به نام خدا

طراحی و تحلیل الگوریتم ها

هدف:

هدف اصلی این درس آشنایی با چگونگی طراحی الگوریتم های کارآ و تحلیل الگوریتم ها از لحاظ کارآیی و پیچیدگی است. در واقع لازم است که در این درس دانشجو با الگوریتم های اساسی و پایه ای در علوم کامپیوتر به همراه کسب دانش کافی به جهت تحلیل آنها از لحاظ کارآیی آشنا شده و بتواند در صورت مواجهه با مسائل جدید در جهت طراحی الگوریتم های مناسب برای آنها اقدام نموده و تحلیل مناسبی جهت کارآیی الگوریتم های طراحی شده ارائه دهد.

سخنی با مدرس دانشجو:

پس از "مبانی نظریه محاسبه" دانشجو با این مهم آشنا شده که اگر مسأله ای دارای راه حل الگوریتمی باشد، هنوز یافتن یک الگوریتم کارآیا برای آن یک امر مهم و در بسیاری از موارد سخت به حساب می آید. لذا در این درس ضمن آشنا شدن با اصول اولیه تحلیل الگوریتم ها نظیر آشنایی با مفاهیم بسیارمقدماتی نظریه پیچیدگی، با الگوریتم های برخی مسائل بنیادی آشنا شده و سـعی می شود با ارائه کران های پایین و بالای زمانی برای آن ها به تحلیل آنها بپردازیم. همچنین در این درس با انواع مختلف الگوریتم ها نیز آشنا شده و مقدمات نظریه الگوریتم های تقریبی را نیز فرا خواهیم گرفت.

سر فصل:

مرور مفاهیم اولیه نظریه پیچیدگی و تحلیل مجانبی (نمادهای O, o, Ω, Θ)، مرور ساختمان های داده ای پایه و معادلات ارجائی (که قبلا در درس های "ساختمان داده ها و الگوریتمها" و "مبانی ترکیبیات" مطالعه شده اند)، الگوریتـم های استقرایی، Divide & Conquer، برنامـه ریزی پویا (شامل مثال های اصلی و متنوع نظیر انواع Sort، ضرب اعداد بزرگ و ماتریس ها و نظایرآن)، الگوریتم های حریصانه، الگوریتم های پیمایشی گراف ها (بالاخص درخت ها)، مفهوم مسأله NP–تمام، کران های پایین و بالا برای پیچیدگی زمانی و حافظه (در حد الگوریتم های ارائه شده در درس و با نظر استاد و تأکید بر محاسبه آن ها)، الگوریتم های تصادفی، الگوریتم‌های تقریبی.

ریز مواد:

- دوره مفاهیم اولیه تحلیل مجانبی و ساختمان داده ها (با توجه به دروس پیش نیاز)
- ارائه ایده های اصلی روش های بنیادی طراحی،بازگشت و استقراء، Divide & Conquer، برنامه ریزی پویا ، الگوریتم های حریصانه ، الگوریتم های تصادفی و مفهوم تقریب.
- تحلیل انواع Sort: HeapSort، Quick Sort ، Sort در زمان خطی و در ضمن آن تحلیل چگونگی ساخت، ساختمان های داده مربوطه.
- الگوریتم های عددی: یافتن Min, Max, Median و نظایر آن، ضرب اعداد و ماتریس ها (الگوریتم های مختلف با تخمین زمان آن ها).
- الگوریتم های حریصانه: مسأله کوله پشتی، کوتاه ترین مسیر، درخت گسترنده مینیمم، فشرده سازی فایل ها.
- تأکید مجدد بر ساختمان های داده و نقش آن ها: درخت های جستجوی باینری، جداول Hash، پشته، صف و نظایر آن ها و بحث های پیشرفته تر(با نظراستاد: نظیر B-tress، انواع Heap و ...)
- بحث دقیق روی پیچیدگی BFS و DFS، یافتن مؤلف های همبندی گراف ها، الگوریتم های شار ماکسیمم - برش مینیمم و تحلیل آن ها.
- برنامه ریزی پویا و برخی الگوریتمها نظیر طولانی ترین زیر دنباله مشترک، ۰و۱ ،کوتاه ترین مسیر، All-pair- کوله پشتی.
- الگوریتمهای تصادفی (با نظر استاد) نظیر Quicksort تصادفی، نمونه برداری تصادفی و....
- الگوریتمهای تقریبی(با نظر استاد) نظیر الگوریتمهای تقریبی برای مسأله کوله پشتی، پوشش رأسی گراف و نظایر آن با تحلیل ضریب تقریب.

منابع:

1: طراحی و تحلیل الگوریتم ها، بهروز قلی زاده، انتشارات دانشگاه شریف

2: طراحی الگوریتم، حمیدرضا مقسمی

3: Introduction to Algorithms, T. Cormen et al.

