



فصل سوم:

روش تقسیم و غلبه





فصل سوم

روش تقسیم و غلبه (Divide and Conquer)

مفهوم تکنیک تقسیم و غلبه

نام گذاری این روش از یک تکنیک جنگی که ناپلئون به کار برد گرفته شده است. در سال ۱۸۰۵ ارتشی از سربازان روسی، و اتریشی با بیش از ۱۵ هزار نفر به جنگ ناپلئون آمدند. در این حال ناپلئون به قلب سپاه آنها حمله کرده و با تقسیم نیروهای دشمن به دو بخش بر آنها پیروز شد. در واقع ناپلئون با تقسیم (Divide) سپاهی بزرگ به دو سپاه کوچکتر و پیروز شدن بر تک تک آنها موفق شد بر آن سپاه بزرگ غلبه یافته و آن را فتح کند (conquer).

در تکنیک تقسیم و غلبه (تقسیم و حل) یک نمونه از مسأله به دو یا چند نمونه کوچکتر تقسیم می‌شود. سپس این نمونه‌های کوچکتر حل شده و با ترکیب حل این نمونه‌های کوچکتر، حل مسأله اصلی حاصل می‌شود. اگر این نمونه‌های کوچکتر قابل حل نبودند آنقدر آنها را به نمونه‌های کوچکتر دیگر تقسیم می‌کنیم تا بالاخره حل شوند. روش تقسیم و غلبه در واقع یک روش حل بالا به پائین (Top-down) می‌باشد. اغلب این تکنیک به صورت بازگشتی پیاده‌سازی می‌شود. ولی ممکن است به صورت یک روال معمولی نیز پیاده‌سازی گردد. در ادامه این فصل با حل چند مسأله استاندارد به صورت تقسیم و غلبه این تکنیک را معرفی می‌کنیم.

مثال اول : جستجوی دودویی

جستجوی دودویی فقط در آرایه‌های مرتب استفاده می‌شود. در این روش عنصر مورد نظر با خانه وسط آرایه مقایسه می‌شود اگر با این خانه برابر بود جستجو تمام می‌شود اگر عنصر مورد جستجو از خانه وسط بزرگتر بود جستجو در بخش بالایی آرایه و در غیر اینصورت جستجو در بخش پائینی آرایه انجام می‌شود (فرض کرده‌ایم آرایه به صورت صعودی مرتب شده است) این رویه تا یافتن عنصر مورد نظر یا بررسی کل خانه‌های آرایه ادامه می‌پابد.

مثال : در لیست زیر می‌خواهیم بینیم عدد 44 در کدام خانه قرار دارد؟

1	2	3	4	5	6	7	8	9	10	11	12
12	20	25	27	29	30	33	44	45	67	78	80

↑ ↑ ↑

Low mid high

طراحی الگوریتم

با توجه به قضیه اصلی زیر که قبلاً بیان کردیم :

$$\begin{cases} T(n) = aT\left(\frac{n}{b}\right) + cn^k \\ T(1) \end{cases} \Rightarrow \begin{cases} T(n) = \Theta(n^{\log_b^a}) & \text{اگر } a > b^k \\ T(n) = \Theta(n^k \log_2 n) & \text{اگر } a = b^k \\ T(n) = \Theta(n^k) & \text{اگر } a < b^k \end{cases}$$

$$a = 1, b = 2, c = 1, k = 0 \Rightarrow a = b^k \Rightarrow 1 = 2^0 \Rightarrow$$

$$W(n) = \Theta(n^0 \log_2 n) = \Theta(\log n)$$

جواب رابطه بازگشتی $W(n)$ برای جستجوی دودویی به صورت دقیق برابر است با :

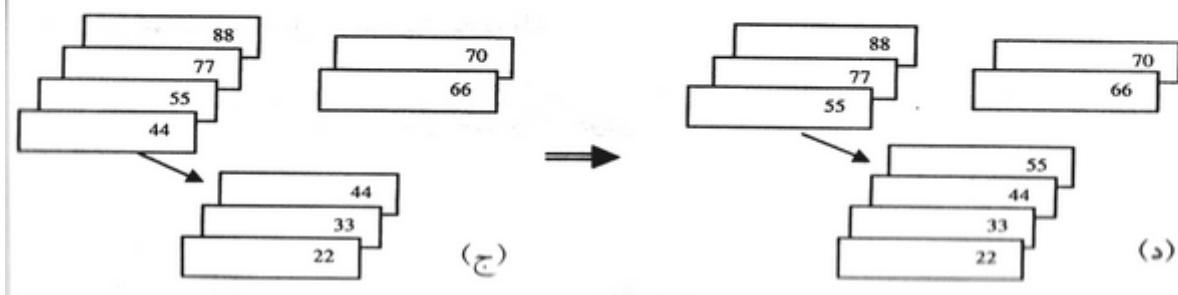
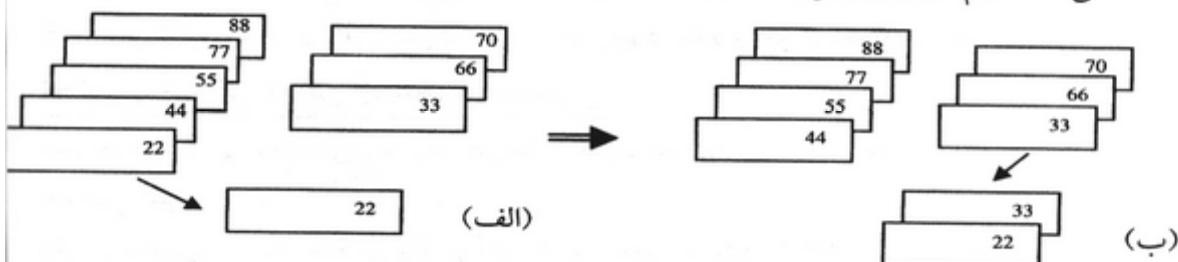
$$W(n) = \lfloor \log n \rfloor + 1 \Rightarrow W(n) = \Theta(\log n)$$

در تست‌ها علت فرمول $\lfloor \log n \rfloor + 1$ را می‌بینید.

مثال دوم : مرتب‌سازی ادغامی

قبل از توضیح مرتب‌سازی ادغام لازم است نحوه ترکیب دو لیست مرتب جدید بیان کنیم.

مثال : فرض کنید دو دسته کارت مرتب شده به شکل زیر داریم. در هر مرحله کارت جلویی هر دسته را با کارت جلویی دسته دیگر مقایسه می‌کنیم و کارت با شماره کوچکتر را در دسته جدید قرار می‌دهیم. هرگاه یکی از دسته‌ها خالی شد، تمام کارت‌های باقی‌مانده در دسته دیگر را در انتهای دسته کارت جدید قرار می‌دهیم :



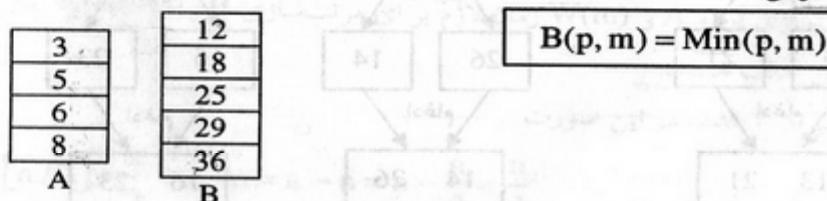
۱۰۷

فصل سوم : روش تقسیم و غلبه (Divide and Conquer)

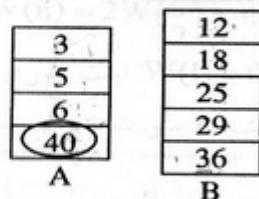
تابع زیر دو آرایه مرتب **A** با p عنصر و **B** با m عنصر را گرفته و یک آرایه **S** با $m + p$ عنصر مرتب شده را بر می گرداند. (شروع اندیس آرایه ها را 1 فرض کرده ایم) :

```
void merge (int p, int m, int A[ ], int B[ ], int S[ ])
{
    int i,j,k;
    i = j = k = 1;
    while (i <= p && j <= m) {
        if (A[i] < B[j]) { S[k] = A[i], i++; }
        else { S[k] = B[j]; j++; }
        k++;
    }
    if (i > p)
        whiel (j <= m)
        { S[k] = B[j]; k++; j++; }
    else if (j>m)
        while (i <= p)
        { S[k] = A[i] ; k ++; i++; }
}
```

