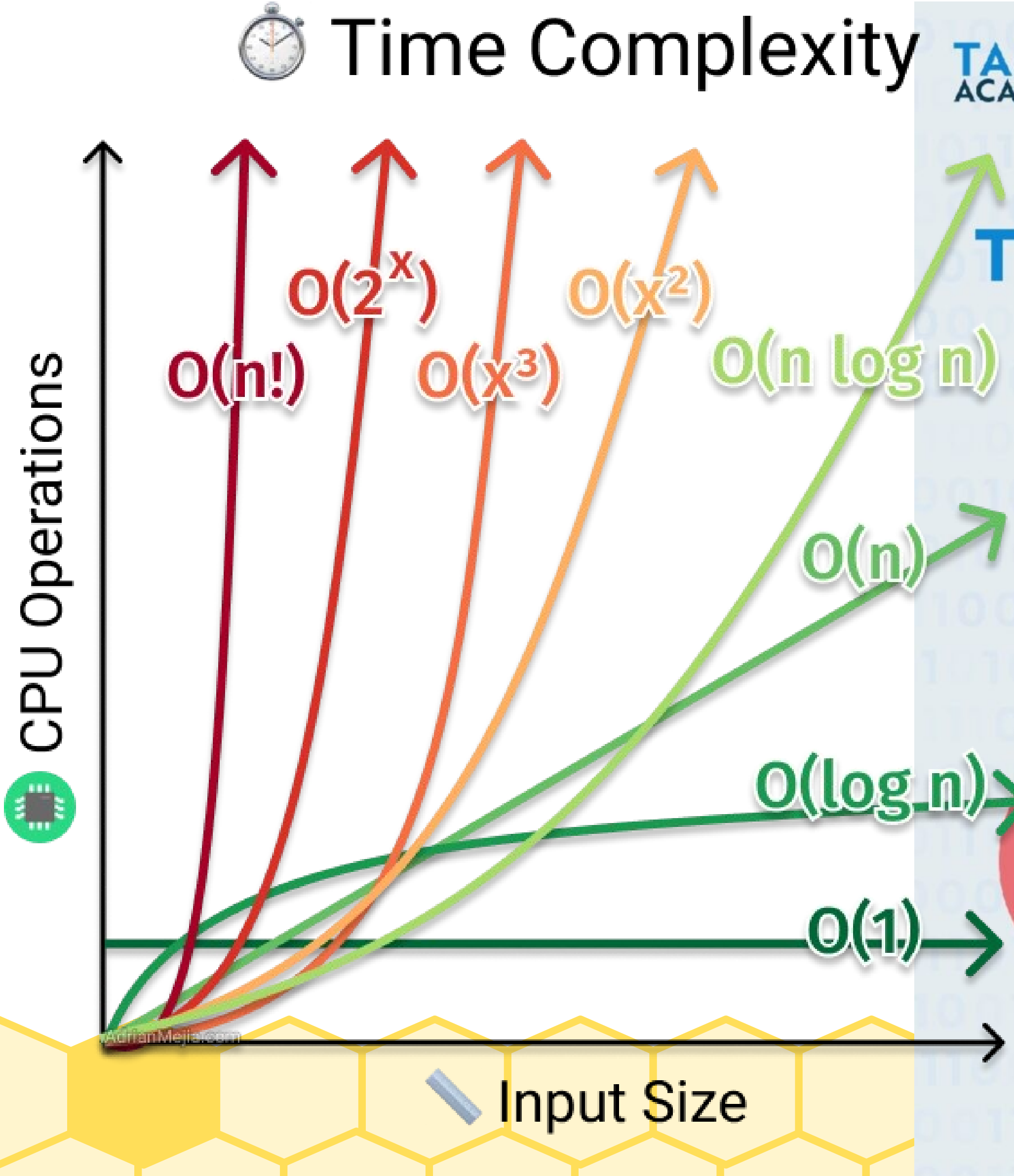
Time Complexity

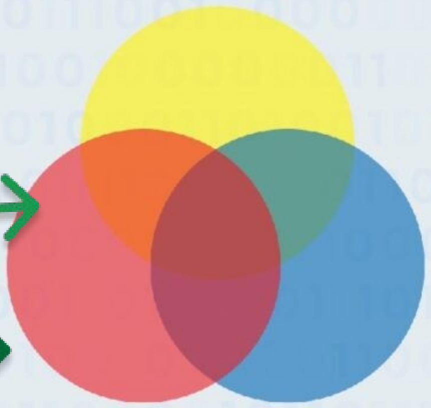
Time complexity of a simple loop when the loop variable is incremented or decremented by a constant amount

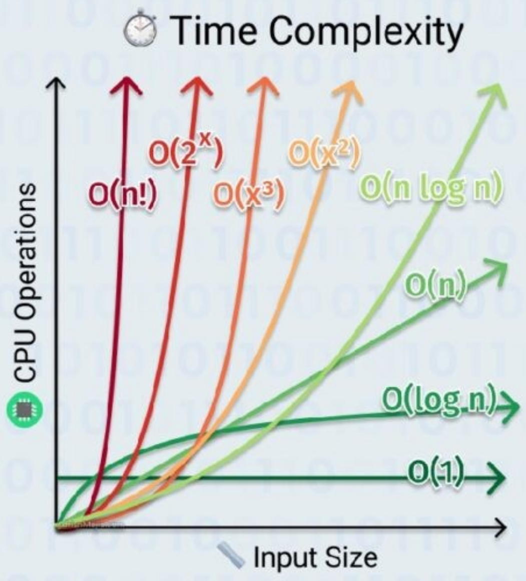
Introduction

- There is always more than one way to solve a problem in computer science with different algorithms. Therefore, it is highly required to use a method to compare the solutions in order to judge which one is more optimal. The method must be:
  - Independent of the machine and its configuration, on which the algorithm is running on.
  - Shows a direct correlation with the number of inputs.
  - Can distinguish two algorithms clearly without ambiguity.



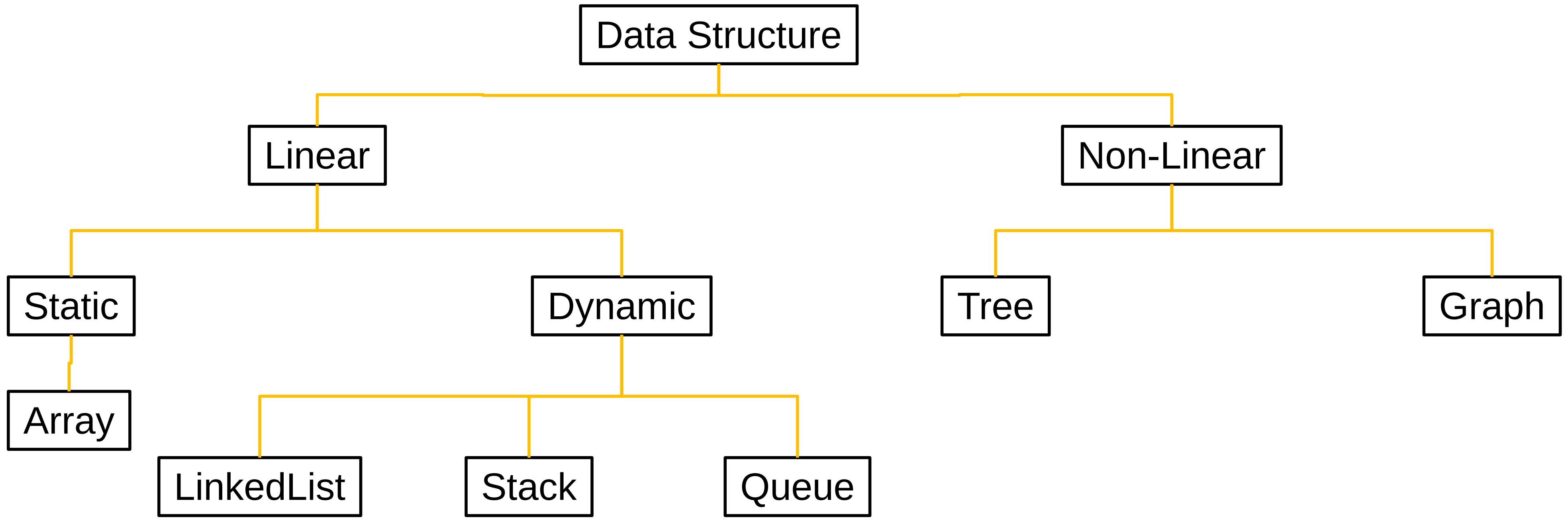
Time and Space Complexity in Data Structure





Why Data Structure?

- Efficient Memory Usage
- Improved Performance
- Enhanced Data Management
- Supports Complex Functionality
- Data Structure Reusability
- Better Problem-Solving


Data Structure
Linear
Non-Linear
Static
Dynamic
Tree
Graph
Array
LinkedList
Stack
Queue

Data Structure Characteristics

- Linear vs. nonlinear: sequence structure, or the order of items in a data set
- Homogeneity vs. heterogeneity: compositional characteristics within a given repository
- Static vs. dynamic data structure: variation in terms of size, structure, and memory location
- There are many approaches to data set classification based on complexity: primitive, non-primitive, and abstract.

Primitive Data Structures

- byte: Stores whole numbers from -128 to 127.
- short: Stores whole numbers from -32,768 to 32,767.
- int: Stores whole numbers from -2,147,483,648 to 2,147,483,647.
- float: Stores floating-point numbers with single precision.
- char: Stores individual characters.
- boolean: Stores true or false values.
- long: Stores large whole numbers.
- double: Stores floating-factor numbers with double precision.

