:

الف- تقسیم n عنصر به دسته های 5 تایی(به جزء احتمالا یک دسته) در زمان O(n).(تعداد دسته ها $\left\lceil \frac{n}{5} \right\rceil$ است.)

ج- نگهداری میانه دست و انتخاب نیانه میانه ها به صورت بازگشتی در زمان $O(\lceil \frac{n}{5} \rceil)$.

د- افراز آرایه اولیه با استفاده از میانه میانه ها به عنوان عنصر افراز.

با فرض اینکه محل عنصر محور بعد از پارتیشن برابر p باشد، اگر p=k ، مساله حل شده است. اگر p>k ، مساله به صورت بازگشتی برای آرایه A[1..p-1] و اگر p<k برای آرایه A[p+1..n] حل می کنیم. با ذکر یک مثال می توان نشان داد که در هر یک از این زیر آرایه ها، حداکثر $\frac{7n}{10}+6$ عنصر وجود دارد. در نتیجه رابطه بازگشتی به صورت زیر است:

$$T(n) = T(\lceil \frac{n}{5} \rceil) + T(\frac{7n}{10}+6) + O(n)$$

3-20) پیچیدگی الگوریتم پیدا کردن مینیمم و ماکزیمم در میان یک آرایه n عنصری، برای n های فرد برابر $\frac{3n}{2}-\frac{3}{2}$ می باشد که با قرار دادن n=2k+1 برابر 3k خواهد شد.

1-21) دو عنصر اول آرایه A[1..n] را مقایسه کرده و عنصر کوچکتر را در متغیر Min و عنصر بزرگتر را در متغیر Max ذخیره می کنیم و به صورت بازگشتی این اعمال را انجام می دهیم. با فرض اینکه مینیم و ماکزیمم آرایه A[n-2] را بدست آورده ایم، حال با 3 مقایسه می توان می نیمم و ماکزیمم کل عناصر را بدست آورد. رابطه بازگشتی به صورت زیر می باشد:

$$\begin{cases} T(n) = T(n-2)+3 \\ T(1) = 0 \\ T(2) = 1 \end{cases} \Rightarrow T(n) = \frac{3n}{2} - 2$$

که با قرار دادن n=2K به رابطه گزینه یک می رسیم.

4-22) ابتدا عنصر اول و دوم را با هم مقایسه کرده (اولین مقایسه) و عنصر کوچکتر را در min و عنصر بزرگتر را در max قرار می‌دهیم. سپس به صورت بازگشتی کوچکترین و بزرگترین عنصر را در آرایه A[3..n] پیدا کرده و با مقایسه کوچکترین عنصر این آرایه با min، عنصر کوچکتر در کل آرایه مشخص می‌شود (مقایسه دوم) و همچنین با مقایسه بزرگترین عنصر آرایه با max، بزرگترین عنصر در کل آرایه مشخص می‌شود (مقایسه سوم)، این رابطه بازگشتی را می‌توان به صورت زیر نوشت:

$$T(n) = T(n-2)+3$$

فصل هفتم: نظریه NP

الگوریتم زمانی چند جمله ای، الگوریتمی است که پیچیدگی زمانی آن در بدترین حالت در حد بالایی، تابع چند جمله ای از اندازه ورودی باشد. یعنی، اگر n اندازه ورودی باشد، یک چندجمله ای p(n) وجود دارد به قسمی که: $W(n) \in O(p(n))$.

الگوریتم هایی با پیچیدگی زمانی $2n, 3n^3 + 4n, 5n + n^{10}, n \lg n$، همگی زمانی چندجمله ای اند و الگوریتم هایی با پیچیدگی زمانی $n!, 2^{\sqrt{n}}, 2^{0,01n}, 2^n -$ ، هیچکدام زمانی چندجمله ای نیستند.

تذکر: $n \lg n$ نسبت به n چند جمله ای نیست. ولی چون $n \lg n < n^2$، به یک چند جمله ای از n محدود می شود و می توان آن را جزء پیچیدگی زمانی چند جمله ای در نظر گرفت.

به مسائلی که نوشتن یک الگوریتم کارآمد برای آن ها غیر ممکن است، کنترل ناپذیر می گویند. در علم کامپیوتر، مسئله ای را کنترل ناپذیر می گویند که حل آن توسط یک الگوریتم زمانی چند جمله ای غیر ممکن باشد.

سه گروه کلی مسائل

اکنون زمان آن رسیده تا سه گروه کلی از مسائل را مورد بحث قرار دهیم که در آن ها مسائل را بر حسب میزان کنترل ناپذیری گروه بندی می کنیم.

۱- مسائلی که الگوریتم های زمانی چند جمله ای برای آن ها یافت شده است.

هر مسئله ای که یک الگوریتم زمانی چند جمله ای برای آن یافته ایم در این گروه قرار داده می شود. برای مرتب سازی الگوریتم های $q(n \lg n)$، برای جست و جو در یک آرایه مرتب یک الگوریتم $\theta(\lg n)$، برای ضرب ماتریس ها یک الگوریتم $\theta(n^{2.38})$ و برای ضرب زنجیره ای ماتریس ها یک الگوریتم $\theta(n^3)$ یافته ایم. چون n میزانی از مقدار داده در ورودی این الگوریتم هاست همگی زمان چند جمله ای دارند.

۲- مسائلی که کنترل ناپذیری آن ها ثابت نشده است.

دو نوع مسئله در این گروه وجود دارد. نوع اول مسائلی اند که نیازمند یک مقدار خروجی غیر چند جمله ای هستند. دومین نوع از کنترل ناپذیری هنگامی رخ می دهد که درخواست های ما منطقی باشند و بتوانیم ثابت کنیم که مسئله را نمی توان در زمان چند جمله ای حل کرد. تعداد اینگونه مسائل چندان زیاد نبود. همه مسائلی که تا این تاریخ کنترل ناپذیری آن ها اثبات شده است نبودن آنها در مجموعه NP نیز اثبات شده است. ولی اکثر مسائلی که به نظر می رسد کنترل ناپذیر باشند در مجموعه NP قرار دارند.

۳- مسائلی که کنترل ناپذیری آن ها ثابت نشده است ولی تاکنون هیچ الگوریتم زمانی چند جمله ای هم برای آن ها پیدا نشده است.

این گروه شامل تمام مسائلی می شود که هیچ الگوریتم زمانی چند جمله ای برای آن ها اثبات نشده است ولی هنوز کسی غیر ممکن بودن این چنین الگوریتمی را اثبات نکرده است. برای مثال اگر مسائل را طوری بیان کنیم که تنها یک حل مورد نیاز باشد، مسئله کوله پشتی صفر و یک، مسئله فروشنده دوره گرد، مسئله حاصل جمع زیر مجموعه ها، مسئله رنگ آمیزی m به ازای $m \geq 3$ و مسئله مدارهای هامیلتونی ، همگی در این گروه قرار می گیرند.

