The ***decrease-and-conquer*** technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance. Once such a relationship is established, it can be exploited either top down or bottom up. The former leads naturally to a recursive implementation, although, as one can see from several examples in this chapter, an ultimate implementation may well be nonrecursive. The bottom-up variation is usually implemented iteratively, starting with a solution to the smallest instance of the problem; it is called sometimes the ***incremental approach***.
There are three major variations of decrease-and-conquer:
decrease by a constant
decrease by a constant factor
variable size decrease

In the ***decrease-by-a-constant*** variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one (Figure 4.1), although other constant size reductions do happen occasionally.
Consider, as an example, the exponentiation problem of computing *an* where $a \neq 0$ and n is a nonnegative integer. The relationship between a solution to an instance of size n and an instance of size $n - 1$ is obtained by the obvious formula $a^n = a^{n-1} \cdot a$. So the function $f(n) = a^n$ can be omputed either "top down" by using its recursive definition or "bottom up" by multiplying 1 by a n times. (Yes, it is the same as the brute-force algorithm, but we have come to it by a different thought process.)

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0, \end{cases}$$

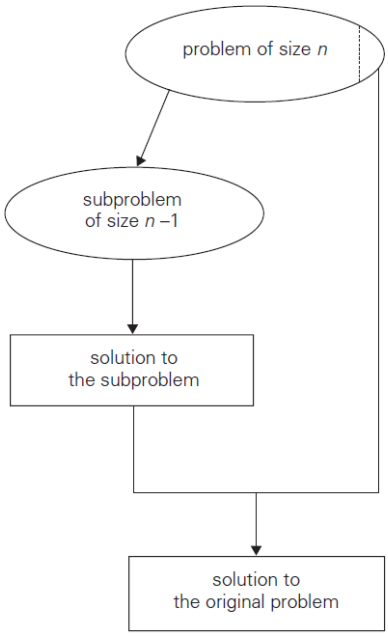


FIGURE 4.1 Decrease-(by one)-and-conquer technique.

The ***decrease-by-a-constant-factor*** technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two. For an example, let us revisit the exponentiation problem. If the instance of
size n is to compute a^n, the instance of half its size is to compute $an/2$, with the obvious relationship between the two: $an = (a^{n/2})^2$. But since we consider here instances with integer exponents only, the former does not work for odd n. If n is odd, we have to compute a^{n-1} by using the rule for even-valued exponents and then multiply the result by a.

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases}$$

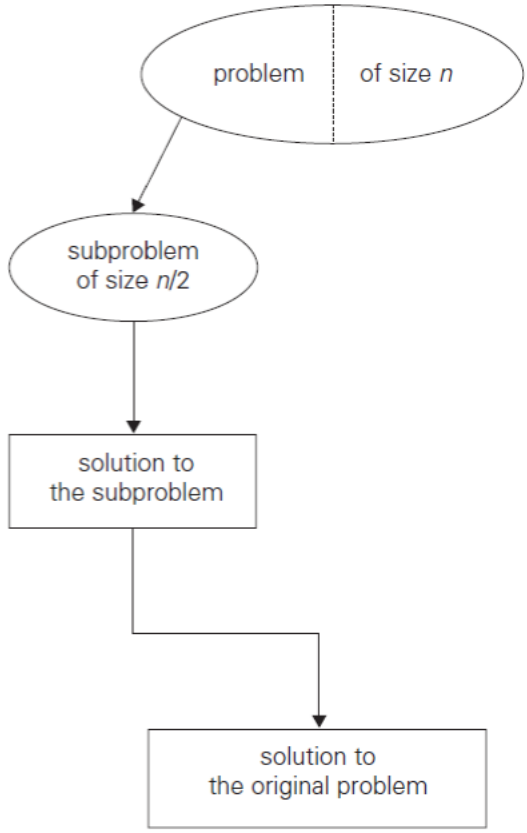


FIGURE 4.2 Decrease-(by half)-and-conquer technique.

Finally, in the ***variable-size-decrease*** variety of decrease-and-conquer, the size-reduction pattern varies from one iteration of an algorithm to another. Euclid's algorithm for computing the greatest common divisor provides a good example of such a situation. Recall that this algorithm is based on the formula gcd*(m, n)* = gcd*(n, m* mod *n)*.

