DSA

Growth of functions & TC

Date
Page

Data Structures: Arrangement of data so that they can be used effecently on memory.

Algorithms:
Sequence of steps on data using effecient data structures to solve a given problem

Other Terminologies.

Database: Collection of information in permanent storage for faster retrieval and updation.

Data Warehousing:
Management of huge amount of legacy data for better analyses.

Big Data:
Analyses of too large or complex data which cannot be dealt with traditional data processing application

Data structures and algorithms are nothing new. If you have done ~~programm~~ programming in any language like C you must have used Arrays. A data structure and sequence of processing steps to solve a problem → algorithm

Data Structures:

These are like the ingredients you need to build efficient algorithms. They are the ways to arrange data so that they (data items) can be used efficiently in memory.
E.g. array, stack, Linked lists

Algorithms

Sequence of steps ~~not~~ performed on the data using efficient data structures to solve a given problem, be it a basic or real life based one.

Legacy data:

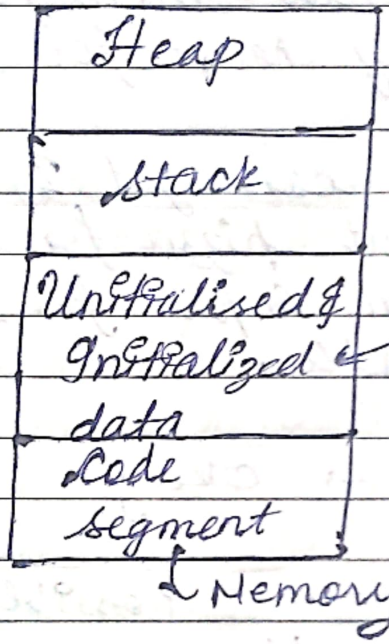
The data we keep at a different place from our fresh data in the database to make the process of retrieval and updation fast.

Memory Layout of C Program

→ When the program starts, its code gets copied to the main memory.
→ The stack holds the memory ~~occ~~ occupied by functions. It stores the activation records of the functions used in the program. And erases them as they get executed.

→ The heap contains the data which is requested by the program as dynamic memory using pointers.

→ Initialized and Uninitialized data segments hold initialized and uninitialized global variables, respectively.



TIME COMPLEXITY & BIGO notation

An analogy to a real life issue

→ I wanted to eat pizza, so I asked my brother to get some from Dominos which is 3 km away.

→ He got pizza, and I was happy only to realize it was too little for 29 friends who came to my house for a surprise visit.

→ My brother can get 2 pizzas for me on his bike, but pizza for 29 friends is too huge of an input for him, which he can't handle.

2 pizzas → Okay.

68 pizzas → Not possible in less time.

Time Complexity

Time Complexity is the study of the efficiency of algorithms. It tells us how much time is taken by an algorithm to process a given input.

For e.g.

Consider 2 algo's 1 & 2 to sort n nos. independently. When I made the prog. run for some input size n, the following results were recorded.

No. of elements	Algo 1	Algo 2	We can see, Ag Algo 1 worked well with smaller inputs, however with inc. elements Algo 2 performs much better
10	90ms	122ms	
70	110ms	124ms	
110	180ms	131ms	
1000	2s	800ms	

E.g. 2 Sending GTA5 to a friend.

→ Imagine you have a friend who lives 5km away from you. You want to send him a game of 60GB how would you send it in the shortest duration?
Note: You both have JIO 1GB/day data

→ The best way would be to send him the game by delivering it to his house. Copy the game to a HDD & make it reach him physically.

→ But for small sized games you can easily send it via internet.

→ As the ~~file~~ file size grows, the time taken to send the game online inc. linearly. O(n) whereas time taken for sending it physically remains constant. $O(n^0)$ or $O(1)$

Calculating order in terms of I/p size

In order to calculate the order (time complexity), the most impactful term containing n is taken into account (Here n refers to size of input). ~~And~~ and smaller

terms are ignored.