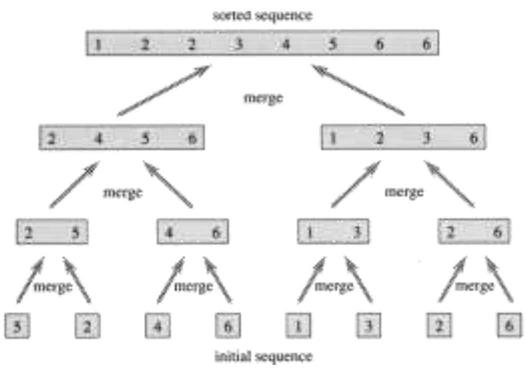
نحوه ارزیابی درس:

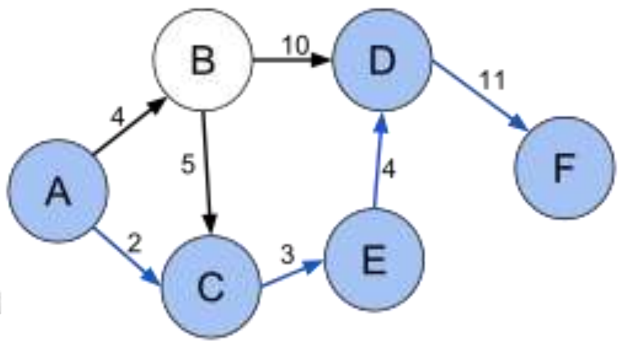
- میان ترم تا 8 نمره
- پروژه تا 2 نمره
- پایان ترم 10

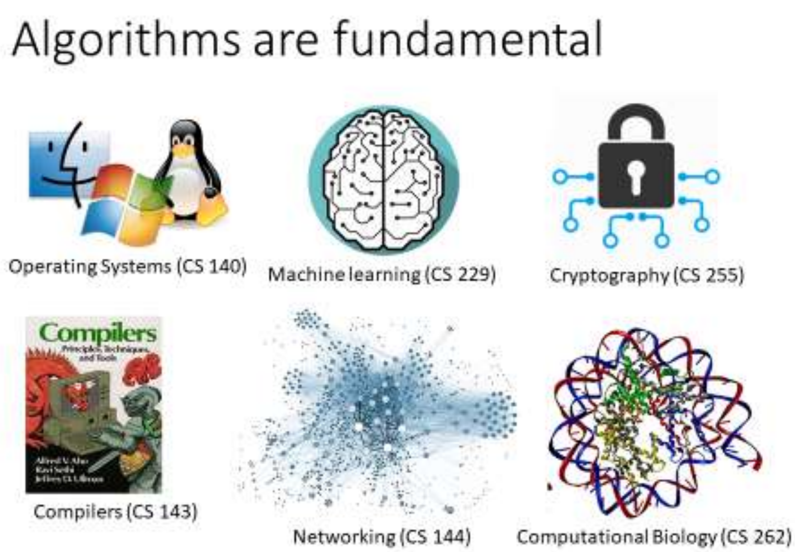
تعریف: هر رویه محاسباتی خوش تعریف که یک یا چند مقدار را به عنوان ورودی می گیرد و یک یا چند مقدار را به عنوان خروجی بر می گرداند یک **الگوریتم** نامیده می شود.

هر دستورالعملی که مراحل انجام کاری را بطور دقیق و با جزییات کافی بیان کند به گونه ای ترتیب مراحل و شرط خاتمه عملیات کاملا مشخص باشد الگوریتم نامیده می شود.

مثال:







Operating system scheduling algorithms include **First-Come-First-Serve, Shortest Job Next, Priority Scheduling, Round Robin, and Multilevel Queue Scheduling**.

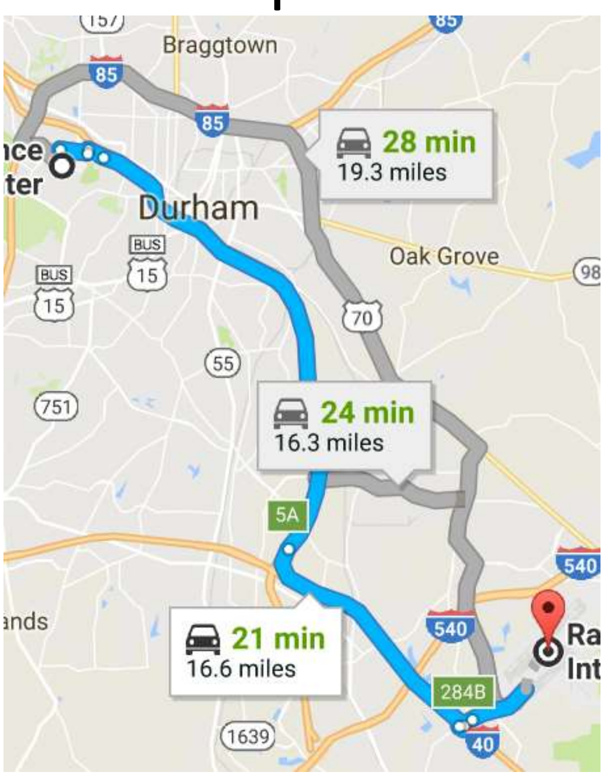
A good compiler makes practical use of **greedy algorithms (register allocation), heuristic search techniques (list scheduling), graph algorithms (dead-code elimination), dynamic programming (instruction selection), automata theory (scanning and parsing), and fixed-point algorithms (data-flow analysis).**

