فصل اول

الگوریتم ها: کارایی، تحلیل و مرتبه

سید ناصر رضوی

E-mail: razavi@Comp.iust.ac.ir

۱۳۸۵

گزیده ای از دعای مکارم الاخلاق
حضرت سجاد (ع)

- خدایا دست های مرا در خدمت نیکی برای مردم قرار بده و خدمتم را با منت نهادن بر آنها ضایع مگردان. حسن خلق عطایم فرما و مرا به رحمت خود از گناه خود بزرگ بینی باز بدار و عمر مرا در راه بندگی و اطاعت خود طولانی بگردان، ای مهربان ترین مهربانان.

الگوریتم ها

- تکنیک های حل مسأله
  - یافتن یک کلمه در دیکشنری
    - جستجوی ترتیبی
    - جستجوی دودویی
- زبان های برنامه نویسی
- الگوریتم: رویه مرحله به مرحله برای حل مساله
- کارآیی: زمان و حافظه

مسائل

- مسأله: سؤالی که ما به دنبال پاسخ آن می باشیم

- مثال ها:

  - A. مرتب سازی یک لیست S متشکل از n عدد به ترتیب غیرنزولی. پاسخ دنباله مرتب از n عدد می باشد.

  - B. تعیین اینکه آیا عدد x در لیست S متشکل از n عدد وجود دارد یا خیر. در صورت وجود x در لیست S پاسخ برابر "بله" و در غیر این صورت پاسخ برابر "خیر" خواهد بود.

پارامترها

- پارامترهای مسأله:
  - A. S و n
  - B. S و n و x
- مسائلی که شامل پارامترها هستند بیانگر کلاسی از مسائل می باشند
  - **نمونه** مسأله: هر انتساب خاصی از مقادیر به پارامترها
  - **راه حل** یک نمونه مسأله: پاسخ سؤال پرسیده شده توسط مسأله در آن نمونه

مثال ها

- مسأله A
  - نمونه: $S = [10, 7, 11, 5, 13, 8]$ و $n = 6$
  - راه حل: $[5, 7, 8, 10, 11, 13]$

- مسأله B
  - نمونه: $S = [10, 7, 11, 5, 13, 8]$ و $n = 6$ و $x = 5$
  - راه حل: «بله، x در S وجود دارد»

الگوریتم

- یک الگوریتم برای مسأله B
  - با شروع از اولین عنصر S، به ترتیب x را با هریک از عناصر S مقایسه کن تا اینکه x را پیدا کنی و یا به آخر لیست S برسی. اگر x پیدا شد، پاسخ «بله» و در غیر این صورت پاسخ «خیر» را تولید کن.

- معایب نوشتن الگوریتم ها به زبان طبیعی:
  - مشکل بودن نوشتن و درک الگوریتم های پیچیده
  - مشکل بودن ترجمه آن به یک زبان برنامه نویسی

- شبه کد ++C

جستجوی ترتیبی

◄ الگوریتم ۱-۱ جستجوی ترتیبی

مسأله: آیا کلید x در آرایه S با n کلید قرار دارد؟

ورودی ها (پارامتر ها): عدد صحیح و مثبت n، آرایه S از کلیدها با اندیس ۱ تا n و کلید x

خروجی ها: $location$، مکان x در S (اگر x در S نباشد خروجی برابر با صفر می باشد)

الگوریتم جستجوی ترتیبی

```
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 ;
}
```

تفاوت های شبه کد با C++

- نحوه استفاده از آرایه ها
  - در C++ فقط مجاز به استفاده از اندیس های صحیح با شروع از صفر هستیم.
  - در شبه کد هر جا که لازم باشد از آرایه هایی استفاده می کنیم که اندیس آنها در بازه دیگری از اعداد صحیح قرار می گیرد و یا اندیس آنها اصلا اعداد صحیح نیستند.
  - در شبه کد طول آرایه ای دو بعدی را هنگام ارسال به زیر برنامه ها متغیر در نظر می گیریم. (مانند الگوریتم ۱–۴)
  - در شبه کد می توانیم آرایه های محلی با طول متغیر تعریف کنیم. مثال:

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

تفاوت های شبه کد با C++

- در شبه کد هر گاه بتوانیم، مراحل را با وضوح بیشتر با استفاده از روابط ریاضی و توضیحات انگلیسی نشان می دهیم.
  - مثال:

$$\text{if}\ \ (\, low \le x \le high \,)\ \{$$

$$\quad \ldots$$

$$\}$$

  - مثال:

$$\text{exchange}\ x\ \text{and}\ y\ ;$$

تفاوت های شبه کد با C++

- در شبه کد از انواع داده ای زیر که در C++ تعریف نشده اند استفاده می کنیم:

نوع داده ای	معنی
index	متغیر صحیحی که به عنوان اندیس به کار می رود.
number	متغیری که می توان آن را به عنوان عدد صحیح (int)و یا حقیقی (real) تعریف کرد
bool	متغیری که می تواند مقادیر true و false را بپذیرد
keytype	متغیری که مقدارش از یک مجموعه مرتب انتخاب می شود

- ساختار کنترلی غیر استاندارد:

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

جمع نمودن عناصر آرایه

◀ الگوریتم ۱–۲ جمع نمودن عناصر آرایه

مسأله: تمام اعداد موجود در آرایه n عنصری S را با هم جمع کنید.

ورودی ها: عدد صحیح و مثبت n، آرایه S با اندیس ۱ تا n.

خروجی ها: sum، حاصل جمع اعداد موجود در S.

الگوریتم جمع نمودن عناصر آرایه

```
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 ;
}
```

مرتب سازی تعویضی

◄ الگوریتم ۱–۳ مرتب سازی تعویضی

مسأله: n کلید را به ترتیب غیر نزولی مرتب کنید.

ورودی ها: عدد صحیح و مثبت n، آرایه S از کلید ها با اندیس ۱ تا n.

خروجی ها: آرایه S حاوی کلیدها به ترتیب غیر نزولی.

