Divide-and-Conquer Algorithms and Recurrence Relations

Section 8.3

Section Summary

- Divide-and-Conquer Algorithms
  - Definition
  - Examples
- Divide-and-Conquer Recurrence Relations
  - Form of Rec. Relations
  - Master Theorem (important tool for solving rec. relations)

Divide-and-Conquer Algorithmic Paradigm

Definition: A *divide-and-conquer algorithm* works by first *dividing* a problem recursively into smaller size problems and then *conquering* the original problem using the solutions of the smaller problems.

Examples:

- Binary search
- Merge sort

Divide-and-Conquer Recurrence Relations

- $f(n)$ represents the number of operations to solve a problem of size n
- Suppose that a recursive algorithm divides a problem of size n into a subproblems, each subproblem is of size n/b.
- Suppose $g(n)$ extra operations are needed in the conquer step.
- Then $f(n)$ satisfies the following recurrence relation:

$$\boldsymbol{f(n) = af(n/b) + g(n)}$$

- This is called a ***divide-and-conquer recurrence relation***.
- *What is the difference with the linear recurrence relations?*

Recursive Binary Search Algorithm

Example: Construct a recursive version of a binary search algorithm.

Solution: Assume we have $a_1, a_2, \ldots, a_n$, an increasing sequence of integers. Initially *low* =1 and *high*= *n*. We are searching for *x*.

procedure *binary search*(*low, high, x :* integers, $1 \leq low \leq high \leq n$)
if (*low*>high) **then**
    return 0
$m := \lfloor (low + high)/2 \rfloor$
if $x == a_m$ **then**
    return *m*
else if $(x < a_m)$ **then**
    return *binary search*(*low*, $m-1$, *x*)
else
    return *binary search*($m+1$, *high*, *x*)

{output is location of *x* in $a_1, a_2, \ldots, a_n$ if it appears, otherwise 0}

Example: Binary Search

- Binary search reduces the search for an element (key) in a list of size *n* to the search in a list of size _?_

  (n, $n-1$, $\lfloor n/2 \rfloor$, $\lfloor n/3 \rfloor$)
- ? operations are needed to implement this reduction (see slide 43 of Lecture 10).
- If $f(n)$ is the number of operations required to search for an element in a list of size n (in the worst case), n is divisible by 2, then