مطالعه الگوریتم ها زمینه های زیر را شامل می شود:

الف: طراحی الگوریتم ها: برای این منظور روشهای محتلفی همچون استقرایی، پویا، تقسیم و غلبه، حریصانه و .... وجود دارد.

ب: معتبر سازی الگوریتم ها: یک الگوریتم وقتی درست است که به ازای هر ورودی مناسب خروجی درست دهد. هدف از این کار مستقل بودن الگوریتم از زبان برنامه نویسی است.

Correctness:

- 12343*9432583 = 116426471969 ?
- (No: 116426371969)
- More complicated:
  Why should I believe this is the shortest path to the airport?
- 

ج: تحلیل الگوریتم ها: هدف تعیین میزان استفاده از حافظه برای ذخیره سازی و مدت زمان استفاده از CPU است.

د: پیاده سازی و تست.

در این درس طراحی و تحلیل الگوریتم ها مطالعه می شوند:

تحلیل الگوریتم ها

در تحلیل الگوریتم ها دو فاکتور مهم مورد توجه است، یکی حافظه مصرفی و دیگری زمان اجرا. الگوریتمی بهتر است که در هر دو مورد بهتر باشد. معمولا کارایی الگوریتم ها را بر اساس زمان اجرا بررسی می کنند، یعنی زمان اجرای عملیات اصلی که در الگوریتم انجام می گیرد از قبیل چهار عمل اصلی.

در تحلیل زمانی هدف شمارش تک تک دستورات اجرا شده نیست چون به زبان برنامه نویسی بستگی دارد، بلکه عملیاتی که مستقل از زبان برنامه نویسی باشد شمارش می شوند. در حالت کلی هرچه اندازه ورودی افزایش یابد زمان اجرا افزایش می یابد.

مثال 1: تابع زیر جمع عناصر یک آرایه را در زبان C محاسبه می کند:

```
float  sum (float list[ ], int n)
{
  float  s=0 ;      int i;
    for (i = 0;  i<n;  i++)
              s = s + list[i];
       return   s;
  }
```

در این برنامه اندازه ورودی n و عمل اصلی s=s+list[i]; که n بار انجام می گیرد.

پس از تعیین اندازه ورودی، یکی یا تعدادی از عملیات را به عنوان عمل یا عملیات اصلی در نظر می گیریم و کار انجام شده توسط الگوریتم متناسب با تعداد دفعاتی است که این عملیات انجام می شوند.

بطور کلی پیچیدگی زمانی یک الگوریتم عبارت است از تعیین تعداد دفعاتی که عمل (عملیات) اصلی به ازای هر اندازه ورودی انجام می شود. لازم به ذکر است قاعد صریحی برای انتخاب عمل اصلی وجود ندارد و معمولا با تجربه مشخص می شود.

مثال 2: تعداد کل مراحل برنامه مثال 1 را محاسبه کنید:

`float sum (float list[ ], int n)`	0
`{`	0
`float s = 0;`	1
`int i;`	0
`for (i = 0; i<n; i++)`	n+1
`s = s + list[i];`	n
`return s;`	1
`}`	0
	2n+3

در مثال فوق زمان اجرای هر عبارت ساده را مساوی ۱ واحد زمانی فرض می‌کنیم. عبارت ساده شامل زیر برنامه نمی‌باشد. یک عبارت ساده می‌تواند به اندازه یک دستور ;x = 2 کوچک باشد و یا مشابه دستور زیر طولانی، در هر حال هر دو را 1 واحد زمانی می‌گیریم :

x = 5 * y + 6 * a - 5 / w ;

توجه کنید } و { و نیز خط اول تعریف تابع و تعریف متغیر دستوراتی نیستند که توسط CPU اجراء شوند پس مرحله اجرائی آنها صفر است. در خط ;float s = 0 چون عدد صفر در s ریخته می‌شود پس یک مرحله می‌باشد. همچنین توجه کنید دستور داخل حلقه n بار انجام می‌شود ولی آزمایش کردن شرط حلقه در خط for به تعداد n + 1 بار صورت می‌گیرد. دستور return نیز توسط CPU باید اجراء شود. همانطور که قبلاً گفتیم اگر عمل اصلی را فقط خط ;s =s+list[i] فرض کنیم آنگاه T(n)=n خواهد بود. پس توجه داشته باشید که گاهی اوقات فقط می‌خواهیم بدانیم یک دستور ویژه چند بار تکرار می‌شود و گاهی اوقات نیز تعداد کل مراحل یا گام‌های برنامه را می‌خواهیم. البته عموماً تنها تعداد اجرای عمل اصلی مدنظر قرار می‌گیرد.