الگوریتم مرتب سازی تعویضی

```
void exchangesort ( int n, keytype S [ ] )
{
    index i, j ;

    for ( i = 1; i <= n - 1; i++)
        for (j = i + 1; j <= n; j++)
            if ( S [j] < S [i] )
                exchange S [i] and S [j];
}
```

ضرب ماتریس ها

◄ الگوریتم ۱-۴ ضرب ماتریس ها

مسأله: حاصل ضرب دو ماتریس n x n را تعیین کنید.

ورودی ها: عدد صحیح ومثبت n، آرایه های دو بعدی A و B که هر یک دارای سطرها و ستون هایی با اندیس ۱ تا n می باشند.

خروجی: آرایه دو بعدی C از اعداد، که سطرها و ستون های آن از ۱ تا n شماره گذاری شده است و حاوی حاصل ضرب A و B می باشد.

الگوریتم ضرب ماتریس ها

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

اهمیت توسعه الگوریتم های کارآ

- کارآیی الگوریتم ها همواره و بدون در نظر گرفتن افزایش سرعت کامپیوترها و کاهش قیمت حافظه، باید مد نظر باشد.

- اجازه دهید این موضوع را با مقایسه جستجوی ترتیبی و جستجوی دودویی بیشتر بررسی نماییم.

جستجوی دودویی

◄ الگوریتم ۱–۵ جستجوی دودویی

مسأله: تعیین کنید آیا x در آرایه مرتب S با n کلید قرار دارد یا خیر.

ورودی ها: عدد صحیح و مثبت n، آرایه مرتب (غیر نزولی) S با اندیس ۱ تا n و کلید x.

خروجی ها: $location$، مکان x در S (اگر x در S نباشد خروجی برابر با صفر می باشد).

الگوریتم جستجوی دودویی

```
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 ;
   }
}
```

تعداد مقایسه های انجام گرفته توسط هر دو الگوریتم

- فرض کنید $n = 32$
  - جستجوی ترتیبی: ۳۲
  - جستجوی دودویی: ۶

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

شکل ۱-۱ عناصری از آرایه که اگر x از تمام عناصر آرایه ای با اندازه ۳۲ بزرگتر باشد، جستجوی دودویی آنها را با x مقایسه می کند. عناصر برحسب ترتیب مقایسه شماره گذاری شده اند.

تعداد مقایسه های انجام گرفته توسط هر دو الگوریتم

- به طور کلی
  - جستجوی ترتیبی: n
  - جستجوی دودویی (اگر n توانی از ۲ باشد): $\lg n + 1$

● **Table 1.1** The number of comparisons done by Sequential Search and Binary Search when x is larger than all the array items

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

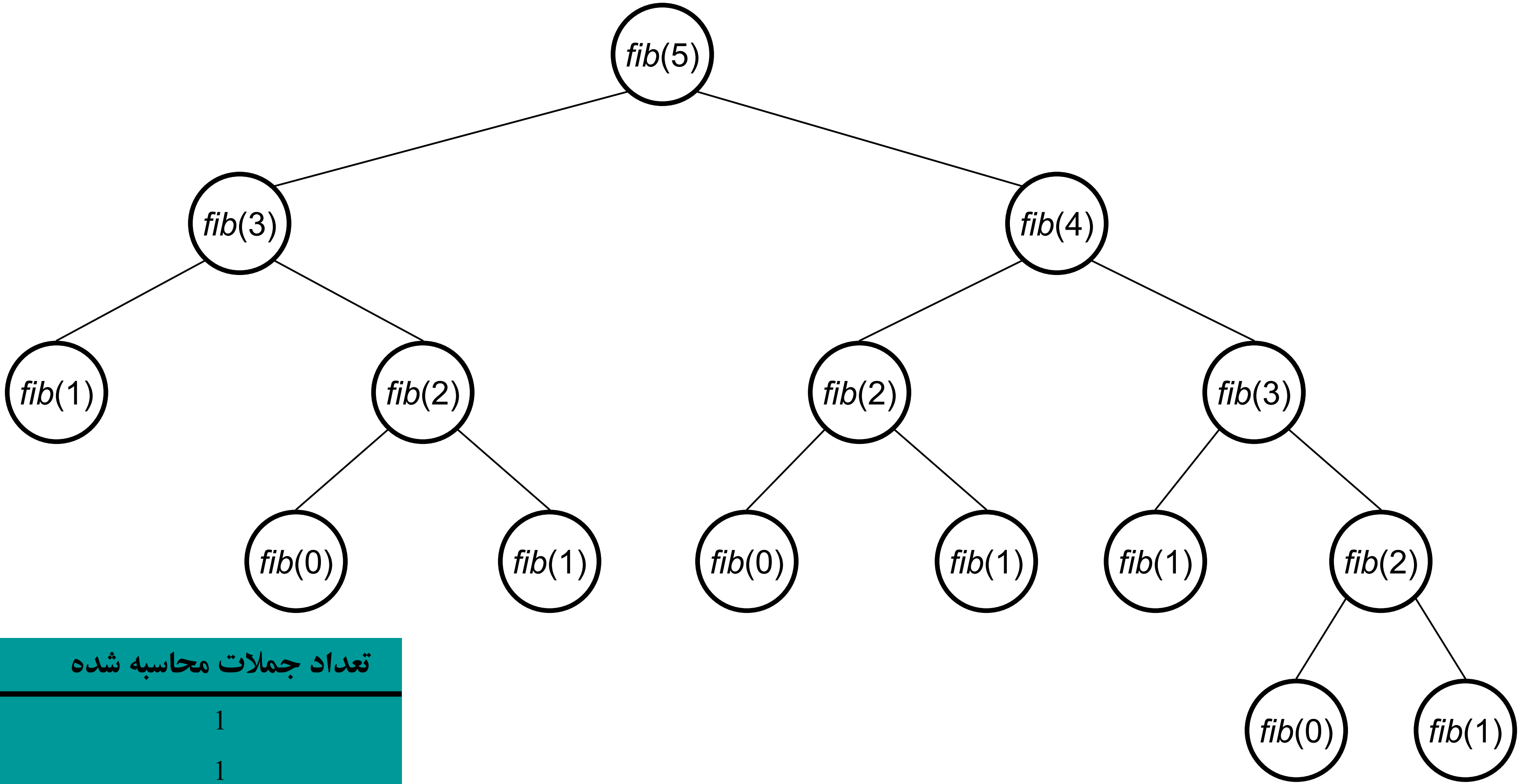
◄ الگوریتم ۱–۶ جمله nاُم فیبونانچی

مسأله: جمله nاُم از دنباله فیبوناچی را تعیین کنید.
ورودی ها: یک عدد صحیح و غیر منفی n.
خروجی ها: fib، جمله nاُم از دنباله فیبوناچی.

