2023

DSA
PART-2

OBZTRON

C++

(THIRD PART)

T.I.M. HAMIM

Index: DSA < Part - 2 >

Index: DSA < Part - 2 >

1. Maths for DSA

Introduction :

2. Recursion

Recursion is a programming technique in which a function calls itself in order to solve a problem. Recursive functions can be used to solve problems that can be broken down into smaller subproblems of the same type. When writing recursive functions, it's important to define base cases that specify when the recursion should stop and when the function should start returning values.

- It helps us in solving bigger/ complex problems in a simple way.
- You can convert recursive solution into iteration and vice versa.
- Space complexity is not constant because of recursive calls.
- It helps us in breaking down bigger problems into smaller problems.

```
void recursion()
{
   recursion(); /* function calls itself */
}

int main()
{
   recursion();
}
```

- The C programming language supports recursion, i.e., a function to call itself.
- But while using recursion, programmers need to be careful to define an exit condition (base condition) from the function, otherwise it will go in infinite loop.
- Recursive function are very useful to solve many mathematical problems like to calculate factorial of a number, generating Fibonacci series, etc.

when to use Recursion :

- Identify if you can break the problem into smaller problems.
- Write the recursive relation if needed.
- Draw the recursive tree.
- About the tree :

1. See the flow of functions, how they are getting in stack.
2. Identify and focus on left tree walls and right walls.

- See how the values and what types of values are returned at each step.
- See where the function call will come out of the main function
- Make sure to return the result of a function call 0f the return type

Variables using recursion:

1. Arguments
2. Return type
3. Body of the function

Types of recurrence relation:

1. Linear recurrence relation (Fibonacci number),
2. Divide and conquer recurrence relation (Binary search)
   (Search space reduce by a factor)

1. Linear recurrence relation:
    1. Factorial
    2. Fibonacci Number

2. Divide and Conquer:
    1. Binary Search
    2. Finding Maximum and Minimum
    3. Merge Sort
    4. Quick Sort
    5. Strassem's Matrix Multiplication

- **Iteration vs Recursion :**

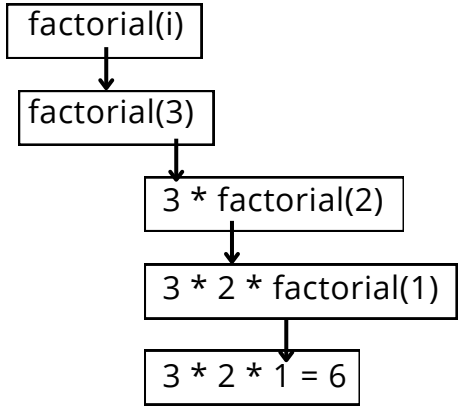
Recursion	**Iteration**
1.Use selection structure(if, else, swtich)	1.Use repetition structure(for, while, do while)
2.Terminates when base case is satisfied.	2.Terminates whene condition fails.
3.It's slow	3.It's fast
4.Code is smaller	4.Code is bigger

- **Make a program which will calculates factorial for a given number using a recursive function.**

```
#include <iostream>

using namespace std;

int factorial(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}

int main() {
  int num = 3;
  int result = factorial(num);
  cout << "Factorial of " << num << " is " << result <<
endl;
  return 0;
}
```

Output:
Factorial of 3 is 6

Here,



Here,

- This is called tail recursion
- This is the last function call

- **What is the output of the following program fragment:**

```
#include <iostream>
#include <string>

using namespace std;

string message(int a) {
  if (a == 1) {
    return "Hamim Jim ";
  }

  return "Hamim Jim \n" + message(a - 1);
}

int main() {
  cout << message(5);
  return 0;
}
```

Output:
Hamim Jim
Hamim Jim
Hamim Jim
Hamim Jim
Hamim Jim

- **What is the output of the following program fragment:**

```
#include <iostream>
#include <string>

using namespace std;

string message(int a) {
  if (a == 1) {
    return to_string(1);
  }

  return to_string(a) + "\n" + message(a - 1);
}

int main() {
  cout << message(5) << endl;
  return 0;
}
```

Output:

5
4
3
2
1

- **What is the output of the following program fragment:**

```
#include <iostream>
#include <string>

using namespace std;

string message(int a) {
    if (a == 5) {
        return to_string(5);
    }

    return to_string(a) + "\n" + message(a + 1);
}

int main() {
    cout << message(1) << endl;
    return 0;
}
```

Output:

1
2
3
4
5

- **What is the output of the following program fragment:**

```
#include <iostream>
#include <string>

using namespace std;

string message(int a, int b) {
  if (a == b) {
    return to_string(b);
  }

  return to_string(a) + "\n" + message(a + 1, b);
}

int main() {
  cout << message(1, 5);
  return 0;
}
```

Output:

1
2
3
4
5

- **What is the output of the following program fragment:**

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

void print(int a) {
  if (a == 5) {
    cout << a << endl;
    return;
  }

  cout << a << endl;
  print(a + 1);
}

int main() {
  print(1);
  return 0;
}
```

Output:

1
2
3
4
5

Here,
return is used for break the Recursion.

- **What is the output of the following program fragment:**

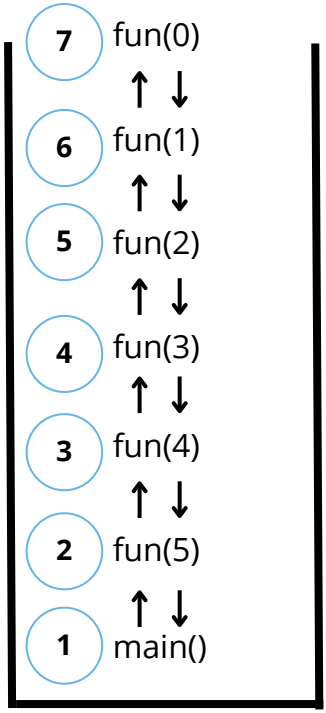
```
#include <iostream>
using namespace std;

void fun(int n) {
  if (n == 0) {
    return;
  }

  cout << n << endl;
  fun(n - 1);
}

int main() {
  fun(5);
  return 0;
}
```

Output:

```
5
4
3
2
1
```

Here,
return is used for break the Recursion.

Here,



Stack memory

- **What is the output of the following program fragment:**

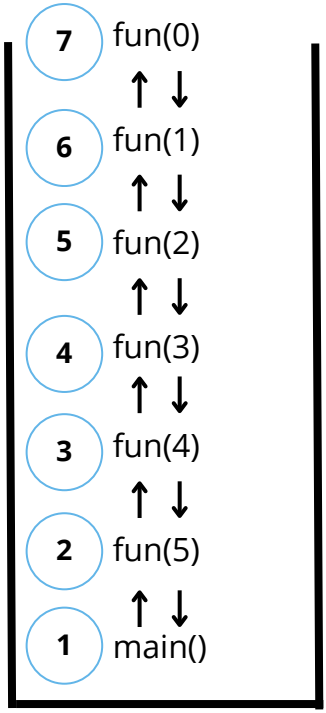
```
#include <iostream>
using namespace std;

void funRev(int n) {
  if (n == 0) {
    return;
  }

  funRev(n - 1);
  cout << n << endl;
}

int main() {
  funRev(5);
  return 0;
}
```

Output:

```
1
2
3
4
5
```

Here,
return is used for break the Recursion.

Here,



Stack memory

- **Make a program that will show the sum of the digits of any number.**

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

int sum(int n) {
  if (n == 0) {
    return 0;
  }

  return (n % 10) + sum(n / 10);
}

int main() {
  int ans = sum(1342);
  cout << ans << endl;
  return 0;
}
```

Output:
10

- **Make a program that will show the multiplication of the digits of any number.**

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

int sum(int n) {
  if (n % 10 == n) {
    return n;
  }

  return (n % 10) * sum(n / 10);
}

int main() {
  int ans = sum(55);
  cout << ans << endl;
  return 0;
}
```

Output:
25

- **Make a program that will reverse a number.**

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

// Recursive function to reverse a number
int reverseNumber(int num, int reversedNum) {
  if (num == 0) {
    return reversedNum;
  }

  int lastDigit = num % 10;
  reversedNum = reversedNum * 10 + lastDigit;

  return reverseNumber(num / 10, reversedNum);
}

int main() {
  cout << "Enter a number: ";
  int num;
  cin >> num;

  int reversedNum = reverseNumber(num, 0);
  cout << "Reversed number: " << reversedNum <<
endl;

  return 0;
}
```

or,

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

// Recursive function to reverse a number
int reverse(int num, int reversedNum) {
  if (num == 0) {
    return reversedNum;
  }

  int lastDigit = num % 10;
  reversedNum = reversedNum * 10 + lastDigit;

  return reverse(num / 10, reversedNum);
}

int reverseNumber(int num) {
  return reverse(num, 0);
}

int main() {
  cout << "Enter a number: ";
  int num;
  cin >> num;

  int reversedNum = reverseNumber(num);
  cout << "Reversed number: " << reversedNum <<
endl;

  return 0;
}
```

Output:

Enter a number: 1824
Reversed number: 4281

- **Make a program that will reverse a number.**

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

int sum = 0;

void reverseNumber(int num) {
  if (num == 0) {
    return;
  }
  int rem = num % 10;
  sum = sum * 10 + rem;
  reverseNumber(num / 10);
}

int main() {
  reverseNumber(1234);
  cout << sum << endl;
  return 0;
}
```

or,

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

int helper(int n, int digits) {
  if (n % 10 == n) {
    return n;
  }
  int rem = n % 10;
  return rem * (int)(pow(10, digits)) + helper(n / 10,
digits - 1);
}

int reverseNumber(int num) {
  int digits = (int)(log10(num));
  return helper(num, digits);
}

int main() {
  cout << reverseNumber(1234) << endl;
  return 0;
}
```

Output:

4321

- **Make a program that will count number of zeros(0) of a number.**

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

int helper(int n, int c) {
  if (n == 0) {
    return c;
  }
  int rem = n % 10;
  if (rem == 0) {
    return helper(n / 10, c + 1);
  }
  return helper(n / 10, c);
}

int countZero(int num) {
  return helper(num, 0);
}

int main() {
  cout << countZero(12049040) << endl;
  return 0;
}
```

Output:

3

- **Given an integer num, return the number of steps to reduce it to zero.**
  In one step, if the current number is even, you have to divide it by 2 , otherwise, you have to subtract 1 from it.

