CSC 8301- Design and Analysis of Algorithms

Lecture 6

Divide and Conquer
Algorithm Design Technique

Divide-and-Conquer

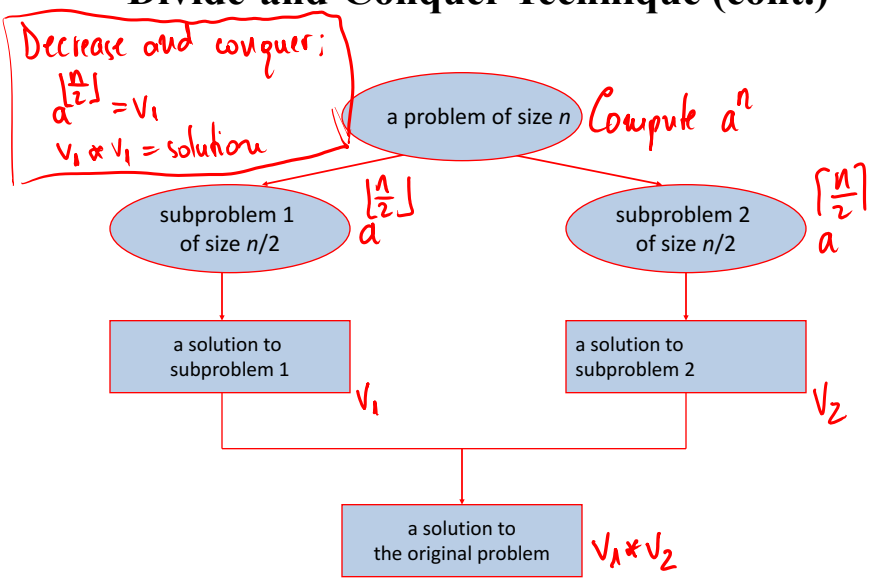
The most-well known algorithm design strategy:

1. Divide a problem instance into two or more smaller instances (ideally of about the same size)

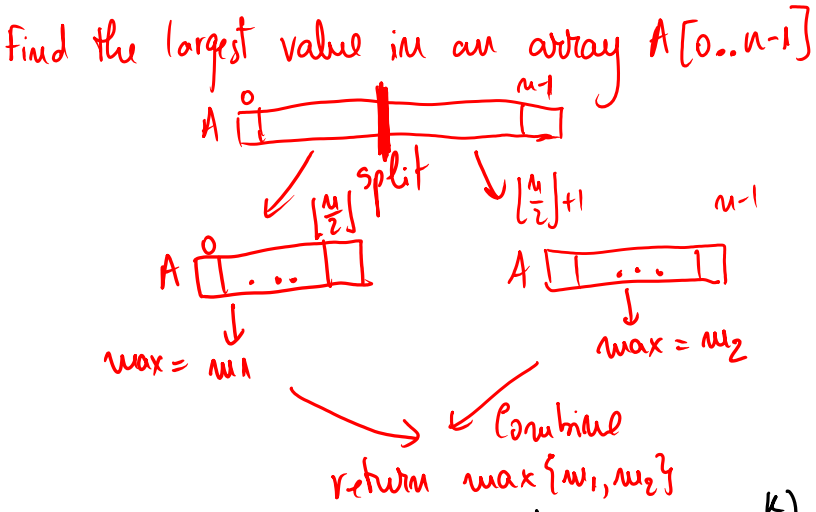
2. Solve the smaller instances (usually recursively)

3. Obtain a solution to the original instance by combining these solutions to the smaller instances

Divide-and-Conquer Technique (cont.)



Divide-and-Conquer Examples



Time Complexity – recurrence formula (assume $n = 2^k$)

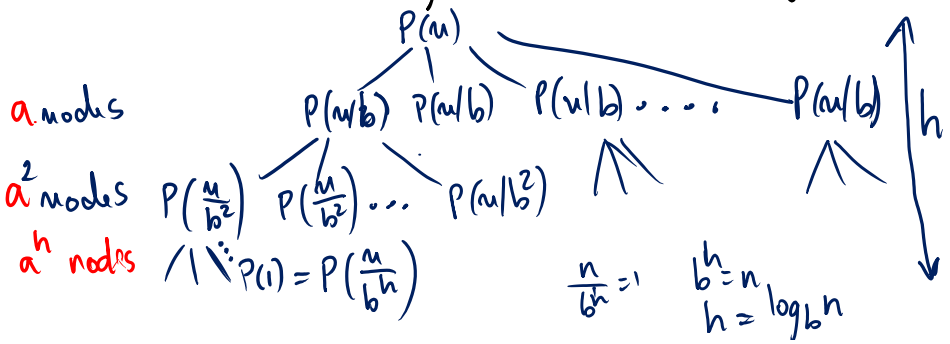
$$C(n) = 2C\left(\frac{n}{2}\right) + 1$$