$$f(n) = f(n/2) + 3$$

Recursive Merge Sort

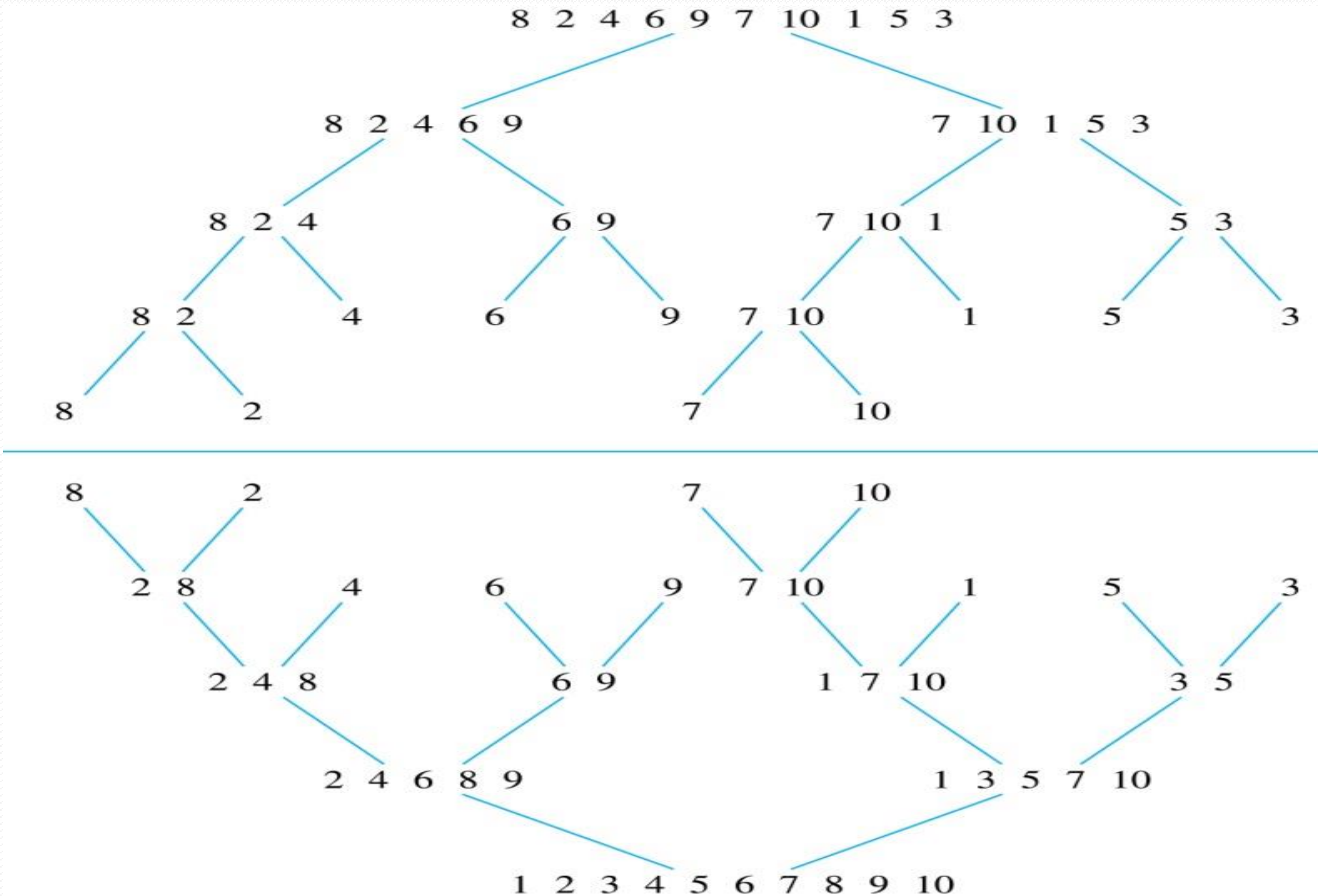
Example: Construct a recursive merge sort algorithm.

Solution: Begin with a list of n elements L.

procedure $mergesort(L = a_1, a_2, \ldots, a_n)$
if $n > 1$ **then**
  $m := \lfloor n/2 \rfloor$
  $L_1 := a_1, a_2, \ldots, a_m$
  $L_2 := a_{m+1}, a_{m+2}, \ldots, a_n$
  $L := merge(mergesort(L_1), mergesort(L_2))$

continued →

Merge Sort



8 2 4 6 9 7 10 1 5 3
8 2 4 6 9
7 10 1 5 3
8 2 4
6 9
7 10 1
5 3
8 2
4
6
9
7 10
1
5
3
8
2
7
10
8
2
7
10
2 8
4
6
9
7 10
1
5
3
2 4 8
6 9
1 7 10
3 5
2 4 6 8 9
1 3 5 7 10
1 2 3 4 5 6 7 8 9 10

Function Merge

- Fuction *merge*, which merges two sorted arrays.

procedure *merge*(L_1, L_2 :sorted lists)
L := empty list
while L_1 and L_2 are both nonempty
    remove smaller of first elements of L_1 and L_2 from its list;
        add it to L
    if this removal makes one list empty
      then remove all elements from the other list and append them to L
return L
{L is the merged list with the elements in increasing order}

Merging Two Lists

Example: Merge the two lists 2,3,5,6 and 1,4.

Solution:

TABLE 1 **Merging the Two Sorted Lists 2, 3, 5, 6 and 1, 4.**

First List	*Second List*	*Merged List*	*Comparison*
2 3 5 6	1 4		$1 < 2$
2 3 5 6	4	1	$2 < 4$
3 5 6	4	1 2	$3 < 4$
5 6	4	1 2 3	$4 < 5$
5 6		1 2 3 4	
		1 2 3 4 5 6	

Merge Sort Recurrence Relation

- The merge sort algorithm splits a list of n (assuming n is even) items to be sorted into two lists with $n/2$ items.
- It uses fewer than n comparisons to merge the two sorted lists.
- Hence, $M(n)$ - the number of comparisons required to sort a list of size n (for even n)

$$M(n) = 2M(n/2) + n$$

Estimating the Functions in Divide-and-Conquer Recurrence Relations

Theorem 1: Let f be an increasing function that satisfies the recurrence relation

$$f(n) = af(n/b) + c$$

whenever n is divisible by b, where $a \geq 1$, b is an integer greater than 1, and c is a positive real number.