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

int helper(int num, int steps) {
  if (num == 0) {
    return steps;
  }
  if (num % 10 == 0) {
    return helper(num / 10, steps + 1);
  }
  return helper(num - 1, steps + 1);
}

int numberOfSteps(int num) {
  return helper(num, 0);
}

int main() {
  cout << numberOfSteps(14) << endl;
  return 0;
}
```

Output:

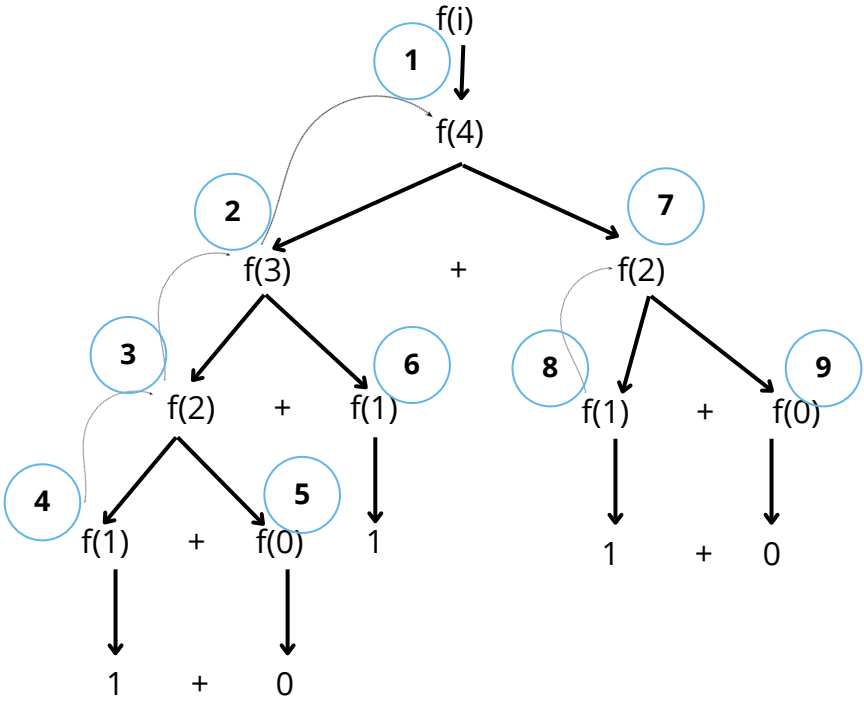
6

- **Make a program which will generates Fibonacci series for a given number using a recursive function**

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

int fibonacci(int i) {
  if (i < 2) {
    return i;
  }

  return fibonacci(i - 1) + fibonacci(i - 2);
}

int main() {
  cout << fibonacci(4) << endl;
  return 0;
}
```

Output:
3

Here,



- **Make a program which will generates Fibonacci series for a given number using a recursive function**

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

int fibonacci(int i) {
  if (i == 0) {
    return 0;
  }

  if (i == 1) {
    return 1;
  }
  return fibonacci(i - 1) + fibonacci(i - 2);
}

int main() {
  cout << fibonacci(5) << endl;
  return 0;
}
```

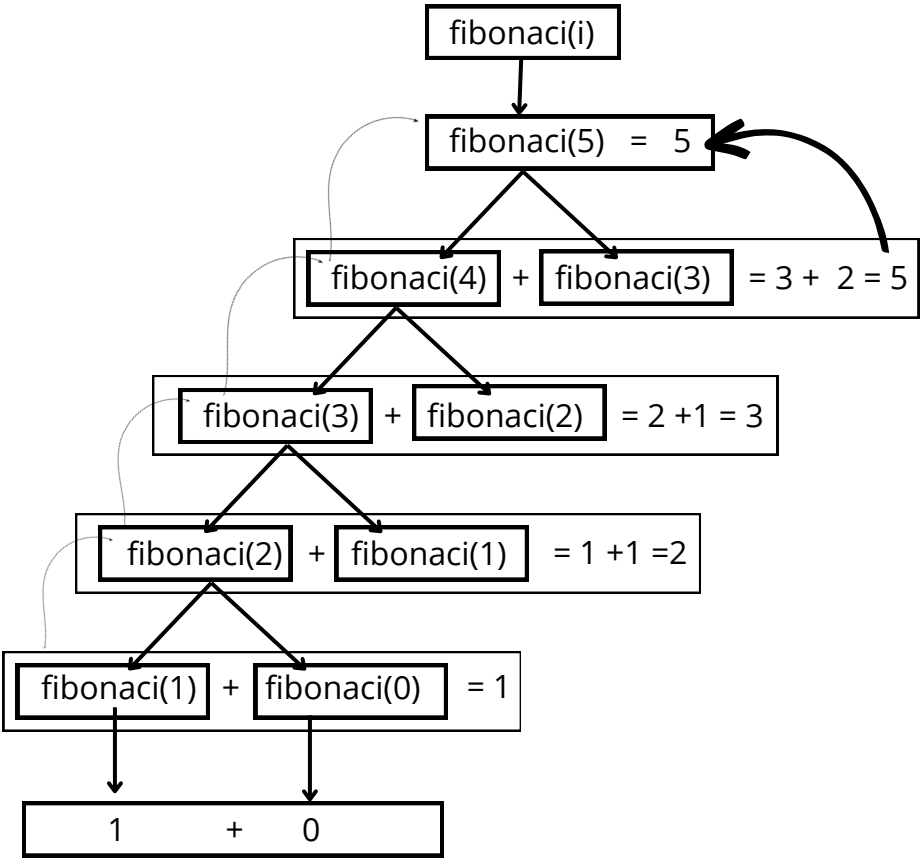
or,

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

int fibonacci(int i) {
  if (i < 2) {
    return i;
  }

  return fibonacci(i - 1) + fibonacci(i - 2);
}

int main() {
  cout << fibonacci(5) << endl;
  return 0;
}
```

Output:

5

Here,


fibonaci(i)
fibonaci(5) = 5
fibonaci(4) + fibonaci(3) = 3 + 2 = 5
fibonaci(3) + fibonaci(2) = 2 +1 = 3
fibonaci(2) + fibonaci(1) = 1 +1 =2
fibonaci(1) + fibonaci(0) = 1
1 + 0

- **Make a program which will generates Fibonacci series for a given number using a recursive function**

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

int fibonacci(int i) {
  if (i == 0) {
    return 0;
  }
  if (i == 1) {
    return 1;
  }
  return fibonacci(i - 1) + fibonacci(i - 2);
}

int main() {
  for (int i = 0; i < 10; i++) {
    cout << fibonacci(i) << "\t";
  }
  cout << endl;

  return 0;
}
```

Output:

```
0	1	1	2	3	5	8	13	21	34
```

- **Make a program which will generates Fibonacci series for a given number using a recursive function**

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

int fibonacci(int i) {
  if (i < 2) {
    return i;
  }

  return fibonacci(i - 1) + fibonacci(i - 2);
}

int main() {
  for (int i = 0; i < 10; i++) {
    cout << fibonacci(i) << "\t";
  }
  cout << endl;

  return 0;
}
```

Output:

```
0	1	1	2	3	5	8	13	21	34
```

- **Make a binary search function using recursion.**

```
public class Test {
   public static int binarySearch(int[] arr, int target, int s,
int e) {
      if (s > e) {
         return -1;
      }
      int m = s + (e - s) / 2;
      if (arr[m] == target) {
         return m;
      }
      if (target < arr[m]) {
         return binarySearch(arr, target, s, m - 1);
      }
      return binarySearch(arr, target, m + 1, e);

   }

   public static void main(String[] args) {
      int[] arr = { 1, 2, 3, 4, 55, 66, 78 };
      int target = 4;
      System.out.println(binarySearch(arr, target, 0,
arr.length));
   }
}
```

Output:

3

Here,

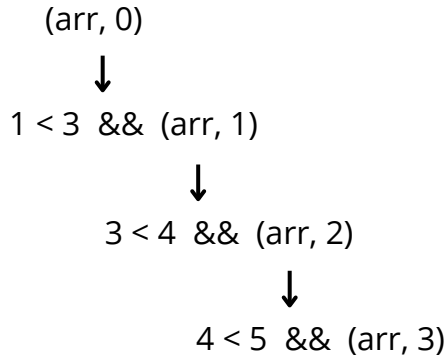
- **Find if an array is sorted or not.**

```
#include <iostream>

using namespace std;

bool helper(int arr[], int index, int length) {
  if (index == length - 1) {
    return true;
  }
  return (arr[index] < arr[index + 1]) && helper(arr,
index + 1, length);
}

bool arraySorted(int arr[], int length) {
  return helper(arr, 0, length);
}

int main() {
  int arr[] = {1, 3, 4, 5};
  int length = sizeof(arr) / sizeof(arr[0]);
  cout << std::boolalpha << arraySorted(arr, length)
<< endl;

  return 0;
}
```

Output:
true

Here,

arr = {1, 3, 4, 5}

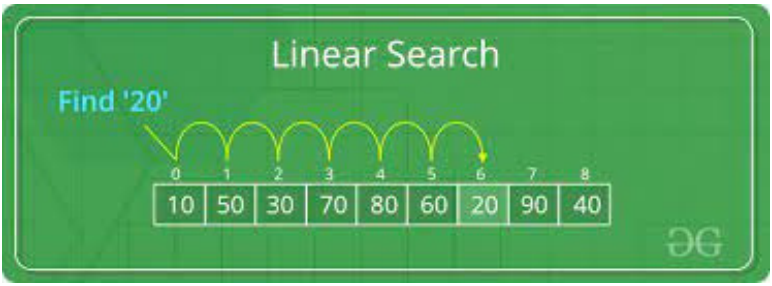


- **Searching :**

1. Linear Search
2. Binary Search
3. Modified Binary Search
4. Binary Search on 2D Arrays

1. Linear Search :

Linear search is a simple searching algorithm that iterates through an array or a list to find a specific element. It sequentially checks each element until the target element is found or until the end of the list is reached.



- **Linear search using recursion.**

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

bool helper(int arr[], int target, int index, int size) {
 if (index > size - 1) {
  return false;
 }
 if (arr[index] == target) {
  return true;
 }
 return helper(arr, target, index + 1, size);
}

bool linearSearch(int arr[], int target, int size) {
 return helper(arr, target, 0, size);
}

int main() {
 int arr[] = {7, 3, 4, 9, 5};
 int target = 4;
 int size = sizeof(arr) / sizeof(arr[0]);
 cout << std::boolalpha << linearSearch(arr, target,
size) << endl;

 return 0;
}
```

or,

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

bool helper(int arr[], int target, int index, int size) {
 if (index > size - 1) {
  return false;
 }
 if (arr[index] == target) {
  return true;
 }
 return helper(arr, target, index + 1, size);
}

bool linearSearch(int arr[], int target, int size) {
 return helper(arr, target, 0, size);
}

int main() {
 int arr[] = {7, 3, 4, 9, 5};
 int target = 4;
 int size = sizeof(arr) / sizeof(arr[0]);
 cout << boolalpha << linearSearch(arr, target, size)
<< endl;

 return 0;
}
```

Output:

```
true
```

- **Linear search using recursion.**
  Return type integer(index of target number).
  < Left index>

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

int findIndex(int arr[], int size, int target, int index) {
  if (index >= size) {
    return -1;
  }
  if (arr[index] == target) {
    return index;
  } else {
    return findIndex(arr, size, target, index + 1);
  }
}

int LinearSearch(int arr[], int size, int target) {
  return findIndex(arr, size, target, 0);
}

int main() {
  int arr[] = {7, 3, 4, 9, 4, 5};
  int size = sizeof(arr) / sizeof(arr[0]);
  int target = 4;
  cout << LinearSearch(arr, size, target) << endl;
  return 0;
}
```

Output:

2

- **Linear search using recursion.**
  Return type integer(index of target number).
  <Right index>

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

int findIndex(int arr[], int size, int target, int index) {
  if (index == -1) {
    return -1;
  }
  if (arr[index] == target) {
    return index;
  } else {
    return findIndex(arr, size, target, index - 1);
  }
}

int LinearSearch(int arr[], int size, int target) {
  return findIndex(arr, size, target, size - 1);
}

int main() {
  int arr[] = {7, 3, 4, 9, 4, 5};
  int size = sizeof(arr) / sizeof(arr[0]);
  int target = 4;
  cout << LinearSearch(arr, size, target) << endl;
  return 0;
}
```

