طراحی الگوریتم

محمدجواد فدائی اسلام

سرفصل

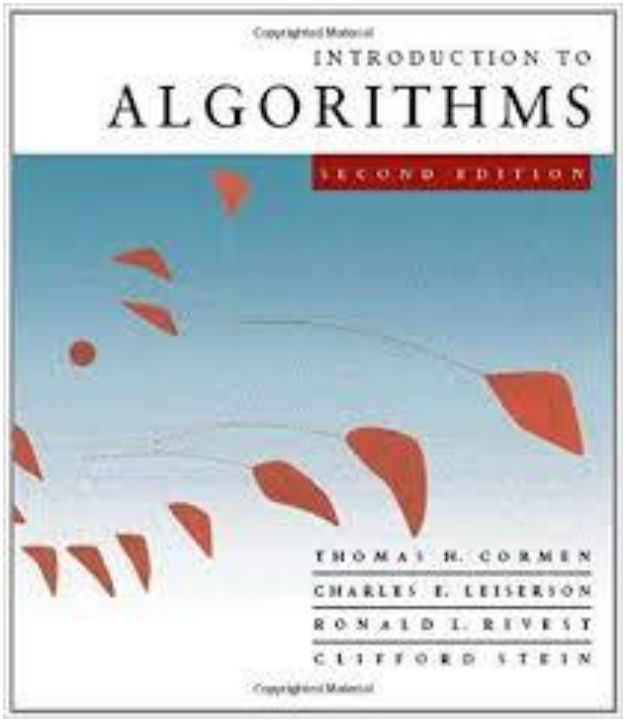
- Algorithms, efficiency, analysis and order
- Divide & Conquer
- Dynamic Programming
- Greedy Approach
- Backtracking

مطالعه بیشتر و ارایه توسط دانشجویان

- **Amortized Analysis**
- **Advanced Data Structures**
  - **Binomial Heap**
  - **Fibonacci Heap**
  - **Disjoint Sets**
- **Network Flow**

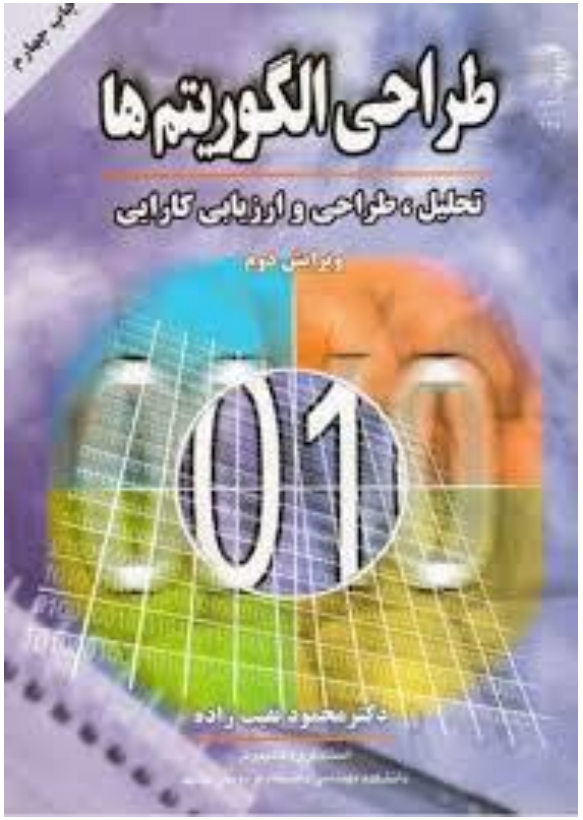
مراجع اصلی

- Introduction to Algorithms, (CLRS)
- Foundations of Algorithms Using C++ Pseudocode



سایر مراجع

- کتاب طراحی الگوریتم دکتر محمود نقیب‌زاده
- کتاب طراحی الگوریتم دکتر بهروز قلی‌زاده
- و هر کتاب دارای تمرین در این حوزه ...



فصل اول

بازدهی، تحلیل و مرتبه الگوریتم ها

الگوریتم ها

- الگوریتم (تعریف): یک دستورالعمل دقیق (بدون ابهام) برای حل یک مساله
- ممکن است روش‌های (الگوریتم‌های) متفاوتی برای حل یک مساله وجود داشته باشد
  - جستجوی یک کلمه در یک دیکشنری
    - sequential search
    - binary search
- نیازمند روشی برای ارزیابی الگوریتم‌ها هستیم
  - ارزیابی زمان و حافظه

اهمیت ساخت الگوریتم های کارآمد

- هر چقدر هم که سرعت کامپیوتر بالا باشد، یا قیمت حافظه کاهش یابد، بازدهی الگوریتم‌ها همواره باید مد نظر باشد.
- چرا؟!
  - مقایسه دو الگوریتم جستجوی ترتیبی و دودویی!

