Data Structures & Algorithms

Day 01

Time & Space Complexity

Time Complexity:

→ The time complexity of algorithm quantifies the time taken by the algorithm to run as a function of the length of the input.

→ Algorithm: time depends on the function of the length of the input, not on the machine in which it is running. How many operations executed depends, not how much.

Estimation of time Complexity:

need to consider the cost of each fundamental instruction and the number of times the instruction is executed.

Algorithm

```
Algorithm ADD SCALAR (A,B)
C ← A+B
return C
```

time Complexity = constant because of one operation

$T(n) = O(1)$

Day 01

✱ Calculating time Complexity:

i) a constant time c is taken to execute one operation
ii) then the total operations for an input length on N are calculated.

✱ Consideration:

During analyses of the algorithm, mostly the worst-case scenario is considered.

Example for Calculating time Complexity

Pseudo-code

```
int a[n];
for(int i=0; i<n; i++)
    cin >> a[i]
for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
        if(i!=j && a[i]+a[j] == Z)
            return true
return false
```

Worst-case Scenario: when there is no pair of elements with sum equals Z.

i) N*C operation required for input
ii) The outer loop i loop runs N times.
iii) For each i, the inner loop j loops runs N times.

i) quadratically => its execution time grows as the square of the input size.

Day 01

time execution = $N*C+N*N*C+C$

→ ignoring lower terms
→ taking higher term

time Complexity = $O(N^2)$

* **Order of growth** is how the time of execution depends on the length of the input.

Note: O(n²)

A quadratic time complexity is considered to be less efficient than a linear time complexity (O(n)).

Another Example

```
int count = 0;
for (int i=N; i>0; i/=2)
    for (int j=0; j<i; j++)
        : count++;
```

how many time **count++** will run.

- When i=N, it will run N times.
- When i=N/2, it will run N/2 times
- When i = N/4, it will run N/4 times.

Total NO of **count++** will run is $N+N/2+N/4+\cdots+1=2*N$

Day 01

time Complexity = O(N)

* Time Complexities with the input range for which they are accepted in Competitive programming:

Input Length	Worst Accepted Time Complexity	Usually type of Solution
10-12	O(N!)	Recursion and Backtracking
15-18	$O(2^N * N)$	Recursion, backtracking and bit manipulation
18-22	$O(2^N * N)$	Recursion, backtracking and bit manipulation
30-40	$O(2^{N/2} * N)$	Meet in the middle, Divide and Conquer
100	$O(N^4)$	Dynamic programming and Constructive
400	$O(N^3)$	Dynamic programming and Constructive
2K	$O(N^2 * \log N)$	Dynamic programming, Binary search, Sorting, Divide and Conquer
10K	$O(N^2)$	Dynamic programming, graph, trees, Constructive
1M	$O(N * \log N)$	Sorting, Binary Search, Divide and Conquer
100M	O(N), O(log N), O(1)	Constructives, Mathematical, Greedy Algorithm

Space Complexity

The amount of memory required by the algorithm to solve given problem is called Space Complexity.

estimation

depends on two paths

I) A fixed part:

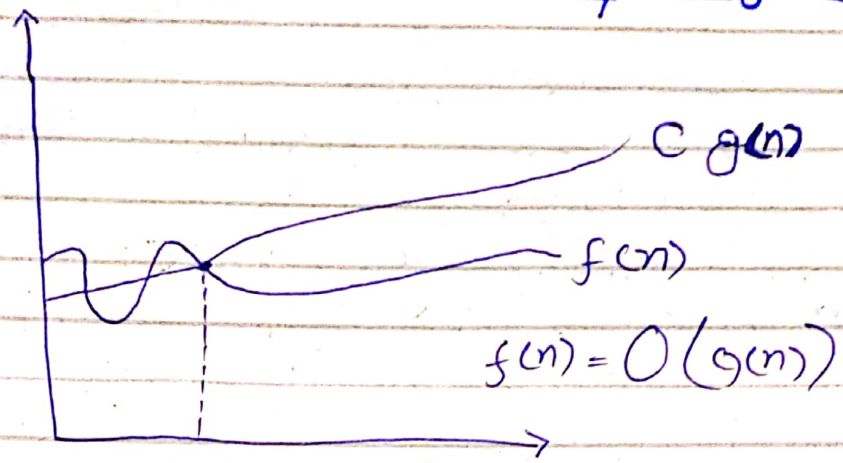
It is independent of the input size. It includes memory for instructions, constants, variable etc.