Insertion Sort

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

```
//Sorts a given array by insertion sort
//Input: An array A[0..n − 1] of n orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
```

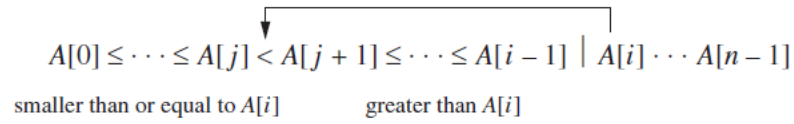


FIGURE 4.3 Iteration of insertion sort: $A[i]$ is inserted in its proper position among the preceding elements previously sorted.

89	**45**	68	90	29	34	17
45	89	**68**	90	29	34	17
45	68	89	**90**	29	34	17
45	68	89	90	**29**	34	17
29	45	68	89	90	**34**	17
29	34	45	68	89	90	**17**
17	29	34	45	68	89	90

FIGURE 4.4 Example of sorting with insertion sort. A vertical bar separates the sorted part of the array from the remaining elements; the element being inserted is in bold.

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Thus, in the worst case, insertion sort makes exactly the same number of comparisons as selection sort (see Section 3.1).

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

Topological Sorting

directed graph, or ***digraph*** for short, is a graph with directions specified for all its edges (Figure 4.5a is an example). The adjacency matrix and adjacency lists are still two principal means of representing a digraph. There are only two notable differences between undirected and directed graphs in representing them: (1) the adjacency matrix of a directed graph does not have to be symmetric; (2) an edge in a directed graph has just one (not two) corresponding nodes in the digraph's adjacency lists. Depth-first search and breadth-first search are principal traversal algorithms for traversing digraphs as well, but the structure of corresponding forests can be
more complex than for undirected graphs.

Note that a back edge in a DFS forest of a directed graph can connect a vertex to its parent. Whether or not it is the case, the presence of a back edge indicates that the digraph has a directed cycle. A ***directed cycle*** in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor. For example, *a, b, a* is a directed cycle in the digraph in Figure 4.5a. Conversely, if a DFS forest of a digraph has no back
edges, the digraph is a ***dag***, an acronym for ***directed acyclic graph***. For topological sorting to be possible, a digraph in question must be a dag. It turns out that being a dag is not only necessary
but also sufficient for topological sorting to be possible; i.e., if a digraph has no directed cycles, the topological sorting problem for it has a solution

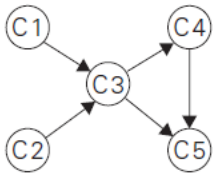


FIGURE 4.6 Digraph representing the prerequisite structure of five courses.

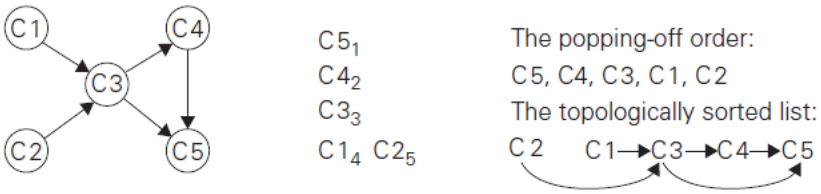


FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

There are two efficient algorithms that both verify whether a digraph is a dag and, if it is, produce an ordering of vertices that solves the topological sorting problem. The first algorithm is a simple application of depth-first search: perform a DFS traversal and note the order in which vertices become dead-ends (i.e., popped off the traversal stack). Reversing this order yields a solution to the topological sorting problem, provided, of course, no back edge

has been encountered during the traversal. If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.

The second algorithm is based on a direct implementation of the decrease-(by one)-and-conquer technique: repeatedly, identify in a remaining digraph a ***source***, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it. (If there are several sources, break the tie arbitrarily. If there are none, stop because the problem cannot be solved The order in which the vertices are deleted yields a solution to the topological sorting problem. The application of this algorithm to the same digraph representing the five courses is given in Figure 4.8. Note that the solution obtained by the source-removal algorithm is different from the one obtained by the DFS-based algorithm. Both of them are correct, of course; the topological sorting problem may have several alternative solutions.

Generating Permutations and Subsets

The most important types of combinatorial objects are permutations, combinations, and subsets of a given set. There are n! permutations of {1, 2, . . . , n}. It is important in many instances to generate a list of such permutations.

Generating Permutations

We are given a sequence of numbers from 1 to n. Each permutation in the sequence that we need to generate should differ from the previous permutation by swapping just two adjacent elements of the sequence.

Input : n = 3

Output : 123 132 312 321 231 213

Input : n = 4

Output : 1234 1243 1423 4123 4132

1432 1342 1324 3124 3142 3412 4312

4321 3421 3241 3214 2314 2341 2431

4231 4213 2413 2143 2134

The Johnson and Trotter algorithm doesn't require to store all permutations of size n-1 and doesn't require going through all shorter permutations. Instead, it keeps track of the direction of each element of the permutation.