سری فیبوناچی(بازگشتی)

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

سری فیبوناچی (الگوریتم تکراری)

```
    index i;
    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 ];
}
```

ارزیابی زمانی الگوریتم ها

- تحلیلی مناسب است که مستقل از نوع کامپیوتر، زبان برنامه نویسی، برنامه نویس، و همه جزییات الگوریتم باشد
- کارایی الگوریتم با تعیین تعداد دفعاتی که **عملیات اصلی** (مثل مقایسه، جمع، ضرب و ...) به عنوان تابعی از **اندازه ورودی** انجام می‌شود، سنجیده می‌شود.
  - تعیین عملیات اصلی نیاز به تجربه و مهارت دارد.
  - همواره تعداد دفعات اجرای عملیات اصلی متناسب با اندازه ورودی (n) است.

تحلیل پیچیدگی درحالات مختلف

- Worst-case $W(n)$
  - حداکثر تعداد دفعاتی که الگوریتم، عمل اصلی را به ازای اندازه ورودی n اجرا می کند.
- Average-case $A(n)$
  - میانگین تعداد دفعاتی که الگوریتم، عمل اصلی را به ازای اندازه ورودی n اجرا می کند.
  - در اینجا باید به هر ورودی یک احتمال داده شود.
- Best-case $B(n)$
  - حداقل تعداد دفعاتی که الگوریتم عمل اصلی را به ازای اندازه ورودی n اجرا می‌کند.

مقایسه

- $A(n)$: زمانی مفید است که الگوریتم به دفعات زیاد با ورودی‌های مختلف استفاده شود.
- $W(n)$: زمانی مفید است که حد بالای زمانی مهم باشد.
- $B(n)$: در موارد بسیار کمی استفاده می‌شود.

مثال: بررسی پیچیدگی محاسبه مجموع عناصر آرایه

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

B(n)=A(n)=W(n) = n

بررسی پیچیدگی محاسبه مجموع عناصر آرایه (حل)

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

B(n)=A(n)=W(n) = n

مثال: بررسی پیچیدگی جستجوی ترتیبی

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

B(n) = ?

W(n) = ?

A(n) = ?

بررسی پیچیدگی جستجوی ترتیبی (حل)

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

B(n) = 1

W(n) = n

A(n) = 1/n(1+2+3+4+…+n)=1/2(n+1)

مثال:بررسی پیچیدگی مرتب سازی تعویضی

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

عمل اصلی؟

بررسی پیچیدگی مرتب سازی تعویضی(حل)

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

مهمترین عمل اصلی در جستجو و مرتب سازی تعداد مقایسه است.

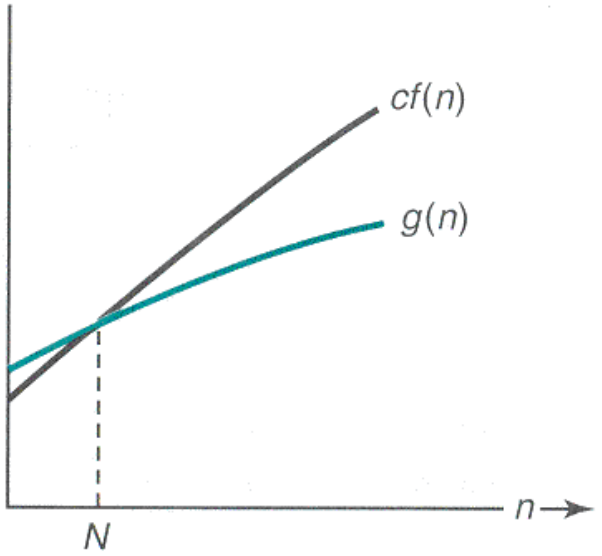
$$B(n)=A(n)=W(n) = n-1 + n-2 + n-3 + \ldots+1=n(n-1)/2$$

مرتبه

- رفتار الگوریتم را وقتی که اندازه ورودی n به اندازه کافی بزرگ باشد، مورد بحث است.
  - مثال: $100n$ نسبت به $0.01n^2$ بازدهی بیشتری دارد.

