SUNBEAM

Sunbeam Institute of Information Technology
Pune and Karad

Module – Data Structures and Algorithms

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Algorithm analysis

- it is done for efficiency measurement and also known as time/space complexity
- It is done to finding time and space requirements of the algorithm
    1. Time - time required to execute the algorithm(ns, us, ms, s)
    2. Space - space required to execute the algorithm inside memory(bytes, kb, mb, Gb)

- finding exact time and space of the algorithm is not possible because it depends on few external factors like
    - time is dependent on type of machine (CPU), number of processes running at that time
    - space is dependent on type of machine (architecture), data types

- Approximate time and space analysis of the algorithm is always done
- Mathematical approach is used to find time and space requirements of the algorithm and it is known as "Asymptotic analysis"
- Asymptotic analysis also tells about behaviour of the algorithm for different input or for change in sequence of input
- This behaviour of the algorithm is observed in three different cases
    1. Best case
    2. Average case
    3. Worst case

To denote time and space complexity, we use 'BigO'/ O( ) notation is used

Time complexity

- time is directly proportional to number of iterations of the loops used in an algorithm
- To find time complexity/requirement of the algorithm count number of iterations of the loops

1. Print 1D array on console

```
void print1DArray(arr[], n) {
    for(i=0; i<n; i++)
        sysout(arr[i]);
}
```

No. of iterations = n

time $\propto$ iterations

time $\propto$ n

$$\boxed{T(n) = O(n)}$$

2. Print 2D array on console

```
void print2DArray(arr[][], m, n) {
    for(i=0; i<m; i++) {
        for(j=0; j<n; j++) {
            sysout(arr[i][j]);
        }
    }
}
```

iterations of outer loop = m

iterations of inner loop = n

total no. of iterations = m * n

Time $\propto$ iterations

Time $\propto$ m * n

$$\boxed{T(m,n) = O(m,n)}$$

$\because$ m $\approx\approx$ n

total iterations = n * n

time $\propto$ itr

time $\propto$ n^2

$$\boxed{T(n) = O(n^2)}$$

Time complexity

3. Add two numbers

```
int addition(n1, n2) {
    res = n1 + n2;
    return res;
}
```

- irrespective of values of n1 & n2, this algorithm will be completed in constant/fixed time.
- constant time requirement, & it is denoted as

$$\boxed{T(n) = O(1)}$$

4. Print table of given number

```
void printTable(int n) {
    for( i = 1; i <= 10; i++)
        sysout( i * n);
}
```

- loop is going to iterate fix number of times
- it will constant time, means constant time requirement

$$\boxed{T(n) = O(1)}$$

5. Print binary of decimal number

2	9	
	4	1
	2	0
	1	0
		1

$(9)_{10} = (1001)_2$

```
void printBinary(n) {
    while(n > 0) {
        sysout(n % 2);
        n = n / 2;
    }
}
```

n	n>0	n%2
9	T	1
4	T	0
2	T	0
1	T	1
0	F	

$n = 9, 4, 2, 1$

$= 9/1, 9/2, 9/4, 9/8$

$= \frac{n}{2^0}, \frac{n}{2^1}, \frac{n}{2^2} \cdots\cdots \frac{n}{2^i}$ ← no. of itr

$\frac{n}{2^i} = 1$

$n = 2^i$

$\log 2^i = \log n$

$i \log 2 = \log n$

$i = \frac{\log n}{\log 2}$

time $\propto$ itr

time $\propto \frac{\log n}{\log 2}$

$\boxed{T(n) = O(\log n)}$

Time complexity

Time complexities : O(1), O(log n), O(n), O(n log n), O(n^2), O(n^3), …., O(2^n), ……

Modification : + or - : time complexity is in terms of n
Modification : * or / : time complexity is in terms of log n

```
for( i=0; i<n; i++) -> O(n)
for( i=n; i>0; i--) -> O(n)
for( i=0; i<n; i+=20) -> O(n)
for( i=1; i<=20; i++) -> O(1)
for( i=n; i>0; i/=2) -> O(log n)
for( i=1; i<n; i*=2) -> O(log n)
```

n=9 i = 9, 4, 2, 1, ~~0~~

i = 1, 2, 4, 8, ~~16~~

```
for(i=0; i<n; i++)            -> n
    for(j=0; j<n; j++)        -> n     = n^2 -> O(n^2)

for(i=0; i<n; i++); -> n
for(j=0; j<n; j++); -> n     = 2n -> O(n)

for(i=0; i<n; i++)            -> n
    for(j=n; j>0; j/=2)       -> log n   = n * log n
                                              ↓
                                          O(n log n)
```

Time complexity

```
for(i=n/2; i<=n; i++)                  -> n
   for(j=1; j+n/2<=n; j++)             -> n
      for(k=2; k<=n; k=k*2)            -> log n
```

Total itr = $n * n * \log n$

$= n^2 \log n$

```
for(i=n/2; i<=n; i++)                  -> n
   for(j=1; j<=n; j=2*j)               -> log n
      for(k=1; k<=n, k=k*2)            -> log n
```

total itr = $n * \log n * \log n$

$= n \log^2 n$

Space complexity

- Finding approximate space requirement of the algorithm to execute inside memory

Total space = Input space + Auxiliary space

Input space: space required to store input

Auxiliary space: space required to process the input

e.g. linear search

```
int linearSearch(arr[], key, n) {
    for(i=0; i<n; i++)
        if(key==arr[i])
            return i;
    return -1;
}
```

input variables = arr

processing variables = key, n, i

input space = n

Auxillary space = 3

Total space $\propto n+3$

$\because n >>>>$

$S(n) = O(n)$

Auxiliary space Complexity

processing vars = k, n, i

Auxilliary space $\propto 3.1$

$AS(n) = O(1)$

Algorithm analysis

Iterative

- loops are used

```
int fact(int num) {
    int f=1;
    for(int i=1; i<=num; i++)
        f *= i;
    return f;
}
```

