بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۰۱)

طراحی الگوریتم‌ها

حسین فلسفین

ادامهٔ مباحث فصل اول:

* *In some cases the number of times the basic operation is done depends not only on the input size, but also on the input's values.*

* *The basic operation is not done the same number of times for all instances of size n.*

* *Such an algorithm does not have an every-case time complexity.*

$$\cancel{T(n)}$$

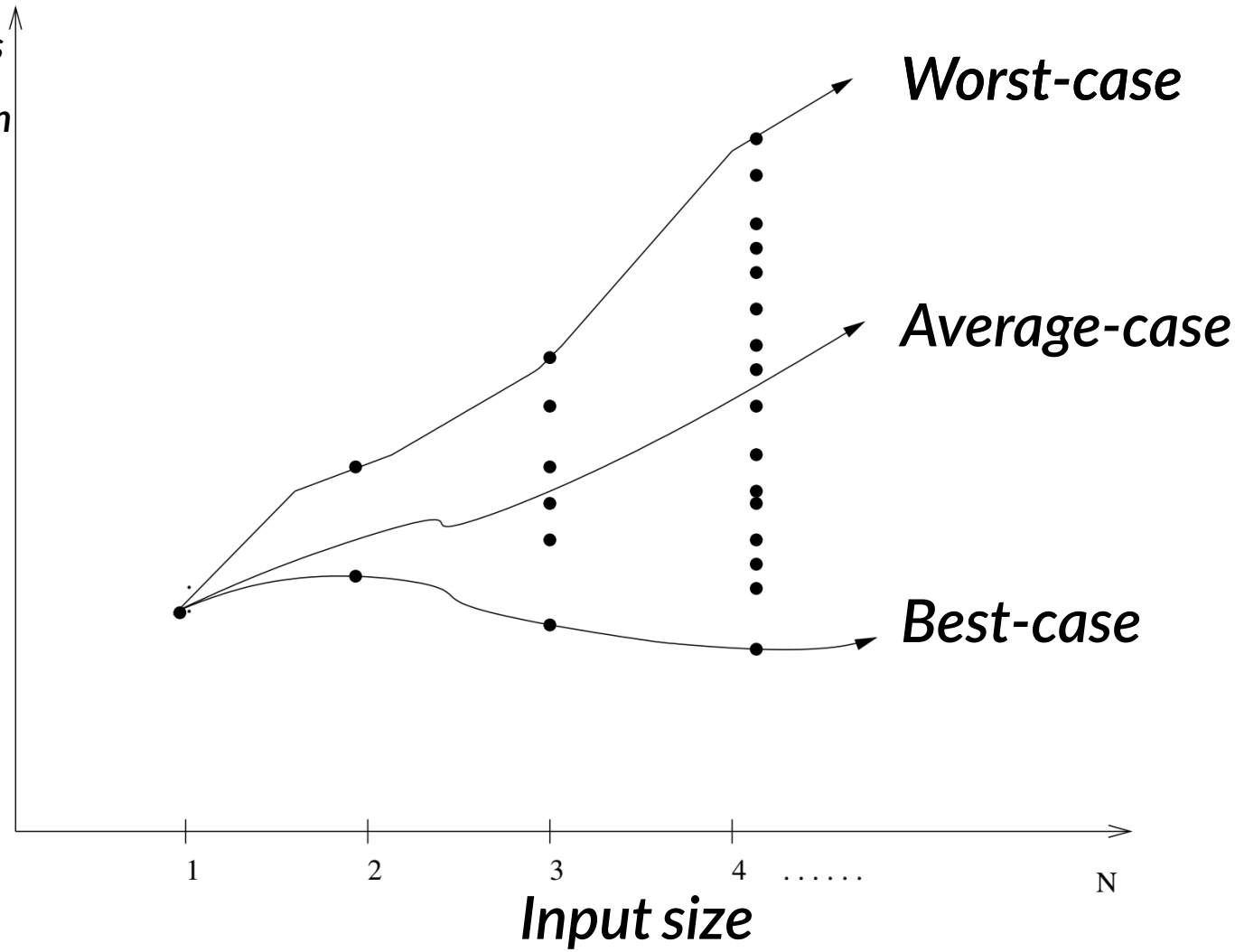
* *However, this does not mean that we cannot analyze such algorithms, because there are three other analysis techniques that can be tried.*