Then

$$f(n) \text{ is } \begin{cases} O(n^{\log_b a}) & \text{if} \quad a > 1 \\ O(\log n) & \text{if} \quad a = 1. \end{cases}$$

Furthermore, when $a > 1$ and $n = b^k$, where k is a positive integer,

$$f(n) = C_1 n^{\log_b a} + C_2$$

where $C_1 = f(1) + c/(a-1)$ and $C_1 = -c/(a-1)$.

Big-O Notation

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is read as "$f(x)$ is big-oh of $g(x)$."]

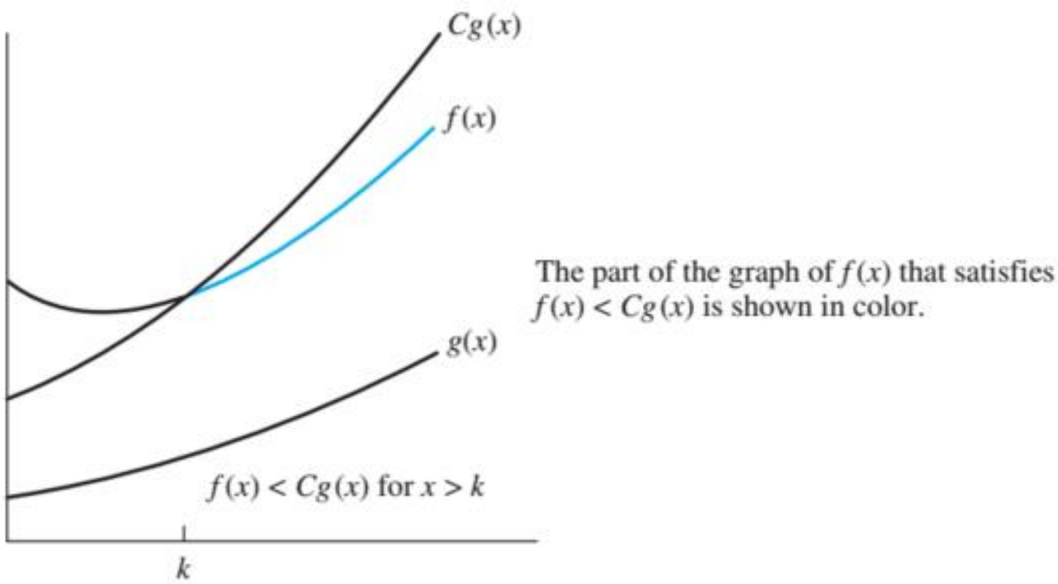


FIGURE 2 **The Function $f(x)$ is $O(g(x))$.**

The term big-O was coined by a German mathematician Paul Bachmann in 1892, and later used by Edmund Landau for number theory, and Donald Knuth for complexity of algorithms.

Match each f(n) with the set O(g(n)) that it belongs to, such that g(n) is the smallest.

Functions

1) $f(n)=C$,
   C – some constant
2) $f(n)= 5n+10$
3) $f(n)= an^2 +bn+c$
4) $f(n)=2n^3 + 3n^2$
5) $f(n)= 5\log_2 n +1$
6) $f(n)= (n+3) \ln n$
7) $f(n)=2^n + n^3 + n^2$

Sets of functions

a) $O(1)$
b) $O(\log n)$
c) $O(n)$
d) $O(n \log n)$
e) $O(n^2)$
f) $O(n^3)$
g) $O(2^n)$

(Time) complexity of an algorithm

- An estimate of the number of operations $f(n)$ used by the algorithm when its input has a particular size n
  - What is n in binary seach, mergesort, matrix multiplication, computing the tree height?
- Complexity is frequently stated using big-O (or other asymptotic notation: Ω, Θ) estimate of $f(n)$.