Non-Primitive Data Structure

- Linear Data Structures
  - Arrays
  - Stacks
  - Queues
  - Linked List
- Non-linear Data Structures
  - Trees: red-black trees, AVL trees, binary search trees, and binary trees.
  - Graphs: A set of nodes linked by using edges, wherein nodes may have any quantity of connections. Graphs are used to symbolize complex relationships among items.
  - Heap: A specialized tree-based structure in which every determined node has a value more or smaller than its kids, relying on whether or not it is a max heap or a min heap.
  - Hash: Data structures that use a hash function to map keys to values. Examples consist of hash sets and hash maps, which provide green retrieval and storage of statistics based on precise keys.

Non-Primitive Data Structures

- Array
  - Array Insertion
  - Array Deletion
  - Array Traversal
- LinkedList
  - LinkedList Insertion
  - LinkedList Deletion
  - LinkedList Traversal
- Stack
  - Push
  - Pop
- Queues

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

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

LinkedList Insertion

```java
import java.util.LinkedList;

public class LinkedListInsertionExample {
    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<>();

        // Insert elements using add() method
        list.add("apple");
        list.add("banana");
        list.add("cherry");

        // Print the LinkedList
        System.out.println("LinkedList after insertion: " + list);
    }
}
```

LinkedList Deletion

```
import java.util.LinkedList;

public class LinkedListDeletionExample {
    public static void main(String[] args) {
        LinkedList<Integer> numbers = new LinkedList<>();

        // Add elements to the LinkedList
        numbers.add(10);
        numbers.add(20);
        numbers.add(30);
        numbers.add(40);

        // Remove a specific element using remove()
        numbers.remove(Integer.valueOf(30));

        // Print the modified LinkedList
        System.out.println(numbers); // Output: [10, 20, 40]
    }
}
```

LinkedList Traversal

```
import java.util.LinkedList;

public class LinkedListTraversalExample {
    public static void main(String[] args) {
        // Create a linked list
        LinkedList<Integer> linkedList = new LinkedList<>();
        linkedList.add(1);
        linkedList.add(2);
        linkedList.add(3);
        linkedList.add(4);
        linkedList.add(5);

        // Traverse the linked list
        for (Integer element : linkedList) {
            System.out.print(element + " ");
        }
    }
}
```

Stack Push and Pop

```
import java.util.Stack;

public class StackPushAndPopExample {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        stack.push(10);
        stack.push(20);
        stack.push(30);
        System.out.println("Stack after push: "+stack);
        //The pop method removes items from the stack.

        int poppedElement = stack.pop(); // Returns 30
        System.out.println("Stack after pop: "+stack);
    }
}
```

Queues

```
import java.util.LinkedList;
import java.util.Queue;

public class QueuesExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        queue.add("Alice");
        queue.add("Bob");
        queue.add("Charlie");
        //The dequeue method removes and returns the most recent/first element
from the queue.

        String dequeuedElement = queue.poll(); // Returns "Alice"
    }
}
```

Trees Insertion

Adding a new element in a binary search tree should be designed in such a way it does not violate each value. Example:

10

/ \

5 15

We can insert a value of 12 as follows:

10

/ \

5 15

/ \ / \

3 7 12 18

Trees Insertion

```java
package binary_tree;

public class BinarySearchTree {
    Node root;

    public BinarySearchTree() {
        root = null;
    }

    // Recursive insertion method
    public Node insert(Node node, int value) {
        if (node == null) {
            return new Node(value);
        }

        if (value < node.data) {
            node.left = insert(node.left, value);
        } else if (value > node.data) {
            node.right = insert(node.right, value);
        }

        // value already present, do nothing
        return node;
    }
    public static void main(String[] args) {
        BinarySearchTree tree = new BinarySearchTree();
        tree.root = tree.insert(tree.root, 50);
        tree.insert(tree.root, 30);
        tree.insert(tree.root, 20);
        System.out.println(tree.root.data);
        System.out.println(tree.root.left.data);
        System.out.println(tree.root.right.data);
    }
}
```

```java
package binary_tree;

public class Node {
    int data;
    Node left, right;

    public Node(int item) {
        data = item;
        left = right = null;
    }
}
```

Trees Deletion

Removing a node in binary trees requires replacing the deleted node with its successor or predecessor (whichever is available first).

For example, given the above tree, if we wanted to delete 10 we could replace it with its in-order successor 12, as follows:

12

/ \

5 15

/ \ / \

3 7 10 18

Trees Deletion

```java
package binary_tree.deletion;

public class BinarySearchTreeDeletion {
    Node root;

    BinarySearchTreeDeletion() {
        root = null;
    }

    void insert(int data) {
        root = insertRec(root, data);
    }

    Node insertRec(Node node, int data) {
        if (node == null) {
            return new Node(data);
        }

        if (data < node.data) {
            node.left = insertRec(node.left, data);
        } else if (data > node.data) {
            node.right = insertRec(node.right, data);
        }

        return node;
    }

    // Function to delete a node from BST
    void deleteNode(int key) {
        root = deleteRec(root, key);
    }

    // Recursive function to delete a node with given key from subtree rooted with
root and returns new root
    Node deleteRec(Node root, int key) {
        if (root == null)  return root;

        // If the key to be deleted is smaller than the root's key, then it lies in
left subtree
        if (key < root.data) {
            root.left = deleteRec(root.left, key);
        } else if (key > root.data) {
            // If the key to be deleted is greater than the root's key, then it
lies in right subtree
            root.right = deleteRec(root.right, key);
        } else {
            // If the node with the key to be deleted has one or no children
            if (root.left == null) {
                return root.right;
            } else if (root.right == null) {
                return root.left;
            }

// If the node to be deleted has two children: Get the inorder successor (smallest
in the right subtree)
            Node temp = minValue(root.right);

            // Copy the inorder successor's content to this node
            root.data = temp.data;

            // Delete the inorder successor
            root.right = deleteRec(root.right, temp.data);
        }

        return root;
    }

    // Function to find the minimum value in a node's right subtree
    Node minValue(Node node) {
        while (node.left != null) {
            node = node.left;
        }
        return node;
    }

    void inorder() {
        inorderRec(root);
    }

    void inorderRec(Node root) {
        if (root != null) {
            inorderRec(root.left);
            System.out.print(root.data + " ");
            inorderRec(root.right);
        }
    }

    public static void main(String[] args) {
        BinarySearchTreeDeletion tree = new BinarySearchTreeDeletion();

        tree.insert(50);
        tree.insert(30);
        tree.insert(20);
        tree.insert(40);
        tree.insert(70);
        tree.insert(60);
        tree.insert(80);

        System.out.println("Inorder traversal before deletion: ");
        tree.inorder();

        tree.deleteNode(20);
        System.out.println("\nInorder traversal after deleting 20: ");
        tree.inorder();
    }
}
```

Trees Traversal

A graph traversal ensures all nodes in a tree data structure are visited exactly once. There are three types of traversals:

- Preorder traversals start by visiting the roots before moving to the subtrees.
- Post-order traversals start with the subtrees moving onto the root.
- Inorder traversals that start with the left child (and the entire subtree) move onto the root and end with visiting the right child.

For example, given our original binary tree, we could do an in-order traversal as follows: 3->5->7->10->12->15->18

Trees Traversal

```java
package binary_tree;

public class BinaryTreeTraversal {
    // Node class representing a node in the
binary tree
    public static class Node {
        int data;
        Node left;
        Node right;

        public Node(int data) {
            this.data = data;
            left = right = null;
        }
    }

    // Root node of the binary tree
    public Node root;

    public BinaryTreeTraversal() {
        root = null;
    }

    // In-order traversal (left, root, right)
    public void inOrderTraversal(Node node) {
        if (node != null) {
            inOrderTraversal(node.left);
            System.out.print(node.data + " ");
            inOrderTraversal(node.right);
        }
    }
// Pre-order traversal (root, left, right)
    public void preOrderTraversal(Node node) {
        if (node != null) {
            System.out.print(node.data + " ");
            preOrderTraversal(node.left);
            preOrderTraversal(node.right);
        }
    }

    // Post-order traversal (left, right, root)
    public void postOrderTraversal(Node node) {
        if (node != null) {
            postOrderTraversal(node.left);
            postOrderTraversal(node.right);
            System.out.print(node.data + " ");
        }
    }

    // Example usage: Create a binary tree and perform
traversals
    public static void main(String[] args) {
        BinaryTreeTraversal tree = new
BinaryTreeTraversal();

        // Sample tree
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);

        System.out.println("In-order traversal:");
        tree.inOrderTraversal(tree.root);
        System.out.println("\nPre-order traversal:");
        tree.preOrderTraversal(tree.root);
        System.out.println("\nPost-order traversal:");
        tree.postOrderTraversal(tree.root);
    }
}
```

Trees Search

Binary search trees provide efficient searching capabilities since each branch only has two options, and each move toward left or right reduces the number of nodes that need to be searched by half. For example, given our original binary tree, we could search for 7 as follows:

10

/ \

5 15

/ \ / \

3 7 12 18

Binary Trees Search

```java
package binary_tree;

public class BinarySearch {
    // Node class represents a node in the tree
    public static class Node {
        int data;
        Node left;
        Node right;

        public Node(int data) {
            this.data = data;
            this.left = null;
            this.right = null;
        }
    }

    private Node root; // Root node of the BST

    public BinarySearch() {
        this.root = null;
    }

    // Insert a new node into the BST
    public void insert(int data) {
        Node newNode = new Node(data);
        if (root == null) {
            root = newNode;
            return;
        }
Node current = root;
        while (true) {
            if (data < current.data) {
                if (current.left == null) {
                    current.left = newNode;
                    break;
                }
                current = current.left;
            } else {
                if (current.right == null) {
                    current.right = newNode;
                    break;
                }
                current = current.right;
            }
        }
    }

    // Search for a node with a specific data value
    public boolean search(int data) {
        Node current = root;
        while (current != null) {
            if (data == current.data) {
                return true; // Found the node
            } else if (data < current.data) {
                current = current.left;
            } else {
                current = current.right;
            }
        }
        return false; // Node not found
    }
}
```

Abstract Data Types

An abstract data type (ADT) serves as a base on which a data structure will be attached without impacting the implementation process. Abstract data types can be classified as

- Built-in/user-defined
- Mutable/immutable

In Java, an abstract class is defined by interface contracts for a specific data structure.

