



Design and Analysis of Algorithms

Algorithm Analysis

Si Wu

School of CSE, SCUT

cswusi@scut.edu.cn

TA: 1684350406@qq.com



Topics

- **Polynomial Running time**
- **Asymptotic Growth**
- **O -notation**
- **Ω -notation**
- **Θ -notation**



Brute Force

Brute force. For many nontrivial problems, there is a natural brute-force search algorithm that checks every possible solution.

- Typically takes 2^n time or worse for inputs of size n .
- Unacceptable in practice.





Polynomial Running time

Desirable Scaling Property. When the input size doubles, the algorithm should slow down by at most some constant factor C .

An algorithm is **poly-time** if the above scaling property holds.

There exist constants $c > 0$ and $d > 0$ such that, for every input of size n , the running time of the algorithm is bounded above by cn^d primitive computational steps.



Polynomial Running time

We say that an algorithm is **efficient** if it has a polynomial running time.

It really works in practice

- In practice, the poly-time algorithms that people develop have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

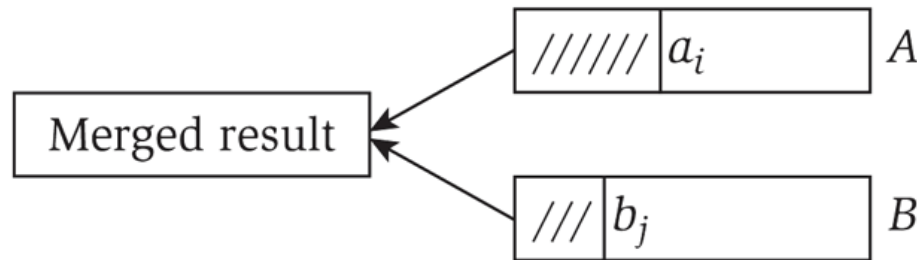
Exceptions. Some poly-time algorithms do have high constants and/or exponents are useless in practice.

Which would you prefer $20n^{120}$ vs. $n^{1+0.02\lg n}$?



Linear Running Time

Merge. Combine two sorted lists A and B into sorted whole.



```
i = 1, j = 1
while (both lists are nonempty) {
    if ( $a_i \leq b_j$ ) append  $a_i$  to output list and increment i
    else          append  $b_j$  to output list and increment j
}
append remainder of nonempty list to output list
```

Merging two lists, each of length n , takes $O(n)$ time.
After each compare, the length of output list increases by 1.



Why It Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1,000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long



Types of Analyses

- **Worst case.** Running time guarantee for any input of size n .
- **Probabilistic.** Expected running time of a randomized algorithm.
- **Average-case.** Expected running time for a random input of size n .

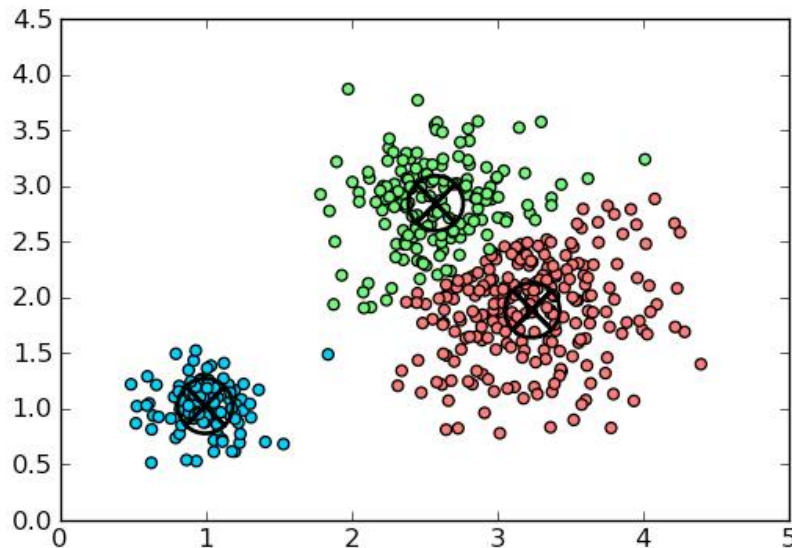


Worst-Case Analysis

Worst case. Running time guarantee for any input of size n .

- Generally captures efficiency in practice.
- But hard to find effective alternative.

Exceptions. Some exponential-time algorithms are used widely in practice because the worst-case instances seem to be rare.



