۱ پیچیدگی الگوریتم

۱-۱ - مقدمه

این قسمت شما را با قالب آنچه که در طول این کتاب در مورد طراحی و تحلیل الگوریتم استفاده خواهیم کرد، آشنا می‌کند.

ما با الگوریتم *insertion – sort* (مرتب‌سازی به روش درج) شروع می‌کنیم. برای نشان دادن چگونگی الگوریتممان از شبه کد استفاده می‌کنیم. بعد از اینکه الگوریتم را مشخص کردیم استدلال می‌کنیم که الگوریتم به درستی مرتب می‌کند و ما زمان اجرای آن را تحلیل می‌کنیم.

این تحلیل نشان می‌دهد که چگونه زمان اجرا متناسب با تعداد عناوینی که باید مرتب شوند، افزایش پیدا می‌کند. ما به دنبال توضیح *insertion-sort* رویکرد تقسیم و حل را برای طراحی الگوریتم‌ها معرفی می‌کنیم و از آن برای گسترش دادن یک الگوریتم که *Mergesort*است استفاده میکنیم و این قسمت را به پایان می‌رسانیم.

Insertion-Sort

ورودی:یک رشته از n عدد $a_1, a_2, \ldots, a_n$

خروجی: یک نظمی از $\{a'_1, a'_2, \ldots, a'_n\}$ از رشته ورودی به طوری که $a'_1 < a'_2 < \ldots < a'_n$.اعدادی که ما می‌خواهیم مرتب کنیم به عنوان کلید *(key)* شناخته می‌شوند.

ما با مرتب‌سازی به روش درج شروع می‌کنیم که یک الگوریتم مفید برای مرتب کردن عناصر با تعداد کم است.

شبه کد ما مرتب‌سازی به روش درج، *Insertion-sort* نامیده می‌شود که یک آرایه *A[1…n]* به عنوان پارامتر می‌گیرد. *A* یک رشته به طول *n* است که باید مرتب شود. در کد، تعداد *n* عنصر در *A* با *Length[A]* مشخص می شود. عددهای ورودی در محل مرتب می‌شوند. *(Sorted in place)* در واقع عددها درون آرایه *A* مجدداً چیده می‌شوند، با حداکثر یک عدد از آنها که بیرون آرایه ذخیره می‌شود.(حداکثر یک حافظه اضافی) وقتی که *insertion-sort*تمام می‌شود آرایه ورودی *A* دارای رشته خروجی مرتب شده است.

INSERTION-SORT(A)		*cost*	*times*
1	**for** $j \leftarrow 2$ **to** *length*$[A]$	c_1	n
2	**do** *key* $\leftarrow A[j]$	c_2	$n-1$
3	$\triangleright$ Insert $A[j]$ into the sorted		
	$\triangleright$ sequence $A[1 \,..\, j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i+1] \leftarrow key$	c_8	$n-1$

شکل ۱-۱

زمان اجرای الگوریتم :

تابع هزینه :

$T(n)$ = $\sum$ تعدادتکرارهرخط * زمان هراجرا خط

$$T(n) = c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 \sum_{j=2}^{n} tj + c_6 \sum_{j=2}^{n} (tj-1) +$$

$$c_7 \sum_{j=2}^{n} (tj-1) + c_8 (n-1)$$

شکل ۲-۱

$t_k =$ تعداد دفعاتی که حلقه ی *while* خط پنجم برای مقدار k اجرا می شود

تابع هزینه می تواند در ۳ حالت بررسی شود:

۱-بهترین حالت (احتمال کم) (آرایه از قبل مرتب باشد)

۲-بدترین حالت (احتمال کم)

۳-حالت متوسط

در بهترین حالت هزینه اجرای تابع *Insertion-sort* خطی است زیرا در بهترین حالت $t_k = 1$ می باشد .

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

عبارت فوق را می توان به فرم $an+b$ نوشت . که در آن a , b ثابت هستند و به هزینه های ثابت C_i بستگی دارند . بنابراین $T(n)$ یک تابع خطی از n می باشد .

