Introduction to Algorithm design and analysis

Example: sorting problem.

Input: a sequence of n number $<a_1, a_2, \ldots, a_n>$

Output: a permutation (reordering) $<a_1', a_2', \ldots, a_n'>$

such that $a_1' \leq a_2' \leq \ldots \leq a_n'$.

Different sorting algorithms:

Insertion sort and Mergesort.

Efficiency comparison of two algorithms

- Suppose n=10^6 numbers:
  - Insertion sort: $c_1 n^2$
  - Merge sort: $c_2 n$ (lg n)
  - Best programmer (c_1=2), machine language, one billion/second computer A.
  - Bad programmer (c_2=50), high-language, ten million/second computer B.
  - $2\,(10^6)^2$ instructions/10^9 instructions per second = 2000 seconds.
  - $50\,(10^6 \lg 10^6)$ instructions/10^7 instructions per second $\approx$ 100 seconds.
  - Thus, merge sort on B is 20 times faster than insertion sort on A!
  - If sorting ten million numbers, 2.3 days VS. 20 minutes.
- Conclusions:
  - Algorithms for solving the same problem can differ dramatically in their efficiency.
  - much more significant than the differences due to hardware and software.

Algorithm Design and Analysis

- Design an algorithm
  - Prove the algorithm is correct.
    - Loop invariant.
    - Recursive function.
    - Formal (mathematical) proof.
- Analyze the algorithm
  - Time
    - Worse case, best case, average case.
    - For some algorithms, worst case occurs often, average case is often roughly as bad as the worst case. So generally, worse case running time.
  - Space
- Sequential and parallel algorithms
  - Random-Access-Model (RAM)
  - Parallel multi-processor access model: *PRAM*

Insertion Sort Algorithm (cont.)

INSERTION-SORT(A)

for j = 2 to length[A]
  do $key \leftarrow A[j]$
    //insert A[j] to sorted sequence A[1..j-1]
    $i \leftarrow j-1$
    while i >0 and A[i]>key
      do $A[i+1] \leftarrow A[i]$ //move A[i] one position right
        $i \leftarrow i-1$
    $A[i+1] \leftarrow key$

Correctness of Insertion Sort Algorithm

- Loop invariant
  - At the start of each iteration of the for loop, the subarray A[1..j-1] contains original A[1..j-1] but in sorted order.
- Proof:
  - Initialization : j=2, A[1..j-1]=A[1..1]=A[1], sorted.
  - Maintenance: each iteration maintains loop invariant.
  - Termination: j=n+1, so A[1..j-1]=A[1..n] in sorted order.

Analysis of Insertion Sort

INSERTION-SORT(A)	*cost*	*times*
1. **for** j = 2 to length[A]	c_1	n
2. **do** *key* $\leftarrow$ A[j]	c_2	n-1
3. //insert A[j] to sorted sequence A[1..j-1]	0	n-1
4. $i \leftarrow j$-1	c_4	n-1
5. **while** i >0 and A[i]>*key*	c_5	$\sum_{j=2}^{n} t_j$
6. **do** A[i+1] $\leftarrow$ A[i]	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7. $i \leftarrow i$-1	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8. A[i+1] $\leftarrow$ *key*	c_8	$n - 1$

(t_j is the number of times the while loop test in line 5 is executed for that value of j)

The total time cost $T(n)$ = sum of *cost* × *times* in each line

$$=c_1 n + c_2(n\text{-}1) + c_4(n\text{-}1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j\text{-}1) + c_7\sum_{j=2}^{n}(t_j\text{-}1) + c_8(n\text{-}1)$$

Analysis of Insertion Sort (cont.)