Output:

4

- **Make a function that will take an array and that will search a target element from the array. After finding the elements that are equal to the target element from the array it will return the indexes of them by an array.**

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

vector<int> searchIndexesRecursive(int inputArray[],
int size, int target, int currentIndex, vector<int>&
indexList) {
  if (currentIndex >= size) {
    return indexList;
  }

  if (inputArray[currentIndex] == target) {
    indexList.push_back(currentIndex);
  }

  return searchIndexesRecursive(inputArray, size,
target, currentIndex + 1, indexList);
}

vector<int> searchIndexes(int inputArray[], int size, int
target) {
  vector<int> indexList;
  return searchIndexesRecursive(inputArray, size,
target, 0, indexList);
}
```

```
int main() {
 int array[] = {4, 2, 6, 3, 2, 7, 2, 8};
 int size = sizeof(array) / sizeof(array[0]);
 int target = 2;

 vector<int> indexes = searchIndexes(array, size,
target);

 cout << "Target Element: " << target << endl;
 cout << "Indexes of target element: ";
 for (int index : indexes) {
 cout << index << " ";
 }
 cout << endl;

 return 0;
}
```

Output:

Target Element: 2
Indexes of target element: 1 4 6

or,

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

vector<int> searchIndexesRecursive(int inputArray[], int
size, int target, int currentIndex, vector<int>& indexList) {
  if (currentIndex >= size) {
    return indexList;
  }

  if (inputArray[currentIndex] == target) {
    indexList.push_back(currentIndex);
  }

  return searchIndexesRecursive(inputArray, size, target,
currentIndex + 1, indexList);
}

vector<int> searchIndexes(int inputArray[], int size, int
target) {
  vector<int> indexList;
  return searchIndexesRecursive(inputArray, size, target,
0, indexList);
}

int main() {
  int array[] = {4, 2, 6, 3, 2, 7, 2, 8};
  int size = sizeof(array) / sizeof(array[0]);
  int target = 2;

  vector<int> indexes = searchIndexes(array, size, target);
```

```
 cout << "Target Element: " << target << endl;
 cout << "Indexes of target element: ";
 for (int index : indexes) {
 cout << index << " ";
 }
 cout << endl;

 return 0;
}
```

Output:

Target Element: 2
Indexes of target element: 1 4 6

or,

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

vector<int> searchIndexesRecursive(int inputArray[], int
size, int target, int currentIndex, vector<int>& indexList) {
  if (currentIndex >= size) {
    return indexList;
  }

  if (inputArray[currentIndex] == target) {
    indexList.push_back(currentIndex);
  }

  return searchIndexesRecursive(inputArray, size, target,
currentIndex + 1, indexList);
}

vector<int> searchIndexes(int inputArray[], int size, int
target) {
  vector<int> indexList;
  return searchIndexesRecursive(inputArray, size, target,
0, indexList);
}

int main() {
  int array[] = {4, 2, 6, 3, 2, 7, 2, 8};
  int size = sizeof(array) / sizeof(array[0]);
  int target = 2;
```

```
 cout << "Target Element: " << target << endl;
 cout << "Indexes of target element: ";

 vector<int> indexes = searchIndexes(array, size,
target);

 if (indexes.empty()) {
 cout << "Not found";
 } else {
 for (int index : indexes) {
 cout << index << " ";
 }
 }

 cout << endl;

 return 0;
}
```

Output:

Target Element: 2
Indexes of target element: [1, 4, 6]

or,

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

vector<int> LinearSearchIndexes(int inputArray[], int
size, int target, int currentIndex) {
   vector<int> indexList;

   if (currentIndex == size) {
      return indexList;
   }

   if (inputArray[currentIndex] == target) {
      indexList.push_back(currentIndex);
   }

   vector<int> ansFromBelowCalls =
LinearSearchIndexes(inputArray, size, target,
currentIndex + 1);
   indexList.insert(indexList.end(),
ansFromBelowCalls.begin(),
ansFromBelowCalls.end());

   return indexList;
}

int main() {
   int array[] = {4, 2, 6, 3, 2, 7, 2, 8};
   int size = sizeof(array) / sizeof(array[0]);
   int target = 2;
```

```
 cout << "Target Element: " << target << endl;
 cout << "Indexes of target element: ";

 vector<int> indexes = LinearSearchIndexes(array,
size, target, 0);

 if (indexes.empty()) {
 cout << "Not found";
 } else {
 for (int index : indexes) {
 cout << index << " ";
 }
 }

 cout << endl;

 return 0;
}
```

Output:

Target Element: 2
Indexes of target element: [1, 4, 6]

or,

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

vector<int> searchIndexesRecursive(int inputArray[],
int size, int target, int currentIndex, vector<int>&
indexList) {
  if (currentIndex >= size) {
    return indexList;
  }

  if (inputArray[currentIndex] == target) {
    indexList.push_back(currentIndex);
  }

  return searchIndexesRecursive(inputArray, size,
target, currentIndex + 1, indexList);
}

int main() {
  int array[] = {4, 2, 6, 3, 2, 7, 2, 8};
  int size = sizeof(array) / sizeof(array[0]);
  int target = 2;

  cout << "Target Element: " << target << endl;
  cout << "Indexes of target element: ";

  vector<int> indexes;
  searchIndexesRecursive(array, size, target, 0, indexes);
```

```
  if (indexes.empty()) {
    cout << "Not found";
  } else {
    for (int index : indexes) {
      cout << index << " ";
    }
  }

  cout << endl;

  return 0;
}
```

Output:

Target Element: 2
Indexes of target element: 1 4 6

or,

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

vector<int> indexList;

void searchIndexesRecursive(int arr[], int size, int target,
int index) {
  if (index == size) {
    return;
  }
  if (arr[index] == target) {
    indexList.push_back(index);
  }
  searchIndexesRecursive(arr, size, target, index + 1);
}

void searchIndexes(int inputArray[], int size, int target) {
  searchIndexesRecursive(inputArray, size, target, 0);
}

int main() {
  int array[] = {4, 2, 6, 3, 2, 7, 2, 8};
  int size = sizeof(array) / sizeof(array[0]);
  int target = 2;

  cout << "Target: " << target << endl;
  searchIndexes(array, size, target);

  cout << "Indexes of target element: ";
```

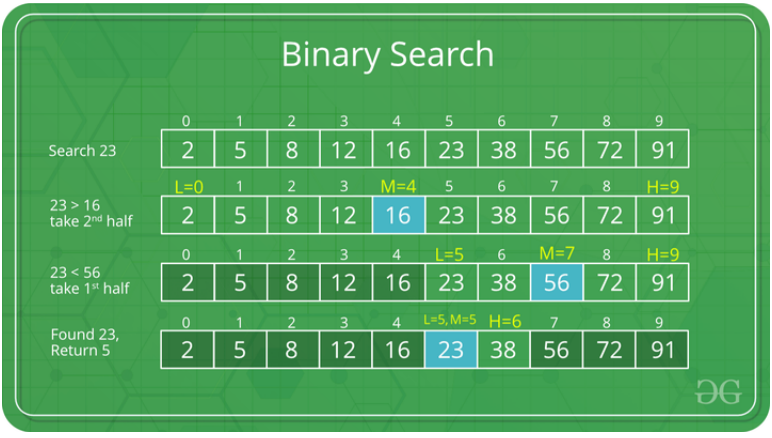
```
  for (int index : indexList) {
    cout << index << " ";
  }
  cout << endl;

  return 0;
}
```

Output:

```
Target: 2
Indexes of target element: 1 4 6
```

2. Binary Search :

Binary search is a search algorithm that finds the position of a target value within a sorted array. It works by repeatedly dividing the search space in half until the target value is found or it is determined that the target value does not exist in the array.



- **Binary search from sorted array.**

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

int binarySearch(int arr[], int target, int left, int
right) {
  if (left <= right) {
    int mid = left + (right - left) / 2;

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

  return -1;
}

int binarySearch(int arr[], int target, int size) {
  return binarySearch(arr, target, 0, size - 1);
}

int main() {
  int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int size = sizeof(arr) / sizeof(arr[0]);
  int target = 6;
```

```
 int result = binarySearch(arr, target, size);

 if (result != -1) {
 cout << "Element " << target << " found at index " <<
result << endl;
 } else {
 cout << "Element " << target << " not found in the
array." << endl;
 }

 return 0;
}
```

Output:
Element 6 found at index 5

- **Binary search from sorted array.**

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

int binarySearch(int arr[], int target, int left, int
right) {
  if (left > right) {
    return -1;
  }

  int mid = left + (right - left) / 2;

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

int binarySearch(int arr[], int target, int size) {
  return binarySearch(arr, target, 0, size - 1);
}

int main() {
  int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int size = sizeof(arr) / sizeof(arr[0]);
  int target = 6;

  int result = binarySearch(arr, target, size);
```

```
 if (result != -1) {
 cout << "Element " << target << " found at index " <<
result << endl;
 } else {
 cout << "Element " << target << " not found in the
array." << endl;
 }

 return 0;
}
```

Output:

Element 6 found at index 5

- **Binary search(First occurrence) from sorted array.**

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

int binarySearch(int arr[], int left, int right, int
target) {
  if (right >= left) {
    int mid = left + (right - left) / 2;

    if (arr[mid] == target) {
      if (mid == 0 || arr[mid - 1] < target) {
        return mid;
      } else {
        return binarySearch(arr, left, mid - 1, target);
      }
    }
    if (arr[mid] > target) {
      return binarySearch(arr, left, mid - 1, target);
    }
    return binarySearch(arr, mid + 1, right, target);
  }

  return -1;
}

int main() {
  int arr[] = {1, 2, 2, 2, 3, 4, 4, 5, 6};
  int target = 4;
  int n = sizeof(arr) / sizeof(arr[0]);
```

```
 int result = binarySearch(arr, 0, n - 1, target);
 if (result != -1) {
 cout << "First occurrence of " << target << " is at
index " << result << endl;
 } else {
 cout << target << " not found in the array" << endl;
 }

 return 0;
}
```

Output:

First occurrence of 4 is at index 5

- **Binary search(Last occurrence) from sorted array.**