در بدترین حالت یعنی آرایه به شکل برعکس مرتب شده باشد t_k=k است .پس تابع هزینه درجه ۲ می شود.

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

و

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

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8).$$

همان طور که پیداست زمانی که برای بدترین حالت مصرف می شود به فرم $An^2 + Bn + C$ می باشد که در آن ضرایب A,B,C ثابت هستند و به هزینه های ثابت C_i بستگی دارند .

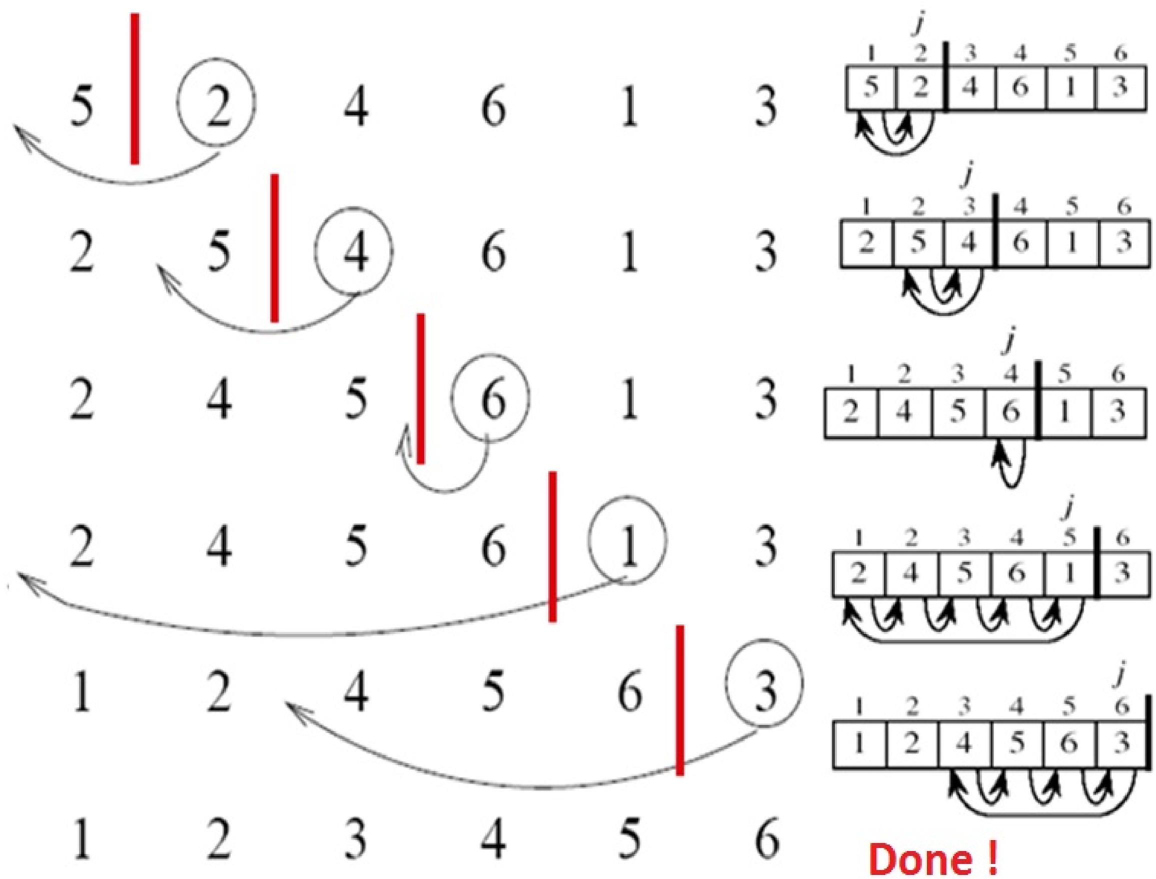
احتمال در اینجا یکنواخت است یعنی عدد $Random$ تولید شده ممکن است در هر جا قرار بگیرد و در حالت متوسط دقیقا ممکن است در وسط قرار بگیرد یعنی فقط نصف اعداد قبلی را پیمایش می کند.

$$p_i = \frac{k+1}{2} t_k = \frac{k+2}{2}$$

$$E(x) = \sum_{x \in D} X.P(x)$$

$$E(t_k) = E_k = \sum_{t_k=1}^{k+1} t_k * (p_k)$$

در حالت متوسط هم هزینه اجرای این تابع درجه ۲ می شود.