- Best case cost: already ordered numbers
  - t_j=1, and line 6 and 7 will be executed 0 times
  - $T(n) = c_1 n + c_2(n\text{-}1) + c_4(n\text{-}1) + c_5(n\text{-}1) + c_8(n\text{-}1)$
    $=(c_1 + c_2 + c_4 + c_5 + c_8)n – (c_2 + c_4 + c_5 + c_8) = cn + c'$
- Worst case cost: reverse ordered numbers
  - $t_j=j$,
  - so $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = n(n+1)/2\text{-}1$, and $\sum_{j=2}^{n}(t_j – 1) = \sum_{j=2}^{n}(j – 1) = n(n\text{-}1)/2$, and
  - $T(n) = c_1 n + c_2(n\text{-}1) + c_4(n\text{-}1) + c_5(n(n+1)/2 \text{ -}1) + + c_6(n(n\text{-}1)/2 \text{ -}1) + c_7(n(n\text{-}1)/2)+ c_8(n\text{-}1) =((c_5 + c_6 + c_7)/2)n_2 +(c_1 + c_2 + c_4 +c_5/2\text{-}c_6/2\text{-}c_7/2+c_8)n\text{-}(c_2 + c_4 + c_5 + c_8) =an^2+bn+c$
- Average case cost: random numbers
  - in average, $t_j = j/2$. $T(n)$ will still be in the order of n^2, same as the worst case.

Merge Sort—divide-and-conquer

- **Divide:** divide the *n*-element sequence into two subproblems of *n*/2 elements each.
- **Conquer**: sort the two subsequences recursively using merge sort. If the length of a sequence is 1, do nothing since it is already in order.
- **Combine**: merge the two sorted subsequences to produce the sorted answer.

Merge Sort –merge function

- Merge is the key operation in merge sort.
- Suppose the (sub)sequence(s) are stored in the array A. moreover, A[p..q] and A[q+1..r] are two sorted subsequences.
- MERGE(A,p,q,r) will merge the two subsequences into sorted sequence A[p..r]
  - MERGE(A,p,q,r) takes $\Theta(r-p+1)$.

MERGE-SORT(A,p,r)

1. **if** $p < r$
2. **then** $q \leftarrow \lfloor (p+r)/2 \rfloor$
3. MERGE-SORT(A,p,q)
4. MERGE-SORT(A,q+1,r)
5. MERGE(A,p,q,r)

Call to MERGE-SORT(A,1,n) (suppose n=length(A))

Analysis of Divide-and-Conquer

- Described by recursive equation
- Suppose $T(n)$ is the running time on a problem of size n.
- $T(n) = \begin{cases} \Theta(1) & \text{if } n \le n_c \\ aT(n/b)+D(n)+C(n) & \text{if } n > n_c \end{cases}$

Where a: number of subproblems

n/b: size of each subproblem

$D(n)$: cost of divide operation

$C(n)$: cost of combination operation

Analysis of MERGE-SORT

- **Divide**: $D(n) = \Theta(1)$
- **Conquer**: $a=2, b=2$, so $2T(n/2)$
- **Combine**: $C(n) = \Theta(n)$
- $T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ 2T(n/2) + \Theta(n) & \text{if } n>1 \end{cases}$
- $T(n) = \begin{cases} c & \text{if } n=1 \\ 2T(n/2) + cn & \text{if } n>1 \end{cases}$

Compute $T(n)$ by Recursive Tree

- The recursive equation can be solved by recursive tree.
- $T(n) = 2T(n/2)+ cn$, (See its Recursive Tree).
- $\lg n+1$ levels, cn at each level, thus
- Total cost for merge sort is
  - $T(n) = cn\lg n + cn = \Theta(n\lg n)$.
  - Question: best, worst, average?
- In contrast, insertion sort is
  - $T(n) = \Theta(n^2)$.

