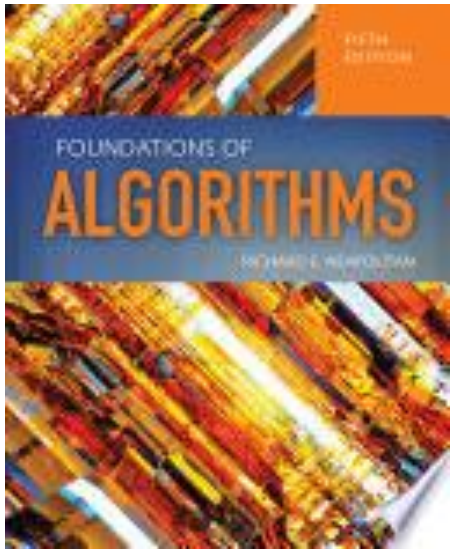
به نام خدا

درس طراحی الگوریتم ها

مرجع درس

- Richard Neapolitan, Kumarss Naimipour, **Foundations of Algorithms**, Jones & Bartlett Learning, 2015.



- **نسخه فارسی:**
- **طراحی الگوریتم ها**
- **نویسندگان: ریچارد نئوپولیتان، کیومرث نعیمی پور**
- **ترجمه: جعفرنژاد یا مترجمان دیگر...**

طرح درس

- فصل اول: الگوریتم ها: کارایی، تحلیل و مرتبه
- فصل دوم: روش تقسیم و حل (Divide-and-Conquer)
- فصل سوم: برنامه ریزی پویا (Dynamic Programming)
- فصل چهارم: روش حریصانه (The Greedy Approach)
- فصل پنجم: روش عقبگرد (The Backtracking Technique)
- فصل ششم: راهبرد شاخه و حد (Branch-and-Bound)
- نظریه NP

نمره درس

- امتحان میان ترم: ۵ نمره
- امتحان پایان ترم: ۱۱ نمره
- تکلیف و پروژه: ۴ نمره
- غیبت بیش از حد: کسر نمره

چرا این درس را مطالعه می کنیم

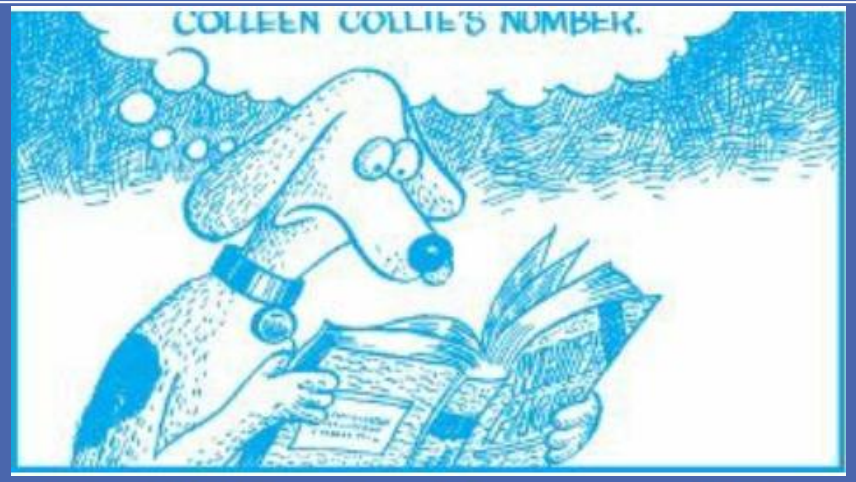
؟

به نام خدا

فصل اول

الگوریتم ها: کارایی، تحلیل و مرتبه

Algorithms: Efficiency, Analysis, and Order



دانشکده مهندسی کامپیوتر – دانشگاه اصفهان
دکتر مرجان کائدی

الگوریتم‌ها

الگوریتم

- الگوریتم: یک تکنیک حل مساله که منجر به روشی گام به گام در حل آن مساله شود.
- یک مساله را با استفاده از چندین تکنیک متفاوت می توان حل کرد. ولی یکی از آنها منجر به الگوریتمی می شود که سریع تر از بقیه است.
- باید تحلیل کنیم که کارایی الگوریتم از لحاظ زمان و حافظه به چه میزان است.

مساله ها

- مساله: پرسشی که به دنبال حل آن هستیم.

- مثال ۱:
- مساله: لیست S متشکل از n عدد را به ترتیب غیرنزولی مرتب کنید.
- پاسخ: اعداد مرتب شده

- مثال ۲:
- مساله: تعیین کنید که آیا x در لیست S متشکل از n عدد وجود دارد یا خیر.
- پاسخ: اگر x در S باشد، پاسخ مثبت است و اگر x در S نباشد، پاسخ منفی است.
- مثلا:

$$S = [10, 7, 11, 5, 13, 8]$$

نمونه ای از یک مساله

- هر مساله حاوی چند پارامتر است و نماینده طبقه ای از مسائل است که هر یک با نسبت دادن مقادیری به این پارامترها به دست می آیند.

- با انتساب مقادیر خاص به پارامترها، نمونه ای از آن مساله به دست می آید.

نمونه ای از یک مساله

- نمونه ای از مساله مثال ۱:

$S = [10, 7, 11, 5, 13, 8]$ and $n = 6$.

[5, 7, 8, 10, 11, 13]

- نمونه ای از مساله مثال ۲:

$S = [10, 7, 11, 5, 13, 8]$, $n = 6$, and $x = 5$.

"yes, x is in S."

- در واقع، الگوریتم یک روال گام به گام برای حل همه نمونه مساله ها است.

الگوریتم ۱: جستجوی ترتیبی (sequential search)

- آیا کلید x در آرایه S با n کلید قرار دارد؟

Inputs (parameters): positive integer n, array of keys S indexed from 1 to n, and a key x.

Outputs: location, the location of x in S (0 if x is not in S.)

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

نکاتی در مورد شبه کد الگوریتم ها

- این الگوریتم ها با شبه کد C++ نوشته شده است.
- ولی با کد C++ دقیقا منطبق نیست.

- مثلا: اندیس ها را از ۱ شروع کرده ایم. البته این تفاوت را می توان با تعریف آرایه n+1 تایی و عدم استفاده از عضو صفرم برطرف کرد.
- طول آرایه ها را در بعضی موارد، متغیر در نظر می گیریم.

```
keytype S[n + 1];

void example (int n)
{
   keytype S[2..n];
   :
}
```

نکاتی در مورد شبه کد الگوریتم ها

- دستورات شرطی را می توانیم به صورت زیر بنویسیم.

```
if (low ≤ x ≤ high) {
        ⋮
}
```

```
if (low <= x && x <= high){
        ⋮
}
```

نکاتی در مورد شبه کد الگوریتم ها

- برای جابجا کردن مقدار دو متغیر ممکن است به صورت زیر کد بنویسیم:

exchange x and y;

```
temp = x;
x = y;
y = temp;
```

نکاتی در مورد شبه کد الگوریتم ها

- ممکن است از ساختار کنترلی غیراستانداردی مانند زیر استفاده کنیم:

```
repeat (n times){
    :
}
```

نکاتی در مورد شبه کد الگوریتم ها

- عملگرهای منطقی و رابطه ای:

Operator	C++ symbol
and	&&
or	||
not	!

Comparison	C++ code
$x = y$	$(x == y)$
$x \neq y$	$(x! = y)$
$(x \leq y)$	$(x <= y)$
$x \geq y$	$(x >= y)$

الگوریتم ۲: محاسبه مجموع عناصر آرایه

- تمام اعداد موجود در آرایه n عنصری را با هم جمع کنید.

Problem: Add all the numbers in the array S of n numbers.

Inputs: positive integer n, array of numbers S indexed from 1 to n.

Outputs: sum, the sum of the numbers in S.

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

الگوریتم ۳: مرتب سازی تعویضی (exchange sort)

- n کلید را به ترتیب غیرنزولی مرتب کنید.

Problem: Sort n keys in nondecreasing order.

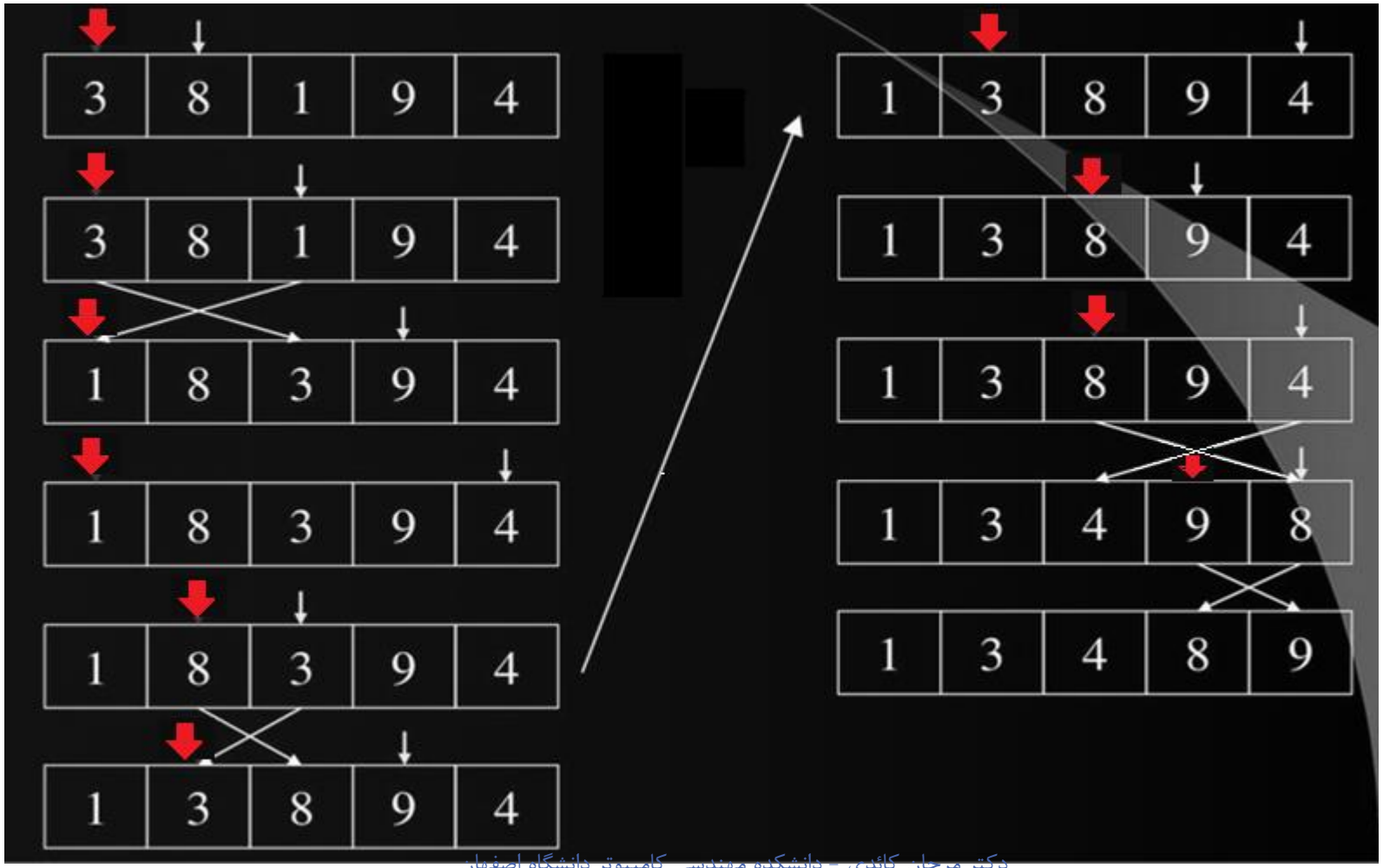
Inputs: positive integer n, array of keys S indexed from 1 to n..