شکل ۳-۱

$Insertion - Sort$

$1 - Best\ Case: \ An + B$

$2 - Worst\ Case: \ A'n^2 + B'n + C$

$3 - Average\ Case: \ A''n^2 + B''n + C \quad A'' < A'$

:*Merge-sort*

:*Merge*

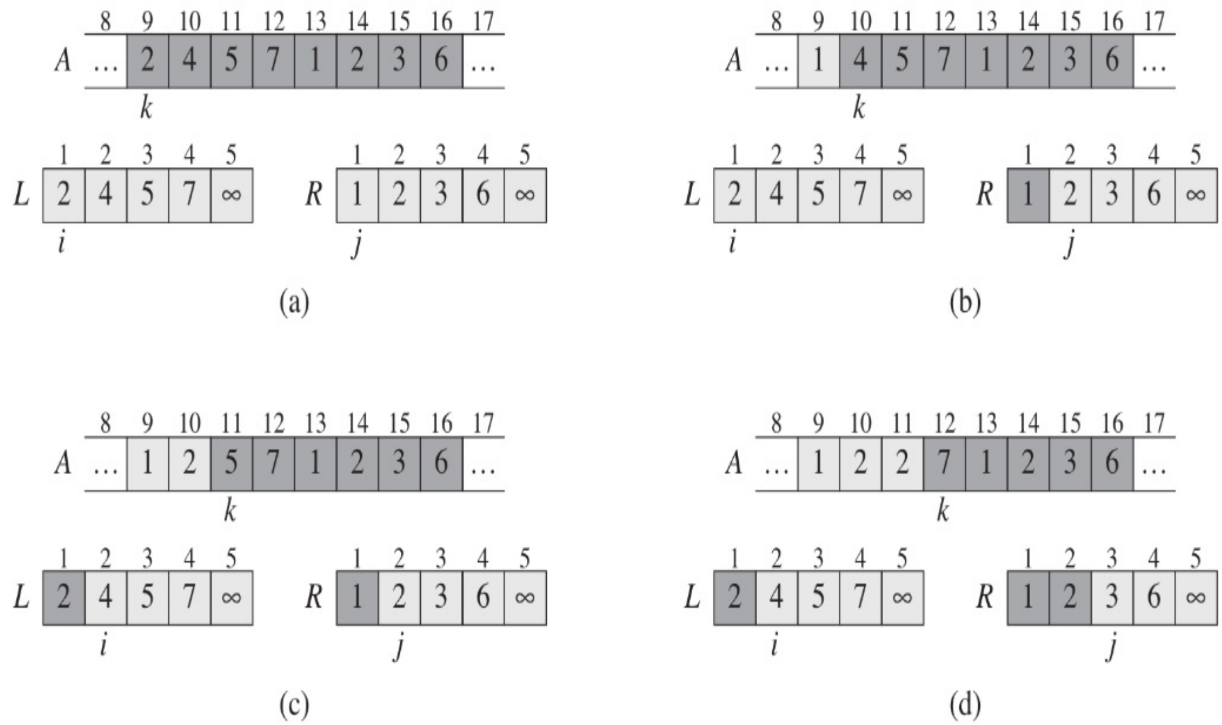
ورودی:یک رشته از n عدد $a_1, a_2, ..., a_n$.ابتدا و انتهای آرایه(p , r) و وسط آرایه - q - به شکلی که اعداد قبل و بعد q در رشته اعداد اولیه مرتب شده هستند .