در برنامه فوق عمل اصلی را مقایسه **A[i]** و **B[j]** در نظر می گیریم. بدین ترتیب بهترین حالت هنگامی رخ می دهد که همه عناصر آرایه **A** با طول کمتر از اولین عنصر آرایه **B** با طول بیشتر، کوچکتر باشد. مثل شکل زیر که تنها به 4 مقایسه نیاز دارد، بنابراین :



بدترین حالت نیز هنگامی رخ می دهد که همه عناصر آرایه اول (به جز آخرین عضو آن) از اولین عنصر آرایه دوم کوچکتر باشند. ولی این آخرین عضو از تمام عناصر آرایه دوم بزرگتر باشد. مثل شکل زیر:



که در این حالت به 8 مقایسه نیاز داریم. یعنی در بدترین حالت تعداد مقایسه ها برابر است با:

$$W(p,m) = p + m - 1$$

طراسی الگوریتم

ابتدا خانه وسطی (6) را با عدد 44 مقایسه می‌کنیم. چون 30 کمتر از 44 است پس نیمه بالایی آرایه یعنی از خانه 7 تا 12 را فقط نگاه می‌کنیم. برای این منظور Low را برابر $mid + 1$ قرار می‌دهیم:

7	8	9	10	11	12
33	44	45	67	78	80

↑ ↑ ↑
Low mid high

حال خانه وسط یعنی خانه 9 را که حاوی عدد 45 است را با 44 مقایسه می‌کنیم. چون 44 کمتر از 45 است پس نیمه پائین این آرایه را فقط نگاه می‌کنیم. برای اینکار high را برابر $mid - 1$ قرار می‌دهیم.

7	8
33	44

↑ ↑
Low high

حال خانه شماره 7 و سپس خانه شماره 8 را با کلید 44 مقایسه می‌کنیم و در می‌یابیم که عدد 44 در خانه شماره 8 قرار دارد. برنامه مربوط به جستجوی بازنی به صورت زیر است:

تابع زیر جستجوی بازنی را (به صورت غیر بازگشتی) انجام می‌دهد و مشخص می‌سازد عدد m در کدام خانه آرایه x وجود دارد. n طول آرایه مرتب شده x است. اگر عدد m در آرایه نباشد تابع مقدار -1 - برمی‌گرداند.

int bsearch (int x [], int n , int m)

```
{
    int low, high , mid;
    low = 0 ;    high = n - 1 ;
    while (low <= high ) {
        mid = (low + high) / 2 ;
        if ( m < x [ mid ] )
            high = mid - 1 ;
        else if (m > x [mid] )
            low = mid + 1 ;
        else return mid
    }
    return - 1 ;
}
```

نکته: مرتبه اجرایی الگوریتم فوق در حالت متوسط و بدترین حالت $O(\log_2 n)$ می‌باشد، چرا که هر بار نصف آرایه مورد بررسی قرار می‌گیرد و بازه عملیات در هر بار اجرای حلقه نصف می‌شود. در بهترین حالت که عدد مورد جستجو دقیقاً در وسط آرایه قرار گرفته باشد، تنها به یک عمل مقایسه نیاز بوده و در نتیجه مرتبه اجرایی الگوریتم فوق در بهترین حالت $O(1)$ است.



حال نسخه بازگشتی الگوریتم جستجوی دودویی را بیان می کنیم :

```
int bsearch (int low, int high, int x, int A[ ]) {
    int mid;
    if (low > high) return -1;
    else {
        mid = (low + high) / 2;
        if (x == A[mid]) return mid;
        else if (x < A[mid])
            return bsearch (low, mid -1, x, A);
        else return bsearch (mid+1, high, x, A);
    }
}
```

نکته ۱ : تابع بازگشتی فوق از نوع «بازگشت انتهایی» است یعنی پس از فراخوانی بازگشتی هیچ عملی انجام نمی شود. توابعی از این نوع را به سادگی می توان به نسخه تکراری (غیر بازگشتی) تبدیل کرد. قبل از نسخه تکراری تابع فوق را بیان کرده ایم. اغلب بهتر است تابع بازگشتی از نوع بازگشت انتهایی را به صورت نسخه تکراری بازنویسی و استفاده کنیم چرا که نسخه تکراری به دلیل عدم نیاز به پشته و نیز فراخوانی های مکرر در مصرف حافظه صرفه جویی کرده و همچنین سریع تر از نسخه بازگشتی است (البته به اندازه یک مضرب ثابت). در بعضی کامپایلرها نظیر برخی از نسخه های LISP کدهای بازگشتی انتهایی به صورت خودکار به کد تکراری ترجمه می شوند.

نکته ۲ : جستجوی دودویی ساده ترین الگوریتم تقسیم و غلبه است چرا که نمونه بزرگ تنها به یک نمونه کوچک تر شکسته شده و لذا ترکیبی از خروجی ها وجود ندارد و حل نمونه اولیه تنها حل نمونه کوچکتر است.

تحلیل پیچیدگی زمانی جستجوی دودویی

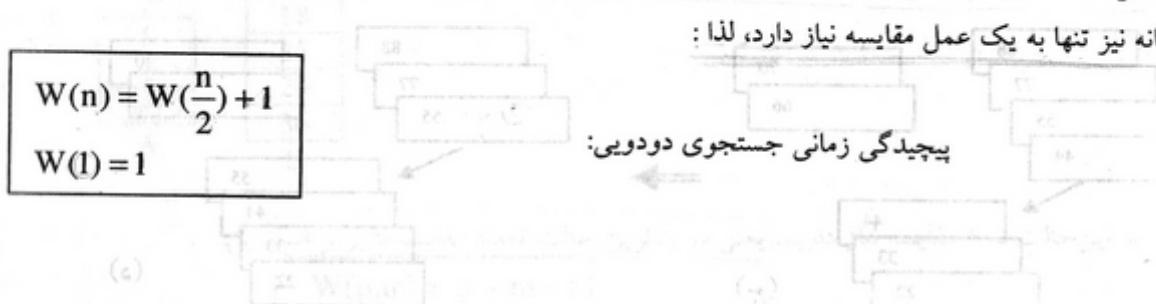
جستجوی دودویی پیچیدگی زمانی در هر حالت ندارد. لذا باید حالت های خوب، بد و متوسط آن را جداگانه حساب کنیم.

یکی از بدترین حالت ها هنگامی رخ می دهد که عدد مورد جستجو (x) آخرین (بزرگترین) عنصر آرایه باشد. در این حالت آرایه باید مرتباً نصف شود تا هنگامی که در نهایت آرایه یک خانه داشته باشد. آرایه ای با یک خانه نیز تنها به یک عمل مقایسه نیاز دارد، لذا :

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

$$W(1) = 1$$

پیچیدگی زمانی جستجوی دودویی:



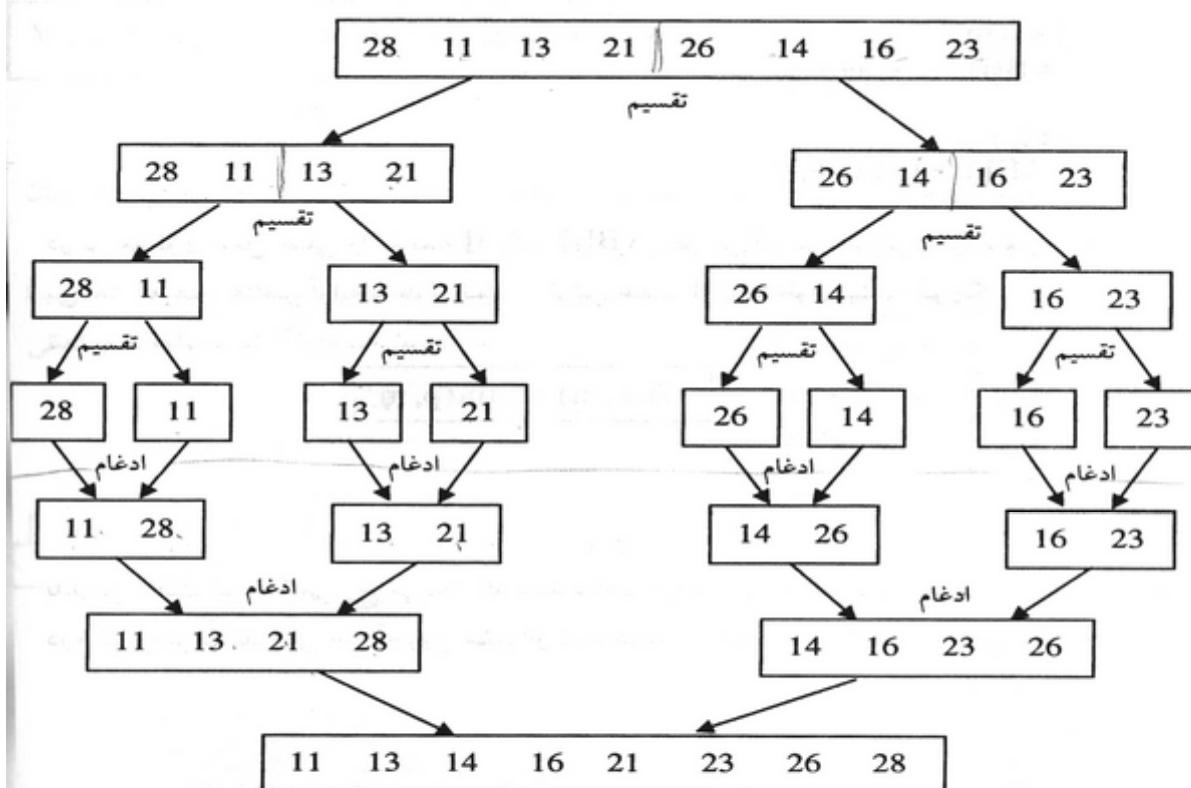
طراحی الگوریتم

ال که تابع **merge** را شرح دادیم به توضیح مرتبسازی ادغام می پردازیم. مرتبسازی آرایه n خانه‌ای با ش ادغام شامل مراحل زیر است :

- آرایه را به دو قسمت هر یک با $\frac{n}{2}$ عنصر تقسیم کن.

- با روش ادغام هر زیر آرایه را مرتب کن. اگر آرایه به قدر کافی کوچک نمی باشد، به صورت بازگشته آن را رچکتر و حل کن.

- از طریق الگوریتم ادغام زیر آرایه‌های مرتب شده را در یک آرایه مرتب، ترکیب کن.
یکل زیر مراحل این الگوریتم را برای یک آرایه فرضی نشان می دهد :



ا توجه به توضیحات فوق تابع مرتبسازی ادغامی را می توان به صورت زیر نوشت. آرایه S ، شامل n عنصر است که تابع **mergeSort** عناصر آن را مرتب کرده و مرتب شده آنها را در داخل خود S قرار می دهد. فرض کرد هایم آرایه ها از ۱ شروع می شود.



```
void mergesort (int n, int S [ ]) {
    int p =  $\lfloor n/2 \rfloor$ , m = n - p;
    int A[1 .. p], B[1 .. m];
    if (n > 1) {
        Copy S[1] through S[p] to A[1] through A[p];
        Copy S[p+1] through S[n] to B[1] through B[m];
        mergesort (P,A);
        mergesort (m, B);
        merge (p,m,A,B, S);
    }
}
```

همان طور که مشاهده می کنید در الگوریتم فوق تابع mergesort به صورت بازگشتی ۲ بار خودش را صدای زند.

تحلیل پیچیدگی زمانی مرتب سازی ادغامی در بدترین حالت

عمل اصلی را مقایسه در تابع ادغام (merge) در نظر می گیریم و می خواهیم تعداد این مقایسه را در بدترین حالت بر حسب اندازه ورودی n که تعداد عناصر آرایه S است به دست آوریم. با توجه به توضیحات مربوط به این الگوریتم داریم :

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

$W(p)$ زمان لازم برای مرتب سازی آرایه A و $W(m)$ زمان لازم برای مرتب سازی B و $(p+m-1)$ زمان لازم برای ادغام A و B در بدترین حالت هستند.

در ابتدا فرض می کنیم n توانی از ۲ باشد، در این صورت :

$$P = \lfloor n/2 \rfloor = \frac{n}{2}, \quad m = n - p = n - \frac{n}{2} = \frac{n}{2}$$

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

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

اگر اندازه ورودی 1 باشد (یعنی آرایه یک خانه‌ای باشد) نیازی به ادغام نبوده و $W(1) = 0$ است. پس رابطه پیچیدگی زمانی برای مرتب سازی ادغام هنگامی که $n > 1$ بوده و n توانی از ۲ باشد به صورت زیر است:

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

$$W(1) = 0$$



طراحی الگوریتم

به قضیه اصلی زیر :

$$T(n) = aT\left(\frac{n}{b}\right) + Cn^k = \begin{cases} T(n) = \Theta(n^{\log_b^a}) & \text{اگر } a > b^k \\ T(n) = \Theta(n^k \log_2 n) & \text{اگر } a = b^k \\ T(n) = \Theta(n^k) & \text{اگر } a < b^k \end{cases}$$

$$a = 2, b = 2, k = 1 \Rightarrow a = b^k \Rightarrow 2 = 2^1$$

این حالت داریم :

$$W(n) = \Theta(n \log n)$$

اثبات کرد حل دقیق رابطه بازگشتی (1) $W(n) = 2W\left(\frac{n}{2}\right) + (n - 1)$ است برابر $W(n) = n \log n - (n - 1) \in \Theta(n \log n)$ می‌باشد.

اگر n توانی از ۲ نباشد، آنگاه می‌توان اثبات کرد که :

$$W(n) = W\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + W\left(\left\lceil \frac{n}{2} \right\rceil\right) + (n - 1) = \Theta(n \log n)$$

ر هر حال مرتب‌سازی ادغام در بدترین حالت از مرتبه $n \log n$ است.

به به دو مثال قبلی می‌توانید راحت‌تر راهبرد « تقسیم و غلبه » که شامل مراحل زیر است را درک کنید : سیم مسأله به یک یا چند نمونه کوچکتر

نمونه‌های کوچکتر، اگر این نمونه‌ها به قدر کافی کوچک نبودند که حل شوند، از تکنیک بازگشتی کوچکتر کردن آنها استفاده می‌کنیم.

صورت نیاز ترکیب جواب نمونه‌های کوچکتر تا با این ترکیب جواب نمونه اولیه به دست آید. عبارت «ورت نیاز» در مرحله سوم به این دلیل است که ممکن است مانند روش جستجوی دودویی عمل ترکیب شته باشیم.

ل پیچیدگی مصرف حافظه در مرتب‌سازی ادغامی

ند که ما اغلب به تحلیل زمانی الگوریتم‌ها می‌پردازیم و نه تحلیل فضایی آنها ولی لازم است این نکته هم راحی الگوریتم‌ها مدنظر قرار گیرد.

کوئیتی که برای مرتب‌سازی ادغامی معرفی کردیم، در هر مرحله نیاز به حافظه کمکی به اندازه n خانه مثلاً در مرحله اول که آرایه S به دو آرایه A و B تقسیم می‌شود، جمع خانه‌های A و B برابر n است. بین ترتیب هنگامی که آرایه A به دو قسمت و آرایه B نیز به دو قسمت تقسیم می‌شود، جمع این ۴





فصل سوم : روش تقسیم و غلبه (Divide and Conquer)

قسمت برابر n خانه است. تعداد مراحل شکستن آرایه n $\log n$ بوده و در نتیجه مرتبه فضای مورد استفاده $\Theta(n \log n)$ است که نسبتاً زیاد می باشد.

الگوریتم های مرتب سازی را می توان به دو دسته درجا (inplace) و برون از جا (outplace) تقسیم بندی کرد. در روش های درجا فضای مورد استفاده وابسته به اندازه آرایه ورودی نبوده و از مرتبه $O(1)$ است ولی در الگوریتم های برون از جا فضای مورد استفاده (مانند مرتب سازی ادغامی) تابعی از اندازه ورودی است و هر چقدر تعداد عناصر آرایه (n) بیشتر شود، فضای کمکی مورد نیاز نیز افزایش می یابد.