K-means clustering algorithm



Asymptotic Growth

In the insertion-sort example, we discussed that when analyzing algorithms we are

- **interested in worst-case running time as function of input size n .**
- **not interested in exact constants in bound.**
- **not interested in lower order terms.**



Asymptotic Growth

We want to express rate of growth of standard functions:

- the leading term with respect to n .**
- ignoring constants in front of it**

Ex. $k_1n + k_2 \sim n$

$$k_2n \log n \sim n \log n$$

$$k_1n^2 + k_2n + k_3 \sim n^2$$

We also want to formalize e.g. that a $n \log n$ algorithm is better than a n^2 algorithm.



O-notation

$O(g(n)) = \{f(n): \text{There exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$

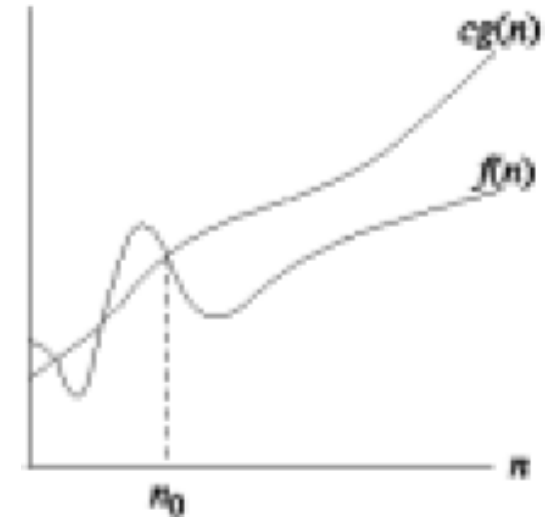
-- $O(.)$ is used to asymptotically **upper bound** a function.

-- $O(.)$ is used to bound worst-case running time.

Ex. $f(n) = 32n^2 + 17n + 1$

- $f(n)$ is $O(n^2)$
- $f(n)$ is also $O(n^3)$
- $f(n)$ is neither $O(n)$ nor $O(n \lg n)$

Typical usage. Insertion-Sort makes $O(n^2)$ compares to sort n elements.





O-notation

Notational abuses

$O(g(n))$ is a set of functions, but computer scientists often write $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$

Ex. Consider $f(n) = 5n^3$ and $g(n) = 3n^2$

- We have $f(n) = O(n^3) = g(n)$.
- Thus, $f(n) = g(n)$. **X**

Non-negative functions. When using big O notation, we assume that the functions involved are non-negative.



O-notation

Ex.

- $1/3n^2 - 3n \in O(n^2)$

Because $1/3n^2 - 3n \leq cn^2$ if $c \geq 1/3 - 3/n$ which holds for $c = 1/3$ and $n > 1$.

- $k_1n^2 + k_2n + k_3 \in O(n^2)$

Because $k_1n^2 + k_2n + k_3 \leq (k_1 + |k_2| + |k_3|)n^2$ and for $c > k_1 + |k_2| + |k_3|$ and $n \geq 1$, $k_1n^2 + k_2n + k_3 \leq cn^2$.

- $k_1n^2 + k_2n + k_3 \in O(n^3)$

As $k_1n^2 + k_2n + k_3 \leq (k_1 + |k_2| + |k_3|)n^3$ (upper bound).



O-notation

Note:

When we say “the running time is $O(n^2)$ ” we mean that the worst-case running time is $O(n^2)$ – the best case might be better.

Use of O -notation often makes it much easier to analyze algorithms; we can easily prove the insertion-sort time bound $O(n^2)$.



Ω -notation

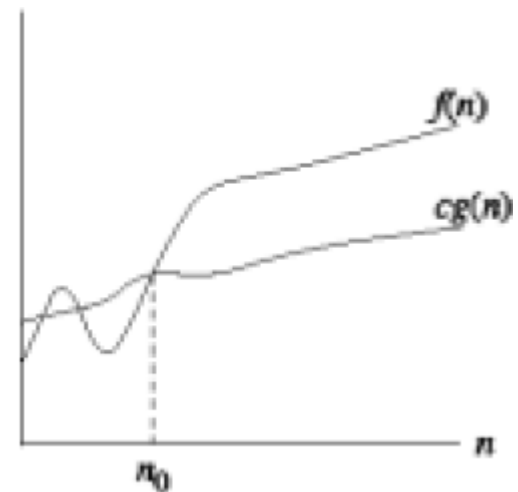
$\Omega(g(n)) = \{f(n): \text{There exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$

- We use Ω -notation to give a **lower bound** on a function.