List

- A segment of the Java Collections Framework is built for array and linked list implementation.

```
import java.util.ArrayList;
import java.util.List;

public class ListDemo {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        System.out.println(names.get(1)); // Output: "Bob"
    }
}
```

Set

- The AbstractSet class in Java provides a body framework for the set interface implementing the collection interface and the abstract collection class.
- It differs from the list interface in that it doesn’t allow duplicate elements.
- It provides methods like add, remove, contain, and size.
- Use the set interface to store a set of numbers:

Set

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class ListDemo {
    public static void main(String[] args) {
        Set<Integer> numbers = new HashSet<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(2); // Duplicate element, not added

        System.out.println(numbers.contains(2)); // Output: true
    }
}
```

Map

- A map interface in Java stores data in key and value pairs with unique keys.
- It provides methods like put, get, remove, and containsKey.
- It is commonly used for key-value-pair storage.

```
import java.util.*;

public class ListDemo {
    public static void main(String[] args) {
        Map<String, Integer> ages = new HashMap<>();
        ages.put("Alice", 25);
        ages.put("Bob", 30);
        ages.put("Charlie", 35);

        System.out.println(ages.get("Bob")); // Output: 30
    }
}
```

TreeSet

```java
import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<String> names = new TreeSet<>();

        // Add elements to the TreeSet
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");
        names.add("David");
        names.add("Alice"); // Duplicates are not allowed

        // Print the sorted elements
        System.out.println("Elements in the TreeSet:");
        for (String name : names) {
            System.out.println(name);
        }

        // Get the first and last elements
        System.out.println("\nFirst element: " + names.first());
        System.out.println("Last element: " + names.last());

        // Check if an element exists
        System.out.println("\nDoes 'Charlie' exist? " + names.contains("Charlie"));
        // Navigating the TreeSet
        System.out.println("\nElements greater than 'Bob':");
        for (String name : names.tailSet("Bob")) {
            System.out.println(name);
        }

        // Remove elements
        names.remove("Alice");
        System.out.println("\nElements after removing 'Alice':");
        for (String name : names) {
            System.out.println(name);
        }
    }
}
```

TreeMap

```java
import java.util.Map;
import java.util.SortedMap;
import java.util.TreeMap;

public class TreeMapDemo {
    public static void main(String[] args) {
        // Create a TreeMap with String keys and Integer values
        TreeMap<String, Integer> studentAges = new TreeMap<>();

        // Add key-value pairs to the TreeMap
        studentAges.put("Alice", 25);
        studentAges.put("Bob", 30);
        studentAges.put("Charlie", 22);
        studentAges.put("David", 28);

        // Access elements using get(key) method
        System.out.println("Alice's age: " + studentAges.get("Alice"));

        // Check if a key exists using containsKey(key)
        if (studentAges.containsKey("Charlie")) {
            System.out.println("Charlie is present in the map.");
        }

        // Iterate through all key-value pairs
        for (Map.Entry<String, Integer> entry : studentAges.entrySet()) {
            System.out.println(entry.getKey() + " : " + entry.getValue());
        }

        // Get the first and last key (sorted order in TreeMap)
        System.out.println("First student (by name): " + studentAges.firstKey());
        System.out.println("Last student (by name): " + studentAges.lastKey());

        // SubMap - get a range of key-value pairs
        SortedMap<String, Integer> twenties = studentAges.subMap("Bob", "David");
        System.out.println("Students in their twenties: " + twenties);

        // Remove a key-value pair
        studentAges.remove("Charlie");
        System.out.println("Map after removing Charlie: " + studentAges);
    }
}
```

Time Complexity

- O(1): Constant Time Complexity
- O(log n) Logarithmic Time Complexity
- O(n) Linear time Complexity
- O(n log n) n Logarithmic Time Complexity
- O(n^2) Quadratic Time Complexity
- O(n^3) Quadratic Time Complexity
- O(2^n) Exponential Time Complexity
- O(n!) Factorial Time Complexity

O(1): Constant Time Complexity

```
const firstElement = (array) => {
  return array[0];
};


let score = [12, 55, 67, 94, 22];
console.log(firstElement(score)); // 12
```

```
const firstElement = (array) => {
  for (let i = 0; i < array.length; i++) {
    return array[0];
  }
};

let score = [12, 55, 67, 94, 22];
console.log(firstElement(score)); // 12
```

O(log n) Logarithmic Time Complexity

O(n) Linear time Complexity

```
const calcFactorial = (n) => {
  let factorial = 1;
  for (let i = 2; i <= n; i++) {
    factorial = factorial * i;
  }
  return factorial;
};

console.log(calcFactorial(5)); // 120
```

O(n log n) n Logarithmic Time Complexity

O(log n) Logarithmic Time Complexity

```
const binarySearch = (array, target) => {
  let firstIndex = 0;
  let lastIndex = array.length - 1;
  while (firstIndex <= lastIndex) {
    let middleIndex = Math.floor((firstIndex + lastIndex) / 2);

    if (array[middleIndex] === target) {
      return middleIndex;
    }

    if (array[middleIndex] > target) {
      lastIndex = middleIndex - 1;
    } else {
      firstIndex = middleIndex + 1;
    }
  }
  return -1;
};

let score = [12, 22, 45, 67, 96];
console.log(binarySearch(score, 96));
```

O(n^2) Quadratic Time Complexity

```
const matchElements = (array) => {
  for (let i = 0; i < array.length; i++) {
    for (let j = 0; j < array.length; j++) {
      if (i !== j && array[i] === array[j]) {
        return `Match found at ${i} and ${j}`;
      }
    }
  }
  return "No matches found 😞";
};

const fruit = ["🍓", "🍐", "🍊", "🍌", "🍍", "🍑", "🍎", "🍈", "🍊", "🍇"];
console.log(matchElements(fruit)); // "Match found at 2 and 8"
```

O(n^3) Quadratic Time Complexity

O(2^n) Exponential Time Complexity

```
const recursiveFibonacci = (n) => {
  if (n < 2) {
    return n;
  }
  return recursiveFibonacci(n - 1) + recursiveFibonacci(n - 2);
};

console.log(recursiveFibonacci(6)); // 8
```

O(n!) Factorial Time Complexity

Time Complexity V/s Space Complexity

- The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.
- Problem-solving using computer requires memory to hold temporary data or final result while the program is in execution. The amount of memory required by the algorithm to solve given problem is called space complexity of the algorithm.

Time Complexity

```java
public class TimeComplexity {

    // Function to find a pair in the given array whose sum is equal to z
    static boolean findPair(int a[], int n, int z) {
        // Iterate through all the pairs
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                // Check if the sum of the pair (a[i], a[j]) is equal to z
                if (i != j && a[i] + a[j] == z)
                    return true;
        return false;
    }

    public static void main(String[] args) {
        // Given Input
        int a[] = {1, -2, 1, 0, 5};
        int z = 0;
        int n = a.length;
        // Function Call
        if (findPair(a, n, z))
            System.out.println("True");
        else
            System.out.println("False");
    }
}
```

Explanation

Assuming that each of the operations in the computer takes approximately constant time, let it be c. The number of lines of code executed actually depends on the value of Z. During analyses of the algorithm, mostly the worst-case scenario is considered, i.e., when there is no pair of elements with sum equals Z. In the worst case,

- N*c operations are required for input.
- The outer loop i loop runs N times.
- For each i, the inner loop j loop runs N times.

So total execution time is N*c + N*N*c + c. Now ignore the lower order terms since the lower order terms are relatively insignificant for large input, therefore only the highest order term is taken (without constant) which is N*N in this case. Different notations are used to describe the limiting behavior of a function, but since the worst case is taken so big-O notation will be used to represent the time complexity.

How to Calculate Time Complexity

- Identify the algorithm: Determine the specific algorithm or code snippet for which you want to calculate the time complexity. It could consist of a series of operations combined with a loop or a recursive function.
- Identify the input size: Identify the elements that make up the algorithm's input size. For example, if the algorithm operates on an array, the input size could be the length of the array.
- Determine the basic operations: Identify the fundamental operations that contribute to the running time of the algorithm. These operations could be comparisons, assignments, arithmetic operations, function calls, or any other significant actions.
- Count the operations: Analyze how many times each basic operation is executed as a function of the input size. You may need to consider different scenarios or branches within the algorithm.

How to Calculate Time Complexity

- Express the count as a function of the input size: Create a mathematical expression that represents the count of basic operations as a function of the input size. It ought to outline the worst-case scenario or the performance limit of the algorithm.
- Simplify the expression: Simplify the mathematical expression by removing constants, lower-order terms, and insignificant factors. Focus on the most dominant term that represents the growth rate of the algorithm.
- Determine the time complexity notation: Use Big O notation to express the condensed expression, where Big O indicates the time complexity's upper bound. O(1) stands for constant time, O(n) for linear time, O(n^2) for quadratic time, and so forth are common notations.

Lets Take and Example

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

it seems like the complexity is O(N * log N). N for the j′s loop and
But it’s wrong. Let’s see why.

- Think about how many times count++ will run.
- When i = N, it will run N times.
- When i = N / 2, it will run N / 2 times.
- When i = N / 4, it will run N / 4 times.
- And so on.

The total number of times count++ will run is N + N/2 + N/4+…+1= 2 * N. So the time complexity will be O(N).

Space Complexity

```java
public class SpaceComplexity {

    // Function to count frequencies of array items
    static void countFreq(int arr[], int n) {
        HashMap<Integer, Integer> freq = new HashMap<>();

        // Traverse through array elements and count frequencies
        for (int i = 0; i < n; i++) {
            if (freq.containsKey(arr[i])) {
                freq.put(arr[i], freq.get(arr[i]) + 1);
            } else {
                freq.put(arr[i], 1);
            }
        }