می توان با بهینه سازی توابع mergesort و merge2 به صورت زیر مرتبه فضای مورد استفاده را از حالت A در دو آرایه B به $\Theta(n \log n)$ تقلیل داد. در این نسخه از پیاده سازی به جای آنکه آرایه ورودی S را در دو آرایه A و B کپی کنیم، به کمک اشاره گرهای low و mid و high آن را به صورت صعودی دو قسمت می کنیم:

```
void mergesort2 (int low, int high, int S[ ])
```

```
{
```

```
    int mid;
    if (low < high) {
        mid = ⌊ (low + high) / 2 ⌋;
        mergesort2 (low, mid, S);
        mergesort2 (mid + 1, high, S);
        merge2 (low, mid, high, S);
    }
}
```

```
/* ***** */
```

```
void merge2 (int low, int mid, int high, int S[ ])
```

```
{
```

```
    int i,j,k;
    int T[low .. high]; //A local array needed for the merging
    i = low; j = mid + 1; k = low;
    while (i <= mid && j <= high) {

```

```
        if (S[i] < S[j]) { T[k] = S[i]; i++; }
        else { T[k] = S[j]; j++; }
        k++;
    }
```

```
    if (i > mid) move S[j] through S[high] to T[k] through T[high];

```

```
    else move S[i] through S[mid] to T[k] through T[high];

```

```
    move T[low] through T[high] to S[low] through S[high];
}
```

در برنامه فوق برای ادغام آرایه S با n خانه به یک آرایه کمکی T با n خانه نیاز داریم لذا مرتبه فضای مورد

لیاز در مرتب سازی ادغام $\Theta(n)$ می باشد و این الگوریتم غیر درجا است.



طراحی الگوریتم

۱

۳. سوم : مرتب‌سازی سریع

بن روشن در هر گذر یک عنصر محوری یا لولا (pivot) انتخاب شده و سایر عناصر بردار به گونه‌ای جا می‌شوند که کلیه عناصر بزرگتر از آن در یک طرف و کلیه عناصر کوچکتر از آن در طرف دیگر عنصر قرار گیرند. بدین ترتیب عنصر لولا در مکان درست خود قرار گرفته است. سپس دو بردار دو طرف این را به صورت بازگشتی به همین روشن مرتب می‌شوند.

جه به توضیحات فوق این الگوریتم از نوع الگوریتم‌های تقسیم و غلبه است که با کوچک کردن بازه عمل ند بازه و سپس ترکیب جوابهای آنها، مسأله را حل می‌کند. این الگوریتم بر اساس جابه‌جا کردن زیاد سر عمل می‌کند.

ریتم زیر آرایه $x[1..x]$ را به صورت معده‌ی مرتب می‌کند. رویه به صورت Quicksort(x,1,n) صدا می‌شود.

```
procedure Partition (Var x: ArrayList; left, right : integer; Var pivotpoint : integer);
Var i,j,pivot : integer;
begin
    i := left ;   j := right+1;   pivot := x[left];
    repeat
        repeat
            i := i + 1;
        until x[i] >= pivot;
        repeat
            j := j - 1;
        until x[j] <= pivot;
        if i < j then swap (x[i], x[j]);
    until i>= j;
    swap (x[left], x[j]);
    pivotpoint := j;
end;
{ **** * * * * * * * * * * * * * * * * * * * * * }
procedure Quicksort (Var x: ArrayList; left, right : integer);
var pivotpoint : integer;
begin
    if (left < right) then begin
        partition (x , left, right, pivotpoint);
        QuickSort(x, left, pivotpoint - 1);
        QuickSort (x, pivotpoint+1, right);
    end; {if}
end;
```





۱۱۳

فصل سوم : روش تقسیم و غلبه (Divide and Conquer)

پردازه Partition آرایه را به گونه‌ای افزایش می‌کند که عنصر محوری (اولین عنصر سمت چپ آرایه) در مکان درست خود قرار گیرد و همچنین مکان قرارگیری آن در آرایه توسط متغیر خروجی pivotpoint برگردانده می‌شود.

مثال ۳ : فرض کنید آرایه اولیه به صورت زیر باشد انجام یک مرحله از الگوریتم فوق به صورت زیر است:

x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]	x[8]	x[9]	x[10]
26	5	37	1	61	11	59	15	48	19

در ابتدا $i=1$ و $j=10$ و $left=1$ و $pivot=26$ می‌شود. سپس از سمت چپ مرتبآ نزدیک شده تا هنگامی که به اولین عنصر بزرگتر از 26 (یعنی 37) برسیم. به همین ترتیب از سمت راست مرتبآ نزدیک می‌شود تا به اولین عنصر کوچکتر از 26 (یعنی 19) برسیم. در اینحال جای این دو خانه یعنی 37 و 19 عوض می‌شود. پس آرایه به صورت زیر درمی‌آید:

26	5	19	1	61	11	59	15	48	37
----	---	----	---	----	----	----	----	----	----

حال دوباره حلقه فوق را تکرار می‌کنیم اینبار عدد 15 با عدد 61 جایگذاشت.

26	5	19	1	15	11	59	61	48	37
----	---	----	---	----	----	----	----	----	----

در مرحله بعد حلقه، زیرا عدد 11 و زیرا عدد 59 اشاره می‌کند و شرط جلوی $j > i$ درست بوده و حلقه‌های repeat تمام می‌شوند.

26	5	19	1	15	11	59	61	48	37
----	---	----	---	----	----	----	----	----	----

در اینحال $x[left]$ که همان 26 است با $[j]x$ یعنی عدد 11 جایگذاشت.

11	5	19	1	15	26	59	61	48	37
----	---	----	---	----	----	----	----	----	----

همانطور که مشاهده می‌کنید پس از یکبار صدا زدن QuickSort تمام اعداد سمت راست عدد 26 از آن بزرگتر و تمام اعداد سمت چپ عدد 26 از آن کوچکتر هستند، یعنی عدد 26 در جای درست خود قرار گرفته است. حال هر یک از دو آرایه سمت راست و چپ 26 را به صورت مجزا با همان روش فوق مرتب می‌کنیم. بدین ترتیب به صورت بازگشتنی کل آرایه را می‌توان مرتب کرد.





طراحی الگوریتم

دول زیر مراحل کامل و خلاصه این الگوریتم برای آرایه مذکور پس از هر بار فرآخوانی Quick Sort داده شده است. علامت برآکت بخشی از زیر لیست‌ها را که هنوز مرتب نشده‌اند نشان می‌دهد.

[26	5	37	1	61	11	59	15	48	19
[11	5	19	1	15]	(26)	[59	61	48	37
[1	5]	(11)	[19	15]	26	[59	61	48	37]
(1)	(5)	11	[19	15]	26	[59	61	48	37]
1	5	11	(15)	(19)	26	[59	61	48	37]
1	5	11	15	19	26	[48	37]	(59)	[61]
1	5	11	15	19	26	37	48	59	[61]
1	5	11	15	19	26	37	48	59	61

نظرور که مشاهده می‌شود QuickSort هفت بار صدا زده می‌شود.

ثو ۱ : در این الگوریتم انتخاب عنصر لولا تأثیر مهمی در سرعت اجراء آن دارد. اگر این عنصر میانه به باشد، آرایه را به دو زیر آرایه تقریباً مساوی افزای خواهد نمود و می‌توان ثابت نمود در این شرایط بهترین روش داده است. اما اگر عنصر لولا کوچکترین یا بزرگترین عنصر آرایه باشد در این شرایط یکی از زیر بها تنهی خواهد شد و در این شرایط بدترین حالت روی داده است. از انجا که الگوریتم یافتن میانه برای بار افزایش کلی مرتبه الگوریتم خواهد شد، امکان استفاده از آن وجود ندارد. در عمل عنصر لولا صورت تصادفی انتخاب می‌شود. در اکثر کتابها برای سادگی عنصر اول آرایه را به عنوان عنصر لولا در نظر گیرند ولی هر عنصر دیگر تصادفی نیز می‌تواند انتخاب شود. یک روش مناسب دیگر برای انتخاب لولا میانه ۳ (Median of three) است که در آن میانه مقادیر کلیدهای اول، وسط و آخر، عنصر لولا می‌شود.
لأ برای کلیدهای 26، 61، 19 داریم :

$$\text{عنصر لولا} = \text{Midian} (26, 61, 19) = 26$$

ک پیاده‌سازی دیگر از تابع partition به صورت زیر است :

```
procedure partition (Var x: Arraylist; left, right : integer; Var pivotpoint : integer)
Var i,j, pivot : integer;
begin
    j := left;    pivot := x[left];
    for i := left +1 to right do
        if x[i] < pivot then begin
            j := j + 1 ;
            swap (x[i], x[j]);
        end;
        swap (x[left], x[j]);
        pivotpoint := j;
    end;
```



۱۱۵

فصل سوم : روش تقسیم و غلبه (Divide and Conquer)

نمونه زیر مراحل کار افزای را مطابق الگوریتم فوق نشان می دهد. عنصر لولا عدد 26 است.