Best, worst, and average-case time complexity

* For a given algorithm, $W(n)$ is defined as the maximum number of times the algorithm will ever do its basic operation for an input size of n. So $W(n)$ is called the worst-case time complexity of the algorithm.
* For a given algorithm, $A(n)$ is defined as the average (expected value) of the number of times the algorithm does the basic operation for an input size of n. $A(n)$ is called the average-case time complexity of the algorithm.
* A final type of time complexity analysis is the determination of the smallest number of times the basic operation is done. For a given algorithm, $B(n)$ is defined as the minimum number of times the algorithm will ever do its basic operation for an input size of n. So $B(n)$ is called the best-case time complexity of the algorithm.

Best, worst, and average-case time complexity

The number of times the algorithm does the basic operation for an instance of size n



- ***If*** $T(n)$ ***exists, then*** $W(n) = A(n) = B(n) = T(n)$**.**
- ***For algorithms that do not have every-case time complexities, we do worst-case and average-case analyses much more often than best-case analyses.***
- ***An average-case analysis is valuable because it tells us how much time the algorithm would take when used many times on many different inputs.***
- ***An average-case analysis would not suffice in a system that monitors a nuclear power plant. In this case, a worst-case analysis would be more useful because it would give us an upper bound on the time taken by the algorithm.***
- ***For both the applications just discussed, a best-case analysis would be of little value.***

We have discussed only the analysis of the time complexity of an algorithm. All the same considerations just discussed also pertain to analysis of memory complexity, which is an analysis of how efficient the algorithm is in terms of memory.

برای الگوریتم ***Sequential Search***

$$B(n) = 1$$

چه هنگام واقع می‌شود؟ کلید x در موضع اول آرایهٔ S ظاهر شود، یعنی $S[1] = x$.

$$W(n) = n$$

چه هنگام واقع می‌شود؟ کلید x در موضع پایانی آرایهٔ S ظاهر شود، یا اینکه اصلاً در آرایه ظاهر نشود.

. . . . . .

$$A(n) = ?$$

محاسبهٔ $A(n)$ تحت سناریوی اول

* ***It is known that x is in S, where the items in S are all distinct.***

* ***We have no reason to believe that x is more likely to be in one array slot than it is to be in another. Based on this information, for $1 \leq k \leq n$, the probability that x is in the kth array slot is $\frac{1}{n}$.***

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

If the probabilities differ, then the average case gives a different outcome. For example, if the probability of finding a number in the first cell equals $\frac{1}{2}$, the probability of finding it in the second cell equals $\frac{1}{4}$, and the probability of locating it in any of the remaining cells is the same and equal to

$$\frac{1-\frac{1}{2}-\frac{1}{4}}{n-2}=\frac{1}{4(n-2)}$$

then, on the average, it takes

$$\frac{1}{2}+\frac{2}{4}+\frac{3+4+\cdots+n}{4(n-2)}=1+\frac{n(n+1)-6}{8(n-2)}=1+\frac{n+3}{8}$$

steps to find a number, which is approximately four times better than $\frac{n+1}{2}$ found previously for the uniform distribution. Note that the probabilities of accessing a particular cell have no impact on the best and worst cases.

محاسبهٔ $A(n)$ تحت سناریوی دوم

- ***x may not be in the array.***
- ***We assign some probability p to the event that x is in the array.***
- ***If x is in the array, we will again assume that it is equally likely to be in any of the slots from 1 to n. The probability that x is in the kth slot is then $\frac{p}{n}$, and the probability that it is not in the array is $1 - p$.***