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

تقسیم و حل

عدم کارآیی



n	تعداد جملات محاسبه شده
0	1
1	1
2	3
3	5
4	9
5	15
6	25

- علت ناکارآیی: محاسبات تکراری
- مثلا در این مثال $fib(2)$ سه بار محاسبه شده است.

تعداد جملات محاسبه شده

- هر بار که n به اندازه ۲ واحد افزایش می یابد، تعداد جملات محاسبه شده بیش از ۲ برابر افزایش می یابد، یعنی:
  - $T(n) > 2 * T(n - 2) > 2^{n/2}$ when $n \geq 2$
  - $$\begin{aligned} T(n) &> 2 * T(n - 2) \\ &> 2 * 2 * T(n - 4) \\ &> 2 * 2 * 2 * T(n - 6) \\ &\ldots \\ &> \underbrace{2 * 2 * \ldots * 2}_{n/2 \text{ بار}} * T(0) = 2^{n/2} \end{aligned}$$
- اثبات بوسیله استقراء

اثبات بوسیله استقراء

◄ قضیه ۱-۱

- **پایه استقراء:**

$$T(2) = 3 > 2 = 2^{2/2}$$
$$\mathrm{T}(3) = 5 > 2.83 \approx 2^{3/2}$$

- **فرض استقراء:**

$$T(m) > 2^{m/2} \qquad , 2 \leq m < n$$

- **گام استقراء:**

$$T(n) = T(n-1) + T(n-2) + 1$$
$$> 2^{(n-1)/2} + 2^{(n-2)/2} + 1$$

طبق فرض استقراء

$$> 2^{(n-2)/2} + 2^{(n-2)/2} = 2 * 2^{(n-2)/2} = 2^{n/2}$$

دنباله فیبوناچی

◄ الگوریتم ۱–۷ جمله n اُم فیبوناچی (تکراری)

مسأله: جمله n اُم از دنباله فیبوناچی را تعیین کنید.

ورودی ها: یک عدد صحیح و غیر منفی n.

خروجی ها: $fib2$، جمله n اُم از دنباله فیبوناچی.

الگوریتم محاسبه جمله n اُم دنباله فیبوناچی (تکراری)

```
int fib2 ( int n)
{
    int f [0 . . n] ;
    f [0] = 0 ;
    if ( n > 0) {
        f [1] = 1 ;
        for ( i = 2; i <= n; i++)
            f [i] = f [i -1] + f [i – 2] ;
    }
    return f [n] ;
}
```

برنامه نویسی پویا

مقایسه دو الگوریتم

- Table 1.2 A comparison of Algorithms 1.6 and 1.7

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 μs†
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.

†1 μs = 10^{-6} second.

تحلیل الگوریتم ها

- هدف از تحلیل الگوریتم ها:
  - بررسی رفتار الگوریتم از نظر زمان اجرا و مقدار حافظه مصرفی قبل از پیاده سازی
  - مقایسه الگوریتم ها از نظر کارآیی
- عوامل موثر در زمان اجرای یک برنامه :
  - سرعت سخت افزار
  - نوع کامپایلر (بهینگی کد مقصد)
  - برنامه نویس (بهینگی کد منبع)

  تاثیر ثابت دارند

  - اندازه ورودی
  - ترکیب داده های ورودی
  - پیچیدگی الگوریتم
  - پارامترهای دیگری که تاثیر ثابت دارند.
- مهمترین عامل، پیچیدگی الگوریتم می باشد که خود تابعی از اندازه ورودی می باشد. ترکیب داده های ورودی را نیز می توان با محاسبه پیچیدگی در حالت های مختلف در نظر گرفت.

تحلیل الگوریتم ها

- تحلیل پیچیدگی
  - محاسبه کارآیی بر حسب زمان
  - مستقل از کامپیوتر، زبان برنامه نویسی، برنامه نویس و تمامی جزئیات الگوریتم
  - محاسبه تعداد دفعات اجرای عملیات اصلی (مقایسه، جمع، ضرب و ...) بر حسب اندازه ورودی
- برای تحلیل پیچیدگی یک الگوریتم تابعی به نام $T(n)$ در نظر می گیریم که در آن n اندازه ورودی می باشد.

اندازه ورودی

- در بسیاری از الگوریتم ها یافتن میزانی منطقی از اندازه ورودی آسان است، مثال:
  - الگوریتم ۱-۱ (جستجوی ترتیبی)
  - الگوریتم ۱-۲ (محاسبه مجموع عناصر آرایه)
  - الگوریتم ۱-۳ (مرتب سازی تعویضی)
  - الگوریتم ۱-۵ (جستجوی دودویی)

  ← اندازه ورودی: n، تعداد عناصر آرایه

  - الگوریتم ۱-۴ (ضرب ماتریس ها) ← اندازه ورودی: n، تعداد سطر ها و ستون ها

- در برخی از الگوریتم ها بهتر است اندازه ورودی را برحسب دو عدد بسنجیم:
  - اگر ورودی الگوریتم یک گراف باشد، اندازه وروی را برحسب تعداد رئوس (n) و تعداد یال ها (m) می سنجیم

اندازه ورودی

- در مواردی باید در تعیین اندازه ورودی احتیاط نمود. مثال:
  - الگوریتم ۱–۶ (جمله n اُم فیبونانچی، بازگشتی)
  - الگوریتم ۱–۷ (جمله n اُم فیبونانچی، تکراری)
- در این دو الگوریتم n اندازه ورودی نمی باشد، بلکه n خود ورودی است.
- در این دو الگوریتم، تعداد نماد های بکار رفته برای کد کردن n میزانی منطقی از اندازه ورودی می باشد.
- اگر از نمایش دودویی استفاده کنیم:
  - اندازه ورودی = تعداد بیت های لازم برای کد کردن n که برابر است با $\lfloor \lg n \rfloor + 1$

عملیات اصلی

- عمل اصلی: دستور یا مجموعه ای از دستورات به طوری که کل کار انجام شده توسط الگوریتم، تقریبا متناسب با تعداد دفعات اجرای این دستور یا مجموعه دستورات باشد.