General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

leaves = a^h
$= a^{\log_b n}$
$= n^{\log_b a}$

n = size of the problem
a = constant = number of subproblems
n/b = size of the subproblems
f(n) = cost of dividing the problem and merging solutions



General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$ # leaves = $n^{\log_b a}$

Master Theorem:
- If $a < b^d$, $T(n) \in \Theta(n^d)$
- If $a = b^d$, $T(n) \in \Theta(n^d \log n)$
- If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

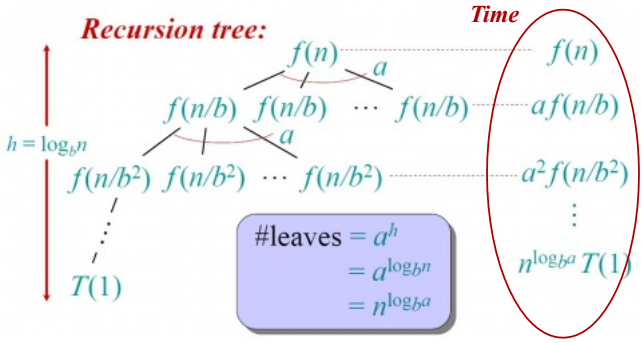
Note: The same results hold with O instead of Θ

Examples:
- $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$ — $a=4, b=2, d=1$; $a=4 > b^d = 2$; $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$
- $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$ — $\Theta(n^2 \log n)$
- $T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$ — $\Theta(n^3)$

Master Theorem – Recursion Tree

$T(n) = aT(n/b) + f(n)$

Visualize this as a recursion tree (branch factor a):



Total time depends on how fast $f(n)$ grows compared with the number of leaves (d compared with $\log_b a$)

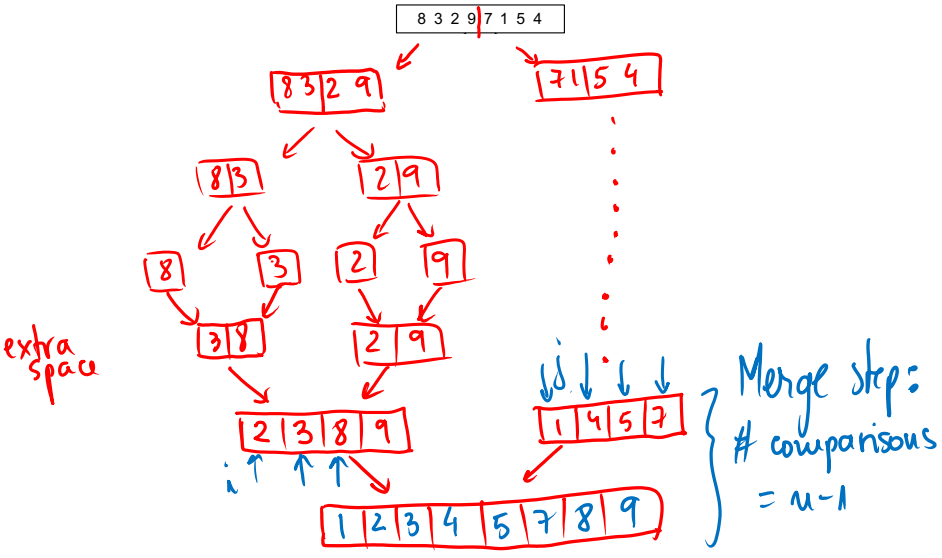
Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Binary search (?)
- Multiplication of large integers
- Closest-pair algorithm

Mergesort

- Split array A[0..*n*-1] in two about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat 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



Merging of Two Sorted Arrays

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$
//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
  if $B[i] \le C[j]$
    $A[k] \leftarrow B[i]$; $i \leftarrow i+1$
  else $A[k] \leftarrow C[j]$; $j \leftarrow j+1$
  $k \leftarrow k+1$
if $i = p$
  copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Analysis of Mergesort

- Recurrence for number of comparisons in the worst case:

  $C(n) = 2\,C(n/2) + C_{merge}(n) = 2\,C(n/2) + n\text{-}1$

  $C(1) = 0$

Master Theorem:
$a = b = 2,\ d = 1 = \log_b a$
$C(n) = \Theta(n \log n)$

Solve exactly ($n = 2^k$)

$C(n) = 2C\left(\frac{n}{2}\right) + n - 1$

$= 2\left[2C\left(\frac{n}{2^2}\right) + \frac{n}{2} - 1\right] + n - 1$

$= 2^2 C\left(\frac{n}{2^2}\right) + (n-2) + (n-1)$

$= 2^2\left[2C\left(\frac{n}{2^3}\right) + \frac{n}{2^2} - 1\right] + (n-2) + (n-1)$

