Divide-and-Conquer

Kuen-Liang Sue
Information Management
NCU

General Plan

- A problem's instance is divided
  - Into several smaller instance
  - Ideally the same size
- The smaller instances are solved
  - Recursively
- Combine solutions obtained for smaller instance
  - to get a solution to the original problem.

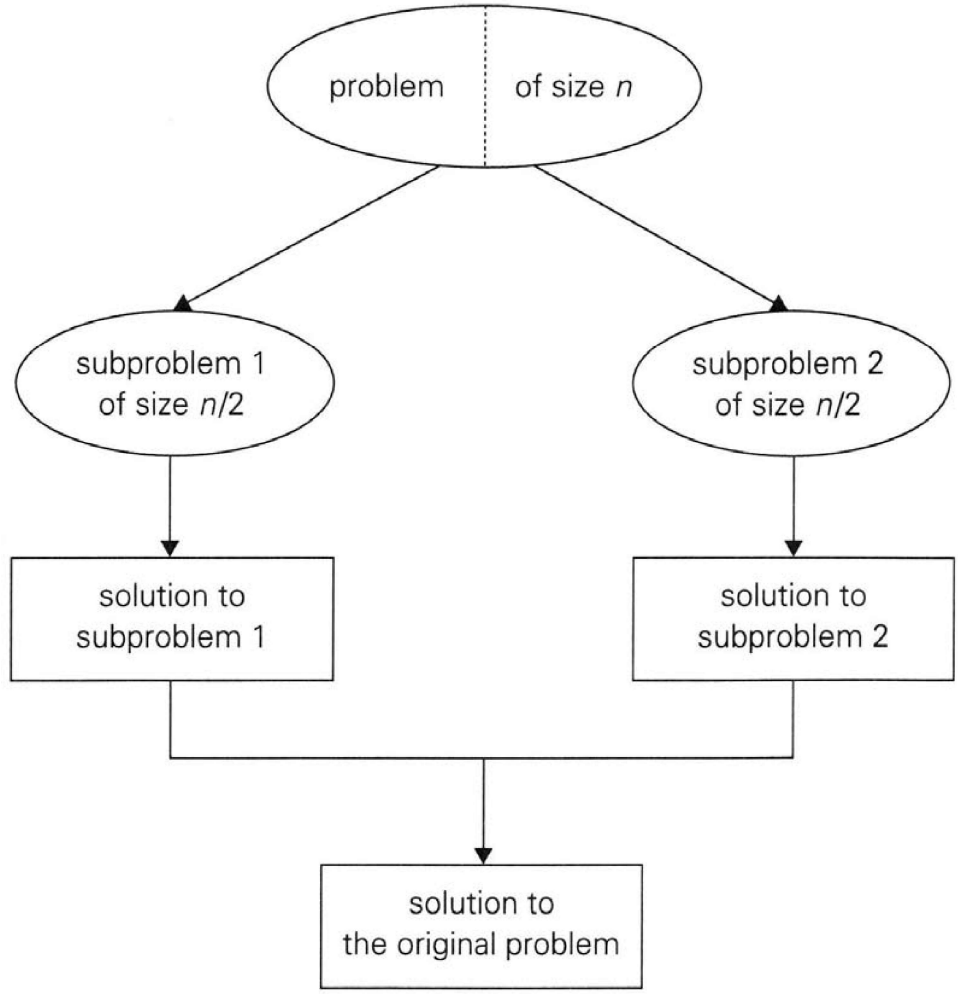


FIGURE 4.1 Divide-and-conquer technique (typical case)

Example: Computing the sum of *n* numbers

- Divide the problem into two instances
  - If n>1, Compute the sum of the
    - First $\lfloor n/2 \rfloor$ numbers
    - Remaining $\lceil n/2 \rceil$ numbers
- Each of these two sums is computed
  - By applying the same method, i.e., recursively
  - Then, get the sum in question

$$a_0 + \cdots + a_{n-1} = \left(a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}\right) + \left(a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1}\right)$$

General divide-and-conquer recurrence

- An instance of size n can be divided into
  - Several instance of size *n/b*
  - Totally, a of them needing to be solved

- Assume size *n* is a power of *b*
  - Recurrence for the running time *T(n):*

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

General divide-and-conquer recurrence

- *f(n):* accounts for the time spent on
  - Dividing the problem into smaller ones
  - Combining their solutions
- For summation example
  - *a=b=2*
  - *f(n)=1*

- The order of growth of *T(n)* depends on
  - The values of the constants *a* and *b*
  - The order of growth of *f(n)*

Efficinecy analysis of divide-and-conquer

- Analysis is simplified by master theorem

MASTER THEOREM If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence equation (4.1), then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

(Analogous results hold for the O and Ω notations, too.)