- مثال: الگوریتم ۱-۱ (جستجوی ترتیبی) و الگوریتم ۱-۵ (جستجوی دودویی)
  - در هر باز گذر از حلقه عنصر x با یک عنصر از S مقایسه می شود.
  - با تعیین اینکه هر یک از این الگوریتم ها چند بار این عمل اصلی را به ازای هر مقدار از n انجام می دهند، می توان کارآیی این دو الگوریتم را مقایسه نمود.

عملیات اصلی

- تحلیل پیچیدگی زمانی:
  - تعیین تعداد دفعاتی که عمل اصلی به ازای هر مقدار از اندازه ورودی انجام می شود.
- انتخاب عمل اصلی بیشتر بر اساس تجربه و داوری انجام می شود.
- در برخی موارد ممکن است بخواهیم دو عمل اصلی متفاوت را در نظر بگیریم.
  - مرتب سازی: مقایسه و انتساب، هر یک به تنهایی می توانند عمل اصلی باشند.

تحلیل پیچیدگی زمانی در حالات مختلف

- در برخی موارد مانند الگوریتم ۱–۲ (جمع نمودن عناصر آرایه)، عمل اصلی همواره به ازای یک نمونه n به یک میزان انجام می شود.
در چنین مواردی:

$T(n) =$ تعداد دفعاتی که الگوریتم عمل اصلی را بازاء یک نمونه n انجام می دهد

- در برخی موارد دیگر، تعداد دفعات اجرای عمل اصلی نه تنها به اندازه ورودی بلکه به مقادیر ورودی نیز بستگی دارد
  - مثال: جستجوی ترتیبی (با اندازه ورودی برابر n)
    - اگر x در اولین مکان آرایه باشد: تعداد مقایسه ها = ۱
    - اگر x در آرایه نباشد: تعداد مقایسه ها = n

پیچیدگی زمانی در هر حالت $T(n)$

- عمل اصلی همواره به ازای هر اندازه نمونه n، به تعداد دفعات یکسانی انجام می گیرد.

- مطالعات موردی:
  - محاسبه مجموع عناصر آرایه (الگوریتم ۱-۲)
  - مرتب سازی تعویضی (الگوریتم ۱-۳)
  - ضرب ماتریس ها (الگوریتم ۱-۴)

تحلیل پیچیدگی زمانی در هر حالت برای الگوریتم ۱-۲ (جمع نمودن عناصر آرایه)

- عمل اصلی: افزودن یک عنصر از آرایه به sum
- اندازه ورودی: n، تعداد عناصر آرایه
- تحلیل پیچیدگی:

$$T(n) = n$$

زیرا مقادیر آرایه هر چه باشند، n بار گذر از حلقه for داریم و بنابراین عمل اصلی n بار انجام می شود.

تحلیل پیچیدگی زمانی در هر حالت برای الگوریتم ۱-۳ (مرتب سازی تعویضی)

- عمل اصلی: مقایسه $S[j]$ و $S[i]$
- اندازه ورودی: n، تعداد عناصر آرایه که باید مرتب شوند.
- تحلیل پیچیدگی:

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1} n - i$$

$$= (n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2}$$

تحلیل پیچیدگی زمانی در هر حالت برای الگوریتم ۱-۱۴ (ضرب ماتریس ها)

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

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} 1 = \sum_{i=1}^{n}\sum_{j=1}^{n} n = \sum_{i=1}^{n} n^2 = n^3$$

پیچیدگی زمانی در بدترین حالت $W(n)$

- حداکثر تعداد دفعاتی که الگوریتم عمل اصلی اش را به ازای یک ورودی با اندازه n انجام می دهد.

- اگر $T(n)$ وجود داشته باشد، آنگاه $W(n) = T(n)$

- مطالعه موردی:
  - جستجوی ترتیبی (الگوریتم ۱-۱)

$$W(n) = n$$

پیچیدگی زمانی در حالت متوسط $A(n)$

- متوسط تعداد دفعاتی که الگوریتم عمل اصلی اش را به ازای یک ورودی با اندازه n انجام می دهد.

- اگر $T(n)$ وجود داشته باشد، آنگاه $A(n) = T(n)$

- انتساب احتمالات به تمام ورودی های ممکن با اندازه n مهم است.
- مطالعه موردی:
  - جستجوی ترتیبی (الگوریتم ۱-۱)

تحلیل پیچیدگی زمانی در حالت متوسط برای الگوریتم ۱-۱ (جستجوی ترتیبی)

- عمل اصلی: مقایسه یک عنصر آرایه با x
- اندازه ورودی: n، تعداد عناصر آرایه
- تحلیل پیچید گی:
- **حالت ۱)** فرض می کنیم که x در S وجود دارد
  - عناصر S همگی متمایز می باشند
  - احتمال وجود x در همه مکانهای آرایه یکسان است $(1/n)$

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

تحلیل پیچیدگی زمانی در حالت متوسط برای الگوریتم ۱-۱ (جستجوی ترتیبی)

- **حالت ۲)** ممکن است که x در S نباشد
  - فرض می کنیم که احتمال وجود x در S برابر p باشد
  - عناصر S همگی متمایز می باشند
  - به شرط وجود x در S، احتمال وجود x در همه مکانهای آرایه یکسان است (p/n)

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

پیچیدگی زمانی در بهترین حالت $B(n)$

- حداقل تعداد دفعاتی که الگوریتم عمل اصلی اش را بازاء یک ورودی با اندازه n انجام می دهد.

