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

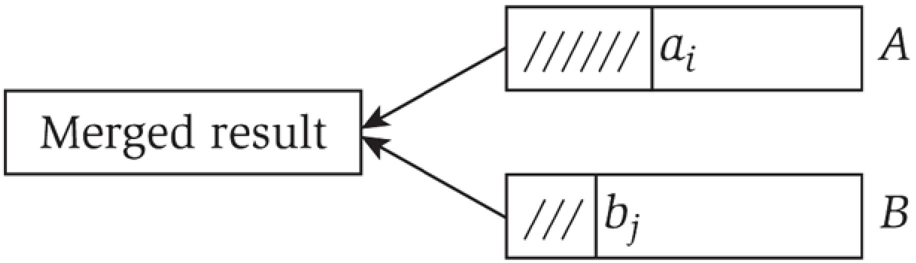
- **In practice, the poly-time algorithms that people develop have low constants and low exponents.**
- **Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.**

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

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

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 ≤ 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

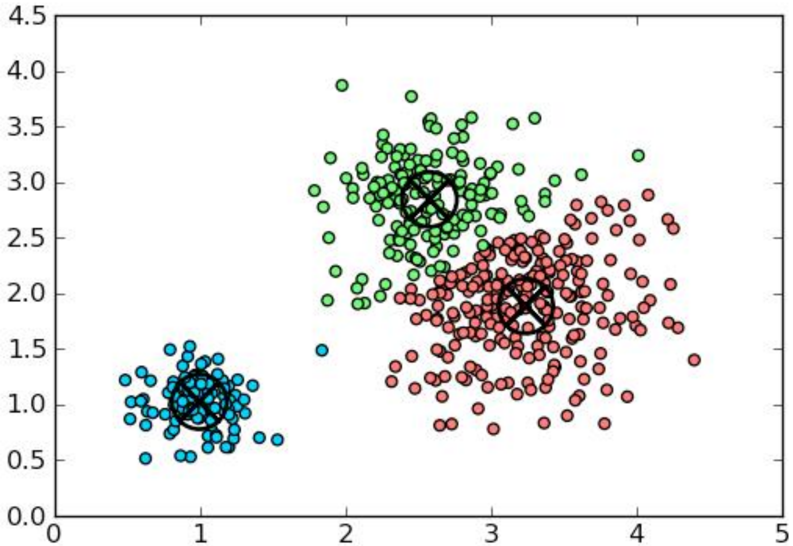
- **Worst case.** **Running time guarantee for any input of size n.**
- **Probabilistic.** **Expected running time of a randomized algorithm.**
- **Average-case.** **Expected running time for a random input of size n.**

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. $k_1 n + k_2 \sim n$

$k_2 n \log n \sim n \log n$

$k_1 n^2 + k_2 n + k_3 \sim n^2$

We also want to formalize e.g. that a *nlogn* algorithm is better than a *n*^{2} algorithm.

O-notation

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

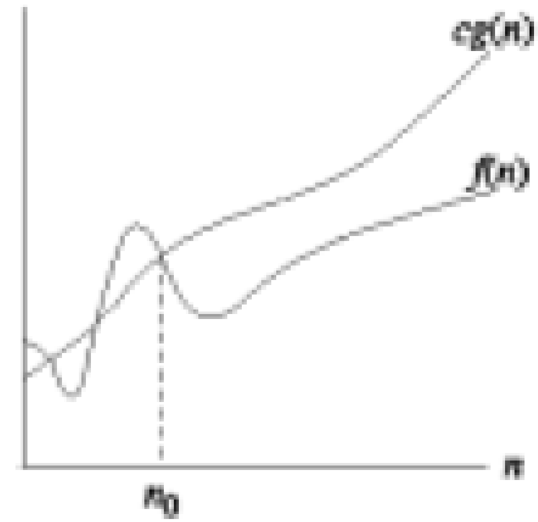
--*O(.)* is used to asymptotically upper bound a function.