Algo 1: $K_1n^2 + K_2n + 36 = O(n^2)$

Highest order term — can ignore other lower order terms

Algo 2: $K_1K_2^2 + K_3K_2 + 8 = O(n^0) = O(1)$

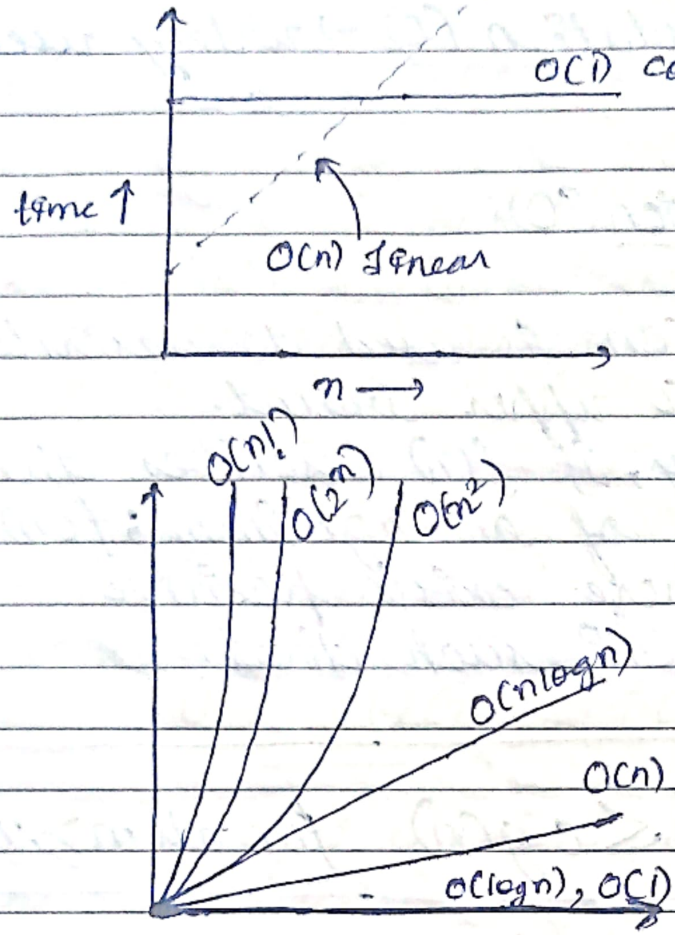
Here we ignored the smaller terms in algo 1 & carried the most impactful term which was the square of input size. Hence the time complexity became n^2. The second algorithm followed just a constant time complexity.

Big O

Big O stands for 'order of' in industry, but it's pretty different from the mathematical def^n of the big O. BigO in mathematics stands for all these complexities our program runs in. But in industry we are asked about the minimum of them. So this was a subtle difference

Visualizing Big O

If we were to plot O(1) & O(n) on a graph, they would look something like this.



Asymptotic notations

Big O, Big Omega, Big Θ

Asymptotic notations gives us an idea about how good a given algorithm is compared to some other algorithm.
Now let's look at the mathematical def" of "order of". Primarily there are 3 types of widely used asymptotic notations.

1) Big Oh notation (O)

2) Big Omega notation (Ω)

Date
Page

Big Omega notation (Ω)

→ Just like O notation provides an asymptotic upper bound, Ω notation provides an asymptotic lower bound.

