Data Structures – Foundation

Motivation: How can you be a better coder, if you don't know how good your code is? 😊

Memory Areas: The below diagram describes the different memory areas in **Java**:

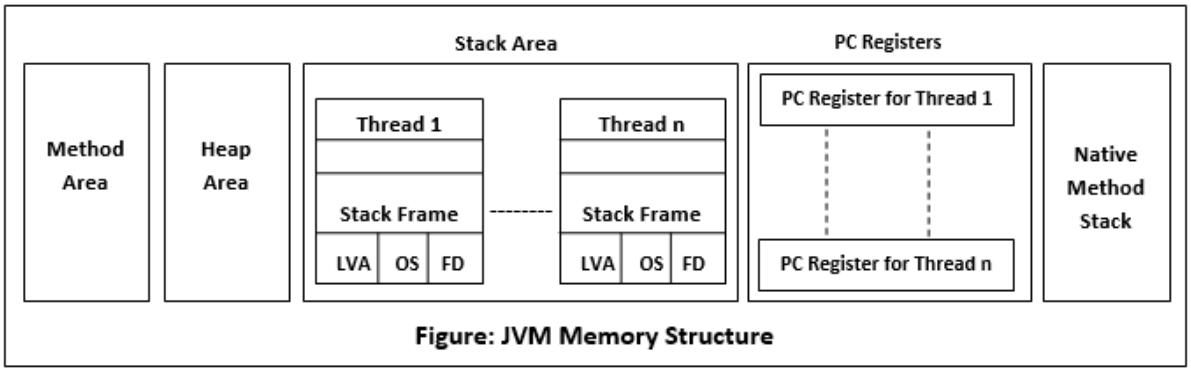


Figure: JVM Memory Structure

Important points to remember:

Method Area: Global variables are stored in Method Area.

Heap Area: Objects are stored in Heap.

Stack Area: Local variables are stored in Stack.

Native method stack: Stack stores the data of the methods in non-java language. (also called as C-Stack)

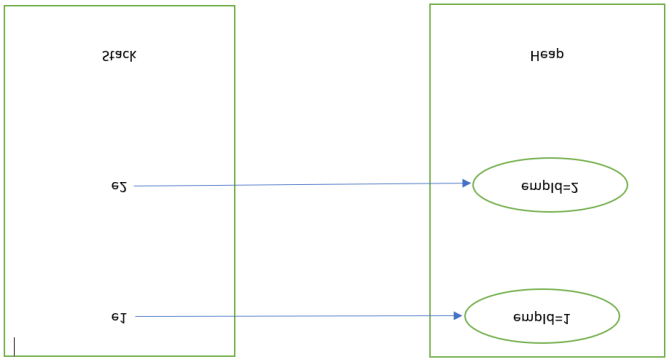
PC Register:

- For non – native methods: JVM thread has a program counter (PC) associated with it. PC stores the available JVM instructions.
- For Native methods: PC value is undefined. PC Register stores the return address of the native pointer.

Simple code:

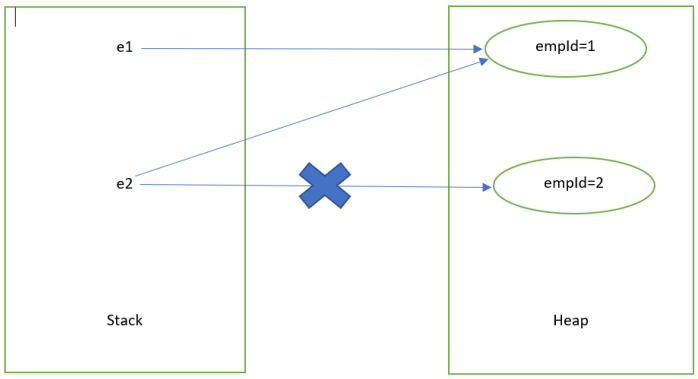
```
public class Employee {
	int empId;
}

public class Test {
	static String Company;
	public static void main(String[] args) {
		Employee e1 = new Employee();
		Employee e2 = new Employee();
		}
}
```

Karthik Chinni



Tweaking the above code to see the change in the memory organization:

```
public class Test {
     static String Company;

     public static void main(String[] args) {
          Employee e1 = new Employee();
          Employee e2 = new Employee();
          e1.empId = 10;
          e2 = e1;
     }
}
```

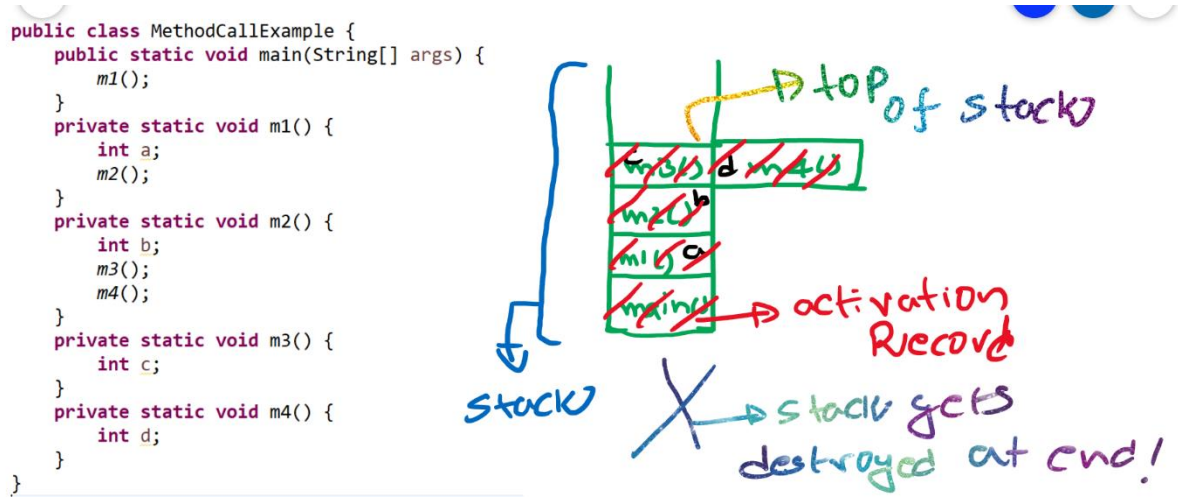


Important points to remember:

1. For every method call, there will be an activation records that gets created.
2. Activation record gets destroyed, once the method call gets completed
3. The stack gets destroyed when the associated thread gets get destroyed.

Sample code:

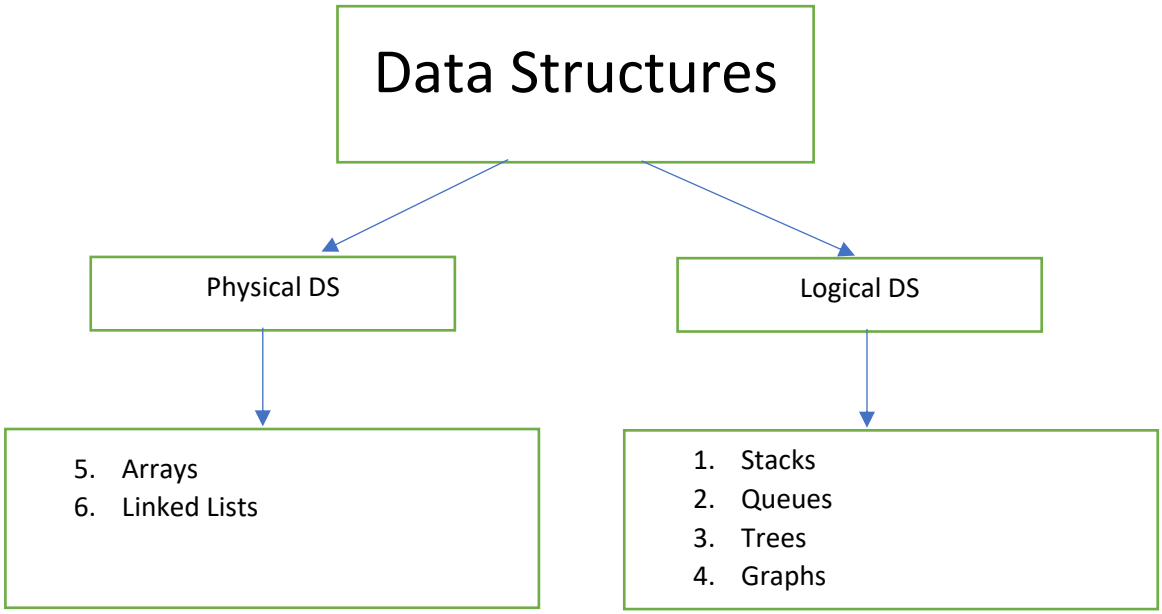
```
public class MethodCallExample {
	public static void main(String[] args) {
		m1();
	}
	private static void m1() {
		int a;
		m2();
	}
	private static void m2() {
		int b;
		m3();
		m4();
	}
	private static void m3() {
		int c;
	}
	private static void m4() {
		int d;
	}
}
```



Karthik Chinni

What are the different types of Data Structures to learn?

1. LinkedList
2. Stacks
3. Queues
4. Trees
5. Graphs



What is Data Structure?

It is nothing but the way the data is organized inside the memory. For example, in a linked list, the data is organized in form of nodes that are interconnected with each other.

Real time usages of various Data Structures: -

- **LinkedList**: - Music player.
- **Stack**: - Recently opened web pages inside a browser.
- **Queue**: - Job scheduling in Operating systems.
- **Trees**: - Folder structures in Operating Systems.
- **Graphs**: - Google Maps.
- **Trie**:- Red lines in a editor.

What is an Algorithm?

An Algorithm is nothing but a step by step process of achieving something.

Example: -

How to make a phone call?

1- Get your phone
2- Unlock it
3- Search for a contact
    a. If the contact exists?
        i. Do you have balance?
            1. if yes, make a call.
            2. If no, terminate?
    b. If contact does not exit
        i. Do you have the number to call?
            1. If no, terminate.
            2. If yes, make a call.
4- So on….

Example: - add two numbers

- Take two numbers in two variables, say a,b.
- Take a new variable sum.
- Add a,b and assign the value to sum.
- Return sum.

```
private int sum(int a, int b) {
        int sum = a+b;
        return sum;
}
```

There are different ways to achieve something. Like, if you want to travel from one place to other, you can do that using different ways, like you take a flight, train, bus, car or even you can walk. Every approach has its own pros and cons. Like taking a flight is faster but not cost efficient, and this may not be applicable every time. You cannot take a flight to travel to next street.

How to compare Algorithms?

Algorithm comparison should be independent of programming language and external factors like hardware. Ideal way to compare two algorithms is to compare their growth rate.

Say, you need to send a file to your friend. How can you do that?

1- By sending an email or via FTP.
2- By taking that file in a hard drive and by travelling by yourself to you friend and handing it over.

In case 2, the time would be constant and it is irrespective of the file size. Like even if it is 1TB file, the travel time taken is the same.

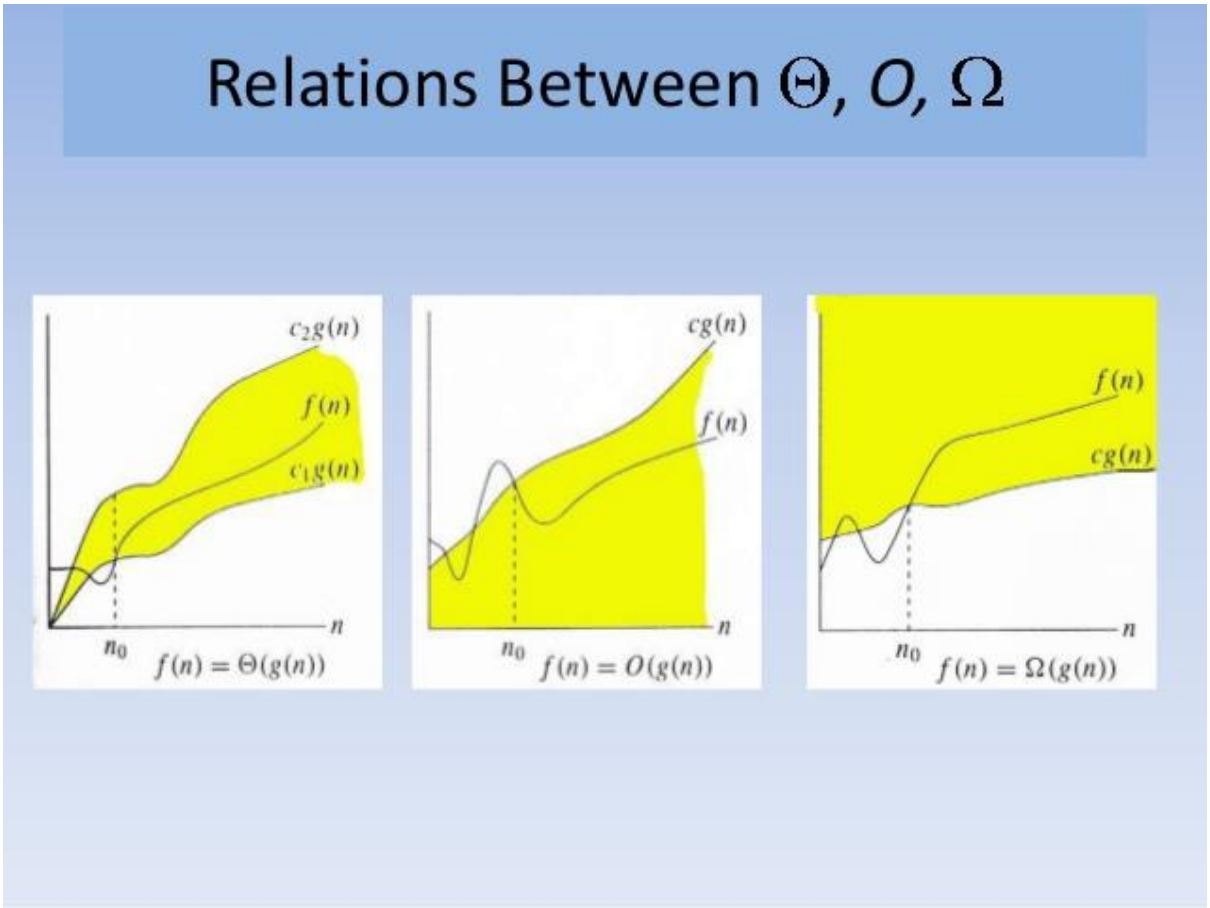
But, in case 1, if it's a small text file, it is sent over internet in no time. But, if the file is larger, say 1 TB, it takes days to transfer that file electronically. So the growth rate depends on the input size of your file.

Case 2 has constant Time Complexity. We denote it using O(1).

Types of Analysis: -

1. **Best Case** − Minimum time required for program execution. **Ω (Big-Omega)**
2. **Average Case** − Average time required for program execution. **Θ (Theta)**
3. **Worst Case** − Maximum time required for program execution. **O (Big-O)**

Diagrammatic notation: -



An algorithm can be represented in form of an expression.

Example: -

f(n)=3n+2
3n+2<=4n for all n>=3

Hence 3n+2 = O(n)

Say, you went to buy a pair of Shoes and Socks. Your friend came and asked you what you are buying. You simply say that you came to buy pair of shoes. You simply ignore socks, even if you had already

bought them. The reason why you ignore is that the value of socks is negligible in comparison to that of shoes.

Similarly,
in **f(n)=3n+2**, we ignore 3 and 2 as they are constants the time complexity becomes **O(n)**
If, **f(n)=3n^2+2n+3**, we ignore all and the time complexity becomes **O(n^2)**

We only concentrate on Big-O notations as we analyse worst cases. There is no point of analysing best case because, most algorithms will be O(1) in the best case.

Problem	Number of Steps	Time Complexity	Space Complexity

`for (int i = 0; i < 5; i++) {`
`  //Some action`
`}`
	5	O(1)	O(1)

`for (int i = 0; i < n; i++) {`
`  //Some action`
`}`
	n	O(n)	O(1)

`for (int i = 0; i < n; i++) {`
`  for (int j = 0; j < n; j++) {`
`  //Some action`
`  }`
`}`
	n^2	O(n^2)	O(1)

`for (int i = 0; i < n; i++) {`
`  //Some action`
`}`
`for (int j = 0; j < n; j++) {`
`  //Some action`
`}`
	n+n	O(n)	O(1)

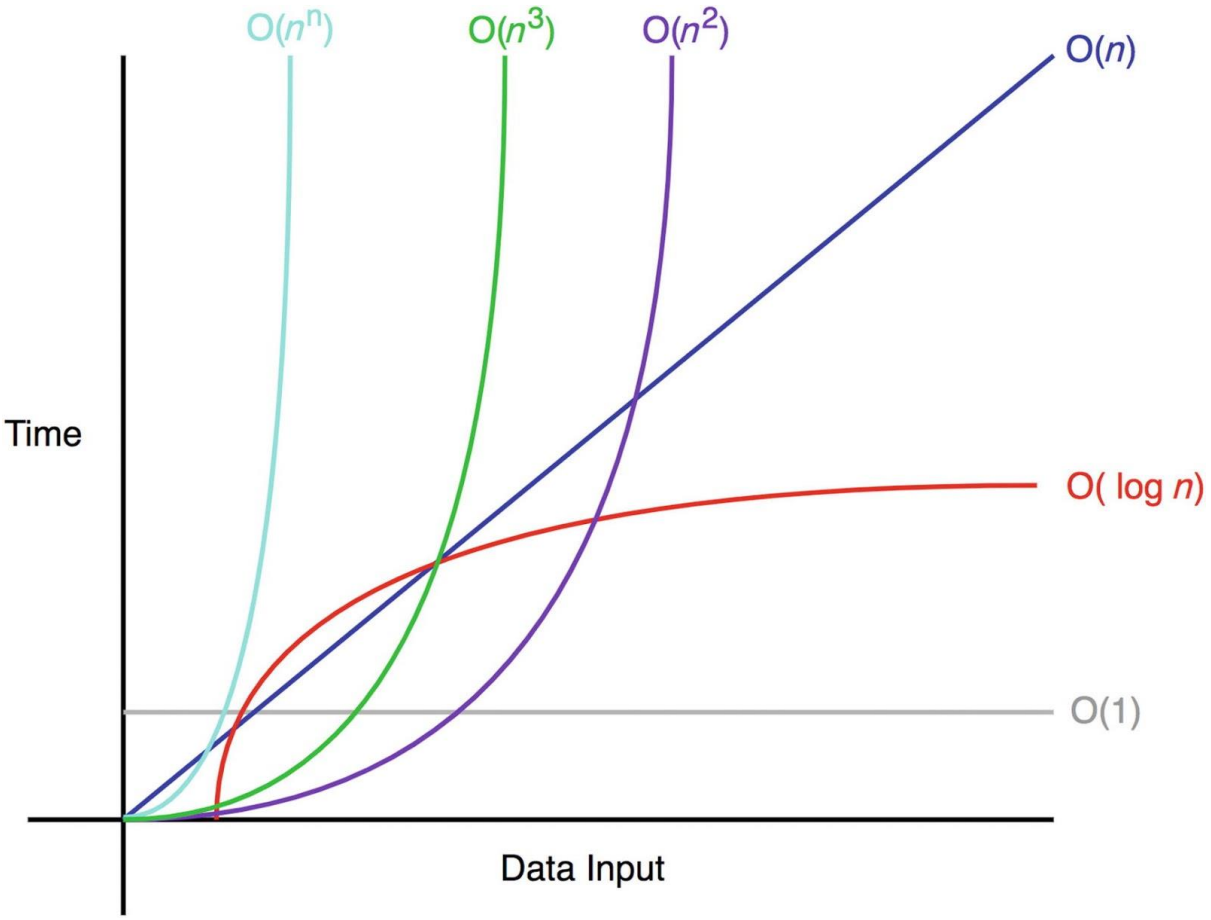
`for (int i = 0; i < n; i=i+2) {`
`  //Some action`
`}`
	n/2	O(n)	O(1)

`for (int i = 1; i <= n; i=i*2) {`
`  //Some action`
`}`
	log n	O(log n)	O(1)

`for (int i = n; i > 0; i=i/2) {`
`  //Some action`
`}`
	log n	O(log n)	O(1)

`int[] input;`
`int sum;`
`private int sum(input) {`
`  for (int i = 0; i < input.length; i++) {`
`  sum+=input[i];`
`  }`
`  return sum;`
`}`
	n	O(n)	O(n)

Graphical representation of various runtime complexities:



Complexities are also known as :-

c

complexity	O(1)	O(log n)	O(n)	O(n^a)	O(a^n)
called as	constant	logarithmic	lenier	polinomial	exponential

Few log formula that will become handy :

- **$\log_a 1 = 0$**
- **$\log_a a = 1$**
- **$\log_a(x*y) = \log_a x + \log_a y$**
- **$\log_a(x/y) = \log_a x - \log_a y$**
- **$\log_a x^p = p \log_a x$**

Log in Mathematics is generally to base 10. But, in Computer Science, log is to base 2, by default.

Karthik Chinni

We can remember the log in the below simple manner. Since the base is 2, we get the log value of a particular number by calculating the number of steps required bring the number to 1, by dividing it by 2.

Let’s take an example:

Say input is 8. We can divide 8 with 2, for 3 times to make its value to be 1.

1. 8/2=4
2. 4/2=2
3. 2/2=1

Now, say input is 16, we need 4 steps.

1. 16/2=8
2. 8/2=4
3. 4/2=2
4. 2/2=1

Log values for first 25 numbers:

$\log_2(x)$	Notation	Value
$\log_2(1)$	lb(1)	0
$\log_2(2)$	lb(2)	1
$\log_2(3)$	lb(3)	1.584963
$\log_2(4)$	lb(4)	2
$\log_2(5)$	lb(5)	2.321928
$\log_2(6)$	lb(6)	2.584963
$\log_2(7)$	lb(7)	2.807355
$\log_2(8)$	lb(8)	3
$\log_2(9)$	lb(9)	3.169925
$\log_2(10)$	lb(10)	3.321928
$\log_2(11)$	lb(11)	3.459432
$\log_2(12)$	lb(12)	3.584963
$\log_2(13)$	lb(13)	3.70044
$\log_2(14)$	lb(14)	3.807355
$\log_2(15)$	lb(15)	3.906891
$\log_2(16)$	lb(16)	4
$\log_2(17)$	lb(17)	4.087463
$\log_2(18)$	lb(18)	4.169925
$\log_2(19)$	lb(19)	4.247928
$\log_2(20)$	lb(20)	4.321928
$\log_2(21)$	lb(21)	4.392317
$\log_2(22)$	lb(22)	4.459432
$\log_2(23)$	lb(23)	4.523562
$\log_2(24)$	lb(24)	4.584963
$\log_2(25)$	lb(25)	4.643856

Linear Search vs Binary Search:

Linear search: Searching for an element, one element at a time without skipping any item.

Binary Search: Cut down your search to half as soon as you find middle of a sorted list.

```
public class LinearSearch {
	public static void main(String[] args) {
		int[] arr = { 10, 20, 30, 40, 50 };
		boolean result = linearSearch(arr, 166);
		System.out.println(result);
	}
	static boolean linearSearch(int[] arr, int x) {
		for (int i = 0; i < arr.length; i++) {
			if (arr[i] == x) {
				return true;
			}
		}
		return false;
	}
}
```

Karthik Chinni

```java
public class BinarySearch {
	public static void main(String[] args) {
		int[] arr = { 10, 20, 30, 40, 50 };
		boolean result = binarySearch(arr, 50, 0, arr.length - 1);
		System.out.println(result);
	}
	private static boolean binarySearch(int[] arr, int x, int low, int high) {
		while (low <= high) {
			int mid = (low + high) / 2;
			if (arr[mid] == x) {
				return true;
			} else if (arr[mid] < x) {
				low = mid + 1;
			} else if (arr[mid] > x) {
				high = mid - 1;
			}
		}
		return false;
	}
}
```

Golden rules to remember before starting to write the code :

1. By looking at the method signature, the interviewer should come to a conclusion that you know what you are trying to write.
    a. Method names should be meaningful.
    b. Return type should be proper.
    c. Correct set of parameters should be taken.
2. Every method you write should have the boundary conditions correctly defined.
    a. Check for null conditions.
    b. Check for empty conditions.
3. Write clean code with proper indentation if you are writing on a piece of paper.

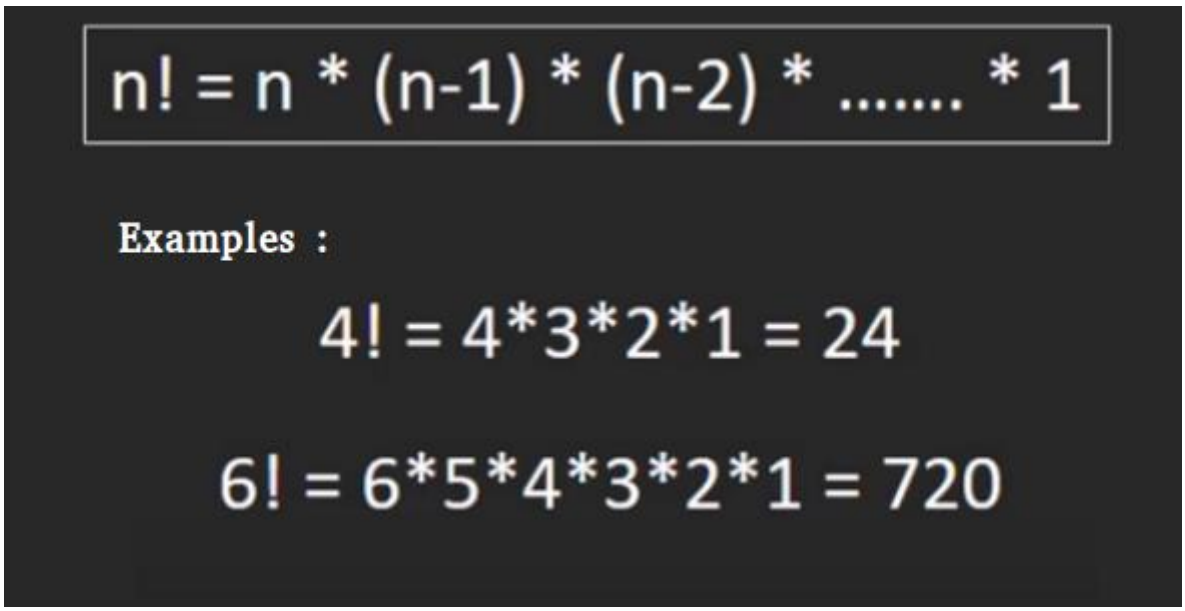
Example code following above rules:

```
public class Employee {
	int empId;
	String empName;

	public Employee(int empId, String empName) {
		super();
		this.empId = empId;
		this.empName = empName;
	}

	public int getEmpId() {
		return empId;
	}

	public void setEmpId(int empId) {
		this.empId = empId;
	}

	public String getEmpName() {
		return empName;
	}

	public void setEmpName(String empName) {
		this.empName = empName;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + empId;
		result = prime * result + ((empName == null) ? 0 : empName.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Employee other = (Employee) obj;
		if (empId != other.empId)
			return false;
		if (empName == null) {
			if (other.empName != null)
				return false;
		} else if (!empName.equals(other.empName))
			return false;
		return true;
	}

	@Override
	public String toString() {
		return "Employee [empId=" + empId + ", empName=" + empName + "]";
	}

}
```

Karthik Chinni

```java
public class EmployeeTest {
	public static Employee changeName(Employee emp, String name) {
		// Base condition 1
		if (emp == null) {
			return null;
		}
		// Base condition 2
		if (name == null) {
			return emp;
		}
		// Base condition 3
		if (name.isEmpty()) {
			emp.setEmpName("Empty");
			return emp;
		}
		// Actual logic.
		emp.setEmpName(name);
		return emp;
	}

	public static void main(String[] args) {
		Employee e1 = new Employee(1, "");
		Employee changedEmployee = changeName(e1, "");
		System.out.println(changedEmployee);
	}

}
```

Factorial Implementation: -



```java
public class FactorialIterative {
	public static void main(String[] args) {
		System.out.println(fact(6));
	}

	static int fact(int n) {
		if (n == 0 || n == 1) {
			return 1;
		}
		int result = 1;
		for (int i = n; i >= 2; i--) {
			result *= i;
		}
		return result;
	}
}
```

Time Complexity: O(n) : Space Complexity : O(1)

Fibonacci Series implementation: -

Fibonacci series

Sum of: 0 + 1 = 1
Sum of: 1 + 1 = 2
Sum of: 1 + 2 = 3
Sum of: 2 + 3 = 5
Sum of: 3 + 5 = 8
Sum of: 5 + 8 = 13
Sum of: 8 + 13 = 21
Sum of: 13 + 21 = 34
Sum of: 21 + 34 = 55
Sum of: 34 + 55 = 89

```java
public class FibonacciSeriesIterative {

	public static int fib(int n) {
		int a = 0;
		int b = 1;
		int c = 1;
		System.out.print(a + "," + b);
		for (int i = 1; i <= n; i++) {// Iteration starts form 1 and not 0.
			a = b;
			b = c;
			c = a + b;
			System.out.print("," + c);
		}
		return c;
	}

	public static void main(String[] args) {
		fib(7);
	}
}
```

Time Complexity: O(n) : Space Complexity : O(1)

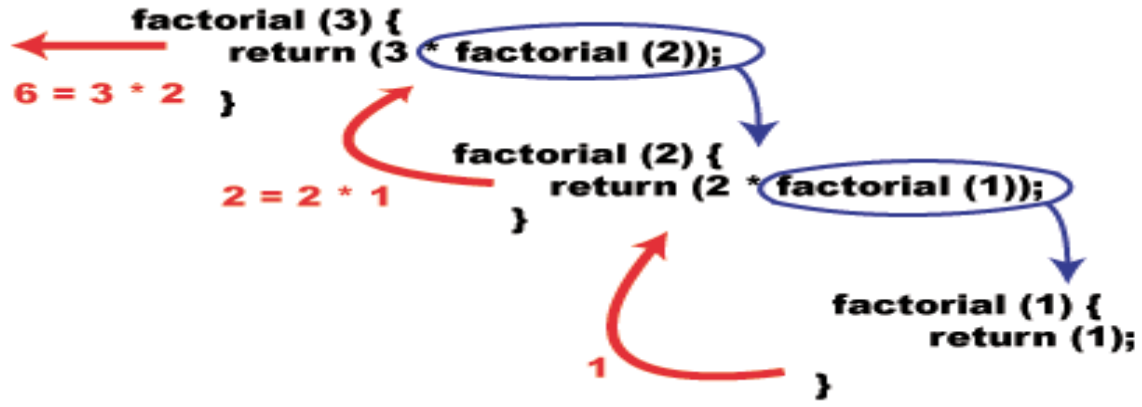
Karthik Chinni

Formulae: -

Formula	Time Complexity : no. of steps taken in terms of input
Iterative	no. of steps in terms of n
Recursive	(no. of recursive calls)*(TC of each call)[Ignore recursive call]

