فصل ۱

مرتبه ی اجرایی

۱.۱ سر فصل ها

۱. مرتبه ی اجرایی (پیچیدگی اجرایی)

۲. رابطه های بازگشتی

۳. روش تقسیم و حل

۴. روش پویا

۵. روش حریصانه

۶. روش عقبگرد

۷. الگوریتم های گراف

۸. الگوریتم های مرتب سازی

۹. مسائل p و np

۲.۱ پیچیدگی اجرایی

پیچیدگی یک الگوریتم ، تابعی است که مدت زمان اجرای استفاده شده توسط الگوریتم را بر حسب تعداد داده های ورودی n اندازه می گیرد .

notation	name
$\mathcal{O}(1)$	constant
$\mathcal{O}(n)$	linear
$\mathcal{O}(\log(n))$	logarithmic
$\mathcal{O}(n^2)$	quadratic
$\mathcal{O}(n^x)$	polynomial
$\mathcal{O}(e^n)$	exponential
$\mathcal{O}(n!)$	factorial

۳.۱ مرتبه ی اجرایی توابع چند جمله ای

$$f(n) = n^m + n^{m-1} + \cdots + n^1 + n + c \Rightarrow f(n) = O^{(n^m)}$$

مثال

$$f(n) = 5^{n^2} + 3n + 4 \Rightarrow f(n) = O^{(n^2)}$$

$$f(n) = n + 6n^8 + n^2 \Rightarrow f(n) = O^{(n^8)}$$

$$n! + 2^n + 1000n^{10} \Rightarrow f(n) = O(n!)$$

۴.۱ مقایسه

$$O(1) < O(\log{(n)}) < O(n) < O(n\log{(n)}) < O(n^2) < O(2^n) < O(n!) < O(n^n)$$

$$\log_2^n = \log{(n)}$$

۵.۱ مرتبه اجرایی حلقه های ساده

```
for( int i = a ; i <= b ; i+=k ) {

}
```

$$\frac{b-a+1}{k}$$

```
for( int i = b ; i >= a ; i-=k ) {

}
```

$$\frac{b-a+1}{k}$$

مثال

```
for( int i = 1 ; i <= n ; i+=1 ) {

}
```

$$\frac{n-1+1}{1}=n$$

```
for( int i = 3 ; i <= n ; i+=2 ) {

}
```

$$\frac{n-3+1}{2}=\frac{n-2}{2}$$

```
for( int i = 9 ; i < 3n+4 ; i+=5 ) {

}
```

$$\frac{3n+4-9}{5}=\frac{3n-5}{5}$$

۶.۱ مرتبه ی لگاریتمی

```
for( int i = a ; i <= b ; i=i*k ) {

}
```

$$\log_k^b - \log_k^a + 1$$

```
for( int i = b ; i >= a ; i=i/k ) {

}
```

$$\log_k^b - \log_k^a + 1$$

مثال

```
for( int i = 1 ; i <= 8 ; i=i*2 ) {

}
```

$$\log_2^8 - \log_2^1 + 1 = 4$$

```
for( int i = 27 ; i <= n ; i=i*3 ) {

}
```

$$\log_3^n - \log_3^{27} + 1 = \log_3^n - 2$$

۷.۱ حلقه های تو در تو

```
for( int i = 1 ; i <= n ; i++ )
    for( int j = 1 ; j <= n ; j++ )
```

$$n^2$$

```
for( int i = 2 ; i <= n ; i+=4 )
    for( int j = n ; j > 3 ; j=j-2 )
```

$$\frac{n-2+1}{4} \times \frac{n-3}{2} \Rightarrow n^2$$

```
for( int i = 1 ; i <= n ; i*=2 )
    for( int j = 1 ; j <= n ; j++ )
```

$$\log{(n+1)} \times n \Rightarrow O(n \log n)$$

۸.۱ حلقه های پشت سرهم

```
for( int i = 1 ; i <= n ; i++ ) {

}

for( int j = 1 ; j <= m ; j++ ) {

}
```

$O(max(n,m))$
Or
$O(n+m)$

۹.۱ ترکیب حلقه های تو در تو و پشت سر هم

```
for( int i = 1 ; i <= n ; i++ ) {
    for( int j = 1 ; j <= n ; j++ ) {

    }
}

for( int k = 1 ; k <= n ; k++ ) {

}
```