```
#include <iostream>

using namespace std;

int binarySearchLastOccurrence(int arr[], int left, int
right, int target) {
  if (right >= left) {
    int mid = left + (right - left) / 2;

    if (arr[mid] == target) {
      if (mid == right || arr[mid + 1] > target) {
        return mid;
      } else {
        return binarySearchLastOccurrence(arr, mid +
1, right, target);
      }
    }

    if (arr[mid] > target) {
      return binarySearchLastOccurrence(arr, left,
mid - 1, target);
    }

    return binarySearchLastOccurrence(arr, mid + 1,
right, target);
  }

  return -1;
}
```

```
int main() {
 int arr[] = {1, 2, 2, 2, 3, 4, 4, 5, 6};
 int target = 4;
 int n = sizeof(arr) / sizeof(arr[0]);

 int result = binarySearchLastOccurrence(arr, 0, n -
1, target);

 if (result != -1) {
 cout << "Last occurrence of " << target << " is at
index " << result << endl;
 } else {
 cout << target << " not found in the array" << endl;
 }

 return 0;
}
```

Output:

Last occurrence of 4 is at index 6

- **Binary search(First and Last occurrence) from sorted array.**

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

vector<int> findFirstAndLastOccurrences(int arr[],
int left, int right, int target) {
  vector<int> result = {-1, -1};

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

    if (arr[mid] == target) {
      if (mid == 0 || arr[mid - 1] < target) {
        result[0] = mid;
      }

      result[1] = mid;
      vector<int> leftResult =
findFirstAndLastOccurrences(arr, left, mid - 1,
target);
      vector<int> rightResult =
findFirstAndLastOccurrences(arr, mid + 1, right,
target);

      if (leftResult[0] != -1) {
        result[0] = leftResult[0];
      }
```

```
 if (rightResult[1] != -1) {
 result[1] = rightResult[1];
 }
 } else if (arr[mid] > target) {
 return findFirstAndLastOccurrences(arr, left, mid -
1, target);
 } else {
 return findFirstAndLastOccurrences(arr, mid + 1,
right, target);
 }
 }

 return result;
}

int main() {
 int arr[] = {1, 2, 2, 2, 3, 4, 4, 5, 6};
 int target = 6;
 int n = sizeof(arr) / sizeof(arr[0]);

 vector<int> result =
findFirstAndLastOccurrences(arr, 0, n - 1, target);

 cout << "Target: " << target << endl;
 if (result[0] != -1) {
 cout << "Indexes of first and last occurrence: ";
 for (int index : result) {
 cout << index << " ";
 }
 cout << endl;
 } else {
```

```
 cout << target << " not found in the array" << endl;
 }

 return 0;
}
```

Output:

Target: 6
Indexes of first and last occurrence: 8 8

- **Roted binary search from roted sorted array.**

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

int rotatedBinarySearch(int arr[], int target, int s, int
e) {
  if (s > e) {
    return -1;
  }

  int m = s + (e - s) / 2;

  if (arr[m] == target) {
    return m;
  }

  if (arr[s] <= arr[m]) {
    if (target >= arr[s] && target <= arr[m]) {
      return rotatedBinarySearch(arr, target, s, m - 1);
    } else {
      return rotatedBinarySearch(arr, target, m + 1,
e);
    }
  }

  if (target >= arr[m] && target <= arr[e]) {
    return rotatedBinarySearch(arr, target, m + 1, e);
  }
```

```
 return rotatedBinarySearch(arr, target, s, m - 1);
}

int main() {
 int array[] = {5, 6, 7, 8, 9, 1, 2, 3};
 int target = 2;

 cout << "Target Element: " << target << endl;
 cout << "Index of target element: " <<
rotatedBinarySearch(array, target, 0, sizeof(array) /
sizeof(array[0]) - 1) << endl;

 return 0;
}
```

Output:

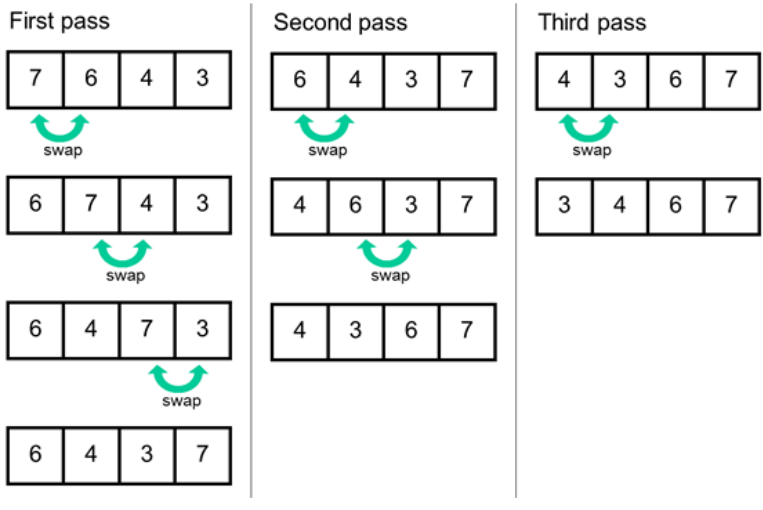
Target Element: 2
Indexe of target element: 6

- **Sorting :**

1. Bubble Sort
2. Selection Sort
3. Merge Sort
4. Quick Sort

1. Bubble Sort :

Bubble sort is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the entire list is sorted.



- **What is the output of the following program fragment:**

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

void BubbleSort(int arr[], int arrLength, int c) {
  if (arrLength == 0) {
    return;
  }

  if (arrLength == c) {
    BubbleSort(arr, arrLength - 1, 0);
  } else {
    if (arr[c] > arr[c + 1]) {
      // swap
      int temp = arr[c];
      arr[c] = arr[c + 1];
      arr[c + 1] = temp;
    }
    BubbleSort(arr, arrLength, c + 1);
  }
}

int main() {
  int arr[] = {4, 6, 4, 2, 8};
  int arrLength = sizeof(arr) / sizeof(arr[0]);

  cout << "Before sort: ";
```

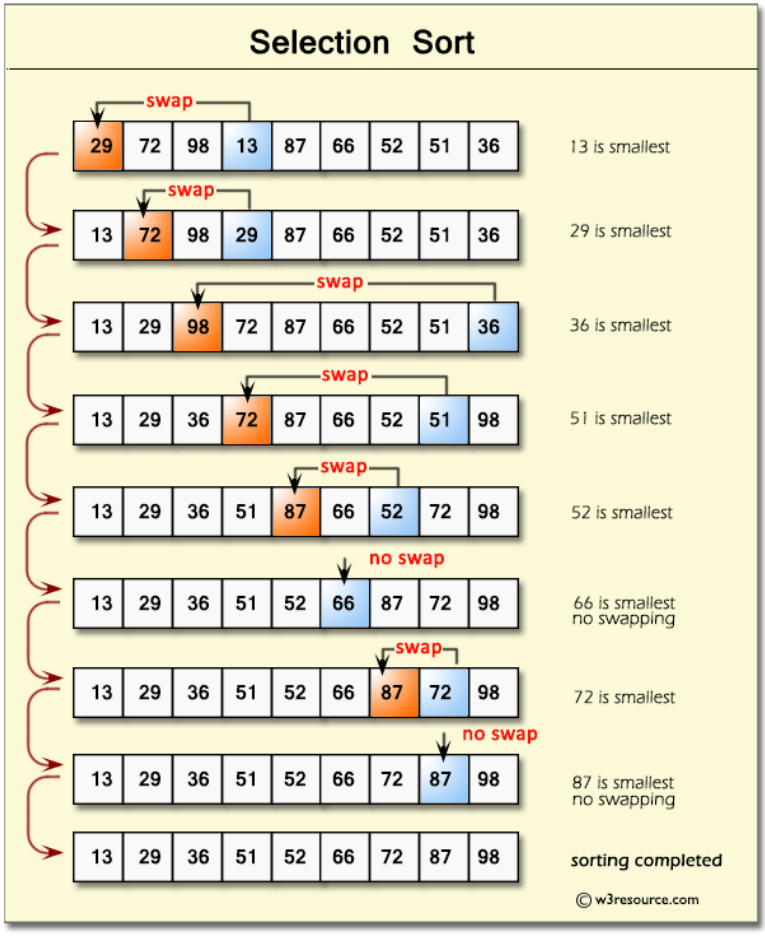
```
  for (int i = 0; i < arrLength; i++) {
    cout << arr[i] << " ";
  }
  cout << endl;

  BubbleSort(arr, arrLength - 1, 0);

  cout << "After sort: ";
  for (int i = 0; i < arrLength; i++) {
    cout << arr[i] << " ";
  }
  cout << endl;

  return 0;
}
```

Output:

Before sort: 4 6 4 2 8
After sort: 2 4 4 6 8

2. Selection Sort :

Selection Sort is a simple sorting algorithm that works by repeatedly finding the minimum element from the unsorted part of the array and swapping it with the first unsorted element. It effectively divides the array into two parts: the sorted part on the left and the unsorted part on the right.



- **What is the output of the following program fragment:**

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

void BubbleSort(int arr[], int arrLength, int c, int max) {
  if (arrLength == 0) {
    return;
  }

  if (arrLength == c) {
    int temp = arr[max];
    arr[max] = arr[arrLength - 1];
    arr[arrLength - 1] = temp;
    BubbleSort(arr, arrLength - 1, 0, 0);
  } else {
    if (arr[c] > arr[max]) {
      BubbleSort(arr, arrLength, c + 1, c);
    } else {
      BubbleSort(arr, arrLength, c + 1, max);
    }
  }
}

int main() {
  int arr[] = {4, 6, 4, 2, 8};
  int arrLength = sizeof(arr) / sizeof(arr[0]);

  cout << "Before sort: ";
```

```
 for (int i = 0; i < arrLength; i++) {
 cout << arr[i] << " ";
 }
 cout << endl;

 BubbleSort(arr, arrLength, 0, 0);

 cout << "After sort: ";
 for (int i = 0; i < arrLength; i++) {
 cout << arr[i] << " ";
 }
 cout << endl;