- اگر $T(n)$ وجود داشته باشد، آنگاه $B(n) = T(n)$

- مطالعه موردی:
  - جستجوی ترتیبی (الگوریتم ۱-۱)

$$B(n) = 1$$

مقایسه

- $A(n)$: زمانی مفید است که الگوریتم بر روی ورودی های متفاوت بسیاری و به دفعات زیاد اعمال می شود.

- $W(n)$: زمانی مفید است که حد بالای زمان مصرفی توسط الگوریتم مهم باشد.

- $B(n)$: معمولا اهمیت چندانی ندارد.

تابع پیچیدگی

- پیچیدگی حافظه
- تابع پیچیدگی: تابعی است که اعداد صحیح مثبت را به اعداد حقیقی غیر منفی نگاشت می کند، مانند:
  - $f(n) = n$
  - $f(n) = n^2$
  - $f(n) = \lg n$
  - $f(n) = 3n^2 + 4n$

اعمال تئوری

- زمان اجرا توسط عوامل زیر تعیین می شود:
  - عمل اصلی
  - دستورالعمل های سربار: تعداد دفعات اجرای این دستورالعمل ها با افزایش اندازه ورودی، افزایش نمی یابد. (قابل چشم پوشی)
  - دستورالعمل های کنترلی: تعداد دفعات اجرای این دستورالعمل ها با افزایش اندازه ورودی، افزایش می یابد.

- ضریب در تابع پیچیدگی زمانی می تواند مهم باشد.
  - مثال: دو الگوریتم برای یک مساله وجود دارد که اولی عمل اصلی را n بار و دومی عمل اصلی را n^2 بار انجام می دهد. اگر زمان انجام عمل اصلی در اولی ۱۰۰۰ برابر زمان انجام عمل اصلی در دومی باشد؛ آنگاه

$$n^2 * t > n * 1000t \rightarrow n > 1000$$

  - یعنی فقط بازاء n های بزرگتر از ۱۰۰۰ الگوریتم اولی بهتر از دومی می باشد.

مرتبه

- الگوریتم های زمانی خطی : مانند n و $100n$
- الگوریتم های زمانی درجه دوم: مانند n^2 و $0.01n^2$
- رفتار نهایی (وقتی n به اندازه کافی بزرگ می باشد)، مانند مقایسه $0.01n^2$ و $100n$

$$0.01n^2 > 100n \rightarrow n > 10,000$$

معرفی شهودی مرتبه

- دسته بندی توابع پیچیدگی، مانند:
  - توابع درجه دوم محض : $5n^2$ و $5n^2 + 100$
  - توابع درجه دوم کامل: $0.1n^2 + n + 100$

Table 1.3 The quadratic term eventually dominates

n	$0.1n^2$	$0.1n^2 + n + 100$
10	10	120
20	40	160
50	250	400
100	1,000	1,200
1,000	100,000	101,100

  - الگوریتم زمانی درجه دوم – $\Theta(n^2)$
  - دسته های پیچیدگی: $\Theta(\lg n), \Theta(n), \Theta(n\lg n), \Theta(n^2), \Theta(n^3), \Theta(2^n)$