برای این مسائل، الگوریتم های شاخه و حد، الگوریتم های عقبگردی و الگوریتم های دیگری یافته ایم که برای بسیاری از نمونه های بزرگ کارآیی دارند. یعنی یک چند جمله ای از n وجود دارد که برای تعداد دفعات انجام عمل اصلی در صورت انتخاب نمونه

که به ازای آن ها هیچ حد چند جمله ای از n وجود نداشته باشد که برای تعداد دفعات انجام عمل اصلی حدی تعیین کند.

نظریه NP

اگر از ابتدا خودمان را به مسائل تصمیم گیری محدود کنیم، بسط این نظریه راحت تر می شود. خروجی مسائل تصمیم گیری پاسخ آری یا خیر است. با این وجود هنگامی که برخی مسائل ذکر شده در بالا را معرفی می کردیم، آنها را به صورت مسائل بهینه سازی عنوان کردیم و این بدان معناست که خروجی یک حل بهینه است. ولی هر مسئله بهینه سازی متناظر با یک مسئله تصمیم گیری است. مسئله بهینه سازی کوله پشتی صفر و یک عبارت از تعیین حداکثر ارزش کلی بود که از قرار دادن قطعات در یک کوله پشتی به دست می آید با این فرض که هر قطعه دارای وزن و ارزش مشخص باشد و کوله پشتی تحمل حمل حداکثر وزن W را داشته باشد. مسئله تصمیم گیری کوله پشتی صفر و یک این است که برای یک ارزش مفروض P تعیین کنیم که آیا می توان کوله پشتی را طوری بار کرد که وزن کل آن از W تجاوز نکند و در عین حال ارزش کل حداقل برابر P باشد. این مسئله دارای همان پارامترهای مسئله بهینه سازی کوله پشتی صفر و یک است به علاوه پارامتر اضافی P .

تاکنون برای حالت تصمیم گیری و بهینه سازی مثال فوق هیچ الگوریتم زمانی چند جمله ای یافت نشده است. ولی اگر بتوانیم یک الگوریتم زمانی چند جمله ای برای حالت بهینه سازی آن بیابیم، برای مسئله تصمیم گیری متناظر با آن نیز یک الگوریتم زمانی چند جمله ای خواهیم داشت. علتش این است که حل یک مسئله بهنیه سازی حلی برای مسئله تصمیم گیری متناظر تولید می کند. به طور مشابه اگر می دانستیم که ارزش کل برای یک نمونه خاص از مسئله کوله پشتی صفر و یک 230 است پاسخ مسئله تصمیم گیری متناظر با آن به ازای $P \leq \$230$ آری و در غیر آن صورت خیر است.

از آن جا که الگوریتم زمانی چند جمله ای برای یک مسئله بهینه سازی خود به خود به الگوریتم زمانی چند جمله ای برای مسئله تصمیم گیری مربوط منجر می شود می توانیم از ابتدا نظر خود را فقط درباره مسائل تصمیم گیری بنا کنیم. برای بسیاری از مسائل تصمیم گیری نشان داده شده است که از یک الگوریتم زمانی چند جمله ای برای مسئله تصمیم گیری می توان به یک الگوریتم زمانی چند جمله ای برای مسئله بهینه سازی مربوطه دست پیدا کرد.

مجموعه P

P عبارت است از مجموعه مسائل تصمیم گیری که توسط الگوریتم های زمانی چند جمله ای قابل حل هستند.

همه مسائل تصمیم گیری که برای آن ها الگوریتم زمانی چند جمله ای یافته ایم به P تعلق دارند. آیا ممکن است مسئله فروشنده دوره گرد در P باشد؟ اگر چه کسی تاکنون موفق به یافتن الگوریتم زمانی چند جمله ای برای این مسئله نشده است، کسی هم تاکنون اثبات نکرده است که حل آن با یک الگوریتم زمانی چند جمله ای غیر ممکن است. بنابراین این امکان وجود دارد که در P باشد. برای آن که بدانیم یک مسئله تصمیم گیری در P نیست باید ثابت کنیم که طراحی یک الگوریتم زمانی چند جمله ای برای آن غیر ممکن است. این کار برای مسئله فروشنده دوره گرد انجام نشده است. تعداد مسائل تصمیم گیری نسبتاً کمی هستند که عدم وجود الگوریتم زمانی چند جمله ای برای آن ها به اثبات رسیده است.

تعریف : الگوریتم غیر قطعی زمان چند جمله ای، یک الگوریتم غیر قطعی است که مرحله تصدیق آن، زمان چند جمله ای باشد.

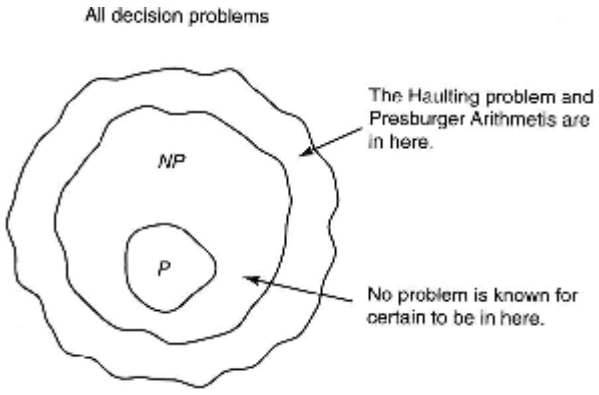
NP مجموعه تمامی مسائل تصمیم گیری است که توسط الگوریتم های غیر قطعی زمانی چند جمله ای قابل حل هستند. NP نشان دهنده چند جمله ای غیر قطعی (Non deterministic polynomial) است.

برای آن که یک مسئله تصمیم گیری در NP باشد باید الگوریتمی وجود داشته باشد که عمل تصدیق را در زمان چند جمله ای انجام دهد. چون این مورد برای مسئله تصمیم گیری فروشنده دوره گرد صادق است، این مسئله در NP است. این بدان معنا نیست که الزاماً الگوریتمی با زمان چند جمله ای برای حل مسئله داریم. در واقع در حال حاضر چنین الگوریتمی برای مسئله فروشنده دوره گرد نداریم.

هدف از معرفی مفاهیم الگوریتم های غیر قطعی و NP ، طبقه بندی الگوریتم ها است. معمولاً الگوریتم های بهتری برای حل واقعی یک مسئله وجود دارد. کدام مسائل تصمیم گیری دیگری در NP هستند؟ مسائل تصمیم گیری دیگر در مثال های " فروشنده دوره گرد" ، " کوله پشتی صفر و یک" و " رنگ آمیزی گراف" همگی NP هستند. به علاوه هزاران مسئله دیگر وجود دارد که هیچ کسی تاکنون نتوانسته با الگوریتم های زمانی چند جمله ای آن ها را حل کند ولی ثابت شده است که در NP هستند، زیرا الگوریتم های غیر قطعی زمانی چند جمله ای برای آن ها ارائه شده است. سرانجام این که تعداد زیادی از مسائل وجود دارد که وجود آن ها در NP بدیهی است، یعنی هر مسئله ای که در P باشد در NP هم هست.

تنها مسائل تصمیم گیری که عدم وجود آن ها در NP به اثبات رسیده است، همان هایی هستند که کنترل ناپذیری آن ها به اثبات رسیده است. یعنی،مسئله توقف، حساب پرزبرگر و دیگر مسائل بحث شده که ثابت شده است در NP نیستند. تعداد این گونه مسائل نسبتاً کم است.

