Searching Algorithms

Searching

Objective : To find out the location of any given element in an array. If element found then search is successful, otherwise search is unsuccessful.

Two approaches –

1. Linear Search (Sequential Search): if array elements are in random order
2. Binary Search: if array elements are in sorted order

Linear Search

Algorithm Linear_Search(a,n, Item): Let **a** be a given array and no. of elements is n. Pos is representing the position of desired element **Item** in array **a**, if element found otherwise, set Pos to -1. Variable i is used as simple variable in loop.

```
1.  Initialize Pos = -1                          C1
2.  For i = 0 to n-1                             C2
3.          If a[i] == Item                      C3
4.                  Set Pos = i                  C4
5.                    Break //break the loop     C5
6.  If Pos < 0                                   C6
7.          Print "Element is not found"         C7
8.  Else    Print Pos                            C8
```

Analysis of Linear Search

- **Best case: When searched Item is present at first element**

Time Complexity function=C1 + C2 + C3 + C4 + C5+C6+C8

=7

Best case time Complexity = O(1)

- **Worst case:**

  i. When searched Item present at last element

  Time Complexity function=C1 + n*C2 + n * C3 + C4 + C5+C6+C8

  =2n + 5= O(n)

  ii. When searched Item is not present

  Time Complexity function=C1+ (n+1)*C2 + n * C3 +C6+C7 = 2n + 4 **=**O(n)

Binary Search

- Best searching approach

Constraint:

- Given array should be in sorted order

  OR

- First sort the given array and then perform search, but then sorting time will be added

Binary Search (Iterative)

Algorithm Binary_Search(a,n, Item): Let a be a given array which is sorted in ascending order and no. of elements is n. Pos is representing the position of desired element *Item* in array a, if element found; otherwise, set Pos to -1. Variable mid is used to keep the mid index of array.

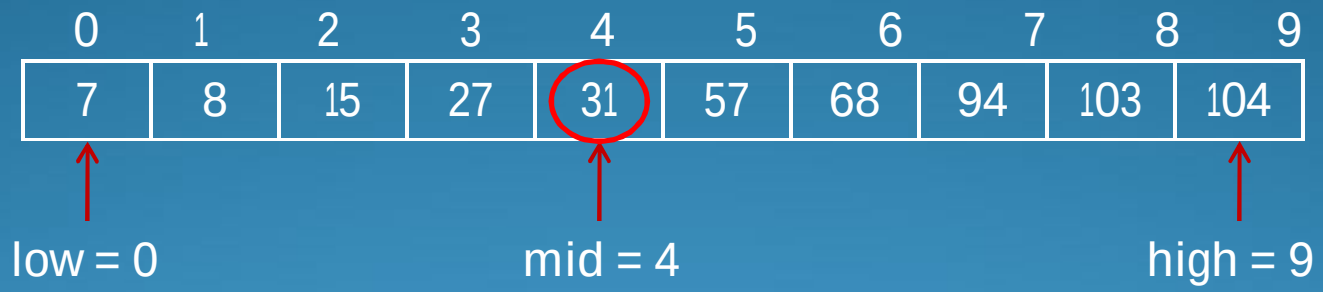
```
1.  Initialize low =0, high = n-1
2.  While low <= high
3.          mid = ⌊(low+high)/2⌋
4.          If a[mid] == Item
5.                  Set Pos = mid
6.                  Break and Jump to step 10
7.          Else if Item < a[mid]
8.                  high = mid -1
9.          Else    low =mid +1
10. If Pos < 0
11.         Print “Element is not found”
12. Else    Print Pos
```