مرتبه - O بزرگ
Asymptotic Upper Bound

- $g(n) = O\big(f(n)\big) \Leftrightarrow \exists c > 0, n_0 \geq 0, \forall n \geq n_0 \Longrightarrow g(n) \leq cf(n)$



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

O بزرگ (ادامه)

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

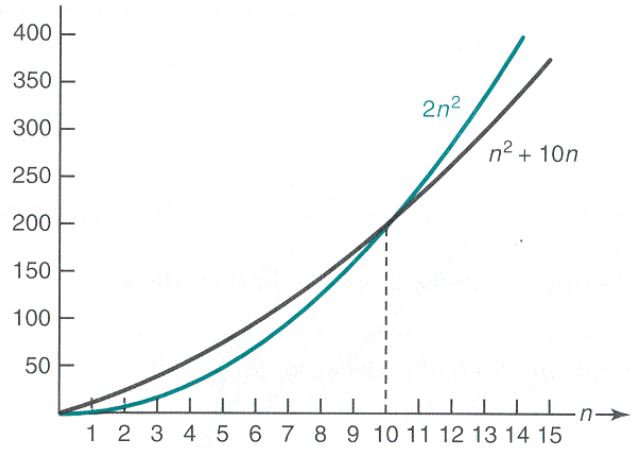


Figure 1.5 ● The function $n^2 + 10n$ eventually stays beneath the function $2n^2$.

- O بزرگ، یک **مرز بالایی مجانبی** را روی تابع قرار می دهد.

O بزرگ (ادامه)

- اثر مولفه درجه دوم، خطی و ثابت در پیچیدگی:

- 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

خصوصیات مرتبه (تابع چندجمله‌ای)

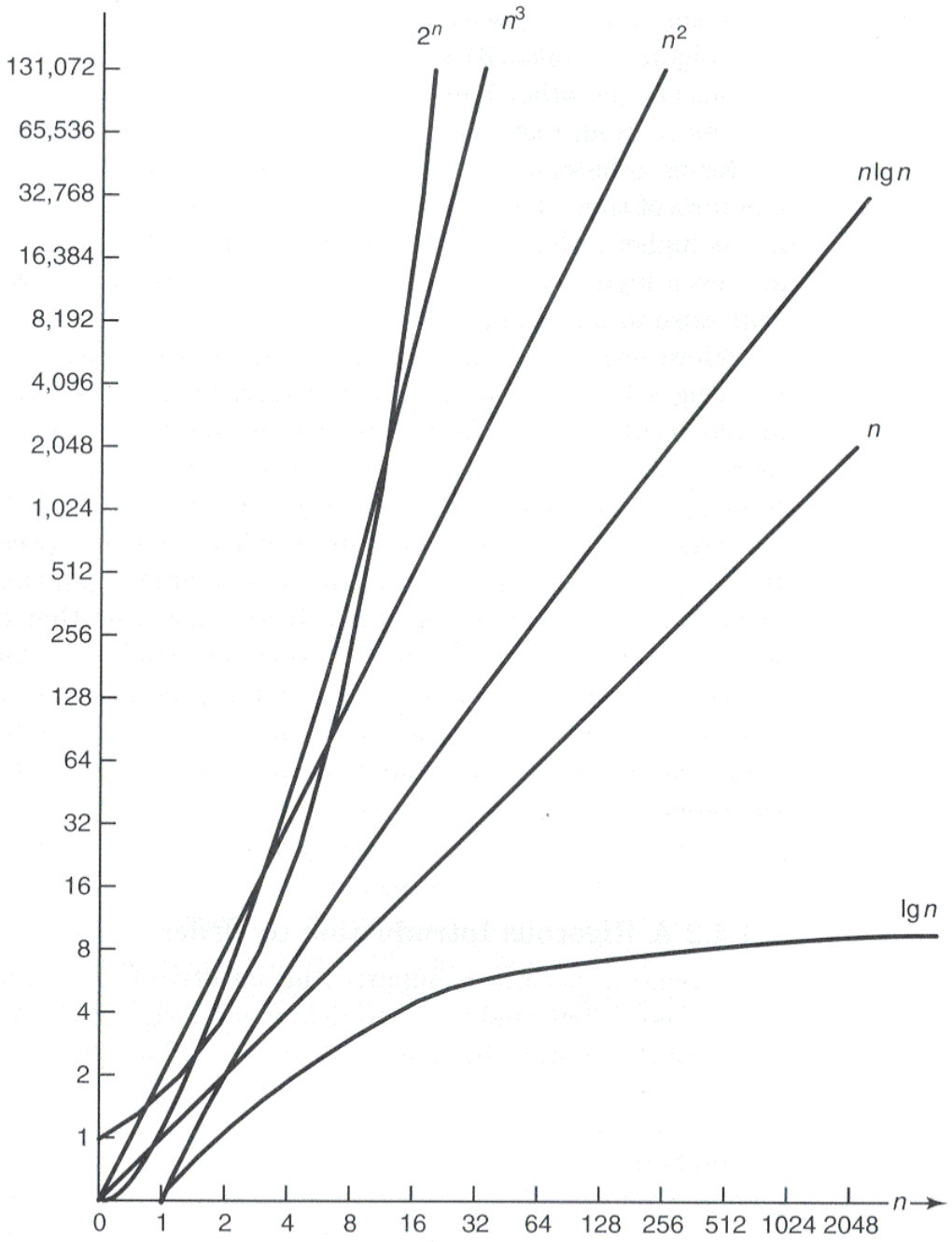
قضیه: اگر تابع چند جمله ای به صورت زیر باشد