خروجی: یک نظمی از $\{a'_1, a'_2, ..., a'_n\}$ از رشته ورودی به طوری که $a'_1 < a'_2 < ... < a'_n$.

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

شکل ۴-۱

برای دریافتن آنچه در این الگوریتم رخ می دهد به شکل زیر توجه کنید.همان طور که پیداست این تابع یک آرایه که از دو بخش مرتب شده تشکیل شده است را دریافت می کند و دو قسمت را با هم ادغام کرده و یک آرایه ی مرتب شده را بازمی گرداند.نحوه ی عملکرد به این شکل است که از ابتدای دو قسمت مرتب شده شروع میکند دو سر را با هم مقایسه میکند هر کدام که کوچک تر بود آن را در آرایه ی اصلی قرار داده و در آن قسمت یک خانه به جلو می رود.همین روند را تکرار می کند تا کل آرایه مرتب شود.



شکل ۵-۱

از تابع *merge* استفاده نموده و تابع *merge-sort* را ارایه می دهیم :

ورودی:یک رشته از n عدد $a_1, a_2, ..., a_n$ و ابتدا و انتهای آرایه $(p\ ,\ r)$

خروجی: یک نظمی از $\{a'_1, a'_2, ..., a'_n\}$ از رشته ورودی به طوری که $a'_1 < a'_2 < ... < a'_n$.

