Divide-and-Conquer Strategy

CMSC 142: Design and Analysis of Algorithms

Department of Mathematics and Computer Science
College of Science
University of the Philippines Baguio

Overview

Divide-and-Conquer

Divide-and-conquer is an algorithmic paradigm in which a problem is solved using the Divide, Conquer, and Combine strategy.

A divide-and-conquer algorithm typically consists of three steps:

1. Divide instance of problem into two or more smaller instances of the same problem (*ideally about the same size*)
2. Solve the smaller instances recursively (*sometimes a different algorithm is employed when instances become small enough*)
3. Obtain solution to the original (larger) instance by combining the solutions from the smaller instances

Divide-and-Conquer Algorithms

Examples of divide-and-conquer algorithms

1. Sorting: Mergesort and Quicksort
2. Binary tree traversals
3. Multiplication of large integers
4. Matrix multiplication: Strassen's algorithm
5. Closest-pair by divide-and-conquer

Divide-and-Conquer

- In the most typical case of divide-and-conquer, a problem's instance of size n can be divided into b instances of size n/b, with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$.)
- Assuming size n is a power of b, the recurrence for the running time $T(n)$ is

$$T(n) = aT(n/b) + f(n) \tag{1}$$

  where $f(n)$ is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions.
- (1) is called the **general divide-and-conquer recurrence**.

Recall. Master Theorem

If $f(n) \in \Theta(n^d)$ with $d \geq 0$ in recurrence equation (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.)

Example. Find the order of growth for the solutions of the ff. recurrences.

a. $T(n) = 3T(n/2) + n^2$

b. $T(n) = 16T(n/4) + n^2$

c. $T(n) = 4T(n/2) + n^3$

d. $T(n) = 3T(n/3) + \sqrt{n}$

Overview

Divide-and-Conquer Sorting Algorithm

Merge Sort

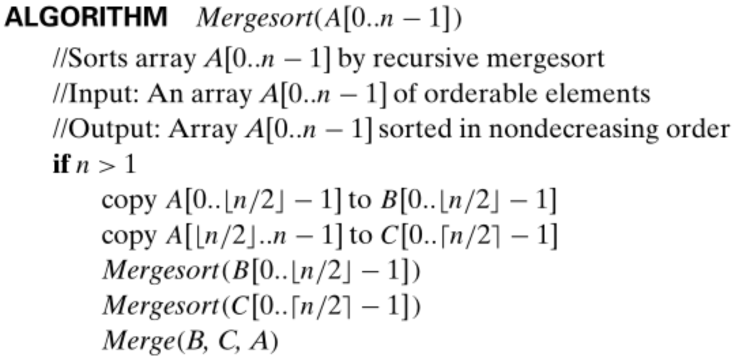
- *Split array* $A[0..n-1]$ into about equal halves and make copies of each half in arrays B and C. *Sort arrays B and C recursively*.
- Then, *merge sorted arrays* B and C into array A. **How?**

  Repeat steps below 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

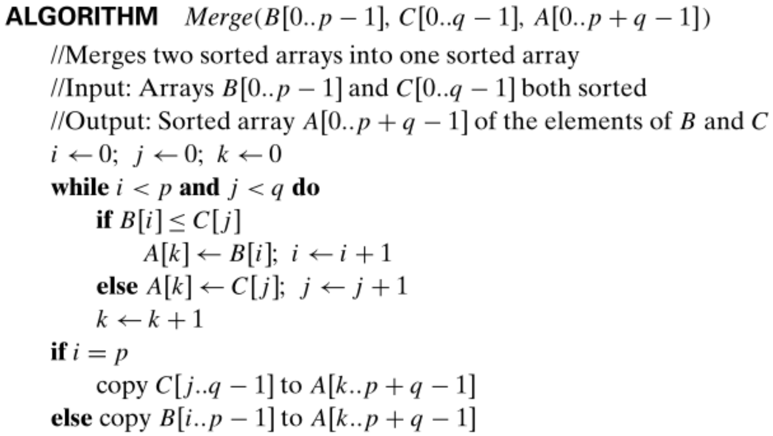
  Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Mergesort



```
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($B, C, A$)
```

Figure: Pseudocode for the mergesort algorithm



```
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 $B[i] \leq C[j]$
            $A[k] \leftarrow B[i]$;  $i \leftarrow i + 1$
        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 $A[k..p+q-1]$
```

Figure: Pseudocode for the merge procedure in Mergesort

Efficiency of Mergesort Algorithm

- Suppose that $n = 2^k$ where k is a natural number. The recurrence relation for the number of key comparisons $C(n)$ is

$$\begin{aligned} C(n) &= 2C\left(\frac{n}{2}\right) + C_{merge}(n) \quad \text{for } n > 1 \\ C(1) &= 0 \end{aligned}$$

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

- **What is $C_{merge}(n)$ in the best-case and worst-case scenarios?**

Efficiency of Mergesort Algorithm

- For the worst-case scenario, we have the recurrence

$$\begin{aligned} C_{worst}(n) &= 2C_{worst}\left(\frac{n}{2}\right) + n - 1 \quad \text{for } n > 1 \\ C_{worst}(1) &= 0 \end{aligned}$$

  and by the Master Theorem, $C_{worst}(n) \in \Theta(n \log n)$.