در شکل زیر مجموعه تمام مسائل تصمیم گیری نشان داده شده است. در این شکل NP حاوی P به عنوان یک زیر مجموعه متعارف است. ولی ممکن است چنین نباشد یعنی هنوز تاکنون کسی ثابت نکرده است که مسئله ای وجود دارد که در NP باشد ولی درP نباشد. بنابراین ممکن است NP - P تهی باشد. در واقع این پرسش که آیا P با NP برابر است یکی از مهترین و بحث برانگیزترین پرسش ها در علم کامپیوتر است. این پرسش از آن رو اهمیت دارد که اکثر مسائل تصمیم گیری به NP تعلق دارند. پس اگر P=NP باشد برای اکثر مسائل تصمیم گیری شناخته شده الگوریتم های زمانی چند جمله ای داریم.



برای آن که نشان دهیم NP ≠ P است باید مسئله ای بیابیم که در NP باشد و در P نباشد. حال آنکه برای نشان دادن P=NP باید برای هر یک از مسائل موجود در NP یک الگوریتم زمانی چند جمله ای بیابیم. بسیاری از پژوهشگران شک دارند که P با NP برابر باشد.

مسائل NP کامل

ممکن است چنین به نظر برسد که مسائل مثال های "فروشنده دوره گرد" ،"کوله پشتی صفر و یک" و"رنگ آمیزی گراف" همگی به یک میزان دشوار نباشد. برای مثال الگوریتم برنامه نویس پویا برای مسئله فروشنده دوره گرد در بدترین حالت $q(n^2 2^n)$ است. از طرف دیگر الگوریتم برنامه نویسی پویا برای مسئله کوله پشتی صفر و یک در بدترین حالت $q(2^n)$ است. الگوریتم برنامه نویسی پویا برای مسئله کوله پشتی صفر و یک $\theta(nw)$ است و این بدان معناست که تا هنگامی که ظرفیت w خیلی خیلی بزرگ نباشد، کارآیی آن زیاد است. نظر به تمام این نکات، شاید مسئله کوله پشتی صفر و یک ذاتاً آسانتر از مسئله فروشنده دوره گرد است. به رغم این تصور این دو مسئله، مسائل دیگر همگی از آن لحاظ هم ارزند که اگر هر یک در P قرار داشته باشد، همگی باید در P باشند. چنین مسائلی را NP کامل می گویند.

تعریف : مسئله ای را NP کامل می گویند اگر:

1- در NP باشد. 2- به ازای هر مسئله دیگر A در NP داشته باشیم: $A \ \alpha \ B$

تذکر: بنا به قضیه قبل اگر می توانستیم نشان دهیم هر مسئله NP کامل در P قرار دارد می توانستیم نتیجه بگیریم که P=NP است. کوک توانست مسئله ای بیابد که NP کامل باشد.

قضیه : یک مسئله C، مسئله NP کامل است اگر :

1- در NP باشد. 2- به ازای یک مسئله NP کامل دیگر مثل B داشته باشیم: $B \ \alpha \ C$

مجموعه تست

۱- کدام گزینه نادرست است؟

1) تمام مسائل P به وسیله یک الگوریتم غیر قطعی در زمان چند جمله ای حل می شوند.

2) تمام مسائل NP به وسیله یک الگوریتم غیر قطعی در زمان چند جمله ای حل می شوند.

3) تمام مسائل NP-hard به وسیله یک الگوریتم غیر قطعی در زمان چند جمله ای حل می شوند.

4) تمام مسائل NP-Complete به وسیله یک الگوریتم غیر قطعی در زمان چند جمله ای حل می شوند.

۲- اگر یک مسئله NP-Complete مانند L وجود داشته باشد که $L \in P$ باشد، در آن صورت:

1) $P = NP$ 2) $P \neq NP$

3) $L \in NP - hard$ 4) $L \notin NP - hard$

۳- گزینه صحیح را انتخاب کنید. (علوم کامپیوتر- دولتی ۸۲)

1) مسائل NP-Complete زیر مجموعه مسائل NP-hard می باشند.

2) مسائل NP-hard زیر مجموعه مسائل NP-Complete می باشند.

3) مسائل NP زیر مجموعه مسائل P می باشند.

4) P=NP

پاسخ تشریحی

3-1) تمام مسائل P ، NP و NP-Complete به وسیله یک الگوریتم غیر قطعی در زمان چند جمله ای حل می شوند. (P زیر مجموعه NP است)(NP-Complete زیر مجموعه NP است)

1-2) توسط کوک ثابت شد که اگر یک مسئله NP-Complete مانند L وجود داشته باشد که $L \in P$ باشد، در آن صورت:P=NP .

1-3) مسائل NP-Complete زیر مجموعه مسائل NP-hard می باشند.

پیوست

حل روابط بازگشتی همگن

برای حل روابط بازگشتی همگن مرتبه اول از روش جایگذاری استفاده می کنیم. برای حل روابط بازگشتی همگن مرتبه دوم با ضرایب ثابت، ابتدا معادله مشخصه آن را پیدا کرده و بعد از حل این معادله با فرض داشتن دو جواب مجزای r_1 و r_2، جواب رابطه بازگشتی برابر است با: $c_1 r_1^n + c_2 r_2^n$ و اگر یک ریشه حقیقی مضاعف داشته باشد، جواب رابطه بازگشتی برابر است با: $c_1 r_1^n + c_2 n r_1^n$

مثال: بازگشتی زیر را حل کنید.

$$T(n) = T(n-1) + 2T(n-2)$$
$$T(0) = 2$$
$$T(1) = 7$$

حل: معادله مشخصه رابطه بازگشتی داده شده برابر است با: $r^2 - r - 2 = 0$. ریشه های این معادله برابر است با : 2 و -1 .

بنابراین جواب کلی برابر است با: $T(n) = c_1 2^n + c_2 (-1)^n$

برای پیدا کردن c_1 و c_2 از مقادیر اولیه استفاده می کنیم:

$$T(0) = 2 \Rightarrow 2 = c_1 2^0 + c_2 (-1)^0$$
$$T(1) = 7 \Rightarrow 7 = c_1 2^1 + c_2 (-1)^1$$

بنابراین داریم:

$$c_1 + c_2 = 2$$
$$2c_1 - c_2 = 7$$

و با حل این دستگاه داریم: $c_1 = 3, c_2 = -1$. در نتیجه جواب برابر است با:

$$T(n) = 3 \times 2^n - (-1)^n .$$

مثال: بازگشتی زیر را حل کنید.

$$T(n+2) = 4T(n+1) - 4T(n)$$
$$T(0) = 1, T(1) = 3$$

حل: معادله مشخصه رابطه بازگشتی داده شده برابر است با: $r^2 - 4r + 4 = 0$. این معادله دارای ریشه مضاعف 2 می باشد.