II) A variable part:

It is dependent on the input size. It includes memory for recursion stack, referenced variable, etc.

Day 02

Space and Time Complexity: Big-O_h Notation



$O(g(n)) = \{ f(n)$: There exist positive constants C and n_0 Such that $0 \leq f(n) \leq C\,g(n)$ for all $n \geq n_0 \}$

O-notation gives an upper bound for a function to within a constant factor. we write $f(n) = O(g(n))$ if there are positive constants n_0 and C such that to the right of n_0, the value of $f(n)$ always lies on or below $C\,g(n)$.

Example

Prove $5n^2 + 3n + 1 = O(n^2)$

Let $f(n) = 5n^2 + 3n + 1$

$= n^2 (5 + 3/n + 1/n^2) < n^2 (5+3+1)$

$= 8n^2$

When $n > 1$

then $5n^2 + 3n + 1 < 8n^2$

So from defination

$0 \leq f(n) \leq C\,g(n)$ for $n \geq 1$

Here $C = 8$, $n_0 = 1$ and $g(n) = n^2$

Hence, $5n^2 + 3n + 1 = O(n^2)$ proved

factorial $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

Day 03

Recursion

Recursion	Iteration
i) Terminates when base case becomes true.	Terminates when condition becomes false.
ii) Used with functions	Used with loops
iii) Every recursive call needs extra space in the stack memory	Every iteration does not require any extra space
iv) Smaller code size	Larger code size

Advantages

i) provides neat and simple way to write code.

DisAdvantages

i) Smaller code is difficult to understand
ii) Takes more time to execute due to function calling and returns overhead.
iii) Has greater space requirements than the iterative program as all functions will remain in the stack untill base case is reached.

Day 03

Summary of Recursion

i) Two types of cases in recursion i.e. recursive case, base case.

ii) The base case is ~~turns out~~ used to terminate the recursive function when the case turns out to be true.

iii) Each recursive call makes new copy of method in the stack memory.

iv) Infinite recursion may lead to running out of the stack.

v) Examples of Recursive algorithms: Merge Sort, Quick Sort, Tower of Hanoi, Fibbonacci Series, Factorial problem etc.

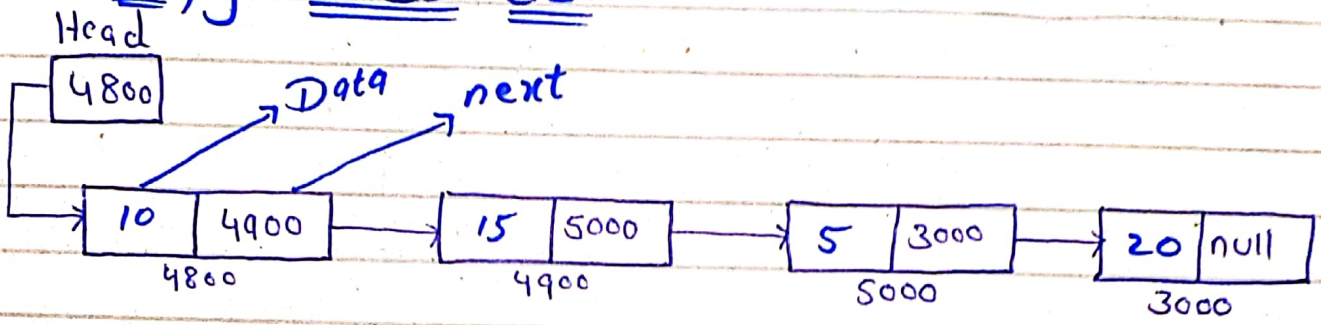
Day 04

Linked-List-Basics

3 types:

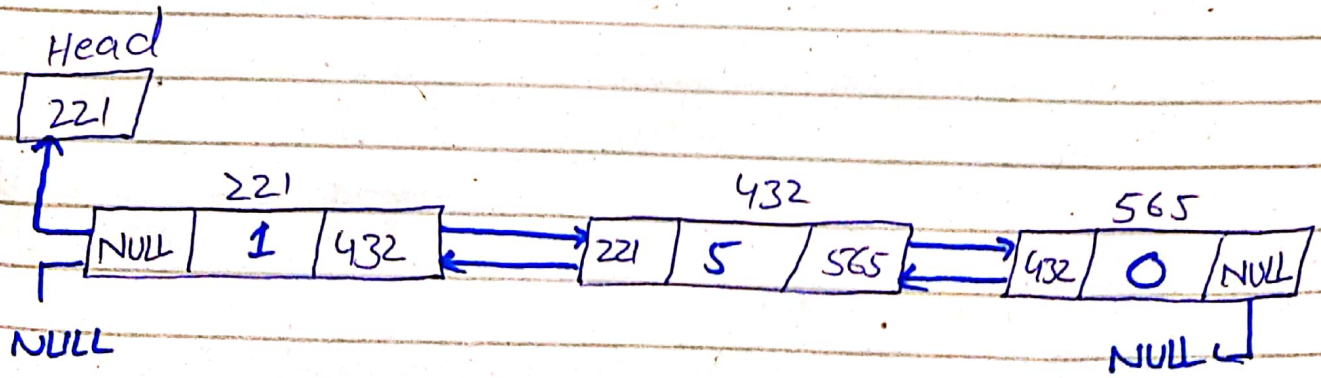
i) Singular Linked_List
ii) Doubly Linked_List
iii) Circularly Linked-List

Singly-Linked-List



Node

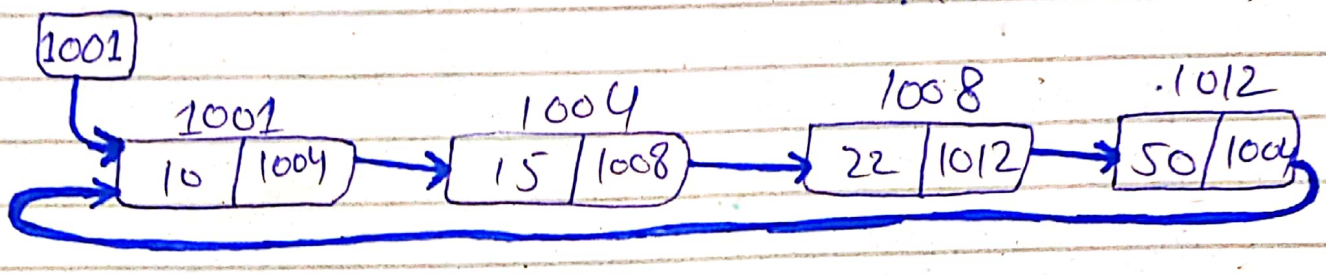
Doubly_Linked_List



Dynamic Memory Allocation: Dynamic allocation is a concept in programming where memory is allocated at runtime, rather than at compile time. This means that memory is allocated during the execution of the program, and can be resized as needed during the program's execution.