Complexity of Binary Search

- **Binary Search Example**: Give a big-O estimate for the number of comparisons used by a binary search.
- **Solution**: Since the number of comparisons used by binary search in the worst case is
- $f(n) = f(n/2) + 3$, where n is even,
- by Theorem 1, it follows that $f(n)$ is $O(?)$

a) n

b) $n/2$

c) $n^{\log_2 1}$

d) **log n**

Recursive algorithm for finding min/max of a sequence

rec_min_max (L)

- if n=1 then min=a_1, max=a_1
- else if n>1
  - split L into L1 and L2

  - (min1, max1)=rec_min_max(L1)
  - (min2, max2)=rec_min_max(L2)

  - min = select_min(min1, min2)
  - max = select_max(max1, max2)
- return (min, max)

____

What is the complexity of this algorithm?

- Recurrence relation for number of operations $f(n)$:
  - $f(n) = 2f(n/2) + 2$
- Big-O estimate of $f(n)$: $f(n) \in O(n)$
  - (since $a=b=2$ in Theorem 1, slide 53)

Solving divide-and-conquer recurrence relation of type $f(n) = af(n/b) + c$, $a=1$

Example 1. **Let $f(n) = f(n/2) + 3$ *and* $f(1)=7$.**

Find $f(n)$, where $n=2^k$, k is a positive integer.

- **Solution**: let's use backwards substitution:
- $f(n) = f(n/2) + 3 =$
- $=(f(n/4) + 3) + 3=$
- $=((f(n/8) + 3) +3) +3=\ldots$
- $= f(n/2^k) + 3k =$
- $= f(1)+ 3\log_2 n=$
- $=3\log_2 n+7$

<u>Answer:</u>

$f(n)=3\log_2 n+7$,

$f(n)\in O(\log_2 n)=O(\log n)$

Derive *O()* estimate of *f(n),* true for any *n>1,* where *f(*1*)*=7, *f(n)* is increasing function of *n*, satisfying recurrence relation ***f(n) = f(n/2) + 3*** for any $n=2^k$ (k-integer, k≥1).

- $n>1 \longrightarrow \exists k$ (k-integer, k≥1, $2^k < n \leq 2^{k+1}$)
- $f(n)$ is increasing function of $n \rightarrow f(n) \leq f(2^{k+1})=$ $=3\log_2 2^{k+1}+7$ (see previous slide) =
- $= 3(k+1)+7 = 3k+10 \leq$
- $\leq 3\log_2 n + 10$ (since $2^k < n$)
- Thus, $\boldsymbol{f(n) \in O(log_2 n) = O(\log n)}$

Estimating the Functions in Divide-and-Conquer Recurrence Relations

Theorem 1: Let f be an increasing function that satisfies the recurrence relation

$$f(n) = af(n/b) + c$$

whenever n is divisible by b, where $a \geq 1$, b is an integer greater than 1, and c is a positive real number. Then

$$f(n) \text{ is } \begin{cases} O(n^{\log_b a}) & \text{if} \quad a > 1 \\ O(\log n) & \text{if} \quad a = 1. \end{cases}$$

Furthermore, when $a > 1$ and $n = b^k$, where k is a positive integer,

$$f(n) = C_1 n^{\log_b a} + C_2$$

where $C_1 = f(1) + c/(a-1)$ and $C_1 = -c/(a-1)$.

Proof of Theorem 1 in the case *a=1*

1. Let $f(n) = f(n/b) + c$, $(a=1)$

a) ***Find $f(n)$, where $n=b^k$, k is a positive integer.***

a) Doing backward subsitutions we get as in example 1:

- $f(n) = f(n/b) + c =$
- $=(f(n/b^2) + c) + c=$
- $=((f(n/b^3) + c) + c) + c= \ldots$
- $= f(n/b^k) + ck =$
- $= f(1) + c\log_b n=$
- $=c\log_b n+f(1)$

<u>Answer:</u>

a) $f(n)=c\log_b n+f(1)$,

$f(n) \in O(\log_b n)= O(\log n)$

Continuing proof of Theorem 1 in the case *a=1*

b) Give O-estimate of $f(n)$ for any $n>1$, where $f(n)$ – increasing function, satisfying $f(n) = f(n/b) + c$ for $n=b^k$, k - positive integer.