        // Traverse through map and print frequencies
        for (Map.Entry<Integer, Integer> x : freq.entrySet())
            System.out.print(x.getKey() + " " + x.getValue() + "\n");
    }

    // Driver Code
    public static void main(String[] args) {
        // Given array
        int arr[] = {10, 20, 20, 10, 10, 20, 5, 20};
        int n = arr.length;

        // Function Call
        countFreq(arr, n);
    }
}
```

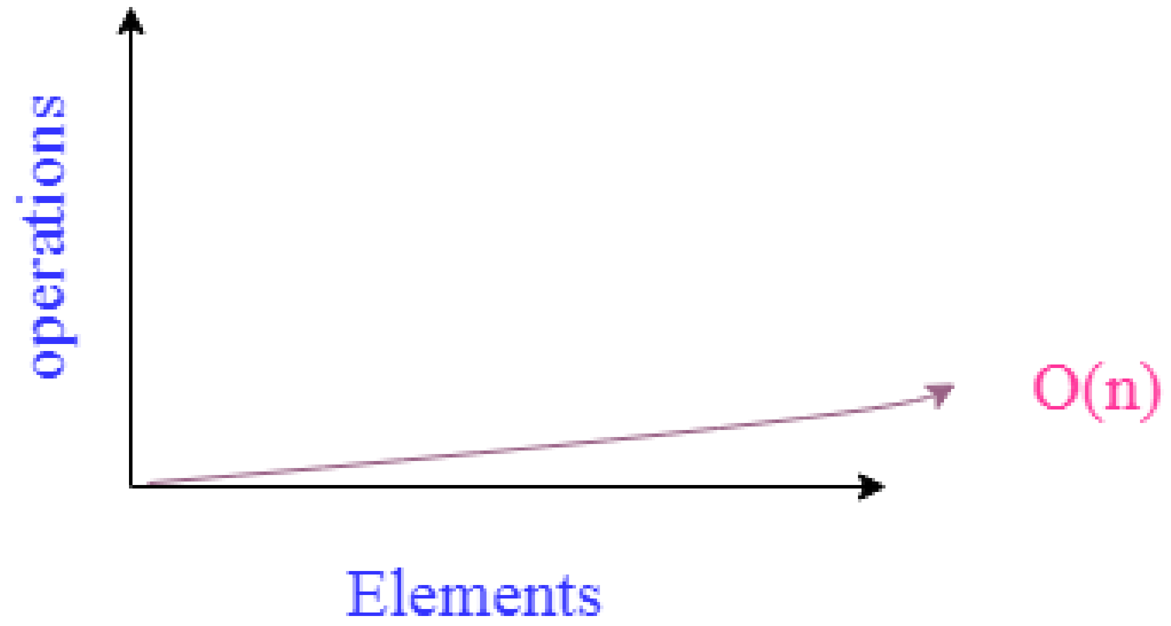
Explanation

- Two arrays of length N, and variable i are used in the algorithm so, the total space used is N * c + N * c + 1 * c = 2N * c + c, where c is a unit space taken.
- For many inputs, constant c is insignificant, and it can be said that the space complexity is O(N).
- There is also auxiliary space, which is different from space complexity. The main difference is where space complexity quantifies the total space used by the algorithm, auxiliary space quantifies the extra space that is used in the algorithm apart from the given input.
- In the above example, the auxiliary space is the space used by the freq[] array because that is not part of the given input. So total auxiliary space is N * c + c which is O(N) only.

General Time Complexities

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

Time Complexity

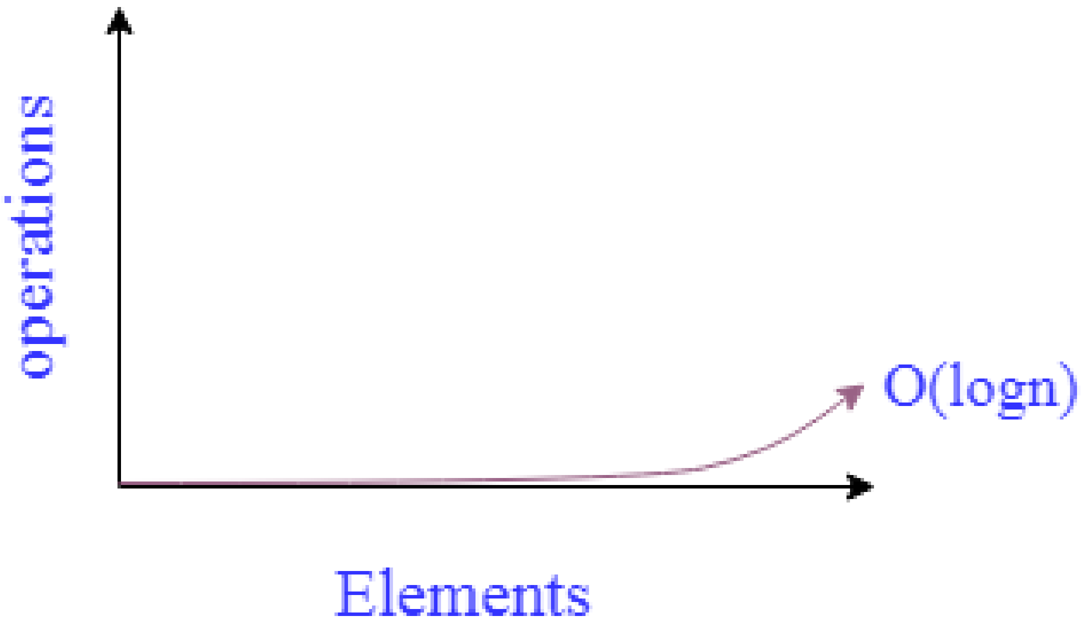


Time Complexity

```
int i = 1;
do
  {
      i++;
  }while(i<=n);
```

loop is executed 'n' times. Therefore, time complexity of this loop is O(n)

Time complexity of a loop when the loop variable is divided or multiplied by a constant amount

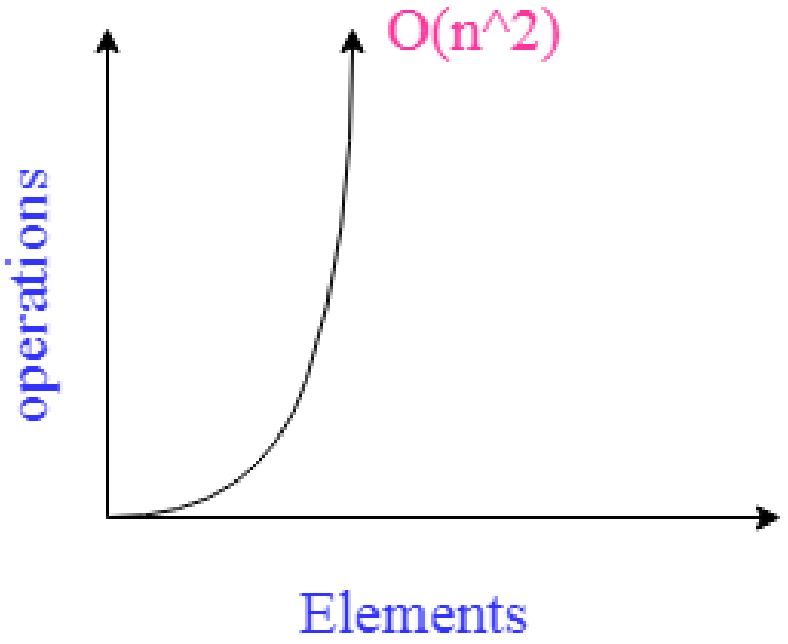


Time complexity

```
int i=1;
do
{
  i = i*c;
 }while(i<=n);
```

Time complexity is O(logn)

Time complexity of a nested loop



Time complexity of a nested loop