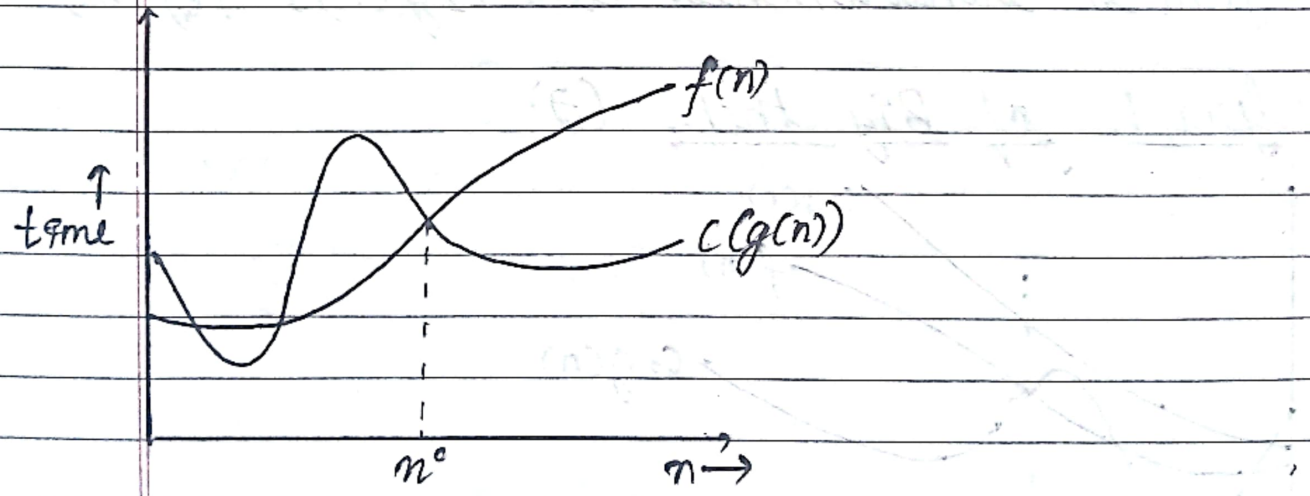
→ Let $f(n)$ define the running time of an algorithm; $f(n)$ is said to be $\Omega(g(n))$ if and only if there exists positive constants c & n^0 such that:

$$0 \leq c(g(n)) \leq f(n) \quad \forall \quad n \geq n^0$$

→ It is used to give the lower bound on a function.

→ If a func^n is $\Omega(n^2)$ it is automatically $\Omega(n)$ as well since it satisfies the above eq^n.

Graph for Big Omega:



Big theta notation (Θ)

→ Let $f(n)$ define the running time of an algorithm.

→ $f(n)$ is said to be $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ & $f(n)$ is $\Omega(g(n))$ both.

$0 \leq f(n) \leq c_1 g(n) \quad \forall n \geq n^0$

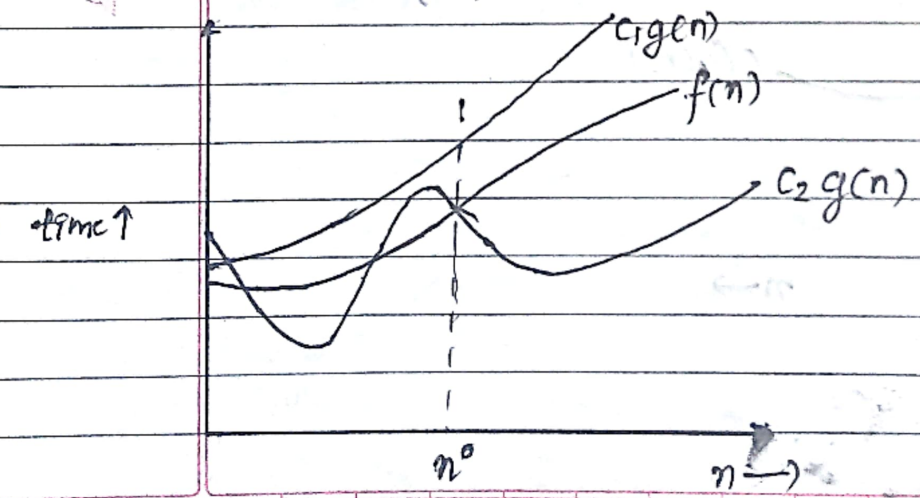
$0 \leq c_2(g(n)) \leq f(n) \quad \forall n \geq n^0$ } for sufficiently large value of n