1. Find out the largest mobile integer in a particular sequence. **A directed integer is said to be mobile if it is greater than its immediate neighbor in the direction it is looking at**.
2. Switch this mobile integer and the adjacent integer to which its direction points.
3. Switch the direction of all the elements whose value is greater than the mobile integer value.
4. Repeat the step 1 until unless there is no mobile integer left in the sequence.

ALGORITHM *JohnsonTrotter*(n)

//Implements Johnson-Trotter algorithm for generating permutations
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$
initialize the first permutation with $\overleftarrow{1}\ \overleftarrow{2} \ldots \overleftarrow{n}$
while the last permutation has a mobile element **do**
    find its largest mobile element k
    swap k with the adjacent element k's arrow points to
    reverse the direction of all the elements that are larger than k
    add the new permutation to the list

Here is an application of this algorithm for $n = 3$, with the largest mobile element shown in bold:

$$\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{\mathbf{3}} \quad \overleftarrow{1}\ \overleftarrow{\mathbf{3}}\ \overleftarrow{2} \quad \overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{\mathbf{2}} \quad \overrightarrow{\mathbf{3}}\ \overleftarrow{2}\ \overleftarrow{1} \quad \overleftarrow{2}\ \overrightarrow{\mathbf{3}}\ \overleftarrow{1} \quad \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}.$$

ALGORITHM *LexicographicPermute*(n)

//Generates permutations in lexicographic order
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$ in lexicographic order
initialize the first permutation with $12 \ldots n$
while last permutation has two consecutive elements in increasing order **do**
    let i be its largest index such that $a_i < a_{i+1}$ $\quad //a_{i+1} > a_{i+2} > \cdots > a_n$
    find the largest index j such that $a_i < a_j$ $\quad //j \geq i + 1$ since $a_i < a_{i+1}$
    swap a_i with a_j $\quad //a_{i+1}a_{i+2} \ldots a_n$ will remain in decreasing order
    reverse the order of the elements from a_{i+1} to a_n inclusive
    add the new permutation to the list

Generating Subsets

knapsack problem, which asks to find the most valuable subset of items that fits a knapsack of a given capacity. The exhaustive-search approach to solving this problem discussed there was based on generating all subsets of a given set of items. In this section, we discuss algorithms
for generating all 2^n subsets of an abstract set A

n	subsets							
0	$\varnothing$							
1	$\varnothing$	$\{a_1\}$						
2	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

FIGURE 4.10 Generating subsets bottom up.

A more challenging question is whether there exists a minimal-change algorithm for generating bit strings so that every one of them differs from its immediate predecessor by only a single bit. (In the language of subsets, we want every subset to differ from its immediate predecessor by either an addition or a deletion, but not both, of a single element.) The answer to this question is yes. For example, for $n = 3$, we can get

$$000 \quad 001 \quad 011 \quad 010 \quad 110 \quad 111 \quad 101 \quad 100.$$

Such a sequence of bit strings is called Binary reflected Gray code.

ALGORITHM *BRGC*(n)

//Generates recursively the binary reflected Gray code of order n
//Input: A positive integer n
//Output: A list of all bit strings of length n composing the Gray code
if $n = 1$ make list L containing bit strings 0 and 1 in this order
else generate list $L1$ of bit strings of size $n - 1$ by calling $BRGC(n - 1)$
    copy list $L1$ to list $L2$ in reversed order
    add 0 in front of each bit string in list $L1$
    add 1 in front of each bit string in list $L2$
    append $L2$ to $L1$ to get list L
return L

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Divide and Conquer Approach
Binary Search

Dr. Shylaja S S

Divide-and-Conquer Approach

Divide-and-Conquer is probably the best-known general algorithm design technique. Though its fame may have something to do with its catchy name, it is well deserved: quite a few very efficient algorithms are specific implementations of this general strategy. Divide-and-conquer algorithms work according to the following general plan:

1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. The smaller instances are solved (typically recursively, though sometimes a different algorithm is employed when instances become small enough).
3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.

The divide-and-conquer technique is diagrammed in Fig. 1, which depicts the case of dividing a problem into two smaller sub problems, by far the most widely occurring case.