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

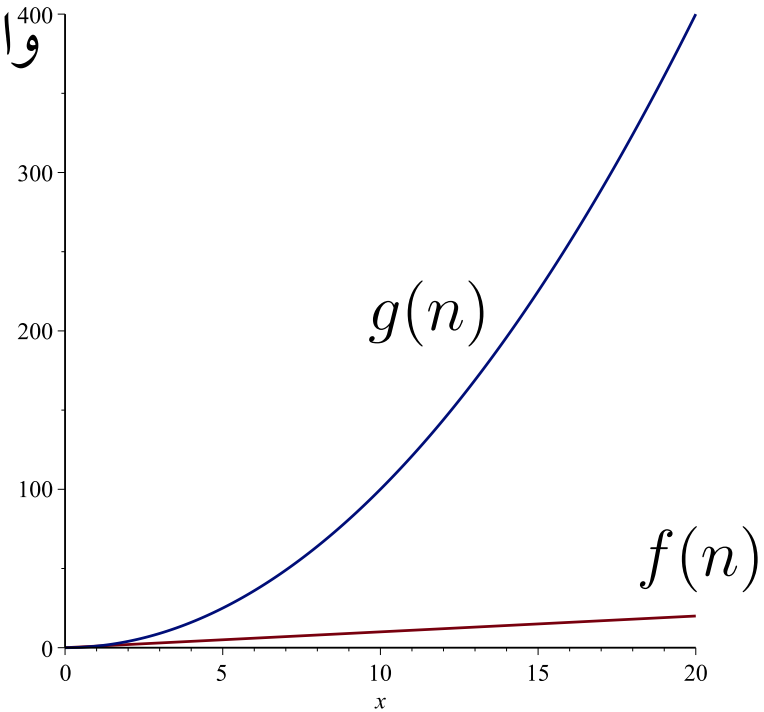
$$\frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p) = n\left(1 - \frac{p}{2}\right) + \frac{p}{2}.$$

نگاهی شهودی و غیررسمی به مفهوم مرتبه

In general, a complexity function can be any function that maps the positive integers to the nonnegative reals. When not referring to the time complexity or memory complexity for some particular algorithm, we will usually use standard function notation, such as $f(n)$ and $g(n)$, to represent complexity functions.

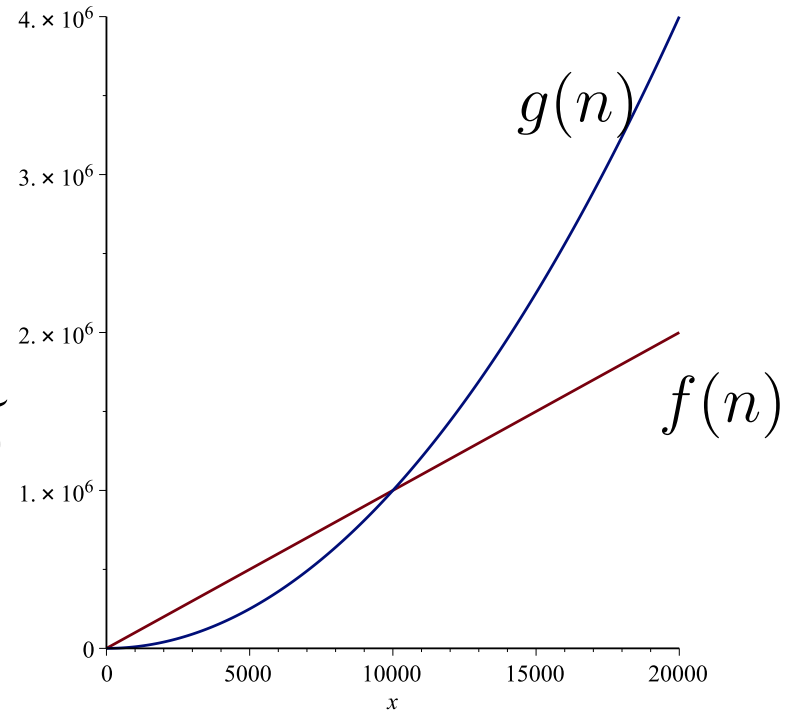
مقایسهٔ رفتار $f(n) = n$ با $g(n) = n^2$:

واضح است که $f(n)$ بهتر از $g(n)$ است



نگاهی شهودی و غیررسمی به مفهوم مرتبه

مقایسهٔ رفتار $f(n) = 100n$ با $g(n) = 0.01n^2$:



آیا $f(n)$ بهتر از $g(n)$ است؟

$f(n) = an + b$ ☞ ***Linear-time complexity function*** $(a \in \mathbb{R}^+)$

$g(n) = an^2+bn+c$ ☞ ***Quadratic-time complexity function*** $(a \in \mathbb{R}^+)$

نگاهی شهودی و غیررسمی به مفهوم مرتبه

Any linear-time algorithm is eventually more efficient than any quadratic-time algorithm.

Eventual = Final

عاقل آن است که اندیشه کند پایان را (سعدی)

In the theoretical analysis of an algorithm, we are interested in eventual behavior.

