1. Algorithm Design Methods:

1.1. Divide-and-Conquer Method:

In the divide-and-conquer approach to solve a large problem, we divide it into some number of smaller problems; solve each of these; and combine these solutions to obtain the solution to the original problem. Often, the generated sub-problems are simply smaller instances of the original problem and may be solved using the divide-and-conquer strategy recursively. The divide-and-conquer approach is a top-down approach. That is, the solution to a top-level instance of a problem is obtained by going down and obtaining solutions to smaller instances.

The divide-and-conquer design strategy involves the following steps:

1. Divide an instance of a problem into one or more smaller instances.
2. Conquer (solve) each of the smaller instances. Unless a smaller instance is sufficiently small, use recursion to do this.
3. If necessary, combine the solutions to the smaller instances to obtain the solution to the original instance.

The reason we say "if necessary" in Step 3 is that in algorithms such as Binary Search the instance is reduced to just one smaller instance, so there is no need to combine solutions.

The abstracted procedure for the divide-and-conquer method as follow:

```
Procedure DandC(p);
begin
  If small (p) then  solve (p);
  Else begin
          Divide p into smaller instance p1, p2, p3, …, pk,
          Apply DandC to each of these subproblems;
         Combine( DandC(p1), DandC(p2), DandC(p3), …, DandC(pk));
```

end;

end;

If the size of the problem p is n and the sizes of sub-problems p_1, p_2, p_3, …, p_k, are n_1, n_2, n_3, …, n_k respectively, then the time complexity for the divide-and-conquer strategy is described as follow:

$$T(n) = \begin{cases} g(n) & \text{if } n \text{ small} \\ T(n_1) + T(n_2) + T(n_3) + \ldots + T(n_k) + f(n) & \text{otherwise} \end{cases}$$

Where $T(n)$: is the execution time of DandC for any inputs with size n.

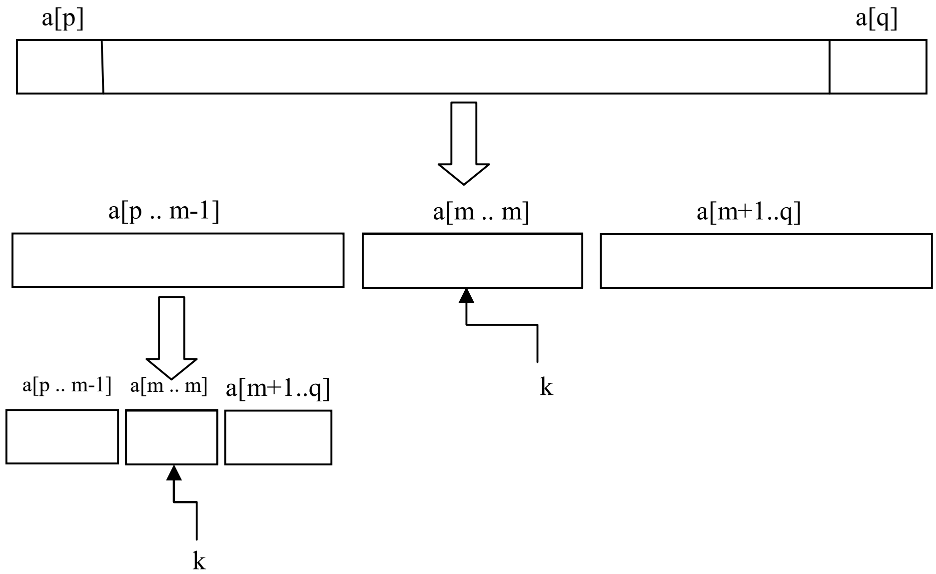
g(n): the computing time of response for small problem.

f(n): the time of dividing and/or combining the problem p to its sub-problems.

EXAMPLES:

1. Binary search:

To locate the element k in sorted list a[1..n]



The idea: To locate the element k in the a[p..q] we locate the k in three sub-list a[p..m-1] , a[m..m], and a[m+1..q]. By comparing k with a[m] two of the sub-list will be removed.