$O(max(n^2,n)) = O(n^2)$

۱۰.۱ حلقه های تو در توی وابسته

```
for( int i = 1 ; i <= n ; i++ ) {
    for( int j = 1 ; j <= i ; j++ ) {

    }
```

```
}
```

$$1+2+3+\cdots+n=\frac{n(n+1)}{2}\Rightarrow O(n^2)$$

۱۱.۱ نماد های پیچیدگی اجرایی

۱.۱۱.۱ big-O Notation

عبارت $g(n)\in O(f(n))$ یعنی برای تابع پیچیدگی مفروض $f(n)$ ، $O(f(n))$ به مجموعه ای از توابع اشاره دارد که برای آنها ثابت های c و n_0 وجود دارند ، بطوریکه برای همه ی $n\geq n_0$ داریم :

$$g(n)\leq cf(n)$$

۲.۱۱.۱ big-Ω Notation

عبارت $g(n)\in\Omega(f(n))$ یعنی برای تابع پیچیدگی مفروض $f(n)$ ، $\Omega(f(n))$ به مجموعه ای از توابع اشاره دارد که برای آنها ثابت های c و n_0 وجود دارند ، بطوریکه برای همه ی $n\geq n_0$ داریم :

$$g(n)\geq cf(n)$$

۳.۱۱.۱ θ Notation

عبارت $g(n)\in\theta(f(n))$ یعنی :

$$g(n)\in O(f(n))$$

و

$$g(n)\in\Omega(f(n))$$

۱۲.۱ خواص توابع رشد

۱.۱۲.۱ بازتابی

$$\left.\begin{array}{r} f(n) = O(f(n)) \\ f(n) = \Omega(f(n)) \end{array}\right\} \Rightarrow f(n) = \theta(f(n))$$

۲.۱۲.۱ تراگذاری

$$\left.\begin{array}{r} f(n) = \theta(g(n)) \\ g(n) = \theta(h(n)) \end{array}\right\} \Rightarrow f(n) = \theta(h(n))$$

۳.۱۲.۱ تقارن برای θ

$$f(n) = \theta(g(n)) \Leftrightarrow g(n) = \theta(f(n))$$

۴.۱۲.۱ تقارن ترانهاده

$$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$

۱۳.۱ نکته

$$\left.\begin{array}{r} f(n) \in O(g(n)) \\ h(n) \in O(k(n)) \end{array}\right\} \Rightarrow f(n) + h(n) \in O(max(g(n), k(n)))$$

$$\left.\begin{array}{r} f(n) \in O(g(n)) \\ h(n) \in O(k(n)) \end{array}\right\} \Rightarrow f(n).h(n) \in O(g(n).k(n))$$

فصل ۲

روش تقسیم و حل

۱.۲ مراحل

۱. تقسیم نمونه ای از یک مسئله به یک یا چند نمونه کوچکتر

۲. حل نمونه های کوچکتر

۳. ترکیب حل نمونه های کوچکتر برای به دست آوردن حل نمونه اولیه (در صورت نیاز)

** دلیل اینکه میگوییم **در صورت نیاز** این است که در بعضی الگوریتم ها مانند جستجوی دودویی نمونه فقط به یک نمونه کوچکتر کاهش می یابد و نیازی به ترکیب حل ها نیست .
** هنگام طراحی الگوریتم های تقسیم و حل معمولاً آن را به صورت یک روال بازگشتی می نویسند .

مثال

الگوریتمی که هر ورودی مسئله به اندازه ی n را به ۲ بخش کم و بیش مساوی تقسیم می کند و زیر مسئله ها را به صورت بازگشتی حل و سپس با هزینه خطی حاصل این دو را با هم ترکیب کرده و جواب مسئله را به دست می آورد برابر است با :

$$T(n) = 2\,T(\frac{n}{2}) + O(n)$$
$$T(n) = O(n \log{(n)})$$

۲.۲ ضرب دو عدد صحیح بزرگ

برای انجام اعمال محاسباتی روی اعداد صحیح بزرگتر از حد قابل نمایش توسط سخت افزار کامپیوتر ، باید از روش تقسیم و حل استفاده کرد .