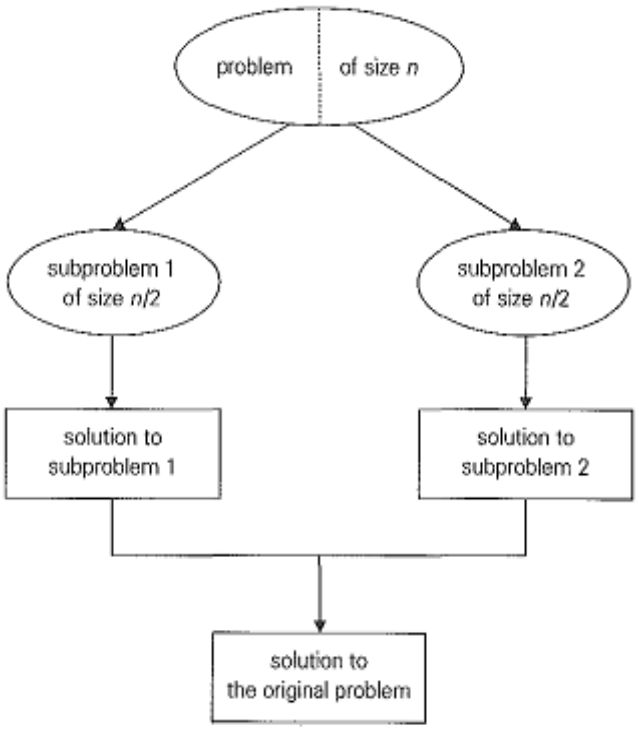


Fig. 1: Divide-and-conquer technique (typical case)

General Divide and Conquer Recurrence

In the most typical cases 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 >= 1 and b >= 1. Assuming that size n is a power of b, we get the following recurrence for the running time:

$T(n) = a * T(n/b) + f(n)$

f(n) is a function that accounts for the time spent on dividing the problem and combining the solutions. The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem.

Master Theorem

For the recurrence:

$$T(n) = a * T(n/b) + f(n)$$

If $f(n) \in \Theta(n^d)$, where d >= 0 in the recurrence relation, then:

If $a < b^d$, $T(n) \in \Theta(n^d)$

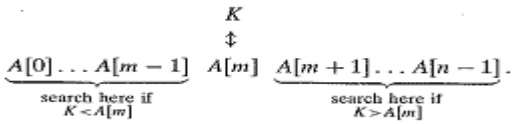
If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Analogous results hold for O and Ω as well!

Binary Search

Binary Search is a remarkably efficient algorithm for searching in a sorted array. It works by comparing the search key K with the array's middle element A[m]. If they match, the algorithm stops. Otherwise, the same operation is repeated recursively for the first half of the array if K < A[m] and for the second half if K > A[m].



As an example, let us apply binary search to searching for K = 70 in the array. 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
iteration1	l						m						r
iteration2								l		m			r
iteration3								l,m	r				

Though binary search is clearly based on a recursive idea, it can be easily implemented as a non recursive algorithm, too. Here is a pseudo code for this non recursive version.

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

// Implements non recursive 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 ← 0; r ← n-1

while l ≤ r do

$m \leftarrow \lfloor (l + r)/2 \rfloor$

if K = A[m] return m

else if K < A[m] r ← m-1

else l←m+1

return -1

Binary Search Analysis

Worst Case: The basic operation is the comparison of the search key with an element of the array. The number of comparisons made is given by the following recurrence:

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

For the initial condition $C_{worst}(1) = 1$, we obtain:

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

For any arbitrary positive integer, n:

$$\boldsymbol{C_{worst}(n) = \lfloor \log_2 n \rfloor + 1}$$

Average Case:

$$\boldsymbol{C_{avg} \approx \log_2 n}$$

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Merge Sort

Dr. Shylaja S S

Merge Sort

Mergesort is a perfect example of a successful application of the divide-and-conquer technique. It sorts a given array A[0 .. n - 1] by dividing it into two halves A[0 .. $\lfloor n/2 \rfloor$ -1] and A[$\lfloor n/2 \rfloor$ … n-1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

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)