نشان خواهیم داد که الگوریتم‌ها را می‌توان بر اساس رفتار ***eventual***شان طبقه‌بندی کرد. در این‌صورت، ما می‌توانیم به‌راحتی تعیین کنیم که آیا رفتار ***eventual*** یک الگوریتم بهتر از دیگری هست یا خیر؟

نگاهی شهودی و غیررسمی به مفهوم مرتبه

n	$0.1n^2$	$0.1n^2 + n + 100$
۱۰	۱۰	۱۲۰
۲۰	۴۰	۱۶۰
۵۰	۲۵۰	۴۰۰
۱۰۰	۱۰۰۰	۱۲۰۰
۱۰۰۰	۱۰۰۰۰۰	۱۰۱۱۰۰

The quadratic term eventually dominates

یعنی

The values of the other terms eventually become insignificant compared with the value of the quadratic term

شهوداً می‌توان اینگونه برداشت کرد که حین دسته‌بندی توابع پیچیدگی ما همواره قادریم ترم‌های ***low-order*** را کنار بنهیم. مثلاً، به‌نظر می‌رسد که تابع $0.1n^3 + 10n^2 + 5n + 25$ با تابع $0.1n^3$ در یک کلاس واقع می‌شوند. ما بعداً به‌شکل رسمی اثبات خواهیم کرد که این کار مجاز است.

نگاهی شهودی و غیررسمی به مفهوم مرتبه

مجموعهٔ همهٔ توابعی که می‌توان آنها را با تابعی به‌فرم an^2 در یک کلاس قرار داد را $\Theta(n^2)$ نام می‌نهیم ($a > 0$). در این‌صورت، تابع را از مرتبهٔ n^2 گوییم.
اگر پیچیدگی زمانی یک الگوریتم در $\Theta(n^2)$ واقع باشد، آن الگوریتم را الگوریتمی ***quadratic-time*** یا الگوریتمی $\Theta(n^2)$ می‌نامیم.

. . . . . .

Complexity Categories

به هرکدام از اینها یک کلاس یا ***Category*** می‌گوییم

به تابعی که کلاس را نمایندگی می‌کند ***Reference function*** می‌گوییم

$\Theta(1)$	***Constant***
$\Theta(\log_2(n))$	***Logarithmic***
$\Theta(n)$	***Linear***
$\Theta(n\log_2(n))$	***Linearithmic***
$\Theta(n^2)$	***Quadratic***
$\Theta(n^3)$	***Cubic***
$\Theta(2^n)$	***Exponential***
$\Theta(n!)$	***Factorial***

An algorithm with exponential or factorial runtime is essentially an impractical algorithm