Efficinecy analysis of divide-and-conquer

- E.g. the summation algorithm
  - *A(n)*: the number of additions
  - Recurrence equation for *A(n):*
    - On inputs of size $n=2^k$

$$A(n) = 2A(n/2) + 1$$

    - For this example, a=2, b=2, and d=0
      - Since $a > b^d$

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$$

Mergesort

- Sort a given array *A*[0..*n*-1]
- Divide it into
  - *A*[0..$\lfloor n/2 \rfloor$-1]
  - *A*[$\lfloor n/2 \rfloor$..*n*-1]
- Sorting each of them recursively
- Merging the two into a single sorted one

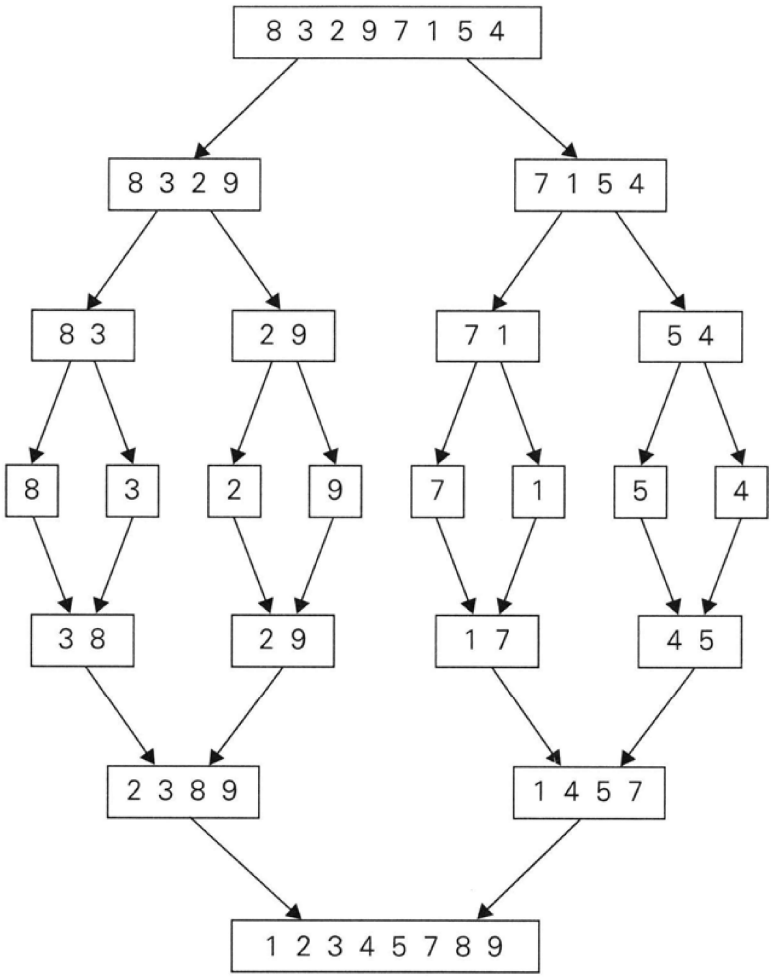


FIGURE 4.2 An example of mergesort operation

Mergesort

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A)

Merging of two sorted arrays

- Steps
  - Two pointers are initialized
    - To point to the first elements of the arrays
  - The elements pointed are compared
    - The smaller is added to a new array
  - The index of that smaller element incremented
    - to point to its immediate successor

Merging of two sorted arrays

- The operation is continued
  - Until one of the two arrays is exhausted
- Remaining elements of the other array
  - Copied to the end of the new array