The merging of two sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

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]
```

The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 is illustrated in Fig. 1.

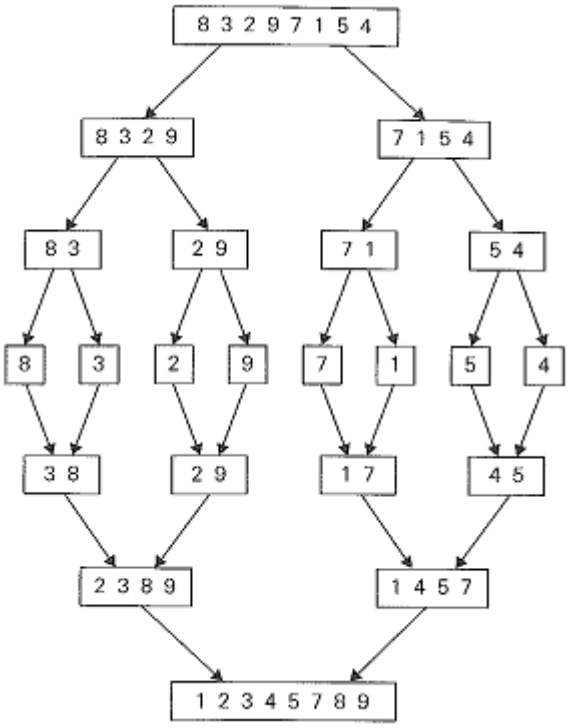


Fig. 1: Example of mergesort operation

Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons C(n) is:

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

The number of key comparisons performed during the merging stage in the worst case is:

$C_{merge}(n) = n - 1$

Using the above equation:

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

Applying Master Theorem to the above equation:

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

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Quick Sort

Dr. Shylaja S S

Quick Sort

Quicksort is another important sorting algorithm that is based on the divide-and-conquer approach. Unlike mergesort, which divides its input's elements according to their position in the array, quicksort divides them according to their value. Specifically, it rearranges elements of a given array A[0 .. n - 1] to achieve its partition, a situation where all the elements before some position s are smaller than or equal to A[s] and all the elements after positions are greater than or equal to A[s]:

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

Obviously, after a partition has been achieved, A[s] will be in its final position in the sorted array, and we can continue sorting the two sub arrays of the elements preceding and following A[s] independently (e.g., by the same method).

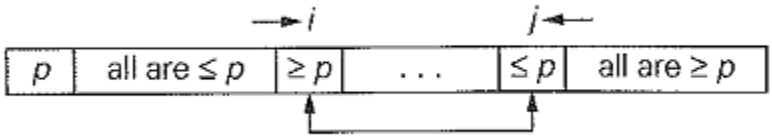
```
ALGORITHM  Quicksort(A[l .. r])
// Sorts a subarray by quicksort
// Input: A subarray A[l … r] of A[0 .. n  -1], defined by its left and right indices l
//and r
// Output: Subarray A[l .. r]  sorted in non decreasing order
if l  <  r
        s ←Partition(A[l .. r])       //s is a split position
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

A partition of A [0 ... n- 1] and, more generally, of its subarray A[l .. r] (0 ≤ l < r ≤ n - 1) can be achieved by the following algorithm. First, we select an element with respect to whose value we are going to divide the subarray. Because of its guiding role, we call this element the pivot. There are several different strategies for selecting a pivot. For now, we use the simplest strategy of selecting the subarray's first element: p = A[l].

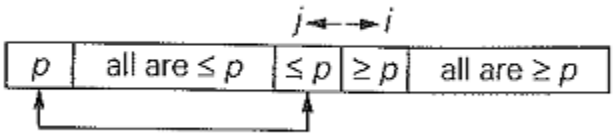
There are also several alternative procedures for rearranging elements to achieve a partition. Here we use an efficient method based on two scans of

the subarray: one is left-to-right and the other right-to-left, each comparing the sub-array's elements with the pivot. The left-to-right scan, denoted below by index i, starts with the second element. Since we want elements smaller than the pivot to be in the first part of the subarray, 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, denoted below by index j, starts with the last element of the subarray. Since we want elements larger than the pivot to be in the second part of the sub array, 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.

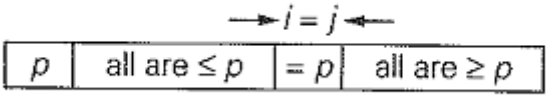
After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed. If scanning indices i and j have not crossed, i.e., i < j, we simply exchange A[i] and A[j] and resume the scans by incrementing i and decrementing j, respectively:



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



Finally, if the scanning indices stop while pointing to the same element, i.e., i = j, the value they are pointing to must be equal top (why?). Thus, we have the array partitioned, with the split positions s = i = j:



We can combine the last case with the case of crossed-over indices (i > j) by exchanging the pivot with A[j] whenever i ≥j.

Here is a pseudocode implementing this partitioning procedure.

```
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 ←A[l]
i ← l;  j ← r  + 1
repeat
        repeat i ← i + 1 until A[i] ≥ p
        repeat j ← j  - 1 until A[j] ≤ p
        swap(A[i],  A[j])
until i ≥ j
swap(A[i],  A[j])       //undo last swap when i ≥ j
swap(A[i],  A[j])
return  j
```

Note that index i can go out of the subarray bounds in this pseudocode. Rather than checking for this possibility every time index i is incremented, we can append to array A[0 .. n- 1] a "sentinel" that would prevent index i from advancing beyond position n. The more sophisticated method of pivot selection mentioned at the end of the section makes such a sentinel unnecessary.