Remarks

- The number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting: $\lceil \log_2 n! \rceil \approx \lceil n \log_2 n - 1.44n \rceil$
- Space requirement: $\Theta(n)$

Divide-and-Conquer Sorting Algorithm

Quicksort

- rearranges the elements of an array $A[0..n-1]$ to achieve its **partition**, where all the elements before some position s are smaller than or equal to $A[s]$ and all the elements after position s are greater than or equal to $A[s]$

```
ALGORITHM  Quicksort(A[l..r])
  //Sorts a subarray by quicksort
  //Input: Subarray of array A[0..n − 1], defined by
  //          its left and right indices l and r
  //Output: Subarray A[l..r] sorted in nondecreasing order
  if l < r
      s ← Partition(A[l..r])  //s is a split position
      Quicksort(A[l..s − 1])
      Quicksort(A[s + 1..r])
```

Finding a Partition

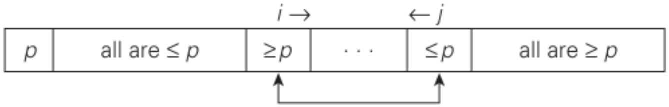
Hoare Partitioning Algorithm

1. Choose a **pivot** element (say the array's first element for simplicity).
2. To achieve a partition, use the left-to-right and right-to-left scan.
   - The left-to-right scan starts with the second element. This scan skips over elements that are smaller than the pivot and stops on encountering the first element greater than or equal to the pivot.
   - The right-to-left scan starts with the last element of the subarray. This scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.

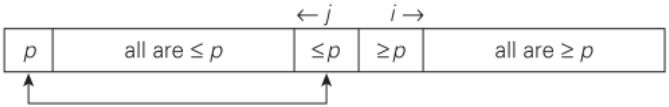
2a. After both scans stop, three situations may arise.

Hoare Partitioning Algorithm (continuation)

Case 1. If scanning indices i and j have not crossed, $(i < j)$, exchange $A[i]$ and $A[j]$ and resume the scans by incrementing i and decrementing j, resp.

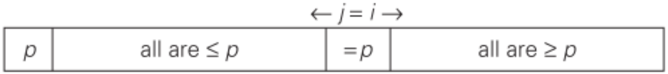


Case 2. If the scanning indices have crossed over, i.e., $i > j$, we will have partitioned the array after exchanging the pivot with $A[j]$.



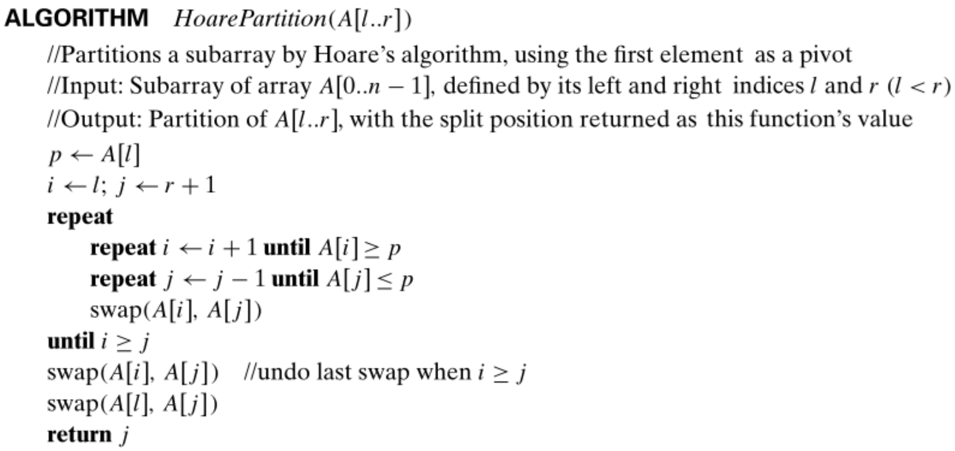
Hoare Partitioning Algorithm (continuation)

Case 3. If the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to the pivot element.



This case can be combined with the case of crossed-over indices $(i > j)$ in Case 2 by exchanging the pivot with $A[j]$ whenever $i \geq j$.