Merge

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 ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i + 1
    else A[k] ← C[j]; j ← j + 1
    k ← 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]
```

How efficient is mergesort?

Assume *n* is a power of 2

- *C(n):* the number of key comparisons

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

- $C_{merge}(n)$
  - the number of key comparisons performed during the merging stage

How efficient is mergesort?

- Analysis of $C_{merge}(n)$
  - At each step
    - One comparison is made
    - Elements needed to be processed is reduced by one
  - In the worst case
    - Neither of the two arrays becomes empty
      - before the other one contains just one elements

How efficient is mergesort?

- For the worst case, $C_{merge}(n)=n-1$

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \text{ for } n > 1, C_{worst}(1) = 0$$

- According to Master Theorem

$$C_{worst}(n) \in \Theta(n \log n)$$

- In fact. the exact solution for $n=2^k$:

$$C_{worst}(n) = n \log_2 n - n + 1$$

How efficient is mergesort?

- Note
  - $C_{worst}(n)$ come very close to
    - Theoretical minimum of comparison-based sorting

$$\lceil \log_2 n! \rceil \approx \lceil n \log_2 n - 1.44n \rceil$$

  - What is the shortcoming of mergesort?
    - The linear amount of extra storage

Quicksort

- Features
  - Divide elements according to their value
  - Rearrange elements to achieve “Partition”
    - All the elements before position *s* are ≤ *A[s]*
    - All the elements after position *s* are ≥ *A[s]*

$$\underbrace{A[0]\ldots A[s-1]}_{\text{all are} \le A[s]}\ A[s]\ \underbrace{A[s+1]\ldots A[n-1]}_{\text{all are} \ge A[s]}$$

Quicksort

- After a partition
  - A[s] will be in its final position
  - Continue sorting the two subarrays independently
    - The elements preceding A[s]
    - The elements following A[s]
    - by the same method

Quicksort

ALGORITHM *Quicksort*$(A[l..r])$

//Sorts a subarray by quicksort
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right indices
// l and r
//Output: The subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
  $s \leftarrow$ *Partition*$(A[l..r])$ //s is a split position
  Quicksort$(A[l..s-1])$
  Quicksort$(A[s+1..r])$

Quicksort

- A partition can be achieved by
  - Select an element, called pivot
    - Based on whose value, we divide the subarray
    - Select the subarray's first element, *p=A[1]*
  - Rearrange elements to achieve a partition
    - Two scans of the subarray
      - Left-to-right
      - Right-to-left

Quicksort: Rearrange elements to archive a partition

- Step1. Left to right scan
  - Start with the second element
  - Skips over elements < **pivot** *p*
  - Stops on encountering the first element ≥ *p*

Quicksort: Rearrange elements to archive a partition

- Step2. Right to left scan
  - Start with the last element of the subarray
  - Skips over elements > *p*
  - Stops on encountering the first elements ≤ *p*

Quicksort:

- Step3. Three situations may arise
  - Case 1: *i* and *j* have not crossed
    - $i < j$
    - Exchange *A[i]* and *A[j]*
    - Resume the scans
      - Increasing *i* and decrementing *j*

$\rightarrow i$ $\quad\quad\quad\quad\quad\quad$ $j \leftarrow$

p	All are $\leq p$	$\geq p$	· · ·	$\leq p$	All are $\geq p$

Quicksort: Three situations

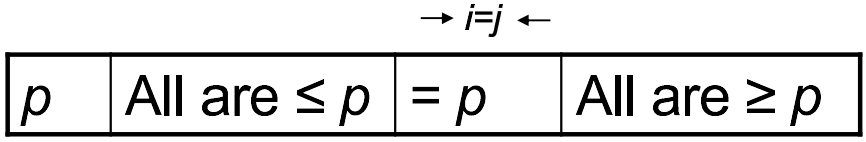
- Case2 : *i* and *j* have crossed over
  - $i > j$
  - Exchange the pivot with whom?
    - *A[j]*

$j \leftarrow$ $\quad\quad\quad\quad$ $\rightarrow i$

p	All are $\leq p$	$\leq p$	$\geq p$	All are $\geq p$

Quicksort: Three situations

- Case 3: Point to the same element
  - $i = j$
  - The value they are pointing must be p (why?)
  - Exchange the pivot with *A[j]* (why not *A[i]* ?)
    - Combine the case with case 2

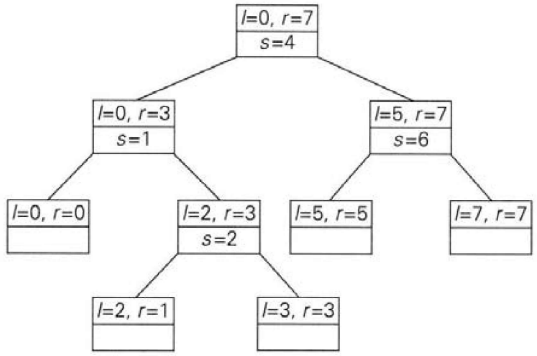


ALGORITHM *Partition*$(A[l..r])$

//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$(A[l], A[j])$
return j

0	1	2	3	4	5	6	7
	i						*j*
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	4	8	2	9	7
				i	*j*		
5	3	1	4	8	2	9	7
				i	*j*		
5	3	1	4	2	8	9	7
				j	*i*		
5	3	1	4	2	8	9	7
2	3	1	4	**5**	8	9	7
	i		*j*				