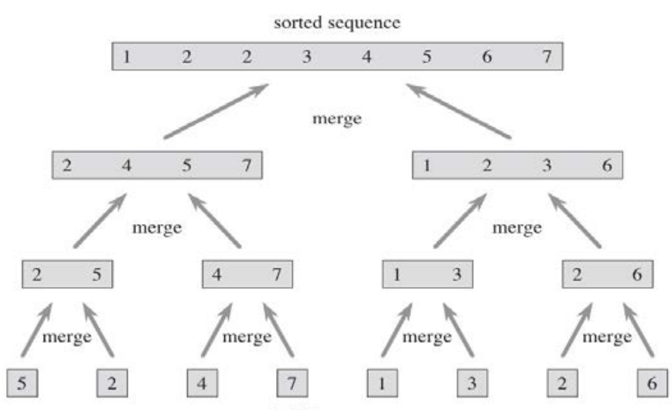
```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

شکل ۱-۶

آن چه در این الگوریتم رخ می دهد را در شکل زیر مشاهده میکنیم(همان طور که پیداست نحوه ی عملکرد این تابع به شکل بازگشتی میباشد.)

رویکرد تقسیم و حل : در یک الگوریتم تقسیم و حل مساله به زیرمساله های کوچک تر تقسیم می شود.هر زیر مساله را به صورت بازگشتی حل می کنیم و سپس حل زیرمساله ها را برای حل مساله ی اصلی ترکیب می کنیم.

الگوریتم *merge-sort* نیز رویکرد تقسیم و حل را به کار گرفته است. در هر مرحله رشته ی اعداد را به دو زیررشته تقسیم کرده و سعی نموده تا مساله را برای دو زیررشته ی جدید حل کند.



شکل ۱-۷

هزینه ی تابع *merge* برابر با $O(n)$ می باشد . اگر تابع هزینه ی *merge-sort* را $T(n)$ در نظر بگیریم داریم:

$$T(n) = 2T(n/2) + O(n) + O(1)$$

رابطه ی فوق برخاسته از رویکرد حل و تقسیم می باشد. به این معنا که هزینه ی مرتب کردن رشته ای از اعداد با طول n با این الگوریتم برابر با هزینه ی مرتب کردن دو رشته از اعداد با طول $n/2$ به علاوه ی هزینه ی ادغام کردن این دو رشته است.

که جلوتر نشان خواهیم داد که حاصل فوق برابر با $O(nlogn)$است

۱ ۴ - تحلیل الگوریتم

در این قسمت الگوریتم های متفاوت را بررسی می کنیم. برای هر الگوریتم *1000s*زمان اجرا در نظر می گیریم.در این زمان تعداد عناصری که می شود *Sort* کردرا محاسبه می کنیم.

در بخش دیگر سرعت سخت افزار را ۱۰ برابر می کنیم و نتایج متفاوتی را برای n بدست خواهیم آورد.

O	*T(n)*	*"n" that can be solved in 1000s*	*For 10 times faster machine*	*Ratio*	*Ratio if was use a machine 1000 times faster 1000.00*
$O(n)$	$100n$	10	100	10	1000.00
$O(n^2)$	$5n^2$	14	45	3.2	31.94
$O(n^3)$	$\frac{n^3}{2}$	12	27	2.3	10.50
$O(2^n)$	2^n	10	13	1.3	2.00

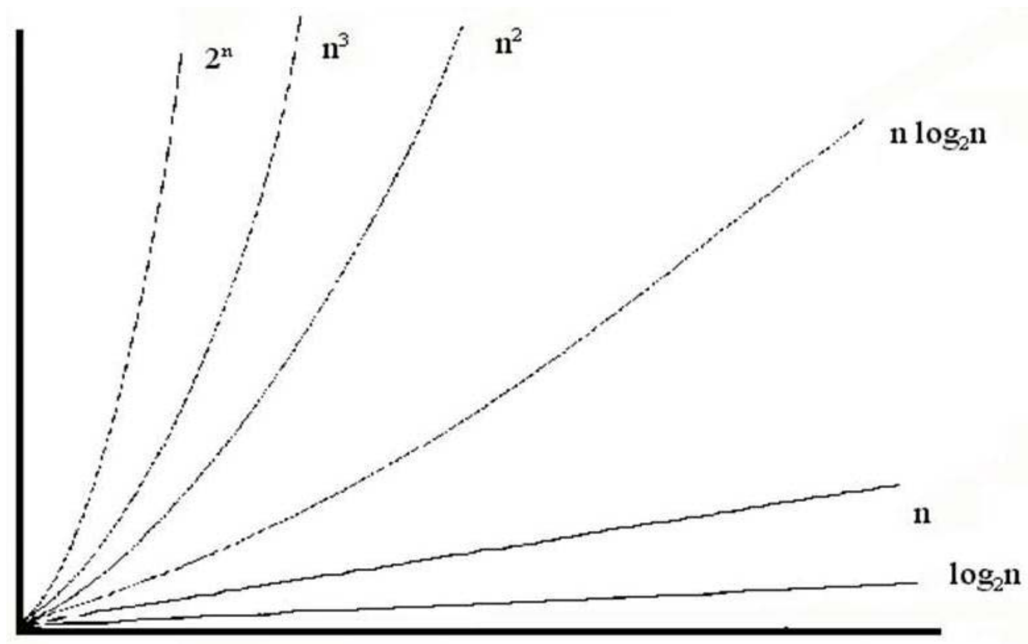
جدول ۱-۱

در بعضی از مواقع محاسبه $T(n)$ عملی نیست.

در حالات کلی نیازی نیست که معادله $T(n)$ تشکیل دهیم و ضرایب آن را بدانیم. این بدین معنی است که فقط دانستن درجه آن کافی است.هدف تنها تخمین و مقایسه الگوریتم ها در زمان اجرایشان است.

(همان طور که از جدول پیداست در الگوریتم هایی با$O(n^2)$ و بالاتر با حتی با چند ده برابر کردن سرعت سخت افزار نسبت n در مقایسه با تغییر سرعت سخت افزار تغییر زیادی نمی کند.)

۱ ۴ - توابع هزینه متداول



شکل ۸-۱

هر چه نمودار تابع هزینه یک الگوریتم به محور افقی نزدیک تر باشد یعنی سرعت رشد پایین تری داشته باشد الگوریتم مربوطه بهینه تر است.

Growth Rate Name	Function
Constant	c
Logarithmic	$log\ n$
Log-Squared	$log_2\ n$
Linear	N
	$n log\ n$
Quadratic	n^2
Cubic	n^3
Exponential	2^n

جدول ٢-١