Remark. The number of key comparisons made before a partition is achieved is $n+1$ if the scanning indices cross over, and n if they coincide.



```
ALGORITHM  HoarePartition(A[l..r])
    //Partitions a subarray by Hoare's algorithm, using the first element as a pivot
    //Input: Subarray of array A[0..n − 1], defined by its left and right indices l and r (l < r)
    //Output: 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 $j$
```

Figure: Pseudocode implementing the Hoare partition procedure

Efficiency of Quicksort

Best Case

- Occurs if all the splits happen in the middle of corresponding subarrays.
- The number of comparisons satisfies the recurrence

$$\begin{aligned} C_{best}(n) &= 2C_{best}(n/2) + n \quad \text{for } n > 1 \\ C_{best}(1) &= 0 \end{aligned}$$

- According to the Master Theorem, $C_{best}(n) = \Theta(n \log n)$

Efficiency of Quicksort

Worst Case

- Occurs if all the splits are skewed to the extreme: one of the two subarrays will be empty, while the size of the other will be just one less than the size of the subarray being partitioned. For example, in increasing arrays.
- The total number of key comparisons made will be equal to

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

Efficiency of Quicksort

Average Case

- Assuming that the partition split can happen in each position s $(0 \le s \le n-1)$ with the same probability $1/n$, then we get the following recurrence relation

$$\begin{aligned} C_{avg}(n) &= \frac{1}{n}\sum_{s=0}^{n-1}\left[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)\right] \\ C_{avg}(1) &= 0 \quad \text{and} \quad C_{avg}(0) = 0 \end{aligned}$$

- Solution: $C_{avg}(n) \approx 2n \ln n \approx 1.38 n \log_2 n$ *(on the average, quicksort makes only 38% more comparisons than in best case.)*

Overview

Binary Tree Traversals and Related Properties

A **binary tree** T is defined as a finite set of nodes that is either empty or consists of a root and two disjoint trees T_L and T_R called, respectively, the left and right subtree of the root.

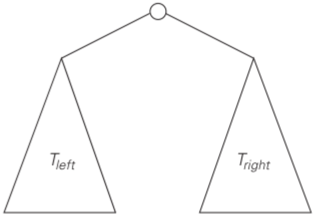


Figure: Standard representation of a binary tree.

Binary Tree Traversals and Related Properties

Many problems about binary trees can be solved by applying the divide-and-conquer technique.

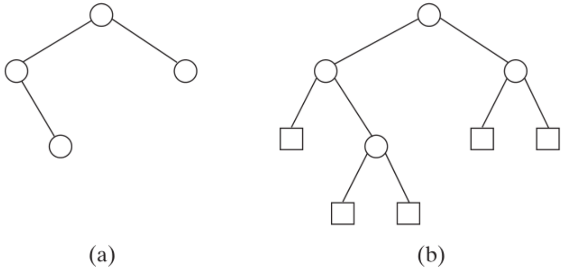
Example

Below is a recursive algorithm for computing the height of a binary tree.

```
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 $\max\{Height(T_{left}), Height(T_{right})\} + 1$
```

Binary Tree Traversals and Related Properties

- Drawing the **tree's extension** by replacing the empty subtrees by special nodes helps in the analysis of tree algorithms. The extra nodes are called **external** while the original nodes are called **internal**.



(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 most important divide-and-conquer algorithms for binary trees are the three classic traversals: preorder, inorder and postorder.

- In the **preorder traversal**, the root is visited before the left and right subtrees are visited (in that order).
- In the **inorder traversal**, the root is visited after visiting its left subtree but before visiting the right subtree.
- In the **postorder traversal**, the root is visited after visiting the left and right subtrees (in that order).

Overview

Multiplication of Large Integers

- In the classic pen-and-pencil algorithm for multiplying two $n-$digit integers, each of the n digits of the first number is multiplied by each of the n digits of the second number, for a total of n^2 digit multiplications.
- **Basic Idea:** for any pair of two-digit integers $a = a_1a_0$ and $b = b_1b_0$, their product c can be computed by the formula

$$c = a * b = c_2 10^2 + c_1 10^1 + c_0$$

where

$$\begin{aligned} c_2 &= a_1 * b_1 \text{ is the product of their first digits} \\ c_0 &= a_0 * b_0 \text{ is the product of their second digits} \\ c_1 &= (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0) \end{aligned}$$

Divide-and-Conquer Multiplication

Karatsuba Algorithm (Anatoly Karatsuba, 1960)