Recursion tree of $T(n)$=2$T(n/2)$+cn

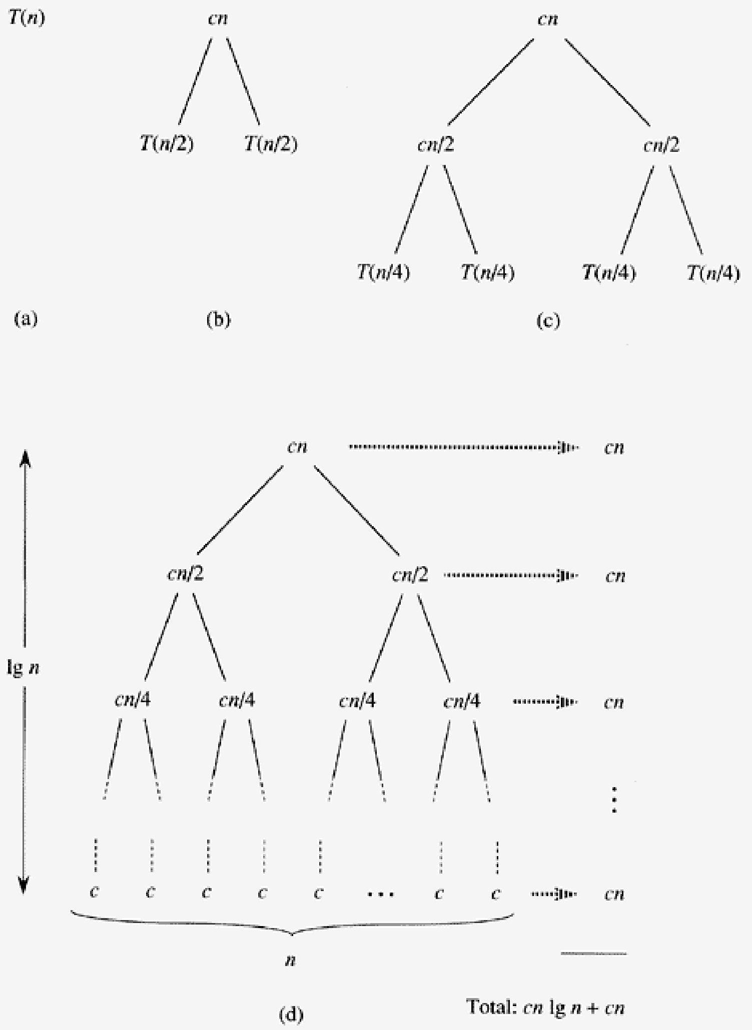
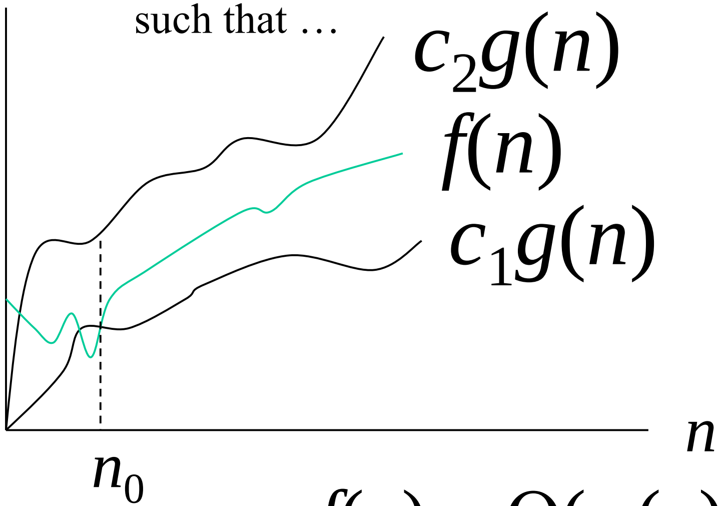


Figure 2.5 The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

Order of growth

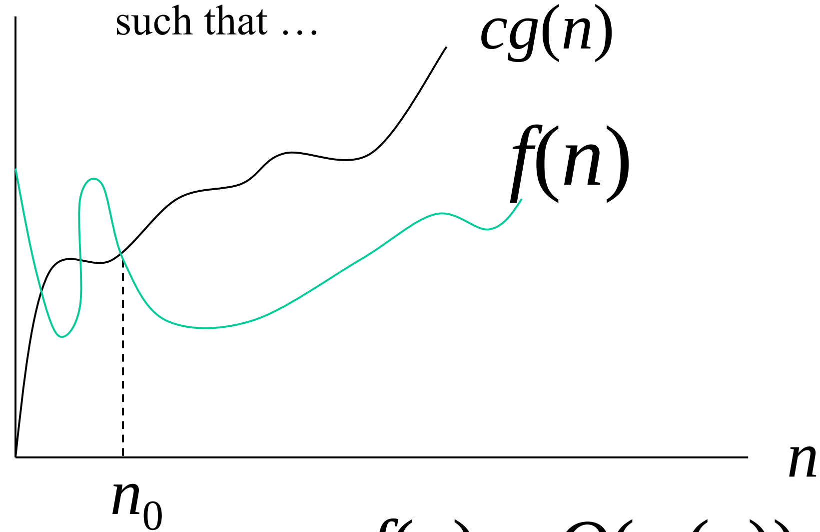
- Lower order item(s) are ignored, just keep the highest order item.
- The constant coefficient(s) are ignored.
- The rate of growth, or the order of growth, possesses the highest significance.
- Use $\Theta(n^2)$ to represent the worst case running time for insertion sort.
- Typical order of growth: $\Theta(1)$, $\Theta(\lg n)$, $\Theta(\sqrt{n})$,$\Theta(n)$, $\Theta(n\lg n)$, $\Theta(n^2)$, $\Theta(n^3)$, $\Theta(2^n)$, $\Theta(n!)$
- Asymptotic notations: Θ, O, Ω, o, ω.