$= 2^3 C\left(\frac{n}{2^3}\right) + (n-2^2) + (n-2) + (n-1)$

$= (n-2^{k-1}) + (n-2^{k-2}) + \ldots + (n-1) = k*n - (2^k - 1)$

$= n\log_2 n - (n-1)$

$= \Theta(n \log n)$

Analysis of Mergesort

- Recurrence for number of comparisons in the worst case:

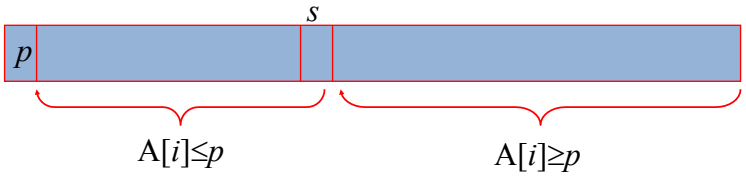
  $C(n) = 2\,C(n/2) + C_{merge}(n) = 2\,C(n/2) + n\text{-}1$
  $C(1) = 0$

  $\Theta(n \log n)$

- All cases have same efficiency: $\Theta(n \log n)$
- Space requirement: $\Theta(n)$ (not in-place)
- Can be implemented without recursion (bottom-up)

Quicksort

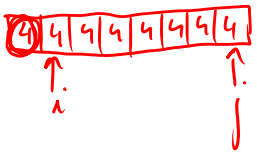
- Select a *pivot* (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first *s* positions are smaller than or equal to the pivot and all the elements in the remaining *n-s* positions are larger than or equal to the pivot (see next slide for an algorithm)



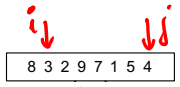
- Exchange the pivot with the last element in the first (i.e., ≤) subarray — *the pivot is now in its final position*
- Sort the two subarrays recursively

Two-Way (Hoar's) Partitioning 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
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r+1$
repeat
  repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
  repeat $j \leftarrow j-1$ **until** $A[j] \leq p$
  swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$ → A[l] will be in its final position
return j



Quicksort Example



Split stage:

8 3 2 4 7 1 5 9 swap(8,5)

new pivot ← 5 3 2 4 7 1 8 9

recurse

5 2 3 4 1 7 8 9 swap(5,1)

1 2 3 4 5 7 8 9

Analysis of Quicksort

- Best-case time efficiency: split in the middle — $\Theta(n \log n)$

How many comparisons?

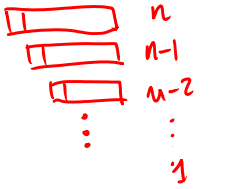
$$C_{best}(n) = 2C_{best}\left(\frac{n}{2}\right) + O(n)$$

- Worst-case time efficiency: sorted array! — $\Theta(n^2)$

All splits skewed to one extreme:

$$C_{worst}(n) = C_{worst}(n-1) + n$$

$$= (n) + (n-1) + \ldots + 1 = \Theta(n^2)$$



- Average case time efficiency: random arrays — $\Theta(n \log n)$

Analysis of Quicksort

- Not stable
- Space efficiency: $\Theta(\log n)$ on average (recursive calls)

- Improvements:
  - better pivot selection: median-of-three partitioning
  - stop recursive calls when unsorted subarrays become small (say, <10 elements) and finish sorting with insertion sort

  These yields about 20% improvement

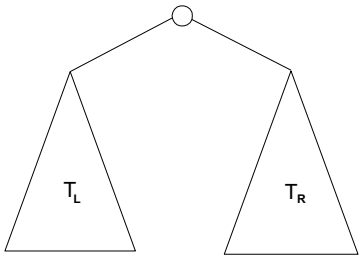
- Considered the method of choice for sorting random files of nontrivial sizes

Binary Tree Algorithms

Binary tree is a divide-and-conquer ready structure!

Example 1: Classic traversals

- *Preorder*: root, left, right
- *Inorder*: left, root, right
- Postorder: left, right, root



Algorithm *Inorder*(*T*)

if $T \neq \varnothing$

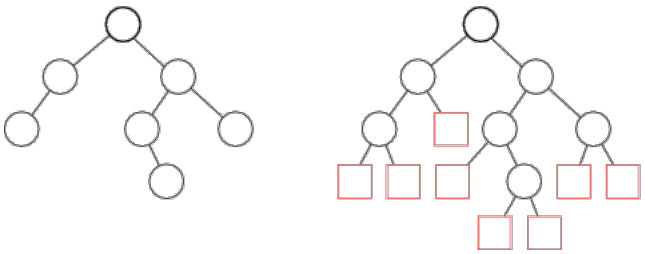
Inorder(T_{left})

print(root of *T*)

Inorder(T_{right})

Extended Binary Tree

- Replace every null subtree of the original tree with special extra nodes (called *extended* or *external*)

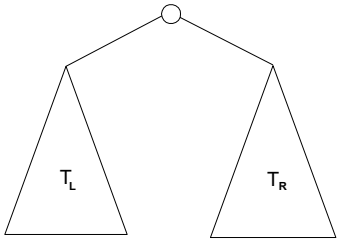


- How many extended nodes in a tree with *n* (original) nodes?

