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.

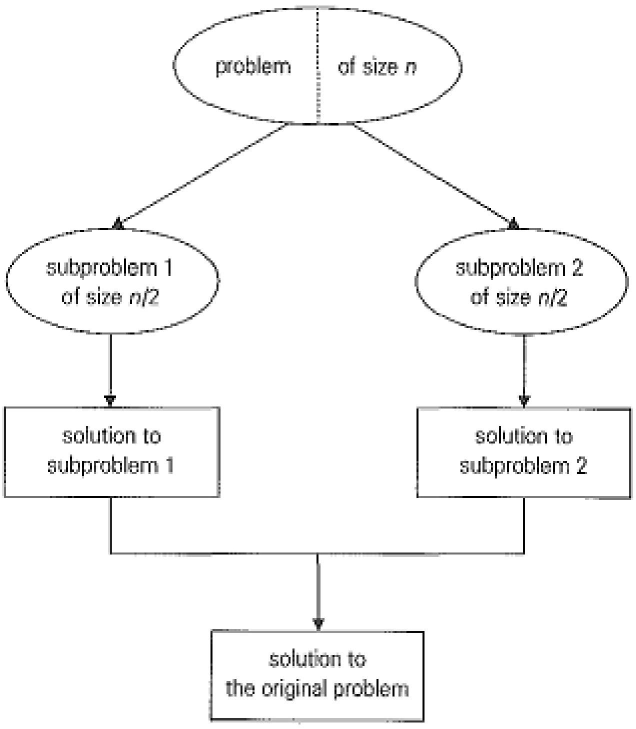

Fig. 1: Divide-and-conquer technique (typical case)

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 $f(n) \in \Theta(nd)$, 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].

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

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

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

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

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

For any arbitrary positive integer, n:

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

Average Case:

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