 return 0;
}
```

Output:

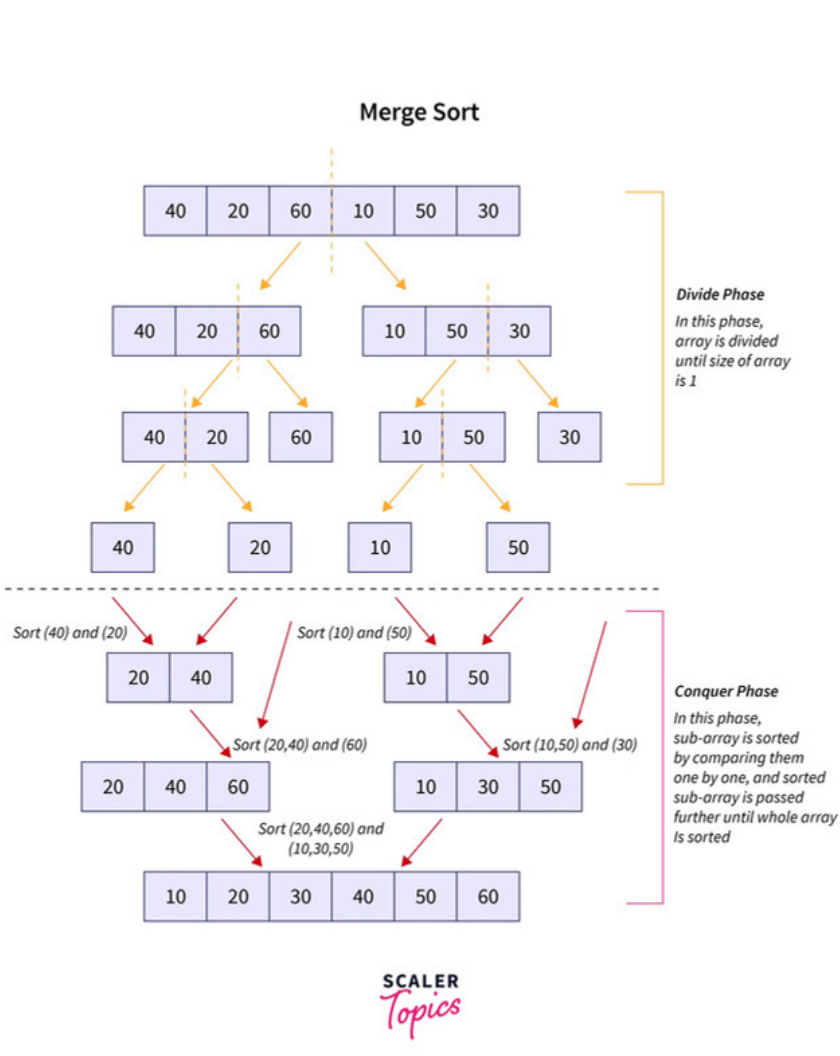
Before sort: 4 6 4 2 8
After sort: 2 4 4 6 8

3. Merge Sort :

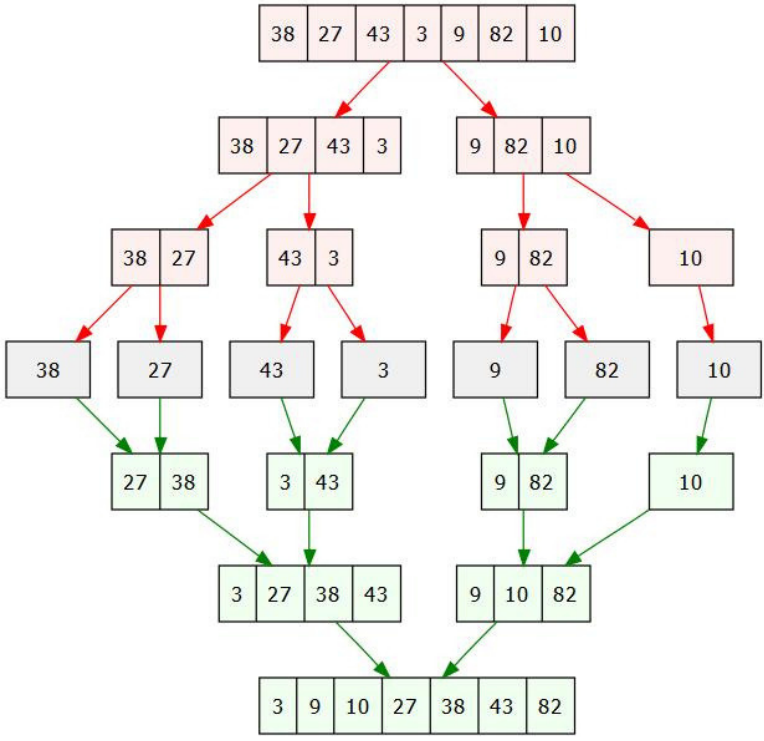
Merge Sort is a widely used comparison-based sorting algorithm that follows the divide-and-conquer strategy. It's known for its stability, consistent time complexity, and efficient performance for large datasets. The main idea behind Merge Sort is to break down a large array into smaller subarrays, sort them individually, and then merge them back together to obtain a sorted array.

The algorithm works as follows:

1. Divide: The unsorted array is divided into two equal halves (or as close to equal as possible).
2. Conquer: Each half is recursively sorted using the Merge Sort algorithm.
3. Merge: The sorted halves are then merged back together in a way that maintains the sorted order. This involves comparing the elements of the two halves and placing them in the correct order in a temporary array. Once all elements are merged, the temporary array becomes the sorted array.



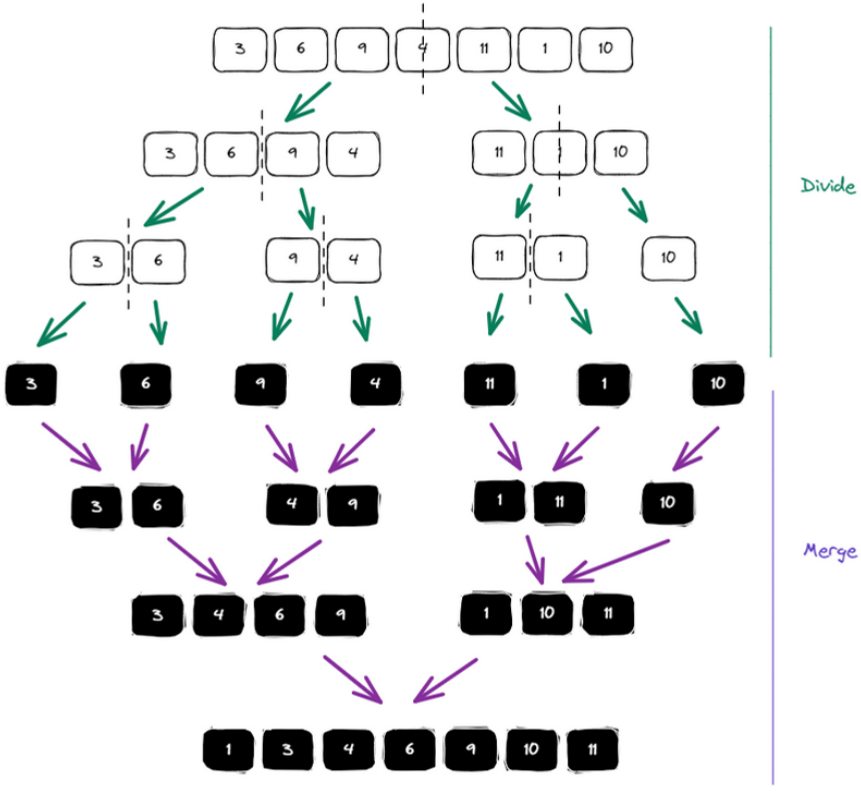
SCALER Topics


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Time Complexity:
Best case: O(n logn)
Worst case: O(n logn)

Where n is the size of the array.

Space Complexity:
O(log n)


3 6 9 4 11 1 10
3 6 9 4
11 1 10
3 6
9 4
11 1
10
3
6
9
4
11
1
10
3 6
4 9
1 11
10
3 4 6 9
1 10 11
1 3 4 6 9 10 11
Divide
Merge

- **What is the output of the following program fragment:**

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

void merge(int arr[], int s, int m, int e, int mix_size)
{
  int* mix = new int[mix_size];
  int i = s;
  int j = m + 1;
  int k = 0;

  while (i <= m && j <= e)
  {
    if (arr[i] < arr[j])
    {
      mix[k] = arr[i];
      i++;
    }
    else
    {
      mix[k] = arr[j];
      j++;
    }
    k++;
  }

  // it may be possible that one of the arrays is not
complete
  // copy the remaining elements
```

```
 while (i <= m)
 {
 mix[k] = arr[i];
 i++;
 k++;
 }

 while (j <= e)
 {
 mix[k] = arr[j];
 j++;
 k++;
 }

 for (int l = 0; l < mix_size; l++)
 {
 arr[s + l] = mix[l];
 }

 delete[] mix; // Release the allocated memory
}

void mergeSort(int arr[], int l, int r)
{
 if (l < r)
 {
 int m = l + (r - l) / 2;
 int mix_size = (r - l) + 1; // Size of the dynamic array mix
 mergeSort(arr, l, m);
 mergeSort(arr, m + 1, r);
 merge(arr, l, m, r, mix_size);
 }
}
```

```
int main()
{
 int arr[] = {12, 11, 13, 5, 6, 7};
 int arr_size = sizeof(arr)/sizeof(arr[0]);

 cout << "Original array: ";
 for (int i = 0; i < arr_size; i++){
 cout << arr[i] << " ";
 }
 cout << endl;

 mergeSort(arr, 0, arr_size - 1);

 cout << "Sorted array: ";
 for (int i = 0; i < arr_size; i++){
 cout << arr[i] << " ";
 }
 cout << endl;

 return 0;
}
```

Output:

Original array: 12 11 13 5 6 7
Sorted array: 5 6 7 11 12 13

- **What is the output of the following program fragment:**

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

void merge(int arr[], int s, int m, int e) {
  int* mix = new int[e - s];
  int i = s;
  int j = m;
  int k = 0;

  while (i < m && j < e) {
    if (arr[i] < arr[j]) {
      mix[k] = arr[i];
      i++;
    } else {
      mix[k] = arr[j];
      j++;
    }
    k++;
  }

  // It may be possible that one of the arrays is not
complete.
  // Copy the remaining elements.
  while (i < m) {
    mix[k] = arr[i];
    i++;
    k++;
  }
```

```
 while (j < e) {
 mix[k] = arr[j];
 j++;
 k++;
 }

 for (int l = 0; l < k; l++) {
 arr[s + l] = mix[l];
 }

 delete[] mix; // Release the allocated memory
}

void mergeSort(int arr[], int s, int e) {
 if (e - s == 1) {
 return;
 }

 int mid = (s + e) / 2;

 mergeSort(arr, s, mid);
 mergeSort(arr, mid, e);

 merge(arr, s, mid, e);
}

int main() {
 int arr[] = {5, 4, 3, 2, 1};
 int arr_size = sizeof(arr) / sizeof(arr[0]);

 cout << "Before sort: ";
```

```
 for (int i = 0; i < arr_size; i++) {
 cout << arr[i] << " ";
 }
 cout << endl;

 mergeSort(arr, 0, arr_size);

 cout << "After sort: ";
 for (int i = 0; i < arr_size; i++) {
 cout << arr[i] << " ";
 }
 cout << endl;

 return 0;
}
```

Output:

Before sort: 5 4 3 2 1
After sort: 1 2 3 4 5

- **What is the output of the following program fragment:**

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

vector<int> merge(const vector<int>& first, const
vector<int>& second);
vector<int> mergeSort(vector<int> arr);

int main() {
  int arr[] = {5, 4, 3, 2, 1};
  int arr_size = sizeof(arr) / sizeof(arr[0]);

  vector<int> vec(arr, arr + arr_size);

  cout << "Before sort: ";
  for (int i = 0; i < arr_size; i++) {
    cout << arr[i] << " ";
  }
  cout << endl;

  vector<int> sortedVec = mergeSort(vec);

  cout << "After sort: ";
  for (int i = 0; i < arr_size; i++) {
    cout << sortedVec[i] << " ";
  }
  cout << endl;
```

```
 return 0;
}

vector<int> merge(const vector<int>& first, const
vector<int>& second) {
 vector<int> mix;
 int i = 0;
 int j = 0;

 while (i < first.size() && j < second.size()) {
 if (first[i] < second[j]) {
 mix.push_back(first[i]);
 i++;
 } else {
 mix.push_back(second[j]);
 j++;
 }
 }

 // Copy the remaining elements
 while (i < first.size()) {
 mix.push_back(first[i]);
 i++;
 }

 while (j < second.size()) {
 mix.push_back(second[j]);
 j++;
 }

 return mix;
}
```

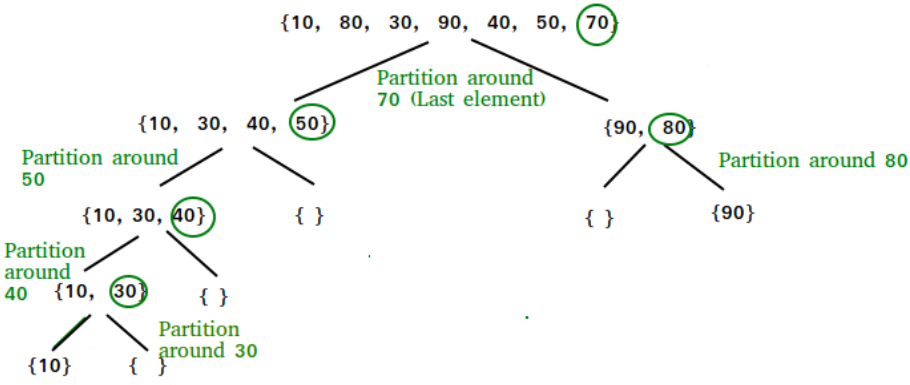
```
vector<int> mergeSort(vector<int> arr) {
 if (arr.size() == 1) {
 return arr;
 }

 int mid = arr.size() / 2;

 vector<int> left(arr.begin(), arr.begin() + mid);
 vector<int> right(arr.begin() + mid, arr.end());

 return merge(mergeSort(left), mergeSort(right));
}
```

Output:

```
Before sort: 5 4 3 2 1
After sort: 1 2 3 4 5
```

4. Quick Sort :

QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.



