Growth of Functions (Ch 3)

When the size of the input arrays is large ($n \gg 1$), we can study asymptotic efficiency of algorithms: how the running time increases as n increases.

3.1 Asymptotic notation

Recall, the insertion sort algorithm has the worst-case running time $an^2 + bn + c$ for some constants a, b, c. We wrote

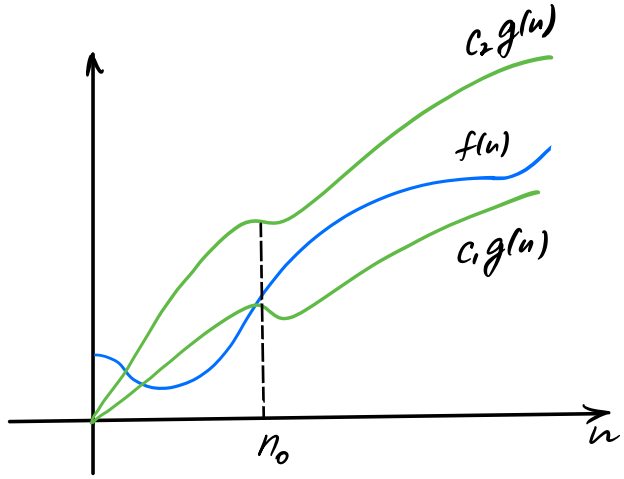
$$T(n) = an^2 + bn + c = \Theta(n^2)$$ "theta of n^2"

Θ-notation ("theta")

Def For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions

$$\Theta(g(n)) = \{ f(n) : \text{there exist const } c_1, c_2 > 0 \text{ and } n_0 : 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \geq n_0 \}$$

$f(n)$ is "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$ for large n



$f(n) = \Theta(g(n))$

We say that $g(n)$ is an asymptotically tight bound for $f(n)$.

<u>Note</u>: $f(n)$ has to be <u>asymptotically nonnegative</u>: $f(n) \geq 0$ for sufficiently large n.

<u>Ex</u> $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

How to find c_1, c_2 and n_0?

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2 \quad \Big| \ \frac{1}{n^2}$$

for all $n \geq n_0$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2 \qquad c_2 \geq \frac{1}{2}$$

$\frac{1}{2} - \frac{3}{n} \leq c_2$ for any $n \geq 1$

$n \geq 1 \Rightarrow \frac{1}{2} - \underbrace{\frac{3}{n}}_{>0} \leq \frac{1}{2} \Rightarrow \frac{1}{2} - \frac{3}{n} \leq c_2$ works for any $c_2 \geq \frac{1}{2}$

$$c_1 \le \frac{1}{2} - \frac{3}{n} \qquad\qquad n \geq n_0 \quad \frac{1}{n} \le \frac{1}{n_0}$$

$$\frac{1}{2} - \frac{3}{n} \geq \frac{1}{2} - \frac{3}{n_0} = c_1 \qquad -\frac{3}{n} \geq -\frac{3}{n_0}$$

if $n \geq 7 \Rightarrow \frac{1}{2} - \frac{3}{n} \geq \frac{1}{2}^{(7} - \frac{3}{7}^{(2} = \frac{7-6}{14} = \frac{1}{14} \equiv c_1$

$\Rightarrow c_1 \le \frac{1}{2} - \frac{3}{n}$ true for $n \geq 7$ and $c_1 \le \frac{1}{14}$

Hence, for $c_1 = \frac{1}{14}$, $c_2 = \frac{1}{2}$, $n_0 = 7$, we have

$$\frac{1}{2} n^2 - 3n = \Theta(n^2)$$

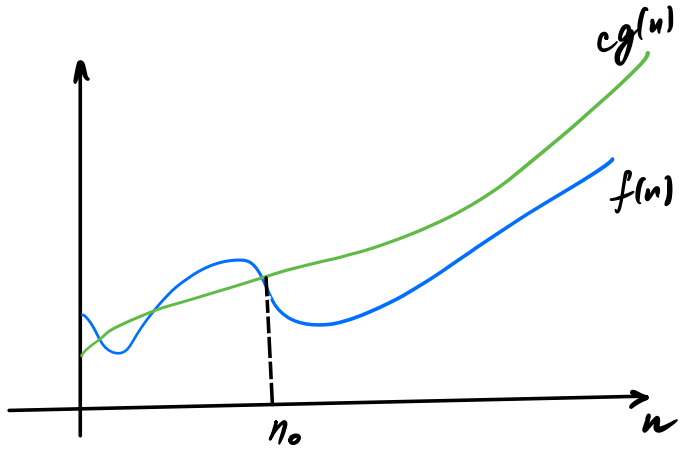
O-notation (big O notation)

The Θ-notation asymptotically bounds a function from above and below. When we have only an asymptotic upper bound, we use O-notation.

Def For a given function $g(n)$, we denote by $O(g(n))$ the set of functions

$$O(g(n)) = \{ f(n) : \text{there exist constants } c \text{ and } n_0 : 0 \le f(n) \le c\,g(n) \text{ for all } n \geq n_0 \}$$

$f(n)$ has an upper bound with a const factor c



$f(n) = O(g(n))$

<u>Note</u> :

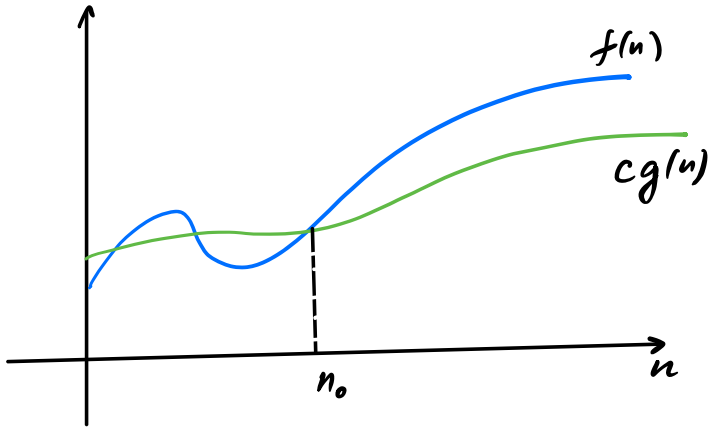
$$f(n) = \Theta(g(n)) \underset{\not\Leftarrow}{\Rightarrow} f(n) = O(g(n))$$

<u>Ω -notation</u> (Omega notation)

Ω-notation provides an <u>asymptotic lower bound</u>.