مثال 3: دستور $x := x + 1;$ در برنامه زیر چندبار اجرا می شود؟ تعداد کل گامهای برنامه چقدر است؟

```
for i: = 1 to m do
   for j: = 1 to n do
         x: = x+1;
```

راه‌حل اول : حلقه‌های داده شده مستقل از یکدیگرند بنابراین طبق آنچه در زبانهای برنامه‌نویسی پاسکال و C خوانده‌اید تعداد اجرای دستور درون حلقه‌ها برابر mn می‌باشد.

راه حل دوم : $\text{تعداد اجرای دستور اصلی} = \sum_{i=1}^{m}\sum_{j=1}^{n}1 = \sum_{i=1}^{m}n = n\left(\sum_{i=1}^{m}1\right) = nm$

حال برای محاسبه تعداد کل گام‌های برنامه ابتدا حلقه بیرونی i را کنار گذاشته و در نظر نمی‌گیریم. در این حال دستور ;x: = x+1 به تعداد n بار و عبارت for j به تعداد (n+1) بار اجرا می‌گردد. یعنی تعداد کل (n+1+n). حال خود این ۲ خط درون حلقه i بوده و به تعداد m بار اجرا می‌شوند یعنی m(n+1)+mn از آنجا که عبارت for i نیز m+1 بار اجرا می‌شود، پس :

$\text{تعداد کل مراحل برنامه} = (m + 1) + m(n + 1) + mn$

مثال 4: دستور $x := x + 1;$ در برنامه زیر چندبار اجرا می شود؟

```
for j: = 1 to n do
   for i: = 1 to j do
         x: = x+1;
```

راه حل اول : حلقه‌های داده شده به یکدیگر وابسته‌اند :

j	تغییرات i	تعداد اجرا شدن دستور اصلی
1	1	1 بار
2	1,2	2 بار
3	1,2,3	3 بار
	......	
n	1,2,3,...,n	n بار

$x := x + 1;$ تعداد اجرا شدن دستور $= 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$

راه حل دوم : $\sum_{j=1}^{n}\sum_{i=1}^{j}1 = \sum_{j=1}^{n}j = \frac{n(n+1)}{2}$

مثال 5: دستور ;x:=x+1 در برنامه زیر چندبار اجرا می شود؟

```
i:=n;
while (i>1) do begin
      x:=x+1;
       i: = i div 2;
end;
```

اگر n = 16 باشد :

i	شرط i >1	تعداد اجرا شدن دستور اصلی
16	درست	۱ بار
8	درست	۱ بار
4	درست	۱ بار
2	درست	۱ بار
1	غلط	–
		جمعاً ٤ بار

حال اگر n را برابر 14 فرض کنیم :

i	شرط i >1	تعداد اجرا شدن دستور اصلی
14	درست	۱ بار
7	درست	۱ بار
3	درست	۱ بار
1	غلط	–
		جمعاً ۳ بار

پس در حالت کلی دستور اصلی به تعداد $\lfloor \log_2^n \rfloor$ بار اجرا می‌شود.

پیچیدگی زمانی در سه حالت بهترین، میانگین و بدترین بررسی می شود.

به عنوان مثال اگر هدف جمع عناصر یک آرایه است:

```
A : Array [1 .. n]  of Integer;
S := 0;
For I := 1  To  n  do
    S := S + A[i] ;
```

همواره $T(n) = n$. اما اگر الگوریتم مورد نظر جستجوی ترتیبی باشد، زمان اجرا در حالت های مختلف متفاوت است:

```
A : Array [1 .. n] of Integer;
For  i := 1   to  n  do
     if (x = A[i])  {
            write ('Yes');
              exit ( ) ; → خروج از برنامه
     }
write ('No');
```

در برنامه فوق عمل اصلی شرط if (x = A[i]) می‌باشد.

در الگوریتم فوق، در بهترین حالت، عنصری که دنبال آن هستیم عنصر اول آرایه است و در بدترین حالت نیز عنصر آخر آن است. در حالت متوسط نیز ابتدا فرض کنید عنصر x در آرایه است و احتمال اینکه در خانه iام باشد $\frac{1}{n}$ است، پس تعداد جستجو برابر است:

$$\sum_{i=1}^{n} i\frac{1}{n} = \frac{n+1}{2}.$$

بنابراین باید نیمی از آرایه در حالت متوسط باید جستجو کرد. حال فرض کنید احتمال وجود x در آرایه p است، پس احتمال اینکه در خانه iام باشد $\frac{p}{n}$ است. پس در حالت متوسط لازم است

$$\sum_{i=1}^{n} i\frac{p}{n} + (1-p)n = n\left(1-\frac{p}{2}\right) + \frac{p}{2}$$