26	5	37	1	61	11	59	15	48	19
	j	i							
26	5	37	1	61	11	59	15	48	19
		j	i						
26	5	1	37	61	11	59	15	48	19
			j		i				
26	5	1	11	61	37	59	15	48	19
				j			i		
26	5	1	11	15	37	59	61	48	19
					j				i
26	5	1	11	15	19	59	61	48	37
					j				i
19	5	1	11	15	26	59	61	48	37

$$i - n = (n)T$$

همان طور که مشاهده می کنید نتیجه زیر آرایه سمت چپ در این روش افزای با روش قبلی متفاوت است ولی در هر دو روش تمام اعداد سمت راست 26 از آن بزرگتر و تمام اعداد سمت چپ آن کوچکتر هستند.
روش دیگر جهت افزای آن است که از سمت راست به چپ حرکت کرده اولین عدد کوچکتر از لولا را که یافتیم با آن جایه جا می کنیم. سپس از چپ به راست حرکت کرده اولین عدد بزرگتر از لولا را یافته و با آن جایه جا می کنیم و این عملیات را آنقدر تکرار می کنیم تا عنصر لولا در مکان درست خود قرار گیرد:

26	5	37	1	61	11	59	15	48	(19)
19	5	37	1	61	11	59	15	48	26
19	5	26	1	61	11	59	15	48	37
19	5	15	1	61	11	59	26	48	37
19	5	15	1	26	11	59	61	48	37
19	5	15	1	11	26	59	61	48	37



علیل پیچیدگی زمانی مرتبسازی سریع

عمل اصلی در زیربرنامه افزار (Partition) را مقایسه $[i]x$ با $pivot$ در نظر بگیریم، در تمام روش‌های ته شده این تعداد برابر $1 - n$ است که n تعداد عناصر آرایه X می‌باشد. چرا که تمام عناصر به جز خود دور با $pivot$ مقایسه می‌شوند. لذا :

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

چیدگی زمانی الگوریتم افزار در تمامی حالات :

لی خود مرتبسازی سریع (QuickSort) فاقد پیچیدگی زمانی در هر حالت است لذا در بدترین حالت و حالت متوسط آن را تحلیل می‌کنیم.

و توان اثبات کرد بدترین حالت در مرتبسازی سریع هنگامی است که آرایه از قبل مرتب باشد چرا که بی‌برنامه Partition در هر بار مقدار left را به عنوان pivotpoint برمی‌گرداند و لذا آرایه به طور مکرر به سورت یک زیرآرایه خالی در سمت چپ و یک زیرآرایه با $1 - n$ عنصر در سمت راست افزار می‌شود. پس از:

$$W(n) = W(0) + W(n-1) + (n-1)$$

۱ - زمان لازم برای افزار و $W(0)$ زمان لازم برای مرتبسازی زیرآرایه سمت چپ (به طول صفر) است دیهی است که $W(0) = 0$ بوده و لذا پیچیدگی زمانی مرتبسازی سریع در بدترین حالت برابر است با :

$$W(n) = W(n-1) + (n-1) \quad : n > 0$$

$$W(0) = 0$$

ه کمک روش جایگذاری به سادگی اثبات می‌شود که برای مورد فوق :

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

پس مرتبسازی سریع در بدترین حالت از مرتبه $\Theta(n^2)$ است.

تذکر : می‌توان کاری کرد که بدترین حالت هنگامی که آرایه مرتب باشد، رخ ندهد، ولی پیچیدگی زمانی در

بدترین حالت همچنان $\frac{n(n-1)}{2}$ خواهد بود.



فصل سوم : روش تقسیم و غلبه (Divide and Conquer)

برای تحلیل پیچیدگی زمانی در حالت میانگین فرض می کنیم احتمال آنکه PivotPoint برگردانده شده توسط روال Partition، هر یک از اعداد ۱ تا n باشد، یکسان است و بدینه است که این احتمال برابر $\frac{1}{n}$ است.

پس داریم :

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

۱ - زمان لازم برای افزایش است. $\frac{1}{n}$ احتمال آنکه Pivotpoint برابر p باشد. $A(p-1)+A(n-p)$ زمان

میانگین برای مرتب‌سازی آرایه است، هنگامی که Pivotpoint برابر با p باشد.

می‌توان اثبات کرد حل معادله فوق برابر عبارت زیر می‌شود :

$$A(n) \approx 2(n + 1)Ln n \in \Theta(n \log n) \Rightarrow A(n) = \Theta(n \log n)$$

پس پیچیدگی زمانی در حالت میانگین برای مرتب‌سازی‌های سریع و ادغام مساوی بوده و هر دو برابر $\Theta(n \log n)$ هستند.

البته به بیانی ساده‌تر (ولی نه دقیق) می‌توان گفت که در حالت متوسط آرایه تقریباً نصب شده و

دو باره خودش را با نصف آرایه صدا می‌زنند یعنی رابطه زیر را داریم که $\Theta(n) \Theta(n)$ زمان افزایش است :

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \Rightarrow T(n) = \Theta(n \log n)$$

توجه کنید که به کمک قضیه اصلی $T(n) = \Theta(n \log n)$ به دست آمده است.

نکته : پیچیدگی روش‌های ادغام و سریع به صورت زیر است :

بدترین حالت	حالات متوسط	بهترین حالت	
$\Theta(n^2)$	$O(n \log n)$	$O(n \log n)$	مرتب‌سازی سریع
$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	مرتب‌سازی ادغام



طراحی الگوریتم

۱۱

ل چهارم : ضرب ماتریس‌های استراسن (Strassen)

ان طور که می‌دانید الگوریتم ساده ضرب ماتریس‌ها به صورت زیر است :

```
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];
    }
```

برنامه فوق دو ماتریس مربعی A و B را ضرب کرده و حاصل را در ماتریس C می‌بیند. هر سه ماتریس $n \times n$ هستند.