For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions:

$$\Omega(g(n)) = \{ f(n) : \text{there exist constants } c \text{ and } n_0 : 0 \le cg(n) \le f(n) \text{ for all } n \geq n_0 \}$$



$f(n) = \Omega(g(n))$

<u>Theorem 3.1</u>

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

<u>Note</u>: when we say that the <u>running time</u> of an algorithm is $\Omega(g(n))$, we mean that for any input of size n, the running time on that input is at least a constant times $g(n)$, for sufficiently large n.

We give a lower bound on the best-case running time of the algorithm.

Running time of insertion sort is both

$\Omega(n)$ and $O(n^2)$.

best-case worst-case

o-notation (little o notation)

Def We define $o(g(n))$ as the set

$o(g(n)) = \{ f(n) :$ for any $c > 0$, there exists a const $n_0 > 0$: $0 \le f(n) < c\,g(n)$ for all $n \geq n_0 \}$

not tight upper bound

Ex $2n = o(n^2)$ but $2n^2 \neq o(n^2)$

since

$$\lim_{n\to\infty} \frac{2n}{n^2} = 0 \qquad \lim_{n\to\infty} \frac{2n^2}{n^2} = 2 \neq 0$$

$f(n) = O(g(n))$

$0 \le f(n) \le c g(n)$ for <u>some</u> const $c > 0$

$f(n) = o(g(n))$

$0 \le f(n) < c g(n)$ for <u>all</u> const $c > 0$

$f(n) = o(g(n))$: $f(n)$ is insignificant to $g(n)$ as $n \to \infty$ or

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

<u>ω - notation</u>

$f(n) \in \omega(g(n))$ iff $g(n) \in o(f(n))$

$\omega(g(n)) = \{ f(n)$: for any $c > 0$ there exists const $n_0 > 0$:

$0 \le c\,g(n) < f(n)$ for all $n \ge n_0\}$

not tight low bound

<u>Ex</u> $\frac{n^2}{2} = \omega(n)$ but $\frac{n^2}{2} \ne \omega(n^2)$

$$\lim_{n\to\infty} \frac{\frac{n^2}{2}}{n} = \infty \qquad \lim_{n\to\infty} \frac{\frac{n^2}{2}}{n^2} = \frac{1}{2} \ne \infty$$

$f(n) = \omega(g(n))$: $f(n)$ is arbitrarily large relative to $g(n)$ as $n \to \infty$

There are different types of algorithm design. The insertion sort uses <u>incremental approach</u>: having array $A[1..j-1]$ sorted, it consider one element $A[j]$ and finds an appropriate position for this element, so that array $A[1..j]$ becomes sorted. Another approach is "<u>divide-and-conquer</u>" approach. It is used in many modern algorithms and exhibits smaller running time for large n. Merge sort uses this idea and it is faster than insertion sorting method for large n.

<u>The divide-and-conquer approach</u>

Algorithms are <u>recursive</u>. They call themselves once or more times and solve related problems but of smaller size.

At the end, results of solving smaller problems are combined to give the solution to the original problem.

The divide-and-conquer approach has three steps.

<u>Divide</u> the problem into subproblems, that are similar to original problem but smaller in size.

<u>Conquer</u> subproblems by solving them recursively. If problem is small enough, then it is solved in a straightforward manner.

Combine: merge solutions of smaller subproblems into the solution of original problem.

Merge sort

Divide the original array into two $n/2$ length subarrays.

Conquer the subproblems (two subarrays) by calling merge sort recursively.

Combine: merge solutions of two subarrays into one.

Note the recursion "bottoms up" when the subarray has length 1, since it is automatically sorted.

The algorithm uses a subroutine Merge(A, p, q, r). It assumes that subarrays $A[p..q]$ and $A[q+1,..r]$ are sorted. $p \leq q < r$. It merges these two subarrays into one and overwrites $A[p..r]$.

The operation count for merge subroutine is $\Theta(n)$, where $n = r-p+1$, the number of elements in these two subarrays.

How does merge subroutine work?

~~Let~~ two subarrays that are sorted be in two piles on table face with the up. smallest element on the top These are input arrays. The output pile/array will have cards face down. At the beginning, the output pile is empty. At a time, one compares the top

cards and picks the smaller one and places it in the output pile face down. This process stops when one of the piles is empty. After that the remaining cards from the other pile are transferred to output pile with face down (since they are sorted and their values are greater than in what was transferred before). Since this process involves at most n steps, the running time is $\Theta(n)$.

To simplify the algorithm, two cards with "sentinel" value are placed at the bottom of each pile. The sentinel element is ∞. When one reaches ∞ element, this means that there are no more cards in the pile.

We also know that we have exactly $n = r - p + 1$ elements to transfer.

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1..n1+1] and R[1..n2+1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1+1] = ∞
R[n2+1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
```

```
        i = i + 1
else    A[k] = R[j]
        j = j + 1
```