2	3	1	4				
	i	*j*					
2	3	1	4				
	i	*j*					
2	1	3	4				
	j	*i*					
2	1	3	4				
1	**2**	3	4				
1							
			ij				
		3	4				
		j	*i*				
		3	4				
			4				
						i	*j*
					8	9	7
						i	*j*
					8	7	9
						j	*i*
					8	7	9
					7	**8**	9
					7		
							9

(a)



(b)

FIGURE 4.3 An example of Quicksort operation. (a) The array's transformations with pivots shown in bold. (b) The tree of recursive calls to *Quicksort* with input values l and r of subarray bounds and split position s of a partition obtained.

Quicksort: Efficiency

- $C(n)$:
  - number of key comparison before a partition is achieved
  - $C(n)=n$ if they coincide (why?)
    - i start with the value l+1
  - $C(n)=n+1$ if the scanning indices cross over

Quicksort: Best case efficiency

- Conditions
  - All splits happened in the middle of subarrays

$$C_{best}(n) = 2C_{best}(n/2) + n \text{ for } n > 1, \quad C_{best}(1) = 0$$

- According to the Master Theorem

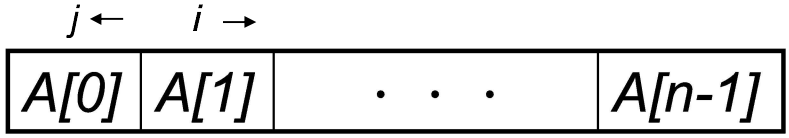
$$C_{best}(n) \in \Theta(n \log_2 n)$$

- Solving it exactly for $n=2^k$ yields

$$C_{best}(n) = n \log_2 n$$

Quicksort: Worst case efficiency

- Condition:
  - All the splits will be skewed to the extreme
- e.g. If *A*[0..*n*-1] is a strictly increasing array
  - Use *A[0]* as the pivot
  - The left-to-right scan will stop on *A[1]*
  - The right-to-left scan will go to reach *A[0]*
  - Indicating the split at position 0



Quicksort: Worst case efficiency

- Result of the partition
  - making n+1 comparisons
  - Exchanging pivot with itself
  - find the strictly increasing array $A[1..n-1]$ to sort
- The sorting will continue
  - until the last one $A[n-2..n-1]$ has been processed
- The total number of key comparisons:

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2)$$

Quicksort: Average case efficiency

- $C_{avg}(n)$
  - The average number of key comparisons
  - On randomly ordered array of size n
- Assume the partition split can happen
  - In each position s ($0 \le s \le n-1$)
  - With the same prob. $1/n$

Quicksort: Average case efficiency

- Recurrence relation

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}\left[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)\right] \text{for } n > 1$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

  - Q: Why n+1? Why not n?
  - A: For most case, i and j cross over. (i.e. n+1)
- Solving this recurrence

$$C_{avg}(n) \approx 2n \ln n \approx 1.38 n \log_2 n.$$

Quicksort

- Further discussion
  - Average case make only 38% more comparison
    - than the best case
  - There are many efforts to refine quicksort
    - Better pivot selection methods
      - Median-of-three partitioning
      - median of the leftmost, right most and middle element
      - Why choose median?
    - Switching to a simpler sort on smaller subfiles
    - Recursion elimination (nonrecursive quick sort)

Binary Tree Traversals and Related Properties

- Binary Tree *T*
  - A finite set of nodes
    - Empty or
    - Consist of a root and two disjoint binary tree
      - T_L and T_R
  - A special case of an ordered tree

Binary Tree Traversals and Related Properties

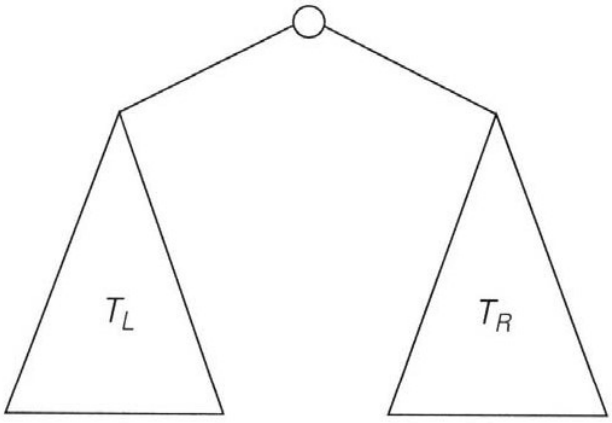


FIGURE 4.4 A standard representation of a binary tree

Compute the height of binary tree

- The height of a tree
  - The length of the longest path
    - From the root to a leaf
  - max. height of the root's left or right subtrees plus 1
  - The height of the empty tree = -1

Binary Tree Traversals and Related Properties

ALGORITHM $Height(T)$

//Computes recursively the height of a binary tree
//Input: A binary tree T
//Output: The height of T
if $T = \varnothing$ **return** -1
else return $\max\{Height(T_L), Height(T_R)\} + 1$

Binary Tree Traversals and Related Properties

- Efficiency analysis
  - *n(T)*
    - The number of nodes in a binary tree *T*
  - *A(n(T))*
    - The number of additions
    - Equal to the number of comparisons max{}
  - Recurrence relation for *A(n(T)):*

$$A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1 \text{ for } n(T) > 0,$$
$$A(0) = 0$$

Binary Tree Traversals and Related Properties

- The most frequently operation
  - Checking if T=Φ