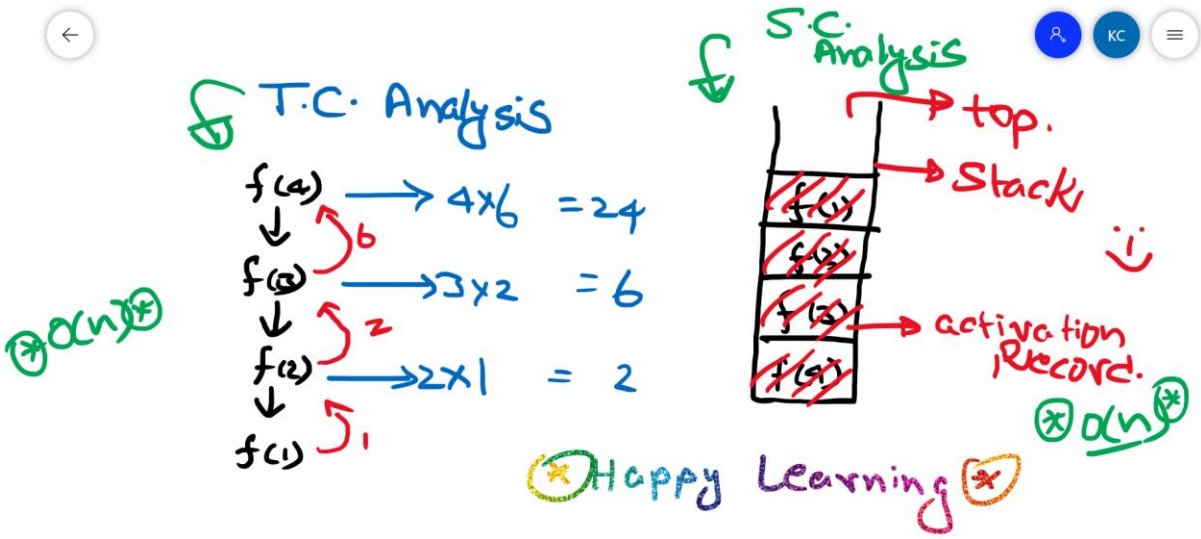
Formula	Space Complexity : Extra space taken in terms of input
Iterative	Extra space in terms of n
Recursive	(max length of stack)*(SC of each call) [Ignore recursive call]

Factorial Using Recursion: -

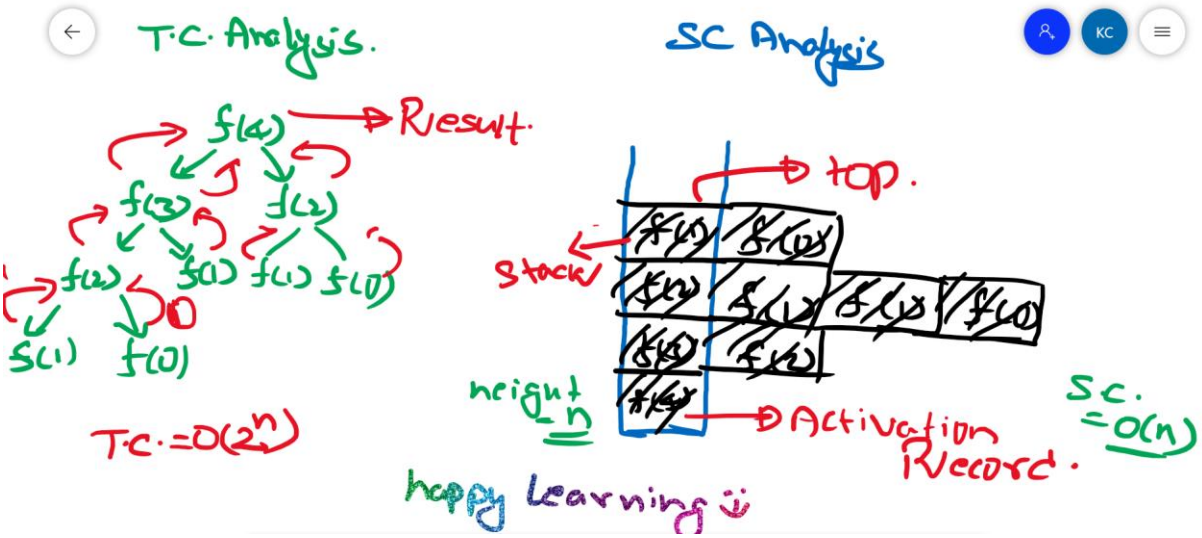



```
public class FactorialRecursive {
	public static void main(String[] args) {
		System.out.println(fact(5));
	}
	static int fact(int n) {
		if (n == 0 || n == 1) {
			return 1;
		}
		return n * fact(n - 1);
	}

}
```

Karthik Chinni




```java
public class FibonacciSeriesRecursive {
	public static int fibonacci(int n) {
		if (n <= 0) {
			return 0;
		}
		if (n == 1) {
			return 1;
		}
		return fibonacci(n - 1) + fibonacci(n - 2);
	}
	public static void main(String[] args) {
		System.out.println(fibonacci(4));
	}

}
```



Karthik Chinni

Comparison: -

Program for Factorial:-

Recursive

```
int fact(int n) {
   if (n == 0 || n == 1) {
      return 1;
   }

   return n * fact(n - 1);
}
```

Iterative

```
int fact(int n) {
   if (n == 0 || n == 1) {
      return 1;
   }

   int result = 1;
   for (int i = 2; i < n; i++) {
      Result *= i;
   }
   return result;
}
```

Program for Fibonacci series: -

Recursive

```
int fib(int n) {
   if (n == 0 || n == 1) {
      return 1;
   }

   return fib(n - 1) + fib(n - 2);
}
```

Iterative

```
int fib(int n) {
   int a = 0, b = 1, c;
   if (n == 0 || n == 1) {
      return n;
   }

   for (int i = 2; i <= n; i++) {
      c = a + b;
      a = b;
      b = c;
   }
   return b;
}
```

	Factorial	
	Time Complexity	Space Complexity
Iterative	O(n)	O(1)
Recursive	O(n)	O(n)

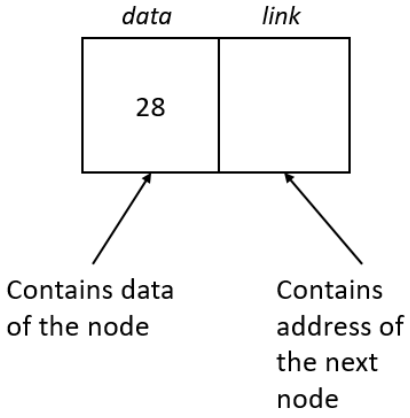
	Fibonacci	
	Time Complexity	Space Complexity
Iterative	O(n)	O(1)
Recursive	O(2^n)	O(n)

Karthik Chinni

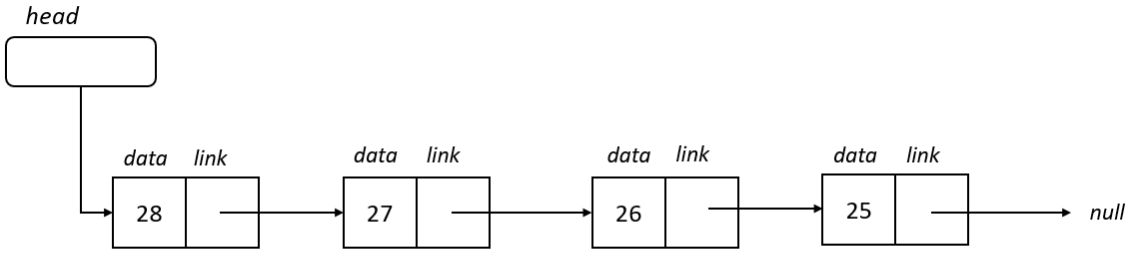
Linked List

1. It's a Physical Data Structure.
2. It is represented in form of Node.
    a. Every Node has two parts
        i. Data
        ii. Pointer to next Node
    b. Node is represented as an Object.
3. Address of the first node is stored in the head. So, if the pointer to head is lost, we lose access to the LinkedList. So, we should be very careful while coding.

Below is the representation of a single Node: -



Below is simple representation of a Linked List: -



Simple linked list representation.

Karthik Chinni

Just a Meme: - 😊


```
public class Node {
	public int data;
	public Node next;

	Node(){

	}
	public Node(int data) {
		this.data = data;
	}
}
```

```
public class LinkedListTraversal {
	private void LLTraversal(Node head) {
		if (head == null) // Base condition
			return;
		Node p = head; // Never do any direct operation on head
		while (p != null) {
			System.out.println(p.data);
			p = p.next;
		}
	}
}
```

Karthik Chinni

Class Explanation:

```
private static void linkedListTraversal(Node head) {
    if (head == null) // Base condition
        return;
    Node p = head; // Never do any direct operation on head
    while (p != null) {
        System.out.print(p.data);
        p = p.next;
    }
}
head-->28-->27-->26-->25-->null
                           ^
                           p
Output
======
28 27 26 25
```

Solve below Hacker Rank problem: -

https://www.hackerrank.com/challenges/print-the-elements-of-a-linked-list/problem

Finding Length of a LinkedList: -

```
public class LinkedListLength {
    public static int getLinkedListLength(Node head) {
        if (head == null) {
            return 0;
        }
        Node p = head;
        int count = 0;
        while (p != null) {
            count++;
            p = p.next;
        }
        return count;
    }
}
```

O(n)-->TC

O(1)-->SC

Karthik Chinni

Insert first in Linked List

```
public Node insertFirst(Node head, int x) {
          Node q = new Node(x);
          // Base Condition
          if (head == null) {
               return q;
          }
          Node p = head;
          q.next = p;
          return q;
}
```

O(1)-->TC

O(1)-->SC

https://www.hackerrank.com/challenges/insert-a-node-at-the-head-of-a-linked-list/problem

Karthik Chinni

Insert First in LinkedList: -

```
public static Node insertFirst(Node head, int x) {
	Node p = new Node(x);
	if (head == null) {
		return p;
	}
	p.next = head;
	return p;
}
```

Insert Last in LinkedList: -

```
public static Node insertLast(Node head, int x) {
	Node p = new Node(x);
	if (head == null) {
		return p;
	}
	Node q = head;
	while (q.next != null) {
		q = q.next;
	}
	q.next = p;
	return head;
}
```

Insert Last in LinkedList: -

```
public static Node insertAtIndex(Node head, int x, int index) {
	int lenght = LinkedListLength.getLinkedListLength(head);
	Node updatedLinkedList = null;
	if (index > lenght) {
		return null;// You can throw an exception here!
	}
	if (index == 0) {
		updatedLinkedList = insertFirst(head, x);
		return updatedLinkedList;
	}
	if (index == lenght) {
		updatedLinkedList = insertLast(head, x);
		return updatedLinkedList;
	}
	Node p = head;
	Node q = new Node(x);
	int count = 1;
	while (p.next != null) {
		Node r = p.next;
		if (index == count) {
			p.next = q;
			q.next = r;
			break;
		}
		p=p.next;
		count++;
	}
	return head;
}
```

Hacker Rank Solution: -

```
static SinglyLinkedListNode insertNodeAtPosition(SinglyLinkedListNode head, int da
ta, int position) {
    return insertAtIndex(head, data, position);
}
public static SinglyLinkedListNode insertFirst(SinglyLinkedListNode head, int x) {
    SinglyLinkedListNode p = new SinglyLinkedListNode(x);
    if (head == null) {
        return p;
    }
    p.next = head;
    return p;
}

public static SinglyLinkedListNode insertLast(SinglyLinkedListNode head, int x) {
    SinglyLinkedListNode p = new SinglyLinkedListNode(x);
    if (head == null) {
        return p;
    }
    SinglyLinkedListNode q = head;
    while (q.next != null) {
        q = q.next;
    }
    q.next = p;
    return head;
}

public static SinglyLinkedListNode insertAtIndex(SinglyLinkedListNode head, int x,
 int index) {
    int lenght = getLinkedListLength(head);
    SinglyLinkedListNode updatedLinkedList = null;
    if (index > lenght) {
        return null;// You can throw an exception here!
    }
    if (index == 0) {
        updatedLinkedList = insertFirst(head, x);
        return updatedLinkedList;
    }
    if (index == lenght) {
        updatedLinkedList = insertLast(head, x);
        return updatedLinkedList;
    }
    SinglyLinkedListNode p = head;
    SinglyLinkedListNode q = new SinglyLinkedListNode(x);
    int count = 1;
    while (p.next != null) {
```

Karthik Chinni

```
        SinglyLinkedListNode r = p.next;
        if (index == count) {
            p.next = q;
            q.next = r;
            break;
        }
        p=p.next;
        count++;
    }
    return head;
}
//Not needed as per our Hacker Rank Problem.
public static int getLinkedListLength(SinglyLinkedListNode head) {
    if (head == null) {
        return 0;
    }
    SinglyLinkedListNode p = head;
    int count = 0;
    while (p != null) {
        count++;
        p = p.next;
    }
    return count;
}
```

Solve below Hacker Rank problem: -

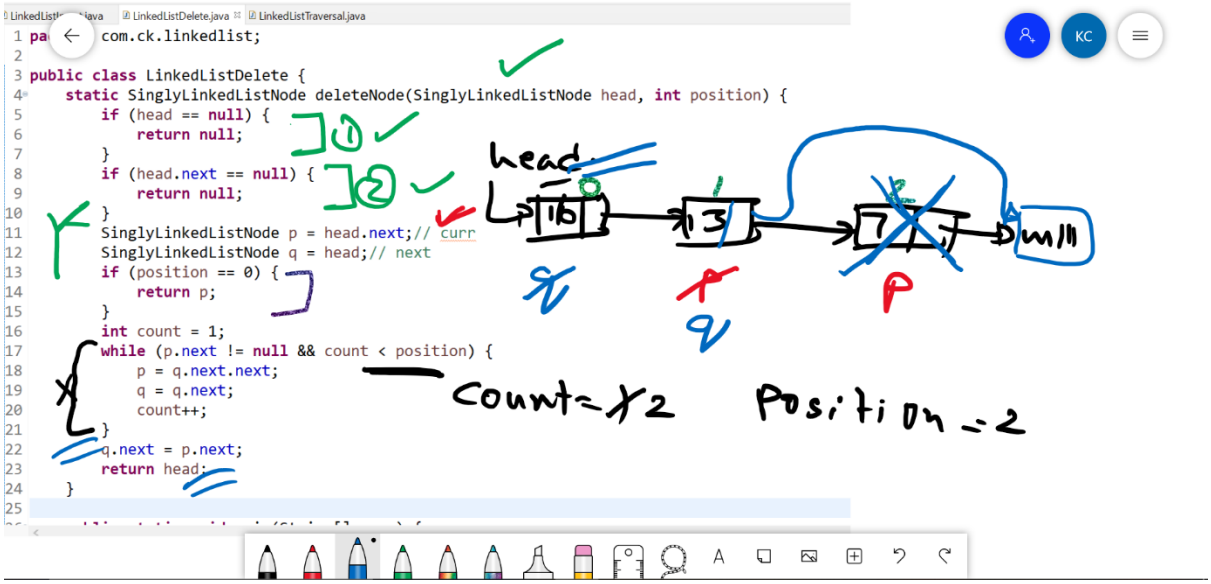
https://www.hackerrank.com/challenges/insert-a-node-at-a-specific-position-in-a-linked-list/problem

https://www.hackerrank.com/challenges/insert-a-node-at-the-tail-of-a-linked-list/problem

Deleting in Linked List

There are three places where deletion can happen.

1. Delete First
2. Delete Last
3. Delete at position



Code in eclipse: -

```
public class LinkedListDelete {
	static SinglyLinkedListNode deleteNode(SinglyLinkedListNode head, int position) {
		if (head == null) {
			return null;
		}
		if (head.next == null) {
			return null;
		}
		SinglyLinkedListNode p = head.next;// curr
		SinglyLinkedListNode q = head;// next
		if (position == 0) {
			return p;
		}
		int count = 1;
		while (p.next != null && count < position) {
			p = q.next.next;
			q = q.next;
			count++;
		}
		q.next = p.next;
		return head;
	}

	public static void main(String[] args) {
		SinglyLinkedListNode head = new SinglyLinkedListNode(16);
		SinglyLinkedListNode node13 = new SinglyLinkedListNode(13);
		SinglyLinkedListNode node7 = new SinglyLinkedListNode(7);
		head.next = node13;
		node13.next = node7;
		deleteNode(head, 2);
		LinkedListTraversal.linkedListTraversalNew(head);
	}
}
```

Karthik Chinni

Hacker Rank code: -

```
static SinglyLinkedListNode deleteNode(SinglyLinkedListNode head, int position) {
        if(head==null){
            return null;
        }
        if(head.next==null){
            return null;
        }
        SinglyLinkedListNode p = head.next;//curr
        SinglyLinkedListNode q = head;//next
        if(position==0){
            return p;
        }
        int count =1;
        while(p.next!=null && count<position){
            p=q.next.next;
            q=q.next;
            count++;
        }
        System.out.println("p="+p.data);
        q.next=p.next;
        return head;
    }
```

Find middle element of a LinkedList :-

Approach 1 :

```
public class FindMiddleOfLinkedList {
      public static Node findMiddle(Node head) {
            if(head == null) {
                  return null;
            }
            if(head.next==null) {
                  return head;
            }
            Node p = head;
            int lenght = LinkedListLength.getLinkedListLength(head);
            int middle = (lenght/2)-1;
            int count = 0;
            while(count<middle) {
                  p=p.next;
                  count++;
            }
            return p;
      }
```

Karthik Chinni

```
	public static void main(String[] args) {
		Node head = new Node(28);
		Node node27 = new Node(27);
		Node node26 = new Node(26);
		Node node25 = new Node(25);
		Node node24 = new Node(24);
		head.next = node27;
		node27.next = node26;
		node26.next = node25;
		node25.next = node24;
		Node middleNode = findMiddle(head);
		System.out.println(middleNode.data);
	}
}
```

Time Complexity -> O(n)+O(n/2) =O(n)

Space Complexity -> O(1)

Approach 2:-

```
	private static Node getMiddleElementOfLinkedList(Node head) {
		if (head == null)// Base condition
			return null;
		Node p = head;// slow pointer
		Node q = head;// fast pointer
		while (q != null && q.next != null) {
			p = p.next;
			q = q.next.next;
		}
		return p;
	}
```

Time Complexity -> O(n/2) =O(n)

Space Complexity -> O(1)

Is Circular Linked List

```
public class IsCircularLinkedList {

	public static boolean isCircular(Node head) {
		if (head == null || head.next == null || head.next.next == null) {
			return false;
		}
		Node p = head;// slow pointer
		Node q = head;// fast pointer
		while (q != null && q.next != null) {
			p = p.next;
			q = q.next.next;
			if (p == q) {
				return true;
			}
		}
		return false;
	}
```

Karthik Chinni

```
        public static void main(String[] args) {
                Node head = new Node(28);
                Node node28 = head;
                Node node27 = new Node(27);
                Node node26 = new Node(26);
                Node node25 = new Node(25);
                Node node24 = new Node(24);
                head.next = node27;
                node27.next = node26;
                node26.next = node25;
                node25.next = node24;
                node24.next = node28;
                boolean isCircular = isCircular(head);
                System.out.println(isCircular);
        }
}
```

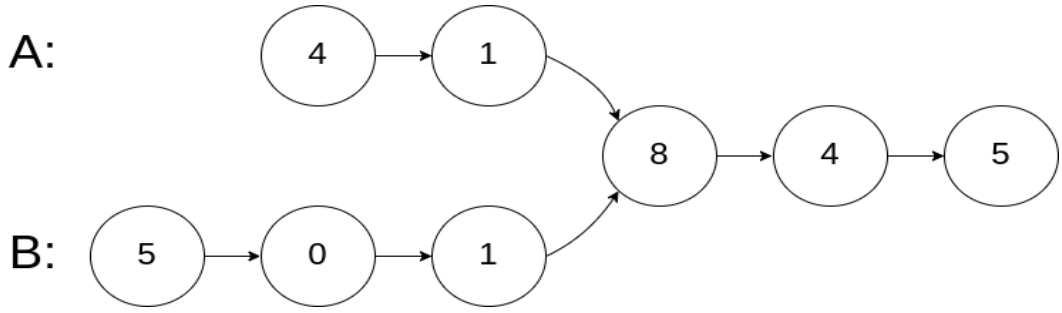
Find if Loop exists: -

```
    static boolean hasCycle(SinglyLinkedListNode head) {
       if (head == null || head.next == null || head.next.next == null) {
           return false;
       }
       SinglyLinkedListNode p = head;// slow pointer
       SinglyLinkedListNode q = head;// fast pointer
       while (q != null && q.next != null) {
           p = p.next;
           q = q.next.next;
           if (p == q) {
               return true;
           }
       }
       return false;
    }
```

Link:

https://www.hackerrank.com/challenges/detect-whether-a-linked-list-contains-a-cycle/problem

Intersection point in a Linked List :-



Karthik Chinni

Algorithm: -

1- Finalize the base conditions
2- Get lengths of two linked lists
3- Start with LinkedList with larger length and iterate until the length difference.
4- Start iterating both LinkedLists until the next pointer becomes null
    a. If both nodes are equal in iteration, return true
    b. Else, return false.

Program: -

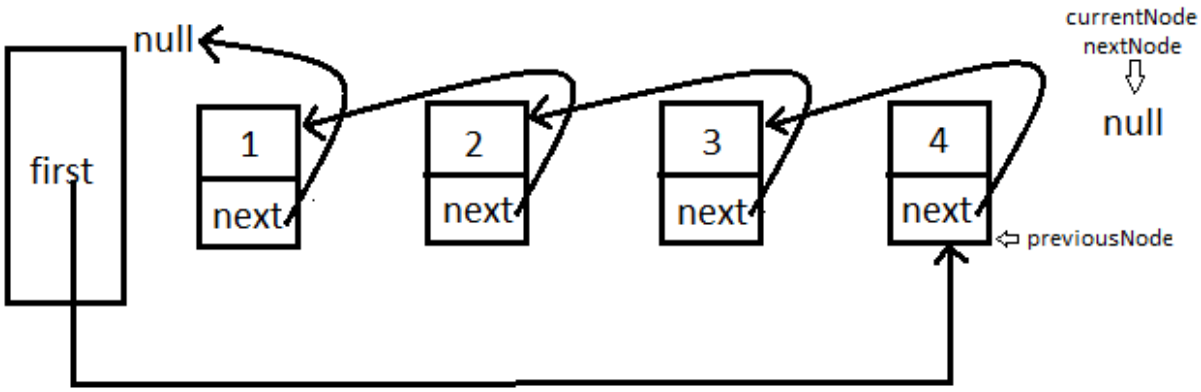
```
public class IntersectionPoint {
	public Node getIntersectionNode(Node headA, Node headB) {
	  //Base condition
		if(headA==null || headB==null) {
			return null;
		}
		// Step 1
		int lenA = getLength(headA);
		int lenB = getLength(headB);
		// Step 2
		Node p = headA;
		Node q = headB;
		while (lenA > lenB) {
			p = p.next;
			lenA--;
		}
		while (lenA < lenB) {
			q = q.next;
			lenB--;
		}
		// Step 3
		while (p != q) {
			p = p.next;
			q = q.next;
		}
		return p;
    }
	private int getLength(Node node) {
		int length = 0;
		while (node != null) {
			node = node.next;
			length++;
		}
		return length;
	}
}
```

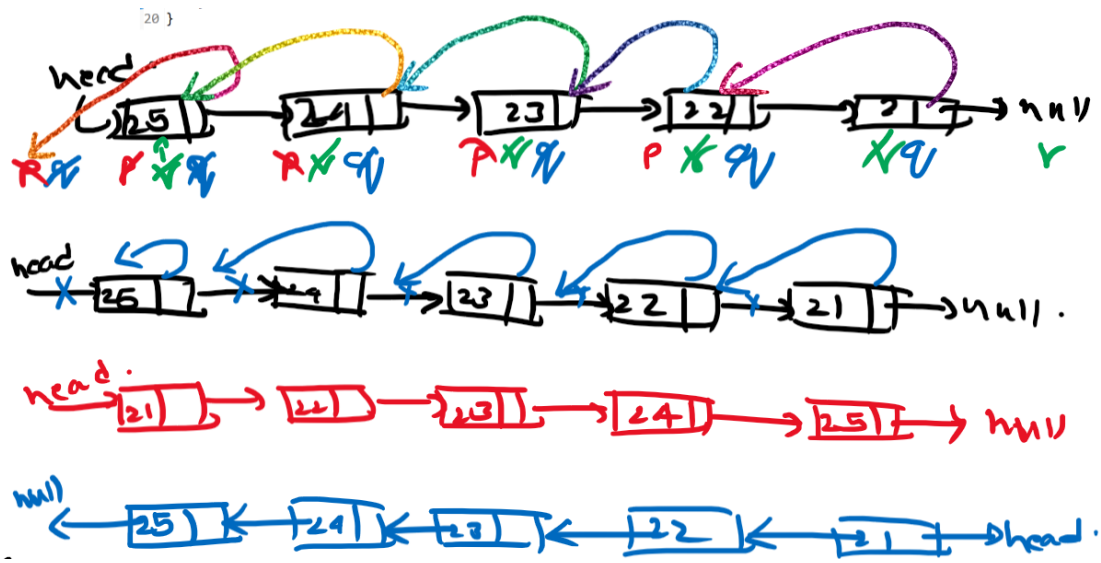
LeetCode Link: -

https://leetcode.com/problems/intersection-of-two-linked-lists/

Karthik Chinni

Find reverse of a LinkedList: -



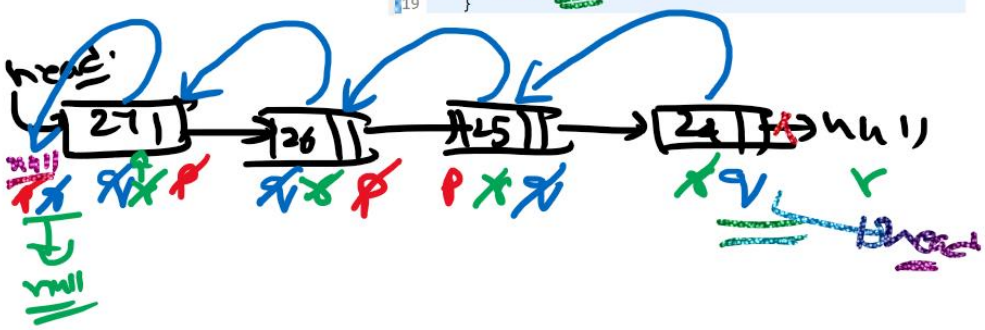



```
public class ReverseLinkedList {
	static Node Reverse(Node head) {
		//Base Condition
		if (head == null) {
			return null;
		}
		Node p = null;//previous
		Node q = null;//current
		Node r = head;//next
		while (r != null) {
			p = q;
			q = r;
			r = r.next;
			q.next = p;//q is behind p. Actual reverse logic.
		}
		return q;
	}
```

Karthik Chinni

```
        public static void main(String[] args) {
                Node head = new Node(28);
                Node node27 = new Node(27);
                Node node26 = new Node(26);
                Node node25 = new Node(25);
                head.next = node27;
                node27.next = node26;
                node26.next = node25;
                LinkedListTraversal.linkedListTraversal(head);
                Node reverseLinkedList = Reverse(head);
                LinkedListTraversal.linkedListTraversal(reverseLinkedList);
        }
}
```

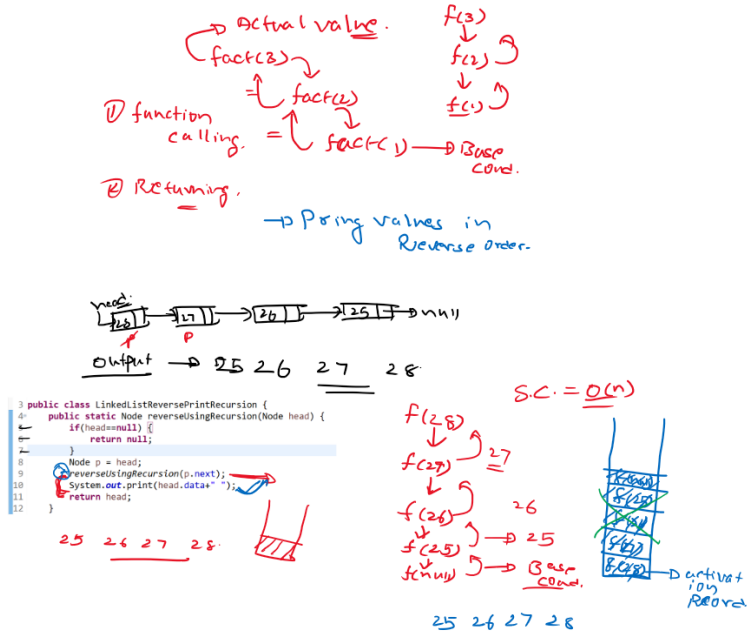
```
4   static Node Reverse(Node head) {
5       //Base Condition
6       if (head == null) {
7           return null;
8       }
9       Node p = null;//previous
10      Node q = null;//current
11      Node r = head;//next
12      while (r != null) {
13          p = q;
14          q = r;
15          r = r.next;
16          q.next = p;//q is behind p. Actual reverse logic
17      }
18      return q;
19  }
```