1. Let a and b be two n-digit integers where n is a positive even integer.
2. Divide both numbers in the middle. Denote the first half of the $a's$ digit by a_1 and the second half by a_0; for b the notations are b_1 and b_0 respectively. In these notations,
$$a = a_1 a_0 = a_1 10^{n/2} + a_0$$
and
$$b = b_1 b_0 = b_1 10^{n/2} + b_0.$$

Divide-and-Conquer Multiplication

Procedure (cont.):

3. Taking advantage of the same trick used for two-digit numbers, we get

$$\begin{aligned} c &= a * b = \left(a_1 10^{n/2} + a_0\right) * \left(b_1 10^{n/2} + b_0\right) \\ &= (a * b)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0) \\ &= c_2 10^n + c_1 10^{n/2} + c_0 \end{aligned}$$

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
$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$

Efficiency Analysis of Karatsuba Algorithm

- If n is a power of 2, the number of multiplications, $M(n)$, the algorithm makes is given by the recurrence

$$\begin{aligned} M(n) &= 3M\left(\frac{n}{2}\right) \qquad \text{for } n > 1 \\ M(1) &= 1. \end{aligned}$$

(Base case: When the digits to be multiplied are single digits, they are multiplied directly. In general, it is worth switching to conventional methods after multiplicands become sufficiently small.)

- Using method of backward substitution for $n = 2^k$, $M(n) = M(2^k) = 3^k$. Since $k = \log_2 n$,

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

Strassen's Matrix Multiplication

- The product C of two 2-by-2 matrices A and B can be found with just seven multiplications. This is accomplished by using the following formula,

$$\begin{aligned}\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} &= \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} \\ &= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}\end{aligned}$$

where

$$\begin{aligned} m_1 &= (a_{00} + a_{11}) * (b_{00} + b_{11}), & m_5 &= (a_{00} + a_{01}) * b_{11}, \\ m_2 &= (a_{10} + a_{11}) * b_{00}, & m_6 &= (a_{10} - a_{00}) * (b_{00} + b_{01}), \\ m_3 &= a_{00} * (b_{01} - b_{11}), & m_7 &= (a_{01} - a_{11}) * (b_{10} + b_{11}). \\ m_4 &= a_{11} * (b_{10} - b_{00}), \end{aligned}$$

Strassen's Matrix Multiplication

Strassen's Algorithm (Volker Strassen, 1969)

1. Let A and B be two $n \times n$ matrices where n is a power of two.(*If n is not a power of 2, matrices can be padded with rows and columns of zeros.*)
2. Divide A, B and their product C into four $n/2 \times n/2$ submatrices each as follows
$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$
One can treat these submatrices as numbers to get the correct product.
3. If the seven products of $n/2 \times n/2$ matrices are computed recursively by the same method, we have Strassen's algorithm for matrix multiplication.

Efficiency Analysis of the Strassen's Matrix Multiplication

- If the size n of the square matrix and n is a power of 2, the number of multiplications, $M(n)$, the algorithm makes is given by the recurrence

$$\begin{aligned} M(n) &= 7M\left(\frac{n}{2}\right) \qquad \text{for } n > 1 \\ M(1) &= 1. \end{aligned}$$

- Using method of backward substitution for $n = 2^k$, $M(n) = M(2^k) = 7^k$. Since $k = \log_2 n$,

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

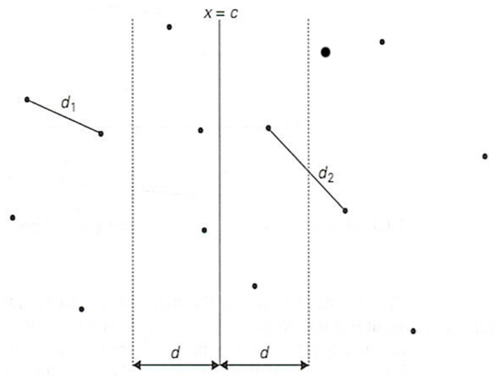
which is smaller than n^3 required by the brute-force algorithm.

Overview

Closest-Pair Problem by Divide-and-Conquer

Procedure

1. Sort the points by $x-$coordinate (*list one*) and then by $y-$coordinate (*list two*).
2. 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 Problem by Divide-and-Conquer

Procedure (cont.)

3. Find recursively the closest pairs for the left and right subsets.
4. 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.
5. 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 six such points since $d \leq d_2$.

Efficiency Analysis

- If n is the number of points (without sorting), the running time of the algorithm is given by the recurrence

$$T(n) \quad = \quad 2T\left(\frac{n}{2}\right) + M(n)$$

where $M(n) \in \Theta(n)$ is the time for the "merging" of solutions to the smaller subproblems.

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

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

which is smaller than n^2 required by the brute-force algorithm.

End of Lecture