Outputs: the array S containing the keys in nondecreasing order.

```
void exchangesort (int n, keytype S[])
{
  index i, j;
  for (i=1;i<= n; i++)
    for (j=i+1; j<= n; j++)
      if (S[j] < S[i])
          exchange S[i] and S[j];
}
```

- کوچکترین عدد پس از نخستین گذر از حلقه for با اندیس i در محل اول، بعدی در محل دوم و الی آخر قرار می گیرد.

مثال مرتب سازی تعویضی



الگوریتم ۴: ضرب ماتریس ها

- حاصلضرب دو ماتریس n×n را تعیین کنید.

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix},$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j}.$$

$$\begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix} \times \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} = \begin{bmatrix} 2\times5+3\times6 & 2\times7+3\times8 \\ 4\times5+1\times6 & 4\times7+1\times8 \end{bmatrix} = \begin{bmatrix} 28 & 38 \\ 26 & 36 \end{bmatrix}$$

$$c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj} \qquad \text{for } 1 \le i, j \le n.$$

ضرب ماتریس ها

Problem: Determine the product of two n x n matrices.
Inputs: a positive integer n, two-dimensional arrays of numbers A and B, each of which has both its rows and columns indexed from 1 to n.
Outputs: a two-dimensional array of numbers C, which has both its rows and columns indexed from 1 to n, containing the product of A and B.

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

- اهمیت ساخت الگوریتم های کارآمد

اهمیت ساخت الگوریتم های کارآمد

- سرعت کامپیوتر هر چقدر بالا باشد و قیمت حافظه هرچقدر کاهش یابد، همچنان کارایی الگوریتم ها باید مورد نظر قرار بگیرد.

- جستجوی دودویی در مقایسه با جستجوی ترتیبی

جستجوی دودویی (binary search)

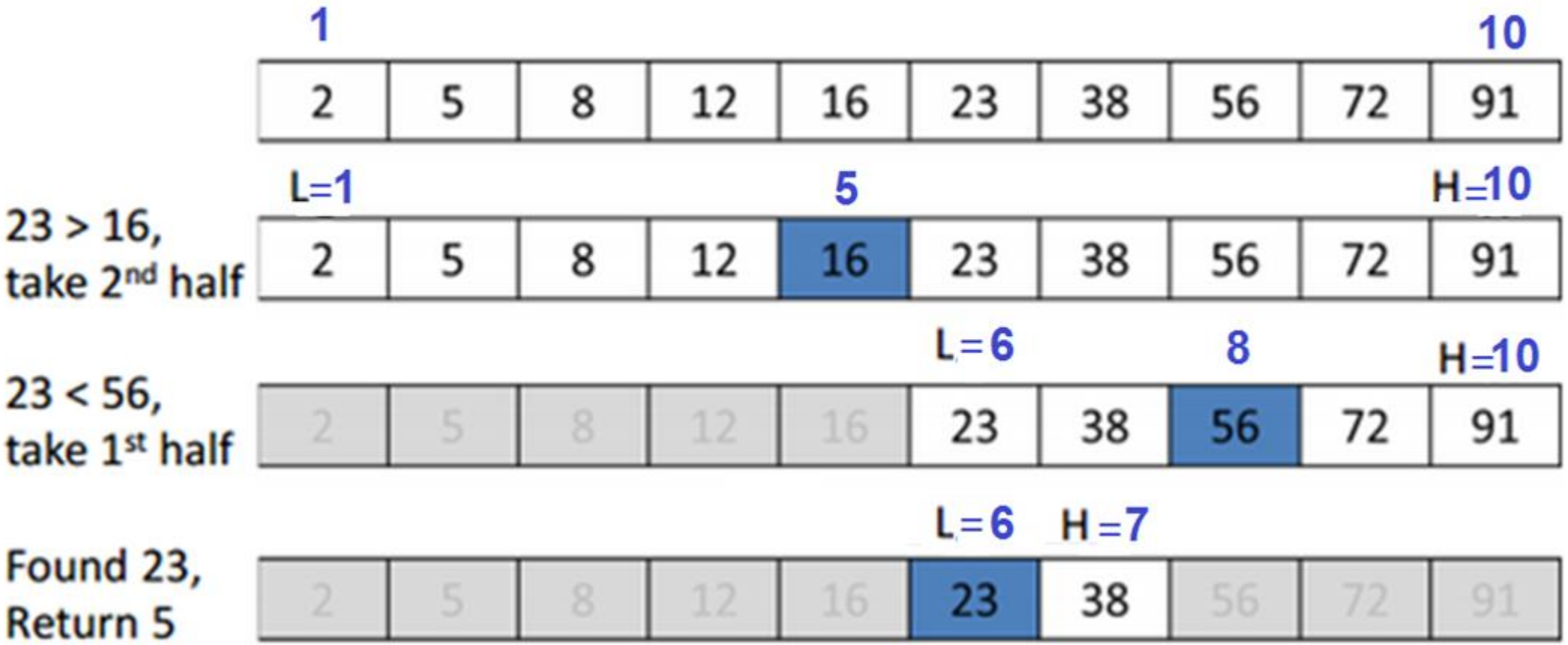
Problem: Determine whether x is in the sorted array S of n keys.
Inputs: positive integer n, sorted (nondecreasing order) array of keys S indexed from 1 to n, a key x.
Outputs: location, the location of x in S (0 if x is not in S).

```
void binsearch (int n, const keytype S[], keytype x, index& location)
{
  index low, high, mid;

  low = 1; high = n;
  location = 0;
  while (low < = high &&  location = = 0){
      mid = ⌊(low + high)/2⌋;
      if (x = = S[mid])
          location = mid;
      else if (x < S[ mid ])
              high = mid - 1;
           else
              low = mid + 1;
    }
}
```

- تعیین کنید که آیا x در یک آرایه مرتب n عضوی S وجود دارد یا خیر.

جستجوی عدد ۲۳ در آرایه ۱۰ عضوی به روش جستجوی دودویی



مقایسه جستجوی دودویی و جستجوی ترتیبی

کار انجام شده توسط جستجوی دودویی و جستجوی ترتیبی را با هم مقایسه می کنیم:

برای این منظور، تعداد مقایسه های انجام شده در هر یک از این دو الگوریتم را تعیین می کنیم.

اگر آرایه حاوی ۳۲ عنصر باشد، و x در آرایه نباشد، الگوریتم جستجوی ترتیبی x را با همه اعضای آرایه مقایسه می کند تا به این نتیجه برسد.

به طور کلی جستجوی ترتیبی حداکثر n مقایسه انجام می دهد تا تعیین کند که آیا x در آرایه S است یا خیر.

در جستجوی دودویی برای آرایه ۳۲ عضوی، بیشترین مقایسه وقتی است که x از همه اعضای آرایه بزرگتر باشد. در این حالت الگوریتم ۶ مقایسه را انجام می دهد.

در هر بار گذر از حلقه while، x فقط یکبار با S(mid) مقایسه می شود.

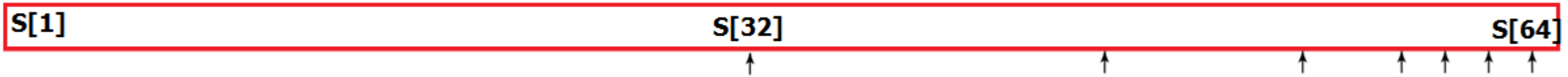
$$6 = \lg 32 + 1$$


S[1]
S[16]
S[24]
S[28]
S[30]
S[31]
S[32]
1st
2nd
3rd
4th
5th
6th

مقایسه جستجوی دودویی و جستجوی ترتیبی

- اگر اندازه آرایه را دو برابر کنید (یعنی حاوی ۶۴ عضو)، جستجوی دودویی فقط یک مقایسه بیشتر برای پیدا کردن X انجام می دهد.

- به طور کلی هر بار که اندازه آرایه دوبرابر می شود، یکبار مقایسه اضافه می شود.


S[1]
S[32]
S[64]

تعداد مقایسه های انجام شده در جستجوی دودویی و ترتیبی، هنگامی که X بزرگتر از همه اعضای آرایه باشد

Array Size	Number of Comparisons by Sequential Search	Number of Comparisons by Binary Search
128	128	8
1,024	1,024	11
1,048,576	1,048,576	21
4,294,967,296	4,294,967,296	33

دنباله فیبوناچی (Fibonacci sequence)

- محاسبه جمله n ام به صورت بازگشتی:

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2} \qquad \text{for } n \geq 2\,.$$

$$f_2 = f_1 + f_0 = 1 + 0 = 1$$
$$f_3 = f_2 + f_1 = 1 + 1 = 2$$
$$f_4 = f_3 + f_2 = 2 + 1 = 3$$
$$f_5 = f_4 + f_3 = 3 + 2 = 5, \text{ etc.}$$

الگوریتم ۶: محاسبه جمله nام فیبوناچی به صورت بازگشتی

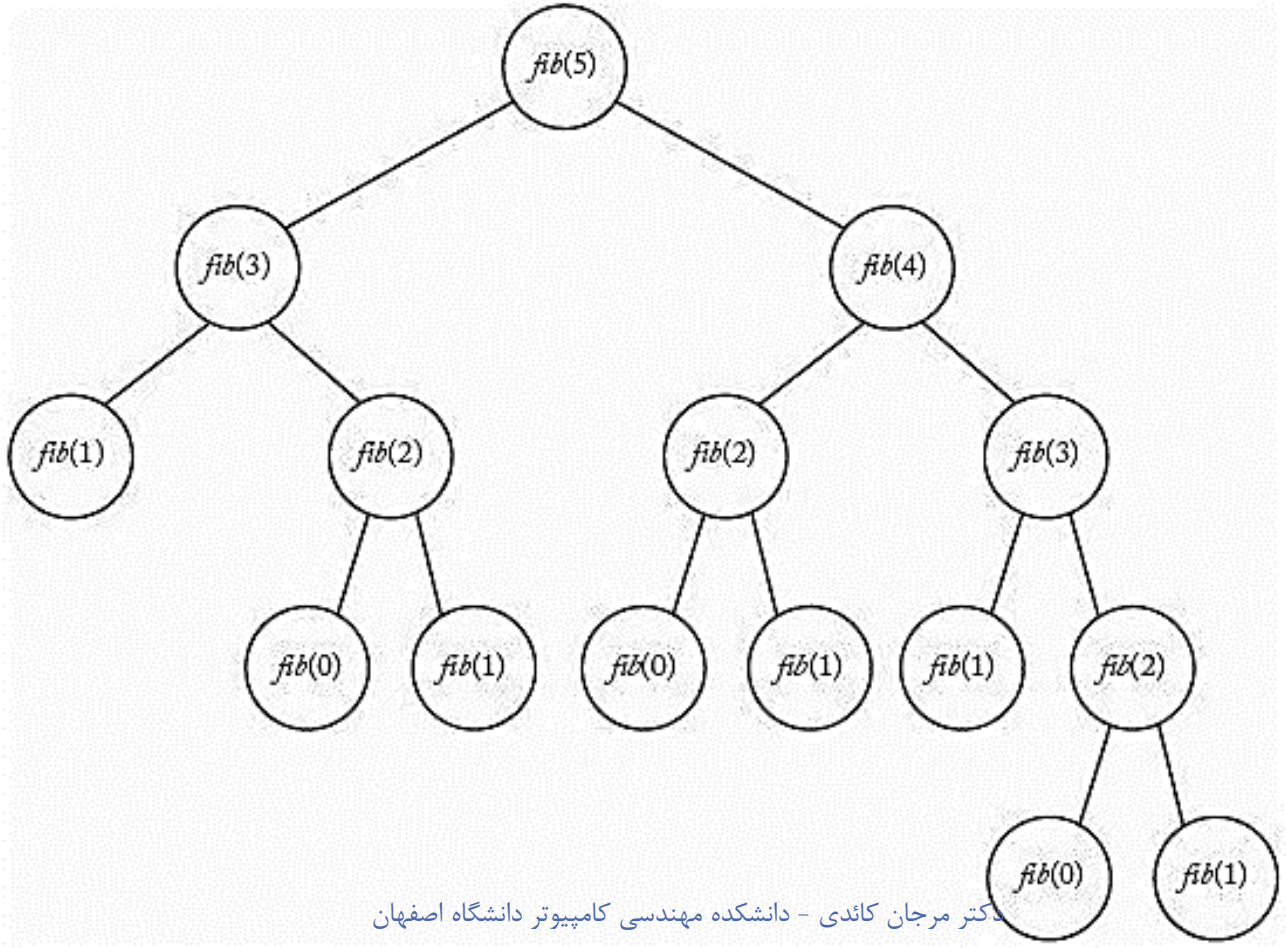
Problem: Determine the nth term in the Fibonacci sequence.