- $n>1 \rightarrow \exists k$ (k-integer, $k \geq 1$, $b^k < n \leq b^{k+1}$)
- $f(n)$ is increasing function of $n \rightarrow f(n) \leq f(b^{k+1})=$
- $=c\log_b(b^{k+1})+f(1)$ (see previous slide) =
- $= c(k+1)+f(1) =$
- $=ck+ f(1)+c \leq$
- $\leq c\log_b n + f(1)+c$ (since $b^k < n$)
- Thus, $f(n) \in O(log_2 n) = O(\log n)$

Solving divide-and-conquer recurrence relation of type $f(n) = af(n/b) + c,\ a>1$

Example 2. Let $f(n) = 5f(n/2) + 3$ ***and*** $f(1)=7$.

Find $f(n)$***, where*** $n=2^k$***, k is a positive integer.***

- **Solution**: let's use backwards substitution:
- $f(n) = 5f(n/2) + 3 =$
- $=5(5f(n/4)+3) + 3=$
- $=5(5(5f(n/8)+3)+3) + 3=\ldots$
- $= 5^k f(n/2^k) + 3(1+5+5^2\ldots+5^{k-1})=$
- $= 5^k f(1)+ 3(5^k-1)/(5-1)=$
- $= 7\cdot 5^k+ \frac{3}{4}(5^k-1) =$
- $= (7\frac{3}{4})5^k - \frac{3}{4}$

Continue:

- *Since $n=2^k$ we get:*
- $(7\frac{3}{4})5^k - \frac{3}{4} = (7\frac{3}{4})\, 5^{\log_2 n} - \frac{3}{4} =$
- $= (7\frac{3}{4})\, (2^{\log_2 5})^{\log_2 n} - \frac{3}{4} =$
- $= (7\frac{3}{4})\, (2^{\log_2 n})^{\log_2 5} - \frac{3}{4} =$
- $= (7\frac{3}{4})\, n^{\log_2 5} - \frac{3}{4}$.

<u>Answer:</u>

$f(n) = (7\frac{3}{4})\, n^{\log_2 5} - \frac{3}{4}$

$f(n) \in O(n^{\log_2 5})$

Give O-estimate of f(n) for any n>1, where f(n) – increasing function, satisfying f(1)=7, f(n) = 5f(n/2) + 3 for n=2^k , k - positive integer.

- *n>1*→ ∃k (k-integer, k≥1, $2^k < n \le 2^{k+1}$)
- *f(n)* is increasing function of *n* → *f(n)* ≤*f*(2^{k+1})=
- =$(7\frac{3}{4})5^{k+1} - \frac{3}{4}$ (see previous slide) =
- =$(7\frac{3}{4})*5*5^{k} - \frac{3}{4} \le$
- $\le (7\frac{3}{4})*5*5^{\log_2 n} - \frac{3}{4}$ (since $2^k < n$)
- = $(7\frac{3}{4})*5*n^{\log_2 5} - \frac{3}{4}$

(since $5^{\log_2 n} = (2^{\log_2 5})^{\log_2 n} = (2^{\log_2 n})^{\log_2 5} = n^{\log_2 5}$)

- Thus, ***f(n)∈O(***$n^{\log_2 5}$)

Proof of Theorem 1 ($a>1$)

2. Let $f(n) = af(n/b) + c$, $(a>1)$

a) ***Find $f(n)$, where $n=b^k$, k is a positive integer.***

b) ***Give O-estimate of $f(n)$ for any n.***

- $f(n) = af(n/b) + c =$
- $=a(af(n/b^2)+c) + c=$
- $=a(a(af(n/b^3)+c)+c) + c=\ldots$
- $= a^k f(n/b^k) + c(1+a+a^2\ldots+a^{k-1})=$
- $= a^k f(1)+ c(a^k-1)/(a-1)=$
- $= (f(1)+\frac{c}{a-1})\, a^k - \frac{c}{a-1}$

Continue:

- *Since n=b^k we get:*
- $a^k = a^{\log_2 n} =$
- $=(b^{\log_b a})^{\log_b n} =$
- $=(b^{\log_b n})^{\log_b a} =$
- $=n^{\log_b a}$
- Substitute into the expression on the previous slide and get an answer:

a) $f(n)=(f(1)+\frac{c}{a-1})\, n^{\log_b a} - \frac{c}{a-1}$.