الگوریتم ساده فوق تعداد ضرب‌ها همواره برابر $T(n) = n^3$ است. تعداد جمع‌ها نیز در برنامه ساده فوق $T(n) = n^3 - n^2$ می‌باشد. با تغییر ساده زیر می‌توان تعداد جمع‌ها را کاهش داد و به $T(n) = n$ رساند:

```
for (i=1; i<= n; i++)
    for (j=1; j <= n; j++) {
        C[i][j] = A[i][1] * B[1][j]
        for (k=2; k <= n; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```

آن خلاصه در الگوریتم استاندارد ضرب ماتریس‌ها داریم :

$$\begin{aligned} &= n^3 = \Theta(n^3) \quad \text{تعداد ضرب‌ها} \\ &= n^3 - n^2 = \Theta(n^3) \quad \text{تعداد جمع‌ها} \end{aligned}$$

حال روش ضرب ماتریس‌ها را به صورت تقسیم و غلبه بیان می‌کنیم. اگر n توانی از 2 باشد می‌توان

$$\text{ماتریس‌های } A \text{ و } B \text{ را به چهار ماتریس کوچکتر که هر یک } \frac{n}{2} \times \frac{n}{2} \text{ هستند تقسیم نمود:}$$

$$\begin{bmatrix} \frac{n}{2} & \frac{n}{2} \end{bmatrix} \times \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

: C_{ij} های بالا را می‌توان با همان روش الگوریتم استاندارد ضرب ماتریس‌ها به دست آورد. یعنی :

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}, \quad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}, \quad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$



فصل سوم : روش تقسیم و غلبه (Divide and Conquer)

گر در الگوریتم بالا عمل اصلی را تعداد ضربها در نظر بگیریم، آنگاه تعداد ضربها برای ماتریس‌های $n \times n$ به ۸ عمل ضرب ماتریس‌های $\frac{n}{2} \times \frac{n}{2}$ نیاز خواهد داشت. از طرف دیگر بدیهی است که ضرب دو ماتریس 1×1 فقط به یک عمل ضرب اسکالر نیاز دارد، لذا برای الگوریتم ساده تقسیم و غلبه ضرب ماتریس‌ها داریم:

$$\left. \begin{array}{l} T(n) = 8T\left(\frac{n}{2}\right) \\ T(1) = 1 \end{array} \right\} \Rightarrow \Theta(n^3)$$

یادآوری: نتیجه فوق از قضیه زیر به دست آمد که در فصل قبلی بیان کرده بودیم:

$$\left. \begin{array}{l} T(n) = aT\left(\frac{n}{b}\right) \\ T(1) = 1 \end{array} \right\} \Rightarrow \begin{cases} \Theta(n^{\log_b a}) & (a \neq 1) \\ \Theta(\log_b n) & (a = 1) \end{cases}$$

همان‌طور که مشاهده می‌شود روش تقسیم و غلبه نیز مانند روش مستقیم از مرتبه $\Theta(n^3)$ بوده و مزیتی ندارد. ولی در سال ۱۹۶۹ استراسن (Strassen) الگوریتمی را معرفی کرد که تعداد ضرب‌های آن از n^3 کمتر بوده و تقریباً $n^{2.81}$ بود که آن را در ادامه شرح می‌دهیم.

استراسن اثبات کرد که حاصل ضرب دو ماتریس $A \times B$ یعنی ماتریس C را می‌توان از روابط زیر به دست آورد (برای حالت $n = 2$ به سادگی می‌توانید آن را اثبات کنید):

$$C = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}$$

که M_i ها از فرمول‌های زیر به دست می‌آیند:

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M_2 = (A_{21} + A_{22})B_{11}$$

$$M_3 = A_{11}(B_{12} - B_{22})$$

$$M_4 = A_{22}(B_{21} - B_{11})$$

$$M_5 = (A_{11} + A_{12})B_{22}$$

$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$





طراحی الگوریتم

۱۰

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

برای ماتریس‌های زیر :

دار M_1 به صورت زیر محاسبه می‌شود :

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22}) =$$

$$\left(\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \right) \times \left(\begin{bmatrix} 8 & 9 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix} \right) = \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix} = \begin{bmatrix} 86 & 75 \\ 278 & 227 \end{bmatrix}$$

حفظ کردن فرمول‌های فوق لازم نیست ولی از فرمول‌های فوق نتیجه می‌شود که این تکنیک به ۷ عمل ضرب و ۱ عمل جمع و تفریق نیاز دارد در حالی که روش معمولی به ۸ عمل ضرب و ۴ عمل جمع نیاز داشت. پس با این تکنیک یک عمل ضرب کم شده ولی در عوض ۱۴ عمل جمع و تفریق اضافه شده است. مشاهده خواهید نزد که برای مقادیر بزرگتر n ، در کل زمان مصرفی کاهش می‌یابد. اینها ایست که برای حالت $n=1$ یعنی ضرب ماتریس‌های 1×1 تنها به یک عمل ضرب و صفر عمل جمع تفریق نیاز داریم.

ا توجه به توضیحات فوق :

$$T(n) = 7T\left(\frac{n}{2}\right) + n^{\log_2 7} = n^{2.81} \quad \text{تعداد ضرب‌ها در استراسن}$$

و به همین ترتیب تعداد جمع‌ها برابر است با :

$$T(n) = 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \quad \text{تعداد جمع‌ها در استراسن}$$

توجه کنید هنگامی که دو ماتریس $\frac{n}{2} \times \frac{n}{2}$ با هم جمع یا تفریق می‌شوند، تمام خانه‌های متناظر آنها باید با هم جمع یا تفریق شوند لذا ماتریس حاصل $\frac{n}{2} \times \frac{n}{2}$ به تعداد $(\frac{n}{2})^2$ خانه دارد که به این میزان عمل جمع یا تفریق اسکالر نیاز است.

می‌توان اثبات کرد (از اثبات صرف‌نظر می‌کنیم) که حل رابطه فوق به صورت زیر می‌شود:

$$T(n) = 6^{\log_2 7} - 6n^2 = 6n^{2.81} - 6n^2 \in \Theta(n^{2.81})$$





فصل سوم : روش تقسیم و غلبه (Divide and Conquer)

جدول زیر روش استراسن را با روش استاندارد مقایسه می کند :

عمل	روش استاندارد	روش استراسن
ضرب	n^3	$n^{2.81}$
جمع / تفریق	$n^3 - n^2$	$6n^{2.81} - 6n^2$

بنابراین روش استراسن کارآمدتر از روش استاندارد است. نکته : می توان اثبات کرد ضرب ماتریس ها حداقل به زمان $\Theta(n^2)$ نیاز دارد. ولی هنوز کسی توانسته است الگوریتمی از مرتبه ۲ برای ضرب ماتریس ها ابتداء کند و از طرف دیگر تا کنون نیز کسی توانسته اثبات کند نوشتمن چنین الگوریتمی غیرممکن است.

مثال پنجم : ضرب اعداد صحیح بزرگ

همان طور که می دانید اعداد صحیح بزرگ را می توان توسط آرایه ذخیره کرد. مثلاً عدد بزرگ 8,173,963,563 را به صورت زیر ذخیره می کنیم :

a[10]	a[9]	a[8]	a[7]	a[6]	a[5]	a[4]	a[3]	a[2]	a[1]
8	1	7	3	9	6	3	5	6	3

برای تعیین مثبت یا منفی بودن عدد هم کافی است آخرین خانه سمت چپ را به عنوان علامت درنظر بگیریم که اگر مثلاً صفر باشد یعنی مثبت و اگر 1 باشد یعنی منفی.

الگوریتم های جمع و تفریق اعداد صحیح بزرگ بسیار ساده بوده و از مرتبه $O(n)$ می باشد که به عنوان تعریف بهتر است آنها را بتوانیم برش می کنیم. بحث ما در اینجا بر سر ضرب اعداد صحیح بزرگ است که در حالت مستقیم معمولی از مرتبه $O(n^2)$ می باشد، چرا که هر یک از ارقام عدد دوم در تمامی ارقام عدد اول ضرب می شود. در ادامه یک روش تقسیم و غلبه برای ضرب اعداد صحیح بزرگ ارائه می کنیم.

در حالت کلی عدد صحیح u با n رقم را می توان به صورت زیر نشان داد :

$$u = x \times 10^m + y$$

که x حاوی m رقم و y حاوی $\left\lceil \frac{n}{2} \right\rceil$ رقم است. مثلاً :

$$967345 = 967 \times 10^3 + 345$$

$$8967345 = 8967 \times 10^3 + 345$$

حال دو عدد صحیح u و v را به صورت زیر در نظر بگیرید :

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

ضرب این دو عدد به صورت زیر خواهد بود:





طراحی الگوریتم

۱۲۲

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

پس ضرب uv در 7 به چهار عمل ضرب روی اعداد صحیح با نیمی از ارقام نیاز دارد. توجه کنید که ضرب یک

عدد در 10^m یا 10^{2m} به سادگی در زمان $O(n)$ صورت می‌پذیرد. مثلاً:

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

پس ما مرتب اعداد را تقریباً نصف می‌کنیم و آنها را در هم ضرب می‌کنیم. این عملیات تقسیم کردن را آنقدر ادامه می‌دهیم تا هنگامی که به یک مقدار آستانه (threshold) برسیم که در آن عمل ضرب به طریق استاندارد انجام پذیرد، مثلاً ممکن است در یک کامپیوتر عمل ضرب اعداد چهار رقمی به شیوه معمول سریع‌تر از روش تقسیم و غلبه فوق باشد.

پیاده‌سازی الگوریتم ضرب فوق به صورت زیر است :