Merging both eq^{n}s

$$0 \leq c_2 g(n) \leq f(n) \leq c_1 g(n) \quad \forall n \geq n_0^0$$

The eq^n simply means that there exist positive constants c_1 & c_2 such that $f(n)$ is sandwiched between $c_2 g(n)$ & $c_1(g(n))$

Graph of Big theta (Θ)



Which one of these to use?
Big theta provides a better picture of a given algorithm's run time, which is why most interviewers expect you to answer in terms of Big theta when they ask "order of" questions. And what you provide as the answer in Big theta, is already a Big O and Big Omega, hence it's recommended.

Hint: you can approach both these graphically making some rough graphs and mathematically finding valid constants c_1 & c_2.

Increasing order of common runtimes

$$1 < \log(n) < n < n\log n < n^2 < n^3 < 2^n < n^n$$

Best — Worse

E.g. of Big O

(1)

$f(n) = n^3 + 1$
$g(n) = n^4$

$0 \leq n^3 + 1 \leq cn^4$

Let's put $n = 2$

$\therefore 0 \leq 9 \leq c \cdot 16$

$\therefore$ It satisfies for $n = 2 \because n^0 = 2$

$\therefore c = 1$

If $\therefore$ It's ~~big O~~ $O(n^4)$

$\therefore$ It's $O(n^5)$ & $O(n^6)$, Let's check for g

$g(n) = n^3$

Which one of these to use?
Big theta provides a better picture of a given algorithm's run time, which is why most interviewers expect you to answer in terms of Big theta when they ask "order of" questions. And what you provide as the answer in Big theta, is already a Big O and Big Omega, hence it's recommended.

Hint: you can approach both these graphically, making some rough graphs and mathematically, finding valid constants c_1 & c_2.

Increasing order of common runtimes

$$1 < \log(n) < n < n\log n < n^2 < n^3 < 2^n < n^n$$

↑ Best — Worse ↓

E.g. of Big O

①

$f(n) = n^3 + 1$
$g(n) = n^4$

$0 \leq n^3 + 1 \leq cn^4$

Let's put $n = 2$

$\therefore 0 \leq 9 \leq c \cdot 16$

$\therefore$ It satisfies for $n = 2 \because n^0 = 2$

$\therefore c = 1$

If $\therefore$ It's ~~Big O~~ $O(n^4)$

$\therefore$ It's Big $O(n^5)$ & $O(n^6)$, Let's check for g

$g(n) = n^3$

$\therefore 0 \leq n^3+1 \leq cn^3$

for $n=2$

$0 \leq 9 \leq 8c$

If $c=2$

$0 \leq 9 \leq 16$

$\therefore 9$ is $O(n^3)$ also

Best Case, Worst Case & Average case analysis of an Algorithm

Life can sometimes be luck for us.

→ Exams getting cancelled when you are not prepared, etc.

→ Best case

Occasionally, we may be unlucky

→ Questions you never prepared being asked in exams, or heavy rain during sports period. etc.

→ Worst case

→ However, life remains balanced overall with a mixture of these lucky & unlucky times.

→ Expected case

These were the analogies let's study of cases of everyday life. Our fortunes fluctuate from time to time, sometimes for the better & sometimes for the worse. Similarly, a program finds it best when it is effortless for it to function. And worse otherwise.

By considering a search algorithm used to perform a sorted array search, we will analyze this thing.

Analysis of a search algorithm

Consider an array that is sorted in an increasing order

1 7 18 28 50 180

We have to search a given number & report whether it's present in the array or not. In this case, we have to two algorithms, and we will be interested in analyzing their performance ~~readily~~ seperately.

1) Algo 1: Start from the first element until an element ~~greater than or~~ equal to the element searched is found

2) Algo 2 :- Check whether the first element or last element is equal to the number. If not, find the number between these 2 elements (center of array); if center element is greater than the number to be searched, repeat the process for the first half, else repeat for the second half until number is found. And in this way, keep dividing your search space making it faster to search.