$$f(n) = a_m n^m + \ldots + a_1 n^1 + a_0$$

در نتیجه داریم:

$$f(n) = O(n^m)$$

مقایسه توابع پیچیدگی



مقایسه توابع پیچیدگی (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.

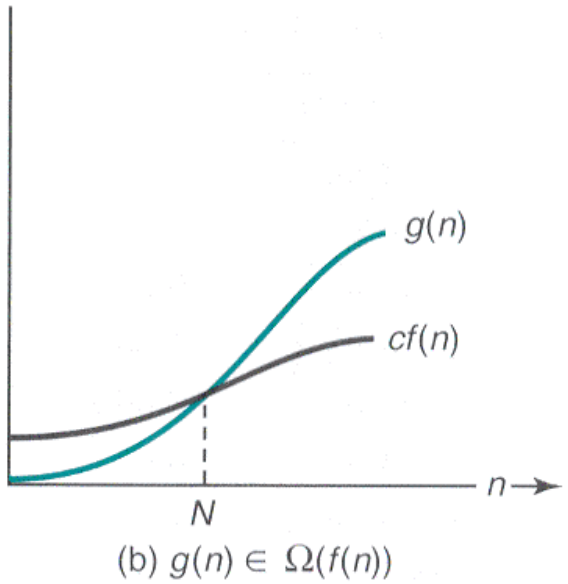
†1 ms = 10^{-3} second.

در کامپیوتر مفروض اجرای هر دستور اصلی ۱ نانوثانیه زمان می‌برد.

امگا Ω

Asymptotic Lower Bound

$$g(n) = \Omega\big(f(n)\big) \Leftrightarrow \exists c > 0, n_0 \geq 0, \forall n \geq n_0 \Longrightarrow g(n) \geq cf(n)$$



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

امگا، یک **مرز پایینی مجانبی** را روی تابع قرار می دهد.

خصوصیات مرتبه (تابع چندجمله‌ای)

قضیه: اگر تابع چند جمله ای به صورت زیر باشد

$$f(n) = a_m n^m + \ldots + a_1 n^1 + a_0$$

در نتیجه داریم:

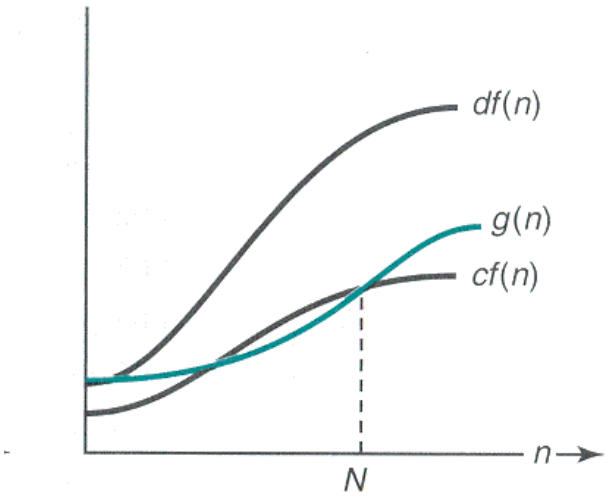
$$f(n) = \Omega(n^m)$$

تتا Θ (هم مرتبه)

$$\theta\big(f(n)\big) = O(f(n)) \cap \Omega(f(n))$$

اگر $g(n) \in \Theta(f(n))$ می گوییم، $g(n)$، تتای $f(n)$ است.