بنابراین جواب کلی برابر است با : $T(n) = c_1 2^n + c_2 n 2^n$. برای پیدا کردن c_1 و c_2 از مقادیر اولیه استفاده می کنیم:

$$T(0) = 1 \Rightarrow c_1 \times 2^0 + c_2 \times 0 \times 2^0 = 1$$
$$T(1) = 3 \Rightarrow c_1 \times 2^1 + c_2 \times 1 \times 2^1 = 3$$

بنابراین داریم:

$$c_1 = 1$$
$$2c_2 + 2c_2 = 3$$

و با حل این دستگاه داریم: $c_1 = 1, c_2 = \frac{1}{2}$. در نتیجه جواب برابر است با:

$$T(n) = 2^n + n 2^{n-1} \qquad n \geq 0$$

الگوریتمی که ارائه شده از مرتبه 2 بود و می خواهیم الگوریتمی ارائه دهیم که بهتر باشد. در الگوریتم بررسی شده، از رابطه زیر برای ضرب u و v استفاده شد:

$$uv = xw \times 10^{2m} + (xz + wy) \times 10^{m} + yz$$

در این رابطه 4 عمل ضرب انجام می گیرد. این ضرب ها عبارتند از : xw, xz , wy , yz .

برای کاهش تعداد ضرب ها دست به یک ابتکار می زنیم. یک متغیر r به صورت $r = (x+y)(w+z)$ تعریف می کنیم. که می توان r را به صورت $r= xw + (xz + yw) + yz$ نیز نوشت. با توجه به این رابطه مشخص است که: $xz + yw = r - xw - yz$

بنابراین داریم:

$$uv = xw \times 10^{2m} + (r - xw - yz) \times 10^{m} + yz$$

در این حالت برای محاسبه uv به انجام 3 عمل ضرب نیاز داریم. این ضرب ها عبارتند از: xw , yz, (x+y)(w+z)

الگوریتم شماره ۲ برای ضرب اعداد صحیح بزرگ

```
large-integer prod(large-integer u , large-integer v){
  large-integer  x,y,z,w,r,p,q;
  int n,m;
  n = maximum(number of digits in u , number of gigits in v)
  if(u==0 || v==0)
     return 0;
  else if (n<= threshold)
        return u×v obtained in the usual way;
  else{
       m=⌊n/2⌋;
       x = u divide 10^m ;    y = u rem 10^m ;
       w = v divide 10^m ;    z = v rem 10^m ;
       r = prod(x+y , w+z);
       p = prod(x,w);
       q = prod(y,z);
       return   p×10^2m +(r-p-q) ×10^m +q ;
    }
}
```

تحلیل پیچیدگی زمانی در بدترین حالت ضرب اعداد صحیح بزرگ (دومین الگوریتم)

بدترین حالت هنگامی رخ می دهد که هیچ یک از این دو عدد صحیح، رقمی مساوی صفر نداشته باشند، زیرا بازگشتی فقط هنگامی پایان می یابد که به حد آستانه ای برسیم. اگر n توانی از2 باشد، در آن صورت w,y,x و z همگی n/2 رقم دارند. بنابراین، تعداد ارقام در x+y و یا w+z در محدوده $\frac{n}{2}+1$ تا $\frac{n}{2}$ می باشد. یعنی اندازه ورودی ها برای فراخوانی های تابع prod(x+y,w+z) در محدوده $\frac{n}{2}+1$ تا $\frac{n}{2}$ می باشد. و اندازه ورودی prod(x,w) و prod(y,z) نیز هر کدام برابر است. چون m=n/2 است، عملیات زمانی خطی جمع، تفریق، $\times 10^m$، rem 10^m و divide 10^m همگی دارای پیچیدگی زمانی خطی بر حسب n هستند. بنابراین W(n) در شرط زیر صدق می کند:

$$3W(\frac{n}{2}) + cn \le W(n) \le 3W(\frac{n}{2}+1) + cn \qquad n > s$$
$$W(s) = 0$$

که s کوچکتر یا مساوی آستانه (threshold) و توانی از 2 است، زیرا در این حالت همه ورودی ها توانی از 2 هستند.برای مقادیری از n که به توانی از 2 محدود نمی شوند، می توان یک دستور بازگشتی نظیر قبلی را ایجاد کرد ولی به صورت پلکانی.

با استفاده از بحث استقرایی می توانیم نشان دهیم که $W(n)$ در نهایت روندی غیر نزولی دارد. بنابراین، با توجه به نامساوی طرف چپ این دستور بازگشتی و قضیه اصلی، داریم: $W(n) \in \Omega(n^{\lg_2^3})$

تذکر: می توان نشان داد که: $W(n) \in O(n^{\lg_2^3})$

با توجه به دو رابطه بالا می توان نتیجه گرفت که:

$$W(n) \in \theta(n^{\lg_2^3}) \approx \theta(n^{1.58})$$

الگوریتم فلوید (الگوریتم دوم)

همانند الگوریتم قبلی است، با این تفاوت که کوتاه ترین مسیرها نیز ایجاد می شود. آرایه P[1..n][1..n] نیز یک خروجی است. در P[i][j] بزرگترین اندیس از یک رأس واسطه روی کوتاه ترین مسیر از vi به vj، اگر حداقل یک رأس واسطه موجود باشد، ذخیره می شود. اگر هیچ راس واسطی وجود نداشته باشد، صفر ذخیره می شود.

الگوریتم زیر کوتاه ترین مسیر از رأس v_q به v_r را با استفاده از آرایه P ایجاد می کند.

```
void  floyd2 (int n,const number W[ ] [ ] , number D[ ] [ ] , index P[ ] [ ] ){
   index  i,j,k;
   for(i=1; i<=n ; i++)
      for(j=1; j<=n ; j++)
         P[i][j]=0;
   D=W;
   for(k=1;k<=n;k++)
      for(i=1;i<=n;i++)
         for(j=1;j<=n;j++)
           if(D[i][k]+D[k][j] < D[i][j])
             {
               P[i][j]=k;
               D[i][j]=D[i][k]+D[k][j];
             }
}
```

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

الگوریتم چاپ کوتاه ترین مسیر

برای چاپ رئوس واسطه روی کوتاه ترین مسیر از رأسی به رأس دیگر در یک گراف موزون از الگوریتم زیر استفاده می شود. ورودی های این الگوریتم آرایه P که توسط الگوریتم قبلی تولید شده و دو اندیس r,q از رئوس گراف که ورودی الگوریتم قبلی هستند، می باشد. خروجی، رئوس واسطه روی کوتاه ترین مسیر از v_q به v_r است.

```
void   path(index  q,r){
  if (P[q][r] !=0){
     path(q , P[q][r]);
     cout<<"v"<<P[q][r] ;
     path(P[q][r] , r);
    }
}
```

تذکر: آرایه P ورودی path نیست. چون فقط متغیرهایی می توانند وارد روال های بازگشتی شوند که مقادیر آن‌ها قابل تغییر باشد. اگر P به طور عمومی در الگوریتم تعریف شود، برای به دست آوردن کوتاه ترین مسیر از v_q به v_r، فراخوانی path در بالاترین سطح به صورت path(q,r) خواهد بود. با توجه به مقدار P، اگر مقادیر q و r به ترتیب برابر 5و3 بود، خروجی عبارت بود از: v_4 v_1، اینها رئوس واسطه روی کوتاه ترین مسیر از v_5 به v_3 هستند.