Time $\propto$ no. of iterations of the loop

Time $\propto n$

$$\boxed{T(n) = O(n)}$$

$AS(n) = O(1)$

Recursive

- recursion is used

```
int rfact(int num) {
    if(num == 1)
        return 1;
    return num * rfact(num-1);
}
```

Time $\propto$ no. of recursive calls

Time $\propto n$

$$\boxed{T(n) = O(n)}$$

$AS(n) = O(n)$

Searching algorithms analysis

(1) (n/2) (n) (n)

11	10	22	9	8	20	23

Best Avg Worst

- Time is directly proportional to number of comparisons
- For searching and sorting algorithms, count number of comparisons done

1. Linear search

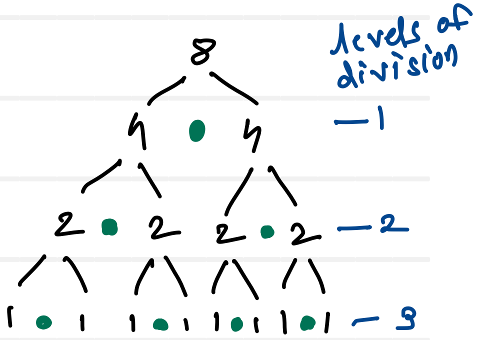
$S(n) = O(1)$

   - Best case - if key is found at few initial locations → $O(1)$
   - Average case - if key is found at middle locations → $O(n)$
   - Worst case - if key is found at last few locations / key is not found → $O(n)$

2. Binary search

$S(n) = O(1)$

   - Best case - if key is found at first few levels → $O(1)$
   - Average case - if key is found at middle levels → $O(\log n)$
   - Worst case - if key is found at last level / not found → $O(\log n)$



$2^3 = 8 \Rightarrow 2^l = n \quad l = \frac{\log n}{\log 2}$

Missing Number

Given an array nums containing n distinct numbers in the range [0, n], return the only number in the range that is missing from the array.

Example 1:

Input: nums = [3,0,1]
Output: 2

Example 2:

Input: nums = [0,1]
Output: 2

Example 3:

Input: nums = [9,6,4,2,3,5,7,0,1]
Output: 8

1. find sum of n numbers
2. find sum of array element
3. find diff of both ← missing number

```
int missingNumber(int nums[]) {
    int n = nums.length;
    int nSum = n * (n+1) / 2;
    int numsSum = 0;
    for(int i=0; i<n; i++)
        numsSum += nums[i];
    return nSum - numsSum;
}
```

Find smallest letter greater than target

You are given an array of characters letters that is sorted in non-decreasing order, and a character target. There are at least two different characters in letters.

Return the smallest character in letters that is lexicographically greater than target. If such a character does not exist, return the first character in letters.

Example 1:

Input: letters = ["c","f","j"], target = "a"

Output: "c"

Example 2:

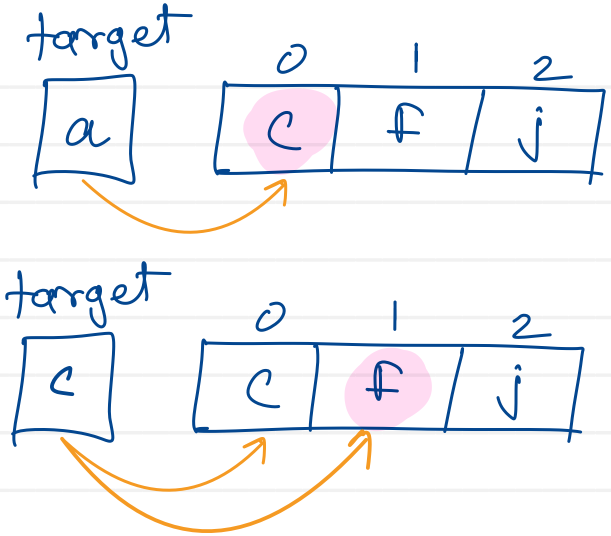
Input: letters = ["c","f","j"], target = "c"

Output: "f"

Example 3:

Input: letters = ["x","x","y","y"], target = "z"

Output: "x"




```
char nextGreatestLetter(char[] letters, char target){
    int n = letters.length;
    for( i=0; i<n ; i++) {
        if ( target < letters[i])
            return letters[i];
    }
    return letters[0];
}
```

Time Complexity

$\boxed{T(n) = O(n)}$

Find first and last position of element in sorted array

Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.
If target is not found in the array, return [-1, -1].
You must write an algorithm with O(log n) runtime complexity.

Example 1:
Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]

Example 2:
Input: nums = [5,7,7,8,8,10], target = 6
Output: [-1,-1]

Example 3:
Input: nums = [], target = 0
Output: [-1,-1]

1. find key into array
2. if key is found, find first index of key
3. if key is not found, return [-1, -1]
4. find last index of key

find first Position

```
left=0, right=nums.length-1
first=-1;
while(left<=right){
  mid=(left+right)/2;
  if(target == nums[mid]){
       first=mid;
       right=mid-1;
  }else if(target < nums[mid])
       right = mid-1;
  else
       left = mid+1;
}
```

find last Position

```
left=0, right=nums.length-1
last = -1;
while(left<=right){
  mid=(left+right)/2;
  if(target == nums[mid]){
       last =mid;
       left = mid+1;
  }else if(target < nums[mid])
       right = mid-1;
  else
       left = mid+1;
}
```

SUNBEAM

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com