Topic 3

Divide-and-Conquer

Divide-and-Conquer

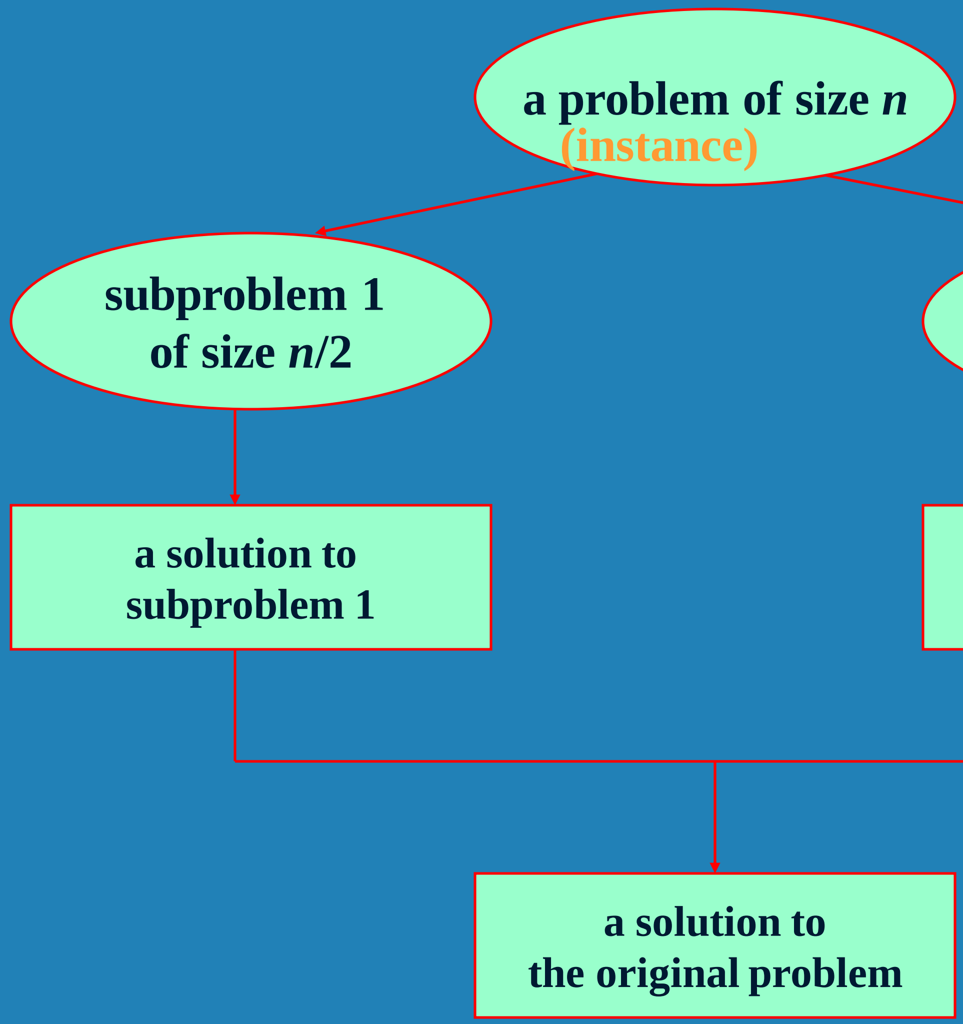
The most-well known algorithm design strategy:

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 (cont.)



It typically leads to a recursive algorithm!

Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Binary search (?)
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Closest-pair and convex-hull algorithms

General Divide-and-Conquer Recurrence

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

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: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$ $\Theta(n\text{^}2)$

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$ $\Theta(n\text{^}2\log n)$

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$ $\Theta(n\text{^}3)$

Binary Search

<u>Binary search</u> is a remarkably efficient algorithm for searching *in a sorted array*.

$$\underbrace{A[0]\dots A[m-1]}_{\substack{\text{search here if}\\ K<A[m]}}\ \ \overset{\substack{K\\ \updownarrow}}{A[m]}\ \ \underbrace{A[m+1]\dots A[n-1]}_{\substack{\text{search here if}\\ K>A[m]}}.$$

As an example, let us apply binary search to searching for $K = 70$ in the array

3	14	27	31	39	42	55	70	74	81	85	93	98

The iterations of the algorithm are given in the following table:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	l						m						r
iteration 2								l		m			r
iteration 3								l,m	r				

Binary Search

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

//Implements nonrecursive binary search
//Input: An array $A[0..n-1]$ sorted in ascending order and
// a search key K
//Output: An index of the array's element that is equal to K
// or -1 if there is no such element
$l \leftarrow 0;\quad r \leftarrow n-1$
while $l \le r$ **do**
  $m \leftarrow \lfloor (l+r)/2 \rfloor$
  if $K = A[m]$ **return** m
  else if $K < A[m]$ $r \leftarrow m-1$
  else $l \leftarrow m+1$
return -1

Binary Search

Number of key comparisons in The worst-case inputs include all arrays that do not contain a given search key.

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 1.$$

For $n=2^k$, and $C_{worst}(1)=1$:

$$C_{worst}(2^k) = k + 1 = \log_2 n + 1.$$

It can be tweaked to get solution valid for arbitrary integer n:

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil.$$

It implies that the worst-case time efficiency is in $\Theta(\log n)$.

It will take no more than $\lceil \log_2(10^6+1) \rceil = 20$ comparisons to do it for any sorted array of size one million!

Analysis of Binary Search

- Optimal for searching a sorted array
- Limitations: must be a sorted array (not linked list)
- Bad (degenerate) example of divide-and-conquer because only one of the sub-instances is solved
- Has a continuous counterpart called *bisection method* for solving equations in one unknown $f(x) = 0$ (see Sec. 12.4)

Binary Tree Algorithms

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

Ex. 1: Classic traversals (preorder, inorder, postorder)

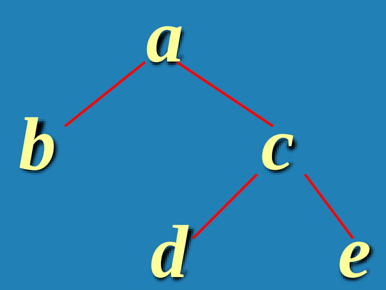
Algorithm *Inorder*(T)

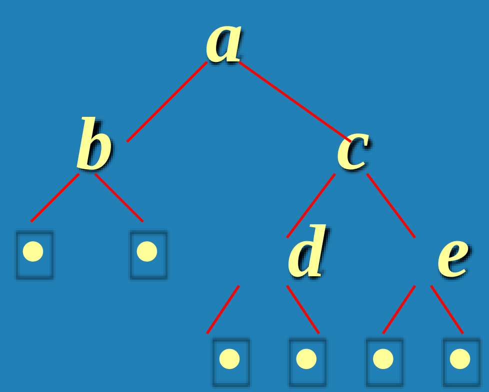
if $T \neq \varnothing$

  Inorder(T_{left})

  print(root of T)

  Inorder(T_{right})





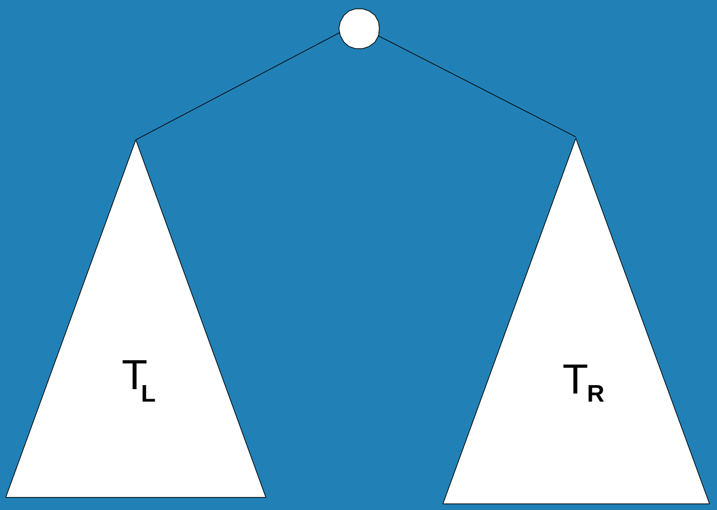
Traversal of Inorder is: b,a,d,c,e

Efficiency: $\Theta(n)$. Why?