مقایسه

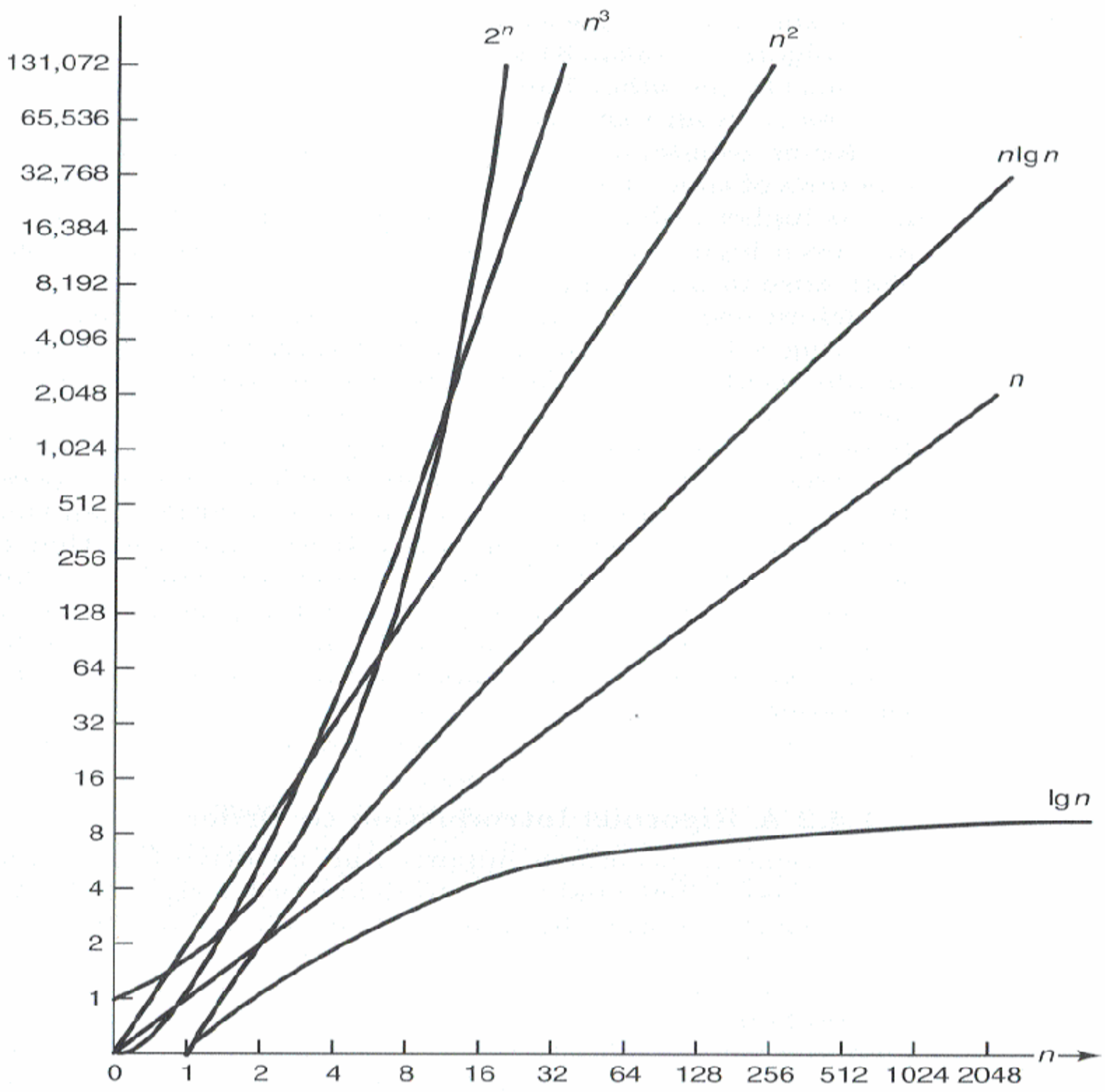


Figure 1.3 • Growth rates of some common complexity functions.

مقایسه (ادامه)

Table 1.4 Execution times for algorithms with the given time complexities

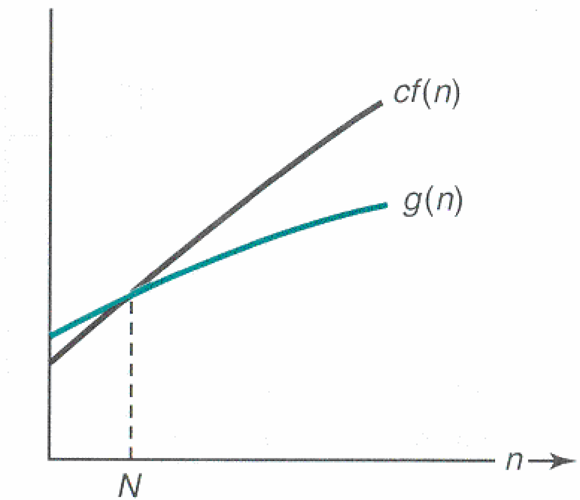
n	$f(n) = lg\, n$	$f(n) = n$	$f(n) = n\, lg\, n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.003 μs*	0.01 μs	0.033 μs	0.10 μs	1.0 μs	1 μs
20	0.004 μs	0.02 μs	0.086 μs	0.40 μs	8.0 μs	1 ms†
30	0.005 μs	0.03 μs	0.147 μs	0.90 μs	27.0 μs	1 s
40	0.005 μs	0.04 μs	0.213 μs	1.60 μs	64.0 μs	18.3 min
50	0.006 μs	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 μs $= 10^{-6}$ second.

†1 ms $= 10^{-3}$ second.

معرفی مرتبه

- **اُی بزرگ (Big O):**
  - **$O(f(n))$** – شامل مجموعه ای از توابع پیچیدگی مانند $g(n)$ می باشد که برای هریک از آنها حداقل یک ثابت حقیقی مثبت مانند c و یک عدد صحیح غیر منفی مانند N وجود دارد به طوری که برای هر $n \geq N$ داشته باشیم، $g(n) \leq c \times f(n)$.



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

معرفی مرتبه

- اگر $g(n) \in O(f(n))$، می گوییم $g(n)$ اُی بزرگ $f(n)$ است.
  مانند: $n^2 + 10n \in O(n^2)$

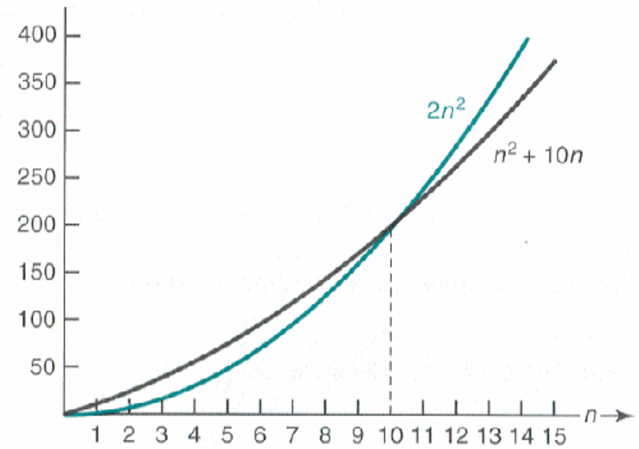


Figure 1.5 • The function $n^2 + 10n$ eventually stays beneath the function $2n^2$.

$$n^2 + 10n \le n^2 + 10n^2 = 11n^2$$

$$\Rightarrow N = 1, c = 11$$

بنابراین N و c منحصر بفرد نمی باشند

- اُی بزرگ یک **حد بالای مجانبی** بر روی یک تابع قرار می دهد.
- معنای شهودی اُی بزرگ: $g(n)$ حداقل به خوبی $f(n)$ می باشد.

اثبات ...