```
Large_int prod(Large_int u, Large_int v)
{
    Large_int x,y,w,z;
    int n,m;
    n = Maximum (number of digits in u, number of digits in v)
    if (u == 0 || v == 0) return 0;
    else if (n <= threshold)
        return u * v;
    else {
        m = ⌊n / 2⌋;
        x = u divide 10m; y = u rem 10m;
        w = v divide 10m; z = v rem 10m;
        return prod(x,w) × 102m + (prod(x,z) + prod(w,y)) × 10m + prod(y,z);
    }
}
```

توجه کنید عملیات 10^m rem یعنی باقی‌مانده تقسیم بر 10^m و عملیات divide 10^m یعنی تقسیم صحیح بر 10^m که به راحتی در زمان $O(n)$ قابل پیاده‌سازی هستند. از آنجا که در الگوریتم فوق تابع prod n بار خودش را با اندازه $\frac{n}{2}$ صدا می‌زند و عملیات جانبی مثل جمع و تفریق و rem و divide همگی از مرتبه $O(n)$ هستند پس برای الگوریتم فوق، پیچیدگی زمانی از رابطه زیر به دست می‌آید (C یک ضریب ثابت است):

$$\left. \begin{array}{l} W(n) = 4W\left(\frac{n}{2}\right) + Cn \\ W(s) = 0 \end{array} \right\} \Rightarrow W(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$$



فصل سوم : روش تقسیم و غلبه (Divide and Conquer)

جواب فوق از قضیه اصلی در فصل قبلی به دست آمده است. مقدار S همان مقدار آستانه است که به ازای آن دیگر نمونه تقسیم نمی شود.

الگوریتم فوق از درجه n^2 بوده و در نتیجه کند است. با یک تغییر می توان مرتبه اجرای الگوریتم را پائین نر آورد. همان طور که در بالا مشاهده کردید تابع prod باید موارد زیر را محاسبه کند :

$xw, xz + yw, yz$

یعنی تابع prod بایستی چهار بار برای محاسبه xw, xz, yw و yz صدا زده شود. حال روابط مذکور را قدری دستکاری می کنیم :

$$xz + yw = (x+y)(w+z) - xw - yz$$

یعنی ابتدا xw و yz را به دست آورده و سپس به کمک عبارت سمت راست رابطه فوق مقدار $xz+yw$ را با یک عمل ضرب به دست می آوریم. سپس در کل به جای ۴ ضرب به ۳ ضرب نیاز خواهیم داشت. لذا:

$$\left. \begin{array}{l} W = 3W\left(\frac{n}{2}\right) + Cn \\ W(s) = 0 \end{array} \right\} \Rightarrow W(n) = \theta(n^{\log_2 3})$$

مثال ششم : یافتن بزرگترین و کوچکترین کلیدها

همان طور که می دانید الگوریتم استاندارد یافتن بزرگترین عنصر (و شبیه آن کوچکترین عنصر) در یک آرایه S ها عنصر به صورت زیر است :

```
max = S[1];
for (i=2 ; i <= n; i++)
    if (S[i] > max)
        max = S[i];
```

اندیس های آرایه S را از 1 تا n در نظر گرفته ایم. بدیهی است که اگر عمل اصلی را تعداد مقایسه کلیدها در نظر بگیریم آنگاه این تعداد برابر است با :

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

می توان قضیه زیر را اثبات کرد :

هر الگوریتمی که بتواند بزرگترین کلید را بین n کلید (برای هر ورودی دلخواه) فقط با مقایسه کلیدها بیابد، باید در هر حالت حداقل $n - 1$ مقایسه را انجام دهد.

توجه کنید که اگر مثلاً آرایه مرتب شده باشد، بزرگترین کلید بدون هیچ مقایسه ای به دست می آید، لذا در قضیه فوق روی عبارت «هر ورودی دلخواه» تأکید شده است.

حال اگر بخواهیم به شبیه مستقیم، بزرگترین و کوچکترین عنصر یک آرایه را پیدا کنیم، یک الگوریتم ساده به صورت زیر است :

فصل سوم : روش تقسیم و غلبه (Divide and Conquer)

مقایسه نیز در ابتدای کار و بیرون حلقه انجام می‌پذیرد لذا مرتبه اجرایی برنامه فوق در تمامی حالات برابر است

با :

$$T(n) = 1 + \left(\frac{n}{2} - 1\right) \times 3 \Rightarrow T(n) = \frac{3n}{2} - 2$$

به عنوان تمرین برنامه فوق را برای n های فرد بازنویسی کرده و نشان دهید که برای n های فرد پیچیدگی

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

زمانی برابر است با :

لذا در کل داریم :

$$T(n) = \begin{cases} \frac{3n}{2} - 2 & \text{اگر } n \text{ زوج باشد} \\ \frac{3n}{2} - \frac{3}{2} & \text{اگر } n \text{ فرد باشد} \end{cases}$$

قضیه : هر الگوریتمی که بتواند کوچکترین و بزرگترین کلیدها را در بین n عنصر از هر فرودی ممکن، تنها با مقایسه کلیدها پیدا کند در بدترین حالت، حداقل به تعداد مقایسه‌های فوق نیاز خواهد داشت.

حال با روش تقسیم و غلبه مسئله بالا را حل می‌کنیم. یعنی آرایه اولیه را به دو آرایه هر کدام با $\frac{n}{2}$ عنصر تقسیم می‌کنیم. آنگاه بزرگترین و کوچکترین عنصر هر دو آرایه را می‌یابیم. بعد از آن بین دو عدد بزرگترین، بزرگتر را و بین دو عدد کوچکترین، کوچکتر را به عنوان جواب برگزاییم.

در برنامه زیر آرایه S حاوی اندیس‌های low .. high بوده و خروجی تابع توسط پارامترهای \min و \max برگردانده می‌شود :

