Divide-and-conquer Algorithms
Binary Search

Divide-and-Conquer Algorithms

Divide-and-Conquer: is an algorithmic technique that solves a computational problem by:

1. Breaking it into *subproblems* that are **smaller instances** of the same type of problem
2. Recursively solving these subproblems
3. Constructing the solution to the problem from the solutions of the subproblems.

Binary Search

Search Problem. Given an array $A[1..N]$ of integers *sorted in ascending order*, and an integer x, return **true** if x is found in A, and **false** otherwise.

Naïve Solution. The straightforward or naïve solution is to start with the first element of A and compare all elements of the array to x until we either find x or we find an element that is larger than x.

The pseudocode is below.

```
ALGORITHM NaiveSearch(N,A[1..N], x)
begin
   for i ← 1 to N do
     if x = A[i] then return(true);
     if x < A[i] then return(false);
  end for
  return(false);
end
```

Worst-Case Complexity of NaiveSearch. Let $x \notin A$ and let $A[N] < x$. Then Algorithm NaiveSearch (as written above) will take $2N$ comparisons.

Proof. Straightforward. The first comparison in the loop will always be false because no element of A is equal to x. The second comparison will always be false becase all elements of A are smaller than x (by our assumption). Therefore, none of the **return** statements inside the loop will be executed, and thus, the loop will execute N times. This implies that there execution will involve $2N$ comparisons.

Divide-and-Conquer solution. A better solution is to do the following:

- Compare the middle point of the array $A[\lceil \frac{N}{2} \rceil]$ to x.
- If $A[\lceil \frac{N}{2} \rceil] = x$, stop and return **true**.
- If $A[\lceil \frac{N}{2} \rceil] > x$, then x can only be found in the first half of A. Search for x *recursively* in $A[1..\lceil \frac{N}{2} \rceil - 1]$.
- If $A[\lceil \frac{N}{2} \rceil] < x$, then x can only be found in the second half of A. Search for x *recursively* in $A[\lceil \frac{N}{2} \rceil + 1, N]$.

An algorithm that solves the Search problem using this approach is called Binary Search.

The exact pseudocode is in Figure 1. To solve the problem, BinarySearch algorithm is called as

BinarySearch(1,N,N,A[1..N],x)

The first two input parameters are the range of indexes in array A over which the search is to be conducted in the current call.

Algorithm Correctness. We prove that the algorithm is correct by induction.

Base Case. $N = 1$. In this case, the only acceptable call is BinarySearch(1,1,1,A[1..1],x). The first comparison of the algorithm ($I = J$) will be evaluated to true ($I = 1$, $J = 1$). If $A[1]$ is indeed x, the $A[I] = x$ comparison will evaluate to true and correct answer will be returned. If $A[1]$ id not x, then $A[I] = x$ will evaluate to false and the correct answer will be returned as well.

```
ALGORITHM BinarySearch(I,J, N,A[1..N], x)
begin

  if I = J then      // Base Case
   if A[I] = x then
    return(true)
   else
    return(false)
  else   // inductive case

  c ← ⌈I + (J−I)/2⌉;
  if x = A[c] then
   return(true);
  else
   if x < A[c] then
    return BinarySearch(I, c-1, N, A, x);
   else   // x > A[c]
    return BinarySearch(c+1, J, N, A, x);
   end if
  end if

end
```

Figure 1: Algorithm BinarySearch

Induction Step. Assume $N > 1$ and assume that for all I, J such that $J - I < N$ BinarySearch(I,J, A[1..N],x) returns the correct result. Consider the algorithm call BinarySearch(1,N,A[1..N],x).

Because $N > 1$, the first comparison, $I = J$ will evaluate to false. The next statement will discover the midpoint between 1 and N and assign its value to c. If $A[c]$ contains x, then the followup comparison will evaluate to true and the algorithm will return the correct answer. Otherwise, if x is smaller than $A[c]$, then x may not be found an any element of the array A with an index greater than c, because A is sorted in ascending order. Therefore, by inductive hypothesis, the call to BinarySearch(1,c-1, A[1..N],x) will return the correct answer.

Similarly, if $x > A[c]$, then, x will not be found in elements of A indexed 1 through $c - 1$, and therefor the correct answer will be discovered by the BinarySearch(c+1,J,A[1..N],c) call.

Computational Complexity. Let us represent the running time of the BinarySearch algorithm in terms of the number of comparison operations that it takes to complete the computation. Let $T(n)$ be the running time of the algorithm on the input array of size n. Then, we can write the following relationships:

$$T(1) = 2$$

$$T(n) = T(\lceil \frac{n}{2} \rceil) + 3$$

We can apply the master theorem here:

$f(n) = 3 = \Theta(1)$
$a = 1$
$b = 2$

We have $f(n) = \Theta(n^{\log_b(a)}) = \Theta(n^{\log_2(1)}) = \Theta(n^0) = \Theta(1)$.

Therefore,

$$T(n) = \Theta(n^{\log_b(a)} \log_2(n)) = \log_2(n)$$