```
int i=0;
do{
    do{
          int j =0;
               j++;
       }while(j<=i);
     i++;
    }while(i<=n-1);
```

Time Complexity of Array

- Access: O(1)
- Search: O(n)
- Insertion (at the end): O(1)
- Insertion (at the beginning or middle): O(n)
- Deletion (from the end): O(1)
- Deletion (from the beginning or middle): O(n)

Applications of Arrays

- Dynamic Programming: Arrays are extensively used in dynamic programming to store intermediate results and optimize recursive algorithms. Dynamic programming algorithms like the Fibonacci series, matrix chain multiplication, and the knapsack problem rely on arrays to store and retrieve previously calculated values efficiently.
- Searching and Sorting: Arrays provide a foundation for searching and sorting algorithms. Common searching algorithms like binary search and sorting algorithms like quicksort, mergesort, and heapsort utilize arrays for efficient data manipulation.
- Implementing Other Data Structures: Arrays serve as the underlying data structure for implementing more complex structures such as stacks, queues, and hash tables.

Time and Space Complexity of Linked List

- The time complexity of the Linked List is O(n) for Accessing, insertion, or deletion at the beginning and searching for an element, where n is the number of elements.
- However, insertion and deletion at the beginning or end take O(1) time in a doubly linked list due to direct access to the head and tail nodes.
- Space complexity for both types of linked lists is O(n), as each element requires space for its data and pointers to the next (and possibly previous) node, resulting in linear space usage proportional to the number of elements.

Time and Space Complexity of Linked List

Operation	TC Singly Linked List	TC Doubly Linked List	Space Complexity
Accessing by Index	O(n)	O(n)	O(1)
Insertion at Beginning	O(1)	O(1)	O(1)
Insertion at End	O(n)	O(1)	O(1)
Insertion at Given Position	O(n)	O(n)	O(1)
Deletion at Beginning	O(1)	O(1)	O(1)
Deletion at End	O(n)	O(1)	O(1)
Deletion at Given Position	O(n)	O(n)	O(1)
Searching	O(n)	O(n)	O(1)

Applications of Linked List

- Memory Management: Linked lists play a vital role in memory management systems. They enable efficient allocation and deallocation of memory blocks by maintaining a linked structure that allows for easy insertion and deletion.
- Implementing Other Data Structures: Linked lists are fundamental in implementing other dynamic data structures such as stacks, queues, and hash tables.
- Polynomial Manipulation: In algebraic calculations, linked lists are used to represent and manipulate polynomials efficiently. Each node in the linked list represents a term in the polynomial, with its coefficient and exponent stored as data.

Time Complexity of Doubly Linked List

- Accessing an element by index: O(n)
- Searching for an element: O(n)
- Insertion (at the beginning): O(1)
- Insertion (at the end, with a tail pointer): O(1)
- Insertion (at the end, without a tail pointer): O(n)
- Insertion (in the middle): O(n)
- Deletion (from the beginning): O(1)
- Deletion (from the end, with a tail pointer): O(1)
- Deletion (from the end, without a tail pointer): O(n)
- Deletion (from the middle): O(n)

Time Complexity of Stack

- Push: O(1)
- Pop: O(1)
- Peek: O(1)

Applications of Stack

- Expression Evaluation and Conversion: Stacks are extensively used in evaluating and converting expressions. Infix to postfix conversion, postfix evaluation, and balancing parentheses are common applications of stacks in expression manipulation.
- Function Call Stack: Stacks are essential for managing function calls in programming languages. When a function is called, the function's local variables and return address are pushed onto the stack, allowing for proper execution and return flow.
- Backtracking Algorithms: Backtracking algorithms, such as depth-first search (DFS), rely on stacks to keep track of visited nodes and potential paths. The stack stores the state information required to backtrack and explore alternative paths.

Time Complexity of Queues

- Enqueue: O(1)
- Dequeue: O(1)
- Peek: O(1)

Applications of Queues

- Job Scheduling: Queues are used in operating systems and task management systems for job scheduling. The first-in-first-out (FIFO) nature of queues ensures fairness and proper execution order.
- Breadth-First Search (BFS) Algorithms: BFS algorithms explore graphs in a level-by-level manner, making queues an ideal data structure for maintaining the order of traversal.
- Printers' Job Management: In spooling systems, queues are employed to manage print jobs, ensuring that they are processed in the order they were received.

Time Complexity of Hash Table

- Search: O(1) - on average, assuming a good hash function and minimal collisions
- Insertion: O(1) - on average, assuming a good hash function and minimal collisions
- Deletion: O(1) - on average, assuming a good hash function and minimal collisions

Applications of Hash Table

- Database Indexing and Searching: Hash tables provide fast retrieval of data, making them suitable for indexing and searching in databases. Hash functions distribute data evenly across the table, allowing for efficient access.
- Caching Mechanisms: Hash tables are employed in caching mechanisms to store frequently accessed data, reducing the need for expensive computations or database queries.
- Symbol Tables: Compilers and interpreters utilize hash tables as symbol tables to store identifiers, keywords, and their associated attributes during the compilation and execution process.
- Key-Value Stores: Hash tables are the foundation for implementing key-value stores, where data is stored and retrieved based on unique keys.

Application of Trees

- File Systems: File systems utilize tree structures to represent directory hierarchies. Each node in the tree represents a directory, with child nodes representing subdirectories and files.
- Database Indexing: Trees are extensively used in database indexing for efficient searching and retrieval of records. B-tree and B+-tree structures are commonly employed to organize and store large volumes of data.
- Hierarchical Relationships: Trees are useful for representing hierarchical relationships in organizations, XML, and JSON data. They allow for efficient navigation and management of hierarchical data.
- Decision-Making Processes: Decision trees and game trees are employed in decision-making processes, such as machine learning algorithms and game AI, to model choices and outcomes.

Time Complexity of Binary Search Trees(BSTs)

- Search: O(log n) - on average for balanced BST, O(n) worst case for unbalanced BST
- Insertion: O(log n) - on average for balanced BST, O(n) worst case for unbalanced BST
- Deletion: O(log n) - on average for balanced BST, O(n) worst case for unbalanced BST

Time Complexity of AVL Tree

- Searching for an element: O(log n)
- Insertion of an element: O(log n)
- Deletion of an element: O(log n)

Time Complexity of B-Tree

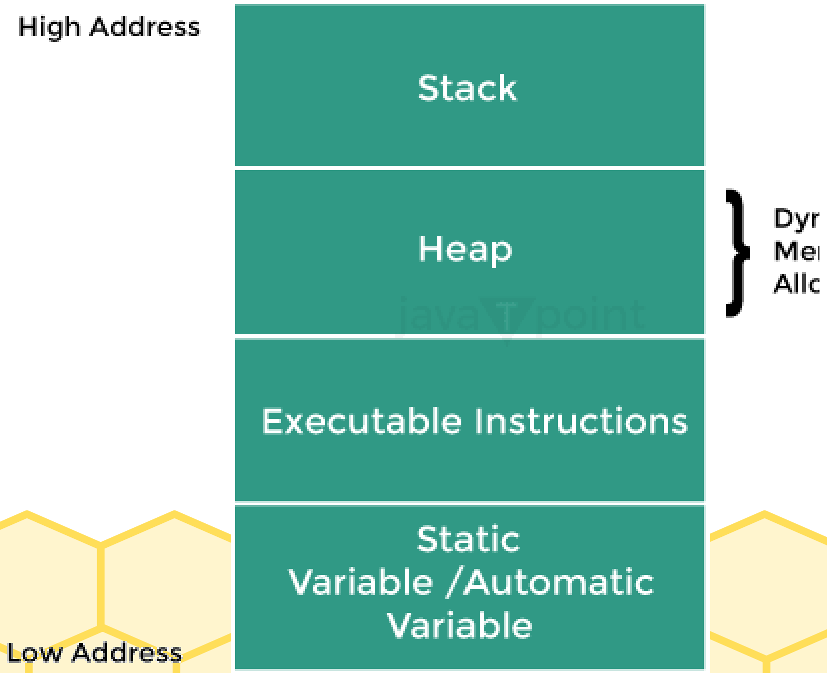
- Searching for an element: O(log n)
- Insertion of an element: O(log n)
- Deletion of an element: O(log n)

Time Complexity of Red-Black Tree

- Searching for an element: O(log n)
- Insertion of an element: O(log n)
- Deletion of an element: O(log n)

Dynamic Memory Allocation

- Memory can be classified into two categories: stack memory and heap memory.
- Local variables and function calls are stored in the stack memory, whereas the more adaptable heap memory can be allocated and released at runtime.



Data Structures Utilizing Dynamic Memory Allocation

- Arrays
- Linked Lists
- Trees
- Graphs: Graphs are structures composed of vertices (nodes) connected by edges.
- Graphs can be dynamic, with vertices and edges added or removed as needed.
- Dynamic memory allocation allows for efficient memory management when creating and managing vertices and edges dynamically.

Applications of Graphs

- Social Network Analysis: Graphs are used to model and analyze social networks, enabling applications such as friend recommendations, community detection, and influence analysis.
- Network Routing Algorithms: Graphs are essential in network routing algorithms, determining the shortest or optimal path between nodes. Dijkstra's algorithm and Bellman-Ford algorithm rely on graphs for efficient routing.
- Web Page Ranking: Graph-based algorithms like Google's PageRank employ graphs to rank web pages based on their importance and connectivity within the web graph.
- Bioinformatics and Computational Biology: Graphs are utilized to model and analyze biological networks, such as protein-protein interaction networks and gene regulatory networks.

Graphs

- Graphs are a common way to present non-linear data.
- It consists of vertices, graph units, (vertex or nodes), and edges (connective pathways between nodes).

Graphs: Adding a Vertex

```
package graphs;

import java.util.ArrayList;
import java.util.List;

public class Graph {
    private int numVertices;
    private List<List<Integer>> adjacencyList;

    public Graph(int numVertices) {
        this.numVertices = numVertices;
        adjacencyList = new ArrayList<>(numVertices);

// Initialize the adjacency list
        for (int i = 0; i < numVertices; i++) {
            adjacencyList.add(new ArrayList<>());
        }
    }

    public void addVertex() {
        numVertices++;
        adjacencyList.add(new ArrayList<>());
    }
}
```

Graphs: Adding an Edge

```
package graphs;

import java.util.ArrayList;
import java.util.List;

public class Graph {
    private int numVertices;
    private List<List<Integer>> adjacencyList;

    public Graph(int numVertices) {
        this.numVertices = numVertices;
        adjacencyList = new ArrayList<>(numVertices);

// Initialize the adjacency list
        for (int i = 0; i < numVertices; i++) {
            adjacencyList.add(new ArrayList<>());
        }
    }

    public void addEdge(int source, int destination) {
// Check if the vertices are within the valid range
        if (source >= 0 && source < numVertices && destination >= 0 && destination <
numVertices) {
// Add the destination vertex to the adjacency list of the source vertex
            adjacencyList.get(source).add(destination);

// If the graph is undirected, add the source vertex to the adjacency list of the
destination vertex as well adjacencyList.get(destination).add(source); // Uncomment this
line for an undirected graph
        }
    }
}
```

Graph Traversal

- Graph searches are performed when each vertex is visited for checkups or updates.
- Depending on the type of issue, two iterations can perform this.
- Breadth-first traversal (BFS) is often implemented in a queue data structure.
- BFS starts at a given node (usually the root) and explores its adjacent nodes, then moves on to the next level of nodes until all nodes have been visited.
- BFS is typically implemented using a queue data structure.

Graph

```java
package graphs;

import java.util.LinkedList;
import java.util.Queue;

class Graph {
    private int numVertices;
    private LinkedList<Integer>[] adjacencyList;

    public Graph(int numVertices) {
        this.numVertices = numVertices;
        adjacencyList = new
LinkedList[numVertices];
        for (int i = 0; i < numVertices; i++) {
            adjacencyList[i] = new LinkedList<>();
        }
    }