Each node is visited/printed once.

Binary Tree Algorithms (cont.)

Ex. 2: Computing the height of a binary tree



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

Efficiency: $\Theta(n)$. Why?

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:

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

Efficiency: $\Theta(n^2)$ single-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:

$$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) + A_2 * B_2,$$

i.e., $(A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2,$ 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: $M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$

What if we count both multiplications and additions?

Example of Large-Integer Multiplication

2135 ∗ 4014

= (21*10^2 + 35) * (40*10^2 + 14)

= (21*40)*10^4 + c1*10^2 + 35*14

where c1 = (21+35)*(40+14) - 21*40 - 35*14, and

21*40 = (2*10 + 1) * (4*10 + 0)

= (2*4)*10^2 + c2*10 + 1*0

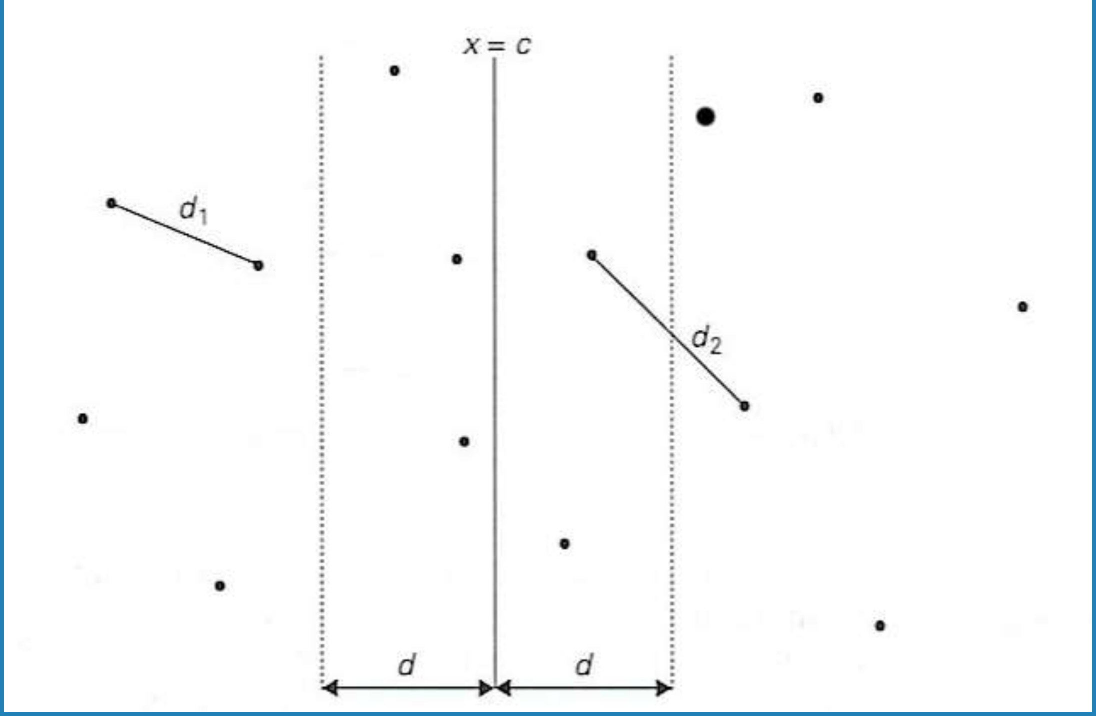
where c2 = (2+1)*(4+0) - 2*4 - 1*0, etc.

This process requires 9 digit multiplications as opposed to 16.