Analysing Algo 1 (Linear search)

→ We might get lucky enough to find our ~~the~~ element to be the first element of the array. Therefore, we only made one comparison which is obviously constant for any size of the array.

- Best case complexity = O(1)

→ If we are not that fortunate, the element we are searching for might be the last one. Therefore, our program made 'n' comparisons.

- Worst case complexity = O(n)

For calculating the average case time, we sum the list of all the possible case's runtime & divide it with total number of cases. Here, we found it to be just O(n).

Sometimes calculation of average case time gets very complicated.

Analyzing algo 2: (Binary Search)

→ If we get lucky, the first element will be the only element that gets compared. Hence, a constant time.

- Best case complexity = O(1)

→ If we get unlucky, we will have to keep dividing the array into halves ~~or~~ until we get a single element. (i.e. array gets finished)

Hence, time taken: $n + \frac{n}{2} + \frac{n}{4} + \cdots + 1 = \log n$

with base 2

- Worst case complexity = O(log n)

What is log n?

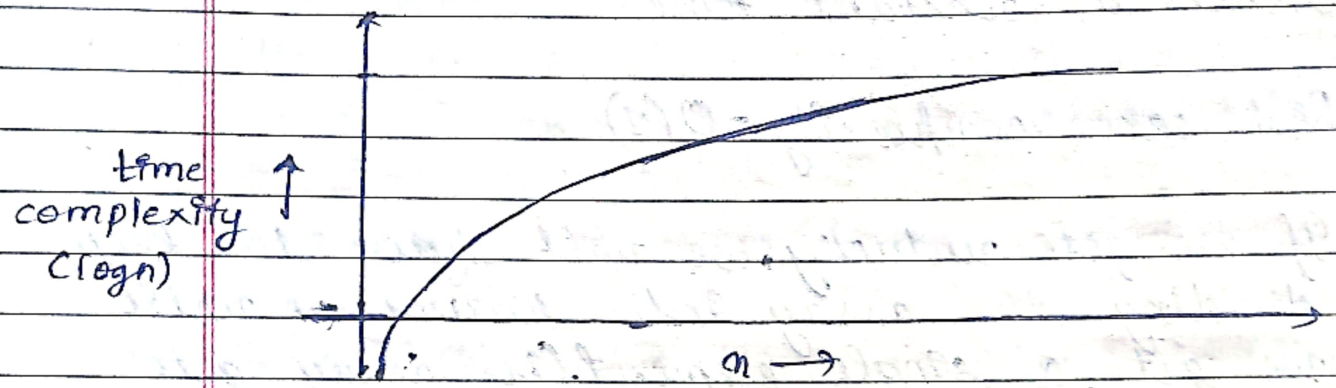
Log(n) refers to how many times I need to divide n units until they can no longer be divided (into halves)

- $\log 8 = 3 \Rightarrow \frac{8}{2} + \frac{4}{2} + \frac{2}{2}$ → Can't break further

1 1 1

- $\log 4 = 2 = \frac{4}{2} + \frac{2}{2}$ → Can't break further

You can refer to the graph below, & you will find how slowly the time complexity (Y-axis) increases when we inc. the input n (X-axis)



Space Complexity

- Time is not the only thing we worry about while analyzing algorithms. Space is equally important.
- Creating an array of size n (size of the input) → O(n) space
- If a function calls itself recursively n times, its space complexity is O(n).
- You might have wondered at some time that why can't we calculate complexity in seconds when dealing with time ~~cost~~ complexities

Here's why:

- Not everyone's computer is equally powerful. So we avoid handling absolute time taken. We just measure the growth of time with an increase in the input size.
- Asymptotic analysis is the measure of how time (runtime) grows with input.