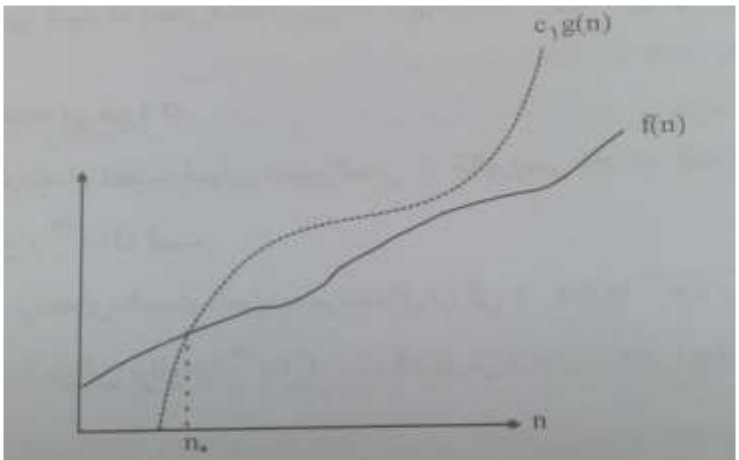
جستجو انجام شود. برای مقادیر مختلف p مقادیر مختلف نتیجه می شود.

در مثال های قبل تعداد دفعاتی که عملیات اصلی در الگوریتم انجام می شود بطور دقیق محاسبه شد. اما در حالت کلی محاسبه آن بطور دقیق لازم نیست، صرفا مرتبه آن مشخص شود کفایت می کند. مثلا برای n های بزرگ در عبارت $5n^2+2n-3$، جمله اول آن یعنی $5n^2$ تعیین کننده است. برای این منظور نماد های مجانبی در ادامه معرفی می شوند:

نماد های مجانبی:

تعریف: O(اوی بزرگBig O) گوییم $\mathrm{f(n)} = \mathrm{O(g(n))}$ اگر عدد طبیعی n_0 وعدد حقیقی مثبت c_1 وجود داشته باشند بطوریکه برای هر $\mathrm{n} \geq n_0$، $|\mathrm{f(n)}| \leq c_1|\mathrm{g(n)}|$

(شکل 1-1 را ببینید). این نماد برای بررسی بدترین حالت رفتاری اجرای الگوریتم استفاده می شود.



شکل ۱-۱ تعبیر هندسی O.

مثال 6: نشان دهید $f(n) = 3n^3 + 2n^2 = O(n^3)$.

مثال 7: نشان دهید $f(n) = 3n^3 + 2n^2 = O(n^4)$.

مثال 8: مرتبه اجرایی برنامه های زیر را بدست آورید:

الف) x := x + 1 ; ⇒ $O(1)$

ب)
```
for i := 1 to n do
        x := x+1;
```
⇒ $O(n)$

ج)
```
for i := 1 to n do
    for j := 1 to n do
        x := x + 1 ;
```
⇒ $O(n^2)$

مثال 9: مرتبه اجرایی برنامه زیر را بدست آورید:

```
x := 0;
i := n ;
while (i > 1) do begin
        x := x + 1;
        i := i div 2;
end;
```

فرض کنید $n = 32$ باشد آنگاه

i	x
32	1
16	2
8	3
4	4
2	5
1	-

پس برای $n = 32$ عمل اصلی ;x := x +1 ۵ بار انجام می‌شود ($5 = \log_2 32$). پس در حالت کلی مرتبه اجرائی الگوریتم فوق برابر $O(\log n)$ می‌باشد. توجه کنید در درس ساختمان داده عموماً منظور از $\log n$ یعنی $\log_2 n$.

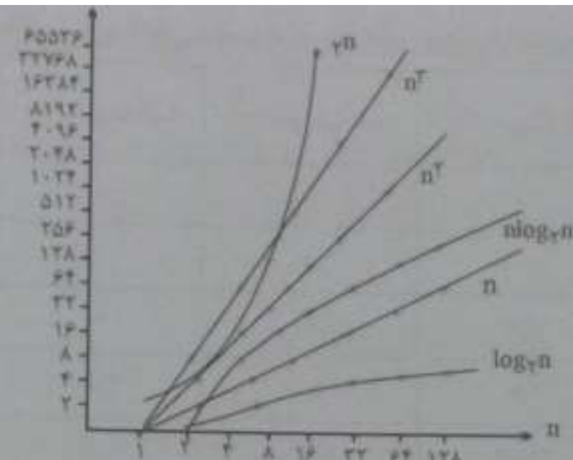
نکته ۲ : در حلقه while که به طور طبیعی شمارنده آن از n تا 1 تغییر می‌کند اگر مرتباً شمارنده آن با دستور ;i:=i div k بر عدد k تقسیم شود مرتبه اجرایی آن $O(\log_k^n)$ خواهد بود. به همین ترتیب اگر شمارنده با دستور ;i:=i*k از 1 تا n تغییر کند باز هم مرتبه اجرایی آن $O(\log_k^n)$ می‌باشد :

```
i: = n;                         i:=1;
while (i >1) {                  while(i < n) {
   ; عمل اصلی          یا          ; عمل اصلی          ⇒     O(log_k^n)
   i: = i div k;                    i:=i*k:
}                                }
```