$f(n)\in O(n^{\log_b a})$

b) Give O-estimate of f(n) for any n>1, where f(n) – increasing function, satisfying f(n) = af(n/b) + c for n=b^k, k - positive integer.

- $n>1 \rightarrow \exists k$ (k-integer, $k \geq 1$, $b^k < n \leq b^{k+1}$)
- $f(n)$ is increasing function of $n \rightarrow f(n) \leq f(b^{k+1})=$
- $=(f(1)+\frac{c}{a-1})\, a^{k+1}-\frac{c}{a-1}$(see previous slide) =
- $=(f(1)+\frac{c}{a-1})*a*a^{k}-\frac{c}{a-1}\leq$
- $\leq (f(1)+\frac{c}{a-1})*a*a^{\log_b n}-\frac{c}{a-1}$(since $b^{k}<n$)
- $= (f(1)+\frac{c}{a-1})*a*n^{\log_b a} - \frac{c}{a-1}$

(since $a^{\log_b n}=(b^{\log_b a})^{\log_b n}=(b^{\log_b n})^{\log_b a}=n^{\log_b a}$)

- Thus, $\boldsymbol{f(n) \in O(n^{\log_b a})}$

Complexity of Merge Sort

Since most frequent operations in merge sort are comparisons – let's give a big-O estimate for the number of comparisons used by merge sort.

- Recurrence relation:
  - $f(n) = 2f(n/2) + n$
- Does it satisfy conditions of Theorem 1?
  - No, since n is not a constant.

Estimating the Size of Divide-and-conquer Functions (*continued*)

Theorem 2. Master Theorem: Let f be an increasing function that satisfies the recurrence relation

$$f(n) = af(n/b) + cn^d$$

whenever $n = b^k$, where k is a positive integer greater than 1, and c and d are real numbers with c positive and d nonnegative. Then

$$f(n) \text{ is } \begin{cases} O(n^d) & \text{if} \quad a < b^d, \\ O(n^d \log n) & \text{if} \quad a = b^d, \\ O(n^{\log_b a}) & \text{if} \quad a > b^d. \end{cases}$$

Complexity of Merge Sort

Solution: Since the number of comparisons used by merge sort to sort a list of n elements is determined by
$M(n) = 2M(n/2) + n$,
by the master theorem $M(n)$ is

a) $O(\log n)$
b) $O(n)$
c) $O(n \log n)$
d) $O(n^2)$

Closest-Pair Problem by Exhaustive Search

Find the two closest points in a set of *n* points in 2D (given by their (x,y) coordinates).

Algorithm:

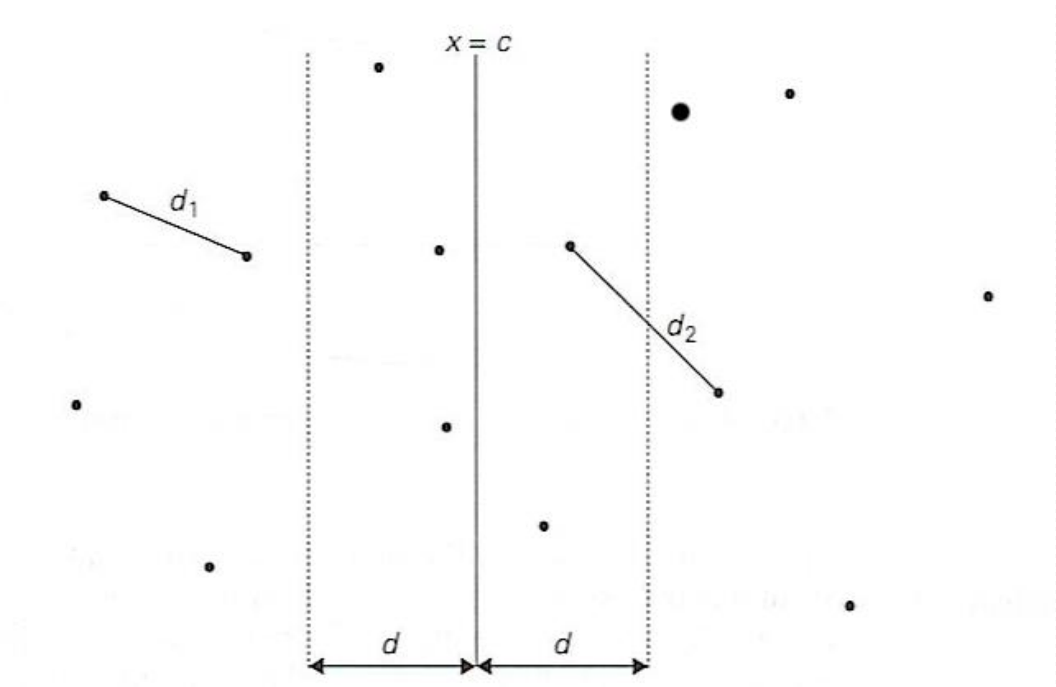
```
d = (x[0] – x[1])² + (y[0] – y[1])²
for(i=0; i < n-2; i++)
        for(j = i + 1; j < n – 1; j++)
                cur_d =(x[i] – x[j])² + (y[i] – y[j])²
                if(cur_d < d)
                        d = cur_d;
                        closestPair =(i , j) ;
return closestPair;
```

What is the asymptotic time efficiency of the algorithm?

Closest-Pair Problem by Divide-and-Conquer

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.

2) Find recursively the closest pairs for the left and right subsets. Set $d = \min\{d_1, d_2\}$



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

$d = \min\{d_1, d_2\}$.

3) For every point $P(x,y)$ in the left d-strip, we inspect points in the right d-strip, that may be closer to P than d.

- These points are located in the rectangle $(0, d) \times (y-d, y+d)$
- distance between each pair of such points $\geq d_2 \geq d$
- there can be at most 6 such points.