--*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 $O(nlgn)$

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



O-notation

Notational abuses

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

Ex. Consider $\boldsymbol{f}(\boldsymbol{n}) = \mathbf{5}\boldsymbol{n}^{\mathbf{3}}$ **and** $\boldsymbol{g}(\boldsymbol{n}) = \mathbf{3}\boldsymbol{n}^{\mathbf{2}}$

- **We have** $\boldsymbol{f}(\boldsymbol{n}) = \boldsymbol{O}\left(\boldsymbol{n}^{\mathbf{3}}\right) = \boldsymbol{g}(\boldsymbol{n})$.
- **Thus,** $\boldsymbol{f}(\boldsymbol{n}) = \boldsymbol{g}(\boldsymbol{n})$. **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 $1/3n^2 - 3n \le cn^2$ **if** $c \ge 1/3\text{-}3/n$ **which holds for** $c = 1/3$ **and** $n > 1$.

- $k_1 n^2 + k_2 n + k_3 \in O(n^2)$

Because $k_1 n^2 + k_2 n + k_3 \le (k_1 + |k_2| + |k_3|)n^2$ **and for** $c > k_1 + |k_2| + |k_3|$ **and** $n \ge 1$, $k_1 n^2 + k_2 n + k_3 \le cn^2$.

- $k_1 n^2 + k_2 n + k_3 \in O(n^3)$

As $k_1 n^2 + k_2 n + k_3 \le (k_1 + |k_2| + |k_3|)n^3$ **(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

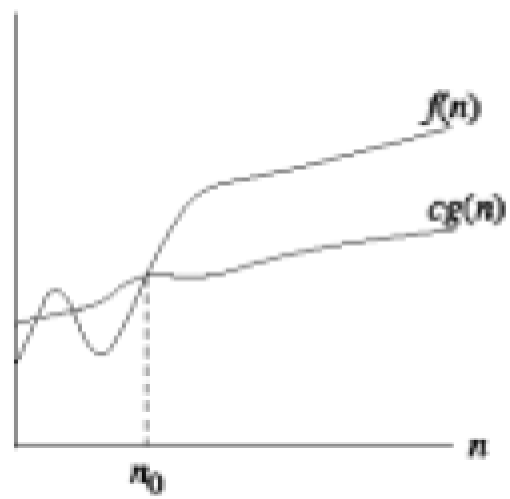
$\Omega(g(n)) = \{f(n)$: There exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$

- **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** $\Omega(n^3 lgn)$

Typical usage. Any compare-based sorting algorithm requires $\Omega(nlgn)$ 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: *$O(n^2)$*– when the input array is reverse sorted.**

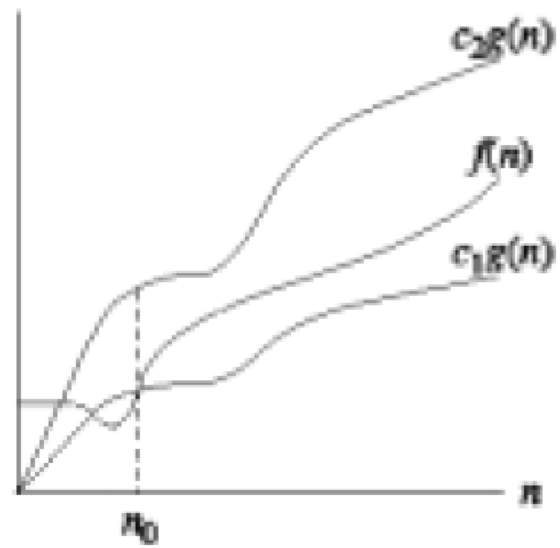
Θ-notation

$\Theta(g(n))$ = {$f(n)$: There exist positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$}

- 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.

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

- $6nlogn + \sqrt{n}log^2n = \Theta(nlogn)$

We need to find $c_1, c_2, n_0 > 0$ such that $c_1nlogn \le 6nlogn + \sqrt{n}log^2n \le c_2nlogn$ for $n \ge n_0$.

- **$c_1nlogn \le 6nlogn + \sqrt{n}log^2n \rightarrow c_1 \le 6 + logn/\sqrt{n}$, which is true if we choose $c_1 = 6$ and $n_0 = 1$.**
- **$6nlogn + \sqrt{n}log^2n \le c_2nlogn \rightarrow 6 + logn/\sqrt{n} \le c_2$, which is true if we choose $c_2 = 7$ and $n_0 = 2$. This is because $logn \le \sqrt{n}$ if $n \ge 2$. So $c_1 = 6$, $c_2 = 7$ and $n_0 = 2$ works.**

Useful Facts

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