Print Reverse order of a LinkedList using Recursion :-

```
public static Node reverseUsingRecursion(Node head) {
		if(head==null) {
			return null;
		}
		Node p = head;
		reverseUsingRecursion(p.next);
		System.out.print(head.data+" ");
		return head;
	}
	public static void main(String[] args) {
		Node head = new Node(28);
		Node node27 = new Node(27);
		Node node26 = new Node(26);
		Node node25 = new Node(25);
		head.next = node27;
		node27.next = node26;
		node26.next = node25;
		reverseUsingRecursion(head);
	}
```

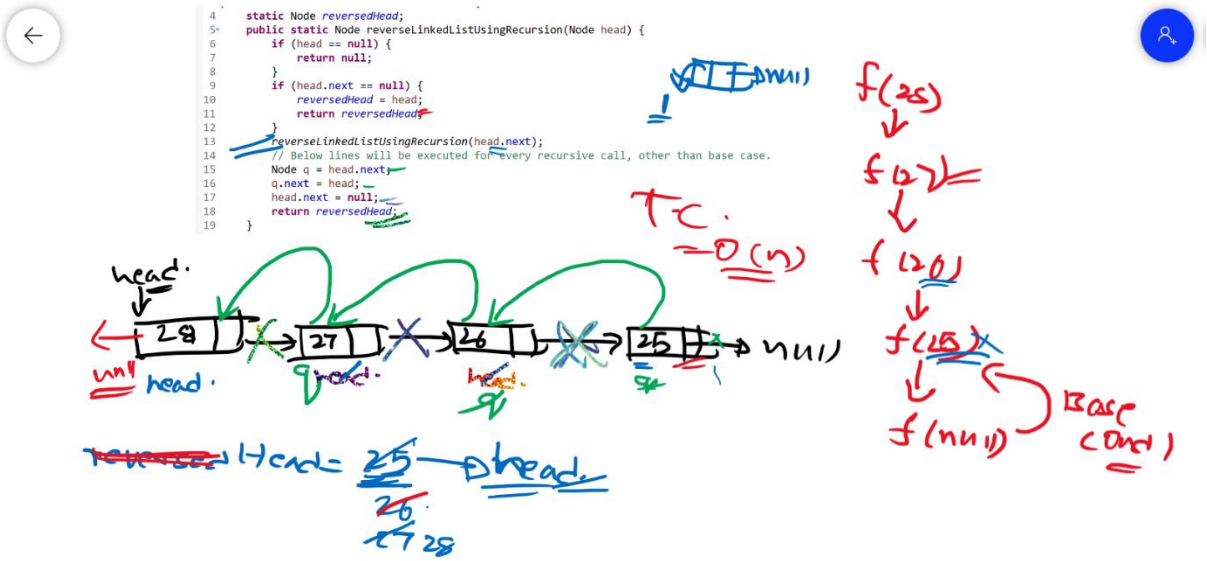
Explanation: -



Karthik Chinni

Using Recursion: -

```java
public class LinkedListReverseRecursion {
	static Node reversedHead;
	public static Node reverseLinkedListUsingRecursion(Node head) {
		if (head == null) {
			return null;
		}
		if (head.next == null) {
			reversedHead = head;
			return reversedHead;
		}
		reverseLinkedListUsingRecursion(head.next);
		// executed for every recursive call, other than base case.
		Node q = head.next;
		q.next = head;
		head.next = null;
		return reversedHead;
	}

	public static void main(String[] args) {
		Node head = new Node(28);
		Node node27 = new Node(27);
		Node node26 = new Node(26);
		Node node25 = new Node(25);
		head.next = node27;
		node27.next = node26;
		node26.next = node25;
		Node reversedLinkedList = reverseLinkedListUsingRecursion(head);
		LinkedListTraversal.linkedListTraversal(reversedLinkedList);
	}
}
```



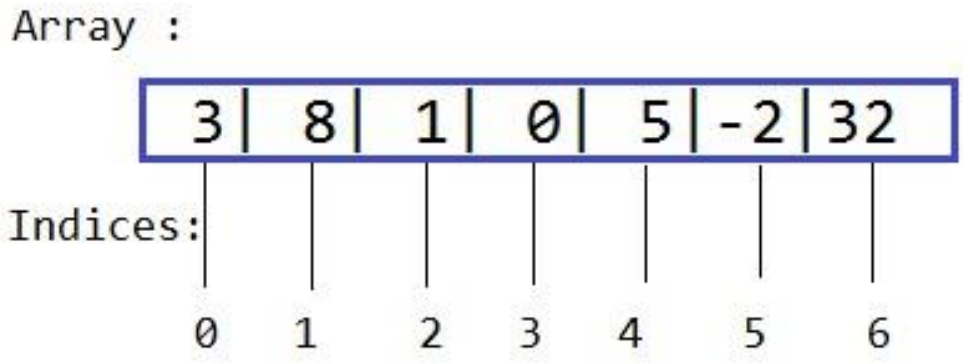
Hacker Rank Link: -

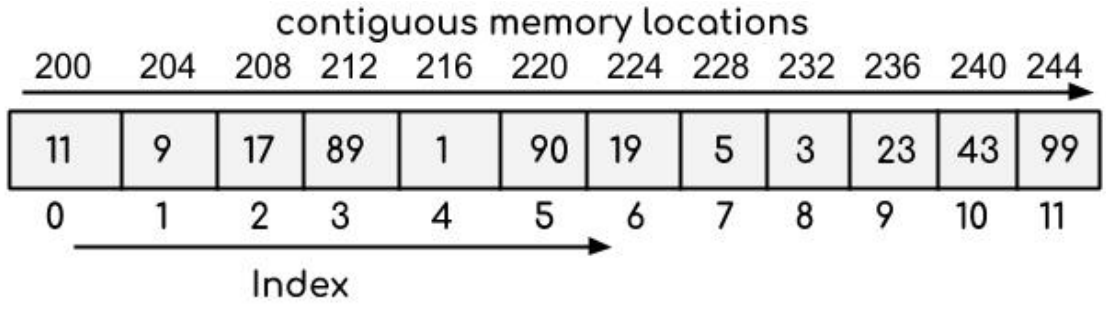
https://www.hackerrank.com/challenges/reverse-a-linked-list/problem

Karthik Chinni

Array: -

- ✓ The length of an array is fixed.
- ✓ An array is a collection of homogeneous (same type) data items.
- ✓ The data items are stored in contiguous memory locations.






```
public class ArrayTraverse {
	public static void main(String[] args) {
		int[] strArray = { 11, 9, 17, 89, 1, 90, 19, 5, 3, 23, 43, 99 };
		for (int i = 0; i < strArray.length; i++) {
			System.out.println(strArray[i]);
		}
	}
}
```

Time Complexity			Time Complexity		
	Array	LinkedList		Array	LinkedList
Insert@End	O(1)	O(n)	Delete@End	O(1)	O(n)
Insert@Begin	O(n)	O(1)	Delete@Begin	O(n)	O(1)

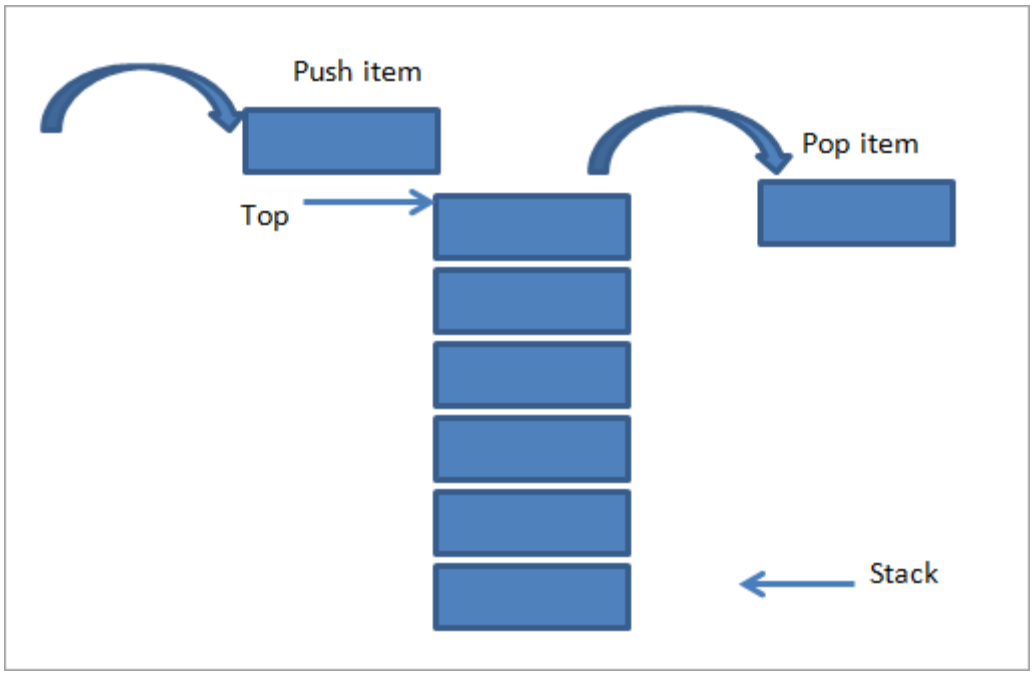
Karthik Chinni

Stacks and Queues:-

Both Stacks and Queues are **Logical data structures.**

Stack :-

1. Insertion and Deletion happens at one end only, called as **top**.
2. This follows **LIFO** (Last in First Out)
3. It supports two main operations:
    1. **push()** => used for insertion on top.
    2. **pop()** => used for deletion from top.
4. This can be implemented using arrays and linked lists.

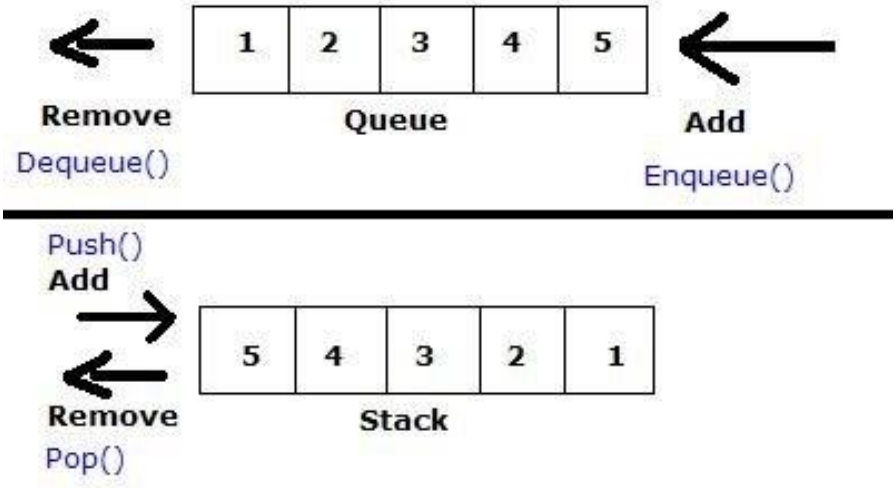


Queue: -

5. Insertion and Deletion happens at two different ends called as **front** and **rear.**.
6. This follows **FIFO**(First In First Out)
7. It supports two main operations :
a. **enque()** => used for insertion on top.
b. **deque()** => used for deletion from top.
8. This can be implemented using arrays and linked lists.

Karthik Chinni

Difference between Stack and Queue:-



Conditions: -

	Using Array	
	Stack with Array	**Queue with Array**
initial	top = -1	front = -1, rear = -1
empty	top == -1	front == rear
full	top == lenght -1	rear = length -1
elements count	top + 1	rear - front

	Using LinkedList	
	Stack With LL	**Queue with LL**
inital	top = null	front = null, rear = null
empty	top == null	front == null
full	n/a	n/a
elements count	maintain count	maintain count

Karthik Chinni

```java
public class MyStackArr {
	static int top = -1;// Initial Condition
	static int[] arr = new int[5];// Max size of 500

	public static void push(int x) {
		if (!isFull())
			arr[++top] = x;
		else
			System.out.println("Stack OverFlow!");
	}

	public static int pop() {
		if (!isEmpty())
			return arr[top--];
		else
			System.out.println("Stack UnderFlow!");

		return -1;// Assume all numbers in stack are positive.
	}

	private static boolean isEmpty() {
		if (top == -1)
			return true;
		else
			return false;
	}

	private static boolean isFull() {
		if (top == arr.length - 1)
			return true;
		else
			return false;
	}

	public static void main(String[] args) {
		push(10);
		push(20);
		push(30);
		push(40);
		push(50);
		printStack();
		System.out.println("Testing pop()");
		System.out.println("Popped out value from stack : "+pop());
		System.out.println("Popped out value from stack : "+pop());
		System.out.println("Popped out value from stack : "+pop());
	}

	private static void printStack() {
		for (int i = 0; i < arr.length; i++) {
			if(arr[i]!=0) {
				System.out.print(arr[i]+" ");
			}
		}
	}
}
```

Karthik Chinni

```java
public class MyQueueArr {
	int rear, front = -1;// Initial condition
	int[] arr = new int[100];

	public void enQueue(int x) {
		if (!isFull()) {
			arr[++rear] = x;
		}
	}

	public int deQueue() {
		if (!isEmpty()) {
			return arr[++front];
		}
		return -1;// Assume numbers in stack are positive.
	}

	private boolean isEmpty() {
		if (front == rear)
			return true;
		else
			return false;
	}

	private boolean isFull() {
		if (rear == arr.length - 1)
			return true;
		else
			return false;
	}
}
```

```java
public class MyQueueArr {
	int rear, front = -1;// Initial condition
	int[] arr = new int[100];

	public void enQueue(int x) {
		if (!isFull()) {
			arr[++rear] = x;
		}
	}

	public int deQueue() {
		if (!isEmpty()) {
			return arr[++front];
		}
		return -1;// Assume numbers in stack are positive.
	}

	private boolean isEmpty() {
		if (front == rear)
			return true;
		else
			return false;
	}

	private boolean isFull() {
		if (rear == arr.length - 1)
			return true;
		else
			return false;
	}
}
```

Karthik Chinni

```java
public class MyQueueLL {

	Node front, rear = null;

	public void push(int x) {
		Node p = new Node(x);
		rear.next = p;
	}

	public int pop() {
		if (!isEmpty()) {
			int x = front.data;
			front = front.next;
			return x;
		}
		return -1;// Assume numbers in stack are positive.
	}

	private boolean isEmpty() {
		if (front == null)
			return true;
		else
			return false;
	}
}
```

https://leetcode.com/problems/implement-queue-using-stacks/

1- Implement queue using two stacks

```java
public class MyQueueUsingTwoStacks {
	static MyStackArr insertStack;
	static MyStackArr deleteStack ;
	static void enQueue(int x) {
		insertStack.push(x);
	}
	static int deQueue() {
		if(!deleteStack.isEmpty()) {
			return deleteStack.pop();
		}
		while(!insertStack.isEmpty()) {
			deleteStack.push(insertStack.pop());
		}
		return deleteStack.pop();
	}
}
```

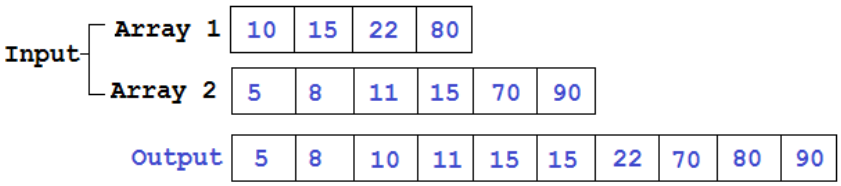
2- Implement stack using two queues

```
public class MyStackUsingTwoQueues {
	static MyQueueArr myQueueInsert;
	static MyQueueArr myQueueDelete;

	void push(int x) {
		myQueueInsert.enQueue(x);
	}

	void pop() {
		while (myQueueInsert.getLenght() > 1) {
			myQueueDelete.enQueue(myQueueInsert.deQueue());
		}
		MyQueueArr temp = myQueueInsert;
		myQueueInsert = myQueueDelete;
		myQueueDelete = temp;
		myQueueDelete.deQueue();
	}
}
```

3- Implement stack using one queue

```
public class MyStackUsingOneQueue {
	static MyQueueArr originalQueue;
	void push(int x) {
		originalQueue.enQueue(x);
	}
	int pop() {
		int size = originalQueue.getLenght();
		while(size>1) {
			originalQueue.enQueue(originalQueue.deQueue());
			size--;
		}
		return originalQueue.deQueue();
	}
}
```

4- Merge two sorted arrays




```java
public class MergingTwoSortedArraysVerbose {
	static int[] merge(int[] arr1, int[] arr2) {
		int[] result = new int[arr1.length + arr2.length];
		// Base case 1
		if (arr1.length == 0 && arr2.length == 0) {
			return result;
		}
		// Base case 2
		if (arr1.length == 0) {
			return arr2;
		}
		// Base case 3
		if (arr2.length == 0) {
			return arr1;
		}
		int i, j, k;
		i = j = k = 0;
		while (i < arr1.length && j < arr2.length) {
			if (arr1[i] < arr2[j]) {
				result[k] = arr1[i];
				i++;
				k++;
			} else if (arr1[i] > arr2[j]) {
				result[k] = arr2[j];
				j++;
				k++;
			}
		}
		while (i < arr1.length) {
			result[k++] = arr1[i++];
		}
		while (j < arr2.length) {
			result[k++] = arr2[j++];
		}
		return result;
	}

	public static void main(String[] args) {
		int[] arr1 = { 1, 3, 5, 7, 9, 20, 30, 40 };
		int[] arr2 = { 2, 4, 6, 8 };
		int[] result = merge(arr1, arr2);
		for (int i = 0; i < result.length; i++) {
			System.out.print(result[i] + " ");
		}
	}
}
```

Karthik Chinni

Brute force solution: -

TC->O(n^2), SC->O(1)

```
public class IsUnique {
	public static boolean isUnique(String input) {
		char[] charArray = input.toCharArray();
		for (int i = 0; i < charArray.length; i++) {
			for (int j = i + 1; j < charArray.length; j++) {
				if (charArray[j] == charArray[i]) {
					return false;
				}
			}
		}
		return true;
	}
	public static void main(String[] args) {
		String input = "Helo Hai";
		System.out.println(input+"  "+isUnique(input));
	}
}
```

Efficient Solution: -

TC->O(n), SC->O(1)

```
public class IsUniqueEfficient {
	public static boolean isUnique(String input) {
		boolean[] chars = new boolean[256];
		char[] charArray = input.toCharArray();
		for (int i = 0; i < charArray.length; i++) {
			int value = charArray[i];
			if (chars[value]) {
				return false;
			}
			chars[value] = true;
		}
		return true;
	}
	public static void main(String[] args) {
		String input = "Helo hai";
		System.out.println(input+"  "+isUnique(input));
	}
}
```

Panagram Anaslysis: -

Character	ASCII
a	97
b	98
c	99
d	100
e	101
f	102
g	103
h	104
i	105
j	106
k	107
l	108
m	109

Character	ASCII
n	110
o	111
p	112
q	113
r	114
s	115
t	116
u	117
v	118
w	119
x	120
y	121
z	122

Character	ASCII
A	65
B	66
C	67
D	68
E	69
F	70
G	71
H	72
I	73
J	74
K	75
L	76
M	77

Character	ASCII
N	78
O	79
P	80
Q	81
R	82
S	83
T	84
U	85
V	86
W	87
X	88
Y	89
Z	90

Character	ASCII
0	48
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57

Pangrams are words or sentences containing every letter of the alphabet at least once; the best-known English example being **A quick brown fox jumps over the lazy dog.**

Algorithm

1- Handle base cases.
   a. null || size == 0 → Invalid input.
   b. size<26 → Not Panagram.
2- Iterate over input.
   a. mark true for every char in input.
3- Iterate over mark array.
   a. if any index is not marked(false) → not Panagram.
4- String is a Panagram.

```
public class CheckAsciiValue {
     public static void main(String[] args) {
          printAsciiValue('a');
     }
     static void printAsciiValue(char ch) {
          System.out.println((int) ch);
     }
}
```

Karthik Chinni

```java
public class PanagramCheck {
	public static void checkPanagram(String input) {
		// Base case 1
		if (input == null || input.length() == 0) {
			System.out.println("Invalid Input");
		}
		// Base case 2
		if (input.length() < 26) {
			System.out.println("Not Panagram");
			return;
		}
		boolean[] markArray = new boolean[26];
		for (int i = 0; i < input.length(); i++) {
			if (input.charAt(i) >= 'A' && input.charAt(i) <= 'Z') {// caps
				markArray[input.charAt(i) - 'A'] = true;
			} else if (input.charAt(i) >= 'a' && input.charAt(i) <= 'z') {// small case
				markArray[input.charAt(i) - 'a'] = true;
			} else {// other char
				continue;
			}
		}
		for (int i = 0; i < markArray.length; i++) {
			if(!markArray[i]) {
				System.out.println("Not panagram");
				return;
			}
		}
		System.out.println("Panagram");
	}

	public static void main(String[] args) {
		checkPanagram("The quick brown fox jumps over the lazy dog");
	}
}
```

Ascii Code	Ascii Value	Ascii Code	Ascii Value	Index	MarkArray
65	A	97	a	0	
66	B	98	b	1	
67	C	99	c	2	
68	D	100	d	3	
69	E	101	e	4	
70	F	102	f	5	
71	G	103	g	6	
72	H	104	h	7	
73	I	105	i	8	
74	J	106	j	9	
75	K	107	k	10	
76	L	108	l	11	
77	M	109	m	12	
78	N	110	n	13	

Karthik Chinni

79	**O**	111	**o**	14	
80	**P**	112	**p**	15	
81	**Q**	113	**q**	16	
82	**R**	114	**r**	17	
83	**S**	115	**s**	18	
84	**T**	116	**t**	19	
85	**U**	117	**u**	20	
86	**V**	118	**v**	21	
87	**W**	119	**w**	22	
88	**X**	120	**x**	23	
89	**Y**	121	**y**	24	
90	**Z**	122	**z**	25	

Trees

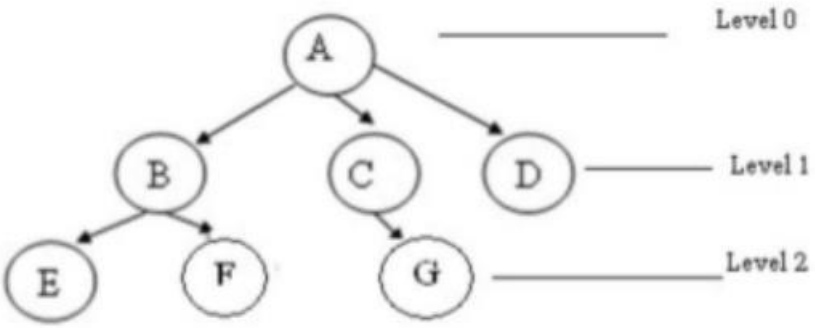
This is how a normal tree looks like

Karthik Chinni

But for programmers, it's all different. It's inverted.

Terminology: -

- **Node:** Each element in the tree.
- **Edge:** Line connecting nodes.
- **Parent Node:** Immediate predecessor of a Node.
- **Child Node:** Immediate successor of a Node.
- **Root Node:** Node without parent.
- **Grand Parent Node:** Parent of Parent.
- **Level:** Distance of Node from root.
- **Siblings:** Children of the same parent Node.
- **Degree:** Maximum number of children a node can have.
- **Leaf Node**: Node without children.
- **Height/Depth**: Level of tree counted from root to the lowermost leaf node.
- **External Node**: All Leaf Nodes.
- **Internal Nodes**: All Non-Leaf Nodes.



- ✔ A is the root node
- ✔ B is the parent of E and F
- ✔ D is the sibling of B and C
- ✔ E and F are children of B
- ✔ E, F, G, D are external nodes or leaves
- ✔ A, B, C are internal nodes
- ✔ Depth of F is 2
- ✔ the height of tree is 2
- ✔ the degree of node A is 3
- ✔ The degree of tree is 3

Binary Trees: -

Tree with degree two is called a **Binary Tree.**

Class Representation of a tree: -

```
public class BTNode {
     int data;
     BTNode[] children = new BTNode[degree];
}
```

For a BinaryTree, the degree is 2, so, it can be represented as below: -

```
public class BTNode {
     int data;
     BTNode[] children = new BTNode[2];
}
```

BinaryTree can be represented as below to understand easier: -

```
public class BTNode{
     int data;
     Node left;
     Node right;
}
```

```
public class PreOrder {
	public static void preOrder(Node root) {
		// Base condition
		if (root == null) {
			return;
		}
		System.out.print(root.data + " ");
		preOrder(root.left);
		preOrder(root.right);
	}
}
```

TC → O(n)

SC → O(h)

```
public class InOrder {
	public static void inOrder(Node root) {
		// Base condition
		if (root == null) {
			return;
		}
		inOrder(root.left);
		System.out.print(root.data + " ");
		inOrder(root.right);
	}
}
```

TC → O(n)

SC → O(h)

```
public class PostOrder {
	public static void postOrder(Node root) {
		// Base condition
		if (root == null) {
			return;
		}
		postOrder(root.left);
		postOrder(root.right);
		System.out.print(root.data + " ");
	}
}
```

Karthik Chinni

TC → O(n)

SC → O(h)

```
public class HeightBTree {
	static int height(Node root) {
		// Base Condition
		if (root == null) {
			return -1;
		}

		int leftHeight = height(root.left);
		int rightHeight = height(root.right);
		if (leftHeight > rightHeight) {
			return leftHeight + 1;
		} else {
			return rightHeight + 1;
		}
	}
}
```

TC → O(n)

SC → O(h)

```
public class NumberOfNodesInBinaryTree {
	static int getNodesCount(Node root) {
		if (root == null) {
			return 0;
		}
		int leftCount = getNodesCount(root.left);
		int rightCount = getNodesCount(root.right);
		return leftCount + rightCount + 1;
	}

	public static void main(String[] args) {
		Node root = new Node(1);
		root.left = new Node(2);
		root.right = new Node(3);
		root.left.left = new Node(4);
		root.left.right = new Node(5);
		System.out.println(getNodesCount(root));
	}
}
```

TC → O(n)

SC → O(h)

```
public class MaximumElementInBinaryTree {
```

```
	static int getMaxElement(Node root) {
		// Base Condition
		if (root == null) {
			return Integer.MIN_VALUE;
		}
		int leftMax = getMaxElement(root.left);
		int rightMax = getMaxElement(root.right);
		return Math.max(root.data, Math.max(leftMax, rightMax));
	}

	public static void main(String[] args) {
		Node root = new Node(-1);
		root.left = new Node(-2);
		root.right = new Node(-3);
		root.right.left = new Node(-4);
		root.right.right = new Node(-5);
		System.out.println(getMaxElement(root));
	}
}
```

TC → O(n)

SC → O(h)

```
public class NumberOfLeafNodes {
	static int getNumberOfLeafNodes(Node root) {
		if (root == null) {
			return 0;
		}
		if (root.left == null && root.right == null) {
			return 1;
		}
		int leftLeaves = getNumberOfLeafNodes(root.left);
		int rightLeaves = getNumberOfLeafNodes(root.right);
		return leftLeaves + rightLeaves;
	}

	public static void main(String[] args) {
		Node root = new Node(1);
		root.left = new Node(2);
		root.right = new Node(3);
		root.right.left = new Node(4);
		root.right.right = new Node(5);
		System.out.println(getNumberOfLeafNodes(root));
	}
}
```

TC → O(n)

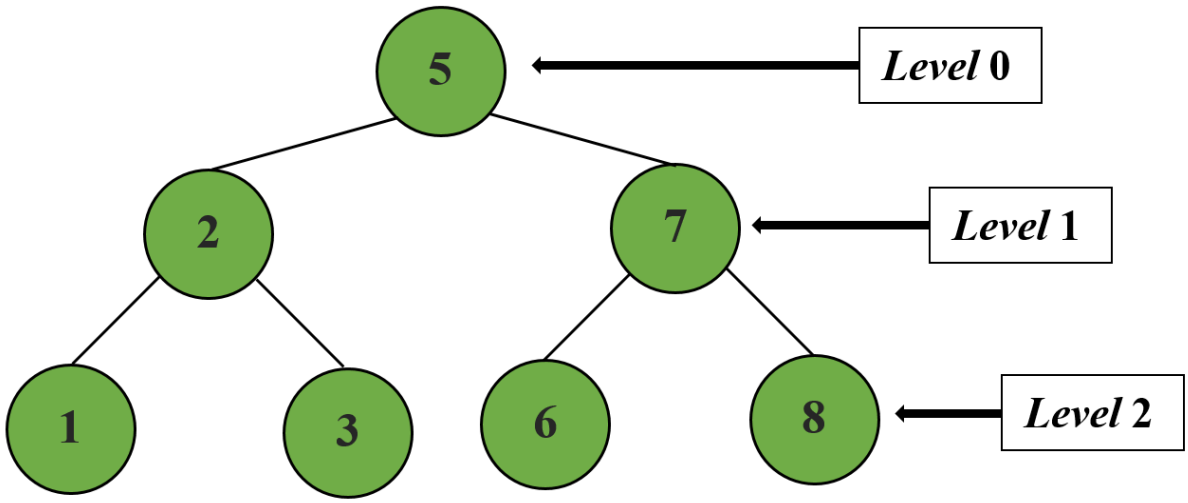
SC → O(h)

```java
public class SearchForElementInBinaryTree {
	static Node search(Node root, int x) {
		if (root == null) {
			return null;
		}
		if (root.data == x) {
			return root;
		}
		Node leftSearch = search(root.left, x);
		if (leftSearch != null) {
			return leftSearch;
		}
		Node rightSearch = search(root.right, x);
		if (rightSearch != null) {
			return rightSearch;
		}
		return null;
	}

	public static void main(String[] args) {
		Node root = new Node(1);
		root.left = new Node(2);
		root.right = new Node(3);
		root.right.left = new Node(4);
		root.right.right = new Node(5);
		System.out.println(search(root, 2).data);
	}
}
```

TC → O(n)

SC → O(h)