نکته: در الگوریتم چاپ کوتاهترین مسیر داریم: **W(n) Î q(n)**

الگوریتم پریم

```
void   prim (int  n, Const number   W[ ][ ] , set_of_edges&  F ){
    index   i,vnear;  number   min; edge  e;
    index   nearest [2..n];  number  distance[2..n];
    F = ϕ ;
    for (i = 2; i <= n; i++) {
         nearest[i] = 1;
         distance[i] = w[1][i];
     }
```

```
        min = ∞;
        for (i = 2; i <= n; i++)
            if (0 <= distance[i] < min) {
                min = distance[i];
                vnear =i ;
             }
      e = edge connecting vertices indexed by vnear and nearest [vnear];
      add  e  to  F;
      distance [vnear] = -1;
      for (i = 2; i <= n; i++) {
          if (w[i][vnear] < distance[i] {
             distance[i] = w[i][vnear];
              nearest[i] = vnear;
          }
      }
}
```

الگوریتم کروسکال

```
void   kruskal (int   n , int  m ,set_of_edges   E , set_of_edges&  F){
    index   i, j;
    set_pointer  p, q;
    edge   e;
    sort  the  m edges in E by weight in nondecreasing order;
    F = ϕ ;
    initial (n);
    while (number of edges in F is less than  n -1)
       {
          e = edge with least weight not yet considered;
          i, j = indices of vertices connected by e;
          p = find (i);
          q = find (j);
          if (! equal (p, q))
          {
              merge (p, q);
              add e to  F;
          }
    }
}
```

- **initial (n)**: n زیرمجموعه متمایز را مقدار دهی اولیه می کند که هریک حاوی دقیقاً یکی از اندیس ها ی 1 تا n است.
- **p=find (i)** : موجب می شود تا p به مجموعه حاوی اندیس i اشاره کند.

اگر p و q هر دو به یک مجموعه اشاره داشته باشند equal(p,q)، ارزش true را باز می گرداند.

الگوریتم دیکسترا

می خواهیم کوتاهترین مسیر از v_1 به همه رئوس دیگر در یک گراف موزون و جهت دار را تعیین کرد. ورودی W، ماتریس همجواری گراف و n تعداد رئوس گراف است $(n \geq 2)$. خروجی الگوریتم، مجموعه یالهای F حاوی یال های موجود در کوتاه ترین مسیر است.

```
void dijkstra(int n,const number W[][],set_of_edges& F){
  index i , vnear; edge e; index touch[2..n]; number length[2..n];
  F= ϕ;
  for( i=2 ; i<=n ;i++){ touch[i]=1; length[i]=W[1][i]; }
  repeat( n-1 times){
    min= ∞;
    for(i=2 ; i<=n; i++)
      if ( 0≤ length[i] < min )
       {
         min=length[i];
         vnear = i;
       }
    e= edge from vertex indexed by touch[vnear] to vertex indexed by vnear;
    add e to F;
    for(i=2 ; i<=n ; i++)
      if ( length[vnear] + W[vnear][i] < length[i])
       {
         length[i]=length[vnear]+W[vnear][i];
         touch[i]=vnear;
       }
    length[vnear]=-1;
  }
}
```

تذکر: این الگوریتم فقط یال های موجود در کوتاه ترین مسیرها را تعیین می کند.

الگوریتم هافمن

قبل از توضیح الگوریتم، ابتدا نوع زیر را اعلان می کنیم:

```
struct nodetype{ char symbol; int frequency; nodetype * left; nodetype * right;};
```

ب- باید از صف اولویت استفاده کرد. در صف اولویت، عنصری با بالاترین اولویت، زودتر از همه خارج می شود. در این مسئله، عنصری با اولویت بیشتر، کاراکتری است که کمترین فراوانی را در فایل دارد. صف اولویت را می توان به صورت لیست پیوندی پیاده سازی کرد.

ابتدا n اشاره گر به رکوردهای nodetype را در صف اولویت PQ به صورت زیر تنظیم کنید. برای هر اشاره گر P در PQ داریم :

$P->symbol =$ کاراکتری در فایل $P->frequency =$ فراوانی آن کاراکتر در فایل

صف فراوانی بر اساس مقدار فراوانی ایجاد می شود، به طوری که فراوانی های کمتر، اولویت بالاتری دارند:

```
for (i = 1;  i<= n–1;  i++){
    remove (PQ, p);
    remove(PQ, q);
    r = new  nodetype;
    r -> left  = p;
    r -> right =  q;
    r -> frequency = p -> frequency + q-> frequency;
    insert(PQ,  r);
    remove(PQ,  r);
    return  r;
}
```

- اگر صف اولویت به صورت heap پیاده سازی شود، در زمان $q(n)$ مقداردهی می شود.
- هر عمل heap مستلزم زمان $\theta(\lg n)$ است. چون $n-1$ گذر از حلقه for i وجود دارد، الگوریتم در زمان $\theta(n \lg n)$ اجرا می شود.

الگوریتم n وزیر

مسئله: قرار دادن n وزیر روی یک صفحه شطرنج $n \times n$ به طوری که هیچ دو وزیری در یک سطر، یک ستون یا یک قطر نباشند. هر خروجی شامل آرایه‌ای از اعداد صحیح col است که از یک تا n اندیس‌گذاری شده‌اند و در آن col[i] ستونی است که در آن وزیر در ردیف iأم قرار داده می‌شود.

```
void   queens ( index  i  ){
    index  j;
    if (promising(i))
        if (i = = n)  cout << col[1] through col[n];
          else for (j = 1; j <=n;j++) {
                col[i + 1] = j;
               queens(i + 1);
            }
}
bool  promising (index i){
    index   k=1;bool   switch= tru;
    while (k < i  &&  switch){
         if (col[i] == col[k]  ||  abs(col[i] – col[k] == i – k)
               switch = false;
         k++;
    }
return switch;
}
```

مرز بالایی تعداد گره‌ها در درخت فضای حالت هرس شده را با شمارش تعداد گره‌ها در کل درخت فضای حالت بـه دسـت آورد. ایـن درخـت آخری حاوی 1 گره در سطح صفر، n گره در سطح 1، n^2 گره در سطح 2 و ... و n^n گره در سطح n است. تعداد کل گره‌ها عبارت است از:

$$1+n+n^2+n^3+...+n^n=\frac{n^{n+1}-1}{n-1}$$

برای نمونه‌ای که در آن $n=8$ است، درخت فضای حالت حاوی این تعداد گره است:

$$\frac{8^{8+1}-1}{8-1}=19;173,961$$

تعداد گره‌های امیدبخش

$$1+8+8\times7+8\times7\times6+8\times7\times6\times5+...+8!=109,601$$

با تعمیم این نتیجه، به یک n دلخواه، حداکثر تعداد گره‌های امیدبخش عبارت است از:

$$1+n+n(n-1)+n(n-1)(n-2)+...+n!$$

الگورتیم عقبگرد برای مسئله حاصل جمع زیر مجموعه‌ها

مسئله: تعیین همهٔ ترکیبات اعداد صحیح موجود در یک مجموعهٔ n عدد صحیح، به طوری که حاصل جمع آن‌ها مساوی مقدار معین W شود.