  - e.g. for a single-node tree
    - Three checking
    - One addition
- Analysis by drawing the tree's extension
  - Its helps in analysis of tree algorithms

Binary Tree Traversals and Related Properties

- Draw the tree's extension
  - Replacing the empty subtrees by special nodes
  - The extra nodes
    - *External* nodes
  - The original nodes
    - *Internal* nodes

Binary Tree Traversals and Related Properties

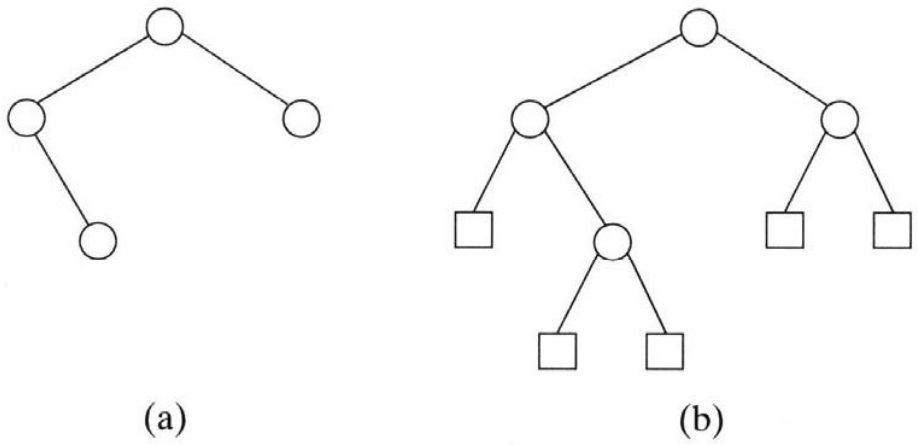


FIGURE 4.5 (a) Binary tree. (b) Its extension. Internal nodes are shown as circles; external nodes are shown as squares.

Binary Tree Traversals and Related Properties

- The algorithm makes
  - One addition for every internal node of the tree
  - One comparison to check
    - Whether the tree is empty for every nodes
  - To ascertain the algorithm's efficiency
    - Need to know how many external nodes
      - an extended tree with n internal nodes can have

Binary Tree Traversals and Related Properties

- How many external nodes *x*?
  - One more than the number of internal nodes *n*
    - $x = n + 1$, (by checking Fig. 4.5)
- Proof:
  - By induction, and $n \geq 0$
    - for n=0, we have the empty tree with 1 external node
  - Assume that:
    - $x = k + 1$
    - For any extended binary tree with $0 \leq k < n$ internal nodes

Binary Tree Traversals and Related Properties

- An extended binary tree T with
  - n internal nodes
    - n_L: the number of internal nodes in the left subtree
    - n_R: the number of internal nodes in the right subtree
  - x external nodes
    - x_L: the number of external nodes in the lrft subtree
    - x_R: the number of external nodes in the right subtree

Binary Tree Traversals and Related Properties

- Since $n > 0$, T has a root
  - $n = n_L + n_R + 1$
- Equation is Correct for left and right subtrees:
  - $x = x_L + x_R = (n_L + 1) + (n_R + 1) = (n_L + n_R + 1) + 1 = n + 1$

Binary Tree Traversals and Related Properties

- Return to the algorithm *Height*
  - The number of comparisons:

$$C(n) = n + x = 2n + 1$$

  - The number of additions:

$$A(n) = n$$

Binary Trees Traversals and Related Properties

- Preorder traversal
  - The root is visited before
    - The left and right subtrees are visited (in that order)
- Inorder traversal
  - The root is visited
    - After Visiting its left subtree
    - But before visiting the right subtree

Binary Trees Traversals and Related Properties

- Postorder traversal
  - The root is visited after
    - Visiting the left and right subtrees (in that order)
- Their efficiencies is identical to
  - The *Height* algorithm
  - Why?
    - A recursive call is made for each node of an extended binary tree

Multiplication of Large Integers

- Problem:
  - Integers are too long
    - to fit in a single word of a computer
  - They require special treatment
- Classic pen-and-pencil algorithm
  - Multiplying two n-digit integers
    - Each of n digits of the first number is multiplied by each of the n digits of the second number
  - Total n^2 digit multiplications

Multiplication of Large Integers

- Design an algorithm
  - With fewer than n^2 digit multiplications
  - By using divide-and-conquer

- A case of two-digit integers, 23 and 14

$$23 = 2 \cdot 10^1 + 3 \cdot 10^0 \text{ and } 14 = 1 \cdot 10^1 + 4 \cdot 10^0.$$

Multiplication of Large Integers

- Multiply them:

$$\begin{aligned} 23*14 &= \left(2 \cdot 10^1 + 3 \cdot 10^0\right) \times \left(1 \cdot 10^1 + 4 \cdot 10^0\right) \\ &= (2 \cdot 1)10^2 + (3 \cdot 1 + 2 \cdot 4)10^1 + (3 \cdot 4)10^0 \end{aligned}$$

- Compute the middle term with
  - One digit multiplication
  - By taking advantage of the products

$$3*1+2*4 = (2+3)*(1+4)-(2*1)-(3*4).$$

Multiplication of Large Integers

- For any pair of two-digit numbers
  - $a = a_1 a_0$ and $b = b_1 b_0$
  - Their product *c* can be compute by:
    $c = a * b = c_2 10^2 + c_1 10^1 + c_0$
  - Where

$c_2 = a_1 * b_1$ is the product of their first digits

$c_0 = a_0 * b_0$ is the product of their second digits

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's digits and the sum of the b's digits minus the sum of c_2 and c_0.

Multiplication of Large Integers