Inputs: a nonnegative integer n.

Outputs: fib, the nth term of the Fibonacci sequence.

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

درخت بازگشتی برای جمله پنجم دنباله فیبوناچی



تعداد جملات محاسبه شده، به ازای $0 \leq n \leq 6$

- در هفت جمله اول، با هر بار افزایش n به میزان ۲، تعداد جملات درخت به بیش از دو برابر افزایش می یابد.

n	Number of Terms Computed
0	1
1	1
2	3
3	5
4	9
5	15
6	25

تعداد جملات در درخت بازگشتی فیبوناچی

- اگر با افزودن ۲ به n، تعداد جملات درخت بیش از ۲ برابر شود، پس برای n که عددی زوج است داریم:

$$
\begin{aligned}
T(n) &> 2 \times T(n-2) \\
&> 2 \times 2 \times T(n-4) \\
&> 2 \times 2 \times 2 \times T(n-6) \\
&\vdots \\
&> \underbrace{2 \times 2 \times 2 \times 2 \times \cdots \times 2}_{n/2 \text{ terms}} \times T(0)
\end{aligned}
$$

قضیه ۱:

- اگر T(n) تعداد جملات موجود در درخت بازگشتی فیبوناچی باشد، آنگاه به ازای n≥2 داریم:

$$T(n) > 2^{n/2}.$$

اثبات قضیه ۱

- پایه استقرا:

$$T(2) = 3 > 2 = 2^{2/2}$$

$$T(3) = 5 > 2.8323 \approx 2^{3/2}$$

- فرض استقرا: فرض می کنیم برای همه مقادیر m کوچکتر از n، رابطه درست است یعنی داریم:

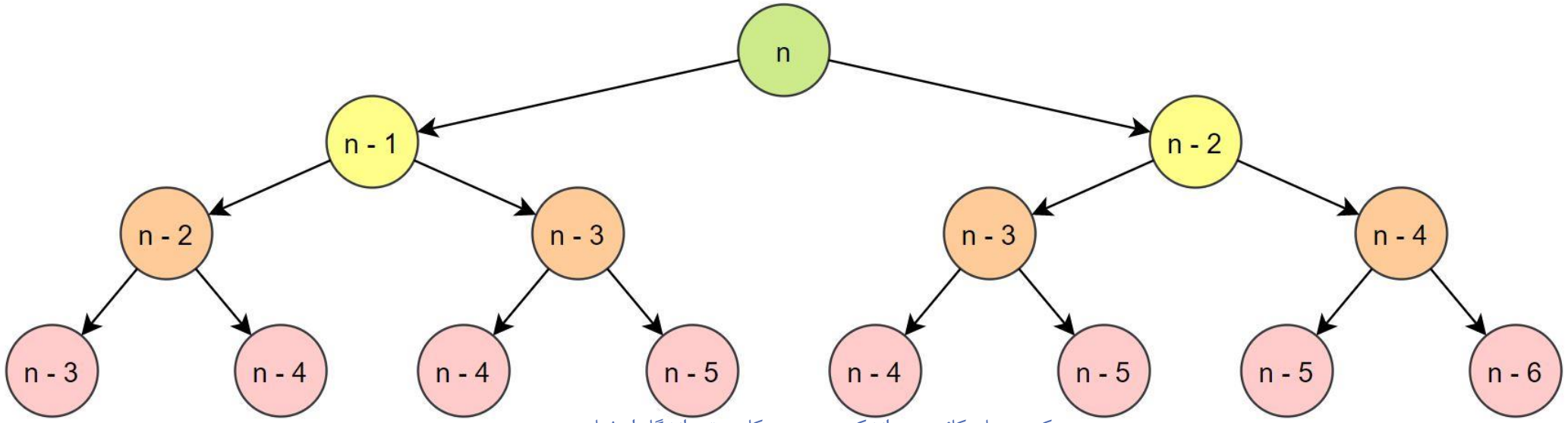
$$T(m) > 2^{m/2}.$$

- گام استقرا: نشان می دهیم رابطه برای n هم درست است:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &> 2^{(n-1)/2} + 2^{(n-2)/2} + 1 \qquad \text{(by induction hypothesis)} \\ &> 2^{(n-2)/2} + 2^{(n-2)/2} = 2 \times 2^{(\frac{n}{2})-1} = 2^{n/2}. \end{aligned}$$

گام استقرا:

$$T(n) = T(n-1) + T(n-2) + 1$$
$$> 2^{(n-1)/2} + 2^{(n-2)/2} + 1 \qquad \text{(by induction hypothesis)}$$
$$> 2^{(n-2)/2} + 2^{(n-2)/2} = 2 \times 2^{(\frac{n}{2})-1} = 2^{n/2}.$$



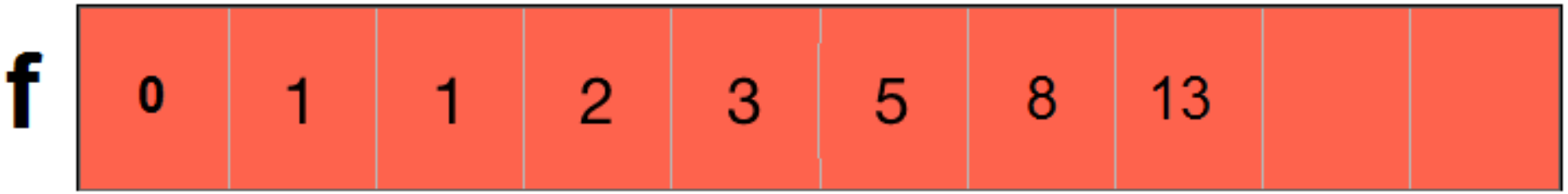
الگوریتم ۷: محاسبه جمله n ام فیبوناچی به روش تکراری یا iterative

Problem: Determine the nth term in the Fibonacci sequence.
Inputs: a nonnegative integer n.
Outputs : fib2, the nth term in the Fibonacci sequence.

- یک الگوریتم کارآمد

```
int fib2 (int n)
{
  index i;
  int f[o .. n];
  f[ o ] = o;
  if (n > o)
       f[ 1 ] = 1;
       for (i = 2; i<= n; i++)
         f[ i ] = f[i - 1] + f [i -2 ];
    }
  return f[ n ];
}
```

مثال: محاسبه سری فیبوناچی به روش تکراری



محاسبه جمله n ام فیبوناچی به روش تکراری

- این الگوریتم برای تعیین جمله n ام، n+1 جمله را محاسبه می کند.

- الگوریتم را می توان بدون استفاده از آرایه f نوشت زیرا در هر بار تکرار حلقه، تنها دو جمله آخر مورد نیاز هستند.

مقایسه الگوریتم های بازگشتی و تکراری فیبوناچی

n	$n+1$	$2^{n/2}$	Execution Time Using Algorithm 1.7	Lower Bound on Execution Time Using Algorithm 1.6
40	41	1,048,576	41 ns*	1048 $\mu s^{\dagger}$
60	61	1.1×10^9	61 ns	1 s
80	81	1.1×10^{12}	81 ns	18 min
100	101	1.1×10^{15}	101 ns	13 days
120	121	1.2×10^{18}	121 ns	36 years
160	161	1.2×10^{24}	161 ns	3.8×10^7 years
200	201	1.3×10^{30}	201 ns	4×10^{13} years

*1 ns $= 10^{-9}$ second.

$^{\dagger}1\ \mu s = 10^{-6}$ second.

- با فرض اینکه کامپیوتر هر جمله را در یک نانوثانیه انجام دهد.

- تحلیل الگوریتم ها

تحلیل الگوریتم ها و اندازه ورودی

- کارایی الگوریتم را با تعیین دفعاتی که یک عمل اصلی انجام می شود به عنوان تابعی از اندازه ورودی تحلیل می کنیم.

- در الگوریتم هایی مثل مرتب سازی، جستجو، و جمع اعضای آرایه، n یعنی تعداد اعضای آرایه را به عنوان اندازه ورودی در نظر می گیریم.

- در ضرب ماتریس ها، تعداد سطرها و ستون ها را به عنوان اندازه ورودی در نظر می گیریم.
- در برخی الگوریتم های مربوط به گراف، تعداد رئوس و تعداد یال ها را به عنوان اندازه ورودی در نظر می گیریم.

اندازه ورودی

- در محاسبه جمله سری فیبوناچی، تعداد بیت ها برای کد کردن x را به عنوان اندازه ورودی درنظر می گیریم:

- مثلا:

$$n = 13 = \underbrace{1101}_{4\ \text{bits}}{}_2$$

- که می شود $\lfloor \lg n \rfloor + 1$

تحلیل الگوریتم ها و دستور اصلی

- پس از تعیین اندازه ورودی، یک یا چند دستور انتخاب می کنیم به طوری که کل کار انجام شده توسط الگوریتم، تقریبا متناسب با تعداد دفعاتی باشد که این دستور یا دستورات انجام می شوند.

- این دستور یا دستورات را عمل اصلی (basic operation) در الگوریتم می گویند.

- تحلیل پیچیدگی زمان (time complexity analysis) یک الگوریتم عبارت است از تعیین تعداد دفعاتی که عمل اصلی به ازای هر اندازه ورودی انجام می شود.

تحلیل پیچیدگی زمانی حالت معمول

- در برخی موارد، تعداد دفعاتی که دستور اصلی اجرا می شود، نه تنها به اندازه ورودی، بلکه به مقادیر ورودی نیز بستگی دارد.
- مثلا در جستجوی ترتیبی، اگر x در اولین خانه آرایه باشد، عمل اصلی یک بار انجام می شود.
- و اگر در آرایه نباشد، n بار انجام می شود.
- ولی در الگوریتم جمع اعضای آرایه، همواره عمل اصلی n بار انجام می شود. در این حالت، T(n) تعداد دفعاتی است که الگوریتم، عمل اصلی را برای یک نمونه از n انجام می دهد.
- T(n) پیچیدگی زمانی الگوریتم در حالت معمول نامیده می شود.