- $5n^2 \in O(n^2)$

  $5n^2 \leq 5n^2 \;\Rightarrow\; N = 0,\ c = 5$

- $T(n) = n(n\text{-}1)/2 \in O(n^2)$

  $n(n\text{-}1)/2 \leq n(n)/2 = (1/2)n^2$

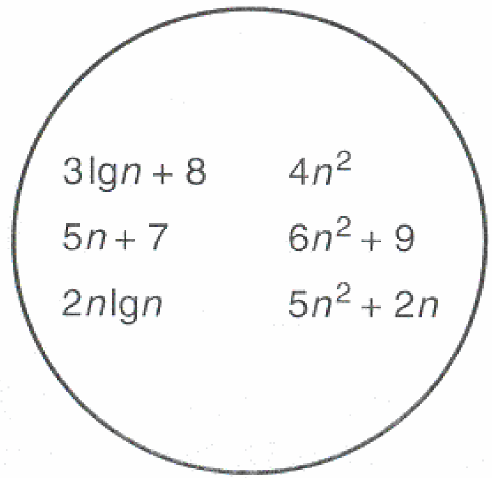
  $\Rightarrow N = 0,\ c = 1/2$

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

  $N = 10,\ c = 1$

- $n \in O(n^2)$

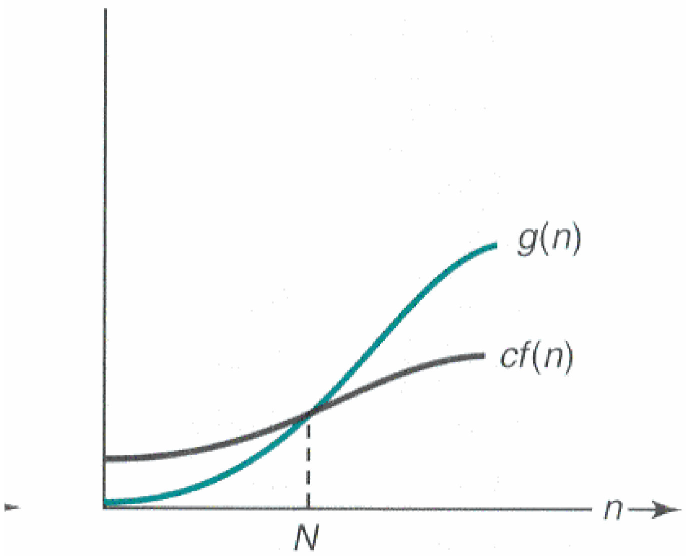
  $N = 1,\ c = 1$



(a) $O(n^2)$

اُمگای $f(n)$

– $\Omega(f(n))$ – شامل مجموعه ای از توابع پیچیدگی مانند $g(n)$ می باشد که برای هریک از آنها حداقل یک ثابت حقیقی مثبت مانند c و یک عدد صحیح غیر منفی مانند N وجود دارد به طوری که برای هر $n \geq N$ داشته باشیم، $g(n) \geq c \times f(n)$.



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

- اگر $g(n) \in \Omega(f(n))$ ، می گوییم که $g(n)$ از اُمگای $f(n)$ می باشد.

-اُمگا یک **حد پایین مجانبی** بر روی یک تابع قرار می دهد.

-تعریف شهودی: $g(n)$ حداقل به بدی $f(n)$ است.

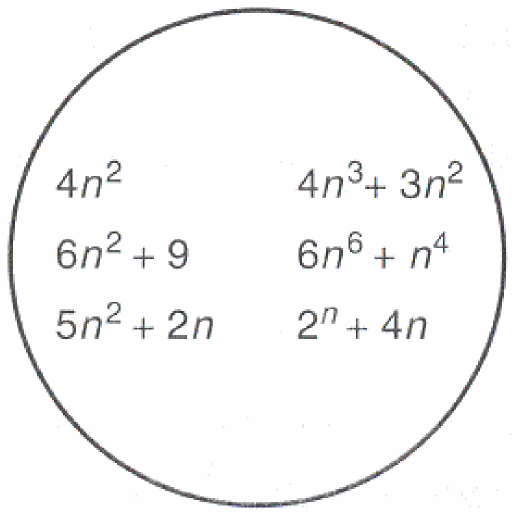
اثبات ...

- $5n^2 \in \Omega(n^2)$
  $5n^2 \geq 1 \times n^2 \;\Rightarrow\; N = 0,\; c = 1$

- $n^2 + 10n \in \Omega(n^2)$
  $n^2 + 10n \geq n^2 \;\Rightarrow\; N = 0,\; c = 1$

- $T(n) = n(n\text{-}1)/2 \in \Omega(n^2)$
  $n \geq 2 \Rightarrow n - 1 \geq n/2 \Rightarrow$
  $n(n\text{-}1)/2 \geq (n/2)(n/2) = n^2/4$
  $\Rightarrow N = 2,\; c = 1/4$

- $n^3 \in \Omega(n^2)$

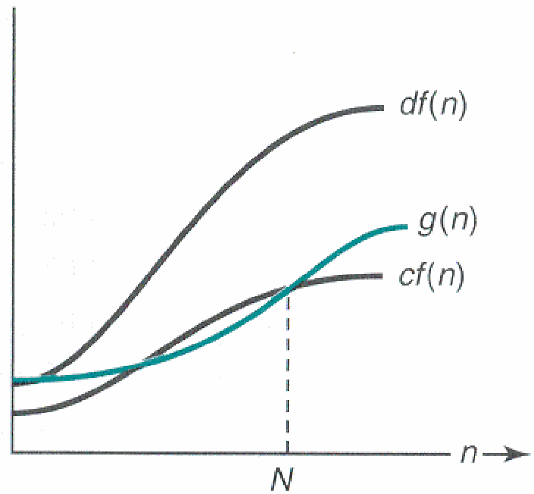


(b) $\Omega(n^2)$

مرتبه $f(n)$

– $\boldsymbol{\Theta(f(n))}$ – شامل مجموعه ای از توابع پیچیدگی مانند $g(n)$ می باشد که برای هریک از آنها ثابت های حقیقی مثبت مانند c و d و یک عدد صحیح غیر منفی مانند N وجود دارد به طوری که برای هر $n \geq N$ داشته باشیم:

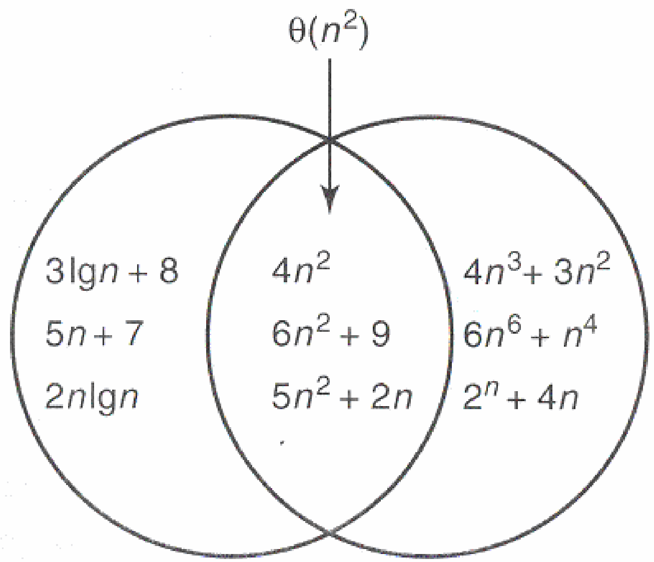
$$c \times f(n) \leq g(n) \leq d \times f(n)$$



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

مرتبه $f(n)$

- $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$
- اگر $g(n) \in \Theta(f(n))$ ، می گوییم $g(n)$ از مرتبه (دقیق) $f(n)$ می باشد.