ورودی‌ها: عدد صحیح مثبت n، آرایهٔ مرتب غیر نزولی از اعداد صحیح مثبت w که از یک تا n اندیس‌گذاری می‌شوند و عدد صحیح مثبت W.

```
void  sos (index  i , int  weight, int  total ){
        if (promising ( i ) )
            if (wight == W)  cout << include[1]  through  include[i];
          else {
                include[i+1] = "yes";
                sos (i+1, weight + w[i+1], total – w[i+1]);
                include[i+1] = "no";
                sos(i+1, weight , total – w[i+1]);
               }
  }
  bool   promsing (index  i) {
     return (weight + total >=W) && (weight == W || weight + w[i+1] <= W);
  }
```

الگوریتم عقبگرد برای مسئله رنگ آمیزی m

ورودی ها: اعداد مثبت و صحیح n وm و گراف بدون جهت حاوی n رأس. گراف توسط یک آرایه دو بعدی W نشان داده می شود که سطرها و ستون های آن از یک تا n اندیس گذاری شده اند و در آن w[i][j] دارای مقدار true است اگر بین رأس iأم و رأس jأم یک یال وجود داشته باشد و در غیر این صورت دارای مقدار false است.

خروجی مربوط به هر رنگ آمیزی یک آرایه vcolor[1..n] است که vcolor[i] رنگ نسبت داده شده به رأس iأم (عدد صحیحی بین یک تا m) است.

```
void   m_coloring  (index i){
```

```
    if (promising(i))
        if (i == n)  cout < < vcolor[1]  through  vcolor[n];
        else   for (color = 1; color <= m; color++)
                {  vcolor[i+1] = color;  m_ coloring(i +1); }
}
bool   promising (index  i){
    index   j; bool  switch; switch = ture; j = 1;
    while (j <i   &&  switch) {
        if (W[i][j]   &&  vcolor[i] == vcolor[j])     switch = false;
        j++;
      }
    return  switch;
}
```

الگوریتم عقبگرد برای مسئله مدارهای هاملیتونی

می خواهیم کلیه مدارهای هامیلتونی در یک گراف متصل و بدون جهت را مشخص کنیم. ورودی این الگوریتم، گراف بدون جهت حاوی nراس است که توسط یک آرایه دو بعدیW[1.n][1..n] نشان داده می شود. در این آرایه اگر بین راس i و j یالی وجود داشته باشد، W[i][j] برابر true است و در غیر اینصورت برابر false است.

خروجی ها: همهٔ مسیرهایی که از یک رأس مفروض آغاز شده اند، کلیهٔ رئوس موجود در گراف را دقیقاً یک بار ملاقات می کنند و به رأس شروع ختم می شوند. خروجی هر مسیر، آرایه ای از اندیس های vindex است که از صفر تا $n - 1$ اندیس گذاری شده اند و در آن vindex[1] اندیس رأس شروع، vindex[0] است.

```
void  hamilitonian  (index  i){
    index  j;
    if  ( promising(i))
        if (i == n-1)   cout < < vindex[0] through vindex[n-1];
        else for (j = 2; j <= n;  j++) {
                vindex[i+1] = j;
                hamiltonian(i +1);
            }
}
bool   promising (index i){
     index   j; bool   switch;
     if (i == n–1  && ! W[vindex[n – 1] ]  [vindex[0] ]
                switch = false;
     else if (i > 0 && ! W[vindex[i – 1] ]  [vindex[i] ]
                switch = false;
     else {
                switch = true;
```

```
            while (j < i && switch){
                if (vindex[i] = = vindex[j])
                    switch = false;
                j++;
             }
        }
  return  switch;
}
```

روش های کاهش استفاده از فضای اضافی در مرتب سازی سریع

1- در روال quicksort، تعیین می کنیم کدام زیر آرایه کوچک تر است و همواره آن را در پشته قرار مـی دهـیم و دیگـری را نگهـداری مـی کنیم. در این نسخه، بدترین حالت استفاده از فضا هنگامی رخ می دهد که partition آرایه را هر بار دقیقاً نصف کند و در نتیجه عمـق پشـته به $\lg n$ برسد. پس استفاده از فضای اضافی در بدترین حالت به $\theta(\lg n)$ اندیس تعلق دارد.

2- نسخه ای از partition وجود دارد که تعداد انتساب های رکوردها را به طرزی چشمگیر کاهش می دهد. این تعداد برابر است با:

$A(n) \approx 0{,}69(n+1) \lg n$

3- هر یک از فراخوانی های بازگشتی در روال quicksort باعث قرار گرفتن low, high و pivotpoint در پشته می شود. مقدار زیـادی از اعمال pop و push بی مورد است. در حالی که نخستین فراخوانی بازگشتی به quicksort پـردازش مـی شـود، فقـط کـافی اسـت مقـادیر pivotpoint و high در پشته نگهداری شوند. در حالی که دومین فراخوانی بازگشتی به quicksort پردازش می شـود، نیـاز بـه نگهـداری هیچ چیز نیست. می توانیم با نوشتن quicksort به شیوه تکرار و دستکاری پشته در روال، از عملیات بیهوده پرهیـز کنـیم. یعنـی بـه جـای استفاده از پشته و بازگشتی، پشته را خودمان می سازیم.

4- الگوریتم های بازگشتی نظیر مرتب سازی سریع را می توان با تعیین یک مقدار آستانه که در آن الگوریتم به جای تقسیم بیشتر نمونه، یک الگوریتم تکراری را فراخوانی می کند، بهبود بخشید.

5- این الگوریتم در موردی که ورودی یک آرایه از قبل مرتب شده باشد، دارای کمترین کارایی است. هرچه آرایه ورودی به حالت مرتب شـده نزدیک تر باشد، کارایی الگوریتم به بدترین حالت نزدیک تر است. بنابراین اگر دلیلی وجود داشته باشد که باور کنیم، آرایـه نزدیـک بـه حالـت مرتب است، می توانیم کارایی را به این ترتیب بهبود بخشیم که همواره نخستین عنصر را به عنوان عنصر محـوری انتخـاب نکنـیم. یـک روش خوب انتخاب میانه بین سه عنصر S[low]، S[mid] و S[high] برای نقطه محوری است. در این مورد، تضمین می شود یکی از زیر آرایه ها خالی نیست (اگر سه مقدار متمایز باشند).