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

    public void breadthFirstTraversal(int
startVertex) {
        boolean[] visited = new
boolean[numVertices];
        Queue<Integer> queue = new LinkedList<>();

        visited[startVertex] = true;
        queue.offer(startVertex);
```

```java
while (!queue.isEmpty()) {
            int currentVertex = queue.poll();
            System.out.print(currentVertex + " ");

            for (int neighbor :
adjacencyList[currentVertex]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.offer(neighbor);
                }
            }
        }
    }
}

public class GraphDemo {
    public static void main(String[] args) {
        Graph graph = new Graph(6);
        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 3);
        graph.addEdge(2, 4);
        graph.addEdge(3, 4);
        graph.addEdge(3, 5);

        System.out.println("Breadth-First
Traversal:");
        graph.breadthFirstTraversal(0);
    }
}
```

Graph Traversal Explanation

- In depth-first traversal (DFS) explores the graph using recursion or a stack data structure to go as far as possible along each branch before backtracking.
- It starts at a given node (usually the root) and explores as deeply as possible before backtracking and visiting other adjacent nodes.

Graph Traversal

```java
package graphs;

import java.util.LinkedList;

class Graph {
    private int numVertices;
    private LinkedList<Integer>[] adjacencyList;

    public Graph(int numVertices) {
        this.numVertices = numVertices;
        adjacencyList = new LinkedList[numVertices];
        for (int i = 0; i < numVertices; i++) {
            adjacencyList[i] = new LinkedList<>();
        }
    }

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

    public void depthFirstTraversal(int startVertex) {
        boolean[] visited = new boolean[numVertices];
        dfsHelper(startVertex, visited);
    }

    private void dfsHelper(int vertex, boolean[] visited) {
        visited[vertex] = true;
        System.out.print(vertex + " ");

        for (int neighbor : adjacencyList[vertex]) {
            if (!visited[neighbor]) {
                dfsHelper(neighbor, visited);
            }
        }
    }
}
```

```java
public class GraphDemo {
    public static void main(String[] args) {
        Graph graph = new Graph(6);
        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 3);
        graph.addEdge(2, 4);
        graph.addEdge(3, 4);
        graph.addEdge(3, 5);

        System.out.println("Depth-First
Traversal:");
        graph.depthFirstTraversal(0);
    }
}
```

Time complexity of a nested loop

in each iteration of i, inner loop is executed 'n' times. The time complexity of a loop is equal to the number of times the innermost statement is to be executed.

On the first iteration of i=0, the inner loop executes 0 times.

On the first iteration of i=1, the inner loop executes 1 times.

On the first iteration of i=n-1, the inner loop executes n-1 times.

The number of times the inner loop is executed is equal to the sum of integers(0,1,2,3?n-1) = n^2/2 - n/2

Time complexity = O(n2).

Time complexity of an infinite loop

- Infinite loop is executed "Infinite times".
- Therefore, there is no "algorithm time complexity" for an infinite loop.

Time complexities of different loops

```
int i=1;
do{
    i++;
  }while(i<=m);

int j=1;
do{
    j++;
  }while(j<=n);
```

Time complexity of different loops is equal to the sum of the complexities of individual loop. Therefore, Time complexity = O(m)+O(n)

Scenario

To read two iterators and need to add them to list and return that list here what is "The function should operate in O(1) time".

If my understanding is correct, if array has 1 element and the process time should take 1 sec and if it has 100 elements then also it should take 1 sec...

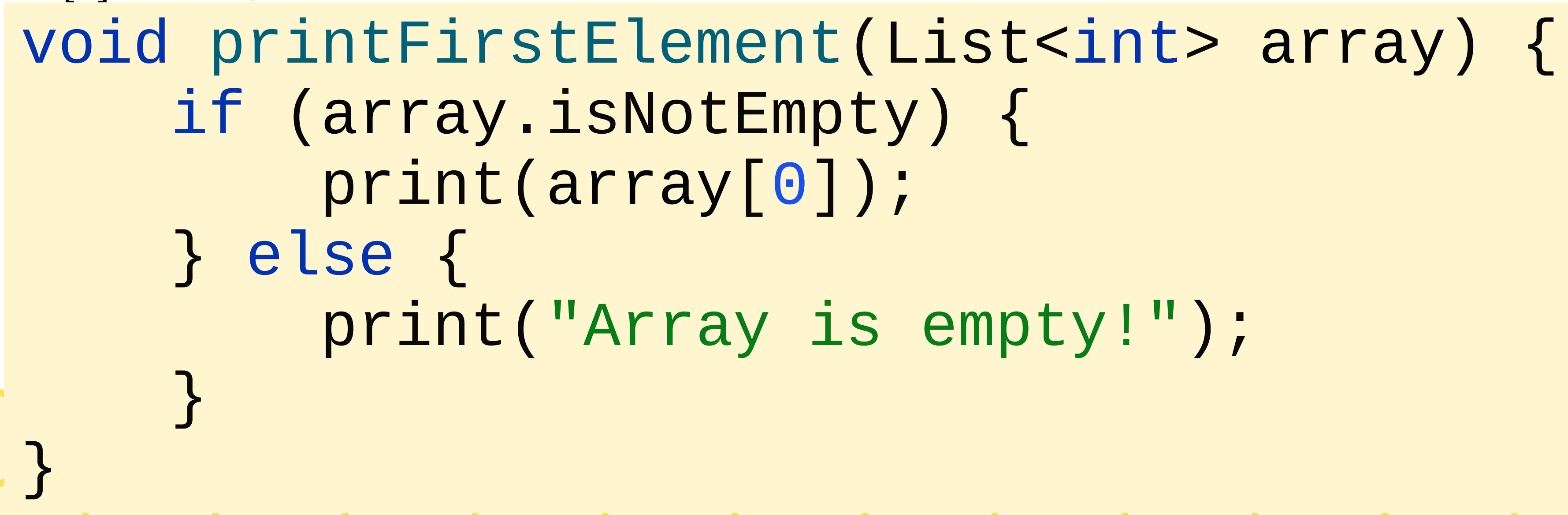
how can i achieve the 1 sec algorithm here for above read and add to new list operation

O(1) Constant Time Complexity

O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set. O(1) time complexity is also called constant time complexity

Independent of List Size, always return first element.

```
boolean IsFirstElementNull(List<string> elements)
{
    return elements[0] == null;
}
```



```
void printFirstElement(List<int> array) {
    if (array.isNotEmpty) {
        print(array[0]);
    } else {
        print("Array is empty!");
    }
}
```

O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.

O(N) Linear Time Complexity

```
boolean ContainsValue(List<string> elements, string value)
{
    foreach (var element in elements)
    {
        if (element == value) return true;
    }

    return false;
}
```

```
void printArrayElements(List<int> array) {
    for (int i = 0; i < array.length; i++) {
        print(array[i]);
    }
}
```

Quadratic Time Complexity (O(n²))

- Quadratic time complexity occurs when the running time grows proportionally to the square of the input size.
- It is often observed in algorithms with nested loops, and the execution time increases rapidly with larger inputs.
- An example is the selection sort algorithm.

Quadratic Time Complexity (O(n²))

```
void selectionSort(List<int> array) {
    final length = array.length;
    for (int i = 0; i < length - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < length; j++) {
            if (array[j] < array[minIndex]) {
                minIndex = j;
            }
        }
        if (minIndex != i) {
            int temp = array[i];
            array[i] = array[minIndex];
            array[minIndex] = temp;
        }
    }
}
```

Cubic Time Complexity ($O(n^3)$)

- Cubic time complexity refers to algorithms where the running time grows proportionally to the cube of the input size.
- These algorithms often involve three levels of iteration or computations, and they become highly inefficient for larger inputs.

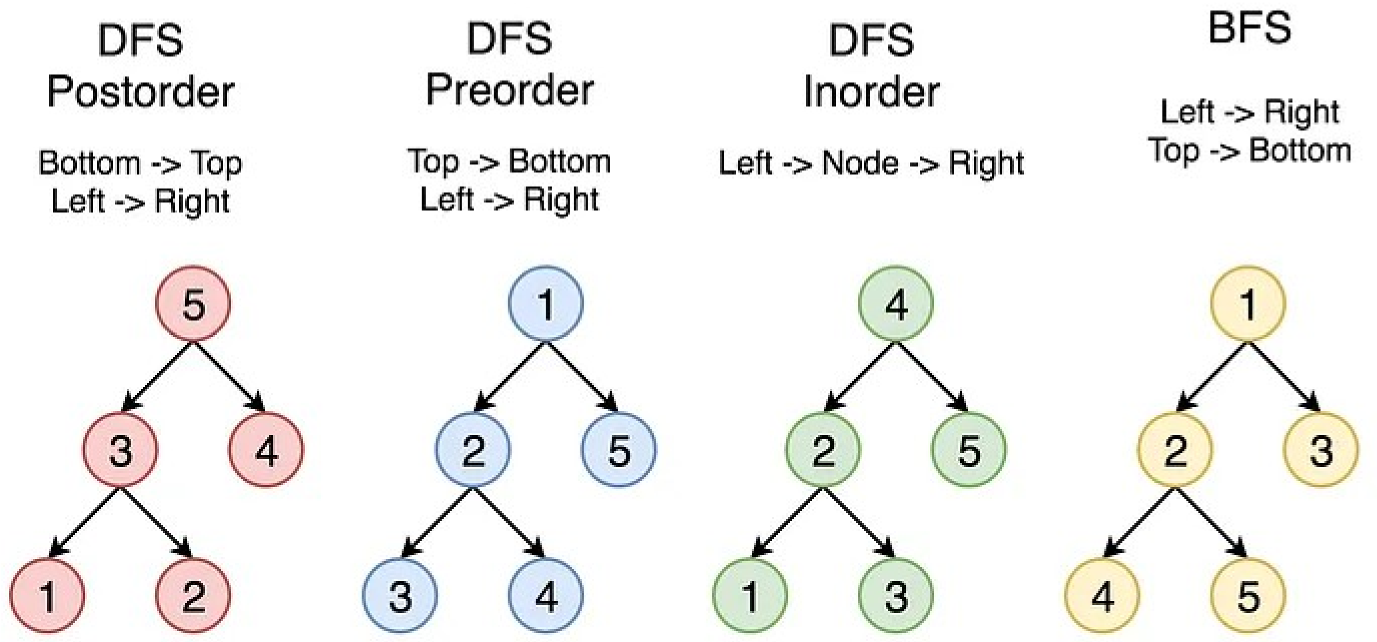
Cubic Time Complexity (O(n³))

```
void cubicAlgorithm(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                // Perform some computations
                print('Computation & iteration $i, $j, 
$k');
            }
        }
    }
}
```

Binary Tree Traversal (O(n))

- The time complexity of traversing a binary tree depends on the traversal type. In-order, pre-order, and post-order traversals all have a time complexity of O(n), where n is the number of nodes in the tree.
- These traversals

visit each node once.