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

اثبات ...

- (Exchange sort) $T(n) = n(n\text{-}1)/2 \in \Theta(n^2)$

$$T(n) \in O(n^2)\ ,\ \mathrm{T}(n) \in \Omega(n^2)$$
$$\Rightarrow T(n) \in O(n^2) \cap \Omega(n^2)$$
$$\Rightarrow T(n) \in \Theta(n^2)$$

اُی کوچک $f(n)$

- **اُی کوچک (Small o):**
  - $o(f(n))$ – شامل مجموعه ای از توابع پیچیدگی مانند $g(n)$ می باشد که برای هریک از آنها به ازای هر ثابت حقیقی مثبت مانند c، حداقل یک عدد صحیح غیر منفی مانند N وجود دارد به طوری که برای هر $n \geq N$ داشته باشیم، $g(n) \leq c \times f(n)$.

- اگر $g(n) \in o(f(n))$ آنگاه می گوییم که $g(n)$ اُی کوچک $f(n)$ می باشد.
- تعریف شهودی: $g(n)$ در نهایت بسیار بهتر از $f(n)$ می باشد.

اثبات ...

- $n \in o(n^2)$

فرض می کنیم $c > 0$ است. باید یک n بیایم که برای هر $n \geq N$ داشته باشیم:

$$n \leq cn^2 \Rightarrow 1/c \leq n$$

بنابراین کافی است که هر $N \geq 1/c$ را انتخاب کنیم. مثلا اگر $c = 0.01$ باشد، باید N بزرگتر مساوی ۱۰۰ باشد.

- $n \notin o(5n)$ (proof by contradiction)

فرض می کنیم $c = 1/6$ باشد. اگر $n \in o(5n)$ باشد، آنگاه باید یک N وجود داشته باشد که برای هر $n \geq N$ داشته باشیم:

$$n \leq (1/6)5n = (5/6)n$$

و این تناقض ثابت می کند که $n \notin o(5n)$

دسته های پیچیدگی

- $g(n) \in o(f(n)) \Rightarrow g(n) \in O(f(n)) - \Omega(f(n))$
- $g(n) \in \Theta(f(n)) \Leftrightarrow f(n) \in \Theta(g(n))$

- Θ توابع پیچیدگی را به مجموعه های مجزا (دسته های پیچیدگی) تقسیم می کند.

- ما اغلب از ساده ترین تابع در یک دسته پیچیدگی برای نمایش بقیه توابع موجود در آن دسته پیچیدگی استفاده می کنیم، مثلا: $\Theta(n^2)$، $\Theta(1)$ و ...

خصوصیات مرتبه

- $g(n) \in O(f(n)) \Leftrightarrow f(n) \in \Omega(g(n))$
- $g(n) \in \Theta(f(n)) \Leftrightarrow f(n) \in \Theta(g(n))$

- اگر $a > 1$ و $b > 1$، آنگاه $\log_a n \in \Theta(\log_b n)$ – یعنی تمامی توابع لگاریتمی در یک دسته پیچیدگی قرار می گیرند.

- اگر $b > a > 0$، آنگاه $a^n \in o(b^n)$ – یعنی توابع نمایی در یک دسته پیچیدگی قرار ندارند.

خصوصیات مرتبه

- برای هر $a > 0$، داریم $a^n \in o(n!)$ – یعنی $n!$ از هر تابع نمایی بدتر است.

- ترتیب زیر را از دسته های پیچیدگی مختلف در نظر بگیرید:

$$\Theta(1),\ \Theta(\lg n),\ \Theta(n),\ \Theta(n\lg n),\ \Theta(n^2),\ \Theta(n^j),\ \Theta(n^k),\ \Theta(a^n),\ \Theta(b^n),\ \Theta(n!),\ \Theta(n^n)$$

که در آن $k > j > 2$ و $b > a > 1$.

اگر $g(n)$ در یک دسته پیچیدگی واقع در سمت چپ دسته پیچیدگی $f(n)$ باشد، آنگاه $g(n) \in o(f(n))$.

خصوصیات مرتبه

- اگر $c \geq 0$، $d > 0$، $g(n) \in O(f(n))$ و $h(n) \in \Theta(f(n))$ آنگاه :

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

استفاده از خصوصیات مرتبه

- تمام توابع لگاریتمی در یک دسته پیچیدگی قرار می گیرند. (ویژگی ۳)

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

- هر تابع لگاریتمی در نهایت بهتر از هر تابع چند جمله ای، هر تابع چند جمله ای در نهایت بهتر از هر تابع نمایی و هر تابع نمایی در نهایت بهتر از هر تابع فاکتوریل می باشد.

$$\lg n \in o(n) \qquad n^{10} \in o(2^n) \qquad 2^n \in o(n!)$$

- با اعمال دو ویژگی آخر به طور مکرر داریم:

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

یعنی در تعیین مرتبه، همواره اجازه حذف جملاتی از مرتبه پایین را داریم.

استفاده از حد برای تعیین مرتبه

◄ قضیه ۱–۳

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \begin{cases} c & g(n) \in \theta(f(n)), c > 0 \\ 0 & g(n) \in o(f(n)) \\ \infty & f(n) \in o(g(n)) \end{cases}$$

تمرین ها

- بخش ۱–۱
  - ۳، ۶، ۷
- بخش ۱–۳
  - ۱۰، ۱۲، ۱۴
- بخش ۱–۴
  - ۱۹، ۲۱، ۲۲، ۲۴
- تمرین های اضافی
  - ۲۶، ۲۸، ۲۹، ۳۱، ۳۲، ۳۳، ۳۴، ۳۵