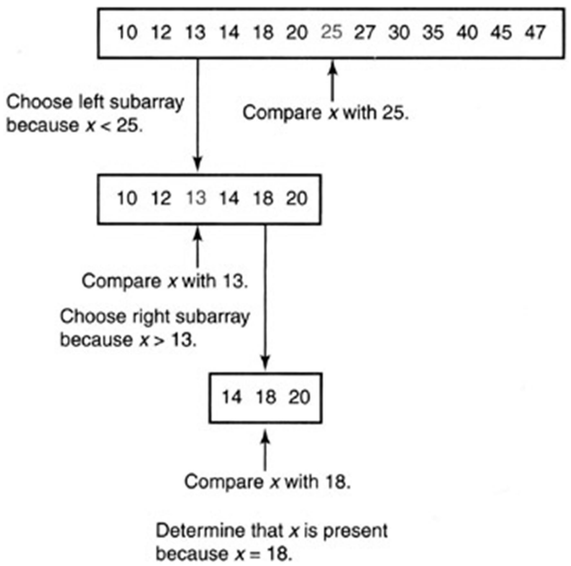


Figure 2.1: The steps done by a human when searching with Binary Search. (*Note*: x = 18.)

A recursive version of Binary Search now follows.

Here we design an algorithm for binary search by using divide-and-conquer method.

```
Function BinSearch(var a:ElemList; p, q:integer; k: Key): integer;
var  m: integer;
Begin
    m := (p +q) div 2;
    If p > q then
       BinSearch:= 0;
    Else if k = a[m] then   BinSearch:= m
   Else if k > a[m] then   BinSearch:= BinSearch(a, m+1, q, k)
   Else   BinSearch:= BinSearch(a, p, m-1, k);
End;
```

Tracing of Algorithm:

Locate the values 101, -14, and 82 in the following sorted list

A[1..9] = (-15, -6, 0, 7, 9, 23, 54, 82, 101)

Locations: 1 2 3 4 5 6 7 8 9

k = 101			k = -14			k = 82		
P	Q	M	P	Q	M	p	q	M
1	9	5	1	9	5	1	9	5
6	9	7	1	4	2	6	9	7
8	9	8	1	1	1	8	9	8
9	9	9	2	1		Found		
Found			Not Found					

Algorithm Analysis:

1- Space complexity:

Each activation requires 7 spaces (4 bytes for a, 4 bytes for each p,q,m, return address and BinSearch, and k)

1^{st} activation	2^{nd} activation	3^{th} activation	…	m^{th} activation
$n+1/2^1$	$n+1/2^2$	$n+1/2^3$	…	$n+1/2^m$

The last comparison is stopped when

$n + 1 = 2^m$

$\log_2 (n + 1) = \log_2 2^m$

$m = \log_2 n$

where n: the no. of comparisons, m: the no. of activations

$S_{\text{BinSearch}}(n) = \mathbf{\Theta}(\log_2 n)$

$S_{\text{BinSearch}}(n) = \mathbf{7} \log_2 n$

2- Time complexity:

$$T_{\text{BinSearch}}(n) = \begin{cases} T_{\text{BinSearch}}(1) & \text{if } n = 1 \\ T_{\text{BinSearch}}(n/2) + c & \text{otherwise} \end{cases}$$