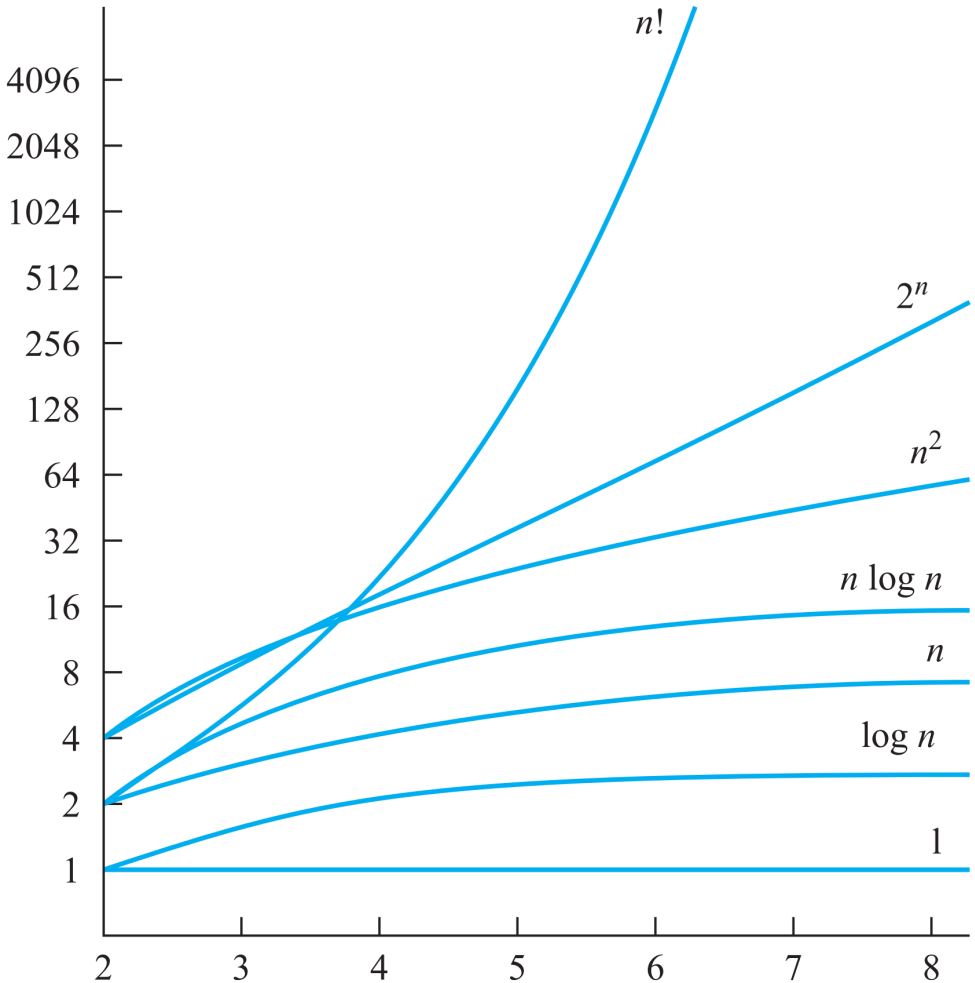
نگاهی شهودی و غیررسمی به مفهوم مرتبه

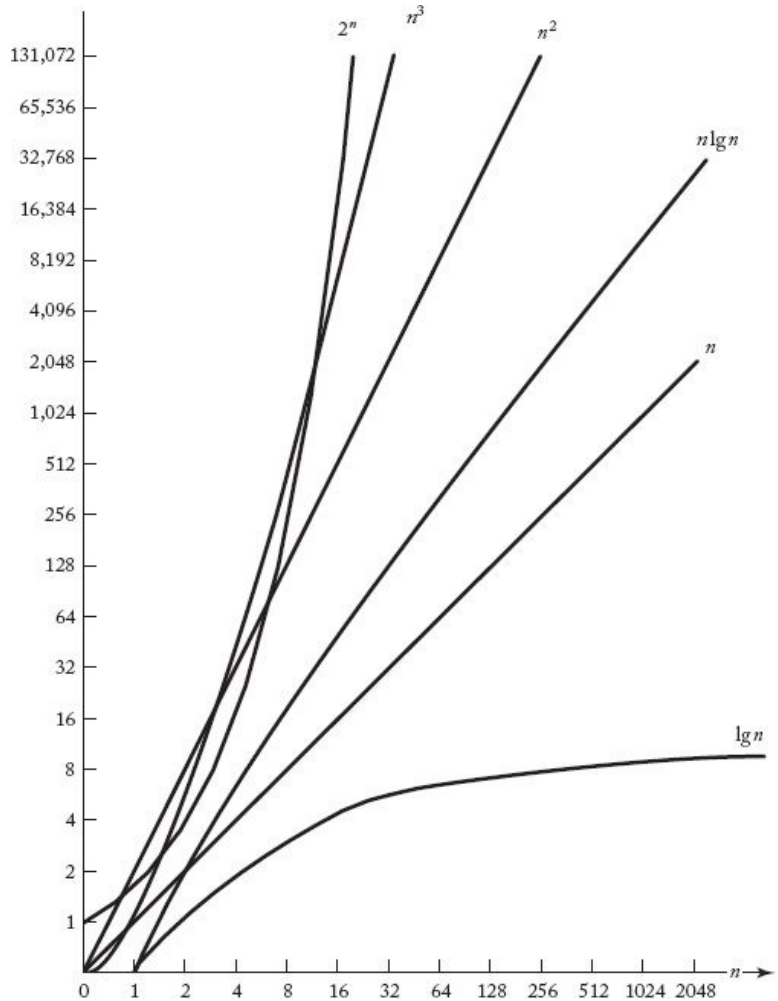
$\Theta(1)$	***Constant***
$\Theta(\log_2(n))$	***Logarithmic***
$\Theta(n)$	***Linear***
$\Theta(n\log_2(n))$	***Linearithmic***
$\Theta(n^2)$	***Quadratic***
$\Theta(n^3)$	***Cubic***
$\Theta(2^n)$	***Exponential***
$\Theta(n!)$	***Factorial***

The functions are listed with the slower-growing functions at the top and faster-growing functions at the bottom. In the context of algorithm analysis, slow growth is good (as n increases, the algorithm takes very little additional time to complete).

