O-notation

DEFINITION A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \le cg(n) \quad \text{for all } n \ge n_0.$$

Ω-notation

DEFINITION A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \ge cg(n) \quad \text{for all } n \ge n_0.$$

Θ-notation

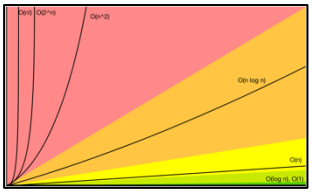
DEFINITION A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 g(n) \le t(n) \le c_1 g(n) \quad \text{for all } n \ge n_0.$$

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n). \end{cases}^{3}$$

In the ***decrease-by-a-constant*** variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one (Figure 4.1), although other constant size reductions do happen occasionally.

The ***decrease-by-a-constant-factor*** technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two. (Can you give an example of such



Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	*linearithmic*	Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

Divide-and-conquer is probably the best-known general algorithm design technique. Though its fame may have something to do with its catchy name, it is well deserved: quite a few very efficient algorithms are specific implementations of this general strategy. Divide-and-conquer algorithms work according to the following general plan:

1. A problem is divided into several subproblems of the same type, ideally of about equal size.
2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

In the ***preorder traversal***, the root is visited before the left and right subtrees are visited (in that order).
In the ***inorder traversal***, the root is visited after visiting its left subtree but before visiting the right subtree.
In the ***postorder traversal***, the root is visited after visiting the left and right subtrees (in that order).

Important Summation Formulas

1. $\sum_{i=l}^{u} 1 = \underbrace{1+1+\cdots+1}_{u-l+1 \text{ times}} = u - l + 1$ (l, u are integer limits, $l \le u$); $\sum_{i=1}^{n} 1 = n$
2. $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$
3. $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$
4. $\sum_{i=1}^{n} i^k = 1^k + 2^k + \cdots + n^k \approx \frac{1}{k+1}n^{k+1}$
5. $\sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = \frac{a^{n+1}-1}{a-1}$ $(a \ne 1)$; $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$
6. $\sum_{i=1}^{n} i2^i = 1 \cdot 2 + 2 \cdot 2^2 + \cdots + n2^n = (n-1)2^{n+1} + 2$
7. $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \approx \ln n + \gamma$, where $\gamma \approx 0.5772\ldots$ (Euler's constant)
8. $\sum_{i=1}^{n} \lg i \approx n \lg n$

$\log_a 1 = 0$
$\log_a a = 1$
$\log_a x^y = y \log_a x$
$\log_a xy = \log_a x + \log_a y$
$\log_a \frac{x}{y} = \log_a x - \log_a y$
$a^{\log_b x} = x^{\log_b a}$
$\log_a x = \frac{\log_b x}{\log_b a} = \log_a b \log_b x$

Sum Manipulation Rules

1. $\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i$
2. $\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$
3. $\sum_{i=l}^{u} a_i = \sum_{i=l}^{m} a_i + \sum_{i=m+1}^{u} a_i$, where $l \le m < u$
4. $\sum_{i=l}^{u} (a_i - a_{i-1}) = a_u - a_{l-1}$

Master Theorem If $f(n) \in \Theta(n^d)$ where $d \ge 0$ in recurrence (5.1), then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Mathematical Analysis

Decide on a parameter (or parameters) for measuring the size of the input
↓
Identify the Algorithm's basic operation
↓
Determine if best, worst, and average case will be different
↓
Set up a sum representing the number of times the operation happens
↓
Find a closed form formula for the expression

Convex Hulls

- Given an arbitrary set of points S, the convex hull of S is the smallest convex set that contain all the points in S.
  - Barricading sleeping tigers

Analysis of Quicksort

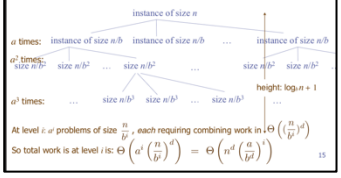
- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: choose 1st element from sorted array! — $\Theta(n^2)$
- Average case: random arrays — $\Theta(n \log n)$
- Improvements:
  - better pivot selection: median-of-three partitioning
  - separate partition for keys equal to pivot
  - switch to insertion sort on small sub-problems
  - elimination of recursion
  - These combine to give 20–25% improvement

Solve the recurrence:

$C(n) = 2\,C(n/2) + n - 1$ (omitting $_{worst}$)
$C(n)/n = 2\,C(n/2)/n + 1 - 1/n$ (divide by n)
$= C(n/2)/(n/2) + 1 - 1/n$ (algebra)
$= C(n/4)/(n/4) + 1 + 1 - 1/n - 2/n$ (subst. $n \Rightarrow n/2$)
$= C(n/8)/(n/8) + 1 + 1 + 1 - 1/n - 2/n - 4/n$
…
$= C(n/n)/(n/n) + 1 + 1 + 1 + \ldots - 1/n - 2/n - 4/n - \ldots$
$C(n)/n = C(1) + \underbrace{1+1+1+\ldots}_{\lg n \text{ terms}} - 1/n - 2/n - 4/n - \ldots$
$\approx \lg n$ (neglecting small terms)
$C(n) = n \lg n$

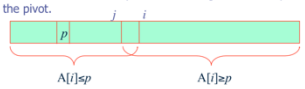
Problem:

- Why are there three cases in the Master Theorem?
  - Look at the recursion tree:



Quicksort

- Select a pivot (partitioning element) – a random element
- Rearrange the list so that all the elements in the first j positions are smaller than or equal to the pivot and all the elements in the last $n-i$ positions are larger than or equal to the pivot.

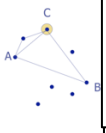


- Exchange the pivot with the element in its partition closest to the center — the pivot is now in its final position
- Sort the two subarrays recursively

As sorting algorithm is called **stable** if it preserves the relative order of any two equal emblements.

Quickhull

- Divide-and-conquer algorithm for Convex Hull of a set of points P
  - Find two points $A, B \in P$ that are both on the convex hull
  - divide the set P into two subsets, P_l left and P_r right of chord AB.
  - Find point C in P_l furthest from AB.
  - discard the points inside △ABC
  - recurse with chords AC and CB
  - …
  - repeat with chord AB and P_r



- $M(n) = M(n-1) + 1$, $n>0$ and $M(0) = 0$
- $M(n) = M(n-1) + 1$
  $= [M(n-2) + 1] + 1$
  $= M(n-2) + 2$
  ….
  $= M(n-i) + i$, $i<=n$ (generalise)

Put $i=n$: $M(n) = M(0) + n$
$= n$

Runtime Analysis of Selection Sort

- $C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1]$
  $= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}$
- $\frac{(n-1)n}{2} \in \Theta(n^2)$
- How many swaps?
  - $\Theta(n)$, or more precisely, $n-1$
  - This is an attractive property that distinguishes Selection Sort from other sorting algorithms

What is Decrease & Conquer?

- Solves a problem instance of size n by:
  - decreasing n by a constant, e.g., 1, or
  - decreasing n by a constant factor, e.g., 2, or
  - decreasing n by a variable amount (e.g., Euclid's algorithm)

◆ A *graph* is a pair (V,E) where
- V is a set of vertices;
- E is a set of edges {u,v}, where u,v are distinct vertices from V.

Insertion Sort

- Decrease-by-one, sort array A[0…n-1]
- Decrease-by-one, A[0….n-2] is sorted: how can we solve A[0….n-1]?
- All we have to do is put A[n-1] in the correct position.
  - Scan sorted subarray from right to left until you find a smaller element than A[n-1]

Runtime Analysis of Insertion Sort

- $C(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2)$
- This is worst case complexity (array of strictly decreasing values)
- In the best case, comparison A[j] > u is executed only once (array is already sorted in non-decreasing order)

  $\sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$
  - almost-sorted files do arise in a variety of applications, and insertion sort preserves its excellent performance on such inputs.

- Partitioning takes three arguments:
  - An array a[].
  - A left index L.
  - A right index R.
- Partitioning rearranges array elements a[L], a[L+1], …, a[R].
  - Elements before L or after R are not affected.
- Partitioning returns an index i such that:
  - When the function is done, a[i] is what a[R] was before the function was called.
    - We move a[R] to a[i].
  - All elements between a[L] and a[i-1] are not greater than a[i].
  - All elements between a[i+1] and a[R] are not less than a[i].

Improving Performance

- We can improve performance by using as pivot the median of three values:
  - The leftmost element.
  - The middle element.
  - The rightmost element.
- Then, the pivot has better chances of being close to the 50th percentile.
- If the file is already sorted, the pivot is the median.
- Thus, already sorted data is:
  - The worst case (slowest running time) when the pivot is the rightmost element.
  - The best case (fastest run time) when the pivot is the median of the leftmost, middle, and rightmost elements.

- Hoare's partition is more efficient than Lomuto
  - Three times fewer swaps on average
  - Better partition with equal values
- Both have worst case performance $O(n^2)$ when array is sorted.

Improving Performance

- We can improve performance by picking pivot randomly
- Quicksort is not stable but is in place! (no need for extra memory like MergeSort)
- better pivot selection: median-of-three partitioning
- separate partition for keys equal to pivot
- switch to insertion sort on small sub-problems
- elimination of recursion
- These combine to give 20–25% improvement