The key process in quickSort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

Time Complexity:
Best Case: Ω (N log (N))
Average Case: θ (N log (N))
Worst Case: O(N^2)

Auxiliary Space Complexity:
O(1)

If we don't consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make O(N).

1. Best Case:-
The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array into roughly equal halves.

In this case, the algorithm will make balanced partitions, leading to efficient Sorting.

The Best case time complexity of quicksort is O(n log n).

2. Average Case:-

Quicksort's average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.
The Average Case Time-complexity is also O(n log n).

3. worst Case:-

The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions.

When the array is already sorted and the pivot is always chosen as the smallest or largest element.

To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (Randomized Quicksort) to shuffle the element before sorting.

Quicksort Worst-case Time Complexity is O(n^2).

- **What is the output of the following Java program fragment:**

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

void QuickSort(int nums[], int low, int high) {
  if (low >= high) {
    return;
  }

  int s = low;
  int e = high;
  int m = s + (e - s) / 2;
  int pivot = nums[m];

  while (s <= e) {
    // It's also reason whyif its already sorted it will not
swap
    while (nums[s] < pivot) {
      s++;
    }
    while (nums[e] > pivot) {
      e--;
    }

    if (s <= e) {
      swap(nums[s], nums[e]);
      s++;
      e--;
    }
  }
```

```
 // Now the pivot is at the correct index, please sort two
halves now
 QuickSort(nums, low, e);
 QuickSort(nums, s, high);
}

int main() {
 int arr[] = {5, 4, 3, 2, 1};
 int arrSize = sizeof(arr) / sizeof(arr[0]);

 cout << "Before sort : ";
 for (int i = 0; i < arrSize; ++i) {
 cout << arr[i] << " ";
 }
 cout << endl;

 QuickSort(arr, 0, arrSize - 1);

 cout << "After sort : ";
 for (int i = 0; i < arrSize; ++i) {
 cout << arr[i] << " ";
 }
 cout << endl;

 return 0;
}
```

Output:

```
Before sort : 5 4 3 2 1
After sort : 1 2 3 4 5
```

- **Pattern :**

- **What is the output of the following Java program fragment:**

```
#include <iostream>

using namespace std;

void TrianglePattern(int r, int c) {
  if (r == 0) {
    return;
  }
  if (r == c) {
    cout << endl;
    TrianglePattern(r - 1, 0);
  } else {
    cout << "*";
    TrianglePattern(r, c + 1);
  }
}

int main() {
  TrianglePattern(4, 0);
  return 0;
}
```

Output:

```
****
***
**
*
```

or,

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

void TrianglePattern(int r, int c) {
  if (r == 0) {
    return;
  }
  if (c < r) {
    cout << "*";
    TrianglePattern(r, c + 1);
  } else {
    cout << endl;
    TrianglePattern(r - 1, 0);
  }
}

int main() {
  TrianglePattern(4, 0);
  return 0;
}
```

Output:

```
****
***
**
*
```

- **What is the output of the following Java program fragment:**

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

void TrianglePattern(int r, int c) {
  if (r == 0) {
    return;
  }
  if (c < r) {
    TrianglePattern(r, c + 1);
    cout << "*";
  } else {
    TrianglePattern(r - 1, 0);
    cout << endl;
  }
}

int main() {
  TrianglePattern(4, 0);
  return 0;
}
```

Output:

```
*
**
***
****
```

or,

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

void TrianglePattern(int r, int c) {
  if (r == 0) {
    return;
  }
  if (r == c) {
    TrianglePattern(r - 1, 0);
    cout << endl;
  } else {
    TrianglePattern(r, c + 1);
    cout << "*";
  }
}

int main() {
  TrianglePattern(4, 0);
  return 0;
}
```

Output:

```
*
**
***
****
```

- **Recursion Subset, Subsequence, String Questions:**

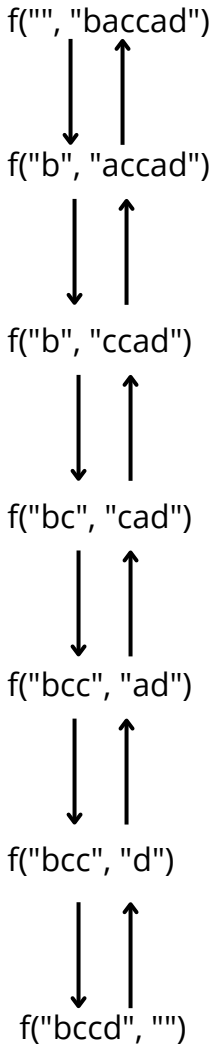
String in recursion:

- **Make a program that will take a string and remove all 'a' character from it after then it will simply print the string.**

```
#include <iostream>
#include <string>

using namespace std;

void skip(string p, string up) {
  if (up.empty()) {
    cout << p << endl;
    return;
  }

  char ch = up[0];

  if (ch == 'a') {
    skip(p, up.substr(1));
  } else {
    skip(p + ch, up.substr(1));
  }
}

int main() {
  skip("", "baccdah");
  return 0;
}
```

Output:
bccdh

Here,

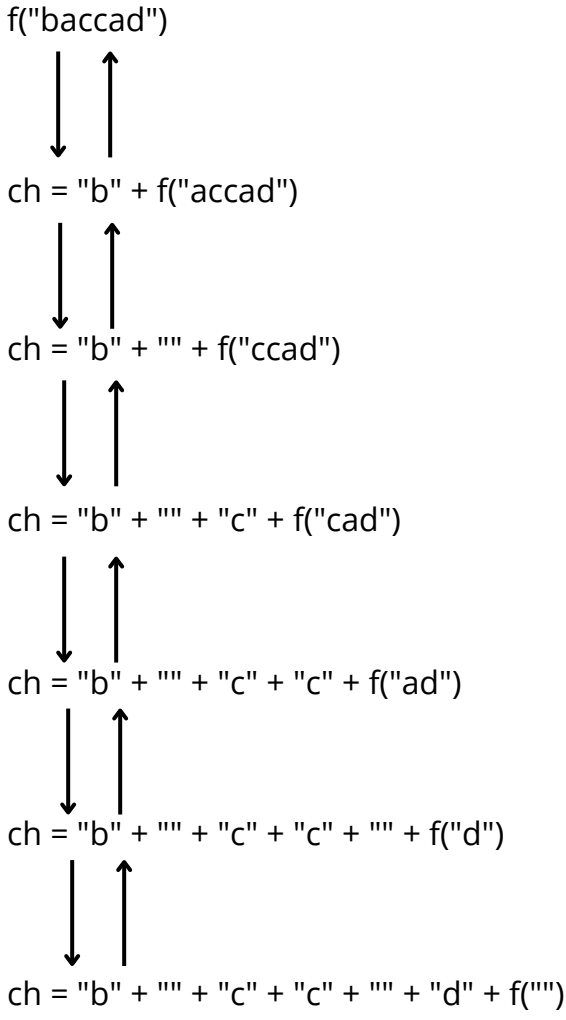


- **Make a program that will take a string and remove all 'a' character from it after then it will simply print the string.**

```
#include <iostream>
#include <string>

using namespace std;

string skip(string up) {
  if (up.empty()) {
    return "";
  }

  char ch = up[0];

  if (ch == 'a') {
    return skip(up.substr(1));
  } else {
    return ch + skip(up.substr(1));
  }
}

int main() {
  cout << skip("baccdah") << endl;
  return 0;
}
```

Output:
bccdh

Here,


f("baccad")
ch = "b" + f("accad")
ch = "b" + "" + f("ccad")
ch = "b" + "" + "c" + f("cad")
ch = "b" + "" + "c" + "c" + f("ad")
ch = "b" + "" + "c" + "c" + "" + f("d")
ch = "b" + "" + "c" + "c" + "" + "d" + f("")

- **Make a program that will take a string and remove all "apple" from it after that it will simply print the string.**

```
#include <iostream>
#include <string>

using namespace std;

string skipApple(string up) {
  if (up.empty()) {
    return "";
  }

  if (up.substr(0, 5) == "apple") {
    return skipApple(up.substr(5));
  } else {
    return up[0] + skipApple(up.substr(1));
  }
}

int main() {
  cout << skipApple("bappleccappledah") << endl;
  return 0;
}
```

Output:
bccdah

- **Make a program that will take a string and remove all "app" only not "apple" from it after that it will simply print the string.**

```
#include <iostream>
#include <string>

using namespace std;

string skipApple(string up) {
  if (up.empty()) {
    return "";
  }

  if (up.substr(0, 3) == "app" && up.substr(0, 5) !=
"apple") {
    return skipApple(up.substr(3));
  } else {
    return up[0] + skipApple(up.substr(1));
  }
}

int main() {
  cout << skipApple("bapplecappcappledah") <<
endl;
  return 0;
}
```

Output:
bappleccappledah

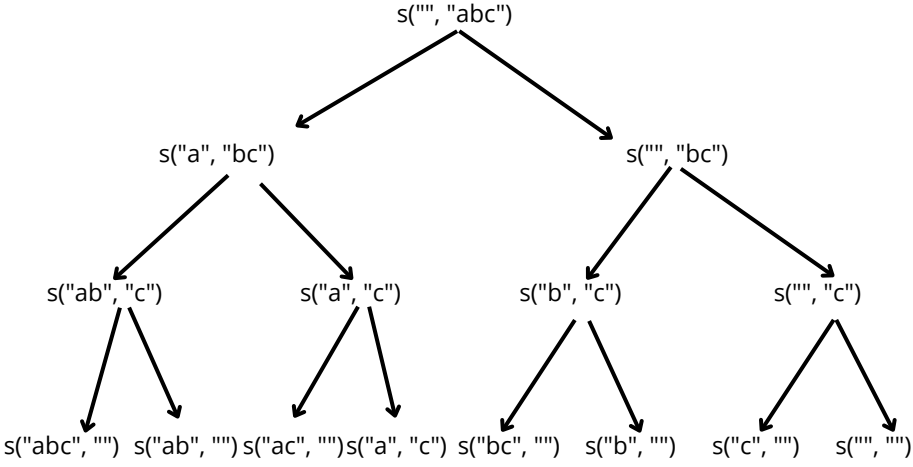
Subsets:

- **Permutation and combination:**

Subset ----> Non-adjacent collection.
[3, 5, 9] ----> [3], [3,5], [3,9], [3,5,9], [5,9], [5], [9]

str = "ABC"
ans = ["a", "b", "c", "ab", "ac", "bc", "abc"]