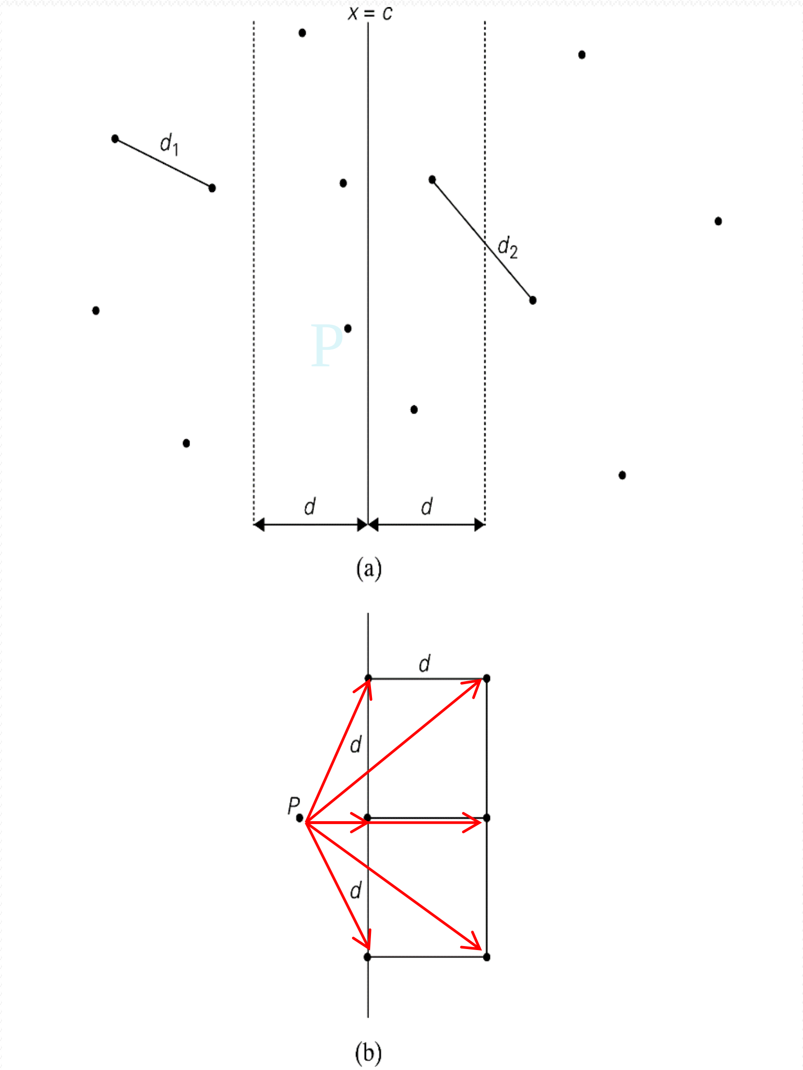


FIGURE 4.7 (a) Idea of the divide-and-conquer algorithm for the closest-pair problem. (b) The six points that may need to be examined for point P.

Pseudocode of Closest Pair Problem

ClosestPair(A[]-array of points in 2d):

if (number of points==2)
(a,b)= (A[0], A[1])
else if (number of points==3)
select closest pair out of 3 pairs
else if (number of points>3) then
Partition(A[], A_{left}[], A_{right}[])
(p,q)=ClosestPair(A_{left} [])
(r, s) =ClosestPair(A_{right} [])
(a,b)=Select((p,q),(r,s), left d-strip, right d-strip)
return (a,b)

Complexity of the Closest-Pair Algorithm

Running time of the algorithm is described by

$$f(n) = 2f(n/2) + 6n,$$

Applying recursion tree method or Master method we get: $f(n) \in \Theta(n \log n)$.

Matrix Multiplication.

We start from the standard algorithm for matrix multiplication:

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

The triple nested loop clearly implies a time $\Theta(n^3)$, which is not quite as bad as it looks since we are dealing with n^2 elements, so $n^3 = (n^2)^{1.5}$.

If we are going to try some clever Divide & Conquer scheme, we could start by coming up with a non-clever one... Here it is: in is a power of 2, we can always subdivide an n-by-n matrix into 4 n/2-by-n/2 ones:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{4.9}$$

so that we rewrite the equation $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}. \tag{4.10}$$

Equation (4.10) corresponds to the four equations

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \tag{4.11}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \tag{4.12}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \tag{4.13}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \tag{4.14}$$

Each of these four equations specifies two multiplications of $n/2 \times n/2$ matrices and the addition of their $n/2 \times n/2$ products. We can use these equations to create a straightforward, recursive, divide-and-conquer algorithm:

The Algorithm:

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
    n = A.rows
    let C be a new n × n matrix
    if n == 1
        c11 = a11 · b11
    else partition A, B, and C as in equations (4.9)
        C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
        C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
        C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
        C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
    return C
```

A pretty immediate conclusion is that multiplying 2 n-by-n matrices involves 8 multiplications of n/2-by-n/2 matrices and 4 additions of n/2-by-n/2 matrices.

Adding 2 n-by-n matrices cost $\Theta(n^2)$, and the total cost of the 4 additions remains $\Theta(n^2)$.

Partitioning of matrices has cost that depends on the method: copying would have a $\Theta(n^2)$ cost (because of the n^2 elements), using indices that provide information on the original array (avoiding copying) could cost as little as $\Theta(1)$.

The recursion relation is: $f(n)=8f(n/2)+n^2$

Applying the master theorem we get:

$f(n) = O(n^3)$,

so we did not reduce time complexity

of the original algorithm for matrix multiplication. ☹

Fast Matrix Multiplication

- One can ask the question: do we need all 8 multiplications or can we find a clever way or coming up with fewer?
- Volker Strassen in 1969 suggested reducing the multiplication of 2 matrices to 7 multiplications of (n/2)x(n/2) matrices and 15 additions of (n/2)x(n/2) matrices.
- Recurrence relation for Strassen algorithm:
  - f(n)=?f(n/?)+?
- Find the complexity of Strassen algorithm (big-O estimate of f(n)).