تحلیل پیچیدگی زمانی حالت معمول برای الگوریتم جمع اعضای آرایه

- عمل اصلی: افزودن یک عنصر آرایه به sum

- اندازه ورودی: n یعنی تعداد اعضای آرایه

$$T(n) = n.$$

تحلیل پیچیدگی زمانی حالت معمول برای الگوریتم مرتب سازی تعویضی

- عمل اصلی: دستور مقایسه S[j] با S[i] (یا دستور انتساب)
- اندازه ورودی: n تعداد عناصری است که باید مرتب شوند.

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

تحلیل پیچیدگی زمانی حالت معمول برای الگوریتم ضرب ماتریس ها

- عمل اصلی: دستور ضرب در داخلی ترین حلقه for
- اندازه ورودی: n یعنی تعداد سطرها و ستون ها

$$T(n) = n \times n \times n = n^3.$$

تحلیل پیچیدگی زمانی حالت معمول برای الگوریتم جستجوی ترتیبی؟

- در الگوریتم جستجوی ترتیبی، به ازای همه نمونه های n , عمل اصلی به تعداد یکسانی انجام نمی شود.

- پس این الگوریتم فاقد پیچیدگی زمانی برای حالت معمول است.

- W(n) یعنی حداکثر دفعات اجرای عمل اصلی را پیچیدگی زمانی در بدترین حالت می گویند.

تحلیل پیچیدگی زمانی در بدترین حالت برای الگوریتم جستجوی ترتیبی

- عمل اصلی: مقایسه یک عنصر آرایه با x

- اندازه ورودی: n یعنی تعداد عناصر موجود در آرایه

$$W(n) = n.$$

تحلیل پیچیدگی زمانی حالت میانگین

- ممکن است علاقمند باشیم بدانیم که الگوریتم به طور میانگین چه کارایی دارد.
- برای محاسبه A(n)، باید به همه خروجی های ممکن با اندازه n، احتمالی را نسبت دهیم.
- اگر x در آرایه باشد، فرض می کنیم احتمال وجود آن در هر یک از خانه های آرایه یکسان است.
- اگر اطلاعاتی داشته باشیم که نشان دهنده توزیع دیگری باشد، باید از آن توزیع (یعنی توزیعی غیر از توزیع یکنواخت) استفاده کنیم.
- یک مقدار میانگین را فقط وقتی می توان معمولی خواند که حالت های واقعی از میانگین انحراف زیادی نداشته باشد.

تحلیل پیچیدگی زمانی در حالت میانگین برای جستجوی ترتیبی

- عمل اصلی: مقایسه یک عنصر آرایه با x
- اندازه ورودی: n یعنی تعداد عناصر موجود در آرایه

- نخست حالتی را تحلیل می کنیم که می دانیم x در آرایه وجود دارد. احتمال اینکه x در خانه k ام حافظه باشد، 1/n است.

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

- بنابراین انتظار می رود که به طور میانگین حدود نیمی از آرایه جستجو شود.

تحلیل پیچیدگی زمانی در حالت میانگین برای جستجوی ترتیبی (ادامه)

- حال موردی را در نظر می گیریم که ممکن است x در آرایه نباشد.
- فرض می کنیم احتمال وجود آن در هر یک از خانه های ۱ تا n مساوی است.
- احتمال اینکه x در خانه kام باشد، p/n است و احتمال آنکه در آرایه نباشد، 1-p است.

نبودن x در آرایه بودن x در آرایه

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

$$= \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p) = n\left(1 - \frac{p}{2}\right) + \frac{p}{2}.$$

- اگر p=1 باشد، مانند حالت قبل به $A(n) = (n+1)/2$ می رسیم.
- اگر p=0.5 آنگاه $A(n)=3n/4+1/4$. یعنی حدود ¾ آرایه به طور میانگین جستجو می شود.

تحلیل پیچیدگی زمانی در بهترین حالت

- B(n) به عنوان حداقل دفعاتی تعریف می شود که عمل اصلی برای اندازه ورودی n اجرا می گردد.
- یعنی حداقل دفعاتی که عمل اصلی برای اندازه ورودی n اجرا می شود.
- B(n) را پیچیدگی زمانی در بهترین حالت می گویند.

تحلیل پیچیدگی زمانی در بهترین حالت برای جستجوی ترتیبی

- عمل اصلی: مقایسه یک عنصر آرایه با x
- اندازه ورودی: n یعنی تعداد عناصر موجود در آرایه

$$B(n) = 1.$$

تحلیل پیچیدگی زمانی در بهترین و بدترین حالت

- برای الگوریتم هایی که فاقد پیچیدگی زمانی در حالت معمول هستند، تحلیل های بدترین حالت و حالت میانگین را بیشتر از تحلیل بهترین حالت انجام می دهیم.

- تحلیل حالت میانگین از آن لحاظ ارزشمند است که می گوید هنگامی که الگوریتم چندین بار روی چندین ورودی متفاوت امتحان شود، چقدر زمان می برد.

- تحلیل بدترین حالت مفیدتر از تحلیل بهترین حالت است زیرا مرز بالایی زمان صرف شده توسط الگوریتم را ارائه می دهد.

تابع پیچیدگی

- تابع پیچیدگی می تواند هر تابعی از اعداد صحیح مثبت به اعداد حقیقی غیرمنفی باشد.

- مثلا:

$$f(n) = n$$
$$f(n) = n^2$$
$$f(n) = \lg n$$
$$f(n) = 3n^2 + 4n$$

تحلیل درستی الگوریتم

- در این درس تحلیل الگوریتم به معنی تحلیل کارایی بر اساس زمان و حافظه است.
- انواع دیگری از تحلیل، مثلا تحلیل درستی الگوریتم در این درس مورد نظر نیست.

- مرتبه الگوریتم

مقدمه ای بر مرتبه

- دو الگوریتم با پیچیدگی زمانی معمول 100n و $0.01n^2$ داریم.
- اگر عمل اصلی هر دو الگوریتم به یک میزان زمان نیاز داشته باشد، و سربارها به یک اندازه باشد، الگوریتم اول در صورتی کارایی بیشتر خواهد داشت که:

$$0.01n^2 > 100n.$$

- با تقسیم طرفین بر 0.01n داریم:

$$n > 10{,}000.$$

- در تحلیل نظری الگوریتم ها رفتار نهایی مورد توجه است.
- می توان تعیین کرد که آیا رفتار نهایی یک الگوریتم از دیگری بهتر است یا خیر.

مثال

- دو الگوریتم برای حل یک مساله با پیچیدگی های زمان معمول به صورت زیر داریم:
- پیچیدگی n برای الگوریتم اول و n^2 برای الگوریتم دوم است. فرض کنید کامپیوتر برای پردازش دستور اصلی الگوریتم اول ۱۰۰۰ برابر پردازش دستور اصلی الگوریتم دوم زمان نیاز داشته باشد.
- از زمان های دستورات سربار صرف نظر می کنیم.
- زمان برای پردازش ورودی با اندازه n توسط الگوریتم اول: $n \times 1{,}000t$
- زمان برای پردازش ورودی با اندازه n توسط الگوریتم دوم: $n^2 \times t$
- می خواهیم ببینیم که الگوریتم اول چه زمانی کارایی بیشتری دارد. باید این نامعادله حل شود:

$$n^2 \times t > n \times 1{,}000t.$$

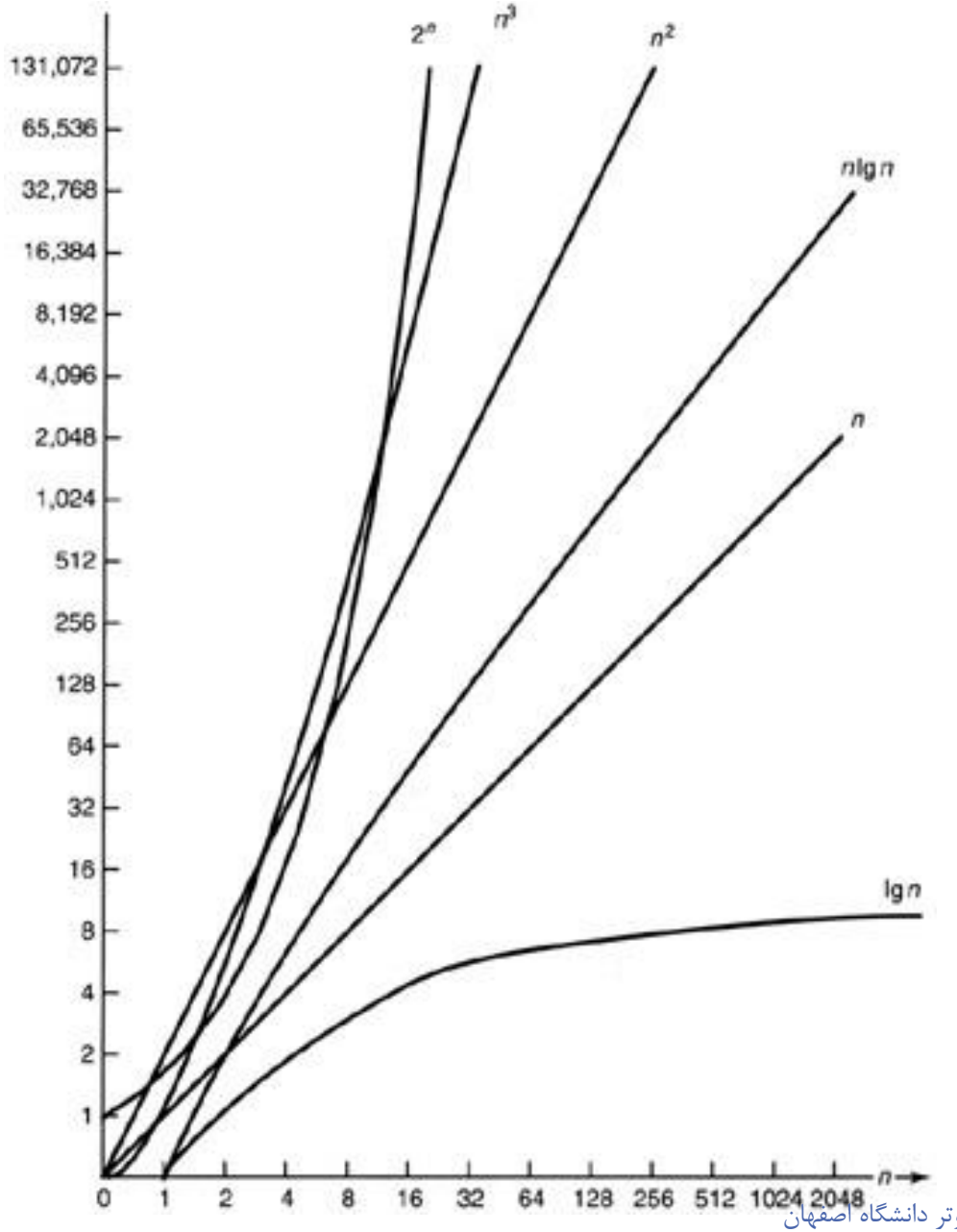
- با تقسیم طرفین نامعادله بر nt داریم: $n > 1{,}000.$
- بنابراین اگر در کاربرد مورد نظر هیچگاه n بزرگتر از ۱۰۰۰ نباشد، باید از الگوریتم دوم استفاده کرد.