- **combination:**



- **Subsequence:**

- **Make a program that will show the subsequences of String.**

```
#include <iostream>
#include <string>

using namespace std;

void subSeq(string p, string str) {
  if (str.empty()) {
    cout << p << " ";
    return;
  }

  char ch = str[0];

  subSeq(p + ch, str.substr(1));
  subSeq(p, str.substr(1));
}

int main() {
  subSeq("", "abc");
  cout << endl; // Add a newline for better output
formatting
  return 0;
}
```

Output:

```
abc ab ac a bc b c
```

- **Make a program that will show the subsequences of String.**

```
#include <iostream>
#include <string>

using namespace std;

void subSeq(string p, string str) {
    if (str.empty()) {
        cout << p << " ";
        return;
    }

    char ch = str[0];

    subSeq(p + ch, str.substr(1));
    subSeq(p, str.substr(1));
}

int main() {
    subSeq("", "abc");
    cout << endl; // Add a newline for better output
formatting
    return 0;
}
```

Output:

abc ab ac a bc b c

- **Make a program that will show the subsequences of String.**

```
#include <iostream>
#include <vector>

using namespace std;

vector<string> subSeq(string p, string str) {
  if (str.empty()) {
    vector<string> list;
    list.push_back(p);
    return list;
  }

  char ch = str[0];

  vector<string> left = subSeq(p + ch, str.substr(1));
  vector<string> right = subSeq(p, str.substr(1));

  left.insert(left.end(), right.begin(), right.end());
  return left;
}

int main() {
  vector<string> result = subSeq("", "abc");

  for (const string& s : result) {
    cout << s << " ";
  }
```

```
  cout << endl; // Add a newline for better output
formatting
  return 0;
}
```

Output:

```
[abc, ab, ac, a, bc, b, c, ]
```

- **Make a program that will show the subsequences of String with ASCII value.**

```
#include <iostream>
#include <string>

using namespace std;

void subSeq(string p, string str) {
  if (str.empty()) {
    cout << p << endl;
    return;
  }

  char ch = str[0];

  subSeq(p + ch, str.substr(1));
  subSeq(p, str.substr(1));
  subSeq(p + to_string(static_cast<int>(ch)),
str.substr(1));
}

int main() {
  subSeq("", "abc");
  return 0;
}
```

Output:

abc
ab
ab99
ac
a
a99
a98c
a98
a9899
bc
b
b99
c

99
98c
98
9899
97bc
97b
97b99
97c
97
9799
9798c
9798
979899

- **Make a program that will show the subsequences of String with ASCII value.**

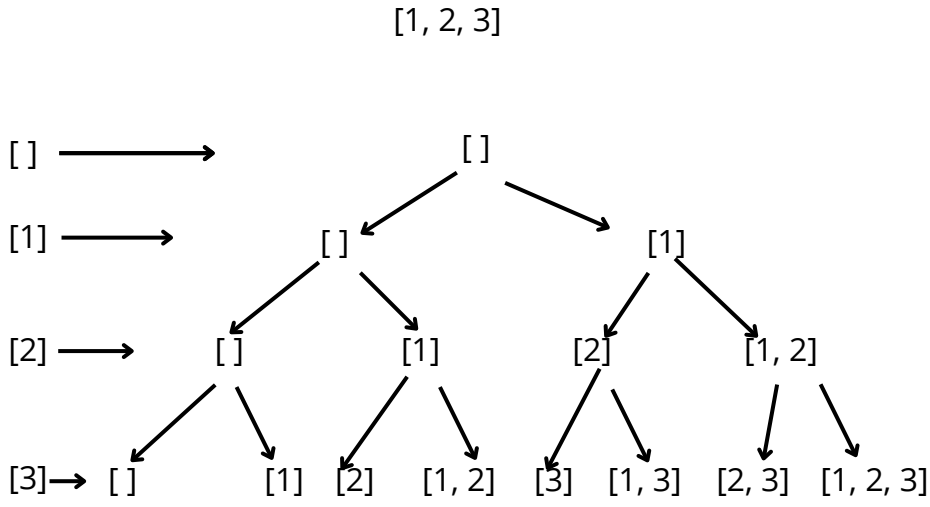
```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

vector<string> subSeqAscii(string p, string str) {
  if (str.empty()) {
    vector<string> list;
    list.push_back(p);
    return list;
  }

  char ch = str[0];

  vector<string> first = subSeqAscii(p + ch,
str.substr(1));
  vector<string> second = subSeqAscii(p,
str.substr(1));
  vector<string> third = subSeqAscii(p +
to_string(static_cast<int>(ch)), str.substr(1));

  first.insert(first.end(), second.begin(),
second.end());
  first.insert(first.end(), third.begin(), third.end());
  return first;
}
```

```
int main() {
  vector<string> result = subSeqAscii("", "abc");

  for (const string& s : result) {
    cout << s << " ";
  }

  cout << endl;
  return 0;
}
```

Output:

```
abc ab ab99 ac a a99 a98c a98 a9899 bc b b99 c  99
98c 98 9899 97bc 97b 97b99 97c 97 9799 9798c
9798 979899
```

- **Subset:**



Time Complexity:
Worst Case: O(n * 2^n)

Auxiliary Space Complexity:
O(2^n * n)

- **Make a program that will show subset of [1, 2, 3].**

```
#include <iostream>
#include <vector>

using namespace std;

vector<vector<int>> subset(const vector<int>& arr)
{
  vector<vector<int>> outer;
  outer.push_back(vector<int>());

  for (int num : arr) {
    int n = outer.size();
    for (int i = 0; i < n; i++) {
      vector<int> internal = outer[i];
      internal.push_back(num);
      outer.push_back(internal);
    }
  }
  return outer;
}

int main() {
  int arr[] = {1, 2, 3};
  vector<int> vec(arr, arr + sizeof(arr) /
sizeof(arr[0]));

  vector<vector<int>> ans = subset(vec);
```

```
  for (const vector<int>& list : ans) {
    for (int num : list) {
      cout << num << " ";
    }
    cout << endl;
  }

  return 0;
}
```

Output:

```
1
2
1 2
3
1 3
2 3
1 2 3
```

- **Make a program that will show subsequencees of a string without duplicate elements.**

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

vector<vector<int>> subset(const vector<int>& arr) {
  vector<vector<int>> outer;
  outer.push_back(vector<int>());
  int start = 0;
  int end = 0;

  vector<int> sortedArr = arr;
  sort(sortedArr.begin(), sortedArr.end());

  for (int i = 0; i < sortedArr.size(); i++) {
    start = 0;
    // if current and previous elements are the same,
start = end + 1
    if (i > 0 && sortedArr[i] == sortedArr[i - 1]) {
      start = end + 1;
    }
    end = outer.size() - 1;
    int n = outer.size();
    for (int j = start; j < n; j++) {
      vector<int> internal = outer[j];
      internal.push_back(sortedArr[i]);
      outer.push_back(internal);
    }
```

```
 }
 return outer;
}

int main() {
 int arr[] = {1, 2, 2};
 vector<int> vec(arr, arr + sizeof(arr) / sizeof(arr[0]));

 vector<vector<int>> ans = subset(vec);

 for (const vector<int>& list : ans) {
 for (int num : list) {
 cout << num << " ";
 }
 cout << endl;
 }
}
```

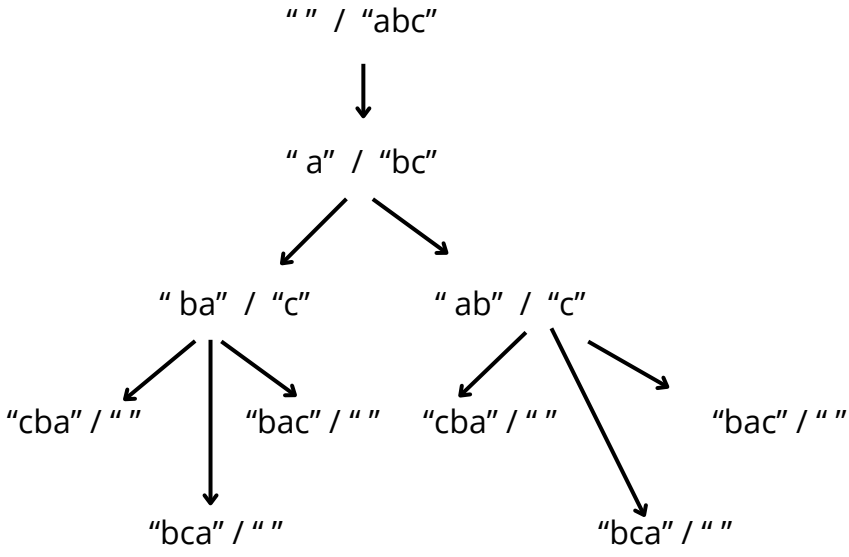
Output:

```
1
2
1 2
2 2
1 2 2
```

Here,
When you find a duplicate element, only add it in the newly created subset of the previous step.

Because of the above point, duplicates have to be together.

- **Permutations:**



- **What is the output of the following Java program fragment:**

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

void permutations(string p, string up) {
  if (up.empty()) {
    cout << p << endl;
    return;
  }

  char ch = up[0];

  for (int i = 0; i <= p.length(); i++) {
    string f = p.substr(0, i);
    string s = p.substr(i);
    permutations(f + ch + s, up.substr(1));
  }
}

int main() {
  permutations("", "abc");
}
```

Output:

```
cba
bca
bac
cab
acb
abc
```

- **What is the output of the following Java program fragment:**

```
#include <iostream>
#include <vector>

using namespace std;

vector<string> permutationList(string p, string up) {
  if (up.empty()) {
    vector<string> list;
    list.push_back(p);
    return list;
  }

  char ch = up[0];

  // local to this call
  vector<string> ans;

  for (int i = 0; i <= p.length(); i++) {
    string f = p.substr(0, i);
    string s = p.substr(i);
    vector<string> subResult = permutationList(f + ch + s,
up.substr(1));
    ans.insert(ans.end(), subResult.begin(),
subResult.end());
  }
  return ans;
}
```

```
int main() {
  vector<string> ans = permutationList("", "abc");

  for (const string& str : ans) {
    cout << str << " ";
  }

  cout << endl;
  return 0;
}
```