یعنی رشد g برابر رشد f است.



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

$$g(n) = \theta\big(f(n)\big) \Leftrightarrow \exists c, d > 0, n_0 \geq 0, \forall n \geq n_0 \Longrightarrow$$

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

o کوچک
little o

$$g(n) = o(f(n)) \Leftrightarrow \forall c > 0, \exists n_0 \geq 0;\ n \geq n_0 \Longrightarrow g(n) < cf(n)$$

- اگر $g(n) \in o(f(n))$ می گوییم، $g(n)$، اوی کوچک $f(n)$ است.
- اگر $g(n) \in o(f(n))$ می گوییم، $g(n)$، بهتر از $f(n)$ است.

ω کوچک(امگای کوچک)
little ω

$$g(n) = \omega\big(f(n)\big) \Leftrightarrow \forall c > 0, \exists n_0 \geq 0;\ n \geq n_0 \Longrightarrow g(n) > cf(n)$$

- اگر $g(n) \in \omega(f(n))$ می گوییم، $g(n)$، امگای کوچک $f(n)$ است.
- اگر $g(n) \in \omega(f(n))$ می گوییم، $g(n)$، بدتر از $f(n)$ است.

دسته بندی توابع پیچیدگی

- $g(n) \in o(f(n)) \Longrightarrow g(n) \in O(g(n))$
- $g(n) \in O\big(f(n)\big) \Longrightarrow f(n) \in \Omega\big(g(n)\big)$
- $g(n) \in \Omega(f(n)) \Longrightarrow f(n) \in O(g(n))$
- همیشه ساده ترین تابع در هر دسته از توابع پیچیدگی به عنوان نماینده آن دسته استفاده می‌شود.
  - $\Theta(1)$, $\Theta(n^2)$, etc

خصوصیات مرتبه

- همه توابع لگاریتمی در یک دسته قرار دارند.
- If $b > 1$ and $a > 1$, then $\log_a n \in \Theta(\log_b n)$
- توابع نمایی در یک دسته قرار ندارند.
- If $b > a > 0$, then $a^n \in o(b^n)$
- n! بدتر از هر تابع نمایی است.
- For all $a > 0$, $a^n \in o(n!)$

سوال

- رادیکال و لگاریتم چه ارتباطی باهم دارند؟

$$\sqrt{n} \; ? \log(n)$$

$$2^n ? 5^{\ln n}$$

$$\ln(n!)$$

نحوه ارزشیابی

- دو نوع سوال ممکن است مطرح شود:

۱- پیچیدگی زمانی یک تابع چند است؟

۲- تعداد دقیق اجرای عمل اصلی را محاسبه کنید.

دستور ❊ چندبار اجرا می شود

```
for(i=1;i<=n;i++)
{
        j=n;
        while(j>=1)
                {
                        j = floor(j/2);(*)
                }
}
```

خصوصیات مرتبه (ادامه)

- ترتیب دسته های پیچیدگی زیر را در نظر بگیرید:

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

تعریف حدی از پیچیدگی‌های محاسباتی

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

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Longrightarrow f(n) = \omega(g(n))$

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c, c > 0 \Longrightarrow f(n) = \theta(g(n))$

پیوستگی

چند نکته درباره توابع لگاریتمی

$$* \log_a^n = \frac{\log_b^n}{\log_b^a} = \underbrace{\log_a^b}_{c} \log_b^n = c \log_b^n$$

$a, b > 1$

c عددی ثابت

$$c_1 < c \Rightarrow \log_b^n > c_1 \log_b^n$$
$$c_2 > c \Rightarrow \log_b^n < c_2 \log_b^n$$

$$\log_b^n = \theta(\log_a^n)$$

$$* \quad 5^{\ln n} = n^{\ln 5}$$

یکدیگر را چند جمله‌ای

$$* \log n! = \log(n \times (n-1) \times (n-2) \times \dots \times 2 \times 1) =$$
$$\log n + \log(n-1) + \log n-2 \dots < n \log n$$

pars rasam