مقدمه ای بر مرتبه

- الگوریتم هایی با پیچیدگی زمانی n و 100 n را الگوریتم های زمان خطی می گویند چون پیچیدگی آنها با اندازه ورودی رابطه خطی دارد.

- ولی الگوریتم های با پیچیدگی زمانی n^2 و $0.01n^2$ را الگوریتم های زمان درجه دوم می گویند.

سرعت رشد برخی توابع پیچیدگی متداول



زمان اجرا برای الگوریتم هایی با پیچیدگی های زمانی مختلف

- با فرض اینکه زمان پردازش هر دستور یک نانو ثانیه است:

n	$f(n) = lg\ n$	$f(n) = n$	$f(n) = n\ lg\ n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.003 μs[*]	0.01 μs	0.033 μs	0.10 μs	1.0 μs	1 μs
20	0.004 μs	0.02 μs	0.086 μs	0.40 μs	8.0 μs	1 ms[‡]
30	0.005 μs	0.03 μs	0.147 μs	0.90 μs	27.0 μs	1 s
40	0.005 μs	0.04 μs	0.213 μs	1.60 μs	64.0 μs	18.3 min
50	0.006 μs	0.05 μs	0.282 μs	2.50 μs	125.0 μs	13 days
10^2	0.007 μs	0.10 μs	0.664 μs	10.00 μs	1.0 ms	4×10^{13} years
10^3	0.010 μs	1.00 μs	9.966 μs	1.00 ms	1.0 s	
10^4	0.013 μs	10.00 μs	130.000 μs	100.00 ms	16.7 min	
10^5	0.017 μs	0.10 ms	1.670 ms	10.00 s	11.6 days	
10^6	0.020 μs	1.00 ms	19.930 ms	16.70 min	31.7 years	
10^7	0.023 μs	0.01 s	2.660 s	1.16 days	31,709 years	
10^8	0.027 μs	0.10 s	2.660 s	115.70 days	3.17×10^7 years	
10^9	0.030 μs	1.00 s	29.900 s	31.70 years		

[*] 1 ns = 10^{-9} second.

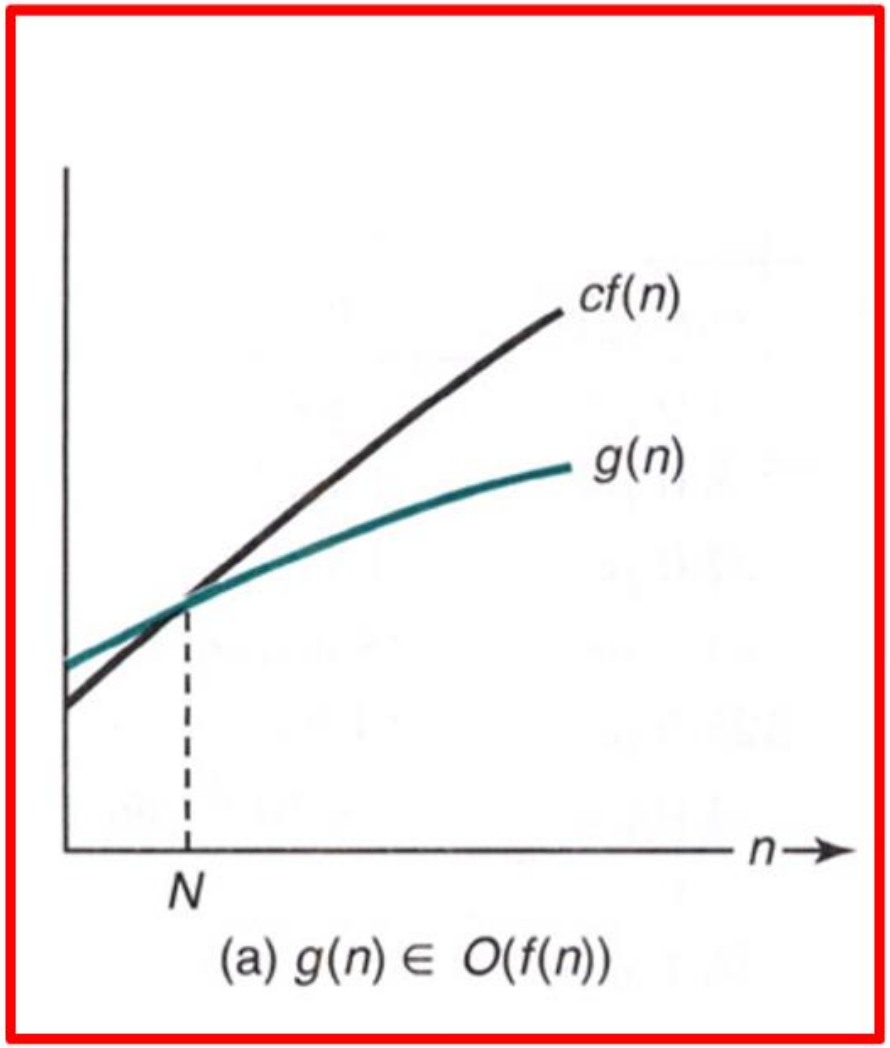
[‡] 1 ms = 10^{-3} second.

زمان اجرا برای الگوریتم هایی با پیچیدگی های زمانی مختلف

- الگوریتم های زمانی با مرتبه بالا وقتی مفید هستند که اندازه مساله خیلی بزرگ نباشد.

- الگوریتم با مرتبه زمانی درجه ۲، به ۳۱/۷ سال نیاز خواهد داشت تا نمونه ورودی با اندازه ۱ میلیارد را پردازش کند.

نماد O بزرگ (big-O)



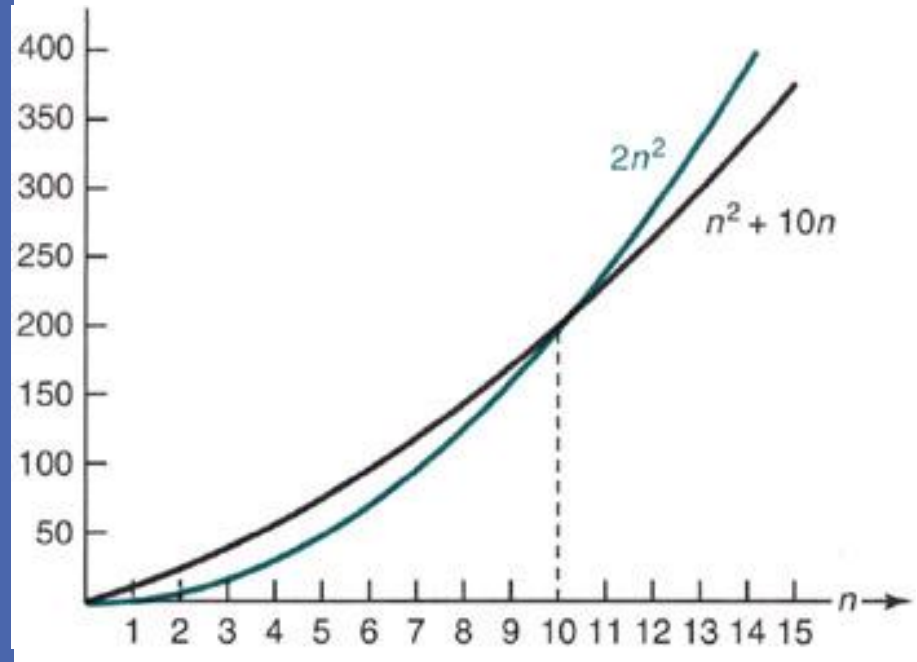
- تعریف: برای تابع پیچیدگی f(n)، O(f(n)) مجموعه ای از توابع پیچیدگی g(n) است
- (می گوییم g(n) ϵ Of(n))
- که برای آنها یک ثابت حقیقی مثبت c و یک عدد صحیح غیرمنفی N وجود دارد به قسمی که به ازای همه n≥N داریم:

$$g(n) \leq c \times f(n).$$

مثال

- این مثال نشان می دهد که اگر چه n^2+10n ابتدا بالای $2n^2$ است ولی برای n های بزرگتر از ۱۰ داریم:

$$n^2 + 10n \leq 2n^2$$



- برای تعریف O بزرگ می توانیم c را برابر ۲ و N را برابر ۱۰ در نظر بگیریم
- تا رابطه زیر به دست آید:

$$n^2 + 10n \in O\left(n^2\right).$$

- می توان گفت این تابع حداقل به خوبی تابع درجه دوم است.

O بزرگ (big-O)

- نماد O رفتار مجانبی توابع را توصیف می کند زیرا فقط با رفتار نهایی سرو کار دارد.
- گفته می شود O یک کران بالای مجانبی را روی تابع قرار می دهد.

مثال

- نشان می دهیم که $5n^2 \in O(n^2)$
- چون به ازای $n \geq 0$ داریم: $5n^2 \leq 5n^2,$
- می توان $c=5$ و $N=0$ را در نظر گرفت تا نتیجه مطلوب به دست آید.

مثال

- قبل تر به دست آوردیم که پیچیدگی الگوریتم مرتب سازی تعویضی طبق معادله زیر مشخص می شود:

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

- به ازای $n \geq 0$ داریم:

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

- می توان $c=1/2$ و $N=0$ را درنظر گرفت تا نتیجه گیری شود که: $T(n) \in O(n^2)$

مثال

- نشان می دهیم که

$$n^2 + 10n \in O(n^2)$$

- از آنجا که به ازای $n \geq 1$ داریم:

$$n^2 + 10n \leq n^2 + 10n^2 = 11n^2,$$

- می توانیم $c=11$ و $N=1$ را درنظر بگیریم تا نتیجه موردنظر به دست آید.

مثال

- نشان می دهیم که:

$$n^2 \in O(n^2 + 10n)$$

- از آنجا که در ازای $n \geq o$ داریم:

$$n^2 \leq 1 \times (n^2 + 10n)\,,$$

- می توانیم $c=1$ و $N=o$ را درنظر بگیریم تا نتیجه موردنظر به دست آید.