روشهای بهبود مرتب سازی ادغامی

در زیر دو روش برای بهبود مرتب سازی ادغامی آورده شده است:

روش اول: اگر می خواستید مرتب سازی ادغامی را با دست انجام دهید، نیازی به تقسیم آرایه نمی باشد تا این که به آرایـه هـای تـک عضـوی برسید. می توان یک نسخه تکراری نوشت که این روش را دنبال کند و در نتیجه از سـربار عملیـات پشـته ای مـورد نیـاز بـرای پیـاده سـازی بازگشتی پرهیز کند. حلقه موجود در الگوریتم، انـدازه آرایـه را تـوانی از 2 در نظـر مـی گیـرد. آن مقـادیری از n کـه تـوانی از 2 نباشـند بـا $2^{\lceil \lg n \rceil}$ بارگذر از حلقه انجام می شوند ولی بر اساس n ادغام نمی شوند. در این الگوریتم، تعداد انتساب های رکوردها را حدودا از $2n\lg n$ به حدودا $n\lg n$ کاهش داده ایم. ثابت کرده ایم که پیچیدگی زمانی تعداد انتساب ها در بدترین حالت برای این الگوریتم تقریبـاً برابـر اسـت بـا :

$T(n) \approx n \lg n$

سپس رکوردها را در یک لیست پیوندی با تنظیم پیوندها به جای انتقال دادن رکوردها مرتب می کنیم. بنابراین نیازی به ایجاد یک آرایه اضافی از رکوردها نمی باشد. چون فضای اشغال شده توسط پیوند به طور چشمگیری کوچکتر از فضای لازم برای نگهداری خود رکورد بزرگ است، مقدار فضای صرفه جویی شده قابل ملاحظه است. به علاوه به خاطر آن که زمان لازم برای تنظیم پیوندها، کوچکتر از زمان لازم برای انتقال دادن رکوردهای بزرگ است، در زمان نیز صرفه جویی می شود. نتیجه این بهبود مرتب سازی ادغامی این است که نیاز به n رکورد اضافی را با n پیوند جایگزین می کند.

الگوریتم یافتن کوچکترین و بزرگترین کلیدها با جفت کردن کلیدها

هدف پیدا کردن کوچکترین و بزرگترین کلیدها در آرایه S[1..n] می باشد.

```
void  find( int  n ,const  keytype  S[ ], keytype&  small, keytype&  large){
      index   i;
      if (S[1]<S[2])  { small=S[1]; large=S[2];  }
      else   { small=S[2]; large=S[1];  }
      for ( i=3 ; i<= n-1 ; i =i+2) {
         if  ( S[i] > s[i+1])    exchange  S[i]  and  S[i+1];
         if  ( S[i] < small)     small=S[i];
         if  ( S[i+1] > large)  large=S[i+1];
       }
}
```

الگوریتم انتخاب

هدف یافتن کوچک‌ترین کلید kام در آرایه S با n کلید متمایز می باشد. $(k \leq n)$

```
keytype  selection (index  low, index  high, index  k){
   index   pivotpoint;
   if (low == high)
        return  S[low];
   else{
         partition(low, high, pivotpoint);
         if (k == pivotpoint)
               return  S[pivotpoint];
         else   if (k <pivotpoint)
                     return   selection(low, pivotpoint - 1, k);
               else
                     return   selection(pivotpoint + 1, high, k);
        }
}
void  partition (index low, index  high,index&  pivotpoint){
   index  i, j;  keytype  pivotitem;
   pivotitem = S[low];
```

```
  for (i= low + 1; i≤high; i++)
     if (S[i] <pivotitem)
          j++;
          exchange S[i] and Sj];
         }
  pivotpoint = j;
  exchange S[low] and S[pivotpoint];
}
```

روال partition ، یک آرایه را طوری افراز می کند که همه کلیدهای کوچک‌تر از یک عنصر محوری، پیش از آن و همه عناصر بزرگ‌تر، پس از آن قرار گیرند. محلی که عنصر محوری در آن قرار می‌گرفت pivotpoint نام داشت. می‌توانیم مسئله انتخاب کار را با افراز کردن تا زمانی که عنصر محوری در محل k ام قرار گیرد، حل کرد. اگر k کوچکتر از pivotpoint باشد، این کار را با افراز بازگشتی زیر آرایه چپی (کلیدهای کوچک‌تر از عنصر محوری) و اگر k بزرگ‌تر از pivotpoint باشد، این کار را با افراز زیر آرایه راستی انجام داد. زمانی کار تمام است که k= pivotponit . بدترین حالت وقتی رخ می‌دهد که ورودی هر فراخوانی بازگشتی حاوی یک عنصر کمتر باشد. برای مثال، این وضعیت هنگامی رخ می‌دهد که آرایه به ترتیب صعودی مرتب شده باشد و k=n باشد. این بدان معناست که پیچیدگی زمانی تعداد مقایسه‌های انجام شده در بدترین حالت عبارت است از:

$$w(n) = \frac{n(n-1)}{2}$$

به طور میانگین، الگوریتم انتخاب فقط تعدادی خطی از مقایسه‌ها را انجام می‌دهد. البته دلیل آن‌که الگوریتم در حالت میانگین کارایی بهتری از الگوریتم مرتب سازی سریع دارد، این است که مرتب سازی سریع، دوبار partition را فراخوانی می کند، حال آن که این الگوریتم تنها یک فراخوانی دارد. به هر حال، هر دو آن‌ها هنگامی که ورودی فراخوانی بازگشتی، n – 1 باشد، به پیچیدگی زمانی، درجه دوم است. حال می خواهیم از رخ دادن این رویداد جلوگیری نماییم.

بهترین حالت این است که pivotpoint آرایه را از وسط افراز کند، زیرا در آن صورت ورودی در هر فراخوانی بازگشتی نصف خواهد شد. به خاطر دارید که میانگین n کلید متمایز، کلیدی است که از نیمی از کلیدها بزرگ‌تر و از نیم دیگر کوچک‌تر باشد (این تعریف در صورتی دقیق است که n فرد باشد). اگر همواره می‌توانستیم برای pivotitem میانگین را انتخاب کنیم، بهترین کارایی حاصل می‌شد.

الگوریتم انتخاب با استفاده از میانگین

برای یافتن کوچک‌ترین کلید kام در آرایه S با n کلید متمایز می توان از الگوریتم زیر استفاده کرد:

```
keytype select (int  n, keytype  index )
 {
  return  selection2 (s, 1, n, k);
}

keytype selection۲ (keytype S[ ], index  low, index   high, index  k)
{
  if (high == low)
        return  S[low];
  else {
         partition۲(S, low, high, pivotpoint);
```

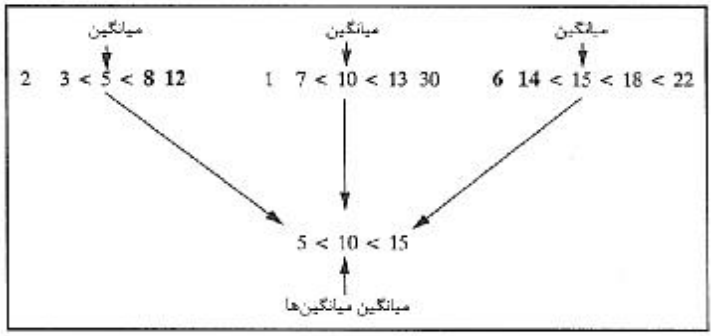
```
            return  S[pivotpoint];
        else
            if (k <pivotpoint)
                return  selection۲(S, low, pivotpoint - 1, k);
            else
                return  selection۲(S, pivotpoint + 1, high, k);
        }
}

void   partition۲ (keytype  S[ ], index  low, index  high, index&  pivotpoint)
{
 const  arraysize = high-low+1;
 const  r =  [arraysize / 5] ;
 index   i, j, mark ,first, last;
 keytype  pivotitem , T[1..r];
 for (i= 1; i<= r; i++)
    {
      first = low + 5 * i - 5;
      last = minimum(low + 5 * i - 1, arraysize);
      T[i] = median of  S[first] through S[last];
    }
 pivotitem = select(r, T ,  ⌊(r +1) / 2⌋);
 j = low;
 for (i= low; i<= high; i++)
     if (S[i] == pivotitem)
     {
      exchange S[i] and S[j];
      mark= j;
      j++ ;
     }
     else  if (S[i] <pivotitem)  { exchange S[i] and S[j]; }
 pivotpoint = j-1;
 exchange  S[mark] and S[pivotpoint];
}
```