$\Rightarrow O(\log_k^n)$

نکته۳ : در جدول زیر مرتبه اجرائی چند تابع به ترتیب صعودی از چپ به راست نوشته شده است:

فاکتوریل	توانی	مرتبه ۲	———	خطی	لگاریتمی	ثابت	نام تابع
$O(n!)$	$O(2^n)$	$O(n^2)$	$O(n\log n)$	$O(n)$	$O(\log n)$	$O(1)$	مرتبه اجراء



شکل ۱–۴ نرخ رشد توابع متداول برای زمان اجرا

جدول۱–۱ مقادیر توابع اجرا

$\log n$	n	$n \log n$	n^2	n^3	2^n
۰	۱	۰	۱	۱	۲
۱	۲	۲	۴	۸	۴
۲	۴	۸	۱۶	۶۴	۱۶
۳	۸	۲۴	۶۴	۵۱۲	۲۵۶
۴	۱۶	۶۴	۲۵۶	۴۰۹۶	۶۵۵۳۶
۵	۳۲	۱۶۰	۱۰۲۴	۳۲۷۶۸	۴۲۹۴۹۶۷۲۹۶

مثال 10: فرض کنید در ماشینی در هر ثانیه یک میلیارد عملیات پایه می تواند انجام شود. جدول زیر زمان اجرای تقریبی الگوریتم ها با مرتبه پیچیدگی محتلف را نشان می دهد:

اندازه ورودی	تابع پیچیدگی			
n	$\log_2 n$	n	n^2	2^n
10	$3\times10^{-9}s$	$10\times10^{-9}=10^{-8}s$	$10^2\times10^{-9}=10^{-7}s$	$2^{10}\times10^{-9}=10^3\times10^{-9}=10^{-6}s$
10^2	$7\times10^{-9}s$	$10^2\times10^{-9}=10^{-7}s$	$10^4\times10^{-9}=10^{-5}s$	بیش از صد میلیارد قرن
10^3	$10\times10^{-9}s$	$10^3\times10^{-9}=10^{-6}s$	$10^6\times10^{-9}=10^{-3}s$	بیش از صدهزار میلیارد قرن

قضیه 1: اگر $a_m > 0$ و $A(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0$ آنگاه داریم $A(n) = O(n^m)$.

اثبات:

چند نکته:

1: اگر تعداد دفعات اجرای دستوری در الگوریتمی $A(n)$ باشد گوییم زمان اجرای آن دستور $O(n^m)$ است.

2: اگر الگوریتمی k دستور به مرتبه های $O(n^{m1})$، $O(n^{m2})$، ....، $O(n^{mk})$ داشته باشد، آنگاه مرتبه آن الگوریتم $O(n^m)$ است که در آن $m = \max\{m1, m2, \dots, mk\}$.

3: اگر یک الگوریتم از دو زیرالگوریتم با زمان اجرای $T_1(n) = O\big(f(n)\big)$ و $T_2(n) = O\big(g(n)\big)$، آنگاه زمان اجرای الگوریتم اصلی برابر است با

$$\mathrm{T}(n) = \max\{O\big(f(n)\big), O\big(g(n)\big)\}$$

4: قانون ضرب: $T_1(n)T_2(n) = O(f(n)g(n))$.

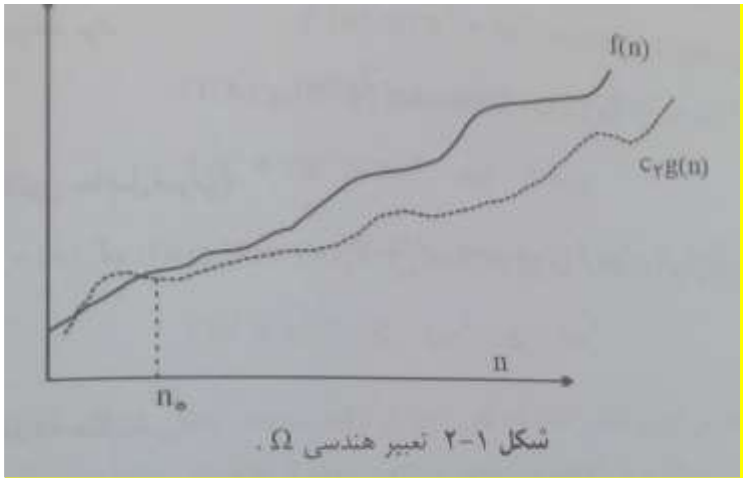
مثال 11: نشان دهید اگر $f(n)=(4n^3 + 5n^2 + 7n)(8\log n)$ آنگاه $f(n) = O(n^3 \log n)$.