Techniques to calculate Time Complexity

- Here are some tricks to calculate complexities

→ Drop the constants

Anything you might think is O(kn) (where k is a const.) is O(n) as well. This is considered as a better representation of the time complexity since the k term won't affect the complexity much for a higher value of n.

→ Drop the non-dominant terms

Anything you represent as $O(n^2+n)$ can be written as $O(n^2)$. Similar to when non-dominant

terms are ignored for a higher value of n.

→ Consider all variables which are provided as input.

O(mn) & O(mnq) might exist for some cases.

In most cases, we try to represent the runtime in terms of the inputs which can be even more than one number.
For e.g.
The time ~~taken~~ taken to paint a park of dimension m * n → O(kmn) → O(mn)

Practice problems

Q1 Find the time complexity of the func1 function in the program shown in the snippet below:

```
#include <stdio.h>

void func1 (int array[], int length)
{
    int sum=0;                    } f1 = K1
    int product=1;                }
    for(int i=0; i<length; i++)   }
    {                             } f2 = K2 n
        sum+= array[i];           }
    }
```

```
for (int i=0; i<length; i++)
{
    product *= array[i];
}
}
```

$f_2 = k_2 n$

```
int main()
{
int arr[] = {3, 4, 66};
func1(arr, 3);

return 0;
}
```

$T_n = f_1 + f_2 + f_3 = k_1 + k_2 n + k_3 n$

$= n(k_2 + k_3)$

$= k_4 n$

$= O(n)$

$= O(length)$

Q.2 Find the time complexity of the func function in the program from program 2.c as follows:

```
void func(int n)
{ int sum=0;
  int product=1;
for(int i=0; i<n; i++)
  {
      for (int j=0; j<n; j++)
        { printf("%d, %d \n", i, j); }
```