There exist positive constants c_1 and c_2
such that there is a positive constant n_0
such that …



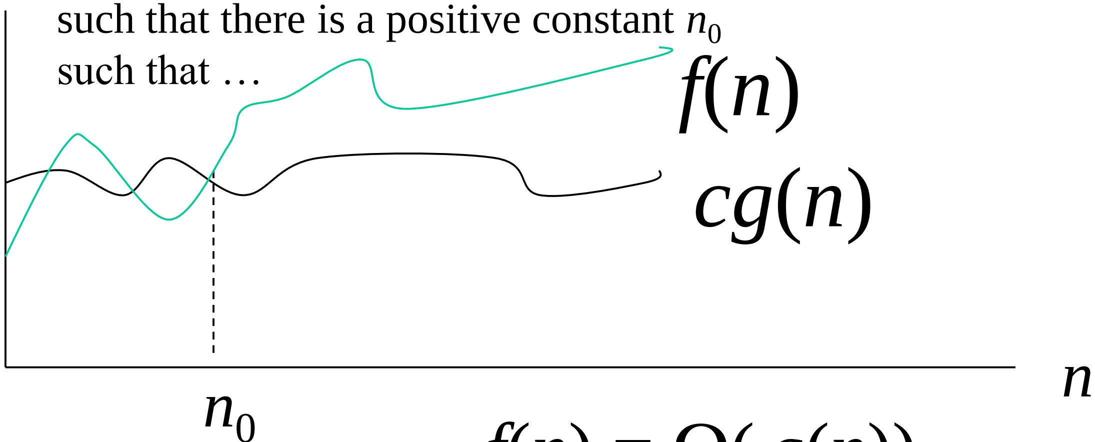
$$f(n) = \Theta(g(n))$$

There exist positive constants c
such that there is a positive constant n_0
such that …



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

There exist positive constants c
such that there is a positive constant n_0
such that …