An example of sorting an array by quicksort is given in Fig. 1.

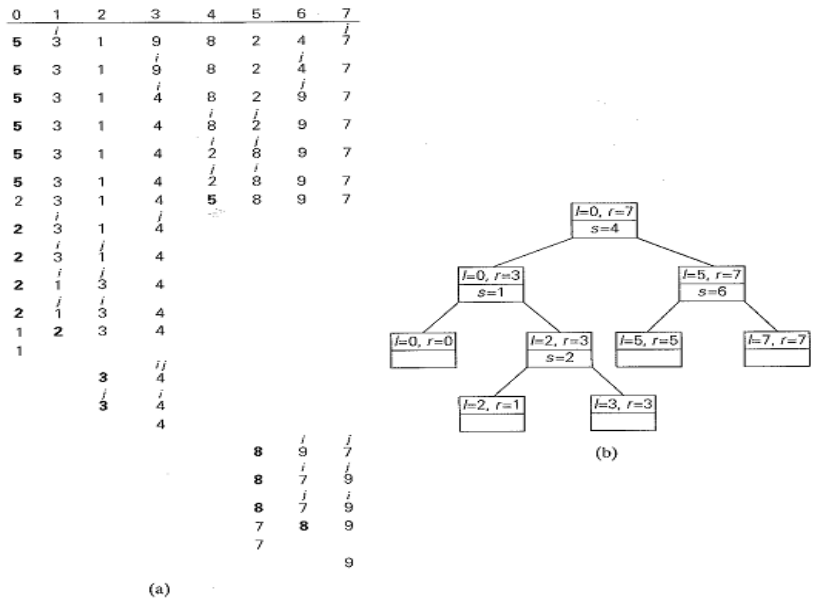


Fig. 1: Example of Quicksort operation. (a) The array's transformations with pivots shown in bold. (b) The tree of recursive calls to Quicksort with input values l and r of subarray bounds and split positions of a partition obtained.

Quick sort Analysis:

Best Case: The number of comparisons in the best case satisfies the recurrence:

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

According to Master Theorem

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

Worst Case: The number of comparisons in the worst case satisfies the recurrence:

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

Average Case: Let $C_{avg}(n)$ be the number of key comparisons made by Quick Sort on a randomly ordered array of size n.

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

The solution for the above recurrence is:

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

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Binary Tree

Dr. Shylaja S S

Binary Tree

A binary tree T is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees T_L and T_R called as the left and right subtree of the root. The definition itself divides the Binary Tree into two smaller structures and hence many problems concerning the binary trees can be solved using the Divide – And – Conquer technique. The binary tree is a Divide – And – Conquer ready structure.

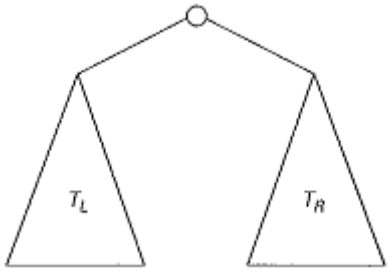


Fig. 1: Standard representation of a Binary Tree

Height of a Binary Tree: Length of the longest path from root to leaf

ALGORITHM Height(T)

//Computes recursively the height of a binary tree

//Input: A binary tree T

//Output: The height of T

if T = Ø return -1

else return max(Height(T_L), Height(T_R))+ 1

Height of a Binary Tree Analysis:

We measure the problem's instance size by the number of nodes n(T) in a given binary tree T. Obviously, the number of comparisons made to compute the maximum of two numbers and the number of additions A(n(T)) made by the algorithm are the same.

We have the following recurrence relation for A(n(T)):

$A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1$ for $n(T) > 0$,

$A(0) = 0$.

Before we solve this recurrence (can you tell what its solution is?), let us note that addition is not the most frequently executed operation of this algorithm. What is? Checking-and this is very typical for binary tree algorithms-that the

tree is not empty. For example, for the empty tree, the comparison T = Ø is executed once but there are no additions, and for a single-node tree, the comparison and addition numbers are three and one, respectively.