در این الگوریتم، تابع ساده‌ای را نشان می دهیم که تابع بازگشتی ما را فراخوانی می‌کند. علتش این است که این تابع ساده باید در دو محل بـا ورودی‌های متفاوت فراخوانی شود. یعنی، در روال partition2 با T به عنوان ورودی فراخوانی می‌شود و به طور عمومی داریم:

```
Kthsmallest = select (n, s, k)
```

در روال partition، می‌توانیم فراخوانی تابع selection را با یک ورودی متشکل از آرایه اولیه و k، برابر با حدود نصف اندازه آن ارایه بیازماییم. همان طور که هم اکنون گفته شد، این نتیجه نخواهد داد، زیرا در نهایت selection دارای نمونه ای به اندازه نمونه اولیه خواهد بود. ولی راهبردی که در ادامه خواهد آمد، نتیجه می‌دهد. لحظه‌ای تصور کنید که n مضرب فردی از 5 باشد. N کلید را به n/5 گروه تقسیم می‌کنیم که هر یک حاوی پنج کلید است. میانگین هر یک از این گروهها را به طور مستقیم تعیین می‌کنیم. (این کار با 6 مقایسه انجام می‌گیرد.) می‌توانیم تابع selection را فراخوانی کنیم تا میانگین n/5 میانگین را تعیین کند. میانگین میانگین‌ها الزاماً میانگین n کلید نیست، ولی همانطور که شکل بعدی نشان می‌دهد، نزدیک به هم هستند.



در این شکل، کلیدهای واقع در طرف چپ کوچک‌ترین میانگین (کلیدهای 2و3) باید کوچک‌تر از میانگین میانگین‌ها و کلیدهای واقع در طرف راست بزرگ‌ترین میانگین (کلیدهای 18و22) باید بزرگ‌تر از میانگین میانگین‌ها باشند. کلیدهای واقع در طرف راست کوچک‌ترین میانگین (کلیدهای 8 و 12) و کلیدهای واقع در طرف چپ کوچک‌ترین میانگین (کلیدهای 6و14) می‌توانند در هر یک از طرفین میانگین میانگین‌ها قرار داشته باشند. تعداد کلیدهایی که می‌توانند در هر یک از طرفین میانگین میانگین‌ها واقع شوند، عبارت است از:

در این شکل، هر فلش یک کلید را نشان می‌دهد. نمی دانیم آیا کلیدهای پررنگ، کوچک‌تر یا بزرگ‌تر از میانگین میانگین‌ها هست یا خیر.

هرگاه n مضرب فردی از 5 باشد، تعداد کلیدهایی که می‌توانند در هریک از طرفین میانگین میانگین‌ها واقع شوند، برابر است با: $2(\frac{n}{5}-1)$

بنابراین، حداکثر تعداد کلیدهایی که می‌تواند در یک طرف از میانگین میانگین‌ها واقع شود عبارت است از:

$$\frac{1}{2}[n-1-2(\frac{n}{5}-1)]+2(\frac{n}{5}-1) = \frac{7n}{10}-\frac{3}{2}$$

تحلیل پیچیدگی زمانی الگوریتم انتخاب با استفاده از میانگین در بدترین حالت

عمل اصلی، مقایسه S[i] با عنصر محوری partition می باشد. برای سهولت،این فرض بازگشتی را می‌پذیریم که n مضرب فردی از 5 است.

دستور بازگشتی به صورت زیر است:

$$W(n) = W(\frac{7n}{10}-\frac{3}{2})+W(\frac{n}{5})+\frac{6n}{5}+n \approx W(\frac{7n}{10})+W(\frac{n}{5})+\frac{11n}{5}$$

- زمان در تابع selection2 هنگامی که از تابع selection2 فراخوانی می‌شود.

اگر n مضرب فردی از 5 باشد، حداکثر تعداد $\frac{7n}{10} - \frac{3}{2}$ کلید در یک طرف pivotpoint قرار دارد. یعنی که برای این فراخوانی selection2 ، این مقدار تعداد کلیدهای ورودی در بدترین حالت است.

- زمان در تابع selection2 هنگامی که از روال partition2 فراخوانی می‌شود.

تعداد کلیدها در ورودی این فراخوانی selection2 برقرارn/5 است.

- تعداد مقایسه‌های لازم برای یافتن میانگین‌ها.

میانگین پنج عنصر را می‌توان با انجام شش مقایسه به دست آورد. هنگامی که n مضربی از 5 باشد، الگوریتم، میانگین دقیقاً n/5 گروه پنج عضوی را می‌یابد. بنابراین، تعداد کل مقایسه‌های لازم برای یافتن میانگین‌ها، 6n/5 است.

- تعداد مقایسه‌های لازم برای افراز آرایه.(این تعداد برابر n است.)

تذکر: می‌توان نشان داد که تساوی تقریبی، برای nهایی که مضرب فردی از 5 نباشند نیز برقرار است.

الگوریتم های استفاده شده در heapsort

۱- الگوریتم siftdown

```
void siftdown ( heap&  H){
   node  parent, largerchild;
   parent = root of  H;
   largerchild = parent's  child containing  larger key;
   while  ( key at parent is smaller  than  key  at  largerchild ) {
      exchange  key at  parent and  key  at  largerchild;
      parent  = largerchild;
      largerchild  =  parent's  child  containing larger  key;
    }
}
```

۲- الگوریتم root

حذف کلید در ریشه با حفظ ویژگی heap .

```
keytype  root  (heap&  H){
  keytype   keyout; keyout = key at  the  root;
  move  the  key at the bottom node to the  root;
  delete the bottom node;
  siftdown(H);
  return   keyout;}
```

قرار دادن کلیدهای موجود در یک دنباله مرتب، درون آرایه S .(با فرض داشتن یک heap با n کلید)

```
void   removekeys ( int  n,  heap  H, keytype  s[ ]){
     index   i;
     for  ( i=n; i>=1; i--)
          s[i] = root(H);
}
```

۴- روال makeheap

```
void   makeheap (int  n , heap& H){
    index  i;
    heap   Hsub;
    for ( i= d-1; i>=0; i--)
      for ( all subtrees Hsub whose roots have depth  i )
           siftdown(Hsub);
}
```