Binary Search Illustration

mid = $\lfloor (low+high)/2 \rfloor$

= $\lfloor (0 + 9) /2 \rfloor$ = 4

item to be searched, Item = 94



If a[mid]<Item

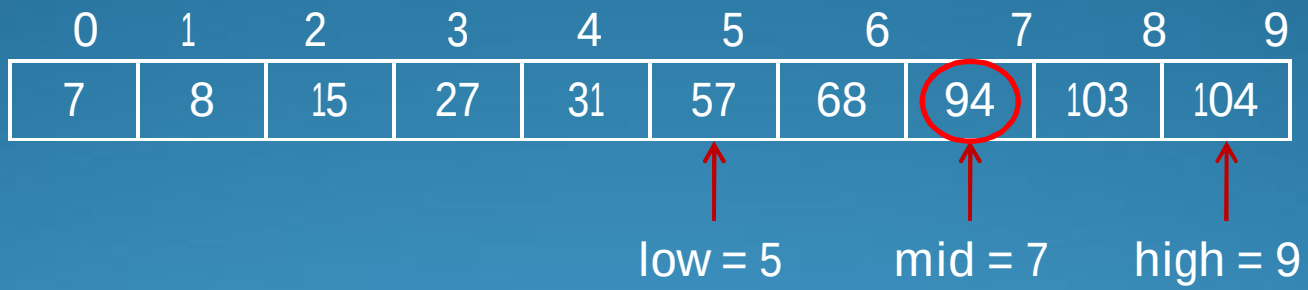
low = mid + 1

Binary Search Illustration

mid = $\lfloor (low+high)/2 \rfloor$

= $\lfloor (5 + 9) /2 \rfloor$ = 7

item to be searched, Item = 94



If a[mid]==Item

Pos = mid

Binary Search (Iterative)

```
1.  Initialize low =0, high = n-1                          C1
2.  While low <= high                                      C2
3.          mid = ⌊(low+high)/2⌋                           C3
4.          If a[mid] == Item                              C4
5.                  Set Pos = mid                          C5
6.                  Break and Jump to step 10              C6
7.          Else if Item < a[mid]                          C7
8.                  high = mid -1                          C8
9.          Else    low =mid +1                            C9
10. If Pos < 0                                             C10
11.         Print "Element is not found"                   C11
12. Else    Print Pos                                      C12
```

Analysis of Iterative Binary Search

DISTINCT & SORTED ELEMENTS:

On successful search

Time Complexity =

$= C1 + (\lfloor \log_2 n \rfloor)*C2 + \lfloor \log_2 n \rfloor * (C3 + C4 + C7 + C8 + C9) + C5 + C6 + C10 + C12$

$= 6 \lfloor \log_2 n \rfloor + 5$

$= O(\log_2 n)$

Unsuccessful search

Time Complexity $= C1 + (\lfloor \log_2 n \rfloor + 1)*C2 + \lfloor \log_2 n \rfloor * (C3 + C4 + C7 + C8 + C9) + C10 + C11$

$= 6 \log_2 n + 4$

$= O(\log_2 n)$

That means in worst case, at most **$\log_2 n$** number of comparisons are required to find out the desired element location which is better than the linear search.

What is the Best Case: When searched element is middle element; then O(1) time is required

Binary Search (Recursive)

Algorithm Bin_Search(a, Item,low, high): Let a is given array which is sorted in ascending order and no. of elements is n. Pos is representing the position of desired element *Item* in array a, if element found otherwise, set Pos to -1. Variable mid is used to keep the mid index of array. Here, initially low and high are assigned the lowest and highest index of array a.

```
1.  If low<=high
2.          mid = ⌊(low+high)/2⌋
3.          If a[mid] == Item
4.                      Set Pos = mid
5.                      Return Pos
6.          Else if Item < a[mid]
7.                      Return Bin_Search(a,Item,low,mid-1)
8.          Else        Return Bin_Search(a,Item,mid+1,high)
9.  Return Pos
```

Complexity analysis of an algorithm

Two main methods:

1. **Direct counting:** Time complexity will be sum of the individual steps (primitive Operations)