مثال

نشان می دهیم که $n \in O(n^2)$

از آنجا که به ازای $n \geq 0$ داریم:

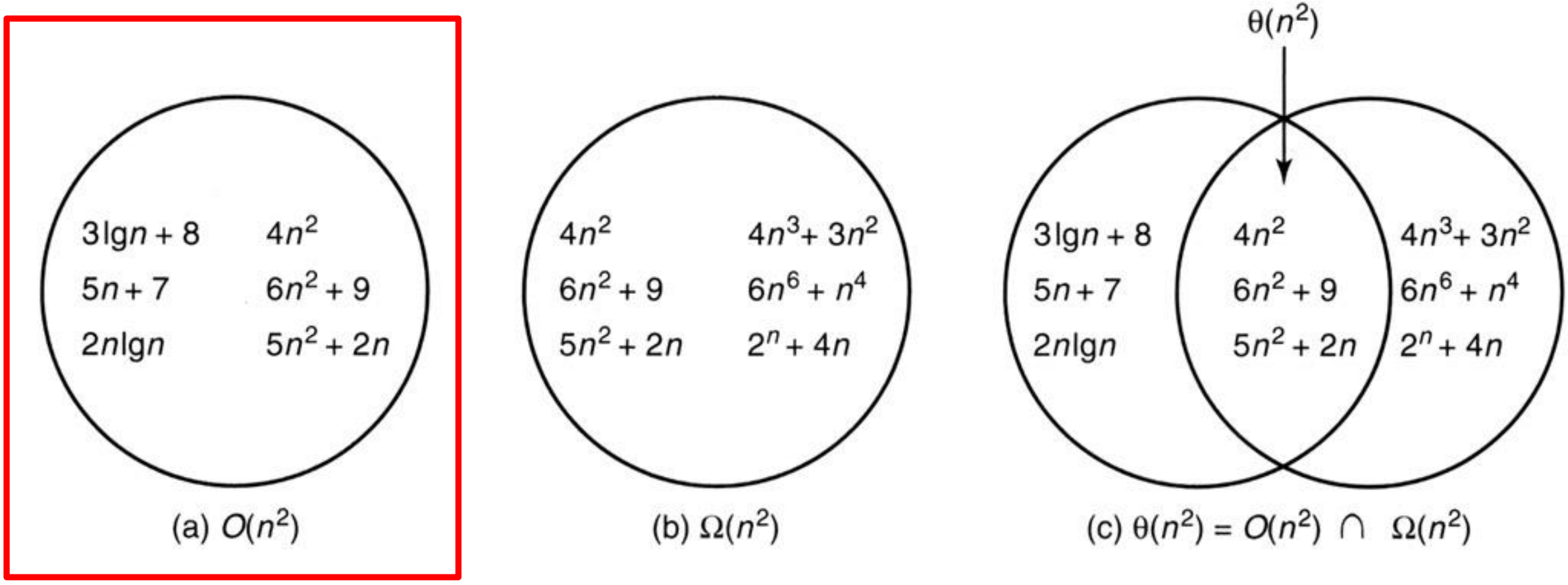
$$n \leq 1 \times n^2,$$

می توانیم $c=1$ و $N=1$ را درنظر بگیریم تا نتیجه مورد نظر به دست آید.

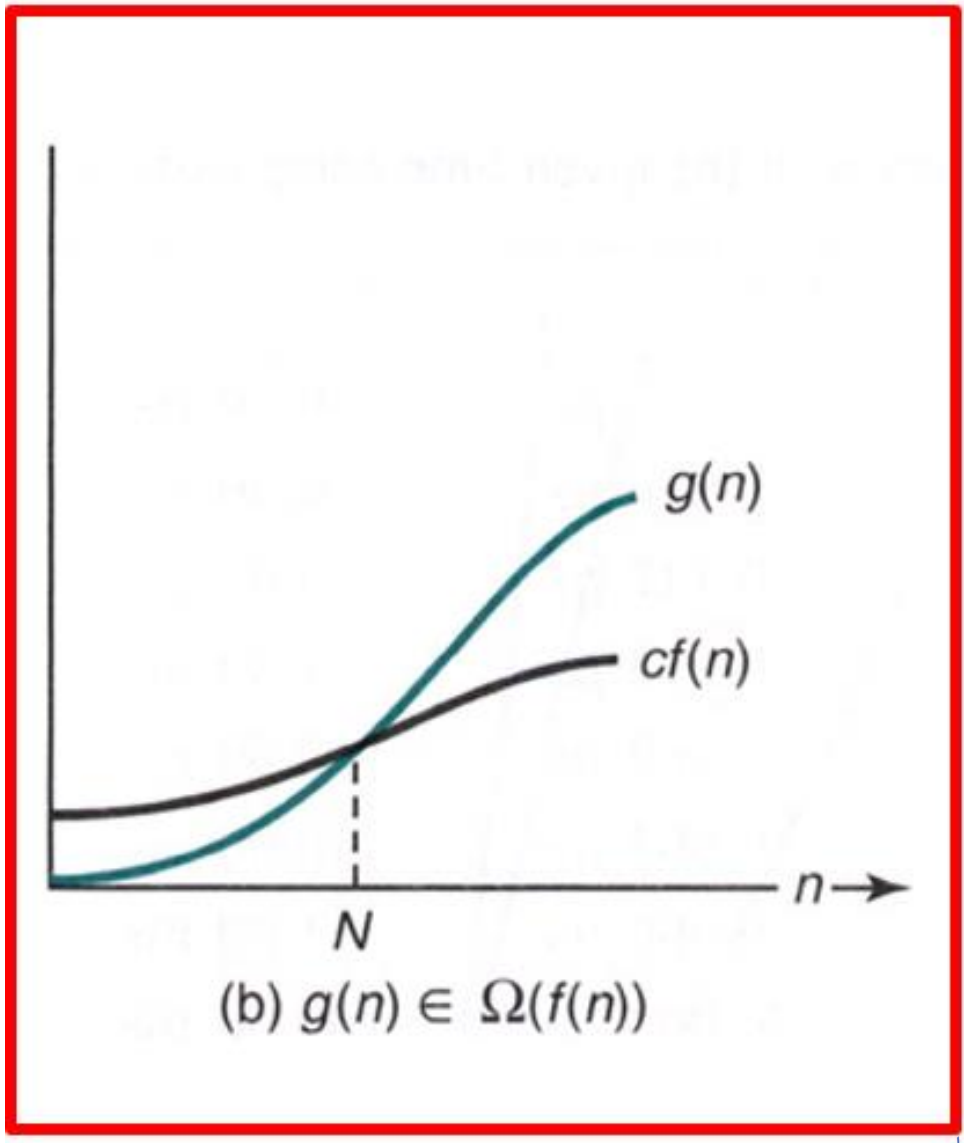
- نیاز نیست که تابع داخل O حتما یک تابع ساده باشد. می تواند هر تابع پیچیدگی باشد.

- ولی معمولا فرض می کنیم یک تابع ساده باشد.

چند عضو نمونه:



نماد امگا (Ω)



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

- تعریف: برای تابع پیچیدگی f(n)، Ω(f(n)) مجموعه ای از توابع پیچیدگی g(n) است
- (می گوییم $g(n) \in \Omega(f(n))$
- که برای آنها یک ثابت حقیقی مثبت c و یک عدد صحیح غیرمنفی N وجود دارد به قسمی که به ازای همه $n \geq N$ داریم:

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

مثال

نشان می دهیم که $5n^2 \in \Omega(n^2)$

از آنجا که به ازای $n \geq 0$ داریم:

$$5n^2 \geq 1 \times n^2,$$

می توانیم $c=1$ و $N=0$ را در نظر بگیریم تا نتیجه موردنظر به دست آید.

مثال

نشان می دهیم که $n^2 + 10n \in \Omega(n^2)$

از آنجا که به ازای $n \geq o$ داریم:

$$n^2 + 10n \geq n^2,$$

می توانیم $c=1$ و $N=o$ را در نظر بگیریم تا نتیجه موردنظر به دست آید.

مثال

- پیچیدگی زمانی الگوریتم مرتب سازی تعویضی را درنظر بگیرید. نشان می دهیم که:

$$T(n) = \frac{n(n-1)}{2} \in \Omega(n^2).$$

- به ازای $n \geq 2$ داریم:

$$n - 1 \geq \frac{n}{2}.$$

- بنابراین به ازای $n \geq 2$ خواهیم داشت:

$$\frac{n(n-1)}{2} \geq \frac{n}{2} \times \frac{n}{2} = \frac{1}{4}n^2,$$

- یعنی می توان $c=1/4$ و $N=2$ را درنظر گرفت تا نتیجه مورد نظر به دست آید.

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

- به لحاظ تحلیلی به این معنا است که در نهایت این تابع حداقل به بدی تابع درجه دوم محض است.

مثال

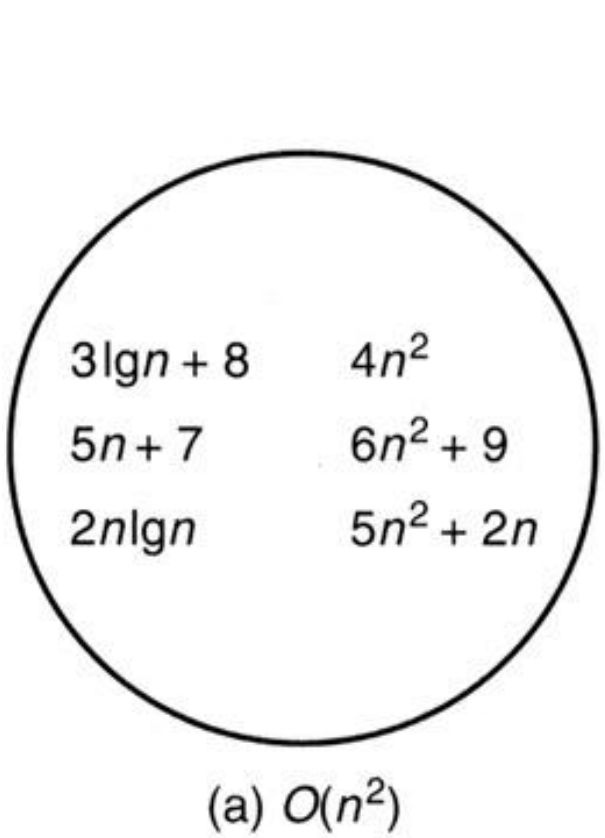
- نشان می دهیم که $n^3 \in \Omega(n^2)$

- از آنجا که اگر $n \geq 1$ باشد، داریم:

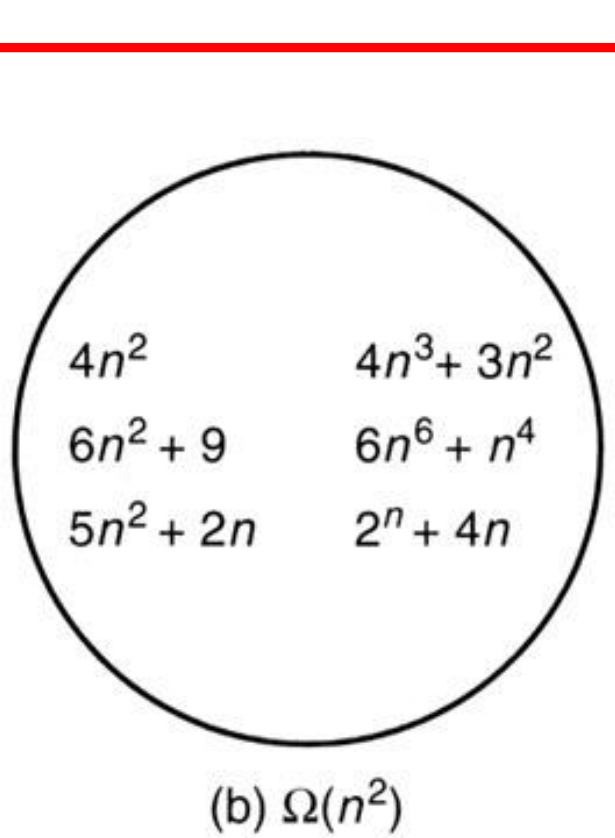
$$n^3 \geq 1 \times n^2,$$

- می توانیم $c=1$ و $N=1$ را درنظر بگیریم تا نتیجه موردنظر به دست آید.