Level Order Traversal: -

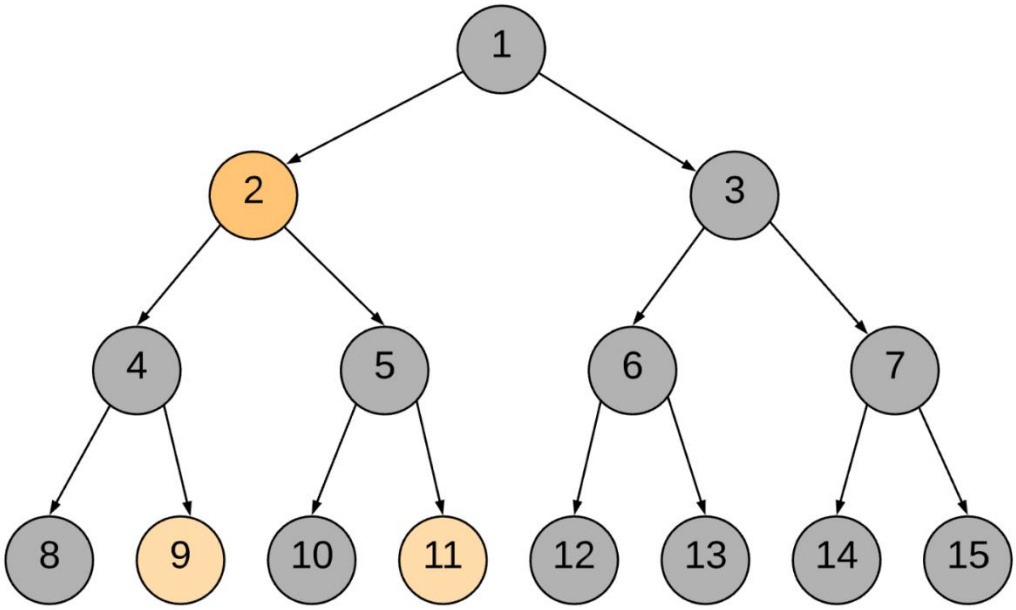



```
public class LevelOrderTraversal {
	public static void printLevel(Node root, int level) {
		if (root == null) {
			return;
		}
		if (level == 1) {
			System.out.print(root.data + " ");
		}
		printLevel(root.left, level - 1);
		printLevel(root.right, level - 1);
	}

	public static void main(String[] args) {
		Node root = new Node(1);
		root.left = new Node(2);
		root.right = new Node(3);
		root.right.left = new Node(4);
		root.right.right = new Node(5);
		int height = HeightBTree.height(root);
		for (int i = 1; i <= height; i++) {
			printLevel(root, i);
		}
	}
}
```

TC → O(n * h)

SC → O(h)

Lowest/Closest Common Ancestor



Lowest Common Ancestor for **Node 9** and **Node 11** is **Node 2**

```
public class LowestCommonAncestor {
	static Node lca(Node root, int node1, int node2) {
		if (root == null) {
			return null;
		}
		if (root.data == node1 || root.data == node2) {
			return root;
		}
		Node Llca = lca(root.left, node1, node2);
		Node Rlca = lca(root.right, node1, node2);
		if (Llca == null && Rlca == null) {
			return null;
		}
		if (Llca == null) {
			return Rlca;
		}
		if (Rlca == null) {
			return Llca;
		}
		return root;
	}
}
```

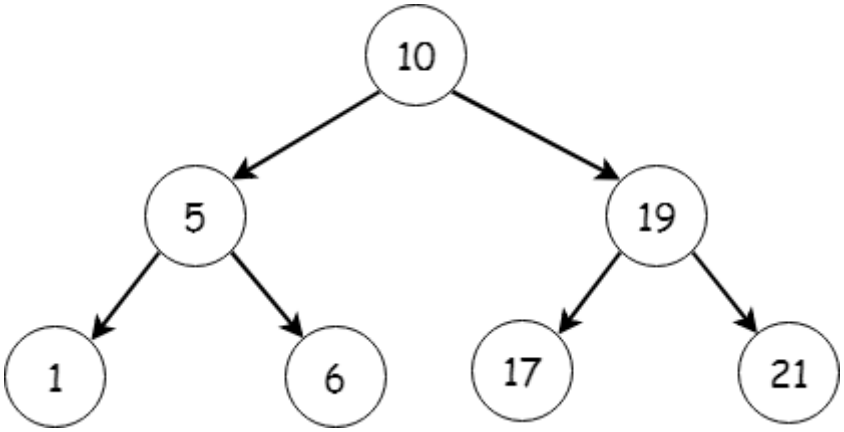
TC → O(n)

SC → O(h)

Karthik Chinni

Binary Search Tree: -

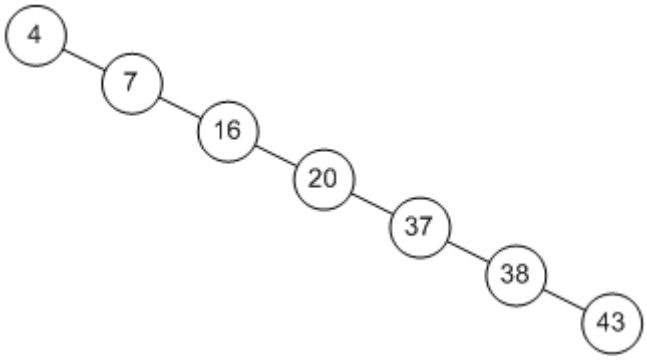
BST→BT+Constraint (Constraint→left.data<root.data<right.data)



Why BST?

Searching in BT is done in O(n). But, searching is a basic operation and this can be done better. So, the above constraint is introduced so that either left or right subtree is ignored for every iteration. So, the searching can be done in o(h).

But problem still exists in worst case: -



So, in worst case, h is still equal to n.

So, the tree needs to be Balanced or Full Binary Tree to get O(log n) time complexity for searching. This can be achieved via rotations:

BinarySearchTree Analysis			
Height	**for given n**	**number of nodes**	**for given h**
min	log n	**min**	h
max	n	**max**	2^n

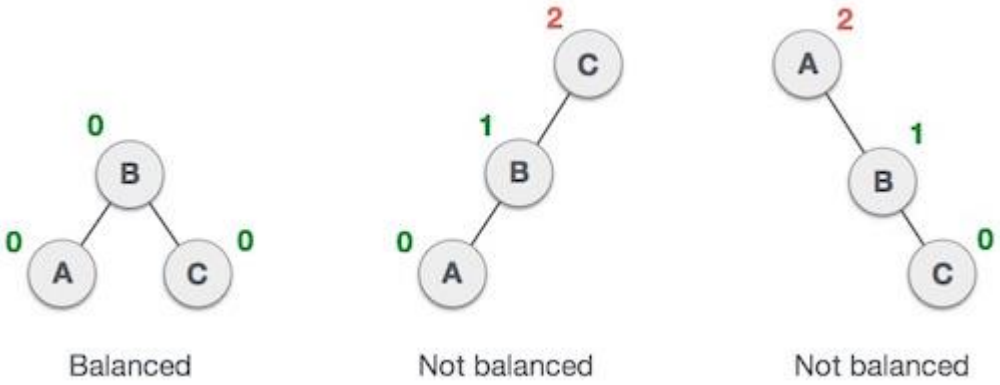
It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson**, **Velski** & **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Here we see that the first tree is balanced and the next two trees are not balanced –

Steps: -

1- Insert
2- Can we do better? (calculate Balance Factor)
3- Rorate.



In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

BalanceFactor = height(left-subtree) − height(right- subtree)subtree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

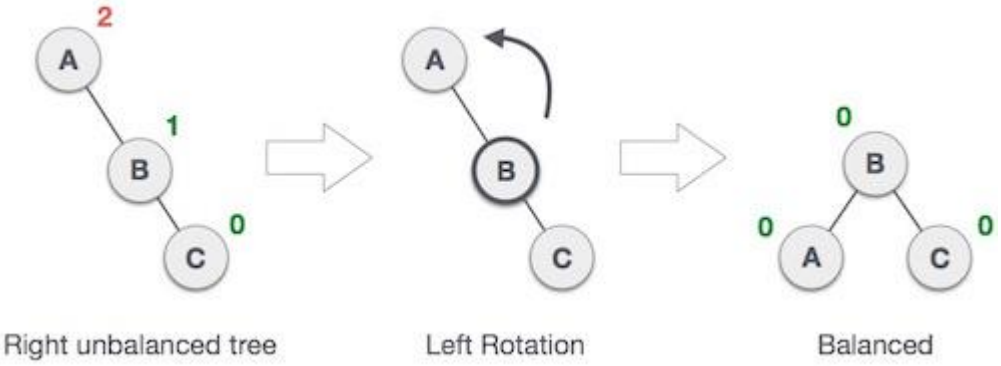
To balance itself, an AVL tree may perform the following four kinds of rotations −

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

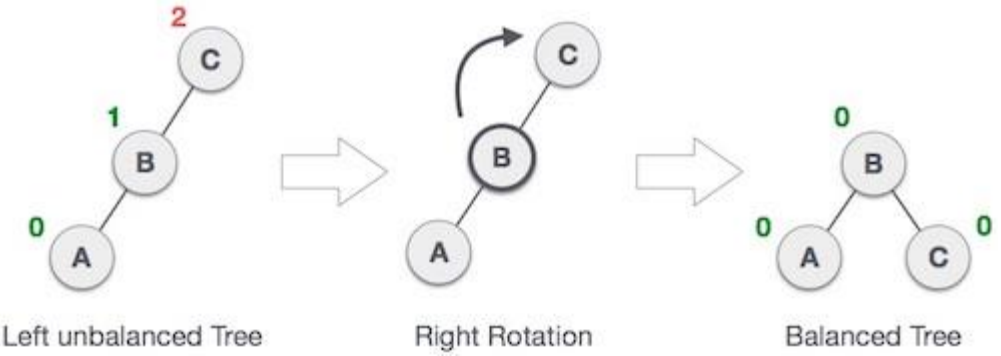
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



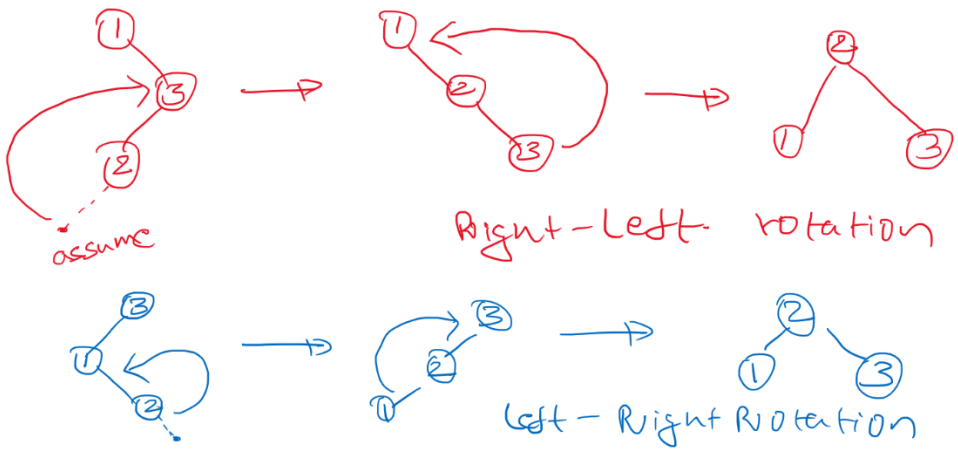
In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.



Karthik Chinni

```java
public class SearchForElementInBinarySearchTree {
	static Node search(Node root, int x) {
		if (root == null) {
			return null;
		}
		if (root.data == x) {
			return root;
		} else if (root.data < x) {
			return search(root.right, x);
		} else {
			return search(root.left, x);
		}
	}

	public static void main(String[] args) {
		Node root = new Node(2);
		root.left = new Node(1);
		root.right = new Node(4);
		root.right.left = new Node(3);
		root.right.right = new Node(6);
		System.out.println(search(root, 3).data);
	}
}


public class InsertBST {
	public static Node insert(Node root, int data) {
		if (root == null) {
			root = new Node(data);
			return root;
		}
		if (data < root.data) {
			root.left = insert(root.left, data);
		} else {
			root.right = insert(root.right, data);
		}
		return root;
	}
}
public class IsBST {
	public boolean isBST(Node root) {
		if (root == null) {
			return true;
		}
		if (root.left != null && root.left.data > root.data) {
			return false;
		}
		if (root.right != null && root.right.data > root.data) {
			return false;
		}
		return isBST(root.left) && isBST(root.right);
	}
}
```

Karthik Chinni

On the below line: -

```
if (root.left != null && root.left.data > root.data)
```

it should be: -

```
if (root.left != null && max(root.left.data) > root.data)
```

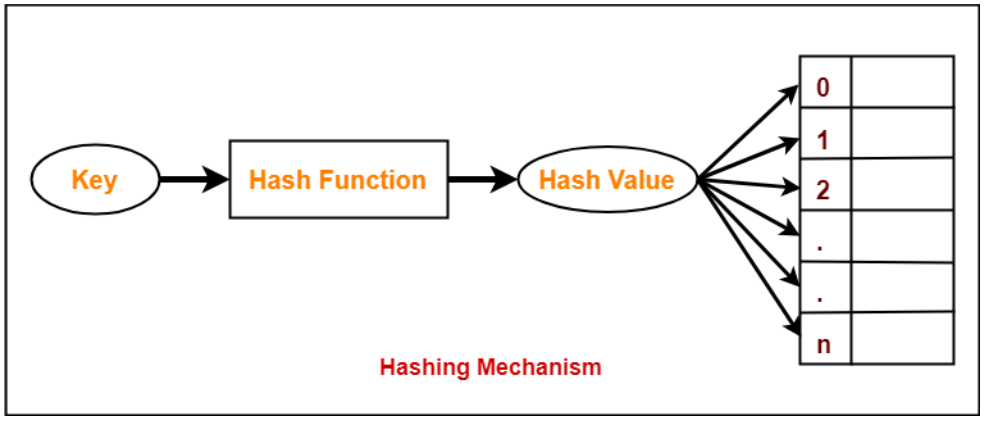
So, now the TC will become **O(n^2)**

So, the comparison should not be with children, but it should be with parent.

```
public class IsBSTBetter {
	public boolean isBST(Node root) {
		return isBSTHealper(root, 0, Integer.MAX_VALUE);
	}

	public boolean isBSTHealper(Node root, int min, int max) {
		if (root == null) {
			return true;
		}
		if (root.data <= min || root.data > max) {
			return false;
		}
		return isBSTHealper(root.left, min, root.data) &&
				isBSTHealper(root.right, root.data, max);

	}
}
```

Hashing: -

Hashing is the process of converting a given key into another value. A **hash function** is used to generate the new value according to a mathematical algorithm. The result of a hash function is known as a **hash value** or simply, a **hash**.



Karthik Chinni

Hashing can be used to build, search, or delete from a table.

The basic idea behind hashing is to take a field in a record, known as the **key**, and convert it through some fixed process to a numeric value, known as the **hash key**, which represents the position to either store or find an item in the table. The numeric value will be in the range of 0 to n-1, where n is the maximum number of slots (or **buckets**) in the table.

The fixed process to convert a key to a hash key is known as a **hash function**. This function will be used whenever access to the table is needed.

One common method of determining a hash key is the **division method** of hashing. The formula that will be used is:

hash key = key % number of slots in the table

Assume a table with 8 slots:

Hash key = key % table size

4 = 36 % 8

2 = 18 % 8

0 = 72 % 8

3 = 43 % 8

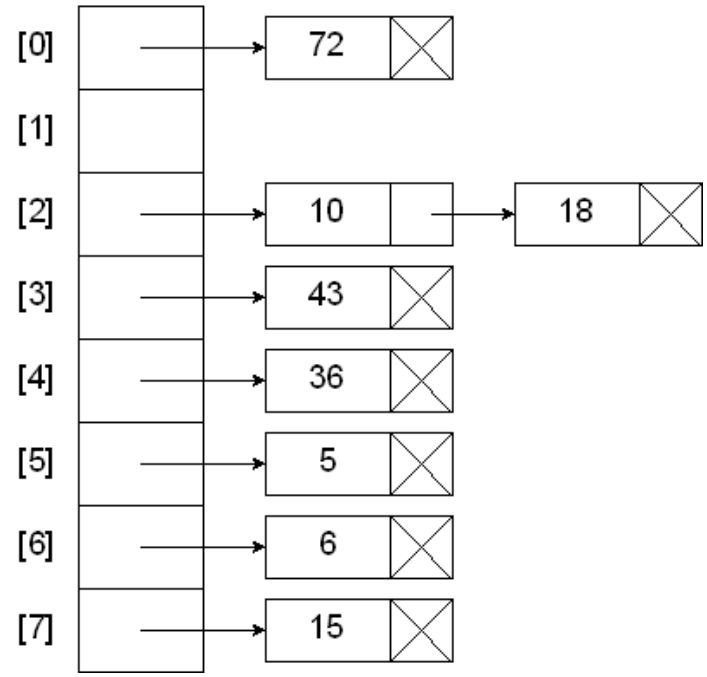
6 = 6 % 8

[0]	72
[1]	
[2]	18
[3]	43
[4]	36
[5]	
[6]	6
[7]	

The problem with above is that there may me collisions. In that case, we can do something like below: -

Hash key = key % table size

4 = 36 % 8
2 = 18 % 8
0 = 72 % 8
3 = 43 % 8
6 = 6 % 8
2 = 10 % 8
5 = 5 % 8
7 = 15 % 8



The above is called as Open Addressing. The problem with above that say our numbers are pointing to the same bucket, then the linked list size increases and that the hash table has many empty buckets in it. We can overcome this by using closed addressing.

Hashing with Linear Probe

When using a linear probe, the item will be stored in the next available slot in the table, assuming that the table is not already full.

This is implemented via a linear search for an empty slot, from the point of collision. If the physical end of table is reached during the linear search, the search will wrap around to the beginning of the table and continue from there.

If an empty slot is not found before reaching the point of collision, the table is full.

[0]	72
[1]	
[2]	18
[3]	43
[4]	36
[5]	
[6]	6
[7]	

Add the keys 10, 5, and 15 to the previous table .

Hash key = key % table size

2 = 10 % 8

5 = 5 % 8

7 = 15 % 8

[0]	72
[1]	15
[2]	18
[3]	43
[4]	36
[5]	10
[6]	6
[7]	5

A problem with the linear probe method is that it is possible for blocks of data to form when collisions are resolved. This is known as **primary clustering**.

Hashing with Quadratic Probe

To resolve the primary clustering problem, **quadratic probing** can be used. With quadratic probing, rather than always moving one spot, move i^2 spots from the point of collision, where i is the number of attempts to resolve the collision.

89 % 10 = 9

18 % 10 = 8

49 % 10 = 9 – 1 attempt needed – 1^2 = 1 spot

58 % 10 = 8 – 3 attempts – 3^2 = 9 spots

69 % 10 = 9 – 2 attempts – 2^2 = 4 spots

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

Hashing with Double Hashing

Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions form the point of collision to insert.

There are a couple of requirements for the second function:

- it must never evaluate to 0
- must make sure that all cells can be probed

A popular second hash function is: $Hash_2(key) = R - (key \% R)$ where R is a prime number that is smaller than the size of the table.

Table Size = 10 elements
$Hash_1(key) = key \% 10$
$Hash_2(key) = 7 - (k \% 7)$

Insert keys : 89, 18, 49, 58, 69

Hash(89) = 89 % 10 = 9

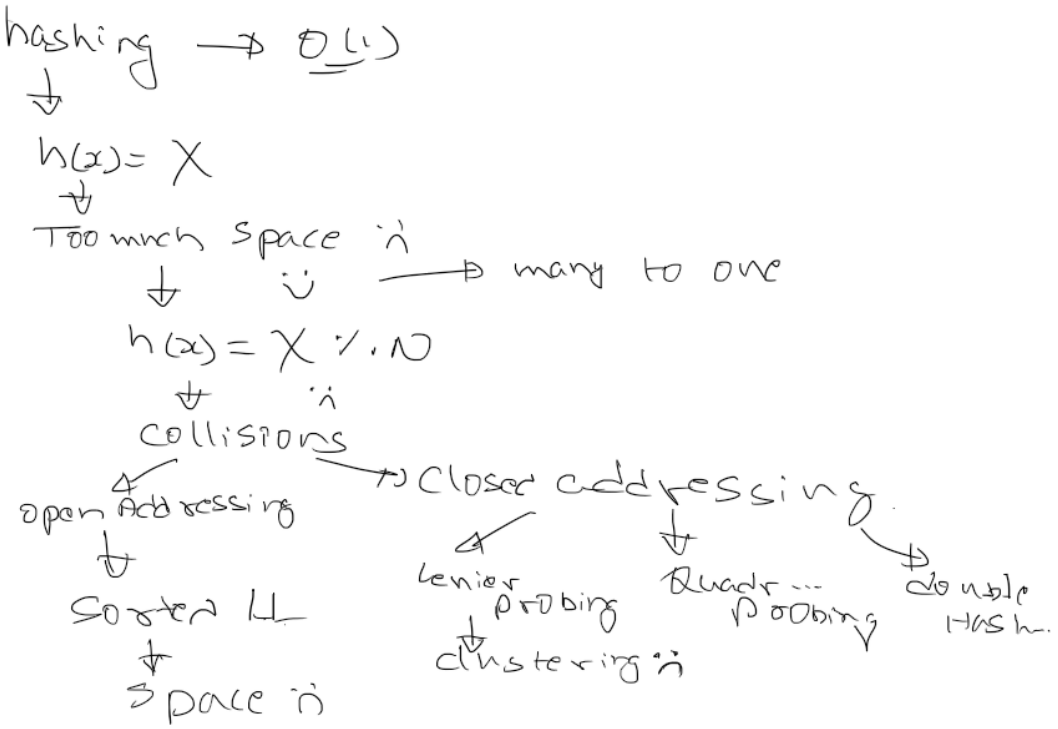
Hash(18) = 18 % 10 = 8

Hash(49) = 49 % 10 = 9 a collision !
= 7 – (49 % 7)
= 7 positions from [9]

Hash(58) = 58 % 10 = 8
= 7 – (58 % 7)
= 5 positions from [8]

Hash(69) = 69 % 10 = 9
= 7 – (69 % 7)
= 1 position from [9]

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

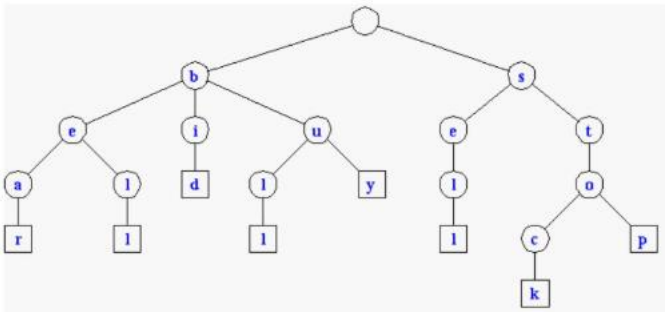


What is Trie?

Trie is an efficient information retrieval data structure. A Trie is a tree in which each node has many children. The value at each node consists of 2 things.

1. A character
2. A Boolean to say whether this character represents the end of a word.

Tries are also known as Prefix Trees.



Representation: -

```
public class TrieNode {
	private char c;
	private HashMap<Character, TrieNode> children = new HashMap<>();
	private boolean isLeaf;

	public TrieNode() {
	}

	public TrieNode(char c) {
		this.c = c;
	}

	public HashMap<Character, TrieNode> getChildren() {
		return children;
	}

	public void setChildren(HashMap<Character, TrieNode> children) {
		this.children = children;
	}

	public boolean isLeaf() {
		return isLeaf;
	}

	public void setLeaf(boolean isLeaf) {
		this.isLeaf = isLeaf;
	}
}
```

Karthik Chinni

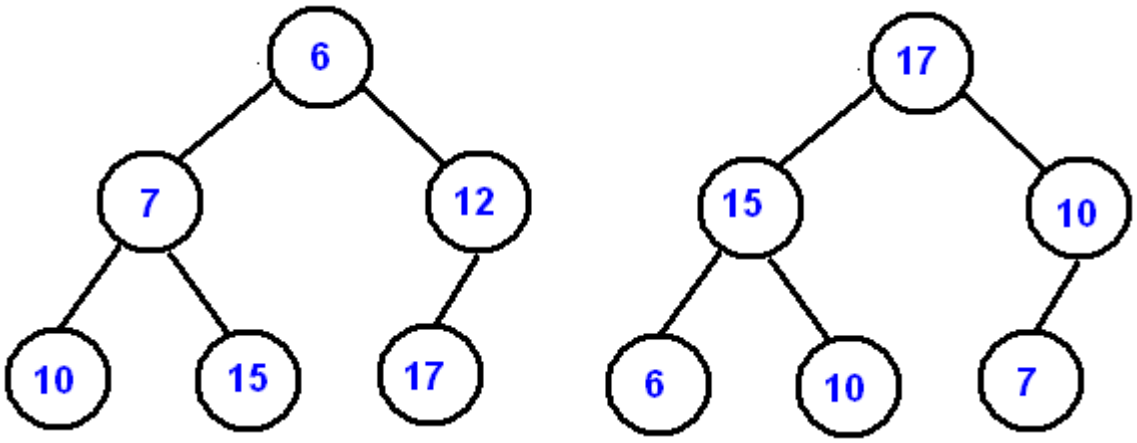
Basic Operations: -

```
public class Trie {
	private TrieNode root;

	public Trie() {
		root = new TrieNode();
	}

	public void insert(String word) {
		HashMap<Character, TrieNode> children = root.getChildren();
		for (int i = 0; i < word.length(); i++) {
			char c = word.charAt(i);
			TrieNode node;
			if (children.containsKey(c)) {
				node = children.get(c);
			} else {
				node = new TrieNode(c);
				children.put(c, node);
			}
			children = node.getChildren();
			if (i == word.length() - 1) {
				node.setLeaf(true);
			}
		}
	}

	public boolean search(String word) {
		HashMap<Character, TrieNode> children = root.getChildren();
		TrieNode node = null;
		for (int i = 0; i < word.length(); i++) {
			char c = word.charAt(i);
			if (children.containsKey(c)) {
				node = children.get(c);
				children = node.getChildren();
			} else {
				node = null;
				break;
			}
		}
		if (node != null && node.isLeaf()) {
			return true;
		} else {
			return false;
		}
	}
}
```

A **binary heap** is a complete binary tree which satisfies the heap ordering property.

There are two types of heap: -

1. **Min heap:** every parent is less than or equal to its children
2. **Max heap:** Every parent is greater than or equal to its children



A binary heap must be a complete tree, children are added at each level from left to right, and usually implemented as arrays. The maximum or minimum value will always be at the root of the tree, this is the advantage of using a heap.

For Heapify, the process of converting a binary tree into a heap, is often has to be done after an insertion or deletion.

To implement the heaps as arrays: -

1. We put the root at array[0]
2. Then we traverse each level from left to right, and so the left child of the root would go into array[1],its right child would be into array[2],etc.

For the node at array[i], we can get left child using this formula **(2i + 1)**, for the right child we can use this one **(2i + 2),** and for parent item **floor((i — 1) / 2)**.This works just with complete binary trees.