```
void inOrderTraversal(TreeNode node) {
    if (node == null) return;
    inOrderTraversal(node.left);
    print(node.val);
    inOrderTraversal(node.right);
}
```

In-order Traversal

Pre-order Traversal

```
void preOrderTraversal(TreeNode node) {
    if (node == null) return;
    print(node.val);
    preOrderTraversal(node.left);
    preOrderTraversal(node.right);
}
```

```
void postOrderTraversal(TreeNode node)
{
    if (node == null) return;
    postOrderTraversal(node.left);
    postOrderTraversal(node.right);
    print(node.val);
}
```

Post-order Traversal

Loglinear Time Complexity (O(n log n))

- Loglinear time complexity, often denoted as O(n log n), is commonly observed in efficient sorting and searching algorithms like Merge Sort and QuickSort.
- It combines elements of linear (O(n)) and logarithmic (O(log n)) time complexities.
- As the input size increases, the running time grows at a rate that is proportional to n multiplied by the logarithm of n.
- This complexity is more efficient than quadratic time complexity (O(n^2)), but not as efficient as linear time complexity (O(n)).

Loglinear Time Complexity (O(n log n))

```
List<int> mergeSort(List<int> array) {
    if (array.length <= 1) {
        return array;
    }

    final mid = array.length ~/ 2;
    final left = array.sublist(0, mid);
    final right = array.sublist(mid);

    final sortedLeft = mergeSort(left);
    final sortedRight = mergeSort(right);

    return merge(sortedLeft, sortedRight);
}
```

Logarithmic Time Complexity (O(log n))

- Logarithmic time complexity is often seen in algorithms that divide the input in half at each step, such as binary search.
- In these algorithms, the input size is effectively reduced by half with each iteration.
- As a result, the running time grows very slowly even for large inputs.
- The binary search algorithm divides the search space in half repeatedly until the target element is found or the search space is exhausted.
- This results in a time complexity of O(log n), where n is the size of the input array.

Logarithmic Time Complexity (O(log n))

```
int binarySearch(List<int> sortedArray, int target) {
    int left = 0;
    int right = sortedArray.length - 1;

    while (left <= right) {
        int mid = left + ((right - left) ~/ 2);

        if (sortedArray[mid] == target) {
            return mid;
        } else if (sortedArray[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}
```

Conceptual

- What is Time Complexity O(1), O(N), O(logn), O(n^2), Time complexity of an infinite loop

Find a pair with the given sum in an array using Brute-Force, Sorting, Hashing

Input

- nums = [8, 7, 2, 5, 3, 1]
- target = 10

Output

- Pair found (8, 2)
- or
- Pair found (7, 3)

Input

nums = [5, 2, 6, 8, 1, 9]
target = 12

Output: Pair not found

Find a triplet with the given sum in an array using Brute-Force, Sorting, Hashing, Naive Recursive

Input:

- nums = [2, 7, 4, 0, 9, 5, 1, 3]
- target = 6

Output: Triplet exists

- The triplets with the given sum 6 are {0, 1, 5}, {0, 2, 4}, {1, 2, 3}

Printing distinct triplets using Brute-Force, Sorting, Hashing, Naive Recursive

Sort the given array in ascending order, and for each element nums[i] in the array, check if the triplet is formed by nums[i] and a pair from subarray nums[i+1…n)

Find a pair with the given sum in an array

- Input:
- nums = [8, 7, 2, 5, 3, 1]
- target = 10

Output:

- Pair found (8, 2)
- or
- Pair found (7, 3)

Input:

nums = [5, 2, 6, 8, 1, 9]
target = 12

Output: Pair not found

Using Brute-Force (Python)

```
# Naive method to find a pair in a list with the given sum
def findPair(nums, target):
    # consider each element except the last
    for i in range(len(nums) - 1):
        # start from the i'th element until the last element
        for j in range(i + 1, len(nums)):
            # if the desired sum is found, print it
            if nums[i] + nums[j] == target:
                print('Pair found', (nums[i], nums[j]))
                return
    # No pair with the given sum exists in the list
    print('Pair not found')


if __name__ == '__main__':
    nums = [8, 7, 2, 5, 3, 1]
    target = 10
    findPair(nums, target)
```

Using Brute-Force(Java)

```
public static void findPair(int[] nums, int target)
 {
    // consider each element except the last
    for (int i = 0; i < nums.length - 1; i++)
    {
        // start from the i'th element until the last element
        for (int j = i + 1; j < nums.length; j++)
        {
            // if the desired sum is found, print it
            if (nums[i] + nums[j] == target)
            {
                System.out.println("Pair found (" + nums[i] + "," + nums[j] + ")");
                return;              }}}
    // we reach here if the pair is not found
    System.out.println("Pair not found");
 }
 public static void main (String[] args)
 {
    int[] nums = { 8, 7, 2, 5, 3, 1 };
    int target = 10;

    findPair(nums, target);
 }}
```

Using Brute-Force(C Lang)

```
void findPair(int nums[], int n, int target)
{
    // consider each element except the last
    for (int i = 0; i < n - 1; i++)
    {
        // start from the i'th element until the last element
        for (int j = i + 1; j < n; j++)
        {
            // if the desired sum is found, print it
            if (nums[i] + nums[j] == target)
            {
                printf("Pair found (%d, %d)\n", nums[i], nums[j]);
                return;
            } }    }
    // we reach here if the pair is not found
    printf("Pair not found");
}
int main(void)
{
    int nums[] = { 8, 7, 2, 5, 3, 1 };
    int target = 10;
    int n = sizeof(nums)/sizeof(nums[0]);
    findPair(nums, n, target);
    return 0;}
```

Using Sorting(C Lang)

- sort the given array in ascending order and maintain search space by maintaining two indices (low and high) that initially points to two endpoints of the array.
- Then reduce the search space nums[low…high] at each iteration of the loop by comparing the sum of elements present at indices low and high with the desired sum.
- Increment low if the sum is less than the expected sum; otherwise, decrement high if the sum is more than the desired sum. If the pair is found, return it.

Using Sorting(C Lang)

```
void findPair(int nums[], int n, int target)
{
    // sort the array in ascending order
    sort(nums, nums + n);
    // maintain two indices pointing to endpoints of the array
    int low = 0;
    int high = n - 1;
    // reduce the search space `nums[low…high]` at each iteration of the
loop
    // loop till the search space is exhausted
    while (low < high)
    {
        // sum found
        if (nums[low] + nums[high] == target)
        {
            cout << "Pair found (" << nums[low] << ", " << nums[high] << ")\
n";
            return;
        }
        // increment `low` index if the total is less than the
desired sum;
        // decrement `high` index if the total is more than
the desired sum
        (nums[low] + nums[high] < target)? low++: high--;
    }

    // we reach here if the pair is not found
    cout << "Pair not found";
}

int main()
{
    int nums[] = { 8, 7, 2, 5, 3, 1 };
    int target = 10;

    int n = sizeof(nums)/sizeof(nums[0]);

    findPair(nums, n, target);

    return 0;
}
```

Using Sorting(Java)

```
public static void findPair(int[] nums, int target)

{

    // sort the array in ascending order

    Arrays.sort(nums);



    // maintain two indices pointing to endpoints of the array

    int low = 0;

    int high = nums.length - 1;



    // reduce the search space `nums[low…high]` at each iteration of the loop



    // loop till the search space is exhausted

    while (low < high)

    {

        // sum found

        if (nums[low] + nums[high] == target)

        {

            System.out.println("Pair found (" + nums[low] + "," +

                                nums[high] + ")");

            return;

        }
```

```
            if (nums[low] + nums[high] < target) {

                low++;

            }

            else {

                high--;

            }

        }

        // we reach here if the pair is not found

        System.out.println("Pair not found");

    }

    public static void main (String[] args)

    {

        int[] nums = { 8, 7, 2, 5, 3, 1 };

        int target = 10;

        findPair(nums, target);

    }}
```

Using Sorting(Python)

```
# Function to find a pair in an array with a given sum using
sorting
def findPair(nums, target):
    # sort the list in ascending order
    nums.sort()
    # maintain two indices pointing to endpoints of the list
    (low, high) = (0, len(nums) - 1)
    # reduce the search space `nums[low…high]` at each
iteration of the loop
    # loop till the search space is exhausted
    while low < high:
        if nums[low] + nums[high] == target:        # target found
            print('Pair found', (nums[low], nums[high]))
            return
        # increment `low` index if the total is less than the desired
sum;
        # decrement `high` index if the total is more than the
desired sum
```

```
 if nums[low] + nums[high] < target:
            low = low + 1
        else:
            high = high - 1
    # No pair with the given sum exists
    print('Pair not found')
if __name__ == '__main__':
    nums = [8, 7, 2, 5, 3, 1]
    target = 10
    findPair(nums, target)
```

Using Hashing(Python)

- a hash table to solve this problem in linear time.
- The idea is to insert each array element nums[i] into a map.
- We also check if difference (nums[i], target - nums[i]) already exists in the map or not. If the difference is seen before, print the pair and return.

Using Hashing(Python)