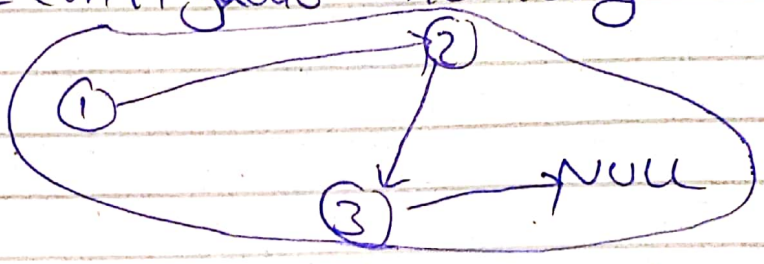
Circular Linked list

{malloc, calloc, realloc}
usage {linkedlist, trees, graphs}



Basic Properties

i) Variable Size
ii) Non-Contiguous memory



iii) Dynamic Memory Allocation
iv) Insertion and Deletion
v) Random Access
time Complexity = O(n)

vi) Efficiency

a) Efficient for insertion and deletion.
b) un Efficient for random access operations.

Corner Cases: That are different from normal cases.

Day #04

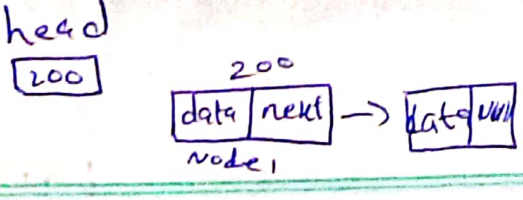
Array List VS Linked List

Insert: O(n) > O(1)

Search: O(1) < O(n)

Java linkedList Implementation

```
import java.util.*;
class BuiltInLL {
    public static void main(String args[]){
        LinkedList<String> list = new LinkedList<String>();

        list.addFirst("is");
        list.addFirst("Ahmad");
        System.out.println(list);
        list.addLast("a");
        list.addLast("Software");
        list.addLast("Engineer");
        System.out.println(list);
        System.out.println("Size:" +list.size());

        for(int i=0; i<list.size(); i++){
            if(i == list.size()-1){
                System.out.println(list.get(i));
            }
            System.out.println(list.get(i)+" -> ");
        }
        list.removeFirst();
        list.removeLast();
        System.out.println(list);
        list.remove(2);
    }
}
```



Day #05

Linked List . Scratch Implimentation

```
Class LL {

    Node head;
    private int size;
    LL(){
        this.size = 0;
    }

    Class Node {
        String data;
        Node next;

        Node(String data){

            this.data=data;
            this.next = null;
            size++;
        }
    }

    // add_last
    public void addlast(String data){

        Node newNode = new Node(data);
        if(head == null){ // corner case
            head = newNode;
            return;
        }

        Node currNode = head;
```

That are different from normal cases.

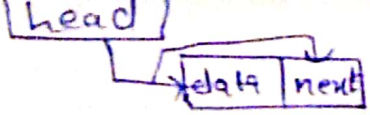
↙ It means that it is of the node

```
while(currNode.next != null){
        currNode = currNode.next;
}

currNode.next = newNode;
}
```

```
// add-first
public void addFirst(String data){
    Node newNode = new Node(data);
    if(head == null){
        head = newNode;
        return;
    }
    newNode.next = head
        head = newNode;
    ↳ This line will store address of newNode in
    head
}
```

```
// delete-first

Public void deleteFirst(){
    if(head == null){
        System.out.println("The list is empty!");
        return;
    }
    Size--;
    head = head.next;
}
```

```
// delete-last
Public void deletelast(){
    if(head == null){
        System.out.println("The list is empty");
        return;
}
```



Day #05

```
    Size--;
if(head.next == null){
        head.next
        head = null;
        return;
}
Node secondLast = head;
Node lastNode = head.next;
 while(lastNode.next != null){
        lastNode = lastNode.next;
        secondlast = secondlast.next;
}
    Secondlast.next = null;

}

        // Size

public int getSize(){
        return size;
}

Public static void main(String
                        args[]){

LL list = new LL();
list.addFirst("a");
list.addFirst("is");
list.addFirst("Ahmad");

list.printList();
```

Day #05

```
list.addlast("software Engineer");

list.printlist();

list.deleteFirst();
list.printlist();

list.deletelast();
list.printlist();

list.addlast("jajja");
list.printlist();

System.out.println("size:" + list.getsize());
}
}
```