- Apply this to *n*-digit integers
  - Multiple two *n*-digit integers a and b
    - *n* is positive even number
  - Divide both number in the middle
  - Denote
    - First half of the *a*'s digits by a_1
    - Second half by a_0
    - First half of the *b*'s digits by b_1
    - Second half by b_0

Multiplication of Large Integers

- We get:

$$c = a * b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$$
$$= (a_1 * b_1) 10^n + (a_1 * b_0 + a_0 * b_1) 10^{n/2} + (a_0 * b_0)$$
$$= c_2 10^n + c_1 10^{n/2} + c_0$$

- Where:

$c_2 = a_1 * b_1$ is the product of their first halves,
$c_0 = a_0 * b_0$ is the product of their second halves,
$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0

Multiplication of Large Integers

- If $n/2$ is even
  - Apply the same method for c_2, c_0 and c_1
- If n is a power of 2
  - Recursive algorithm for computing
    - the product of two n-digit integers
  - Stopped when n becomes one or small enough

Multiplication of Large Integers

- *M(n)*: The number of multiplications
- Recurrence for *M(n)*:

$$M(n) = 3M(n/2) \text{ for } n > 1, M(1) = 1$$

  - Since Multiplication of n-digit numbers requires three multiplications of n/2-digit numbers

Multiplication of Large Integers

- Solving it by backward substitutions
  - $n=2^k$

$$M(2^k) = 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2 M(2^{k-2})$$
$$= \cdots = 3^i M(2^{k-i}) = \cdots = 3^k M(2^{k-k}) = 3^k$$

- Since $k=\log_2 n$, $a^{\log_b c} = c^{\log_b a}$

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

Strassen's Matrix Multiplication

- The product *C* of two 2X2 matries *A* and *B*
  - Brute force: 8 multiplications
  - Strassen's method: 7 multiplications
- Formula:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ b_{10} & b_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

Strassen's Matrix Multiplication

- Where,

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$$
$$m_2 = (a_{10} + a_{11}) * b_{00}$$
$$m_3 = a_{00} * (b_{01} - b_{11})$$
$$m_4 = a_{11} * (b_{10} - b_{00})$$
$$m_5 = (a_{00} + a_{01}) * b_{11}$$
$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$$
$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$$

Strassen's Matrix Multiplication

- Comparison
  - Strassen's algorithm makes
    - 7 multiplication and
    - 18 addition/subtractions
  - Brute force
    - 8 multiplication and
    - 4 additions
- Question:
  - Would you use strassen's algorithm?

Strassen's Matrix Multiplication

- Consider
  - its superiority as Matrix order *n* goes to infinity
- Assumption
  - A,B: nxn matrices, where n is a power of 2
- Divide A, B and their product C into
  - Four n/2-by-n/2 submatrices:

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

Strassen's Matrix Multiplication

- Strassen's Method
  - Treat these submatrices as numbers to get the correct product
    - $C_{00}=A_{00}*B_{00}+A_{01}*B_{10}$ or
    - $C_{00}=M_1+M_4-M_5+M_7$
  - The 7 products of n/2-by-n/2 matrices are computed recursively

Strassen's Matrix Multiplication

- Efficiency
  - *M(n):* the number of multiplications
    - Recurrence relation:

$$M(n) = 7M(n/2) \text{ for } n > 1, M(1) = 1$$

  - Since $n=2^k$

$$M(2^k) = 7M(2^{k-1})] = 7^2 M(2^{k-2}) = \cdots$$
$$= 7^i M(2^{k-i}) \cdots = 7^k M(2^{k-k}) = 7^k.$$

Strassen's Matrix Multiplication

- Since $k=log_2 n$

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$$

- Smaller than n^3
  - required by brute-force algorithm

Strassen's Matrix Multiplication: Efficiency

- *A(n):* the number of additions
  - Multiplications of two nxn matrices needs
    - 7 Multiplications of two n/2-by-n/2 matrices
    - 18 additions of two n/2-by-n/2 matrices
    - As n=1, no addition are made
      - Since two numbers are simply multiplied
  - Recurrence relation:

$$A(n) = 7A(n/2) + 18(n/2)^2 \text{ for } n > 1, A(1) = 0$$

Strassen's Matrix Multiplication: Efficiency

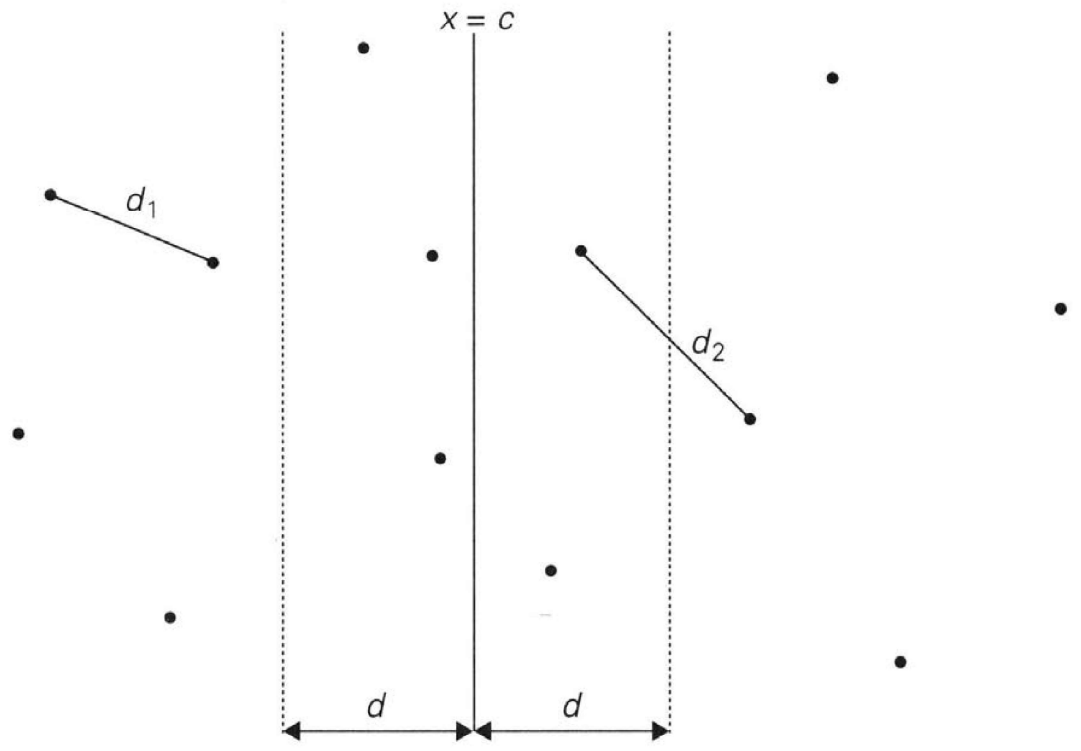
- According to Master theorem
  - We get

$$A(n) \in \Theta(n^{\log_2 7}) = M(n) \in \Theta(n^{\log_2 7})$$

  - same order of growth as the number of multiplication

- So, the Strassen's algorithm in $\Theta(n^{\log_2 7})$