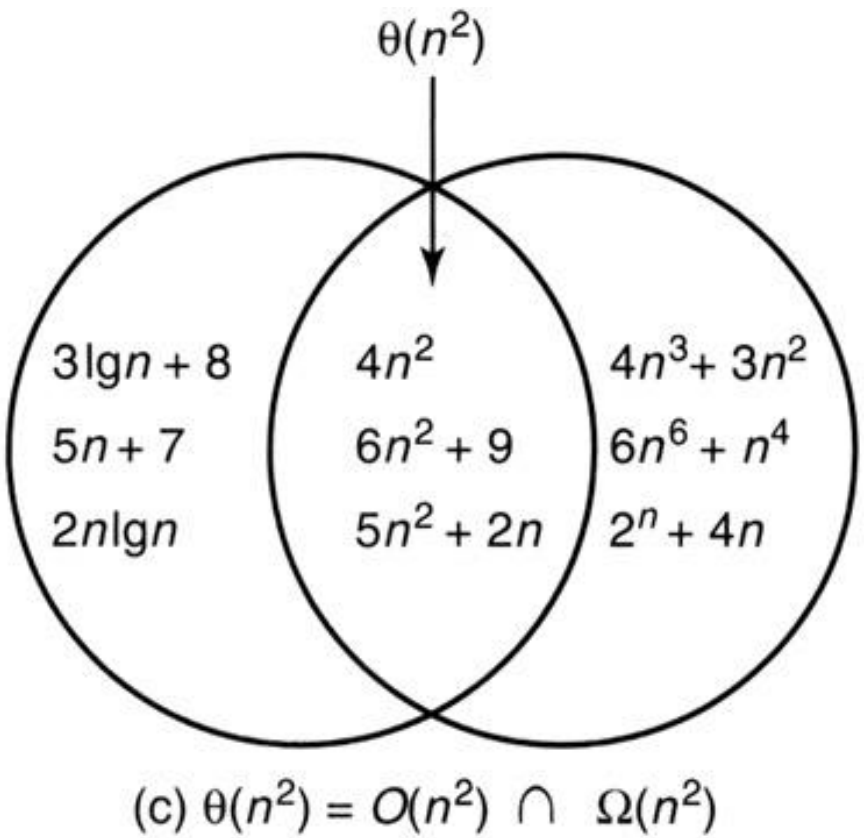
چند عضو نمونه:



(a) $O(n^2)$



(b) $\Omega(n^2)$



(c) $\theta(n^2) = O(n^2) \cap \Omega(n^2)$

نکته

- اگر تابعی هم در $O(n^2)$ و هم در $\Omega(n^2)$ باشد، می توان نتیجه گرفت یعنی حداقل به خوبی تابع درجه دوم محض و همچنین حداقل به بدی تابع درجه دوم محض است.

- این دقیقا نتیجه ای است که برای تعریف دقیق مرتبه نیاز داریم.

نماد تتا (θ)

- تعریف: برای یک تابع پیچیدگی مفروض f(n) داریم:

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n)).$$

- یعنی θ (f(n)) مجموعه ای از توابع پیچیدگی g(n) است که برای آنها ثابت های حقیقی مثبت c و d و عدد صحیح غیرمنفی N وجود دارد به نحوی که:

$$c \times f(n) \leq g(n) \leq d \times f(n).$$

مثال

- پیچیدگی زمانی الگوریتم مرتب سازی تعویضی را درنظر بگیرید.

- آنچه که تا کنون گفته شد به همراه هم اثبات می کند که:

$$T(n) = \frac{n(n-1)}{2} \quad \text{is in both} \quad O(n^2) \quad \text{and} \quad \Omega(n^2).$$

$$\Rightarrow T(n) \in O(n^2) \wedge \Omega(n^2) = \Theta(n^2)$$

مثال

- با استفاده از برهان خلف نشان می دهیم که n در $\Omega(n^2)$ نیست.

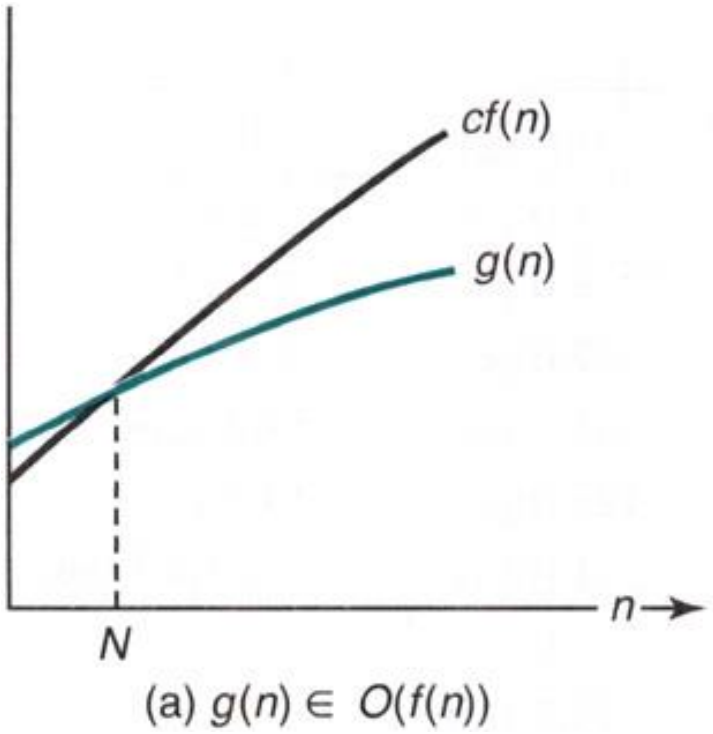
- با فرض اینکه $n \in \Omega(n^2)$ باشد، باید یک ثابت مثبت c و یک عدد صحیح غیرمنفی N وجود داشته باشد به قسمی که برای $n \geq N$ داشته باشیم:

$$n \geq cn^2.$$

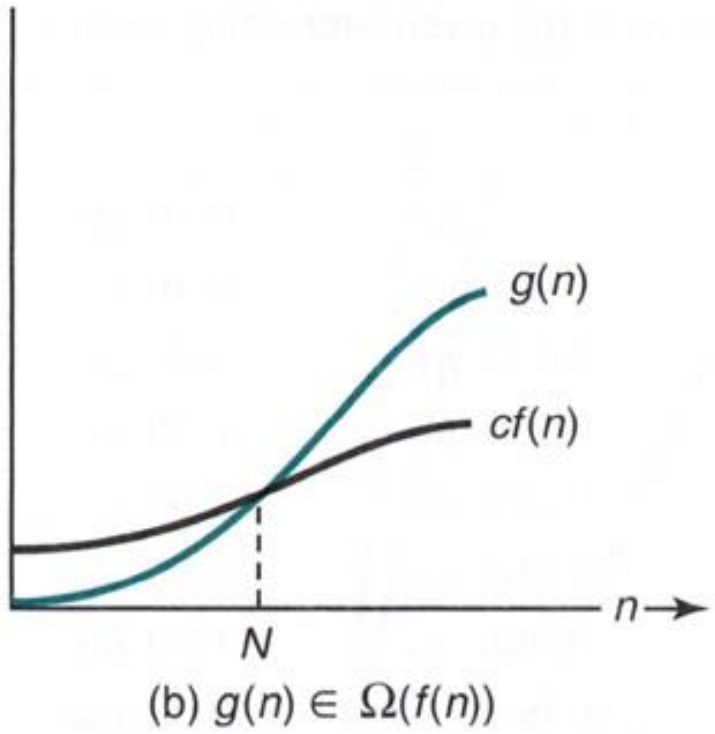
- اگر طرفین نامعادله را بر cn تقسیم کنیم به ازای $n \geq N$ داریم: $\frac{1}{c} \geq n.$

- ولی به ازای $n > 1/c$ این نامعادله نمی تواند برقرار باشد. پس به ازای همه مقادیر $n \geq N$ این نامعادله نمی تواند برقرار باشد. این تناقض نشان می دهد که n در $\Omega(n^2)$ نیست.