To insert into heap: -

1. Always add new items to the end of the array
2. Then we have to fix the heap(heapify process)
3. We compare the new item against its parent
4. If the item is greater than its parent, we swap it with its parent
5. We then rinse and repeat

Implementation: -

```
public class Heap {

	private int[] heap;
	private int size;

	public Heap(int capacity) {
		heap = new int[capacity];
	}

	public void insert(int value) {
		if (isFull()) {
			throw new IndexOutOfBoundsException("Heap is full");
		}
		heap[size] = value;
		fixHeapAbove(size);
		size++;
	}

	private void fixHeapAbove(int index) {
		int newValue = heap[index];
		while (index > 0 && newValue > heap[getParent(index)]) {
			heap[index] = heap[getParent(index)];
			index = getParent(index);
		}
		heap[index] = newValue;
	}

	public boolean isFull() {
		return heap.length == size;
	}

	public int getParent(int index) {
		return (index - 1) / 2;
	}
}
```

Default implementations in Java: -

```
public class HeapUsingJava {
    static PriorityQueue<Integer> minHeap = new PriorityQueue<>();
    static PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Comparator.reverseOrder());
}
```

Problem: -

You are given arrival and departure time of trains reaching to a particular station. You need to find minimum number of platforms required to accommodate the trains at any point of time.

```
arrival[] = {1:00, 1:40, 1:50, 2:00, 2:15, 4:00}
departure[] = {1:10, 3:00, 2:20, 2:30, 3:15, 6:00}
No. of platforms required in above scenario = 4
```

Explanation: -

Time	Activity	Platform_needed
1:00	Arrival	1
1:10	Departure	0
1:40	Arrival	1
1:50	Arrival	2
2:00	Arrival	3
2:15	Arrival	4
2:20	Departure	3
2:30	Departure	2
3:00	Departure	1
3:15	Departure	0
4:00	Arrival	1
6:00	Departure	0

1. Combine the departure and arrival list
2. Sort the list based on time in increasing order
3. for each element in new list
   3.1. If arrival time, arrival++
   3.2. If departure time, departure++
   3.3. Add (arrival - departure) into new list
4. Return maximum number in new list

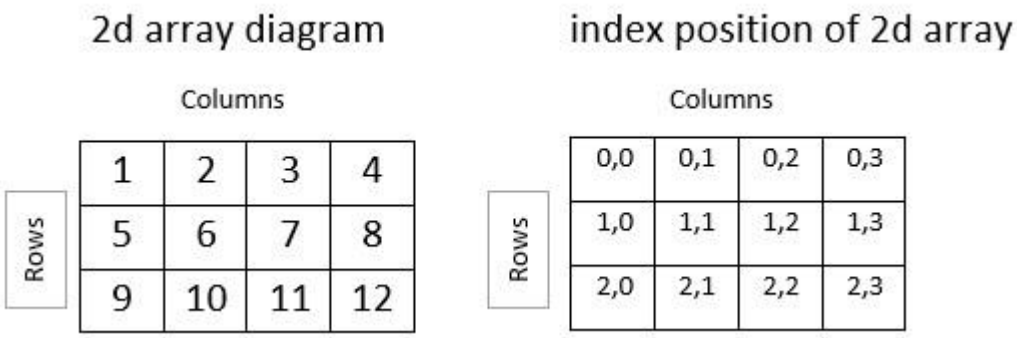
Karthik Chinni

```java
public class TrainTime {
	Integer time;
	Boolean isArrival;

	public TrainTime(Integer time, Boolean isArrival) {
		super();
		this.time = time;
		this.isArrival = isArrival;
	}

	public Integer getTime() {
		return time;
	}

	public void setTime(Integer time) {
		this.time = time;
	}

	public boolean isArrival() {
		return isArrival;
	}

	public void setArrival(boolean isArrival) {
		this.isArrival = isArrival;
	}

	@Override
	public String toString() {
		return "\n" + "TrainTime [time=" + time + ", isArrival=" + isArrival + "]";
	}
}
```

```java
public class PlatformProblem {
    public static int findMinPlatformsNeeded(List<TrainTime> trainsArrAndDepList) {
        System.out.println(trainsArrAndDepList);
        trainsArrAndDepList.sort((TrainTime obj1, TrainTime obj2) -> obj1.getTime().compareTo(obj2.getTime()));
        Integer arrival = 0;
        Integer departure = 0;
        System.out.println(trainsArrAndDepList);
        List<Integer> minPlatformsNeeded = new ArrayList<>();
        for (TrainTime trainTime : trainsArrAndDepList) {
            if (trainTime.isArrival()) {
                arrival++;
            } else {
                departure++;
            }
            minPlatformsNeeded.add(arrival - departure);
        }
        return Collections.max(minPlatformsNeeded);
    }
}
```

```java
public class PlatformProblemTest {
	public static void main(String[] args) {
		Integer arr[] = { 100, 140, 150, 200, 215, 400 };
		Integer dep[] = { 110, 300, 220, 230, 315, 600 };
		List<TrainTime> trainTimeList = new ArrayList<>();
		for (int i = 0; i < arr.length; i++) {
			trainTimeList.add(new TrainTime(arr[i], true));
		}
		for (int i = 0; i < dep.length; i++) {
			trainTimeList.add(new TrainTime(dep[i], false));
		}
		int minPlatformsNeeded = PlatformProblem.findMinPlatformsNeeded(trainTimeList);
		System.out.println(minPlatformsNeeded);
	}

}
```

Karthik Chinni



Search in a row wise and column wise sorted matrix

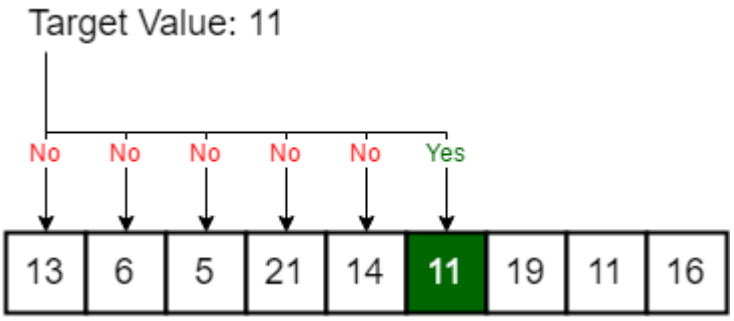
Given an n x n matrix and a number x, find the position of x in the matrix if it is present in it.

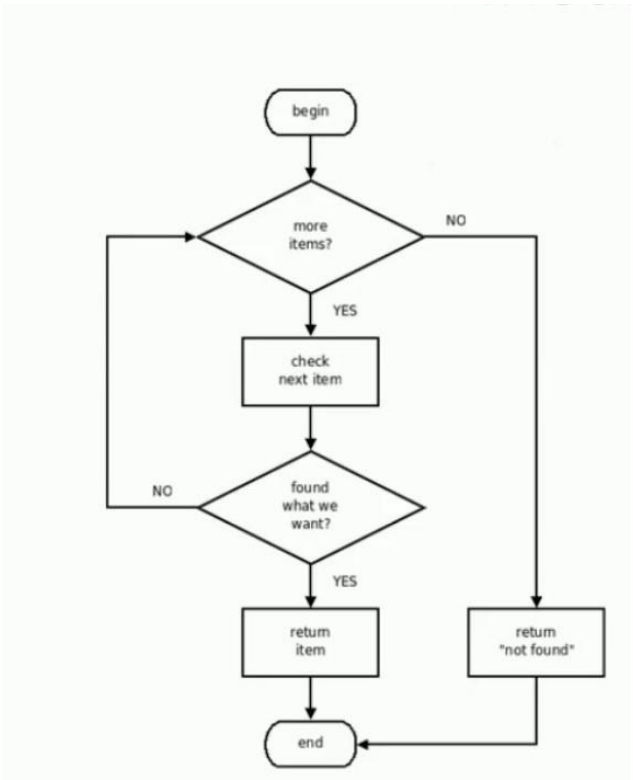
```
public class SearchSortedMatrix {
	private static void search(int[][] matrix, int n, int x) {
		// Base Condition.
		if (matrix.length == 0) {
			System.out.print("Invalid Input!");
			return;
		}
		int i = 0, j = n - 1;
		while (i < n && j >= 0) {
			if (matrix[i][j] == x) {
				System.out.print("Found number at Column: " + i + " Row: " + j);
				return;
			}
			if (matrix[i][j] > x) {
				j--;
			} else {
				i++;
			}
		}
		System.out.print("Number not found");
	}

	public static void main(String[] args) {
		int mat[][] = { { 10, 20, 30, 40 },
					{ 15, 25, 35, 45 },
					{ 27, 29, 37, 48 },
					{ 32, 33, 39, 50 } };
		// mat = new int[0][0];
		search(mat, 4, 29);
	}
}
```

Searching and Sorting Algorithms: -

1) Searching
   a) Linear Search
   b) Binary Search
2) Sorting
   a) Insertion Sort
   b) Selection Sort
   c) Bubble Sort
   d) Merge Sort
   e) Quick Sort
   f) Counting Sort
   g) Radix Sort
3) Time and Space Complexity Analysis

Linear Search





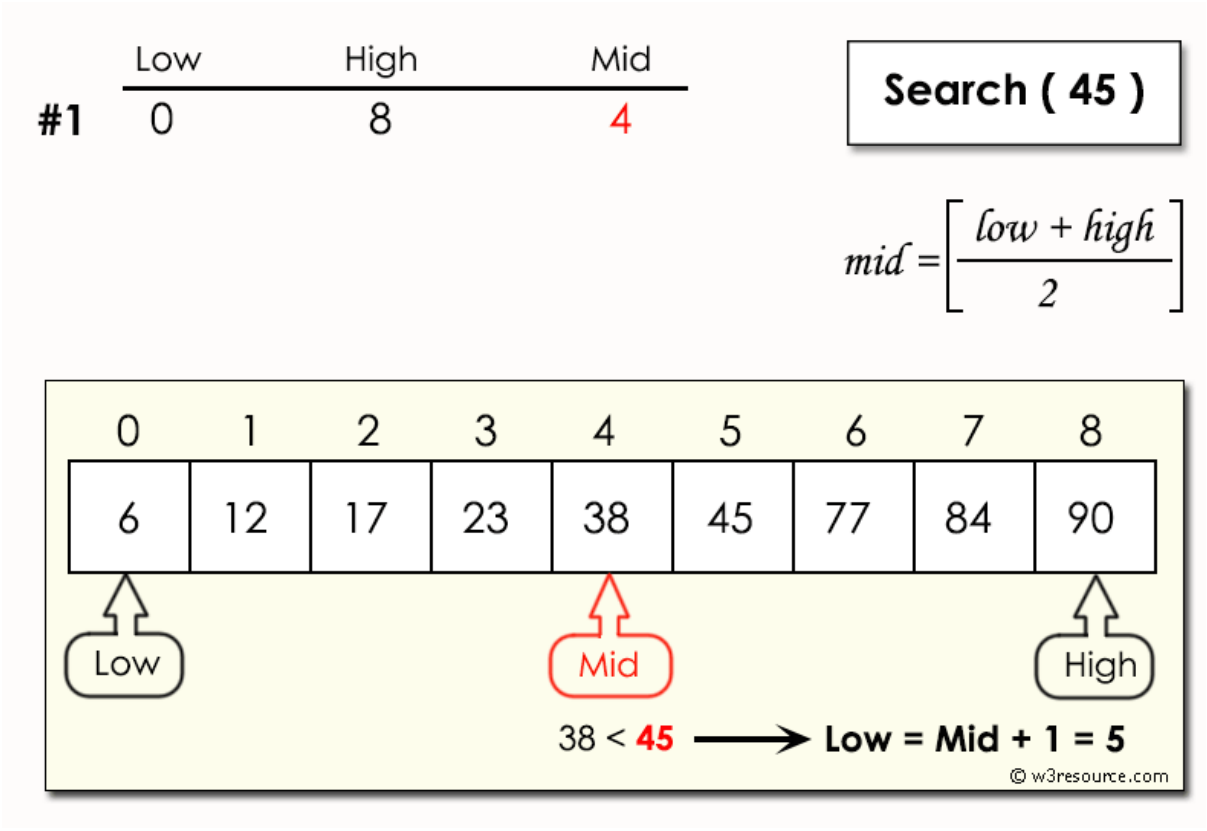
Karthik Chinni

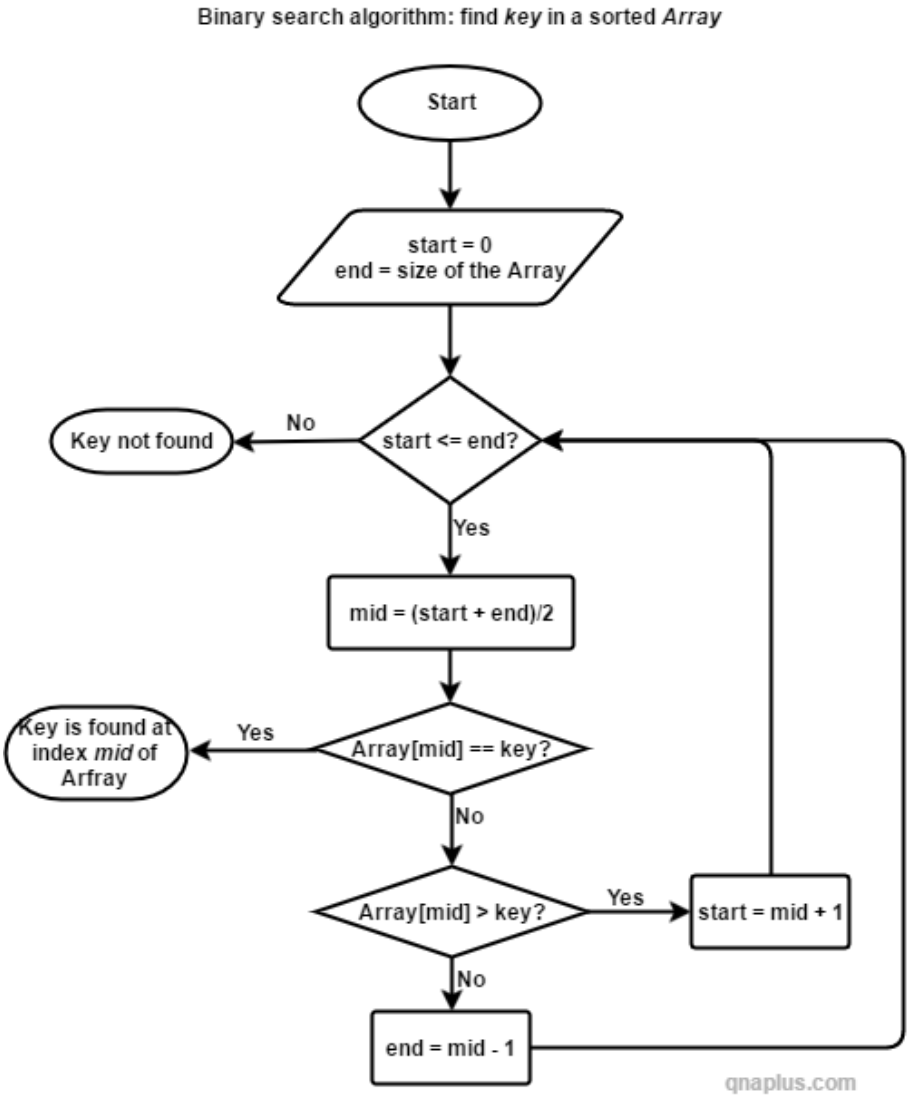
```java
public class LinearSearch {
	public static void main(String[] args) {
		int[] arr = { 10, 20, 30, 40, 50 };
		boolean result = linearSearch(arr, 60);
		System.out.println(result);
	}

	static boolean linearSearch(int[] arr, int x) {
		for (int i = 0; i < arr.length; i++) {
			if (arr[i] == x) {
				return true;
			}
		}
		return false;
	}
}
```

TC: - O(n)

SC: - O(1)

Binary Search: -



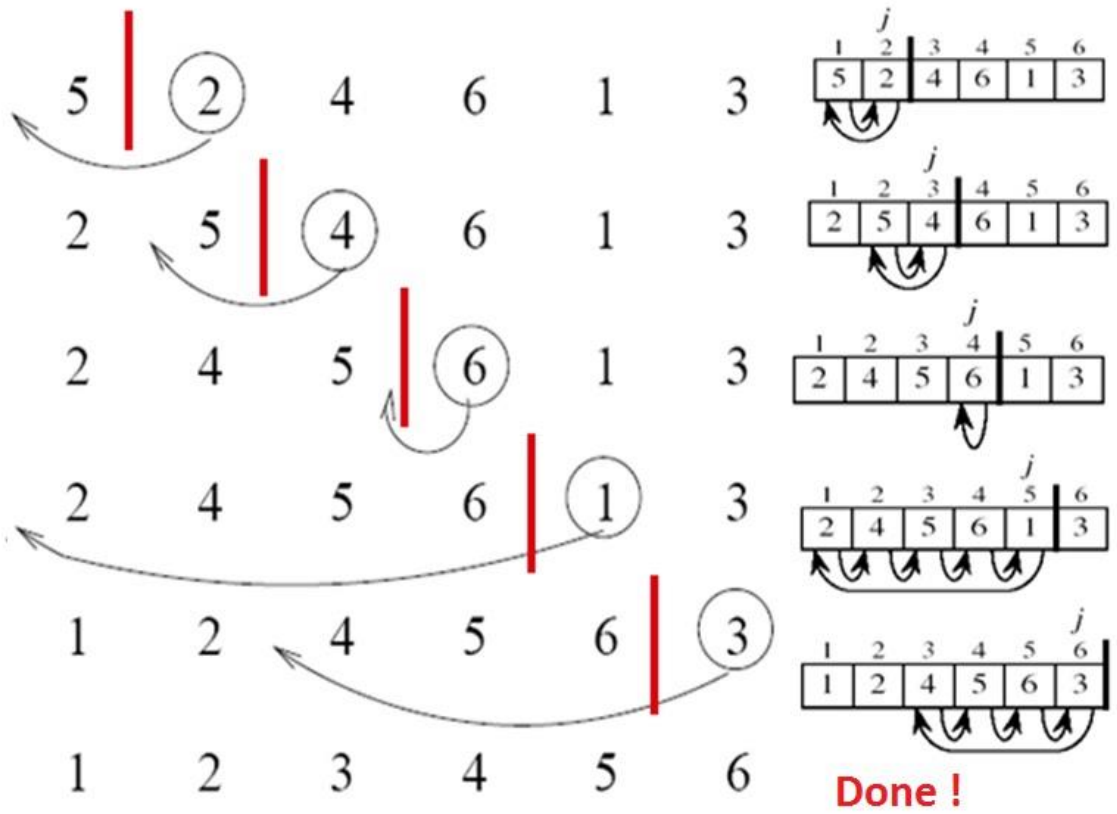



```java
public class BinarySearch {
	public static void main(String[] args) {
		int[] arr = { 10, 20, 30, 40, 50 };
		boolean result = binarySearch(arr, 50, 0, arr.length - 1);
		System.out.println(result);
	}

	private static boolean binarySearch(int[] arr, int x, int start, int end) {
		while (start <= end) {
			int mid = (start + end) / 2;
			if (arr[mid] == x) {
				return true;
			} else if (arr[mid] < x) {
				start = mid + 1;
			} else if (arr[mid] > x) {
				end = mid - 1;
			}
		}
		return false;
	}
}
```

Karthik Chinni

```
public class SortUtil {
	public static int[] swap(int[] arr, int index1, int index2) {
		int temp = arr[index1];
		arr[index1] = arr[index2];
		arr[index2] = temp;
		return arr;
	}
}
public class SortTester {
	public static void main(String[] args) {
		int[] inputDataArr = { 6, 4, 2, 1, 3, 5 };
		InsertionSortImpl.insertionSort(inputDataArr);
		for (int i = 0; i < inputDataArr.length; i++) {
			System.out.print(inputDataArr[i] + "  ");
		}
	}
}
```

Insertion Sort: -



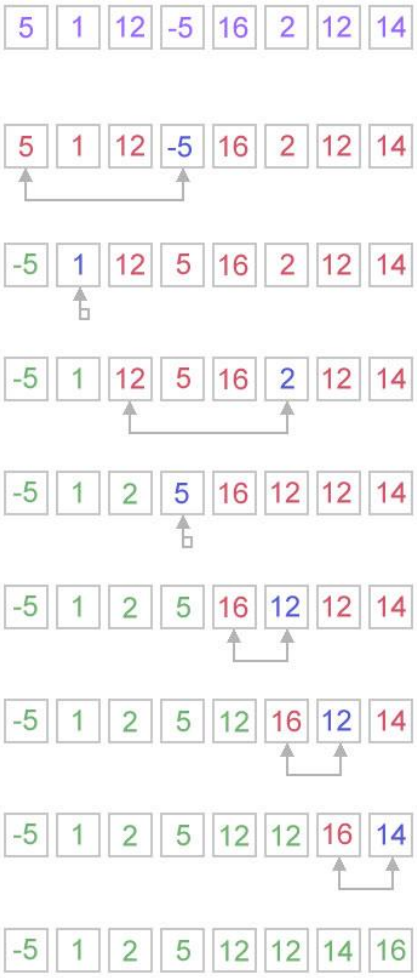
Karthik Chinni

```java
public class InsertionSortImpl {
	public static void insertionSort(int[] inputDataArr) {
		int length = inputDataArr.length;
		int temp;
		// i=>for cards iteration of cards till end
		// j=>where to start?
		for (int i = 1; i < length; i++) {
			for (int j = i; j > 0; j--) {
				if (inputDataArr[j] < inputDataArr[j - 1]) {
					SortUtil.swap(inputDataArr, j, j-1);
				}
			}
		}
	}
}
```

TC: - O(n)

SC: - O(1)

Selection Sort: -



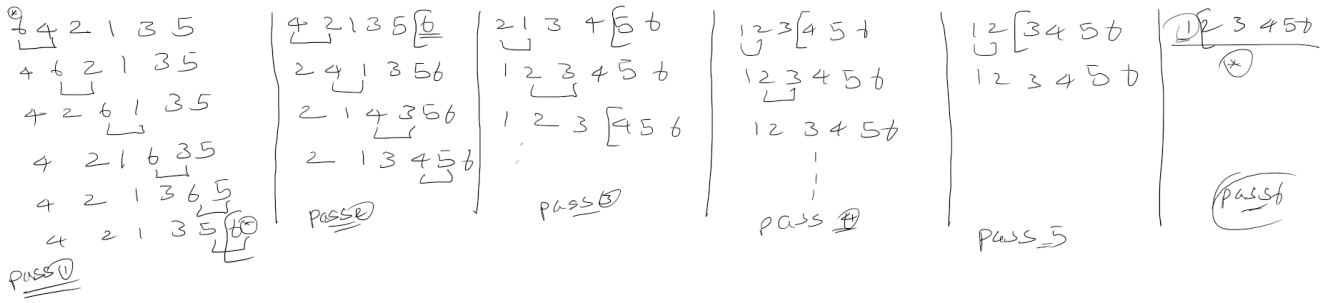
Karthik Chinni

```java
public class SelectionSortImpl {
	public static void selectionSort(int[] inputDataArr) {
		int length = inputDataArr.length;
		int smallestIndex;
		for (int i = 0; i < length; i++) {
			smallestIndex = i;
			for (int j = i + 1; j < length; j++) {
				if (inputDataArr[smallestIndex] > inputDataArr[j]) {
					smallestIndex = j;
				}
			}
			SortUtil.swap(inputDataArr, i, smallestIndex);
		}
	}
}
```

TC: - O(n)

SC: - O(1)

Bubble Sort: -



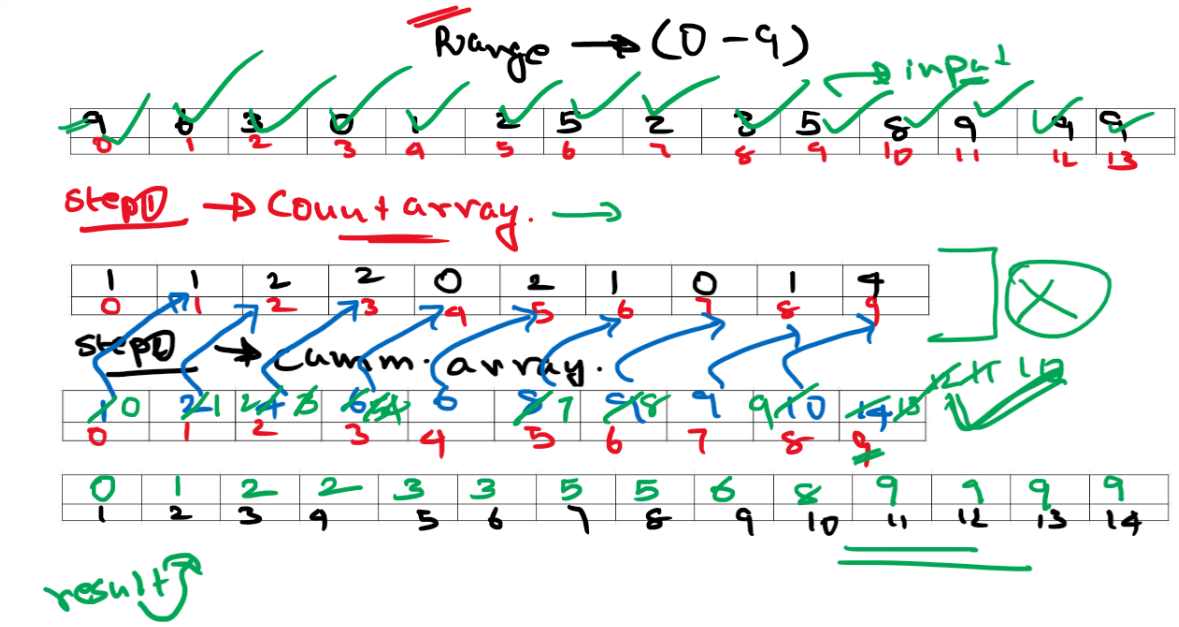
```java
public class BubbleSortImpl {
	static int[] bubbleSort(int[] inputDataArr) {
		for (int i = 0; i < inputDataArr.length - 1; i++) {
			for (int j = 0; j < inputDataArr.length - i - 1; j++) {
				if (inputDataArr[j] > inputDataArr[j + 1]) {
					SortUtil.swap(inputDataArr, j, j + 1);
				}
			}
		}
		return inputDataArr;
	}
}
```

TC: - O(n)

SC: - O(1)

Karthik Chinni

Radix Sort: -



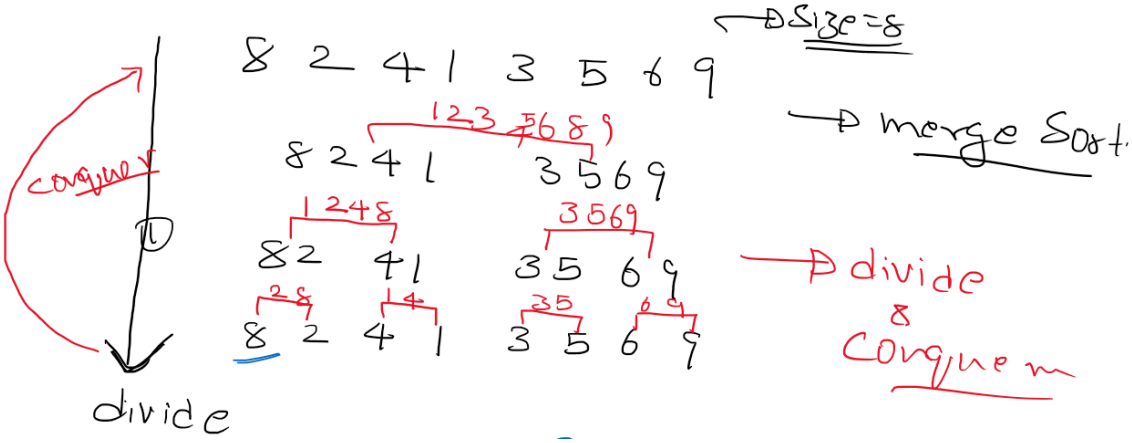

```
public class CountingSort {
	public static void main(String[] args) {
		int[] arr = { 9, 6, 3, 0, 1, 2, 5, 2, 3, 5, 8, 9, 9, 9 };
		System.out.println("Before Sorting!");
		for (int i = 0; i < arr.length; i++) {
			System.out.print(arr[i] + "  ");
		}
		int[] countArray = new int[arr.length];
		int[] cumulativeArray = new int[arr.length
		int[] result = new int[arr.length];

		// Filling countArray
		for (int i = 0; i < arr.length; i++) {
			countArray[arr[i]]++;
		}

		// Filling cumulativeArray
		cumulativeArray[0] = countArray[0];
		for (int i = 1; i < countArray.length; i++) {
			cumulativeArray[i] = cumulativeArray[i - 1] + countArray[i];
		}

		// Finding where every element in input array should be placed
		for (int i = 0; i < arr.length; i++) {
			result[cumulativeArray[arr[i]] - 1] = arr[i];
			cumulativeArray[arr[i]]--;
		}

		// Printing result array
		System.out.println("\n"+"After Sorting!");
		for (int i = 0; i < result.length; i++) {
			System.out.print(result[i] + "  ");
		}
	}
}
```

TC: - O(n+r) → n: input size; r: range size

SC: - O(n+r) → n: input size; r: range size

Karthik Chinni

Merge Sort: -



Need to understand the below two to understand Merge sort: -

1) Recursion
2) Merge two sorted arrays

```
public class MergingTwoSortedArraysVerbose {
	static int[] merge(int[] arr1, int[] arr2) {
		int[] result = new int[arr1.length + arr2.length];
		// Base case 1
		if (arr1.length == 0 && arr2.length == 0) {
			return result;
		}
		// Base case 2
		if (arr1.length == 0) {
			return arr2;
		}
		// Base case 3
		if (arr2.length == 0) {
			return arr1;
		}
		int i, j, k;
		i = j = k = 0;
		while (i < arr1.length && j < arr2.length) {
			if (arr1[i] < arr2[j]) {
				result[k] = arr1[i];
				i++;
				k++;
			} else if (arr1[i] > arr2[j]) {
				result[k] = arr2[j];
				j++;
				k++;
			}
		}
```

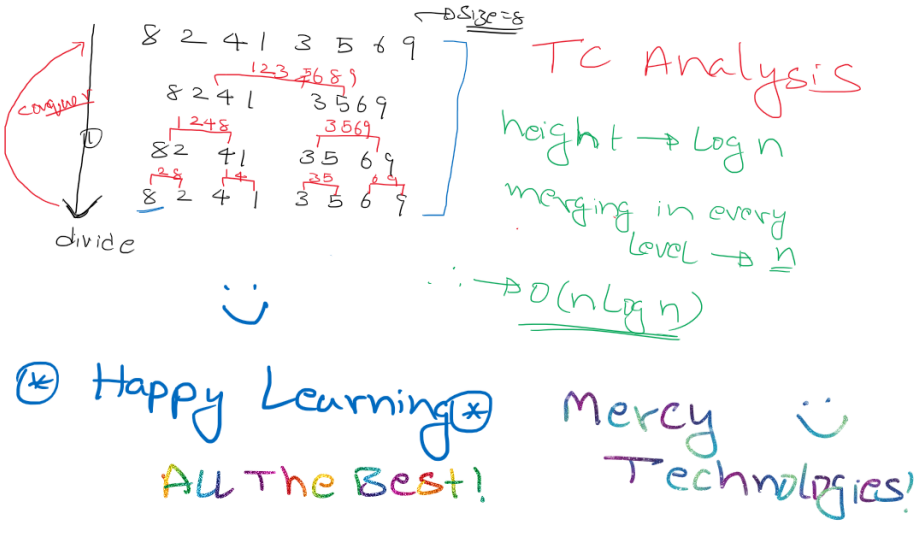
Karthik Chinni

```
		while (i < arr1.length) {
			result[k] = arr1[i];
			i++;
			k++;
		}
		while (j < arr2.length) {
			result[k] = arr2[j];
			j++;
			k++;
		}
		return result;
	}

	public static void main(String[] args) {
		int[] arr1 = { 1, 3, 5, 7, 9, 20, 30, 40 };
		int[] arr2 = { 2, 4, 6, 8 };
		int[] result = merge(arr1, arr2);
		for (int i = 0; i < result.length; i++) {
			System.out.print(result[i] + " ");
		}
	}
}
```

Better way to code: -

```
public class MergingTwoSortedArrays {
	public static void main(String[] args) {
		int[] arr1 = { 1, 3, 5, 7 };
		int[] arr2 = { 2, 4, 6, 8 };
		int[] result = new int[arr1.length + arr2.length];
		int i, j, k;
		i = j = k = 0;

		while (i < arr1.length && j < arr2.length) {
			result[k++] = arr1[i] < arr2[j] ? arr1[i++] : arr2[j++];
		}
		while (i < arr1.length) {
			result[k++] = arr1[i++];
		}
		while (j < arr2.length) {
			result[k++] = arr2[j++];
		}

		for (int k2 = 0; k2 < result.length; k2++) {
			System.out.print(result[k2] + "  ");
		}
	}

}
```

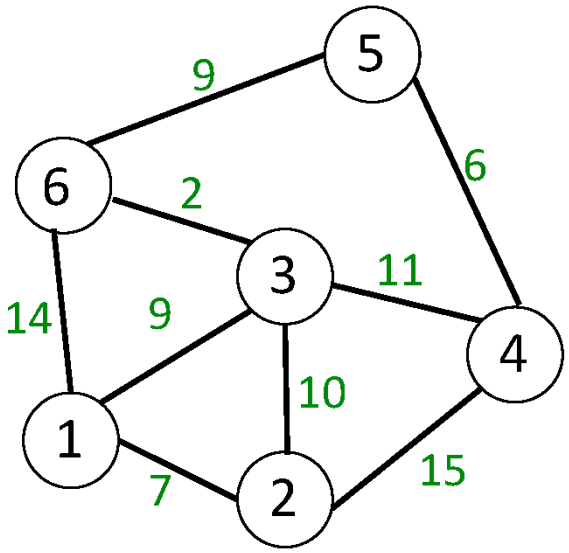
```java
public class MergeSort {
	public static void main(String[] args) {
		int[] arr = { 8, 2, 4, 1, 3, 5, 6, 9 };
		int[] helper = new int[arr.length];

		mergeSortHelper(arr, helper, 0, arr.length - 1);
		for (int i = 0; i < arr.length; i++) {
			System.out.print(arr[i] + "  ");
		}
	}

	static void mergeSortHelper(int[] arr, int[] helper, int low, int high) {
		// Base condition
		if (low >= high) {
			return;
		}
		int mid = (low + high) / 2;
		mergeSortHelper(arr, helper, low, mid);
		mergeSortHelper(arr, helper, mid + 1, high);
		merge(arr, helper, low, mid, high);
	}

	static void merge(int[] arr, int[] helper, int low, int mid, int high) {
		// Copy all elements into the helper array for this iteration
		for (int k = low; k <= high; k++) {
			helper[k] = arr[k];
		}
		// This is the only additional assignment you should do and nothing else.
		int i = low;
		int j = mid + 1;
		int k = low;

		while (i <= mid && j <= high) {
			arr[k++] = helper[i] < helper[j] ? helper[i++] : helper[j++];
		}

		while (i <= mid) {
			arr[k++] = helper[i++];
		}
		while (j <= high) {
			arr[k++] = helper[j++];
		}
	}
}
```



Karthik Chinni

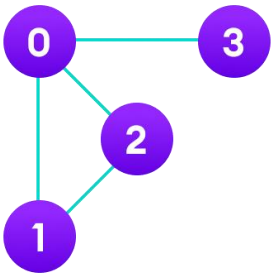
A **graph** is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices.