$T_{\text{BinSearch}}(n) = T_{\text{BinSearch}}(n/2) + c$

$$= T_{\text{BinSearch}}\,(n / 2^2) + 2c$$

$$= T_{\text{BinSearch}}\,(n / 2^3) + 3c$$

$$= T_{\text{BinSearch}}\,(n / 2^m) + mc$$

Suppose $n = {}^{2m}$

$$m = \log_2 n$$

$$= T_{\text{BinSearch}}\,(n / 2^m) + mc$$

$$= T_{\text{BinSearch}}\,(1) + c \log_2 n$$

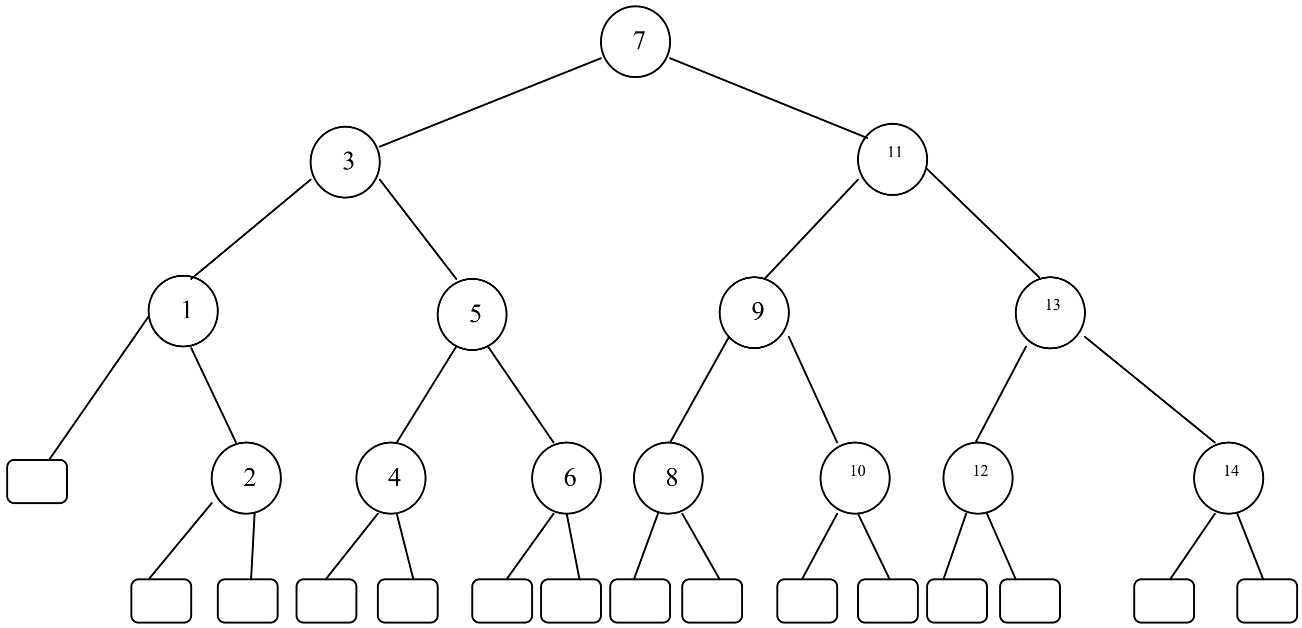
$$T_{\text{BinSearch}}\,(n) = \boldsymbol{\Theta}(\log_2 n)$$

This the worst case time complexity for the successful search of the binary search algorithm, while the best case time complexity is equal to $\boldsymbol{\Theta}(1)$.

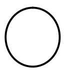
Example: Draw the binary search decision tree for a list of 14 elements and then find:

1- The maximum, minimum, and average number of comparisons for the successful search.

2- The average number of comparisons for the failure search.



binary search decision tree when n= 14

 Internal node (represent successful state)

External node (represent failure state)

The maximum number of comparisons for the successful search = 4 comparisons.

The minimum number of comparisons for the successful search = 1 comparison.

The average number of comparisons for the successful search

$$= \frac{(1 * 0) + (2 * 1) + (4 * 2) + (7 * 3)}{1 + 2 + 4 + 7} = \frac{31}{14} = 2.214 \text{ comparisons}$$

The average number of comparisons for the failure search

$$= \frac{(1 * 3) + (14 * 4)}{1 + 14} = \frac{59}{15} = 3.933 \text{ comparisons}$$

The abstract:

The successful searches			The failure searches		
Best case	Average case	Worst case	Best case	Average case	Worst case
$\Theta(1)$	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$		