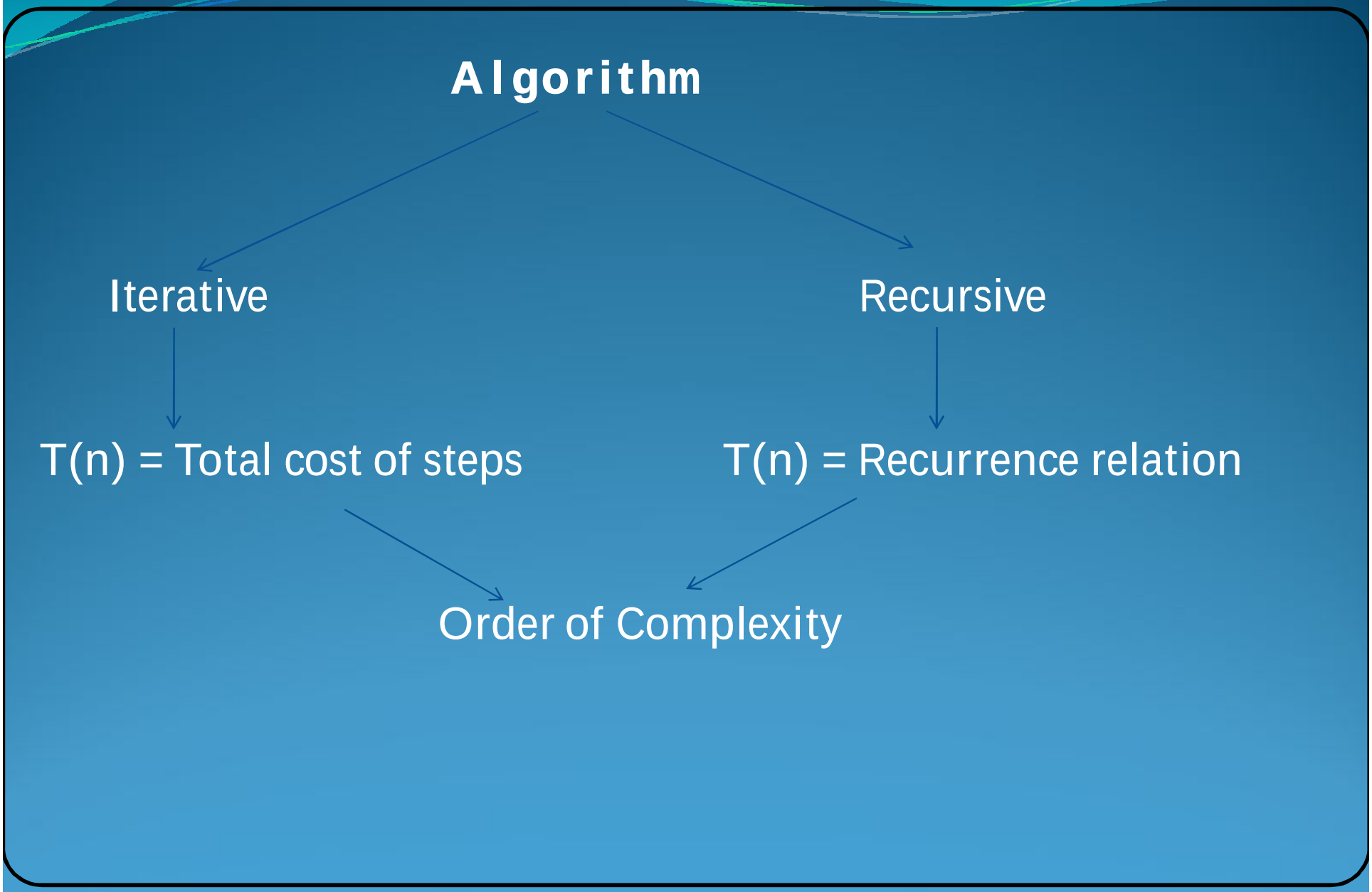
✓Best for repeated iterations (loops).

2. **Recurrence equation(*Recurrence Relations*): A**n equation describing the function in terms of its behavior on smaller inputs.

✓Best for recursive functions and structures.

Analyzing Performance of Non-Recursive Routines is (relatively) Easy

Recurrence Relation



Recurrence Relation (Cont…)

Let T(n) be a recurrence relation, then

T is a recurrence function and n is a parameter to function T

Therefore, T(n) is a Sequence term generated by function T for parameter n

Examples :

1.

$$T(n) = \begin{cases} 0 & \text{for } n=0 \\ T(n-1) + n & n>0 \end{cases}$$

2.

$$T(n) = \begin{cases} 1 & \text{for } n=0 \\ T(n/2) + 1 & n>0 \end{cases}$$

3.

$$T(n) = \begin{cases} 0 & \text{for } n=0 \\ 1 & n=1 \\ 2T(n-1) + 1 & n>1 \end{cases}$$

Recurrence Relation (Cont…)

Base Case:

- Recurrence relation must include two equations:

i. Equation for general case
ii. Equation for the base case

- The base case is often an O(1) operation,
- In some recurrence relations the base case involves input of size one, hence base case is like $T(1) = O(1)$.
- However, there are cases when the base case has size zero. In such cases the base could be $T(0) = O(1)$.

Solving Recurrence Relations

$T(n) = 2\,T(n/2) + O(n)$ // recursive case

$T(1) = O(1)$ // Base Case

We'll write n instead of $O(n)$ in the first line below because it makes the algebra much simpler.

$T(n) = 2\,T(n/2) + n$

$= 2\,[\mathbf{2\,T(n/4) + n/2}] + n$

$= 4\,T(n/4) + 2n$

$= 4\,[\mathbf{2\,T(n/8) + n/4}] + 2n$

$= 8\,T(n/8) + 3n$

$= 16\,T(n/16) + 4n$

.....

.....

$= 2^k\,T(n/2^k) + k\,n$

We know that $T(1) = 1$ and this is a way to end the derivation above.

In particular we want $T(1)$ to appear on the right hand side of the = sign.