More precisely, a graph is a data structure (V, E) that consists of

- A collection of vertices V
- A collection of edges E, represented as ordered pairs of vertices (u,v)



Vertices and edges in the graph,

V = {0, 1, 2, 3}

E = {(0,1), (0,2), (0,3), (1,2)}

G = {V, E}

Karthik Chinni

Graph Terminology

- **Adjacency**: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
- **Path**: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
- **Directed Graph**: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge

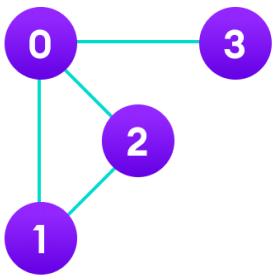
Graph Representation

Graphs are commonly represented in two ways:

1. Adjacency Matrix

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.

If the value of any element `a[i][j]` is 1, it represents that there is an edge connecting vertex i and vertex j.
The adjacency matrix for the graph we created above is



	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Karthik Chinni

Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

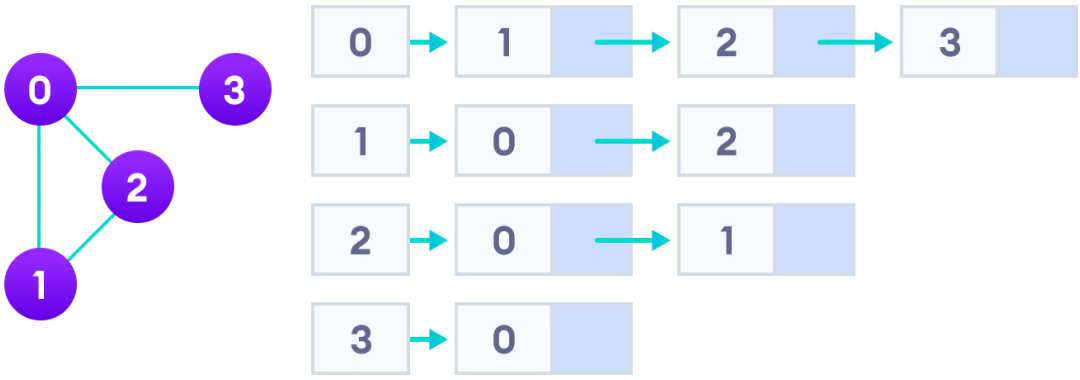
Edge lookup(checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices(V x V), so it requires more space.

2. Adjacency List

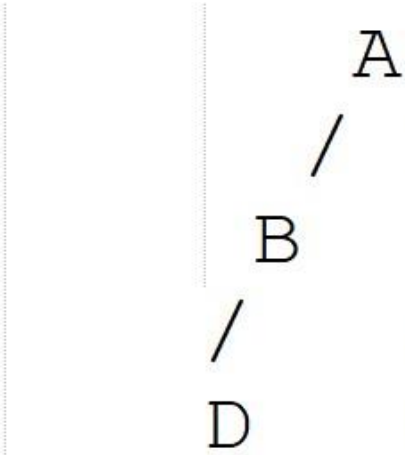
An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

The adjacency list for the graph we made in the first example is as follows:



An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.



DFS=>A, B, D, C, E, F

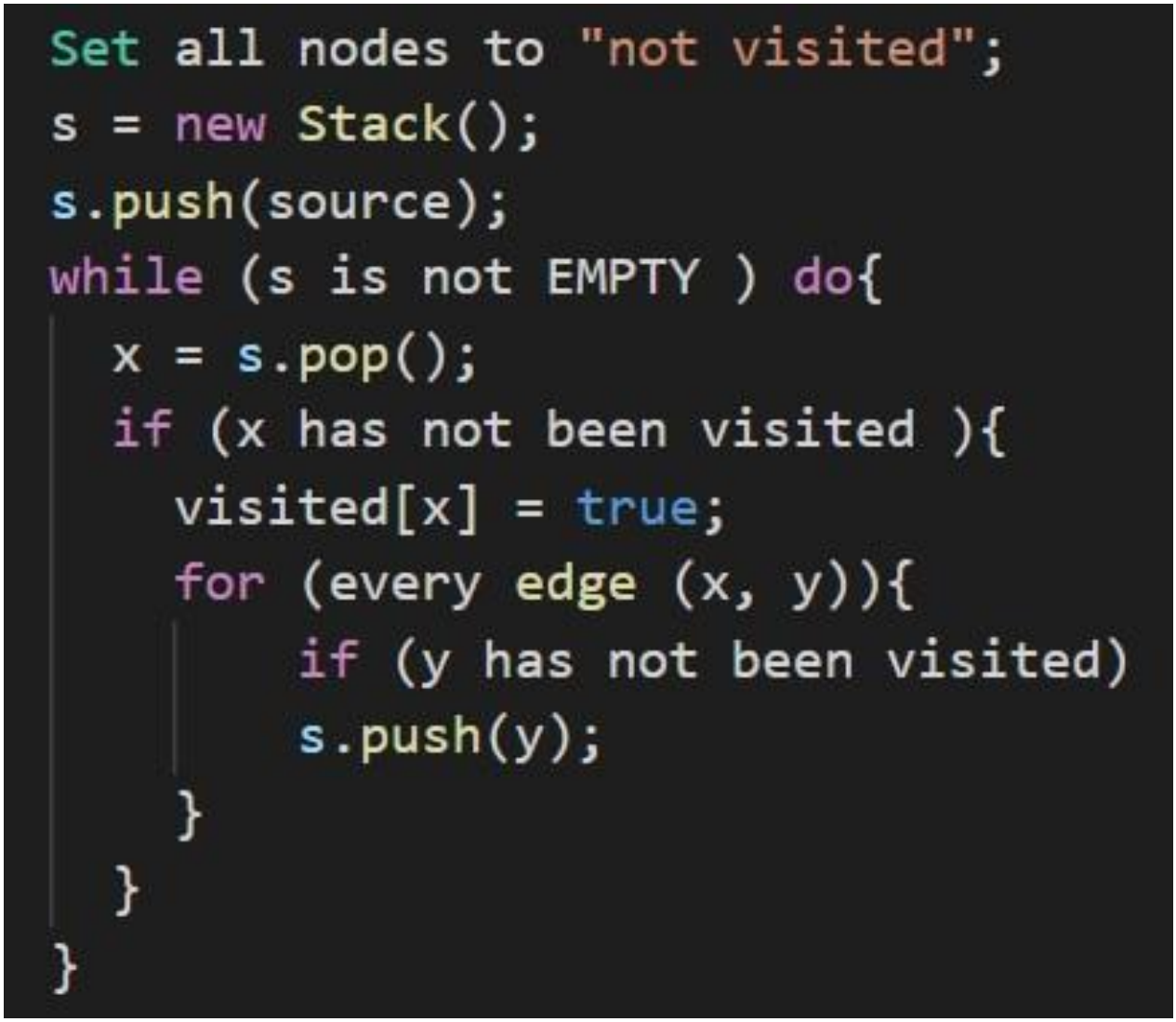
BFS=>A, B, C, D, E, F

```
Set all nodes to "not visited";
q = new Queue();
q.enqueue(initial node);
while(q not empty) do{
   x = q.dequeue();
   if( x has not been visited ){
     visited[x] = true;
     for (every edge (x, y)){
         if (y has not been visited)
         q.enqueue(y);
     }
   }
}
```

```
public class MyGraph {
	private int numVertices;
	private LinkedList<Integer> adjLists[];
	private boolean visited[];

	MyGraph(int vertices) {
		numVertices = vertices;
		adjLists = new LinkedList[vertices];
		visited = new boolean[vertices];
		for (int i = 0; i < vertices; i++)
			adjLists[i] = new LinkedList<Integer>();
	}

	void addEdge(int source, int destination) {
		adjLists[source].add(destination);
	}

	public int getNumVertices() {
		return numVertices;
	}

	public void setNumVertices(int numVertices) {
		this.numVertices = numVertices;
	}

	public LinkedList<Integer>[] getAdjLists() {
		return adjLists;
	}

	public void setAdjLists(LinkedList<Integer>[] adjLists) {
		this.adjLists = adjLists;
	}

	public boolean[] getVisited() {
		return visited;
	}

	public void setVisited(boolean[] visited) {
		this.visited = visited;
	}
}
public class MyBFS {
	static void doBFS(MyGraph g, int source) {
		boolean[] visitedArray = g.getVisited();
		// Create a new queue for BFS
		LinkedList<Integer> queue = new LinkedList<Integer>();
		// Mark the current node as visited and enqueue it
		visitedArray[source] = true;
		queue.add(source);
		while (queue.size() > 0) {
			// pop a vertex from queue and print it
			source = queue.poll();
			System.out.print(source + " ");
			// Traverse all adj vertices,if not visited, mark visited and enqueue
			Iterator<Integer> it = g.getAdjLists()[source].listIterator();
			while (it.hasNext()) {
				int curr = it.next();
				if (!visitedArray[curr]) {
					visitedArray[curr] = true;
					queue.add(curr);
				}
			}
		}
	}
```

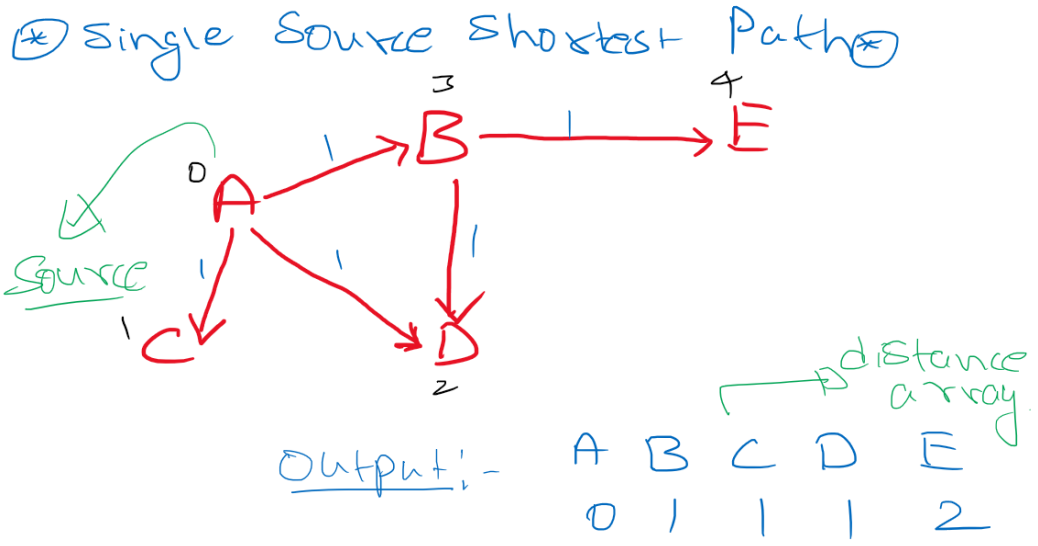
Karthik Chinni

```
        public static void main(String args[]) {
                MyGraph g = new MyGraph(6);
                g.addEdge(0, 1);
                g.addEdge(0, 2);
                g.addEdge(1, 3);
                g.addEdge(2, 4);
                g.addEdge(2, 5);
                System.out.println("Following is Breadth First Traversal");
                doBFS(g, 0);
        }
}
```



```
public class MyDFS {
	static void doDFS(MyGraph g, int source) {
		boolean[] visitedArray = g.getVisited();
		visitedArray[source] = true;
		System.out.print(source + " ");
		Iterator itr = g.getAdjLists()[source].listIterator();
		while (itr.hasNext()) {
			int adj_node = (int) itr.next();
			if (!visitedArray[adj_node])
				doDFS(g, adj_node);
		}
	}

	public static void main(String args[]) {
		MyGraph g = new MyGraph(6);
		g.addEdge(0, 1);
		g.addEdge(0, 2);
		g.addEdge(1, 3);
		g.addEdge(2, 4);
		g.addEdge(2, 5);
		System.out.println("Following is Depth First Traversal");
		doDFS(g, 0);
	}
}
```

Single Source Shortest Path problem for unweighted graphs: -

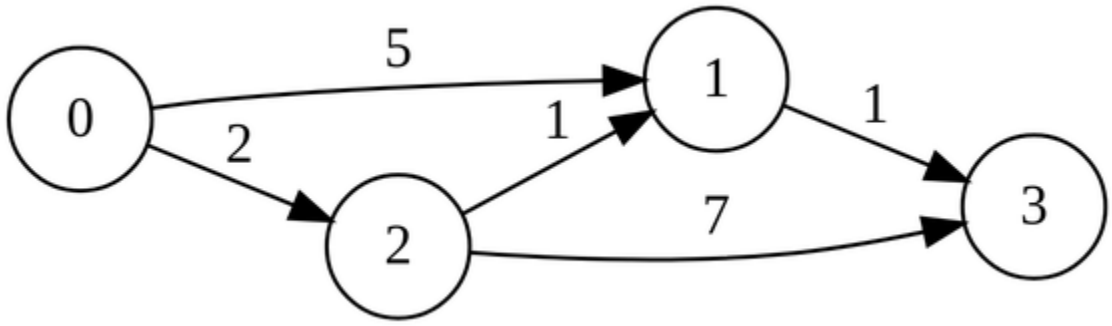


```
public class MyGraphWithDistance extends MyGraph {
	private int distance[];

	MyGraphWithDistance(int vertices) {
		super(vertices);
		distance = new int[vertices];
		for (int i = 0; i < distance.length; i++) {
			distance[i] = -1;
		}
	}

	public int[] getDistance() {
		return distance;
	}

	public void setDistance(int[] distance) {
		this.distance = distance;
	}
}
public class MySSSP {
	static void getSSSP(MyGraphWithDistance g, int source) {
		boolean[] visitedArray = g.getVisited();
		int[] distanceArray = g.getDistance();
		LinkedList<Integer> queue = new LinkedList<Integer>();
		visitedArray[source] = true;
		// Mark distance of source to be 0
		distanceArray[source] = 0;
		queue.add(source);
		while (queue.size() > 0) {
			// pop a vertex from queue and print it
			source = queue.poll();
			System.out.print(source + " ");
			// Traverse all adj vertices,if not visited, mark visited and enqueue
			Iterator<Integer> it = g.getAdjLists()[source].listIterator();
			while (it.hasNext()) {
				int curr = it.next();
				if (!visitedArray[curr]) {
					visitedArray[curr] = true;
					distanceArray[curr] = distanceArray[source] + 1;
					queue.add(curr);
				}
			}
		}
	}
	private static void printShortestPath(MyGraphWithDistance g) {
		System.out.println("\nFollowing is Shortest path using BFS");
		int[] distanceArray = g.getDistance();
		for (int i = 0; i < distanceArray.length; i++) {
			System.out.print(distanceArray[i] + " ");
		}
	}

	public static void main(String args[]) {
		MyGraphWithDistance g = new MyGraphWithDistance(5);
		g.addEdge(0, 1);
		g.addEdge(0, 2);
		g.addEdge(0, 3);
		g.addEdge(3, 2);
		g.addEdge(3, 4);
		System.out.println("Following is Breadth First Traversal");
		getSSSP(g, 0);
		printShortestPath(g);
	}
}
```

Karthik Chinni

Djkstra's Algorithm: -

Used to find the shortest path for weighted graphs.




```
public class MyWeightedGraph {
	private int numVertices;
	private LinkedList<GraphNode> adjLists[];
	private Integer distance[];

	MyWeightedGraph(int vertices) {
		numVertices = vertices;
		adjLists = new LinkedList[vertices];
		distance = new Integer[vertices];
		for (int i = 0; i < vertices; i++) {
			adjLists[i] = new LinkedList<GraphNode>();
			distance[i] = Integer.MAX_VALUE;
		}
	}

	void addEdge(int source, int destination, int weight) {
		adjLists[source].add(new GraphNode(source, destination, weight));
	}

	public int getNumVertices() {
		return numVertices;
	}

	public void setNumVertices(int numVertices) {
		this.numVertices = numVertices;
	}

	public LinkedList<GraphNode>[] getAdjLists() {
		return adjLists;
	}

	public void setAdjLists(LinkedList<GraphNode>[] adjLists) {
		this.adjLists = adjLists;
	}

	public Integer[] getDistance() {
		return distance;
	}

	public void setDistance(Integer[] distance) {
		this.distance = distance;
	}

	@Override
	public String toString() {
		return "MyWeightedGraph [numVertices=" + numVertices + ", adjLists=" +
Arrays.toString(adjLists) + ", distance="
```

Karthik Chinni

```
                        + Arrays.toString(distance) + "]";
        }
}

public class MyDijkstraImpl {
        static void getSSSPUsingDijkstraAlgo(MyWeightedGraph g, int source) {
                Integer[] distanceArray = g.getDistance();
                // Create a new PQ
                PriorityQueue<Integer> pq = new PriorityQueue<>();
                // Mark distance of source to be 0
                distanceArray[source] = 0;
                pq.add(source);
                while (pq.size() > 0) {
                        // pop a vertex from queue and print it
                        source = pq.poll();
                        System.out.print(source + " ");
                        // Traverse all adj vertices,if not visited, mark visited and enqueue
                        Iterator<GraphNode> it = g.getAdjLists()[source].listIterator();
                        while (it.hasNext()) {
                                GraphNode curr = it.next();
                                if (distanceArray[curr.getDestination()] >
distanceArray[source] + curr.getWeight()) {
                                        distanceArray[curr.getDestination()] =
distanceArray[source] + curr.getWeight();
                                        pq.add(curr.getDestination());
                                }
                        }
                }
        }

        private static void printShortestPath(MyWeightedGraph g) {
                System.out.println("\nFollowing is Shortest path using Dijkstra's
Algorithm");
                Integer[] distanceArray = g.getDistance();
                for (int i = 0; i < distanceArray.length; i++) {
                        System.out.print(distanceArray[i] + " ");
                }
        }

        public static void main(String args[]) {
                MyWeightedGraph g = new MyWeightedGraph(4);
                g.addEdge(0, 1, 5);
                g.addEdge(0, 2, 2);
                g.addEdge(1, 3, 1);
                g.addEdge(2, 1, 1);
                g.addEdge(2, 3, 7);
                System.out.println("Following is the traversal using Dijkstra's Algorithm");
                getSSSPUsingDijkstraAlgo(g, 0);
                printShortestPath(g);
        }
}
```

```java
public class GraphNode {
	int source;
	int destination;
	Integer weight;

	public GraphNode(int source, int destination, Integer weight) {
		this.source = source;
		this.destination = destination;
		this.weight = weight;
	}

	public int getSource() {
		return source;
	}

	public void setSource(int source) {
		this.source = source;
	}

	public int getDestination() {
		return destination;
	}

	public void setDestination(int destination) {
		this.destination = destination;
	}

	public Integer getWeight() {
		return weight;
	}

	public void setWeight(Integer weight) {
		this.weight = weight;
	}

	@Override
	public String toString() {
		return "Edge [source=" + source + ", destination=" + destination + ",
weight=" + weight + "]";
	}
}
```

We have 10 identical bottles of identical pills (each bottle contains 100 of pills). Out of 10 bottles 9 have 1 gram of pills but 1 bottle has pills of weight of 1.1 gram. Given a measurement scale, how would you find the heavy bottle? You can use the scale only once.

Answer:

First, arrange the bottles on shelf and now take, 1 pill from the first bottle, 2 pills from the second bottle, 3 pills from the third bottle, and so on. Ideally you would have **(10)*(11)/2=55** pills weighing 55 grams, when you put the entire pile of pills on the weighing scale. The deviation from 55 g would tell you which bottle contains the heavy pills.

If it is .1 gram more, it is 1st bottle which has heavy pill, if it is .2 more, gram 2nd bottle has heavy pills, if it is .3 more, gram 3rd bottle has heavy pills.

How to measure exactly 4 gallons of water from 3 gallon and 5-gallon jars, Given, you have unlimited water supply from a running tap.

Solution:

Step 1. Fill 3-gallon jar with water. (5p – 0, 3p – 3)

Step 2. Pour all its water into 5-gallon jar. (5p – 3, 3p – 0)

Step 3. Fill 3-gallon jar again. (5p – 3, 3p – 3)

Step 4. Pour its water into 5-gallon jar until it is full. Now you will have exactly 1-gallon water remaining in 3-gallon jar. (5p – 5, 3p – 1)

Step 5. Empty 5-gallon jar, pour 1-gallon water from 3 gallon jar into it. Now 5-gallon jar has exactly 1 gallon of water. (5p – 1, 3p – 0)

Step 6. Fill 3-gallon jar again and pour all its water into 5-gallon jar**,** thus 5-gallon jar will have exactly 4 gallons of water. (5p – 4, 3p – 0)

Karthik Chinni

You've got someone working for you for seven days and a gold bar to pay him. The gold bar is segmented into seven connected pieces. You must give them a piece of gold at the end of every day. What and where are the fewest number of cuts to the bar of gold that will allow you to pay him 1/7th each day?

Puzzle Solution:

Lets split the chain as,

Day 1: Give A (+1)

Day 2: Get back A, give B (-1, +2)

Day 3: Give A (+1)

Day 4:Get back A and B, give C (-2,-1,+4)

Day 5:Give A (+1)

Day 6:Get back A, give B (-1,+2)

Day 7:Give A (+1)

Puzzle: Four people need to cross a rickety bridge at night. Unfortunately, they have only one torch and the bridge is too dangerous to cross without one. The bridge is only strong enough to support two people at a time. Not all people take the same time to cross the bridge. **Times for each person: 1 min, 2 mins, 7 mins and 10 mins.** What is the shortest time needed for all four

It is 17 mins.

1 and 2 go first, then 1 comes back. Then 7 and 10 go and 2 comes back. Then 1 and 2 go again, it makes a total of 17 minutes.

Karthik Chinni

Recursion

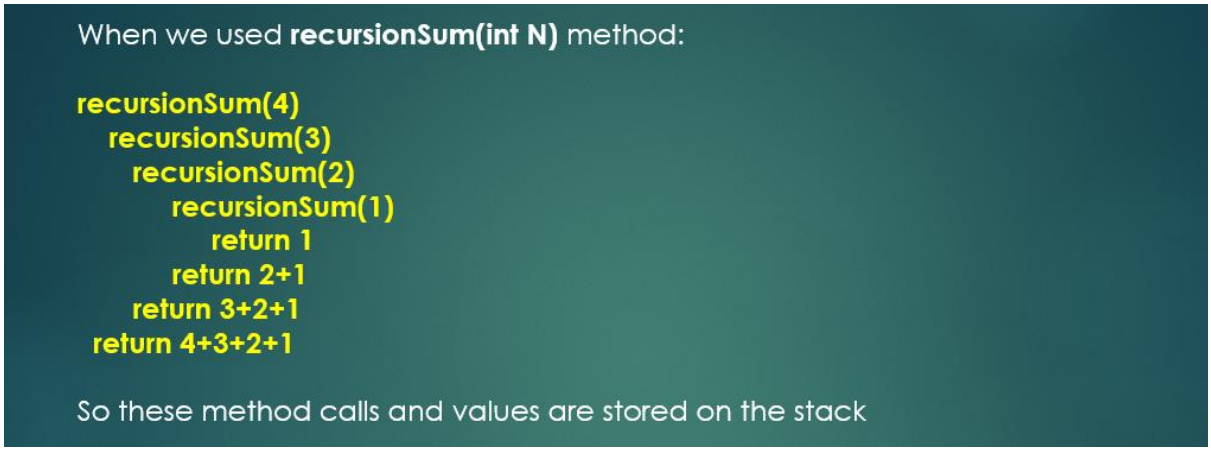
- A method/procedure where the solution to a problem depends on solutions to smaller instances of the same problem
- So we break the task into smaller subtasks
- The approach can be applied to many types of problems and recursion is one of the central ideas of computer science
- We have to define base cases in order to avoid infinite loops
- We can solve problems with recursion or with iteration

Head VS tail recursion

- If the recursive call occurs at the end of a method → it is called a tail recursion
- The tail recursion is similar to a loop
- The method executes all the statements before jumping into the next recursive call
- If the recursive call occurs at the beginning of a method, it is called a head recursion.
- The method saves the state before jumping into the next recursive call

```
public class RecursionExample {
    public static void main(String[] args) {
        int result = recursionSum(3);
        System.out.println(result);
    }

    private static int recursionSum(int n) {
        // Base case
        if (n == 1) {
            return 1;
        }
        return n + recursionSum(n - 1);
    }
}
```

Karthik Chinni

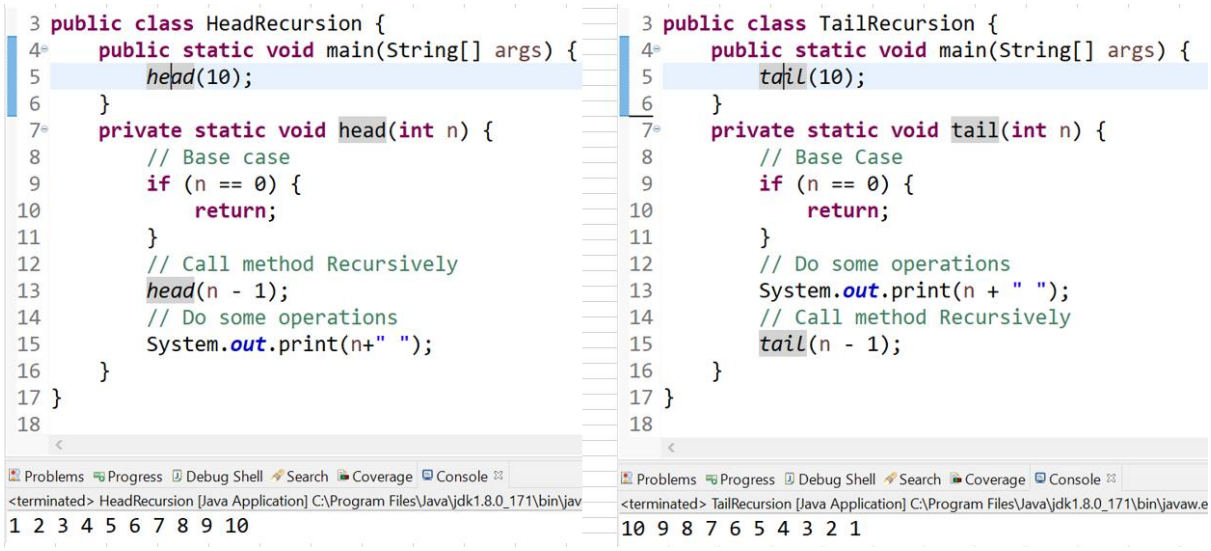


Comparing recursive implementation against iterative implementation → **recursion is at least twice slower because first we unfold recursive calls (pushing them on a stack) until we reach the base case and then we traverse the stack and retrieve all recursive calls**

Recursion vs Iterative example:

```
public class SumOfNumbers {
     static int result = 0;

     public static int sumUsingIterative(int n) {
          for (int i = 1; i <= n; i++) {
               result += i;
          }
          return result;
     }

     public static int sumUsingRecursion(int n) {
          if (n == 1) {
               return 1;
          }
          return n + sumUsingRecursion(n - 1);
     }
}
```

In case of Head Recursion, we need to save state and need to backtrack, while in Tail Recursion, we need not save the state. This is just similar to a loop in java.



```java
public class HeadRecursion {
    public static void main(String[] args) {
        head(10);
    }
    private static void head(int n) {
        // Base case
        if (n == 0) {
            return;
        }
        // Call method Recursively
        head(n - 1);
        // Do some operations
        System.out.print(n+" ");
    }
}
```