مثال 12: اگر زمان اجرای یک الگوریتم $1000n^2$ و زمان اجرای الگوریتم دیگری $10n^3$ باشد، با نماد O الگوریتم اول از دومی بهتر است ولی برای $n < 100$ الگوریتم دو سریعتر است. بنابراین هنگام مقایسه دقیق سرعت الگوریتم باید به محدوده n توجه کرد. البته لازم به ذکر است که معمولا مقایسه الگوریتم ها برای n های بزرگ صورت می گیرد.

نکته: الگوریتم های با پیچیدگی $O(2^n)$ فقط برای مقادیر کوچک n کارایی دارند و برای nهای بزرگ الگوریتمهای با پیچیدگی هایی از مراتبی مانند $n, n^2, n^3, \log n, nlog\ n$ مفید هستند.

تعریف: Ω اومگای بزرگ (بررسی زمان اجرا در بهترین حالت)

گوییم $\mathrm{f(n)} = \Omega(\mathrm{g(n)})$ اگر عدد طبیعی n_0 وعدد حقیقی مثبت c_2 وجود داشته باشند بطوریکه برای هر $\mathrm{n} \geq n_0$، $|\mathrm{f(n)}| \geq c_2|\mathrm{g(n)}|$ (شکل 1-2 را ببینید).

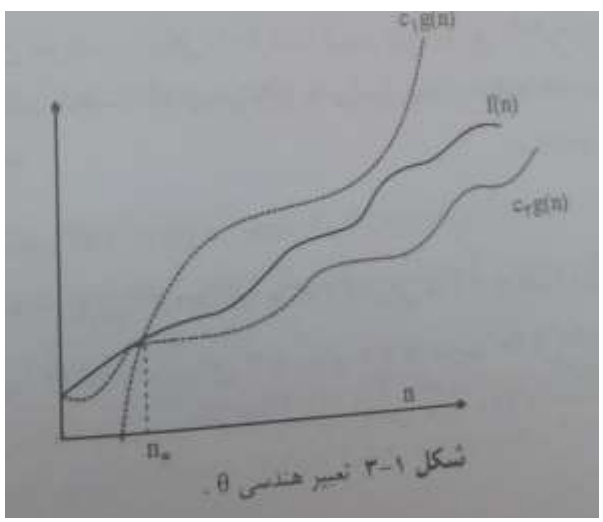


مثال 13: نشان دهید $f(n)=3n^3 + 2n^2 = \Omega(n^3)$.

تعریف: θ (بررسی حالت میانگین زمان اجرا)

گوییم $f(n) = \theta(g(n))$ اگر عدد طبیعی n_0 واعداد حقیقی مثبت c_1 و c_2 وجود داشته باشند بطوریکه برای هر $n \geq n_0$، $c_2 g(n) \leq f(n) \leq c_1 g(n)$

(شکل 1-3 را ببینید).



نتیجه 1: $f(n) = \theta(g(n))$ اگر و فقط اگر $f(n) = O(g(n))$ و $f(n) = \Omega(g(n))$.

مثال 14: نشان دهید $f(n)=3n^3 + 2n^2 = \theta(n^3)$

تعریف: (o اوی کوچک) گوییم $f(n) = o(g(n))$ اگر

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

مثال 15: $3n^3 + 2n^2 = o(n^4)$.

تعریف: (ω اومگای کوچک) گوییم $f(n) = \omega(g(n))$ اگر

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$$

مثال 16: مثال 13: $3n^3 + 2n^2 = \omega(n^2)$.

مقایسه خواص O، Ω، θ، o و ω

۱ـ خاصیت تقارنی : این خاصیت را فقط θ دارد :

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

۲ـ خاصیت بازتابی :

$$f(n)=\theta(f(n)) \quad , \quad f(n)=\Omega(f(n)) \quad , \quad f(n)=O(f(n))$$

خاصیت بازتابی را o و ω ندارند.

۳ـ خاصیت ترانهاده تقارنی :

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

$$f(n)=o(g(n)) \Leftrightarrow g(n)=\omega(f(n))$$

۴ـ خاصیت تعدی :

$$f(n)=O(g(n)) \quad , \quad g(n)=O(h(n)) \Rightarrow f(n)=O(h(n))$$
$$f(n)=\Omega(g(n)) \quad , \quad g(n)=\Omega(h(n)) \Rightarrow f(n)=\Omega(h(n))$$
$$f(n)=\theta(g(n)) \quad , \quad g(n)=\theta(h(n)) \Rightarrow f(n)=\theta(h(n))$$
$$f(n)=o(g(n)) \quad , \quad g(n)=o(h(n)) \Rightarrow f(n)=o(h(n))$$
$$f(n)=\omega(g(n)) \quad , \quad g(n)=\omega(h(n)) \Rightarrow f(n)=\omega(h(n))$$