Binary Tree Algorithms (cont.)

Example 2: Computing the height of a binary tree

- The height is the length of the longest path (counting edges) on the way from the root to a leaf
- The height of a single node is 0



$h(T) = \max\{h(T_L), h(T_R)\} + 1$ **if** $T \neq \varnothing$ **and** $h(\varnothing) = -1$

Efficiency: $\Theta(n)$

Multiplication of Large Integers

Consider the problem of multiplying two (large) n-digit integers represented by arrays of their digits such as:

A = 12345678901357986429 B = 87654321284820912836

The grade-school algorithm:

$$
\begin{array}{rl}
 & a_1\ a_2 \ldots\ a_n \\
 & b_1\ b_2 \ldots\ b_n \\
(d_{10}) & d_{11} d_{12} \ldots\ d_{1n} \\
(d_{20}) & d_{21} d_{22} \ldots\ d_{2n} \\
 & \ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots \\
(d_{n0}) & d_{n1} d_{n2} \ldots\ d_{nn}
\end{array}
$$

Efficiency: n^2 one-digit multiplications

First Divide-and-Conquer Algorithm

A small example: $A * B$ where $A = 2135$ and $B = 4014$

$A = (21 \cdot 10^2 + 35)$, $B = (40 \cdot 10^2 + 14)$

So, $A * B = (21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14)$

$= 21 * 40 \cdot 10^4 + (21 * 14 + 35 * 40) \cdot 10^2 + 35 * 14$

In general, if $A = A_1A_2$ and $B = B_1B_2$ (where A and B are n-digit, A_1, A_2, B_1, B_2 are $n/2$-digit numbers),

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

Recurrence for the number of one-digit multiplications $M(n)$:

$$M(n) = 4M(n/2), \quad M(1) = 1$$

Solution: $M(n) = n^2$

Second Divide-and-Conquer Algorithm

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

The idea is to decrease the number of multiplications from 4 to 3:

$$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (\mathbf{A_1 * B_2 + A_2 * B_1}) + A_2 * B_2,$$

I.e., $(\mathbf{A_1 * B_2 + A_2 * B_1}) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2$, which requires only 3 multiplications at the expense of (4-1) extra add/sub.

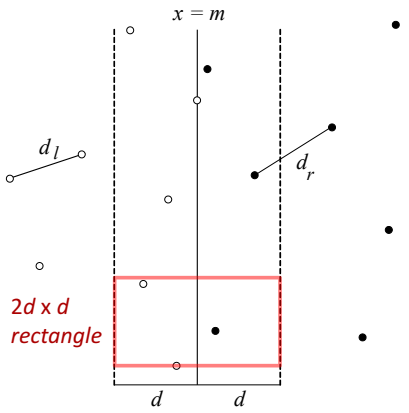
Recurrence for the number of multiplications $M(n)$:

$$M(n) = 3M(n/2), \quad M(1) = 1$$

Solution: $M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$

Closest-Pair Problem by Divide-and-Conquer

Step 1 Divide the points given into two subsets P_l and P_r by a vertical line $x = m$ so that half the points lie to the left or on the line and half the points lie to the right or on the line.



$d = \min\{d_l, d_r\}$

Closest Pair by Divide-and-Conquer (cont.)

Step 2 Find recursively the closest pairs for the left and right subsets.

Step 3 Set $d = \min\{d_l, d_r\}$

We can limit our attention to the points in the symmetric vertical strip S of width $2d$ as possible closest pair. (The points are stored and processed in increasing order of their y coordinates.)

Step 4 Scan the points in the vertical strip S from the lowest up. For every point $p(x,y)$ in the strip, inspect points in in the strip that may be closer to p than d. There can be no more than 5 such points following p on the strip list!

Efficiency of the Closest-Pair Algorithm

Recurrence for the Running time of the algorithm

$$T(n) = 2T(n/2) + M(n), \text{ where } M(n) \in O(n)$$

By the Master Theorem (with $a = 2$, $b = 2$, $d = 1$)

$$T(n) \in O(n \log n)$$

Homework

Reading: Chapter 5, Appendix B (pp. 487-491)

Exercises:

- 5.1: 1, 2, 3, 6, 8
- 5.2: 1, 7, 8, 9
- 5.3: 1, 2, 5, 8
- 5.4: 2, 3
- 5.5: 2

Next: Transform and Conquer (Ch. 6)