Algorithms, Design and Analysis

Big-Oh analysis,

Brute Force, Divide and conquer intro

Types of formulas for basic operation count

- Exact formula
  e.g., C(n) = n(n-1)/2
- Formula indicating order of growth with specific multiplicative constant
  e.g., C(n) ˜ 0.5 n^2
- Formula indicating order of growth with unknown multiplicative constant
  e.g., C(n) ˜ cn^2

Order of growth

- Most important: Order of growth within a constant multiple as n? 8
- Example:
  - How much faster will algorithm run on computer that is twice as fast?
  - How much longer does it take to solve problem of double input size?
- See table 2.1

Table 2.1

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Asymptotic growth rate

- A way of comparing functions that ignores constant factors and small input sizes
- O(g(n)): class of functions f(n) that grow *no faster* than g(n)
- T (g(n)): class of functions f(n) that grow *at same rate* as g(n)
- O(g(n)): class of functions f(n) that grow *at least as fast* as g(n)

Big-oh

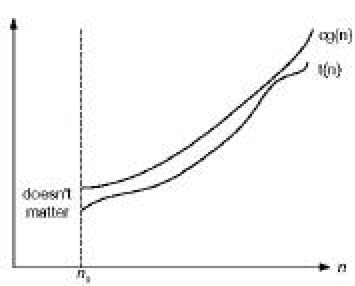


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Big-omega

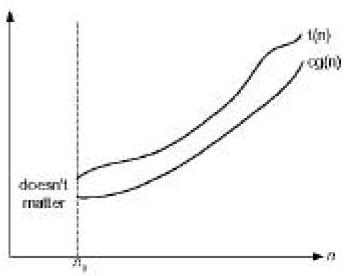


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Big-theta

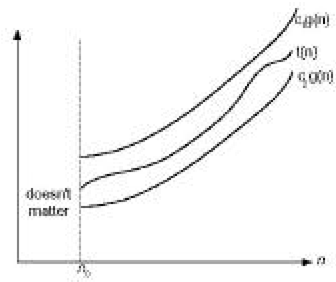


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Establishing rate of growth: Method 1 – using limits

lim$_{n?\ 8}$ $T(n)/g(n)$ =

- 0 order of growth of ***T*(*n*)** < order of growth of ***g*(*n*)**
- *c*>0 order of growth of ***T*(*n*)** = order of growth of ***g*(*n*)**
- 8 order of growth of ***T*(*n*)** > order of growth of ***g*(*n*)**

Examples:

- $10n$ vs. $2n^2$
- $n(n+1)/2$ vs. n^2
- $\log_b n$ vs. $\log_c n$

L'Hôpital's rule

If

- $lim_{n?\ 8}$ *f*(*n*) = $lim_{n?\ 8}$ *g*(*n*) = 8
- and the derivatives *f*´, *g*´ exist,

Then

$$\lim_{n?\ 8} \frac{f(n)}{g(n)} = \lim_{n?\ 8} \frac{f\'(n)}{g\'(n)}$$

Example: log*n* vs. *n*

Establishing rate of growth: Method 2 – using definition

- *f*(*n*) is O(*g*(*n*)) if order of growth of *f*(*n*) = order of growth of *g*(*n*) (within constant multiple)
- There exist positive constant *c* and non-negative integer n_0 such that

$$f(n) = c\ g(n) \text{ for every } n = n_0$$

Examples:

- 10*n* is O(2n^2)
- 5*n*+20 is O(10*n*)

Basic Asymptotic Efficiency classes

1	constant
log *n*	logarithmic
n	linear
n log *n*	*n* log *n*
n^2	quadratic
n^3	cubic
2^n	exponential
n!	factorial

More Big-Oh Examples

- 7n-2

  7n-2 is O(n)

  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \bullet n$ for $n \geq n_0$

  this is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$

  $3n^3 + 20n^2 + 5$ is $O(n^3)$

  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 21$

- 3 log n + log log n

  3 log n + log log n is O(log n)

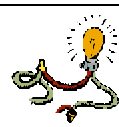
  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + \log \log n \leq c \bullet \log n$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 2$

Big-Oh Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Intuition for Asymptotic Notation



Big-Oh
- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

big-Omega
- f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

big-Theta
- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

little-oh
- f(n) is o(g(n)) if f(n) is asymptotically **strictly less** than g(n)

little-omega
- f(n) is ω(g(n)) if is asymptotically **strictly greater** than g(n)

Brute Force

A straightforward approach usually based on problem statement and definitions

Examples:

1. Computing a^n ($a > 0$, n a nonnegative integer)
2. Computing $n!$
3. Selection sort
4. Sequential search

More brute force algorithm examples:

- Closest pair
  - Problem: find closest among n points in k-dimensional space
  - Algorithm: Compute distance between each pair of points
  - Efficiency:
- Convex hull
  - Problem: find smallest convex polygon enclosing n points on the plane
  - Algorithm: For each pair of points p_1 and p_2 determine whether all other points lie to the same side of the straight line through p_1 and p_2
  - Efficiency:

Brute force strengths and weaknesses

- Strengths:
  - wide applicability
  - simplicity
  - yields reasonable algorithms for some important problems
    - searching
    - string matching
    - matrix multiplication
  - yields standard algorithms for simple computational tasks
    - sum/product of n numbers
    - finding max/min in a list
- Weaknesses:
  - rarely yields efficient algorithms
  - some brute force algorithms unacceptably slow
  - not as constructive/creative as some other design techniques

Divide and Conquer

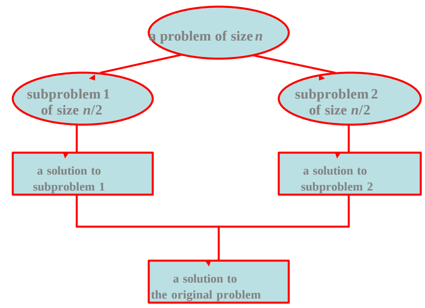
The most well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances

2. Solve smaller instances recursively

3. Obtain solution to original (larger) instance by combining these solutions

Divide-and-conquer technique



Divide and Conquer Examples

- Sorting: mergesort and quicksort
- Tree traversals
- Binary search
- Matrix multiplication-Strassen's algorithm
- Convex hull-QuickHull algorithm

General Divide and Conquer recurrence:

$T(n) = aT(n/b) + f(n)$ where $f(n)$? $\mathrm{T}(n^k)$

1. $a < b^k$ $T(n)$? $\mathrm{T}(n^k)$
2. $a = b^k$ $T(n)$? $\mathrm{T}(n^k \lg n)$
3. $a > b^k$ $T(n)$? $\mathrm{T}(n^{\log_b a})$

Note: the same results hold with O instead of T.

Mergesort

Algorithm:

- Split array A[1..n] in two and make copies of each half in arrays B[1.. $\lfloor n/2 \rfloor$] and C[1.. $\lceil n/2 \rceil$]
- Sort arrays B and C
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Mergesort Example

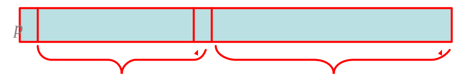
7 2 1 6 4

Efficiency of mergesort

- All cases have same efficiency: T(*n* log *n*)
- Number of comparisons is close to theoretical minimum for comparison-based sorting:
  - $\lceil \log n ! \rceil \sim \lceil n \lg n - 1.44\, n \rceil$
- Space requirement: T(*n*) (NOT in-place)
- Can be implemented without recursion (bottom-up)

Quicksort

- Select a *pivot* (partitioning element)
- Rearrange the list so that all the elements in the positions before the pivot are smaller than or equal to the pivot and those after the pivot are larger than the pivot (See algorithm *Partition* in section 4.2)
- Exchange the pivot with the last element in the first (i.e., = sublist) – the pivot is now in its final position
- Sort the two sublists



The partition algorithm

```
Algorithm Partition(A[l..r])
//Partitions a subarray by using its first element as a pivot
//Input: A subarray A[l..r] of A[0..n − 1], defined by its left and right
//        indices l and r (l < r)
//Output: A partition of A[l..r], with the split position returned as
//        this function's value
p ← A[l]
i ← l; j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])  //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```


Quicksort Example

15 22 13 27 12 10 20 25

Efficiency of quicksort

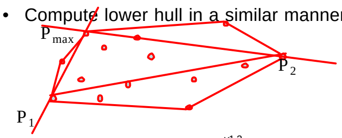
- *Best case*: split in the middle — T(*n* log *n*)
- *Worst case*: sorted array! — T(n^2)
- *Average case*: random arrays — T(*n* log *n*)
- Improvements:
  - better pivot selection: median of three partitioning avoids worst case in sorted files
  - switch to insertion sort on small subfiles
  - elimination of recursion

  these combine to 20-25% improvement
- Considered the method of choice for internal sorting for large files (*n* = 10000)

QuickHull Algorithm

Inspired by Quicksort compute Convex Hull:

- Assume points are sorted by *x*-coordinate values
- Identify extreme points P_1 and P_2 (part of hull)
- Compute upper hull:
  - find point P_{max} that is farthest away from line P_1P_2
  - compute the hull of the points to the left of line P_1P_{max}
  - compute the hull of the points to the left of line $P_{max}P_2$
- Compute lower hull in a similar manner



Efficiency of QuickHull algorithm

- Finding point farthest away from line P_1P_2 can be done in linear time
- This gives same efficiency as quicksort:
  - *Worst case*: T(n^2)
  - *Average case*: T(*n* log *n*)
- If points are not initially sorted by x-coordinate value, this can be accomplished in T(*n* log *n*) — no increase in asymptotic efficiency class
- Other algorithms for convex hull:
  - Graham's scan
  - DCHull

  also in T(*n* log *n*)