مثال 17: فرض کنید آرایه A[1,..n] داده شده است و هدف مرتب سازی آن به صورت صعودی به کمک الگوریتم مرتب سازی حبابی است.

```
Algorithm Bubble _Sort (A, n)
(a) for i = ١ to  n-١  do
        {
(b)         for  j  =  n  downto   i + ١ do
            {
(c)             if  A[j-١] > A[j]   then
                {
(d)                 temp  =  A[j-١]
(e)                 A[j-١]  =  A[j]
(f)                 A[j]  =  temp
                }
            }
```

الگوریتم فوق نخست کوچک ترین عنصر را پس از n-1 مقایسه در جای اول و عنصر کوچک بعدی را پس از n-2 مقایسه در جای دوم و در نهایت عنصر n-1 ام را با یک مقایسه در جای n-1 قرار می دهد.

در این الگوریتم بعد از تکرار iام، آخرین i عضو آرایه مرتب شده هستند. در هر تکرار این الگوریتم در قسمت مرتب نشده دنبال بزرگترین عضو است.

الف: تحلیل بدترین پیچیدگی زمانی:

این حالت زمانی اتفاق می افتد که آرایه بصورت نزولی مرتب شده باشد. تعداد اجرای دستورات الگوریتم در جدول زیر خلاصه شده است:

```
Algorithm Bubble _Sort (A, n)
(a) for i = ۱ to n-۱ do
      {
(b)       for j = n downto i+۱ do
          {
(c)          if A[j-۱] > A[j] then
             {
(d)             temp = A[j-۱]
(e)             A[j-۱] = A[j]
(f)             A[j] = temp
             }
          }
```

تعداد دفعات اجرا	دستورالعمل
n − ۱	(a)
(n − ۱) + (n − ۲) + ... + ۲ + ۱	(b)
(n − ۱) + (n − ۲) + ... + ۲ + ۱	(c)
(n − ۱) + (n − ۲) + ... + ۲ + ۱	(d)
(n − ۱) + (n − ۲) + ... + ۲ + ۱	(e)
(n − ۱) + (n − ۲) + ... + ۲ + ۱	(f)

ب: تحلیل بهترین پیچیدگی زمانی:

این حالت زمانی اتفاق می افتد که آرایه بصورت صعودی مرتب شده باشد. تعداد اجرای دستورات الگوریتم در جدول زیر خلاصه شده است:

دستورالعمل	تعداد دفعات اجرا
(a)	$n-۱$
(b)	$(n-۱)+(n-۲)+...+۲+۱$
(c)	$(n-۱)+(n-۲)+...+۲+۱$
(d)	۰
(e)	۰
(f)	۰

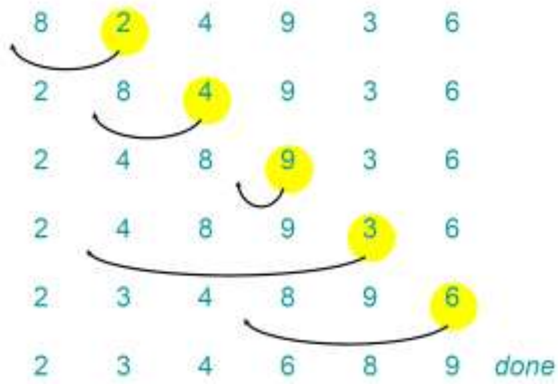
ج: تحلیل پیچیدگی در حالت میانگین:

طبق **نتیجه 1** پیچیدگی در این حالت $\theta(n^2)$ است.

مثال 18: تحلیل الگوریتم مرتب سازی درجی

فرض کنید آرایه A[1,..n] داده شده است و هدف مرتب سازی آن به صورت صعودی به کمک الگوریتم مرتب سازی درجی است.

```
INSERTION-SORT (A, n)      ⊳ A[1 . . n]
  for j ← 2 to n
       do key ← A[j]
          i ← j − 1
          while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i − 1
          A[i+1] = key
```



در این الگوریتم پس از i تکرار، i عنصر اول آرایه مرتب شده هستند. فرض کنید هدف محاسبه تعداد دفعات اجرای دستور مقایسه ای است.

الف: بدترین حالت: بدترین حالت زمانی است که حلقه داخلی همواره اجرا شود. مجموع تعداد اجرای حلقه داخلی $2+3+\cdots n=\frac{n(n+1)}{2}-1=O(n^2)$ است.

ب: بهترین حالت: این حالت زمانی اتفاق می افتد که حلقه داخلی اجرا نشود. جمع تعداد اجرای حلقه داخلی $1+1+\cdots+1=\Omega(n)$ است.

ج: حالت میانگین: در این حالت لازم است در هر مرحله نصف آرایه تا جایی که مرتب شده مقایسه شود. در نتیجه تعداد اجرای حلقه داخلی $O(n^2)$ است.