  - Which is better than $\Theta(n^3)$ of brute-force method

Closest-Pair by Divide-and-Conquer



(a)

Closest-Pair by Divide-and-Conquer

- Preworks
  - Let $P_1=(x_1,y_1)\ldots P_n=(x_n,y_n)$ be n points in the plane
  - n is a power of 2
- Assume
  - points are ordered in ascending order of their x coordinates
- Step1
  - Divide them into two subsets
    - S_1 and S_2 of $n/2$ points
    - By drawing a vertical line $x=c$
    - c= the median of the x coordinates

Closest-Pair by Divide-and-Conquer

- Step2
  - Find recursively the closest pairs
    - For left subset S_1
    - For right subset S_2
  - Let d_1 and d_2 be the smallest distance
    - Between pairs of points in S_1 and S_2, *respectively*
  - $d=\min\{d_1,d_2\}$

Closest-Pair by Divide-and-Conquer

- Good!!
- Q: Is d the smallest distance between all pairs of points?
- Why not?

Closest-Pair by Divide-and-Conquer

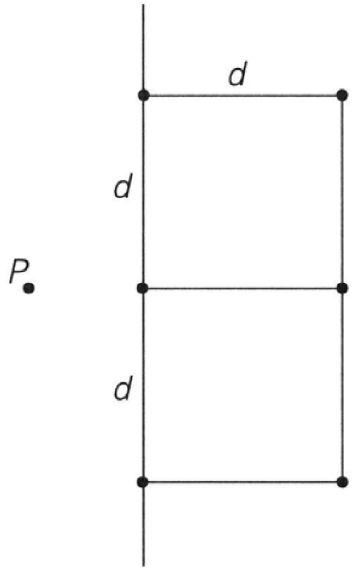
- d is not the smallest distance between
  - All pairs of points in S_1 and S_2
  - A closer pair of points can lie on opposite sides
  - Need to examine such points
- Only attention to the points
  - In the symmetric vertical strip of width *2d*
  - Reason
    - The distance between any other pair of points is greater than *d*

Closest-Pair by Divide-and-Conquer

- Let C_1 and C_2 be the subsets of points
  - in the left and right pairs of strip, respectively

- For every point $P(x,y)$ in C_1
  - Inspect points in C_2
    - that may be closer to P than d
  - Such points must have their y coordinates
    - In the interval $[y-d, y+d]$

Closest-Pair by Divide-and-Conquer



(b)

FIGURE 4.6 (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.

Closest-Pair by Divide-and-Conquer

- Critical observation
  - How many such points is possible?
  - There can be no more than ***six*** such points
    - Any pair of points in C_2 is at least d apart from each other
- Maintain lists of points in C_1 and C_2
  - Sorted in ascending order of their y coordinates
    - Like projections of the points on the dividing line

Closest-Pair by Divide-and-Conquer

- Process C_1 points P sequentially
  - To fetch up to six candidates
    - A pointer into the C_2 list may oscillate
      - within an interval of width $2d$
  - compute the distances
    - between P and these candidates

Closest-Pair by Divide-and-Conquer

- Analysis
  - *M(n)* for the "merging" of solutions is in *O(n)*
  - Recurrence for *T(n)*
    - The running time of this algorithm

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

  - Applying the *O* version of Master Theorem

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

Convex-Hull by divide-and-conquer

Divide-and-Conquer

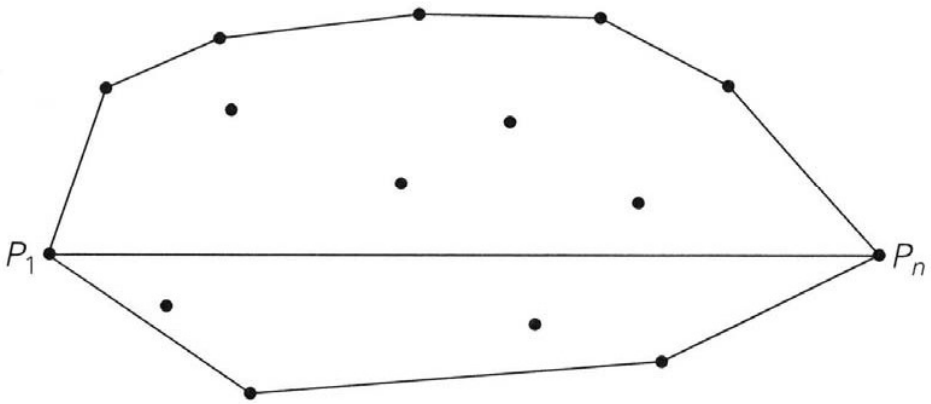


FIGURE 4.7 Upper and lower hulls of a set of points

Convex-Hull by divide-and-conquer

- Problem:
  - Finding the smallest convex polygon
    - Contains *n* given points in the plane
- Let $P_1=(x_1,y_1),\ldots,P_n=(x_n,y_n)$
  - A set of *S* of *n* points in the plane
- Assume that the points are sorted
  - In increasing order of their *x* coordinates

Convex-Hull by divide-and-conquer

- Quickhull
  - The leftmost point P_1 and the rightmost P_n
    - Must belong to the set's convex hull
  - Let $\overrightarrow{P_1P_n}$ be the straight line
    - Through points P_1 to P_n directed from P_1 to P_n

Convex-Hull by divide-and-conquer

- Separates the points into two sets
  - S_1: the set of points to the left of or on this line
  - S_2: the set of points to the right of or on this line
  - The convex hull S_1 consists of
    - The line segment with the end points at P_1 , P_n
    - An upper boundary (upper hull)
  - The convex hull S_2 consists of
    - The line segment with the end points at P_1 , P_n
    - An lower boundary (lower hull)

Convex-Hull by divide-and-conquer

- The convex hull of the entire set S
  - Composed of the upper and lower hulls
  - They can be constructed independently
- How to construct upper hull ?
  - Identify vertex P_{max} in S_1
    - P_{max} is the farthest vertex from the line $\overrightarrow{P_1P_n}$
    - P_{max} maximizes the area of triangle $P_1P_nP_{max}$

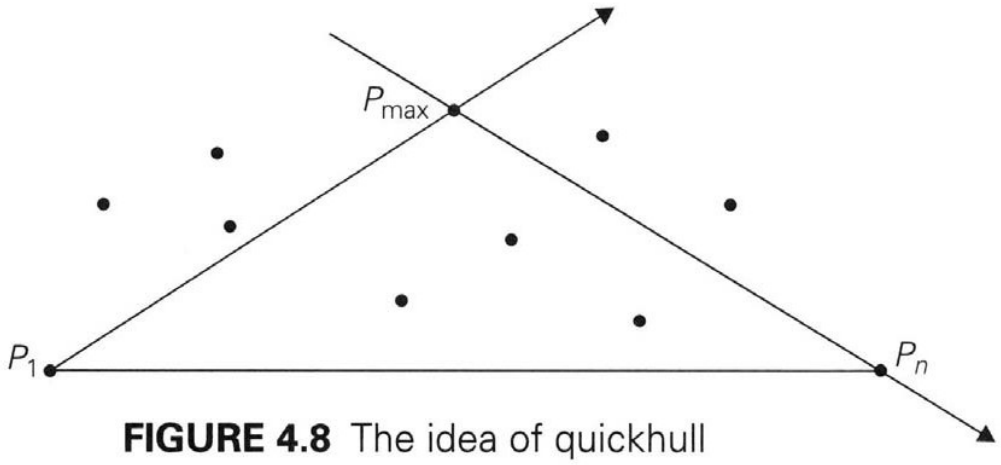


