Divide-and-Conquer Algorithm

Divide-and-conquer is a top-down technique for designing algorithms that consists of dividing the problem into smaller subproblems hoping that the solutions of the subproblems are easier to find and then composing the partial solutions into the solution of the original problem.

Little more formally, divide-and-conquer paradigm consists of following major phases:

- Breaking the problem into several sub-problems that are similar to the original problem but smaller in size,
- Solve the sub-problem recursively (successively and independently), and then
- Combine these solutions to subproblems to create a solution to the original problem.

Binary Search (simplest application of divide-and-conquer)

Binary Search is an extremely well-known instance of divide-and-conquer paradigm. Given an ordered array of n elements, the basic idea of binary search is that for a given element we "probe" the middle element of the array. We continue in either the lower or upper segment of the array, depending on the outcome of the probe until we reached the required (given) element.

Problem Let $A[1 \ldots n]$ be an array of non-decreasing sorted order; that is $A[i] \leq A[j]$ whenever $1 \leq i \leq j \leq n$. Let $'q'$ be the query point. The problem consist of finding $'q'$ in the array A. If q is not in A, then find the position where $'q'$ might be inserted.

Formally, find the index i such that $1 \leq i \leq n+1$ and $A[i\text{-}1] < x \leq A[i]$.

Sequential Search

Look sequentially at each element of A until either we reach at the end of an array A or find an item no smaller than 'q'.

Sequential search for 'q' in array A

```
for i = 1 to n do
    if A [i] ≥ q then
        return index i
    return n + 1
```

Analysis

This algorithm clearly takes a $\theta(r)$, where r is the index returned. This is $\Omega(n)$ in the worst case and $O(1)$ in the best case.

If the elements of an array A are distinct and query point q is indeed in the array then loop executed $(n + 1) / 2$ average number of times. On average (as well as the worst case), sequential search takes $\theta(n)$ time.

Binary Search

Look for $'q'$ either in the first half or in the second half of the array A. Compare $'q'$ to an element in the middle, $\lceil n/2 \rceil$, of the array. Let $k = \lceil n/2 \rceil$. If $q \leq A[k]$, then search in the $A[1 \ldots k]$; otherwise search $T[k+1 \ldots n]$ for $'q'$. Binary search for q in subarray $A[i \ldots j]$ with the promise that

$$A[i-1] < x \leq A[j]$$

```
If i = j then
    return i (index)
k= (i + j)/2
```

```
if q ≤ A [k]
    then return Binary Search [A [i-k], q]
    else return Binary Search [A[k+1 . . j], q]
```

Analysis

Binary Search can be accomplished in logarithmic time in the worst case , i.e., $T(n) = \theta(log\ n)$. This version of the binary search takes logarithmic time in the best case.

Iterative Version of Binary Search

Interactive binary search for q, in array $A[1\ .\ .\ n]$

```
if q > A [n]
    then return n + 1
i = 1;
j = n;
while i < j do
    k = (i + j)/2
    if q ≤ A [k]
        then j = k
        else i = k + 1
return i (the index)
```

Analysis

The analysis of Iterative algorithm is identical to that of its recursive counterpart.


Back