Ex. $f(n) = 32n^2 + 17n + 1$

- $f(n)$ is both $\Omega(n^2)$ and $\Omega(n)$
- $f(n)$ is neither $\Omega(n^3)$ nor $\Omega(n^3 \lg n)$

Typical usage. Any compare-based sorting algorithm requires $\Omega(n \lg n)$ compares in the worst case.





Ω -notation

Ex.

- $1/3n^2 - 3n \in \Omega(n^2)$

Because $1/3n^2 - 3n \geq cn^2$ if $c \leq 1/3 - 3/n$ which holds for $c = 1/6$ and $n > 18$.

- $k_1n^2 + k_2n + k_3 \in \Omega(n^2)$
- $k_1n^2 + k_2n + k_3 \in \Omega(n)$



Ω -notation

Note:

When we say “the running time is $\Omega(n^2)$ ” we mean that the best-case running time is $\Omega(n^2)$ – the worst case might be worse.

Insertion-Sort:

- Best case: $\Omega(n)$ – when the input array is already sorted.
- Worst case: $\Omega(n^2)$ – when the input array is reverse sorted.



Θ -notation

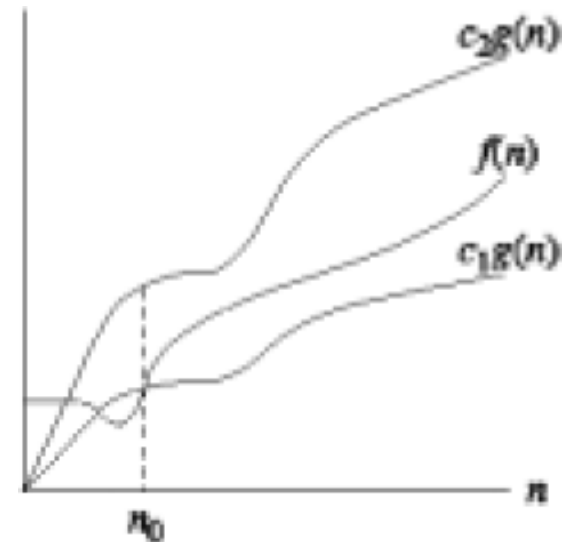
$\Theta(g(n)) = \{f(n): \text{There exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\}$

- We use Θ -notation to give a **tight bound** on a function.
- $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Ex. $f(n) = 32n^2 + 17n + 1$

- $f(n)$ is $\Theta(n^2)$
- $f(n)$ is neither $\Theta(n)$ nor $\Theta(n^3)$

Typical usage. Merge-Sort makes $\Theta(n \lg n)$ compares to sort n elements.





Θ -notation

Ex.

- $k_1n^2+k_2n+k_3 \in \Theta(n^2)$
- $6n\log n + \sqrt{n}\log^2 n = \Theta(n\log n)$

We need to find $c_1, c_2, n_0 > 0$ such that $c_1n\log n \leq 6n\log n + \sqrt{n}\log^2 n \leq c_2n\log n$ for $n \geq n_0$.

- $c_1n\log n \leq 6n\log n + \sqrt{n}\log^2 n \rightarrow c_1 \leq 6 + \log n / \sqrt{n}$, which is true if we choose $c_1 = 6$ and $n_0 = 1$.
- $6n\log n + \sqrt{n}\log^2 n \leq c_2n\log n \rightarrow 6 + \log n / \sqrt{n} \leq c_2$, which is true if we choose $c_2 = 7$ and $n_0 = 2$. This is because $\log n \leq \sqrt{n}$ if $n \geq 2$. So $c_1 = 6, c_2 = 7$ and $n_0 = 2$ works.



Useful Facts

- If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c > 0$, then $f(n)$ is $\Theta(g(n))$.

By definition of the limit, there exists n_0 such that for all $n \geq n_0$

$$\frac{1}{2}c \leq \frac{f(n)}{g(n)} \leq 2c$$

Thus, $f(n) \leq 2cg(n)$ for all $n \geq n_0$, which implies $f(n)$ is $O(g(n))$.

Similarly, $f(n) \geq \frac{1}{2}cg(n)$ for all $n \geq n_0$, which implies $f(n)$ is $\Omega(g(n))$.

- If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$, then $f(n)$ is $O(g(n))$ but not $\Theta(g(n))$.