FIGURE 4.8 The idea of quickhull

Convex-Hull by divide-and-conquer: Construct the upper hull using quckhull

- Identify all points P_i of S_1
  - which are to the left of the line $\overrightarrow{P_1P_{\max}}$
  - P_i , P_1 and P_{max} will make up the $S_{1,1}$
- The points P_j of S_1
  - which are to the left of the line $\overrightarrow{P_{\max}P_n}$
  - P_j , P_1 and P_{max} will make up the $S_{1,2}$
- The points inside $\Delta P_1P_{\max}P_n$ can be eliminated

Convex-Hull by divide-and-conquer: Construct the upper hull using quckhull

- Continue constructing
  - The upper hull of $S_{1,1}$ and $S_{1,2}$ recursively
- Concatenate them to get upper hull of entire S_1

- How check a point is to the left of a line?
  - A very useful fact from analytical geometry

Convex-Hull by divide-and-conquer: Construct the upper hull using quckhull

- If $p_1=(x_1,y_1)$, $p_2=(x_2,y_2)$, $p_3=(x_3,y_3)$
  - The area of $\Delta p_1 p_2 p_3$ is equal to
    - ½ magnitude of the determinant

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 + x_3y_1 + x_2y_3 - x_3y_2 - x_2y_1 - x_1y_3$$

- The sign of this expression is positive
  - Iff $p_3=(x_3,y_3)$ is to the left of the line $\overrightarrow{P_1P_2}$

Convex-Hull by divide-and-conquer: Efficiency

- Using the formula, we can
  - Check whether a point lies to the left of a line
    - in constant time
  - Find the distance from the point to the line

- Efficiency
  - in the average case $\Theta(n \log n)$
  - In the worst case $\Theta(n^2)$
    - [PSh85] is with worst-case efficiency in $\Theta(n \log n)$

SUMMARY

- *Divide-and-conquer* is a general algorithm design technique that solves a problem's instance by dividing it into several smaller instances (ideally, of equal size), solving each of them recursively, and then combining their solutions to get a solution to the original instance of the problem. Many efficient algorithms are based on this technique, although it can be both inapplicable and inferior to simpler algorithmic solutions.

- Time efficiency $T(n)$ of many divide-and-conquer algorithms satisfies the equation $T(n) = aT(n/b) + f(n)$. The *Master Theorem* establishes the order of growth of this equation's solutions.

- *Mergesort* is a divide-and-conquer sorting algorithm. It works by dividing an input array into two halves, sorting them recursively, and then *merging* the two sorted halves to get the original array sorted. The algorithm's time efficiency is in $\Theta(n \log n)$ in all cases, with the number of key comparisons being very close to the theoretical minimum. Its principal drawback is a significant extra storage requirement.

- *Quicksort* is a divide-and-conquer sorting algorithm that works by partitioning its input's elements according to their value relative to some preselected element. Quicksort is noted for its superior efficiency among $n \log n$ algorithms for sorting randomly ordered arrays but also for the quadratic worst-case efficiency.

- *Binary search* is a $O(\log n)$ algorithm for searching in sorted arrays. It is an atypical example of an application of the divide-and-conquer technique because it needs to solve just one problem of half the size on each of its iterations.

- The classic traversals of a binary tree—*preorder*, *inorder*, and *postorder*—and similar algorithms that require recursive processing of both left and right subtrees can be considered examples of the divide-and-conquer technique. Their analysis is helped by replacing all the empty subtrees of a given tree with special *external nodes*.

- There is a divide-and-conquer algorithm for multiplying two n-digit integers that requires about $n^{1.585}$ one-digit multiplications.

- *Strassen's algorithm* needs only seven multiplications to multiply two 2-by-2 matrices but requires more additions than the definition-based algorithm. By exploiting the divide-and-conquer technique, this algorithm can multiply two n-by-n matrices with about $n^{2.807}$ multiplications.

- The divide-and-conquer technique can be successfully applied to two important problems of computational geometry: the closest-pair problem and the convex-hull problem.