```
find (low, high, S, max, min)
{
    if (low == high) { max=min=S[low]; return; }
    if (low == high - 1)
    {
        if (S[low] < S[high])
            { max = S[high]; min = S[low]; }
        else { max = S[low]; min = S[high]; }
        return ;
    }
    mid = (low + high) / 2 ;
    find (low, mid, S, max1, min1);
    find (mid+1, high, S, max2, min2);
    max = Maximum (max1, max2);
    min = Minimum (min1, min2);
}
```



بیچیدگی زمانی الگوریتم فوق از رابطه بازگشتی زیر به دست می‌آید :

$$T(n) = \begin{cases} 0 & \text{اگر } n = 1 \\ 1 & \text{اگر } n = 2 \\ T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2 & \text{در غیره} \end{cases}$$

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

با حل رابطه فوق به نتیجه زیر می‌رسیم :

$$T(n) = \begin{cases} \frac{3n}{2} - 2 & \text{اگر } n = 2k \\ \frac{3n}{2} - \frac{3}{2} & \text{اگر } n = 2k + 1 \end{cases}$$

مثال هفتم : مسئله انتخاب

مسئله انتخاب (Selection) پیدا کردن k امین بزرگترین کلید یا k امین کوچکترین کلید در لیستی از n کلید می‌باشد. البته در این مسئله فرض می‌کنیم آرایه نامرتب است چرا که اگر آرایه مرتب باشد به سادگی جواب به دست می‌آید.

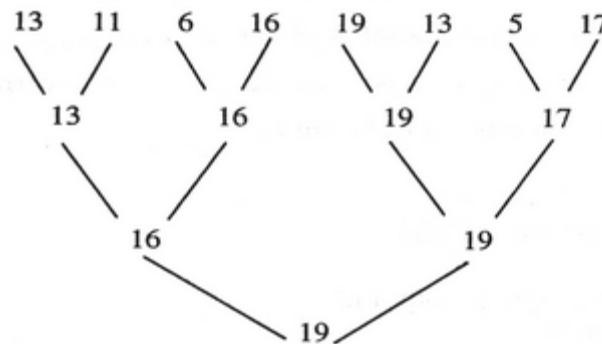
یک راه ساده برای حل مسئله فوق آن است که ابتدا لیست را مرتب کرده و سپس k امین عنصر را استخراج کنیم که در این حالت مرتبه الگوریتم $\Theta(n \log n)$ خواهد بود. ولی در ادامه روش‌هایی سریع‌تر که از مرتبه $\Theta(n)$ هستند را معرفی می‌کنیم. در حالت کلی این مسئله را می‌توان در زمان $\Theta(n)$ حل کرد.

حالت خاص اول که $k = 1$ باشد همان ماکزیمم یا مینیمم یا بی است که در قسمت قبلی حل کردیم. لذا ابتدا حالت خاص دوم که $k = 2$ است را در نظر می‌گیریم یعنی یافتن بزرگترین کلید دوم.

یافتن بزرگترین کلید دوم

یک روش ساده آن است که با الگوریتم استاندارد یافتن بزرگترین کلید در زمان $T_1(n) = n-1$ عنصر ماکزیمم را یافته آن را کنار بگذاریم، سپس در زمان $T_2(n) = n-2$ بزرگترین عنصر را در $n-1$ داده باقی مانده به دست آوردیم که در کل $T(n) = 2n-3$ می‌شود. ولی با تکنیک تورنمانت می‌توان این زمان را کاهش داد.

روش تورنمانت همان روشی است که در مسابقات ورزشی حذفی استفاده می‌شود. فرض کنیم ۸ عدد به صورت درخت زیر با یکدیگر مسابقه دهند و هر بار برنده عدد بزرگتر باشد :



برای حالت $n = 8$ به سه دور مسابقه و در کل ۷ مسابقه نیاز است و در حالت کلی $\lceil \log_2 n \rceil$ دور مسابقه و $n-1$ مسابقه نیاز داریم. با توجه به شکل فوق مشخص است که برندۀ دور آخر، حتماً بزرگترین کلید است ولی بازنده این دور الزاماً بزرگترین کلید دوم نیست. در شکل فوق بازنده دور آخر ۱۶ است در حالی که دومین کلید بزرگتر ۱۷ می‌باشد.

برای یافتن بزرگترین کلید دوم می‌توانیم کلیدهایی را که به بزرگترین کلید باخته‌اند در نظر بگیریم و سهیم با الگوریتم ساده، ماکریتم یابی بزرگترین آنها را پیدا کنیم. مثلاً در شکل فوق باید ماکریتم لیست [13, 17, 16] را بیابیم که ۱۷ می‌شود. از آنجا که تعداد عناصر لیست $\lceil \log n \rceil$ می‌باشد پس تعداد مقایسه‌های لازم در این لیست $\lceil \log_2 n \rceil - 1$ است. از طرف دیگر برای یافتن بزرگترین عدد (در مثال فوق عدد ۱۹) به $n-1$ مقایسه نیاز داشتیم لذا تعداد کل مقایسه‌های لازم برای یافتن کلید دوم برابر است با :

$$T(n) = (n-1) + \lceil \log_2 n \rceil - 1 = n + \lceil \log_2 n \rceil - 2$$

قضیه : هر الگوریتمی که بتواند بزرگترین کلید دوم را برای هر ورودی دلخواه تنها با مقایسه کلیدها پیدا کند، باید در بدترین حالت حداقل تعداد مقایسه‌های زیر را انجام دهد :

$$n + \lceil \log_2 n \rceil - 2$$

یافتن کوچکترین کلید k ام

برای حل این مسئله از تابع **partition** که در الگوریتم مرتب‌سازی سریع بیان کردیم، استفاده می‌کنیم. روال آرایه ورودی را به گونه‌ای افزار می‌کرد که همه کلیدهای کوچکتر از عنصر محوری، قبل از آن و همه عناصر بزرگتر از محور، بعد از آن قرار می‌گرفتند. مکانی که عنصر محوری در آن قرار می‌گرفت را **pivotpoint** نامگذاری کرده بودیم. حال با تکنیک تقسیم و غلبه مسئله انتخاب را با افزار کردن تا هنگامی که



طراحی الگوریتم

۱۲۸

عنصر محوری در محل k ام قرار گیرد، حل می‌کنیم. اگر k کوچکتر از pivotpoint باشد، زیرآرایه سمت چپ را به صورت بازگشته افزایش می‌کنیم و اگر k بزرگتر از pivotpoint باشد، زیرآرایه سمت راست را افزایش می‌کنیم. هنگامی که $k = \text{pivotpoint}$ شد، کار تمام است. الگوریتم مورد نظر به صورت زیر است:

```
int selection (int x[ ], int left, int right, int k)
{
    int pivotpoint;
    if (left == right)    return x[left];
    else {
        partition (x, low, high, pivotpoint);
        if(k == pivotpoint)
            return x[pivotpoint];
        else if(k < pivotpoint)
            return selection (x, left, pivotpoint-1, k);
        else
            return selection(x, pivotpoint +1 , right, k);
    }
}

void partition (int x[ ], int left, int right, int & pivotpoint)
{
    int i,j,pivot;
    j=left;  pivot = x[left];
    for (i=left+1; i <= right ; i++)
        if (x[i] < pivot) {
            j++;
            swap (x[i] , x[j]);
        }
    swap (x[left], x[j]);
    pivotpoint = j;
}
```

توجه کنید روال **partition** دقیقاً همان روال گفته شده در مرتب‌سازی سریع است فقط به زبان C نوشته شده است. در زبان C++ متغیر خروجی با علامت **&** مشخص می‌شود که معادل Var پاسکال است. پس در خط اول عنوان تابع **partition** عبارت **int & pivotpoint** به این معناست که خروجی این تابع توسط متغیر **pivotpoint** برگردانده می‌شود. در اینجا نیز بدترین حالت وقتی رخ می‌دهد که آرایه به ترتیب صعودی مرتب باشد و $k=n$ را بخواهیم. در این حالت ورودی هر فراخوانی بازگشته حاوی یک عنصر کمتر است و بنابراین

مشابه بدترین حالت مرتب‌سازی سریع $W(n) = \frac{n(n-1)}{2}$ خواهد بود. بهترین حالت نیز هنگامی رخ می‌دهد که **pivotpoint** آرایه را از وسط افزایش کند چرا که در این صورت ورودی در هر فراخوانی بازگشته نصف مرتب‌سازی می‌توان ثابت کرد. متوسط الگوریتم فوق $A(n) = 3n = \Theta(n)$ است.





فصل سوم : روش تقسیم و غلبه (Divide and Conquer)

با آنکه مرتب‌سازی در حالت متوسط از مرتبه $\Theta(n \log n)$ بود ولی الگوریتم فوق در حال متوسط از مرتبه $\Theta(n)$ می‌باشد علت آن است که تابع مرتب‌سازی سریع در هر بار دوبار خودش را فراخوانی می‌کند ولی تابع selection تنها یک فراخوانی دارد.

تذکر : با آنکه الگوریتم فوق در بدترین حالت $\Theta(n^2)$ بود ولی می‌توان با اصلاح کردن آن کاری کرد که مسئله انتخاب در بدترین حالت از مرتبه $\Theta(n)$ یعنی خطی باشد.
هايفيل (1976) اثبات کرد که حد پائين برای يافتن k امين کوچکترین کلید در مجموعه n کلیدی (برای $k > 1$) به صورت زير می‌باشد :

$$n + (k-1) \left\lceil \log_2 \left(\frac{n}{k-1} \right) \right\rceil - k \in \Theta(n)$$

توجه کنید که برای $2^k = n$ قبلاً اثبات کردیم $W(n) = n + \lceil \log_2 n \rceil - 2$ است که حالت خاصی از فرمول بالا می‌باشد.

کجا نباید از تقسیم و غلبه استفاده کرد

در صورت امکان در موارد زیر از روش تقسیم و غلبه استفاده نمی‌کنیم چرا که مرتبه الگوریتم نمایی می‌شود:

۱- هنگامی که نمونه‌ای با اندازه n به دو یا چند نمونه تقسیم شود که اندازه آنها نیز تقریباً برابر n است، مثلًا:

$$T(n) = 2T(n-1) \Rightarrow T(n) = \Theta(2^n)$$

۲- هنگامی که نمونه‌ای با اندازه n ، تقریباً به n نمونه با اندازه $\frac{n}{c}$ تقسیم شود که c یک مقدار ثابت است.

توجه کنید که همه مسأله را نمی‌توان به نمونه‌های کوچکتر تقسیم کرد و سپس نتایج حاصله را با هم ترکیب نمود. مثلًا اگر بخواهیم بزرگترین عدد بین ۲۰ عدد را پیدا کنیم می‌توان آن را به دو دسته ۱۰ تایی تقسیم کرد، سپس ماکزیمم هر دسته را یافته و با مقایسه این دو بزرگترین را پیدا کرد. ولی مثلًا در مورد یک دستگاه ۲۰ معادله ۲۰ مجھولی نمی‌توان آن را به دو دستگاه کوچکتر ۱۰ معادله ۱۰ مجھول تقسیم کرد.