k_1

k_2

```
    }
  }
}
```

O/P: $K_1 + K_2 n^2$

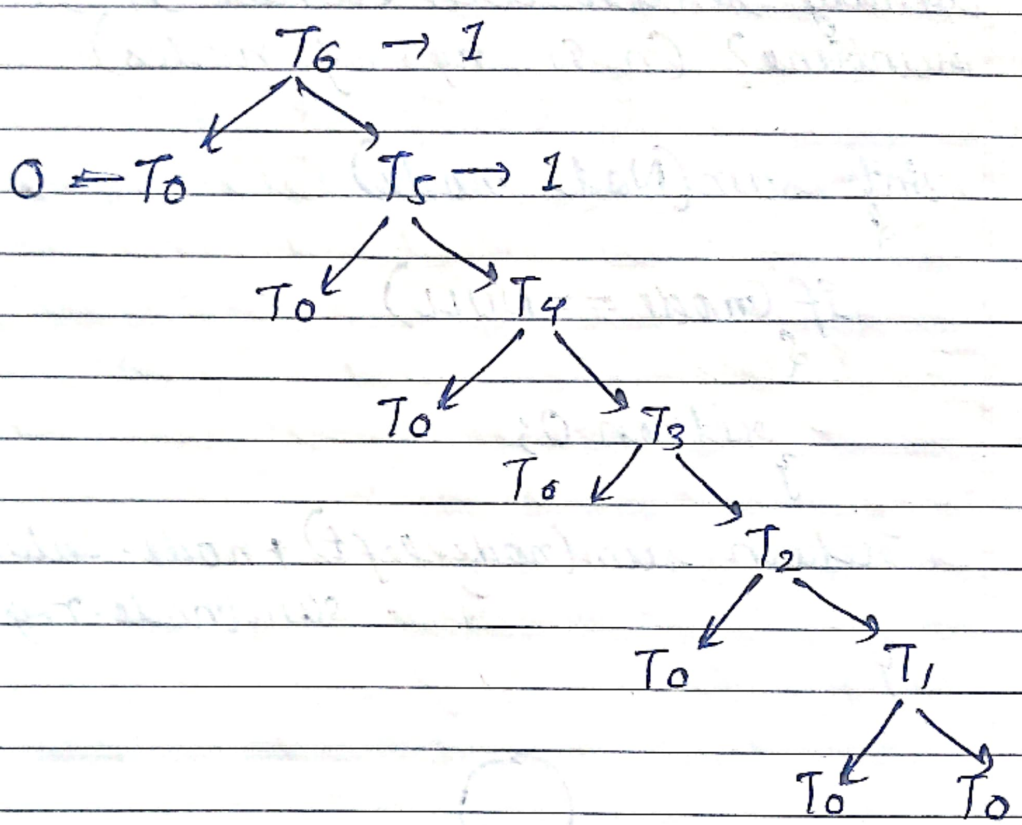
$= O(n^2)$

Q.3

```
#include<stdio.h>
#include<stdlib.h>

int random(int a)
{
  int i;
  int num = (rand() %(a+1));
}

int function(int n)
{
  int i;                          // K1=0
  if(n<=0)
  {
  return 0;
  }
  else
  { i=random(n-1); -> 1
    printf("this\n");
  return function(i)+function(n-1-i);
  }
}

int main() {
   function(6);
    return 0;
    }
```

Consider the recursive algorithm above, where the random(int n) spends one unit time to return a random integer which is evenly distributed within the range $[0, n]$. If the average processing time is $T(n)$, what is the value of $T(6)$?

Ans:



$= 6$

Q.4 Which of the foll. are equivalent to $O(n)$? Why?

a) $O(N+P)$, where $P < N/9$
$= O(n)$

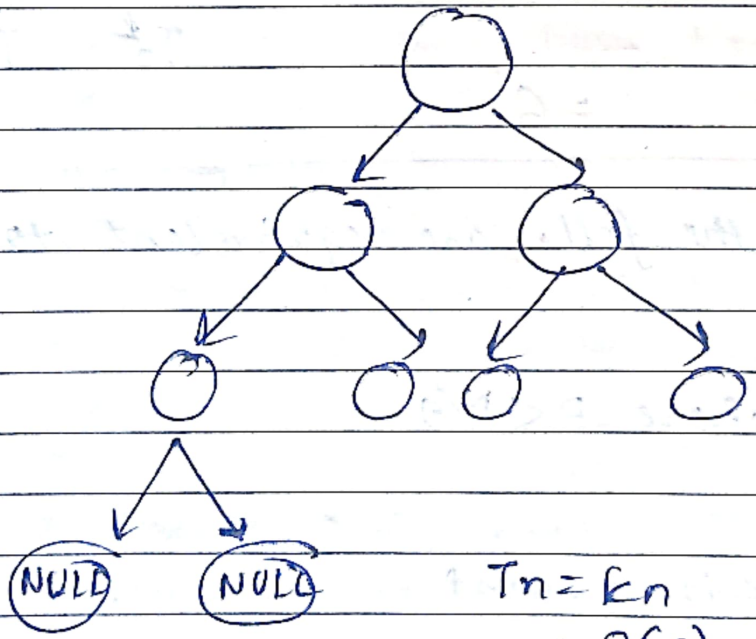
b) $O(9N-K)$ K is a const
$O(n)$

c) $O(N + \overbrace{8}^{\text{const.}} \log n)$
$O(n)$

Let) $O(N+N^2)$ (N^2 is not a const.)

5) The following simple code sums up the value of all the nodes in a balanced binary search tree. What is its runtime? (n is no. of nodes)

```
int sum(Node node)
{
  if (node == NULL)
  {
    return 0;
  }
  return sum(node.left) + node.value +
                          sum(node.right);
}
```



$T_n = kn$
$= O(n)$

6) Find the complexity of the given code which tests whether a given number is prime or not?

```
Int isPrime(Int n){
    if(n==1){
        return 0;
    }

for(Int i=2; i*i<n; i++){
    if(n%i==0)
        return 0;
}
return 1;
}
```