Problems Progress Debug Shell Search Coverage Console

<terminated> HeadRecursion [Java Application] C:\Program Files\Java\jdk1.8.0_171\bin\jav

```
1 2 3 4 5 6 7 8 9 10
```

```java
public class TailRecursion {
    public static void main(String[] args) {
        tail(10);
    }
    private static void tail(int n) {
        // Base Case
        if (n == 0) {
            return;
        }
        // Do some operations
        System.out.print(n + " ");
        // Call method Recursively
        tail(n - 1);
    }
}
```

Problems Progress Debug Shell Search Coverage Console

<terminated> TailRecursion [Java Application] C:\Program Files\Java\jdk1.8.0_171\bin\javaw.e

```
10 9 8 7 6 5 4 3 2 1
```

In head recursion, there is a heavy dependency on the previous function call result. To avoid this, we use a accumulator in tail recursion, so that the dependency on the previous function call result is avoided.

```java
public class FactorialHeadRecursion {
	public static int factorial(int n) {
		// Base Case
		if (n == 1) {
			return 1;
		}
		// Do some operations
		int res = factorial(n - 1);
		int result = n * res;
		return result;
	}
	public static void main(String[] args) {
		System.out.println(factorial(5));
	}

}
```

```java
public class FactorialTailRecursion {
	public static int factorial(int n, int accumulator) {
		// Base Case
		if (n == 1) {
			return accumulator;
		}
		return factorial(n - 1, accumulator * n);
	}

	public static void main(String[] args) {
		System.out.println(factorial(5, 1));
	}

}
```

Karthik Chinni

```java
public class FibHeadRecursion {
	public static void main(String[] args) {
		System.out.println(fibHead(10));
	}

	private static int fibHead(int n) {
		// Base Case
		if (n == 0 || n == 1) {
			return n;
		}

		// Make recursive calls
		int fib1 = fibHead(n - 1);
		int fib2 = fibHead(n - 2);
		// Make some operations
		int result = fib1 + fib2;
		return result;
	}
}
```

In tail recursion, we need to use two accumulators as we have two recursive calls. Initial values for the two accumulators should be 0 and 1 respectively.

```java
public class FibTailRecursion {
	public static void main(String[] args) {
		System.out.println(fibTail(10, 0, 1));
	}

	private static int fibTail(int n, int acc1, int acc2) {
		// Base Case
		if (n == 0) {
			return acc1;
		}
		if (n == 1) {
			return acc2;
		}
		return fibTail(n - 1, acc2, acc1 + acc2);
	}
}
```

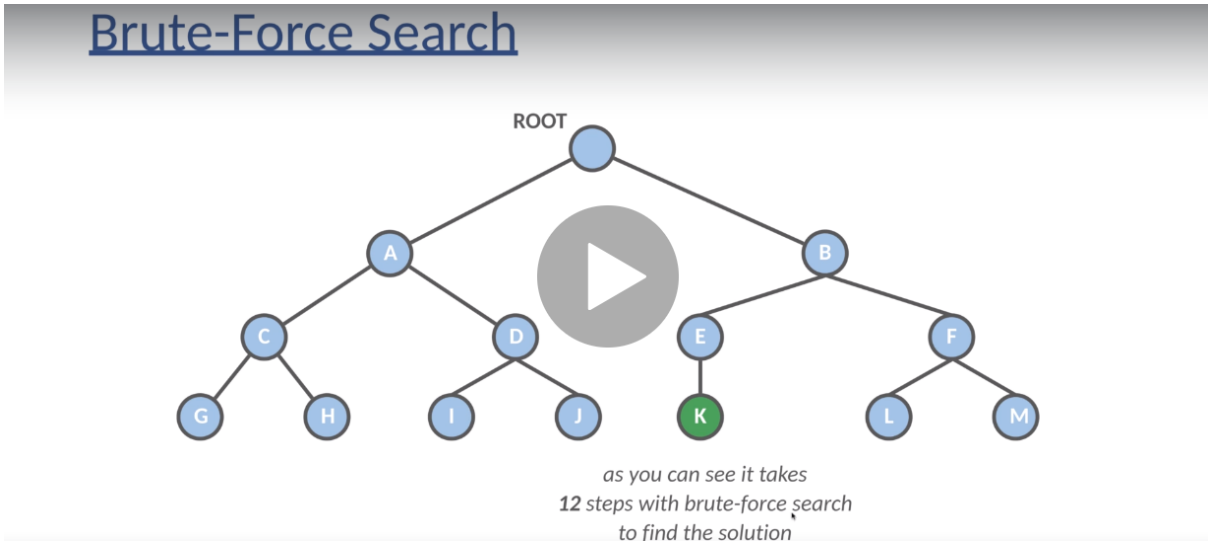
Backtracking

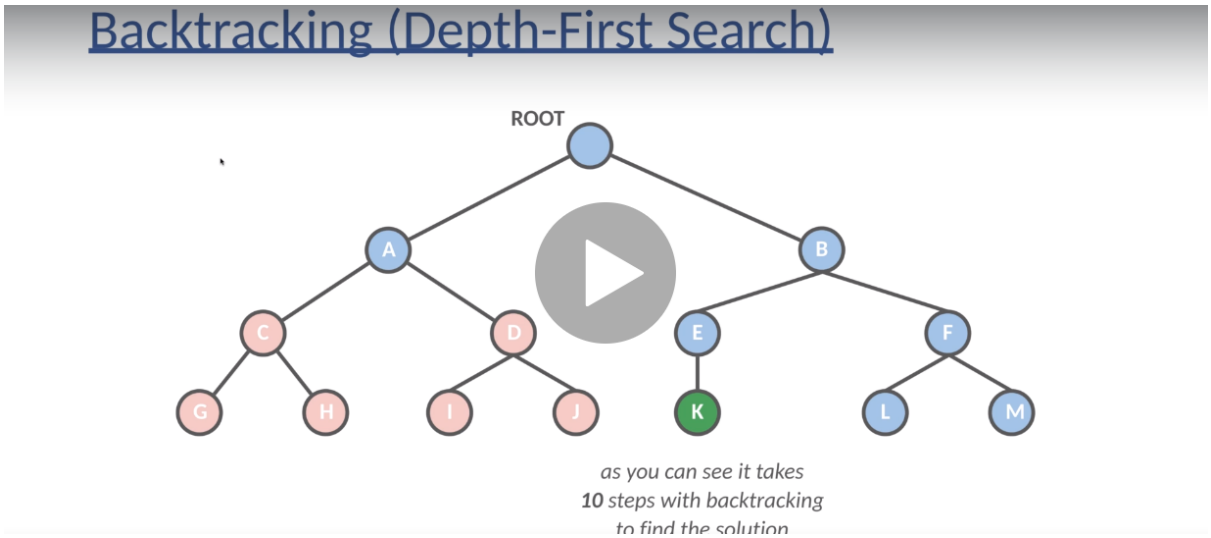
- IT IS A FORM OF RECURSION !!!
- General algorithm for finding all solutions to some computationall problems → ”constraint satisfaction problems”
- We iincrementally build candidates to the solutions
- If partial candidate A cannot be completed to a valid solution: we abandon A as a solution
- For example: eight-queens problem or sudoku
- Backtracking is often much faster than brute force enumeration of all complete candidates, because it can eliminate a large number of candidates with a single test
- Backtracking is an important tool for solving constraint satisfaction problems → combinatorial optimization problems!!!

- The partial candidates are represented as the nodes of a tree structure
- *"**potential search tree**"*
- Each partial candidate is the parent of the candidates that differ from it by a single extension step
- The leaves of the tree are the partial candidates that cannot be extended any further
- The backtracking algorithm traverses this search tree recursively, from the root down (like DFS)

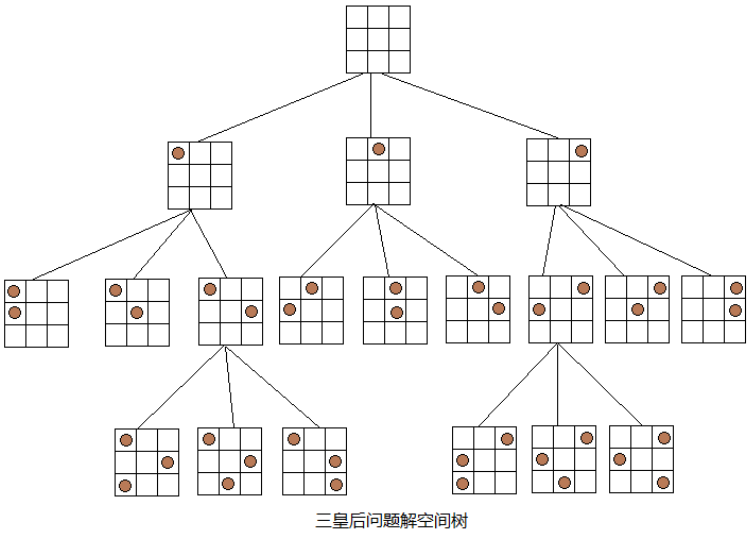
- This is why backtracking is sometimes called depth-first search !!!
  - For every node the algorithm checks whether the given node can be completed to a valid solution
  - If it can not → the whole subtree is skipped !!!
  - Recursively enumerates all subtree of the node

Brute force vs Backtracking:





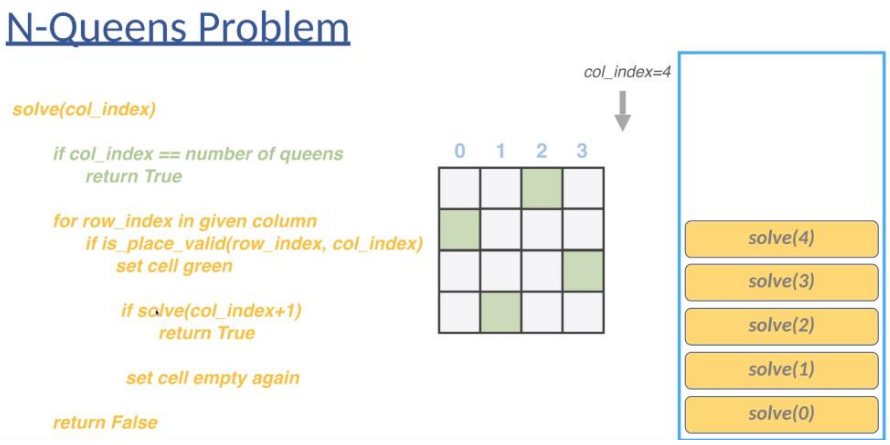
N-Queen's problem with n=3



By Above, we can conclude that there is no valid solution for this, but, we see that many states are discarded by backtracking.

Algorithm:

```
private boolean setQueens(int colIndex) {
	if (colIndex == numOfQueens) {
		return true;
	}
	for (int rowIndex = 0; rowIndex < numOfQueens; ++rowIndex) {
		if (isPlaceValid(rowIndex, colIndex)) {
			chessTable[rowIndex][colIndex] = 1;
			if (setQueens(colIndex + 1)) {
				return true;
			}
			// BACKTRACKING !!!
			chessTable[rowIndex][colIndex] = 0;
		}
	}
	return false;
}
```



Karthik Chinni

```java
public class QueensProblem {

	private int[][] chessTable;
	private int numOfQueens;

	public QueensProblem(int numOfQueens) {
		this.chessTable = new int[numOfQueens][numOfQueens];
		this.numOfQueens = numOfQueens;
	}

	public void solve() {
		if (setQueens(0)) {
			printQueens();
		} else {
			System.out.println("There is no solution...");
		}
	}

	private boolean setQueens(int colIndex) {
		if (colIndex == numOfQueens) {
			return true;
		}
		for (int rowIndex = 0; rowIndex < numOfQueens; ++rowIndex) {
			if (isPlaceValid(rowIndex, colIndex)) {
				chessTable[rowIndex][colIndex] = 1;
				if (setQueens(colIndex + 1)) {
					return true;
				}
				// BACKTRACKING !!!
				chessTable[rowIndex][colIndex] = 0;
			}
		}
		return false;
	}

	private boolean isPlaceValid(int rowIndex, int colIndex) {
		for (int i = 0; i < colIndex; i++)
			if (chessTable[rowIndex][i] == 1)
				return false;
		for (int i = rowIndex, j = colIndex; i >= 0 && j >= 0; i--, j--) {
			if (chessTable[i][j] == 1)
				return false;
		}
		for (int i = rowIndex, j = colIndex; i < chessTable.length && j >= 0; i++, j--) {
			if (chessTable[i][j] == 1)
				return false;
		}
		return true;
	}

	private void printQueens() {
		for (int i = 0; i < chessTable.length; i++) {
			for (int j = 0; j < chessTable.length; j++) {
				if (chessTable[i][j] == 1) {
					System.out.print(" * ");
				} else {
					System.out.print(" - ");
				}
			}
			System.out.println();
		}
	}

	public static void main(String[] args) {
		QueensProblem problem = new QueensProblem(10);
		problem.solve();
	}
}
```

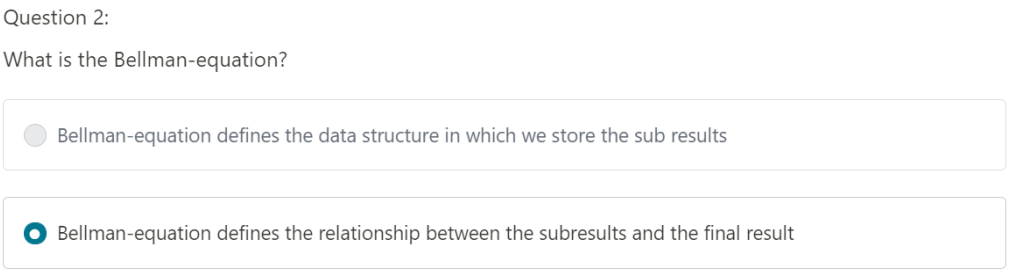
Karthik Chinni

Dynamic Programming

- Dynamic programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems
- It is applicable to problems exhibiting the properties of overlapping subproblems
- The method takes far less time than other methods that don't take advantage of the subproblem overlap
- We need to solve different parts of the problem (subproblems) + combine the solutions of the subproblems to reach an overall solution
- We solve each subproblems only once → we reduce the number of computations
- Subproblems can be stored ('**memoization'**) !!!!

Divide & Conquer

- ✓ Several problems can be solved by combining optimal solutions to ***non-overlapping*** sub-problems
- ✓ This strategy is called "**divide and conquer**" method
- ✓ This is why merge sort / quick sort are not classified as dynamic programming problems
- ✓ **Overlapping subproblems → dynamic programming**
- ✓ **Non-overlapping subproblems → divide and conquer method**



Fibonacci sequence: 0 1 1 2 3 5 8 13 21 34 ...

Fibonacci numbers are defined by the recurrence relation

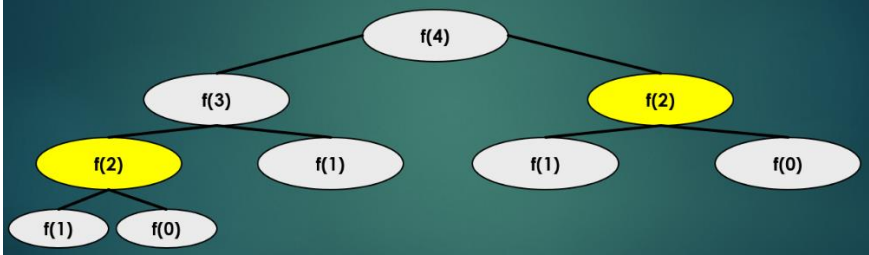
$$F(N) = F(N-1) + F(N-2)$$

$$F(0) = 0 \quad F(1) = 1$$

With generator functions we can get a closed form: „Binet formula"

What is the problem with the recursive formula? We calculate same problems over and over again

f(n) = f(n-1) + f(n-2)



OVERLAPPING SUBPROBLEMS !!!

Fibonacci numbers

- Solution: use dynamic programming and memoization in order to avoid recalculating a subproblem over and over again
- We should use an associative array abstract data type (hashtable) to store the solution for the subproblems // **O(1)** time complexity
- On every **f()** method call → we insert the calculated value if necessary
- Why is it good? Instead of the exponential time complexity we will have **O(N)** time complexity + requires **O(N)** space

Implementation: -

```
public class FibUsingDP {
	public static void main(String[] args) {
		Map<Integer, Integer> table = new HashMap<>();
		table.put(0, 0);
		table.put(1, 1);

		// This has exponential T.C.
		System.out.println(fibRecursion(9));
		// They have linear T.C.
		System.out.println(fibMemoization(9, table));
		System.out.println(fibTabulation(9, table));
	}

	private static int fibRecursion(int n) {
		if (n == 0 || n == 1) {
			return n;
		}
		return fibRecursion(n - 1) + fibRecursion(n - 2);
	}

	// Top-Down Approach (imagine a recursion tree)
	private static int fibMemoization(int n, Map<Integer, Integer> table) {
		if (!table.containsKey(n)) {
			table.put(n, fibMemoization(n - 1, table) + fibMemoization(n - 2, table));
		}
		// O(1) Constant Time Complexity
		return table.get(n);
	}

	// Bottom-Up Approach (imagine a recursion tree - Order is just opposite)
	private static int fibTabulation(int n, Map<Integer, Integer> table) {
		for (int i = 2; i < n; i++) {
			table.put(i, table.get(i - 1) + table.get(i - 2));
		}
		// O(1) Constant Time Complexity
		return table.get(n);
	}
}
```

Output: -

```
34
34
34
```

Knapsack Problem

- Its is a problem in combinatorial optimization
- Given a set of items, each with a mass ***w*** and a value ***v***, determine the number of each item to include in a collection so that the total weight ***M*** is less than or equal to a given limit and the total value is as large as possible
- The problem often arises in resource allocation where there are financial constraints

Applications: -

- Has lots of applications of course
- Finding the least wasteful way to cut raw materials
- Selection of investments and portfolios
- Selection of assets for asset-backed securitization
- Cconstruction and scoring of tests in which the test-takers have a choice as to which questions they answer

Divisible problem (Greedy Solution)

- If we can take fractions of the given items, then the greedy approach can be used
- Sort the items according to their values, it can be done in **O(N*logN)** time complexity
- Start with the item that is the most valuable and take as much as possible. In the final item that you take, take fraction to adjust the max that can be taken.
- Than try with the next item from our sorted list
- This linear search has **O(N)** time complexity
- Overall complexity: **O(N*logN) + O(N) = O(N*logN)** !!!
- So we can solve the divisible knapsack problem quite fast

0-1 knapsack problem

- In this case we are not able to take fractions: we have to decide whether to take an item or not
- Greedy algorithm will not provide the optimal result !!!
- **Dynamic programming** is the right way !!!

Time complexity

- Running time of Knapsack: **O(n*W)**
- **BUT** it is not polynomial, it is pseudo-polynomial
- Numeric algorithm runs in **pseudo-polynomial time** if its running time is polynomial in the *numeric value* of the input, but is exponential in the *length* of the input (the number of bits required to represent it)

N = 3 items **W = 5kg** capacity of knapsack

Item #1 w_1 = 4kg v_1 = $10
Item #2 w_2 = 2kg v_2 = $4
Item #3 w_3 = 3kg v_3 = $7

	0	1	2	3	4	5
No items	0	0	0	0	0	0
First item	1	0	0	0	0	10
First two items	2	0	0	4	4	10
All items	3	0	0	4	7	10

weights [kg]

$S[i][w] = Math.max(S[i-1][w] \; ; \; v_i + S[i-1][w-w_i])$

$S[3][5] = Math.max(S[2][5] \; ; \; \$7 + S[2,5-3]) = max(10,11)$

0-1 Knapsack Problem

- what items to include?
- we start with the last item (last row and last column) and we keep comparing the items right above (below) each other
- if the 2 values are the same: it means we have not included the given item in the knapsack (so we take 1 step upwards in the S table)
- otherwise we take 1 step upwards and take as many steps to the left as the w weight of that item

0-1 Knapsack Problem

$S[i][w] = Math.max(S[i-1][w] \; ; \; v_i + S[i-1][w-w_i])$

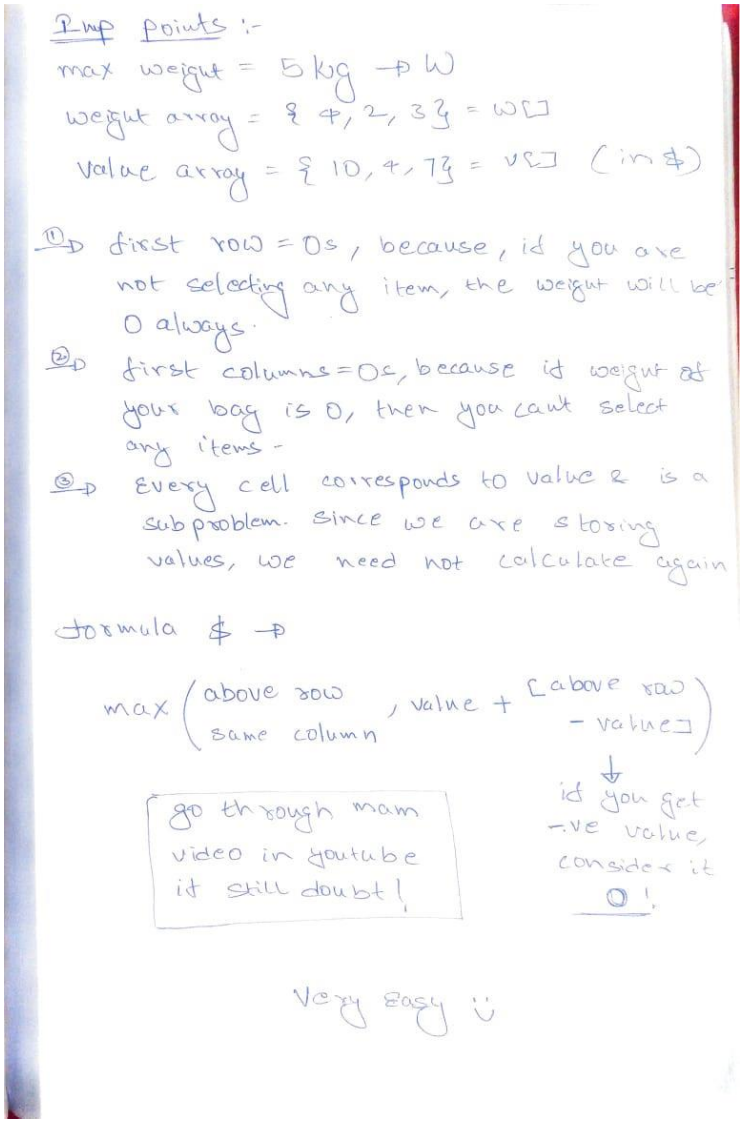
the maximum profit that fit inside a knapsack of weight w, choosing from the first i items

do not take i-th item

we take i-th item

- we have to use this S[][] two-dimensional array (list)
- we are only considering $S[i-1][w-w_i]$ if it can fit $w > w_i$
- if there is not room for it: the answer is just S[i-1][w] !!!

Karthik Chinni




```
public class Knapsack {
        private int numOfItems;
        private int capacityOfKnapsack;
        private int[][] knapsackTable;
        private int[] weights;
        private int[] values;
        private int totalBenefit;

        public Knapsack(int numOfItems, int capacityOfKnapsack, int[] weights, int[] values) {
                this.numOfItems = numOfItems;
                this.capacityOfKnapsack = capacityOfKnapsack;
                this.weights = weights;
                this.values = values;
                // +1 as we need to handle base case.
                // First row and column are 0's as we are initializing here.
                this.knapsackTable = new int[numOfItems + 1][capacityOfKnapsack + 1];
        }
```

Karthik Chinni

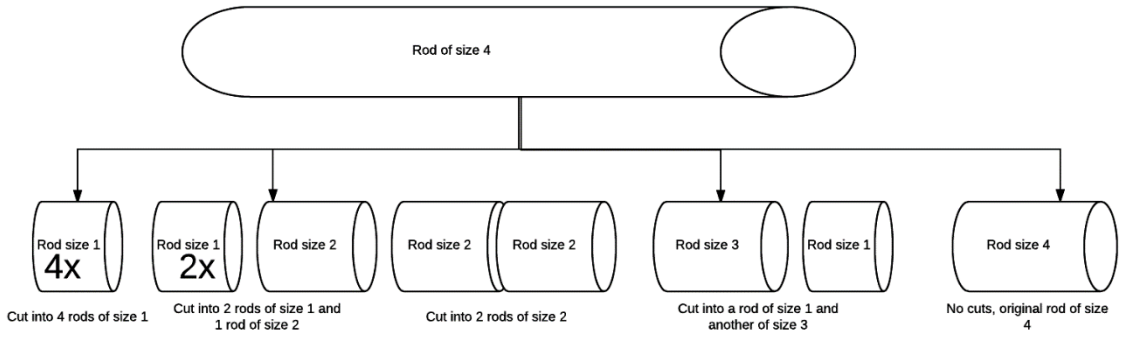
```
	public void solve() {
		// time complexity: O(N*W)
		// first column and row should be 0's. So, starting with index 1.
		for (int i = 1; i < numOfItems + 1; i++) {
			for (int w = 1; w < capacityOfKnapsack + 1; w++) {
				// Same column just above the Row
				int notTakingItem = knapsackTable[i - 1][w];
				int takingItem = 0;

				if (weights[i] <= w) {
					// Same column above Row - weight + value
					takingItem = values[i] + knapsackTable[i - 1][w - weights[i]];
				}

				knapsackTable[i][w] = Math.max(notTakingItem, takingItem);
			}
		}
		totalBenefit = knapsackTable[numOfItems][capacityOfKnapsack];
	}

	public void showResult() {
		System.out.println("Total benefit: " + totalBenefit);

		//Start from bottom right.
		for (int n = numOfItems, w = capacityOfKnapsack; n > 0; n--) {
			//If above item is not equal, then move left in above row by values times.
			if (knapsackTable[n][w] != 0 && knapsackTable[n][w] != knapsackTable[n - 1][w]) {
				System.out.println("We take item: #" + n);
				w = w - weights[n];
			}
		}
	}
}

public class KnapsackApp {

	public static void main(String[] args) {

		int numOfItems = 3;
		int capacityOfKnapsack = 5;

		// int[] weightOfItems = {4,2,3};
		// int[] profitOfItems = {10,4,7};

		int[] weightOfItems = new int[numOfItems + 1];
		weightOfItems[1] = 4;
		weightOfItems[2] = 2;
		weightOfItems[3] = 3;

		int[] profitOfItems = new int[numOfItems + 1];
		profitOfItems[1] = 10;
		profitOfItems[2] = 4;
		profitOfItems[3] = 7;

		Knapsack knapsack = new Knapsack(numOfItems, capacityOfKnapsack, weightOfItems, profitOfItems);
		knapsack.solve();
		knapsack.showResult();
	}
}
```

Output: -

```
Total benefit: 11
We take item: #3
We take item: #2
```

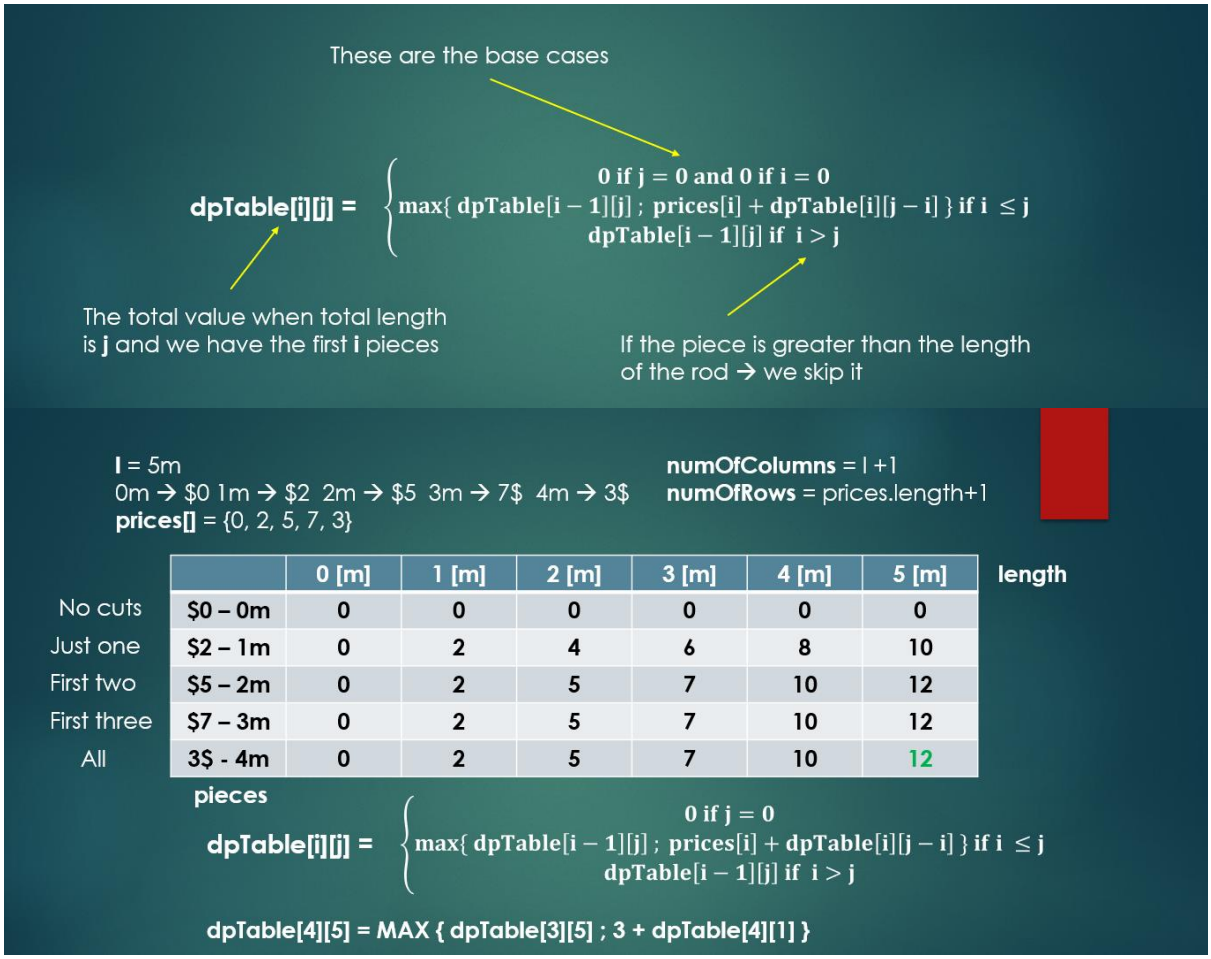
Rod cutting problem: -

- Given a rod with certain length l
- Given the prices of different lengths
- How to cut the rod in order to maximize the profit?
- This is the rod cutting problem



My understanding after solving the problem: -

1- First try constructing the matrix
2- Try to figure out a formula on how did you achieve that.
3- Write code for the same.
4- That's it!!!
5- Dynamic Programming is that simple!!!