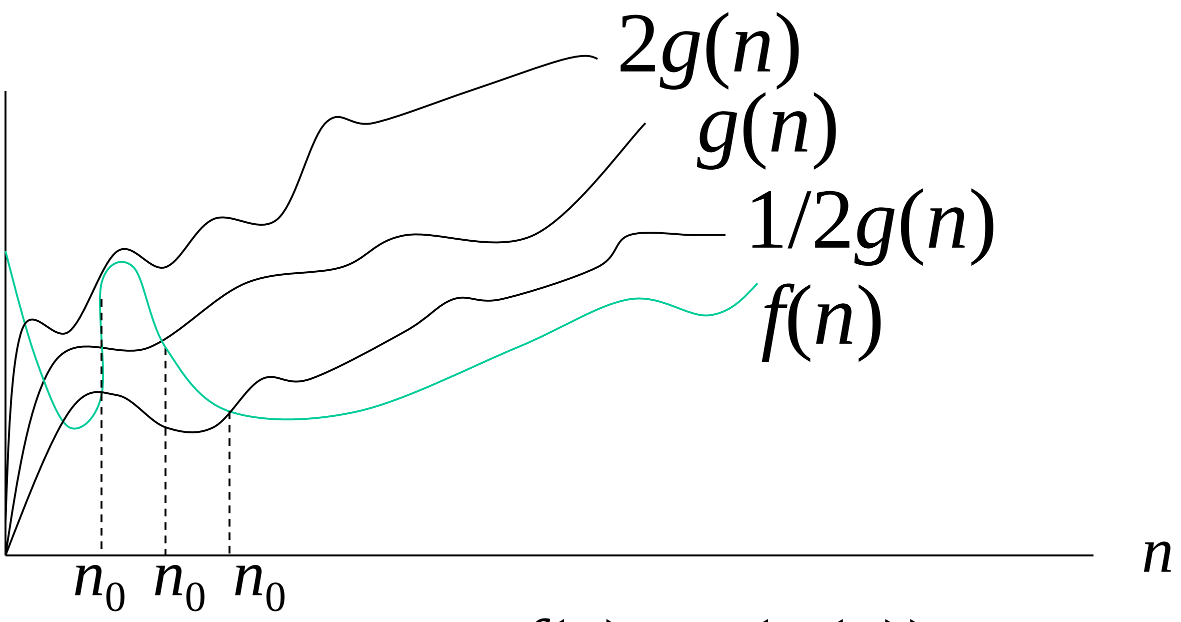
$$f(n) = \Omega(g(n))$$

Prove $f(n)=an^2+bn+c=\Theta(n^2)$

- a, b, c are constants and $a>0$.
- Find c_1, and c_2 (and n_0) such that
  - $c_1 n^2 \le f(n) \le c_2 n^2$ for all $n \ge n_0$.
- It turns out: $c_1 = a/4$, $c_2 = 7a/4$ and
  - $n_0 = 2\cdot\max(|b|/a, \mathrm{sqrt}(|c|/a))$
- Here we also can see that lower terms and constant coefficient can be ignored.
- How about $f(n)=an^3+bn^2+cn+d$?

o-notation

- For a given function $g(n)$,
  - $o(g(n))=\{f(n)$: for any positive constant c,there exists a positive n_0 such that $0\leq f(n) \leq cg(n)$ for all $n\geq n_0\}$
  - Write $f(n) \in o(g(n))$, or simply $f(n) = o(g(n))$.



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

Notes on o-notation

- O-notation may or may not be asymptotically tight for upper bound.
  – $2n^2 = O(n^2)$ is tight, but $2n = O(n^2)$ is not tight.
- o-notition is used to denote an upper bound that is not tight.
  – $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.
- Difference: for some positive constant c in O-notation, but all positive constants c in o-notation.
- In o-notation, $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinitely: i.e.,
  – $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$.

ω-notation

- For a given function $g(n)$,
  - $\omega(g(n))=\{f(n)$: for any positive constant c, there exists a positive n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$
  - Write $f(n) \in \omega(g(n))$, or simply $f(n) = \omega(g(n))$.
- ω-notation, similar to o-notation, denotes lower bound that is not asymptotically tight.
  - $n^2/2 = \omega(n)$, but $n^2/2 \ne \omega(n^2)$
- $f(n) = \omega(g(n))$ if and only if $g(n)=o(f(n))$.
- $\lim\limits_{n \to \propto} \dfrac{f(n)}{g(n)} = \propto$

Techniques for Algorithm Design and Analysis

- Data structure: the way to store and organize data.
  - Disjoint sets
  - Balanced search trees (red-black tree, AVL tree, 2-3 tree).
- Design techniques:
  - divide-and-conquer, dynamic programming, prune-and-search, laze evaluation, linear programming, …
- Analysis techniques:
  - Analysis: recurrence, decision tree, adversary argument, amortized analysis,…

NP-complete problem

- Hard problem:
  - Most problems discussed are efficient (poly time)
  - An interesting set of hard problems: NP-complete.
- Why interesting:
  - Not known whether efficient algorithms exist for them.
  - If exist for one, then exist for all.
  - A small change may cause big change.
- Why important:
  - Arise surprisingly often in real world.
  - Not waste time on trying to find an efficient algorithm to get best solution, instead find approximate or near-optimal solution.
- Example: traveling-salesman problem.