K1

K2

$i^2 = n-1$

$i = 2$

$i = 3$

⋮

$i = \sqrt{n}$

$= O(\sqrt{n})$

7) What is the time complexity of the foll. snippet

```
Int isPrime(Int n){
    for(Int i=2; i*i<10000; i++){
        if(n%i==0)
            return 0;
    }
return 1;
}
```

K1

OCD

Properties Of Asymptotic Notations

Reg- Oh → $f(n) \leq c \cdot g(n)$ $a \leq b$

Reg-Omega → $f(n) \geq c \cdot g(n)$ $a \geq b$

Theta (θ) : $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ $a = b$

Small (o) : $f(n) < c \cdot g(n)$ ~~$a = b$~~ $a < b$

Small (ω) : $f(n) > c \cdot g(n)$

	Reflexive	Symmetric	Transitive
O	✓	✗	✓
Ω	✓	✗	✓
θ	✓	✓	✓
o	✗	✗	✓
ω	✗	✗	✓

COMPARISON OF TIME COMP.

$O(c) < O(\log\log n) < O(\log n) < O(\sqrt{n}) < O(n) < O(n\log n)$

$< O(n^2) < O(n^3) < O(n^k) < O(2^n) < O(n^n) < O(2^{2^n})$

→ Binary Search → $\log_2 n$

→ Sequential Search → $O(n)$

→ Quick Sort → Avg. $O(n\log n)$, Worst $O(n^2)$

→ Merge Sort → $O(n\log n)$

→ Insertion Sort → Worst $O(n^2)$ Best $O(n)$

→ Bubble Sort → $O(n^2)$

→ Heap Sort → $O(n\log n)$

→ Selection Sort → $O(n^2)$

→ Height of CBT = $O(\log_2 n)$

→ Insertion in Heap = $O(\log_2 n)$

→ Construct Heap : $O(n\log n)$

→ Delete from Heap $O(\log_2 n)$

→ ~~Huf~~ Huffman : $(n\log n)$

→ Prims → $O(n^2)$, $O[(V+E)\log V]$

→ Kruskal → $O(E \log E)$

→ DFS, BFS → $O(V+E)$

→ All pair Shortest : $O(n^2)$

→ Dijkstra : $O(V^2)$

Selection sort.

```
n ← length[A]
for j = 1 to n-1
    do smallest ← j
        for i = j+1 to n
            do if A[i] < A[smallest]
                then smallest ← i
        exchange A[j] ↔ A[smallest]
```

Q1 $f_1(n) = n^2 \log_2 n$ $f_2(n) = n(\log_2 n)^{10}$

a) $f_1(n) = O\, f_2(n)$

b) $f_1(n) = \Omega\, f_2(n)$

By putting values

Let $n = 16$

$16^2 \times \log_2 16 = 16^2 \times 4$

$16^2 \times 16 (\log_2 16)^{10} = 16 (4)^{10}$

By Simplification

Ex $f_1(n) = n^2 \log_2 n$ $\quad f_2(n) = n(\log_2 n)^{10}$

$f_1(n) = n \cdot n \log n$ $\quad f_2(n) = n \log_2 n \cdot (\log_2 n)^9$

$= n \cdot$ $\quad = (\log_2 n)^9$

$= \log n$ $\quad = 9 \log_2 (\log_2 n)$

$= \log n$ $\quad = \log_2 (\log_2 n)$

$f_2(n) \leq c \cdot f_1(n)$

Q.2. $f_1(n) = 2^n$, $f_2(n) = n^{3/2}$, $f_3(n) = n \log_2 n$,

$f_4(n) = n \hat{} \log_2 (n)$

Arr. in inc. order

Look the comp.

$n = 16$

$f_1(n) = 2^{16}$ $\quad f_2(n) = 16^{3/2}$ $\quad f_3(n) = 16 \times 4$ $\quad f_4(n) =$

$f_4(n) = 16^4$

$n \log_2 n < n^{3/2} < n^{\lg n} < 2^n$