This means we want:

$n/2^k = 1$ *OR* $n = 2^k$ *OR* $\log_2 n = k$

Continuing with the previous derivation we get the following since $k = \log_2 n$:

$= 2^k\,T(n/2^k) + k\,n$

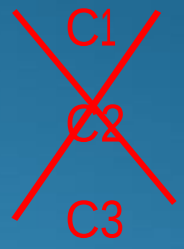
$= 2^{\log_2 n}\,T(1) + (\log_2 n)\,n = n + n \log_2 n$

$= O(n \log n)$

Illustration 1: Factorial Recursive

Algorithm Fact_Rec(n): Let n is number for which factorial is to be calculated.

```
1. If n==1
2.         Return 1
3. Return Fact_Rec(n-1) * n
```



Therefore,

$$T(n) = \begin{cases} 1 & \text{for } n=1 \\ T(n-1) + 1 & n>1 \end{cases}$$

Now,

How to solve this recurrence relation ?

Solving Recurrence Relation

Substitution method – try to find the pattern

While the equation certainly expresses the complexity function, a non-recursive (or *closed-form) version would be more useful.*

T(1) = 1

T(n) = T(n -1) + 1

T(n) = T(n -1) + 1 //T(n - 1) = T(n - 2) + 1

= T(n - 2) + 1 + 1

= T(n - 2) + 2 //T(n - 2) = T(n - 3) + 1

= T(n - 3) + 1 + 2

= T(n - 3) + 3 //T(n - 3) = T(n - 4) + 1

= T(n - 4) + 4

= : : :

= T(n - k) + k

If we set n-k=1=> k=n-1 we have:

T(n) = T(1) + n-1

= 1+n-1

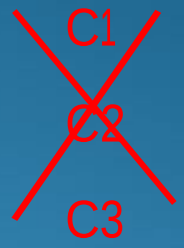
T(n)= n

Hence, Time complexity = O(n)

Illustration 2: Fibonacci Recursive

Algorithm Fibo_Rec(n): Let n is number means n^{th} is to be calculated.

```
1. If n==1
2.         Return 0
3. If n==2
4.         Return 1
5. Return Fibo_Rec(n-1) + Fibo_Rec(n-2)
```



Therefore,

$$T(n) = \begin{cases} 0 & \text{for } n=1 \\ 1 & n=2 \\ T(n-1) + T(n-2) + 1 & n>2 \end{cases}$$

Solving Recurrence Relation

This the case of **Deep recursion where** if the function makes more than one call to itself, you have a deep recursion, that takes exponential time.

- It grows like exponential tree.

Time complexity = $O(2^n)$

Analysis of Binary Search Recursive

Recurrence relation –

$$T(n) = \begin{cases} 1 & \text{for } n=1 \\ T(\lfloor n/2 \rfloor) + 1 & n>1 \end{cases}$$

$T(n) = T(n/2) + 1$

$= T(n/2*2) + 1 + 1$

$= T(n/2*2*2) + 3$

$= T(n/2^4) + 4$

.....

$= T(n/2^k) + k$

Let $n/2^k = 1$, $n = 2^k$, $\log_2 n = k * \log_2 2$

i.e. $k = \log_2 n$

Now,

$T(n) = T(1) + \log_2 n = 1 + \log_2 n$

$= O(\log_2 n)$

Recurrence Relations to Remember

Recurrence	Algorithm	Big-Oh Solution
`T(n) = T(n/2) + O(1)`	Binary Search	$O(\log n)$
`T(n) = T(n-1) + O(1)`	Sequential Search	$O(n)$
`T(n) = 2 T(n/2) + O(1)`	tree traversal	$O(n)$
`T(n) = T(n-1) + O(n)`	Selection Sort (other n^2 sorts)	$O(n^2)$
`T(n) = 2 T(n/2) + O(n)`	Mergesort (average case Quicksort)	$O(n \log n)$

Find the complexity of the below function.

```
void function(int n)
{
        if(n <= 1)
                return;
        if (n > 1)
        {
                print ("*");
                function(n/2);
                function(n/2);
        }
}
```

T(n)=2T(n/2)+1
By solving above recurrence relation substitution method we get
Time complexity =O(n)

Find the complexity of the below function.

```
function(int n)
{
        if (n <= 1)
                return;
        for (i=1 ; i <= 3 ; i++ )
                function (n – 1).
}
```

T(n)=3T(n-1)+1

By solving above recurrence relation substitution method we get
Time complexity =O(3^n)

Find the complexity of the below function.

```
function(int n)
{
        if ( n < 2 )
                return;
        else
                counter = 0;
        for i = 1 to 8 do
                function (n/2);
        for I =1 to n^3 do
                counter = counter + 1;
}
```

$T(n)=8T(n/2)+n^3+1$

By solving above recurrence relation time complexity $=O(n^3 \log n)$

- **What does it mean when we say that an algorithm X is asymptotically more efficient than Y?**
  Options:
- X will always be a better choice for small inputs
- X will always be a better choice for large inputs
- Y will always be a better choice for small inputs
- X will always be a better choice for all inputs