در این ترتیب، اگر $f(n)$ در کلاسی بالاتر از کلاس حاوی $g(n)$ واقع باشد، آنگاه $f(n)$ عاقبت (**eventually**) در زیر $g(n)$ قرار می‌گیرد در ترسیم.

نگاهی شهودی و غیررسمی به مفهوم مرتبه





نگاهی شهودی و غیررسمی به مفهوم مرتبه

The simplifying assumption is that it takes 1 nanosecond (10^{-9} second) to process the basic operation for each algorithm.

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.

نگاهی شهودی و غیررسمی به مفهوم مرتبه

The table shows a possibly surprising result. One might expect that as long as an algorithm is not an exponential-time algorithm, it will be adequate. However, even the quadratic-time algorithm takes 31.7 years to process an instance with an input size of 1 billion. This is not to say that algorithms whose time complexities are in the higher-order categories are not useful. Algorithms with quadratic, cubic, and even higher-order time complexities can often handle the actual instances that arise in many applications.

We again assume that one elementary step takes one nanosecond.
(*$\log$ *denotes the logarithm with basis* 2*.)

n	$100n \log n$	$10n^2$	$n^{3.5}$	$n^{\log n}$	2^n	$n!$
10	3 μs	1 μs	3 μs	2 μs	1 μs	4 ms
20	9 μs	4 μs	36 μs	420 μs	1 ms	76 years
30	15 μs	9 μs	148 μs	20 ms	1 s	$8 \cdot 10^{15}$ y.
40	21 μs	16 μs	404 μs	340 ms	1100 s	
50	28 μs	25 μs	884 μs	4 s	13 days	
60	35 μs	36 μs	2 ms	32 s	37 years	
80	50 μs	64 μs	5 ms	1075 s	$4 \cdot 10^7$ y.	
100	66 μs	100 μs	10 ms	5 hours	$4 \cdot 10^{13}$ y.	
200	153 μs	400 μs	113 ms	12 years		
500	448 μs	2.5 ms	3 s	$5 \cdot 10^5$ y.		
1000	1 ms	10 ms	32 s	$3 \cdot 10^{13}$ y.		
10^4	13 ms	1 s	28 hours			
10^5	166 ms	100 s	10 years			
10^6	2 s	3 hours	3169 y.			
10^7	23 s	12 days	10^7 y.			
10^8	266 s	3 years	$3 \cdot 10^{10}$ y.			
10^{10}	9 hours	$3 \cdot 10^4$ y.				
10^{12}	46 days	$3 \cdot 10^8$ y.				

As the table shows, polynomial-time algorithms are faster for large enough instances. The table also illustrates that constant factors of moderate size are not very important when considering the asymptotic growth of the running time.

تعریف رسمی مفهوم الگوریتم ***polynomial-time*** را به‌زودی خواهیم دید.

This table shows the maximum input sizes solvable within one hour with the above six hypothetical algorithms. In (a) we again assume that one elementary step takes one nanosecond, (b) shows the corresponding figures for a ten times faster machine. Polynomial-time algorithms can handle larger instances in reasonable time. Moreover, even a speedup by a factor of 10 of the computers does not increase the size of solvable instances significantly for exponential-time algorithms, but it does for polynomial-time algorithms.

	$100n \log n$	$10n^2$	$n^{3.5}$	$n^{\log n}$	2^n	$n!$
(a)	$1.19 \cdot 10^9$	60000	3868	87	41	15
(b)	$10.8 \cdot 10^9$	189737	7468	104	45	16

Polynomial-time algorithms are sometimes called "good" or "efficient".

قبل از آنکه به‌شکل ***rigorous*** و ***formal*** سراغ مفهوم مرتبه برویم: چرا اینقدر ***order of growth*** یا همان ***eventual behaviour*** مهم است؟

* *A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones.*

* *When we have to solve small instances, it is not immediately clear how much more efficient an algorithm is compared to other algorithms or even why we should care which of them is faster and by how much.*

* *It is only when we have to solve large instances that the difference in algorithm efficiencies becomes both clear and important. For large values of n, it is the function's order of growth that counts.*