

## CS 570 Analysis of Algorithms

**Text:** *Introduction to algorithms*, by T. Cormen, C. Leiserson and R. Rivest, McGraw-Hill.

**Instructor:** Ming-Deh Huang, Sal 314, x-04783, [huang@cs.usc.edu](mailto:huang@cs.usc.edu)

### Grading

1. Homework – 30% (7-10 assignments)
2. Midterm – 30%
3. Final – 40%

Objective: to study important concepts and techniques in the design and analysis of algorithms

Assume familiarity with the following:

- Recursion, math. induction
- basic data structures including lists, stacks, queues, arrays, as well as data structures for graphs and trees (adj. lists etc). (see Ch. 5, 11)
- Big-O notation for analyzing growth of function (see Ch 2).

Part I of the text covers most of these concepts.

## 1. Introduction

(a) Divide and conquer: 2.3.

(b) Heapsort: 6.1 - 6.5.

## 2. Design and analysis techniques

(a) Dynamic programming: Ch. 15.

(b) Greedy algorithms: 16.1-16.3.

(c) Amortized analysis: 17.1 - 17.3.

## 3. Data structures

(a) Binomial heaps: Ch. 19

(b) Fibonacci heaps: 20.1 - 20.4.

## 4. Graph algorithms

(a) Review of elementary graph algorithms:  
22.1 - 22.4.

(b) Minimum spanning trees: 23.1 - 23.2.

(c) Shortest Paths: 24.1-24.3, 25.1-25.3.

5. Number theoretic and randomized algorithms:  
Ch. 31.

6. NP-Completeness: Ch. 34.

7. Approximation algorithms: Ch. 35.

In analyzing an algorithm, assumptions have to be made as to:

1. how the size of an input is measured
2. what counts as a single step (or a single unit of space) in the computation. The time or space needed by an algorithm is expressed as a function of the input size.

For each input size  $n$ , let  $T(n)$  be the most number of steps taken for inputs of size  $n$ .

Then  $T(n)$  defines the *worst case time complexity* of an algorithm.

However, most often we shall be interested in *asymptotic worst case complexity*. That is, the asymptotic growth of  $T(n)$ , the worst case complexity function. — Big-O, Big- $\Theta$  notation

Improvement in the speed of computers increases the importance of (asymptotically) efficient algorithms.

If you have an  $O(n^2)$  algorithm speeding up the computer 100 times does not mean you'll be able to handle inputs of size 100 times what you could handle before.

But if yours is an  $O(n)$  algorithm, then the increase in the problem size that can be handled is proportional.

Make clear the underlying assumptions in complexity analysis.

Example: Sorting a sequence of integers by comparisons

1. input size: the number of integers in the input sequence
2. one step: one comparison

We should be aware that

1. other ops are being ignored
2. the size of integers involved is not being taken into account( so compare(1,2) and compare (10000, 389) both counts as one step.)

## **Divide and Conquer**

1. exemplify the style of our discussion
2. demonstrate interplay with mathematics –  
divide-and-conquer vs recurrence equations

Example – Merge-sort

## Divide-and-conquer approach –

1. divide the problem into two or several sub-problems
2. solve the subproblems (often recursively)
3. combine the solutions to the subproblems in some way into a solution to the original problem

Caution: Such strategy doesn't work all the time.

The sorting problem: Given a sequence of  $n$  numbers, to arrange them in ascending order.

Eg. 4,1,2,8,7,3,6,5 into 1,2,3,4,5,6,7,8

For simplicity we assume that  $n$  is a power of two.

To apply divide-and-conquer,

- Divide the sequence into two halves:  $\langle 4, 1, 2, 8 \rangle$  and  $\langle 7, 3, 6, 5 \rangle$
- Recursively sort the two subsequences:  $\langle 1, 2, 4, 8 \rangle$ ,  $\langle 3, 5, 6, 7 \rangle$
- Combine the two (merge). Because the two subseq are already sorted, this can be done in at most  $n - 1$  comparisons: repeatedly select the larger of the largest of the two subseq. to be output and deleted.

Running time: each number of the first subseq is compared to once only, so merge takes  $O(n)$  time.

Hence we have the recurrence:

$$T(n) = 2T(n/2) + c.n$$

for some constant  $c$ .

$$T(n) = O(n \log n).$$

Multiplying two integers  $x$  and  $y$  by divide-conquer

Assume both of  $N$  bits and  $N = 2^n$ .

size of  $x$ :  $N$  (number of bits)

Write

$$x = a2^{\frac{N}{2}} + b$$

$$y = c2^{\frac{N}{2}} + d$$

$$xy = ac2^N + (ad + bc)2^{\frac{N}{2}} + bd$$

$a, b, c, d$  all of  $N/2$  bits.

Mult:  $ac, bd, ad, bc$

$$T(N) = 4T(N/2) + cN$$

for some constant  $c$

$$T(N) = O(N^2)$$

Improvement: reduce the number of subtasks from 4 to something smaller

Three mult. suffices:

$$\text{Mult: } (a + b)(c + d) = (ad + bc) + ac + bd$$

$$\text{Mult: } ac$$

$$\text{Mult: } bd$$

$$T(N) = 3T(N/2) + cN$$

for some constant  $c$

$$T(N) = O(N^{\log_2 3})$$

Remark: Best asymp. algorithm for integer mult. (due to Schonhage-Strassen) uses  $O(n \log n \log \log n)$  bit ops. (See Sec. 7.5 of the book "The Design and Analysis of Computer Algorithms", by Aho, Hopcroft and Ullman, Addison-Wesley)

## Heapsort

A heap: an array  $A[1, \dots, n]$  that satisfies the *heap property* –  $A[i] \leq A[2i]$  and  $A[i] \leq A[2i + 1]$ .

Picture the array as a binary tree rooted at  $A[1]$ , where  $A[2i]$  and  $A[2i + 1]$  are the two children of  $A[i]$ .

Heapify at  $i$  – making  $A[i, \dots, n]$  satisfy the heap property, assuming the property is already satisfied at  $2i$  and  $2i + 1$ .

1. Determine  $j \in \{i, 2i, 2i + 1\}$  so that  $A[j]$  is the min. among  $A[i]$ ,  $A[2i]$  and  $A[2i + 1]$ . Swap the elements in  $A[j]$  and  $A[i]$ .
2. If  $j \neq i$  then recursively heapify at  $j$ . (Note: the heap property is satisfied except possibly at  $j$ )

To turn an array  $A[1, \dots, n]$  into a heap.

Note:  $A[1 + n/2], \dots, A[n]$  are leaves, so heap property is satisfied at these locations.

From  $i = n/2$  down to 1, Heapify-at[ $i$ ].

Time for building a heap:  $O(n)$

Number of nodes of height  $h$  is  $O(\frac{n}{2^h})$ .

Cost of Heapify at these nodes is  $O(h)$ .

$$\sum_h \frac{nh}{2^h} = n \sum_h \frac{h}{2^h} = O(n)$$

Heapsort  $A[1, \dots, n]$

Turn  $A[1, \dots, n]$  into a heap.

$j \leftarrow n;$

While  $j \geq 1$  begin

1. Output  $A[1]$ .
2. Move  $A[n]$  to  $A[1]$ .
3. Heapify at  $A[1]$ .

end

$O(n \log n)$  time.

In-place sorting :  $O(1)$  extra space needed.