Closest-Pair Problem by Divide-and-Conquer

Step 0 Sort the points by x (list one) and then by y (list two).

Step 1 Divide the points given into two subsets S_1 and S_2 by a vertical line $x = c$ 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.



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

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

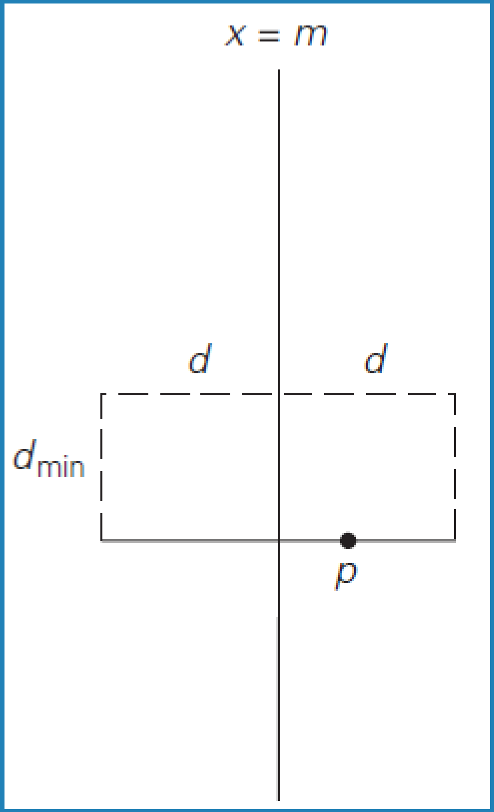
Step 3 Set $d = \min\{d_1, d_2\}$

We can limit our attention to the points in the symmetric vertical strip of width $2d$ as possible closest pair. Let C_1 and C_2 be the subsets of points in the left subset S_1 and of the right subset S_2, respectively, that lie in this vertical strip. The points in C_1 and C_2 are stored in increasing order of their y coordinates, taken from the second list.

Step 4 For every point $P(x,y)$ in C_1, we inspect points in C_2 that may be closer to P than d. There can be no more than 6 such points (because $d \le d_2$)!

Closest Pair by Divide-and-Conquer: Worst Case

Rectangle that may contain points closer than *d* to point *p*:



Closest Pair by Divide-and-Conquer: Algorithm

ALGORITHM *EfficientClosestPair*(P, Q)

//Solves the closest-pair problem by divide-and-conquer
//Input: An array P of $n \geq 2$ points in the Cartesian plane sorted in
// nondecreasing order of their x coordinates and an array Q of the
// same points sorted in nondecreasing order of the y coordinates
//Output: Euclidean distance between the closest pair of points

if $n \leq 3$
  return the minimal distance found by the brute-force algorithm
else
  copy the first $\lceil n/2 \rceil$ points of P to array P_l
  copy the same $\lceil n/2 \rceil$ points from Q to array Q_l
  copy the remaining $\lfloor n/2 \rfloor$ points of P to array P_r
  copy the same $\lfloor n/2 \rfloor$ points from Q to array Q_r
  $d_l \leftarrow EfficientClosestPair(P_l, Q_l)$
  $d_r \leftarrow EfficientClosestPair(P_r, Q_r)$
  $d \leftarrow \min\{d_l, d_r\}$
  $m \leftarrow P[\lceil n/2 \rceil - 1].x$
  copy all the points of Q for which $|x - m| < d$ into array $S[0..num - 1]$
  $dminsq \leftarrow d^2$
  for $i \leftarrow 0$ **to** $num - 2$ **do**
    $k \leftarrow i + 1$
    while $k \leq num - 1$ **and** $(S[k].y - S[i].y)^2 < dminsq$
      $dminsq \leftarrow \min((S[k].x - S[i].x)^2 + (S[k].y - S[i].y)^2, dminsq)$
      $k \leftarrow k + 1$
return $sqrt(dminsq)$

Efficiency of the Closest-Pair Algorithm

Running time of the algorithm (without sorting) is:

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

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

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

So the total time is $\Theta(n \log n)$.