```
# Function to find a pair in an array with a given sum using hashing
def findPair(nums, target):
    # create an empty dictionary
    d = {}
    # do for each element
    for i, e in enumerate(nums):
        # check if pair (e, target - e) exists
        # if the difference is seen before, print the pair
        if target - e in d:
            print('Pair found', (nums[d.get(target - e)], nums[i]))
            return
        # store index of the current element in the dictionary
        d[e] = i
    # No pair with the given sum exists in the list
    print('Pair not found')
if __name__ == '__main__':
    nums = [8, 7, 2, 5, 3, 1]
    target = 10
    findPair(nums, target)
```

Using Hashing(Java)

```
class Main

{

    // Function to find a pair in an array with a given sum using hashing

    public static void findPair(int[] nums, int target)

    {

        // create an empty HashMap

        Map<Integer, Integer> map = new HashMap<>();

        // do for each element

        for (int i = 0; i < nums.length; i++)

        {

            // check if pair (nums[i], target-nums[i]) exists


            // if the difference is seen before, print the pair

            if (map.containsKey(target - nums[i]))

            {

                System.out.printf("Pair found (%d, %d)",

                    nums[map.get(target - nums[i])], nums[i]);

                return;

            }
```

```
            // store index of the current element i
            map.put(nums[i], i);
        }
        // we reach here if the pair is not found
        System.out.println("Pair not found");
    }
    public static void main (String[] args)
    {
        int[] nums = { 8, 7, 2, 5, 3, 1 };
        int target = 10;

        findPair(nums, target);
    }
}
```

Using Hashing(C++)

```
#include <iostream>
#include <unordered_map>
using namespace std;

// Function to find a pair in an array with a given sum using hashing
void findPair(int nums[], int n, int target)
{
    // create an empty map
    unordered_map<int, int> map;

    // do for each element
    for (int i = 0; i < n; i++)
    {
        // check if pair (nums[i], target - nums[i]) exists

        // if the difference is seen before, print the pair
        if (map.find(target - nums[i]) != map.end())
        {
            cout << "Pair found (" << nums[map[target - nums[i]]] << ", "
                 << nums[i] << ")\n";
            return;
        }
        // store index of the current element in the map
        map[nums[i]] = i;
    }
    // we reach here if the pair is not found
    cout << "Pair not found";
}
int main()
{
    int nums[] = { 8, 7, 2, 5, 3, 1 };
    int target = 10;
    int n = sizeof(nums)/sizeof(nums[0]);
    findPair(nums, n, target);
    return 0;
}
```

Find a triplet with the given sum in an array

Given an unsorted integer array, find a triplet with a given sum in it

Example

- Input:

- nums = [2, 7, 4, 0, 9, 5, 1, 3]
- target = 6

- Output: Triplet exists.

- The triplets with the given sum 6 are {0, 1, 5}, {0, 2, 4}, {1, 2, 3}

Naive Recursive Approach

- Similar to the 0–1 Knapsack problem and uses recursion.

- We either consider the current item or exclude it and recur for the remaining elements for each item.

- Return true if we get the desired sum by including or excluding the current item.

Naive Recursive Approach(C++)

```
#include <iostream>
using namespace std;

// Naive recursive function to check if triplet exists in an array
// with the given sum
bool isTripletExist(int nums[], int n, int target, int count)
{
    // if triplet has the desired sum, return true
    if (count == 3 && target == 0) {
        return true;
    }

    // return false if the sum is not possible with the current configuration
    if (count == 3 || n == 0 || target < 0) {
        return false;
    }
```

Naive Recursive Approach(C++)

```
 // recur with including and excluding the current element
    return isTripletExist(nums, n - 1, target - nums[n - 1], count + 1) ||
            isTripletExist(nums, n - 1, target, count);
}
int main()
{
    int nums[] = { 2, 7, 4, 0, 9, 5, 1, 3 };
    int target = 6;
    int n = sizeof(nums) / sizeof(nums[0]);
    isTripletExist(nums, n, target, 0) ? cout << "Triplet exists":
                                cout << "Triplet doesn't exist";
    return 0;
}
```

Naive Recursive Approach(Java)

```
class Main

{

    // Naive recursive function to check if triplet exists in an array

    // with the given sum

    public static boolean isTripletExist(int[] nums, int n, int target, int count)

    {

        // if triplet has the desired sum, return true

        if (count == 3 && target == 0) {

            return true;

        }


        // return false if the sum is not possible with the current configu

        if (count == 3 || n == 0 || target < 0) {

            return false;

        }


        // recur with including and excluding the current element

        return isTripletExist(nums, n - 1, target - nums[n - 1], count + 1) ||

                    isTripletExist(nums, n - 1, target, count);

    }
```

```
    public static void main(String[] args)
    {
        int[] nums = { 2, 7, 4, 0, 9, 5, 1, 3 };
        int target = 6;

        if (isTripletExist(nums, nums.length, target, 0)) {
            System.out.println("Triplet exists");
        }
        else {
            System.out.println("Triplet doesn't exist");
        }
    }
}
```

Naive Recursive Approach(Python)

```
# Naive recursive function to check if triplet exists in a list
# with the given sum
def isTripletExist(nums, n, target, count):
    # if triplet has the desired sum, return true
    if count == 3 and target == 0:
        return True
    # return false if the sum is not possible with the current configuration
    if count == 3 or n == 0 or target < 0:
        return False
    # recur with including and excluding the current element
    return isTripletExist(nums, n - 1, target - nums[n - 1], count + 1) or\
        isTripletExist(nums, n - 1, target, count)
if __name__ == '__main__':
    nums = [2, 7, 4, 0, 9, 5, 1, 3]
    target = 6
    if isTripletExist(nums, len(nums), target, 0):
        print('Triplet exists')
    else:
        print('Triplet doesn\'t exist')
```

Using Hashing

- The idea is to insert each array element into a hash table.
- Then consider all pairs present in the array and check if the remaining sum exists in the map or not.
- If the remaining sum is seen before and triplet doesn’t overlap with each other, i.e., (i, j, i) or (i, j, j), print the triplet and return.

Hashing(Python)

```python
# Function to check if triplet exists in a list with the given sum
def isTripletExist(nums, target):
    # create an empty dictionary
    d = {}
    # insert (element, index) pair into the dictionary for each input element
    for i, e in enumerate(nums):
        d[e] = i
    # consider each element except the last element
    for i in range(len(nums) - 1):

        # start from the i'th element until the last element
        for j in range(i + 1, len(nums)):
            # remaining sum
            val = target - (nums[i] + nums[j])

            # if the remaining sum is found, we have found a triplet
            if val in d:
                # if the triplet doesn't overlap, return true
                if d[val] != i and d[val] != j:
                    return True

    # return false if triplet doesn't exist
    return False


if __name__ == '__main__':

    nums = [2, 7, 4, 0, 9, 5, 1, 3]
    target = 6

    if isTripletExist(nums, target):
        print('Triplet exists')
    else:
        print('Triplet doesn\'t exist')
```

Hashing(C++)

```
#include <iostream>
#include <algorithm>
#include <unordered_map>
using namespace std;

// Function to check if triplet exists in an array with the given sum
bool isTripletExist(int nums[], int n, int target)
{
    // create an empty map
    unordered_map<int, int> map;

    // insert (element, index) pair into the map for each array element
    for (int i = 0; i < n; i++) {
        map[nums[i]] = i;
    }

    // consider each element except the last element
    for (int i = 0; i < n - 1; i++)
    {
        // start from the i'th element until the last element
        for (int j = i + 1; j < n; j++)
        {
            // remaining sum
            int val = target - (nums[i] + nums[j]);
```

```
 // if the remaining sum is found, we have found a triplet
            if (map.find(val) != map.end())
            {
                // if the triplet doesn't overlap, return true
                if (map[val] != i && map[val] != j) {
                    return true;
                }}}}
    // return false if triplet doesn't exist
    return false;
}
int main()
{
    int nums[] = { 2, 7, 4, 0, 9, 5, 1, 3 };
    int target = 6;
    int n = sizeof(nums) / sizeof(nums[0]);
    isTripletExist(nums, n, target) ? cout << "Triplet exists":
                            cout << "Triplet Don't Exist";
    return 0;}
```

Printing distinct triplets

- sort the given array in ascending order, and for each element nums[i] in the array, check if the triplet is formed by nums[i] and a pair from subarray nums[i+1…n).

Hashing(C++)

```
#include <iostream>
#include <algorithm>
#include <unordered_map>
using namespace std;

// Function to check if triplet exists in an array with the given sum
bool isTripletExist(int nums[], int n, int target)
{
    // create an empty map
    unordered_map<int, int> map;

    // insert (element, index) pair into the map for each array element
    for (int i = 0; i < n; i++) {
        map[nums[i]] = i;
    }

    // consider each element except the last element
    for (int i = 0; i < n - 1; i++)
    {
        // start from the i'th element until the last element
        for (int j = i + 1; j < n; j++)
        {
            // remaining sum
            int val = target - (nums[i] + nums[j]);
```

```
 // if the remaining sum is found, we have found a triplet
          if (map.find(val) != map.end())
          {
              // if the triplet doesn't overlap, return true
              if (map[val] != i && map[val] != j) {
                  return true;
              }}}}
    // return false if triplet doesn't exist
    return false;
}
int main()
{
    int nums[] = { 2, 7, 4, 0, 9, 5, 1, 3 };
    int target = 6;
    int n = sizeof(nums) / sizeof(nums[0]);
    isTripletExist(nums, n, target) ? cout << "Triplet exists":
                                   cout << "Triplet Don't Exist";
    return 0;}
```