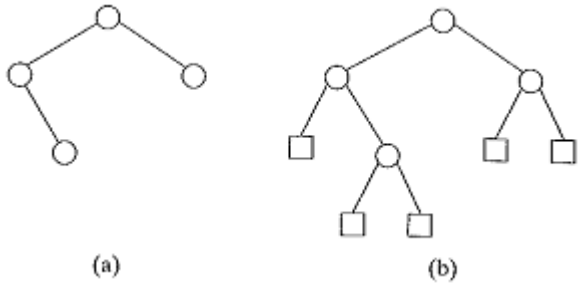


Fig. 2: (a) Binary tree. (b) Its extension. Internal nodes are shown as circles; external nodes are shown as squares.

It helps in the analysis of tree algorithms to draw the tree's extension by replacing the empty subtrees by special nodes. The extra nodes (shown by little squares in Fig. 2) are called external; the original nodes (shown by little circles) are called internal. By definition, the extension of the empty binary tree is a single external node.

It is easy to see that the height algorithm makes exactly one addition for every internal node of the extended tree, and it makes one comparison to check whether the tree is empty for every internal and external node. Thus, to ascertain the algorithm's efficiency, we need to know how many external nodes an extended binary tree with n internal nodes can have. Checking Fig. 2 and a few similar examples, it is easy to hypothesize that the number of external nodes x is always one more than the number of internal nodes n:

$$x=n+1 \qquad \text{---------- (1)}$$

To prove this formula, consider the total number of nodes, both internal and external. Since every node, except the root, is one of the two children of an internal node, we have the equation

$$2n + 1 = x + n,$$

which immediately implies equation (1).

Note that equation (1) also applies to any nonempty full binary tree, in

which, by definition, every node has either zero or two children: for a **full binary tree**, n and x denote the numbers of parental nodes and leaves, respectively.

Returning to algorithm Height, the number of comparisons to check whether the tree is empty is

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

while the number of additions is

$$A(n) = n$$

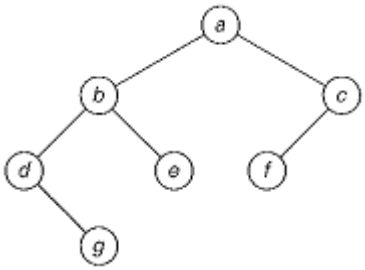
Binary Tree Traversal

The most important divide-and-conquer algorithms for binary trees are the three classic traversals: preorder, inorder, and postorder. All three traversals visit nodes of a binary tree recursively, i.e., by visiting the tree's root and its left and right subtrees. They differ just by the timing of the root's visit:

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).



Preorder: a, b, d, g, e, c, f
Inorder: d, g, b, e, a, f, c
Postorder: g, d, e, b, f, c, a

Fig. 3: Binary tree and its traversals

Algorithm Inorder(T)
if T ≠ Ø
    Inorder(T_{left})
    print(root of T)
    Inorder(T_{right})

Algorithm Preorder(T)
if T ≠ Ø
    print(root of T)
    Preorder(T_{left})
    Preorder(T_{right})

Algorithm Postorder(T)
if T ≠ Ø
    Postorder(T_{left})
    Postorder(T_{right})
    print(root of T)

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Multiplication of Large Integers and Strassen's Matrix Multiplication

Dr. Shylaja S S

Multiplication of large Integers

Some applications, notably modern cryptology, require manipulation of integers that are over 100 decimal digits long. Since such integers are too long to fit in a single word of a modern computer, they require special treatment. This practical need supports investigations of algorithms for efficient manipulation of large integers. In this section, we outline an interesting algorithm for multiplying such numbers. Obviously, if we use 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 the total of n^2 digit multiplications. (If one of the numbers has fewer digits than the other, we can pad a shorter number with leading zeros to equal their lengths.) Though it might appear that it would be impossible to design an algorithm with fewer than n^2 digit multiplications, it turns out not to be the case. The miracle of divide-and-conquer comes to the rescue to accomplish this feat.

To demonstrate the basic idea of the algorithm, let us start with a case of two - digit integers, say, 23 and 14. These numbers can be represented as follows:

$23 = 2. 10^1 + 3. 10^0$ and $14 = 1. 10^1 + 4. 10^0$

Now let us multiply them:

$23 * 14 = (2 . 10^1 + 3. 10^0) * (1 . 10^1 + 4. 10^0)$

$= (2 * 1)10^2 + (2 * 4 + 3 * 1)10^1 + (3 * 4)10^0.$

The last formula yields the correct answer of 322, of course, but it uses the same four digit multiplications as the pen-and-pencil algorithm. Fortunately, we can compute the middle term with just one digit multiplication by taking advantage of the products 2 * 1 and 3 * 4 that need to be computed anyway:

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

Of course, there is nothing special about the numbers we just multiplied. 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

$c_2 = a_1 * b_1$ 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.