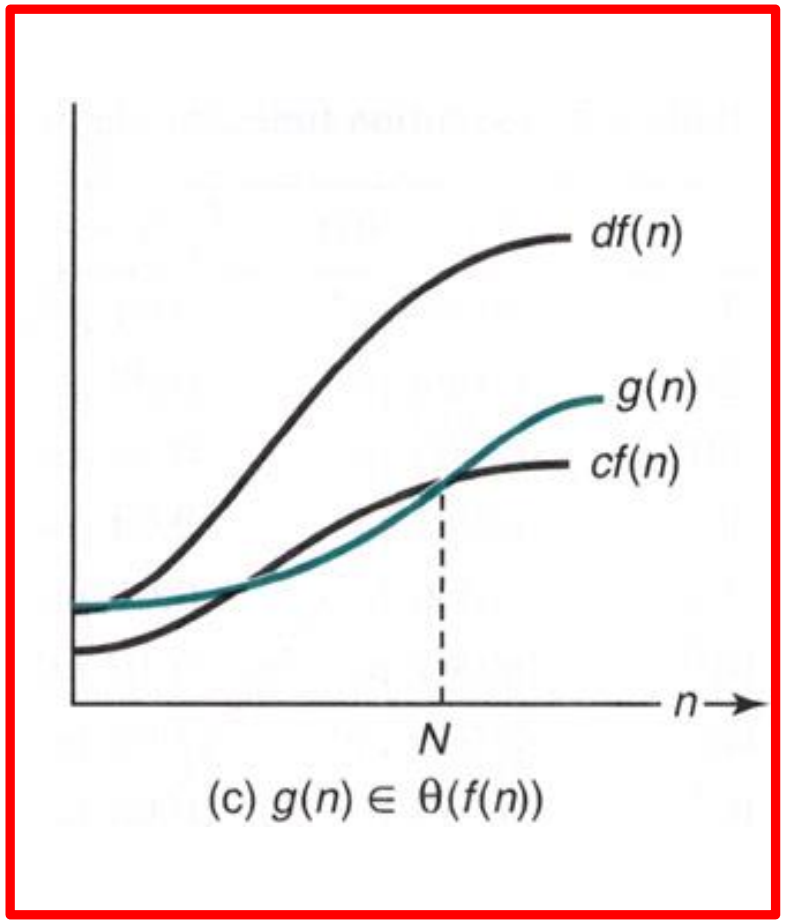
نمایشی از O و Ω و Θ



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

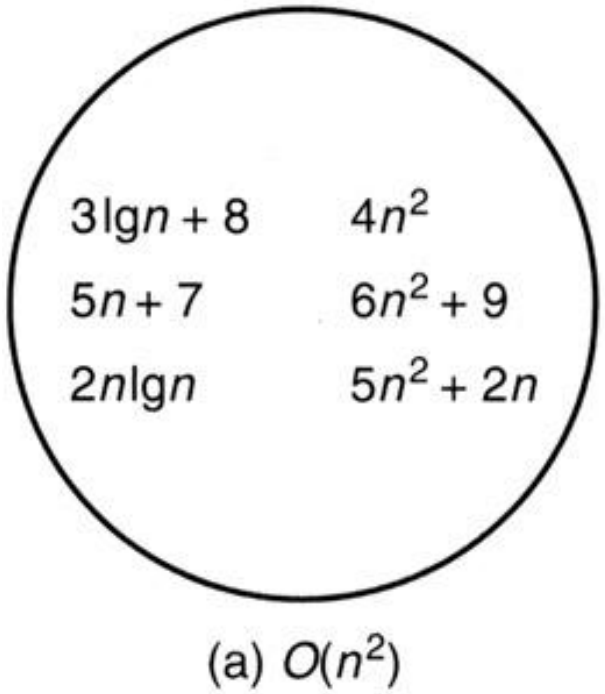


(b) $g(n) \in \Omega(f(n))$

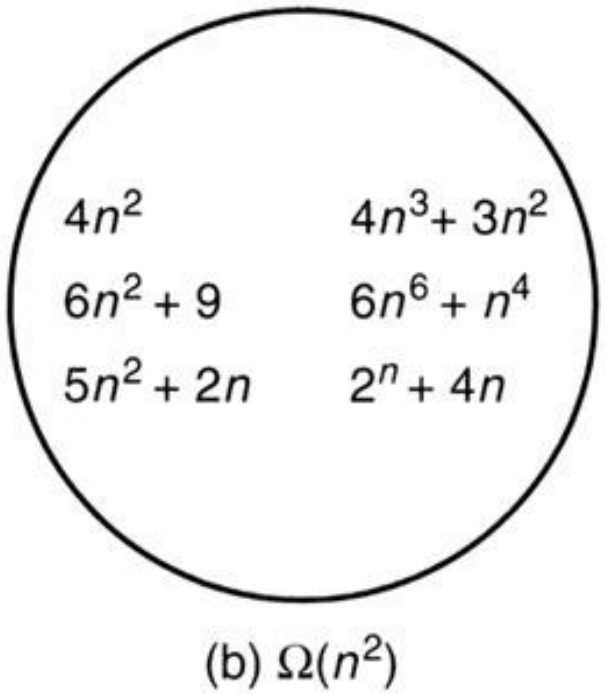


(c) $g(n) \in \theta(f(n))$

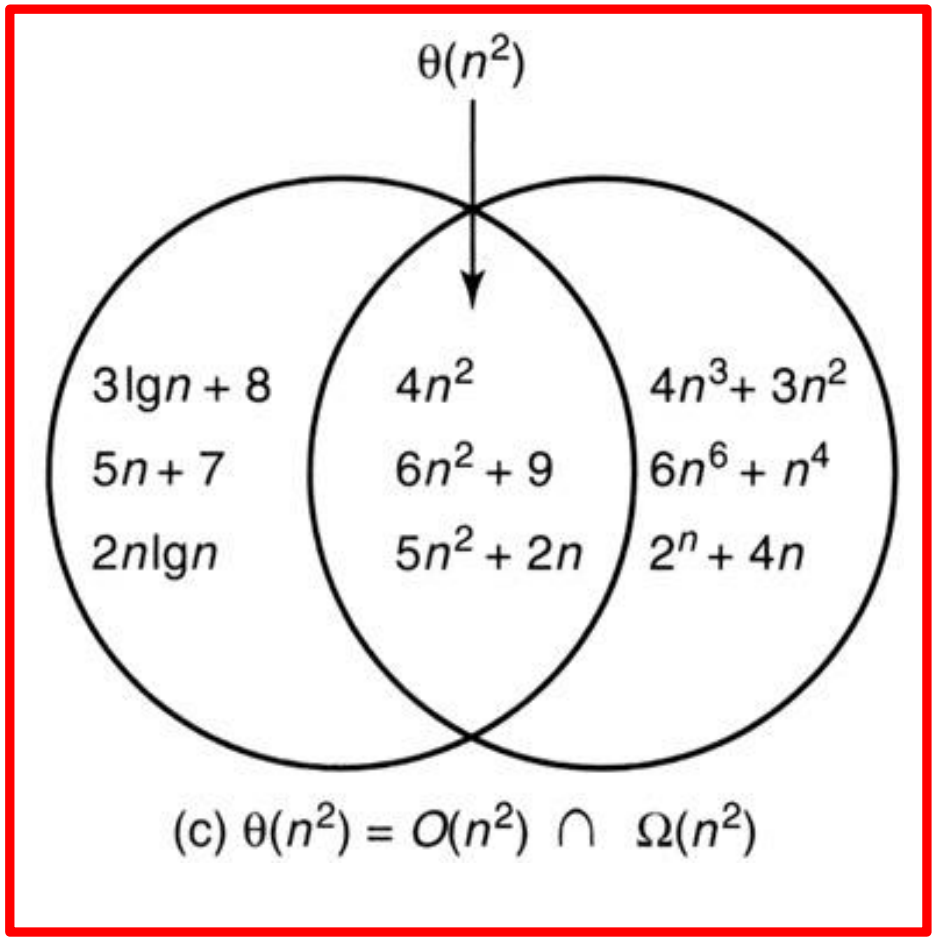
چند عضو نمونه:



(a) $O(n^2)$



(b) $\Omega(n^2)$



(c) $\theta(n^2) = O(n^2) \cap \Omega(n^2)$

نماد o کوچک (o)

- تعریف: برای تابع پیچیدگی f(n)، o(f(n)) مجموعه ای از توابع پیچیدگی g(n) است (می گوییم g(n) ϵ o(n)) که این شرط را برآورده سازند: به ازای هر ثابت حقیقی مثبت c، یک عدد صحیح غیرمنفی N وجود دارد به قسمی که به ازای همه n≥N داریم:

$$g(n) \leq c \times f(n).$$

نماد o کوچک (o)

- این تعریف می گوید که شرط باید برای هر ثابت حقیقی مثبت c برقرار باشد.
- چون این شرط به ازای هر c مثبت برقرار است، برای مقادیر کوچک دلخواه c هم برقرار است.
- برای مثال اگر $g(n) \in o(f(n))$ باشد، یک N وجود دارد به قسمی که به ازای n>N داریم:

$$g(n) \leq 0.00001 \times f(n).$$

- می بینیم که با بزرگ شدن n، g(n) نسبت به f(n) ناچیز می شود.
- برای اهداف تحلیلی، اگر g(n) در o(f(n)) باشد، g(n) درآنها بسیار بهتر از توابعی نظیر f(n) می شود.

مثال

- نشان می دهیم که: $n \in o(n^2).$
- فرض می کنیم c>0 است. باید یک N پیدا کنیم به قسمی که برای n≥N داشته باشیم: $n \leq cn^2.$
- اگر طرفین این نامعادله را بر cn تقسیم کنیم، داریم: $\frac{1}{c} \leq n.$
- بنابراین کافی است هر N≥1/c را انتخاب کنیم.
- توجه شود که مقدار N به ثابت c بستگی دارد. مثلا اگر c=0.00001 باشد، باید N را حداقل مساوی ۱۰۰۰۰۰ در نظر بگیریم. یعنی به ازای n≥100000 داریم: $n \leq 0.00001n^2.$

مثال

- نشان می دهیم که n در $o(5n)$ نیست.
- برهان خلف:
- فرض می کنیم که $n \in o(5n)$ باشد.
- اگر $c=1/6$ باشد، در این صورت باید یک N وجود داشته باشد به نحوی که به ازای $n \geq N$ داشته باشیم:

$$n \leq \frac{1}{6}5n = \frac{5}{6}n.$$

- و این تناقض نشان می دهد که n در $o(5n)$ نیست.

قضیه

- اگر $g(n) \in o(f(n))$ آنگاه: $g(n) \in O(f(n)) - \Omega(f(n))$.

- یعنی g(n) در O(f(n)) است ولی در Ω(f(n)) نیست.

ویژگی های مرتبه

1. $g(n) \in O(f(n))$ if and only if $f(n) \in \Omega(g(n))$.

2. $g(n) \in \Theta(f(n))$ if and only if $f(n) \in \Theta(g(n))$.

3. If $b > 1$ and $a > 1$, then $\log_a n \in \Theta(\log_b n)$.

ویژگی سوم بدان معنا است که همه توابع پیچیدگی لگاریتمی در یک دسته پیچیدگی قرار دارند. این دسته را با $\Theta(\lg n)$ نشان می دهیم.

مثال

- ویژگی سوم بیان می کند که همه توابع پیچیدگی لگاریتمی در یک دسته پیچیدگی قرار دارند. مثلا:

$$\Theta(\log_4 n) = \Theta(\lg n)$$

- یعنی رابطه میان آنها مانند رابطه بین $7n^2+5n$ و n^2 است.

ویژگی های مرتبه

4. If $b > a > 0$, then

$$a^n \in o(b^n).$$

5. For all $a > 0$

$$a^n \in o(n!).$$

- این بدین معنا است که n! از هر تابع با پیچیدگی نمایی بدتر است.

ویژگی های مرتبه

- ۶. ترتیب دسته های پیچیدگی زیر را درنظر بگیرید:

$$\Theta(\lg n) \quad \Theta(n) \quad \Theta(n \lg n) \quad \Theta(n^2) \quad \Theta(n^j) \quad \Theta(n^k) \quad \Theta(a^n) \quad \Theta(b^n) \quad \Theta(n!),$$

- که در آن: $k > j > 2$ and $b > a > 1$

- اگر تابع پیچیدگی g(n) در دسته ای واقع در طرف چپ دسته حاوی f(n) باشد، در آن صورت:

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

مثال

- ویژگی ۶ بیان می کند که هر تابع لگاریتمی در نهایت بهتر از هر تابع چند جمله ای است و هر تابع چند جمله ای در نهایت بهتر از هر تابع نمایی است و هر تابع نمایی در نهایت بهتر از هر تابع فاکتوریل است.

- مثلا:

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

ویژگی های مرتبه

7. If $c \geq 0$, $d > 0$, $g(n) \in O(f(n))$ and $h(n) \in \Theta(f(n))$, then

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

مثال

- ویژگی های ۶ و ۷ را می توان به کرات به کار برد. برای مثال نشان می دهیم که

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

- با استفاده مکرر از ویژگی های ۶ و ۷ داریم: $7n^2 \in \Theta(n^2)$
- یعنی $10n \lg n + 7n^2 \in \Theta(n^2)$
- یعنی $3 \lg n + 10n \lg n + 7n^2 \in \Theta(n^2)$
- یعنی $5n + 3 \lg n + 10n \lg n + 7n^2 \in \Theta(n^2)$

چند نکته

- درعمل، به طور مکرر به این ویژگی ها رجوع نمی کنیم. بلکه می دانیم اجازه حذف جملاتی از مرتبه پایین را داریم.

چند نکته

- اگر بتوانیم نشان دهیم که

$$T(n) \in O(f(n)) \quad \text{and} \quad T(n) \in \Omega(f(n))$$

- می توانیم نتیجه بگیریم که:

$$T(n) \in \Theta(f(n))$$

چند نکته

- عبارت های زیر را می توان به جای هم به کار برد. هر دو یک معنا دارند.

$$f(n) = \Theta(n^2) \qquad \text{instead of} \qquad f(n) \in \Theta(n^2)$$

- و همچنین:

$$f(n) = O(n^2) \qquad \text{instead of} \qquad f(n) \in O(n^2)$$

قضیه: استفاده از حد برای تعیین مرتبه

$$\lim_{n\to\infty}\frac{g(n)}{f(n)}=\begin{cases} c & \text{implies } g(n)\in\Theta(f(n)) \text{ if } c>0 \\ 0 & \text{implies } g(n)\in o(f(n)) \\ \infty & \text{implies } f(n)\in o(g(n)) \end{cases}$$

مثال

- داریم:

$$\frac{n^2}{2} \in o\left(n^3\right)$$

- زیرا:

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

مثال

- به ازای b>a>o داریم:

$$a^n \in o(b^n)$$

- زیرا:

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

- این حد صفر است زیرا: $0 < a/b < 1$
- این همان ویژگی شماره ۴ از ویژگی های مرتبه است.

پایان فصل اول