اگر n تعداد ارقام عدد صحیح u باشد ، آن را به دو عدد صحیح یکی x با $\lceil \frac{n}{2} \rceil$ رقم و دیگری y با $\lfloor \frac{n}{2} \rfloor$ تبدیل می کنیم :

$$u = x \times 10^m + y \qquad\qquad m = \lfloor \frac{n}{2} \rfloor$$

مثال :

$$12345 = 123 \times 10^2 + 45$$

۱.۲.۲ ضرب u و v

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

پس می توان u و v را با ۴ عمل ضرب روی اعداد صحیح (با حدود نیمی از ارقام) و اجرای عملیات زمان خطی در هم ضرب کنیم .

فصل ۳

روش برنامه نویسی پویا

در روش پویا 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 جدول استفاده می شود .

برنامه نویسی پویا از این لحاظ که نمونه را به نمونه های کوچکتر تقسیم می کند مشابه روش تقسیم و حل است .

۱.۳ مراحل بسط یک الگوریتم برنامه نویسی پویا

۱. ارائه ی یک ویژگی بازگشتی برای حل نمونه ای مسئله

۲. حل نمونه ای از مسئله به شیوه پایین به بالا با حل نمونه های کوچکتر

۲.۳ الگوریتم هایی که به روش برنامه نویسی پویا حل می شوند

۱. دنباله فیبوناچی

۲. ضرب دو جمله ای

۳. ضرب زنجیره ای ماتریس ها

۴. درخت های جستجوی دودویی بهینه

۵. کوله پشتی ۰ و ۱

۶. فلوید

۷. فروشنده دوره گرد

فصل ۴

روش حریصانه

الگوریتم حریصانه با انجام یک سری انتخاب ، که در جای خود بهینه است ، عمل کرده به امید اینکه یک حل بهینه کلی یافت شود .
در الگوریتم حریصانه همواره جواب بهینه حاصل نمی شود .
بهینه بودن الگوریتم باید تعیین شود .
در روش حریصانه ، تقسیم به نمونه های کوچکتر صورت نمی پذیرد .

۱.۰.۴ نحوه ی کار الگوریتم حریصانه

الگوریتم حریصانه ، کار را با یک مجموعه ی تهی آغاز کرده و به ترتیب عناصری به مجموعه اضافه می کند تا این مجموعه حلی برای نمونه ای از یک مسئله را نشان دهد .
هر 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 انجام شده که یک شرط بهینه را در همان برهه برآورده می سازد .

بررسی امکان سنجی

تعیین می کند که آیا مجموعه ی جدید برای رسیدن به حل ، عملی است یا خیر .

بررسی راه حل

تعیین می کند که آیا مجموعه ی جدید ، حل نمونه را ارائه می کند یا خیر .

۱.۴ نمونه هایی از الگوریتم های حریصانه

۱. خرد کردن پول

۲. زمانبندی

۳. کد هافمن

۴. کوله پشتی کسری

۵. دایکسترا

۶. پریم

۷. کراسکال

فصل ۵

الگوریتم های گراف

۱.۵ نمایش گراف

۱. ماتریس همجواری (Adjacency Matrices)

۲. لیست همجواری (Adjacency List)

۲.۵ پیمایش گراف

۱. سطحی (BFS : Breadth First Search)

۲. عمقی (DFS : Depth First Search)

در پیمایش DFS از پشته و در پیمایش BFS از صف استفاده می شود .

۳.۵ کوتاهترین مسیر (Shortest Paths)

۱.۳.۵ Single-Source All Destination

- Dijkstra
- Bellman-Ford

۲.۳.۵ All-Pairs

- Matrix Multiplication
- Floyd-Warshall

۴.۵ درخت پوشای کمینه (Minimum Spanning Tree)

۱.۴.۵ درخت پوشا

یک زیر گراف متصل است که حاوی همه ی رئوس موجود در گراف بوده و یک درخت باشد یعنی چرخه نداشته باشد .

۲.۴.۵ الگوریتم های تعیین درخت پوشای کمینه

1. Prim
2. Kruskal