Now we apply this trick to multiplying two n-digit integers a and b where n is a positive even number. Let us divide both numbers in the middle-after all, we promised to take advantage of the divide-and-conquer technique. We denote the first half of the a's digits 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_1a_0 implies that a= $a_1 10^{n/2} + a_0$, and b = b_1b_0 implies that b = $b_1 10^{n/2} + b_0$. Therefore, taking advantage of the same trick we used for two-digit numbers, 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

If n/2 is even, we can apply the same method for computing the products c_2, c_0, and c_1. Thus, if n is a power of 2, we have a recursive algorithm for computing the product of two n-digit integers. In its pure form, the recursion is stopped when n becomes one. It can also be stopped when we deem n small enough to multiply the numbers of that size directly.

How many digit multiplications does this algorithm make? Since multiplication of n-digit numbers requires three multiplications of n/2-digit numbers, the recurrence for the number of multiplications M (n) will be

$M(n) = 3M(n/2)$ for $n > 1$, $M(1) = 1$.

Solving it by backward substitutions for $n = 2^k$ yields

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

$= \ldots = 3^i M(2^{k-i}) = \ldots = 3^kM(2^{k-k}) = 3^k$

Since $k = \log_2 n$: $M(n) = 3^{\log_2 n} = n^{\log_2 3} = n^{1.585}$

Strassen's Matrix Multiplication

Now that we have seen that the divide-and-conquer approach can reduce the number of one-digit multiplications in multiplying two integers, we should not be surprised that a similar feat can be accomplished for multiplying

matrices. Such an algorithm was published by V. Strassen in 1969. The principal insight of the algorithm lies in the discovery that we can find the product C of two 2-by-2 matrices A and B with just seven multiplications as opposed to the eight required by the brute-force algorithm. This is accomplished by using the following formulas:

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

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

where,

$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$

$m_2 = (a_{10} + a_{11}) * b_{00}$

$m_3 = a_{00} * (b_{01} - b_{11})$

$m_4 = a_{11} * (b_{10} - b_{00})$

$m_5 = (a_{00} + a_{01}) * b_{11}$

$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$

$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

Thus, to multiply two 2-by-2 matrices, Strassen's algorithm makes seven multiplications and 18 additions/subtractions, whereas the brute-force algorithm requires eight multiplications and four additions. These numbers should not lead us to multiplying 2-by-2 matrices by Strassen's algorithm. Its importance stems from its asymptotic superiority as matrix order n goes to infinity.

Let A and B be two n-by-n matrices where n is a power of two. (If n is not a power of two, matrices can be padded with rows and columns of zeros.) We can divide A, B, and their product C into four n /2-by-n /2 submatrices each as follows:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$

It is not difficult to verify that one can treat these submatrices as numbers to get the correct product. For example, C_{00} can be computed either as

$A_{00} * B_{00} + A_{01} * B_{10}$ or as $M_1 + M_4 - M_5 + M_7$ where M_1, M_4, M_5, and M_7 are found by Strassen's formulas, with the numbers replaced by the corresponding submatrices. If the seven products of n/2-by-n/2 matrices are computed recursively by the same method, we have Strassen's algorithm for matrix multiplication.

Let us evaluate the asymptotic efficiency of this algorithm. If M(n) is the number of multiplications made by Strassen's algorithm in multiplying two n-by-n matrices (where n is a power of 2), we get the following recurrence relation for it:

$M(n)=7M(n/2)$ for $n> 1$, $M(1)=1$.

Since $n = 2^k$,

$$M(2^k) = 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2M(2^{k-2}) = \ldots$$

$$= 7^i M(2^{k-i}) \ldots = 7^k M(2^{k-k}) = 7^k$$

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.

Since this saving in the number of multiplications was achieved at the expense of making extra additions, we must check the number of additions A(n) made by Strassen's algorithm. To multiply two matrices of order $n > 1$, the algorithm needs to multiply seven matrices of order n/2 and make 18 additions of matrices of size n/2; when $n = 1$, no additions are made since two numbers are simply multiplied.
These observations yield the following recurrence relation:
$A(n) = 7\,A(n/2) + 18(n/2)^2$ for $n>1$, $A(1) =0$

According to the Master Theorem, $A(n) \in \Theta(n^{\log_2 7})$. In other words, the number of additions has the same order of growth as the number of multiplications. This puts Strassen's algorithm in $\Theta(n^{\log_2 7})$, which is a better efficiency class than $\Theta(n^3)$ of the brute-force method.