Finding the solution at end is same as the 0/1 knapsack problem.

Karthik Chinni

```java
public class RodCutting {

	private int[][] dpTable;
	private int lenghtOfRod;
	private int[] prices;

	public RodCutting(int lengthOfRod, int[] prices) {
		this.prices = prices;
		this.lenghtOfRod = lengthOfRod;
		this.dpTable = new int[prices.length + 1][lengthOfRod + 1];
	}

	public void solve() {
		for (int i = 1; i < prices.length; i++) {
			for (int j = 1; j <= lenghtOfRod; j++) {
				if (i <= j) {
					// If piece < length of Rod, we consider it and make calculation.
					dpTable[i][j] = Math.max(dpTable[i - 1][j], prices[i] + dpTable[i - 1][j - i]);
				} else {
					// If piece is greater than length of Rod, we copy value from above column.
					dpTable[i][j] = dpTable[i - 1][j];
				}
			}
		}
	}

	public void showResult() {
		System.out.println("Optimal profit: $" + dpTable[prices.length - 1][lenghtOfRod]);
		for (int n = prices.length - 1, w = lenghtOfRod; n > 0;) {
			if (dpTable[n][w] != 0 && dpTable[n][w] != dpTable[n - 1][w]) {
				System.out.println("We make cut: " + n + "m");
				w = w - n;
			} else {
				n--;
			}
		}
	}
}

public class RodCuttingApp {
	public static void main(String[] args) {
		int lengthOfRod = 5;
		int[] prices = { 0, 2, 5, 7, 3 };

		RodCutting cutting = new RodCutting(lengthOfRod, prices);
		cutting.solve();
		cutting.showResult();
	}
}
```

```
Optimal profit: $12
We make cut: 3m
We make cut: 2m
```

Subset sum problem

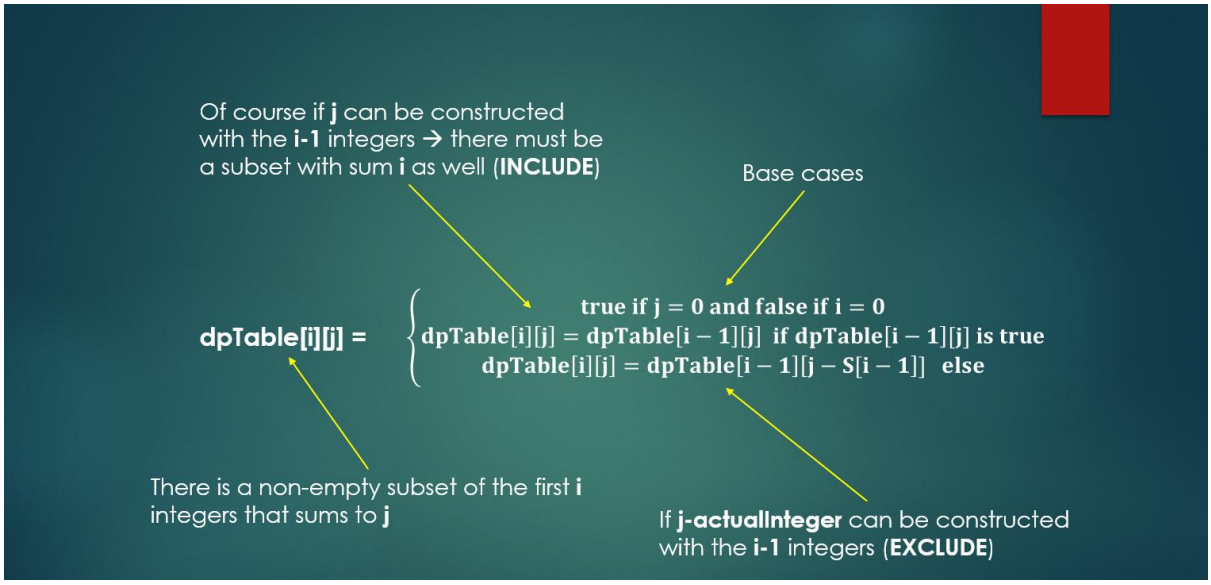
- One of the most important problems in complexity theory
- The problem: given an S set of integers, is there a non-empty subset whose s sum is zero or a given integer?
- For example: given the set {5,2,1,3} and s=9 the answer is YES because the subset {5,3,1} sums to 9
- The problem is NP-complete → we have efficient algorithms when the problem is small !!!
- Special case of knapsack-problem

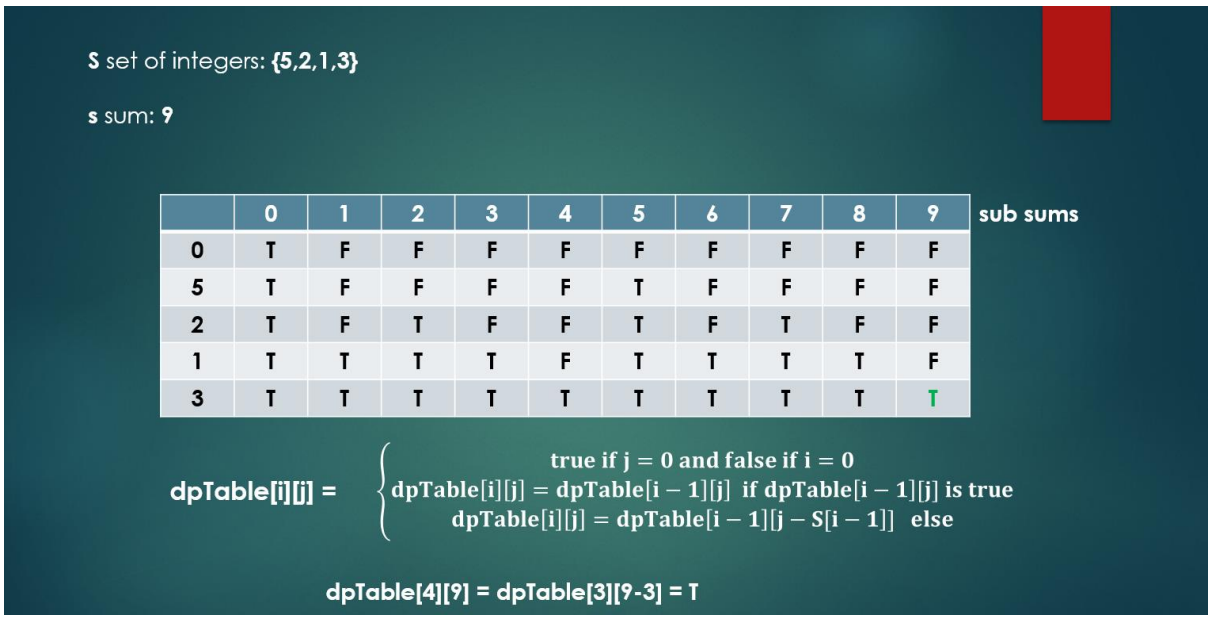
Solutions

1.) Naive approach - brute force search

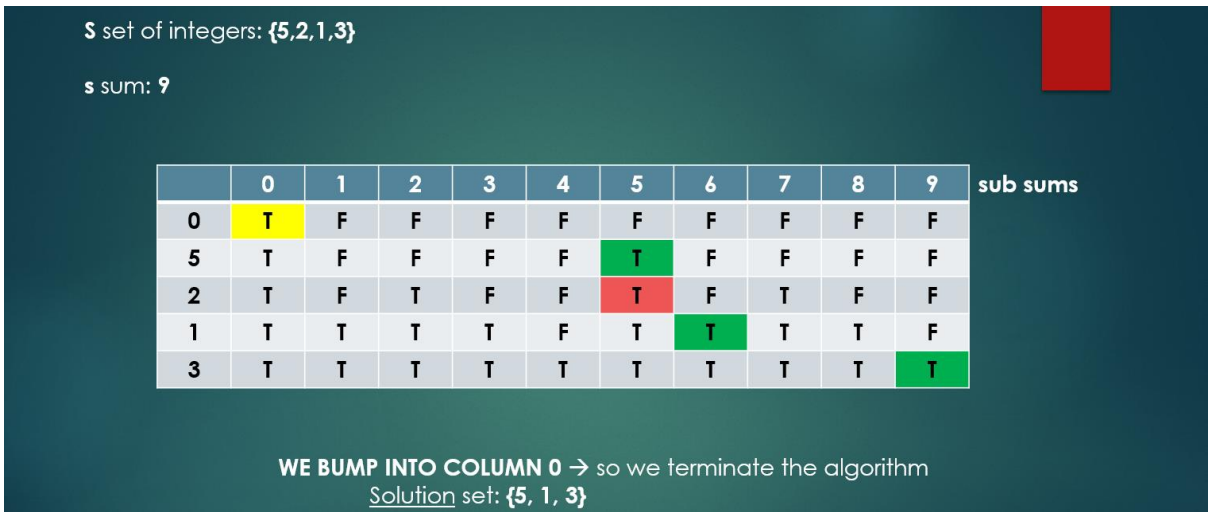
- Generate all the subsets of the given set of integers
- **N** is the number of integers in the set **S**
- Check whether the sum of all subsets is equal to **s** or not
- Time complexity: exponential // **O($N * 2^N$)**

2.) Dynamic Programming: we want to avoid calculating the same problems over and over again ... we create a dynamic programming table and memoize





	0	1	2	3	4	5	6	7	8	9
0	T	F	F	F	F	F	F	F	F	F
5	T	F	F	F	F	T	F	F	F	F
2	T	F	T	F	F	T	F	T	F	F
1	T	T	T	T	F	T	T	T	T	F
3	T	T	T	T	T	T	T	T	T	T



	0	1	2	3	4	5	6	7	8	9
0	T	F	F	F	F	F	F	F	F	F
5	T	F	F	F	F	T	F	F	F	F
2	T	F	T	F	F	T	F	T	F	F
1	T	T	T	T	F	T	T	T	T	F
3	T	T	T	T	T	T	T	T	T	T

Karthik Chinni

```java
public class SubsetSumProblem {

	private boolean[][] dpTable;
	private int[] numbers;
	private int sum;

	public SubsetSumProblem(int[] numbers, int sum) {
		this.numbers = numbers;
		this.sum = sum;
		this.dpTable = new boolean[numbers.length + 1][sum + 1];
	}

	public void solveProblem() {

//			for(int i=0;i<=this.sum;i++){ // if sum is not zero and subset is 0 -> no feasible solution
//				this.dpTable[0][i] = false;
//			}

			for (int i = 0; i <= this.numbers.length; i++) { // if sum is 0 the we can make the empty subset to make sum 0
				this.dpTable[i][0] = true;
			}

			for (int rowIndex = 1; rowIndex <= numbers.length; ++rowIndex) {
				for (int columnIndex = 1; columnIndex <= sum; ++columnIndex) {

					if (columnIndex < numbers[rowIndex - 1]) {
						this.dpTable[rowIndex][columnIndex] = this.dpTable[rowIndex - 1][columnIndex];
					} else {
						if (this.dpTable[rowIndex - 1][columnIndex]) {
							this.dpTable[rowIndex][columnIndex] = true;
						} else {
							this.dpTable[rowIndex][columnIndex] = this.dpTable[rowIndex - 1][columnIndex
									- numbers[rowIndex - 1]];
						}
					}
				}
			}
	}

	public void hasSolution() {
		for (int i = 0; i < this.numbers.length + 1; i++) {
			for (int j = 0; j < this.sum + 1; j++) {
				System.out.print(dpTable[i][j] + " ");
			}
			System.out.println();
		}
		if (this.dpTable[numbers.length][sum]) {
			System.out.println("There is a solution for the problem...");
		} else {
			System.out.println("No feasible solution for the problem...");
		}
	}

	public void showSums() {

		int columnIndex = this.sum;
		int rowIndex = this.numbers.length;

		while (columnIndex > 0 || rowIndex > 0) {

			if (this.dpTable[rowIndex][columnIndex] == this.dpTable[rowIndex - 1][columnIndex]) {
				rowIndex = rowIndex - 1;
			} else {
				System.out.println("We take item: " + numbers[rowIndex - 1]);
				columnIndex = columnIndex - numbers[rowIndex - 1];
				rowIndex = rowIndex - 1;
			}
		}
	}
}

public class SubSetApp {
	public static void main(String[] args) {

		int[] numbers = { 5, 2, 3, 1 };
		int sum = 9;

		SubsetSumProblem subsetSumProblem = new SubsetSumProblem(numbers, sum);
		subsetSumProblem.solveProblem();
		subsetSumProblem.hasSolution();
		subsetSumProblem.showSums();

	}
}
```

```
true false false false false false false false false false
true false false false false true false false false false
true false true false false true false true false false
true false true true false true false true true false
true true true true true true true true true true
There is a solution for the problem...
We take item: 1
We take item: 3
We take item: 5
```

Karthik Chinni

Bin Packing Algorithms

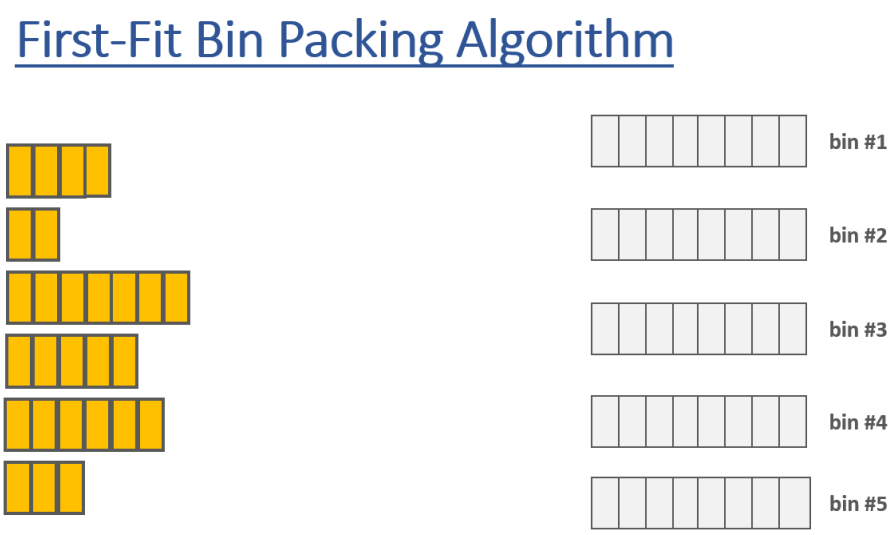
- **bin packing problem** is about how to fit several items into bins (containers) in an efficient way
- it is an **NP-complete** problem
- items of different $\mathbf{w_i}$ volumes must be packed into a finite **N** number of bins (containers) - each of these bins has volume **V**
- in of course the aim is to **minimize the number of bins** used
- when the number of bins is restricted to **1** and each item is characterized by both a volume and a value - the problem of maximizing the value of items that can fit in the bin is known as the **knapsack problem**

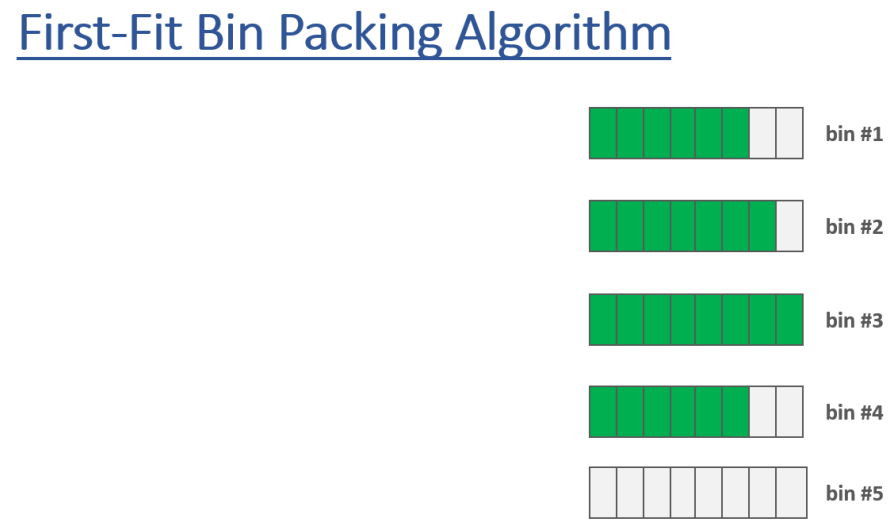
Approaches: -

1.) **naive approach – '' brute-force search"**
- iiterate over all bins and try to put the current item in the bin and (if it fits) call the same method with the next item.

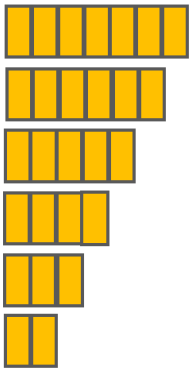
2.) **first-fit algorithm**
- iterate over all the items we want to put into bins - if we are not able to put it into a given bin we try to put it into the next one
- yields non-optimal solutions in the main

3.) **first-fit decreasing algorithm**
- sorting the items in decreasing order may be helpful - after sorting we use first-fit algorithm
- yields non-optimal solutions in the main

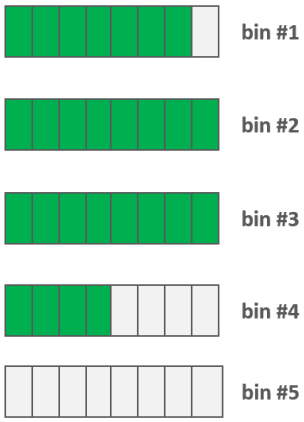




First-Fit Decreasing Bin Packing Algorithm



First-Fit Decreasing Bin Packing Algorithm



Applications: -

- we have **N** groups of people with group sizes w_1, w_2 ... w_N. We have minibuses with capacity **C**. What is the optimal number of minibuses when the groups must stay together?
- **virtual machines** in programming languages often have to solve this problem
- **television advertisements**: we are given a certain time slot (for example **10** minutes). How do we pack the most commercials into each time slot and maximize our daily profits?

```
public class Bin {
	private int id;
	private int capacity;
	private int currentSize;
	private List<Integer> items;

	public Bin(int capacity, int id) {
		this.capacity = capacity;
		this.id = id;
		this.items = new ArrayList<Integer>();
	}

	public boolean put(Integer item) {
		if (this.currentSize + item > this.capacity)
			return false;
		this.items.add(item);
		this.currentSize += item;
		return true;
	}

	@Override
	public String toString() {
		String contentOfBin = "Items in bin #" + this.id + ": ";
		for (Integer item : this.items) {
			contentOfBin += item + " ";
		}
		return contentOfBin;
	}
}
```

Karthik Chinni

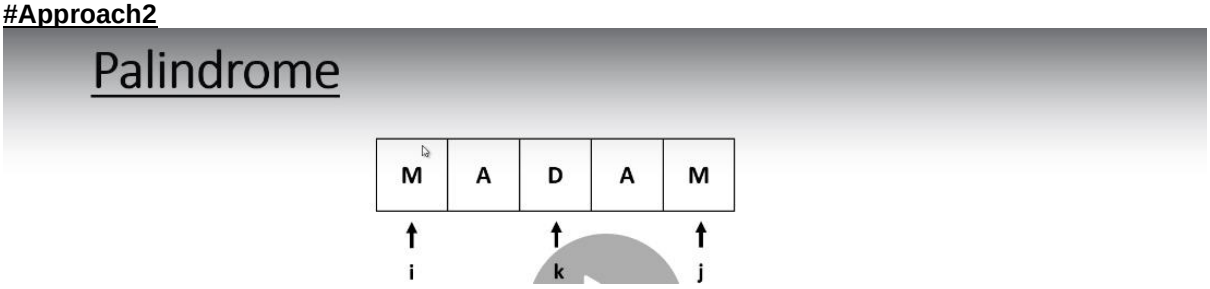
```
public class FirstFitDecreasingAlgorithm {

	private List<Bin> bins;
	private List<Integer> items;
	private int binCapacity;
	private int binCounter = 1;

	public FirstFitDecreasingAlgorithm(List<Integer> items, int binCapacity) {
		this.items = items;
		this.binCapacity = binCapacity;
		this.bins = new ArrayList<>(this.items.size());
	}

	public void solveBinPackingProblem() {
		// TC => O(n log n)
		Collections.sort(this.items, Collections.reverseOrder());

		if (this.items.get(0) > this.binCapacity) {
			System.out.println("No feasible solution...");
			return;
		}

		this.bins.add(new Bin(binCapacity, binCounter)); // first bin !!!

		// TC => O(n^2)
		for (Integer currentItem : items) {
			boolean isOk = false; // track whether we have put the item into a bin or not
			int currentBin = 0;
			while (!isOk) {
				if (currentBin == this.bins.size()) { // item doesn't fit in last bin -> try new
					Bin newBin = new Bin(binCapacity, ++binCounter);
					newBin.put(currentItem);
					this.bins.add(newBin);
					isOk = true;
				} else if (this.bins.get(currentBin).put(currentItem)) { // current item fits in bin
					isOk = true;
				} else {
					currentBin++; // trying the next bin
				}
			}
		}
		// Total TC => O(n log n) + O(n^2) = O(n^2)
	}

	public void showResults() {
		for (Bin bin : this.bins) {
			System.out.println(bin);
		}
	}
}

public class BinPackingApp {
	public static void main(String[] args) {

		List<Integer> items = Arrays.asList(5, 5, 5);
		int binCapacity = 10;

		FirstFitDecreasingAlgorithm algorithm = new FirstFitDecreasingAlgorithm(items, binCapacity);
		algorithm.solveBinPackingProblem();
		algorithm.showResults();

	}
}
```

```
Items in bin #1: 5 5
Items in bin #2: 5
```

Karthik Chinni

Interview Questions: -

#Approach2




```
public class Palindrome {

	public static void main(String[] args) {
		System.out.println(isPalindrome1("madam"));
		System.out.println(isPalindrome2("madam"));
	}

	/**
	 * Reverse the given String.
	 * Compare original String with reversed String.
	 */
	public static boolean isPalindrome1(String originalString) {
		String reversedString = "";
		for (int i = originalString.length() - 1; i >= 0; i--) {
			reversedString = reversedString + originalString.charAt(i);
		}
		return originalString.equals(reversedString);
	}

	/**
	 * Have 3 indexes i->start,j->end,k->middle
	 * Start comparing i and j until they meet k
	 * if i != j in current iteration , return false.
	 */
	public static boolean isPalindrome2(String originalString) {
		int i = 0;
		int j = originalString.length() - 1;
		int k = (i + j) / 2;

		//Using <= so that even length Strings are also handled.
		while (i <= k && j >= k) {
			if (originalString.charAt(i) == originalString.charAt(j)) {
				i++;
				j--;
			} else {
				return false;
			}
		}
		return true;
	}
}
```

Reverse an Integer: -

We cannot use the same method as above as String reverse because this is integer and this needs different operators. We made use of a char[] to deal with String. But, since this is Integer, we need to use the mathematical operations to get the reverse of the given number.

This is also an important interview question.

```
public class ReverseInteger {
	public static void main(String[] args) {
		System.out.println(getIntegerReverse(12345678));
	}

	private static int getIntegerReverse(int n) {
		int reversedNumber = 0;
		int reminder = 0;
		while (n > 0) {
			reminder = n % 10;
			reversedNumber = (reversedNumber * 10) + reminder;
			n = n / 10;
		}
		return reversedNumber;
	}
}
```

Output: -

```
87654321
```

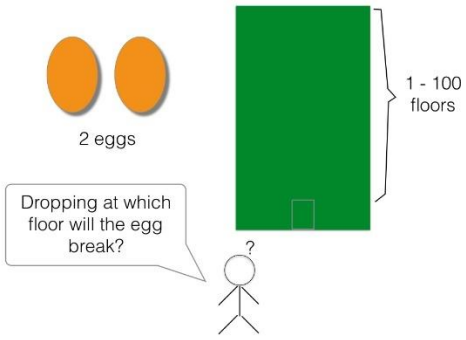
Egg Dropping problem: -

The problem itself is that there are **N** number of eggs and building which has **M** floors. Write an algorithm to find the minimum number of drops is required to know the floor from which if egg is dropped, it will break.

(usually, **N=2** and **M=100**)

Assumptions: -

1- Egg that doesn't break can be reused.
2- If egg breaks when dropped from a given floor, then it will break when dropped from any other higher floor.
3- If egg doesn't break when dropped from given floor, then it will not break even when dropped from any lower floor.



One Egg Problem

Let's first consider a simplified version of the problem. What if you have only one egg and you need to find the floor. You don't have any other option exception trying floors on one-by-one basis.

Consider that egg breaks, say, at floor 4, then 3. Is the last safe floor where the egg will not break when dropped.
TC => O(N)

Karthik Chinni

Floor#100			Floor#100	
Floor#99			Floor#99	
"			"	
"			"	
"			"	
Floor#4	<- Egg breaks here.		Floor#4	
Floor#3			Floor#3	<- Last safe floor.
Floor#2			Floor#2	
Floor#1			Floor#1	

Total steps involved: - "4" (in worst case, its 100)

Many Eggs Problem: -

Say, we have infinite number of eggs. Now, we can use binary search to find the solution to the problem.

1- We drop the egg at floor 50.
2- If egg breaks, then we can ignore the floors 51 to 100, else we can ignore floors 1 to 49.
3- Thus, steps to find the critical floor (last safe floor) is **$\log_2(100) \sim 6.643856 \Rightarrow 7$**
4- So, answer is 7 steps.

TC => O(log N)

Two Eggs Problem: -

Say, we are using binary search strategy to find solution to two eggs problem:

- we'd start with floor 50, and drop the egg from the 50th floor.
- If the egg breaks, we know our *n* is below 50, if it's intact the floor we want is above 50.
- Supposing the egg breaks, the problem here is now we've lost an egg and only narrowed down our search by 50 numbers.
- The next step would normally be to split that 50 again and dropping the egg at floor 25, but if it breaks, we still haven't found our floor. If it doesn't, it we've only delayed the inevitable by one more step — and still no solution. **Worst case scenario: we have to drop our eggs 50 times.**

- We cannot check in one-by-one floor just like the one egg problem as we have two eggs and we need to somehow make use of the second egg to reduce the number of steps taken. If we check one by one, the steps are 100.

Can we make it better? Of course, yes!

- The next solution takes a little bit of the linear approach and mixes in a little of the splitting from our binary approach.
- We can start off by dropping an egg at floor 10, increasing the drop floor by 10 at a time, then going back to drop one floor at a time until we find that n.
- If our egg breaks at floor 10, we know n is one of the 9 floors below us.
- Worst case, the egg drops and doesn't break until floor 100 (10 drops) and we drop the second egg but don't break it for floors 91–99.
- **It brings our worst case drop count to 19 drops.**

What is the problem in above approach?

- Even though we completed 10 drops (till 100th floor), we again need to do 9 more drops in worst case to solve the problem.
- So, the problem is that there is a constant number of linear iterations that we need to make to find the solution in any given floor.

Final Solution: -

- **Note: - To reduce worst cases, we need to make sure that the number of steps in every iteration are same.** 😊
- If getting to the higher floors means more drops overall, we need to decrease the drops we need to perform linearly. We're essentially trying to make all possible scenarios take the same number of drops to solve.
- If we drop our first egg from floor x (10 in our 10-floor strategy), the linear portion of our strategy is x-1 (9 in the above strategy). Our drop count is: **x + (x - 1)**

Karthik Chinni

- ✓ If the egg doesn't break on the first drop our drop count increases by one, so we'll need to remove a drop from our floor by floor drop count — the next drop should be from x-1 floors up.
- ✓ Every additional floor jump will need to have one less floor, so that when we get to the linear portion of the solution, we'll have one less floor to check.
- ✓ We continue removing one floor until we only have 1 floor to check:

```
x + (x-1) + (x-2) + (x-3) + ... + 1
Which simplifies to x(x + 1)/2

x(x + 1)/2 = 100

x = 13.651
```

https://medium.com/@khopsickle/2-eggs-and-100-floors-a032beb77aaa

1													
2	15												
3	16	28											
4	17	29	40										
5	18	30	41	51									
6	19	31	42	52	61								
7	20	32	43	53	62	70							
8	21	33	44	54	63	71	78						
9	22	34	45	55	64	72	79	85					
10	23	35	46	56	65	73	80	86	91				
11	24	36	47	57	66	74	81	87	92	96			
12	25	37	48	58	67	75	82	88	93	97	100		
13	26	38	49	59	68	76	83	89	94	98	101	103	
> 14	27	39	50	60	69	77	84	90	95	99	102	104	105

DP Questions:

1 Two Eggs and 100 Floor Classic Puzzle
2 Five pirates and gold coin Puzzle
3 Six pirates and Gold Coin puzzle
4 Probability of having boy
5 Random Airplane Seats
6 Inverted playing card puzzle
7 Flipping Coins Puzzle
8 Three hat colours Microsoft Puzzle
9 25 horses 5 tracks Puzzle
10 Gold Bar Puzzle
11 Crossing the Bridge Puzzle
12 Will you accept the bet?
13 The Puzzle of 100 Hats
14 Man fell in Well Puzzle
15 Minimum Number of Weights
16 One Bulb with 3 Switches
17 Find the minimum number of aircraft
18 Burning ropes to measure time
19 Connect 3 houses with 3 wells
20 Probability of having boy
21 Ant and Triangle Problem
22 The Man in the Elevator
23 Find the survivor
24 Free the prisoner's puzzle