Output:
cba bca bac cab acb abc

- **What is the output of the following Java program fragment:**

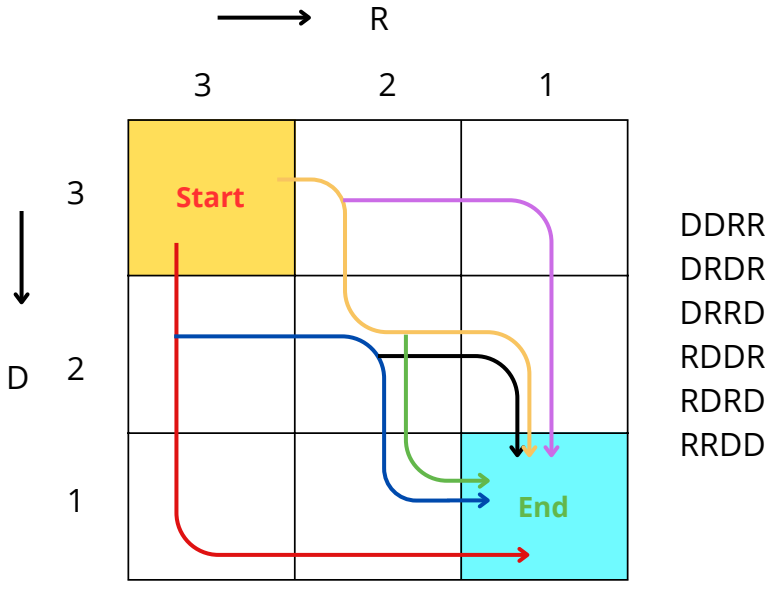
```
class Test{
  public static void main(String[] args) {
    System.out.println(permutationsCount("", "abc"));
  }

  static int permutationsCount(String p, String up){
    if(up.isEmpty()){
      return 1;
    }

    int count = 0;
    char ch = up.charAt(0);
    for(int i=0; i<=p.length(); i++){
      String f = p.substring(0, i);
      String s = p.substring(i, p.length());
      count = count + permutationsCount(f + ch + s,
up.substring(1));
    }

    return count;
  }
}
```

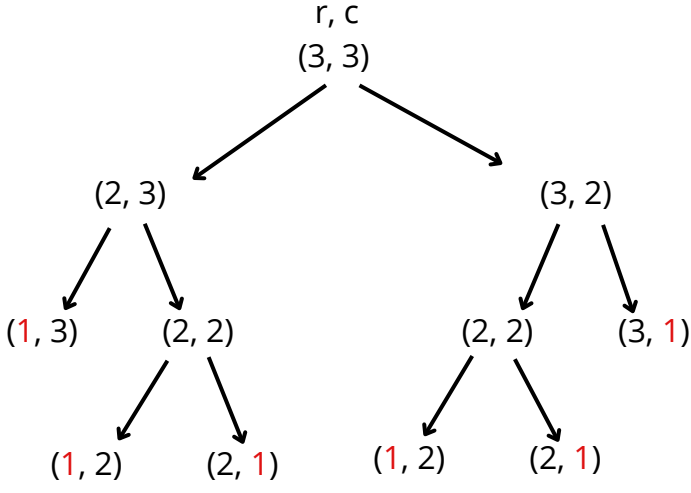
Output:

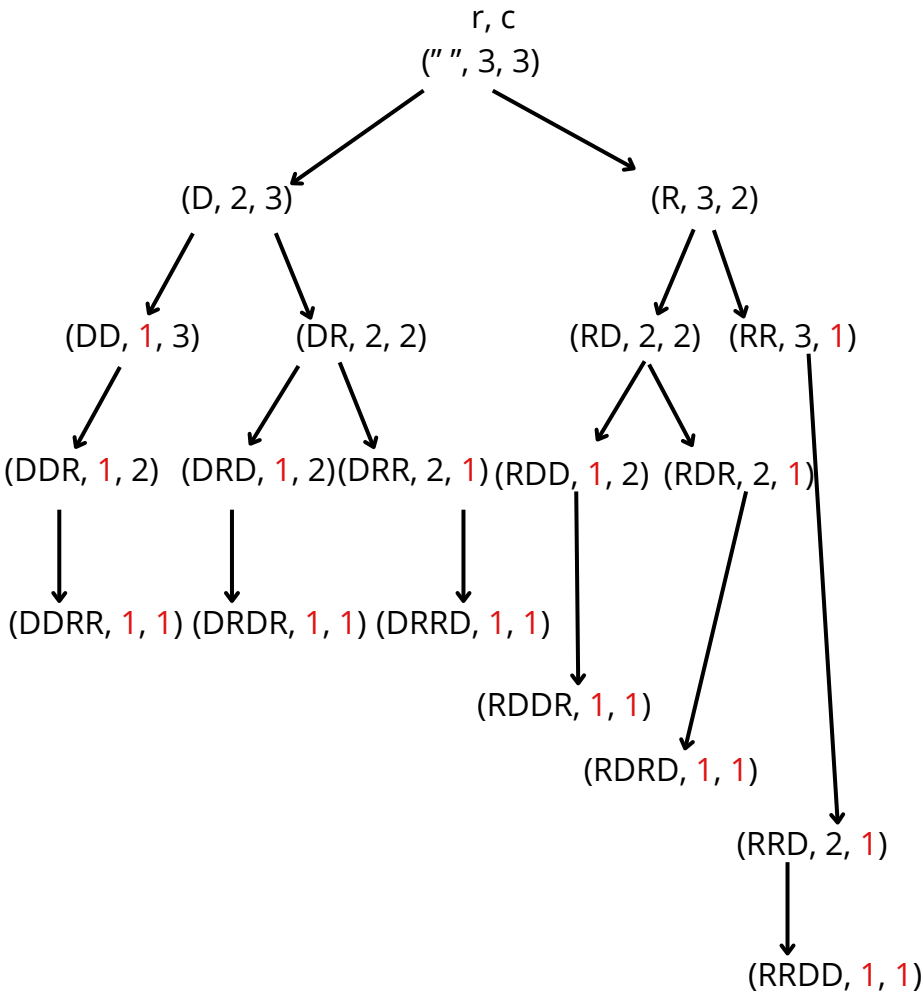
6

- **Backtracking Introduction:**

- **Backtracking (Maze Problem) :**






r, c
(" ", 3, 3)
(D, 2, 3)
(R, 3, 2)
(DD, 1, 3)
(DR, 2, 2)
(RD, 2, 2)
(RR, 3, 1)
(DDR, 1, 2)
(DRD, 1, 2)
(DRR, 2, 1)
(RDD, 1, 2)
(RDR, 2, 1)
(DDRR, 1, 1)
(DRDR, 1, 1)
(DRRD, 1, 1)
(RDDR, 1, 1)
(RDRD, 1, 1)
(RRD, 2, 1)
(RRDD, 1, 1)

- **What is the output of the following Java program fragment:**

```
#include <iostream>

using namespace std;

int count(int r, int c) {
  if (r == 1 || c == 1) {
    return 1;
  }

  int left = count(r - 1, c);
  int right = count(r, c - 1);
  return left + right;
}

int main() {
  cout << count(3, 3) << endl;
  return 0;
}
```

Output:
6

- **What is the output of the following Java program fragment:**

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

void path(string p, int r, int c) {
  if (r == 1 && c == 1) {
    cout << p << endl;
    return;
  }

  if (r > 1) {
    path(p + 'D', r - 1, c);
  }

  if (c > 1) {
    path(p + 'R', r, c - 1);
  }
}

int main() {
  path("", 3, 3);
  return 0;
}
```

Output:

```
DDRR
DRDR
DRRD
RDDR
RDRD
RRDD
```

- **What is the output of the following Java program fragment:**

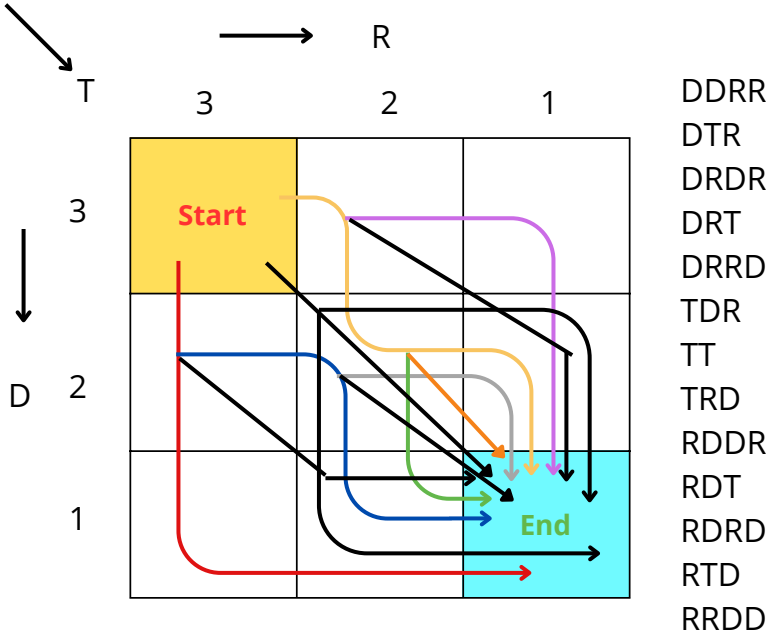
```
#include <iostream>
#include <vector>

using namespace std;

vector<string> pathRet(string p, int r, int c) {
  if (r == 1 && c == 1) {
    vector<string> list;
    list.push_back(p);
    return list;
  }

  vector<string> list;

  if (r > 1) {
    vector<string> downPaths = pathRet(p + 'D', r - 1,
c);
    list.insert(list.end(), downPaths.begin(),
downPaths.end());
  }

  if (c > 1) {
    vector<string> rightPaths = pathRet(p + 'R', r, c -
1);
    list.insert(list.end(), rightPaths.begin(),
rightPaths.end());
  }

  return list;
}
```

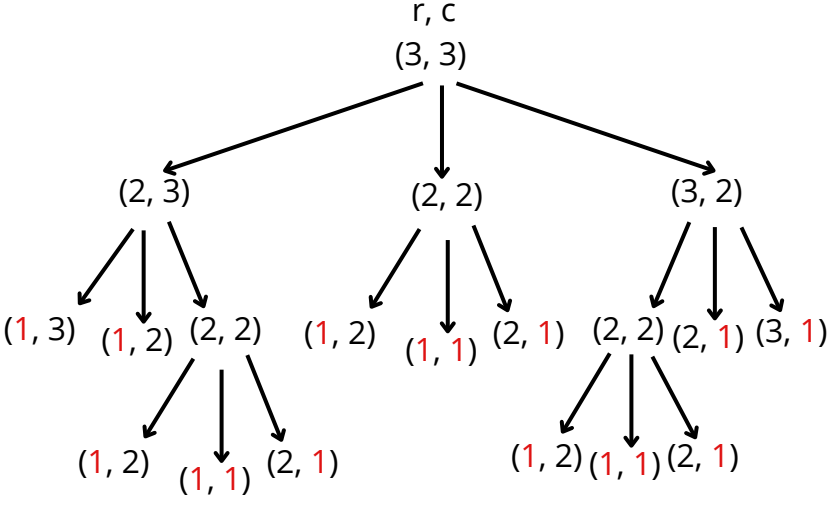
```
int main() {
  vector<string> result = pathRet("", 3, 3);

  for (const string& path : result) {
    cout << path <<" ";
  }

  return 0;
}
```

Output:

DDRR DRDR DRRD RDDR RDRD RRDD

- **Backtracking (Maze Problem--Diagonal) :**





- **What is the output of the following Java program fragment:**

```
#include <iostream>

using namespace std;

int count(int r, int c) {
  if (r == 1 || c == 1) {
    return 1;
  }

  int left = count(r - 1, c);
  int middle = count(r - 1, c - 1);
  int right = count(r, c - 1);
  return left + middle + right;
}

int main() {
  cout << count(3, 3) << endl;
  return 0;
}
```

Output:
13

- **What is the output of the following Java program fragment:**

```
#include <iostream>

using namespace std;

void path(string p, int r, int c) {
  if (r == 1 && c == 1) {
    cout << p << endl;
    return;
  }

  if (r > 1) {
    path(p + 'D', r - 1, c);
  }

  if (r > 1 && c > 1) {
    path(p + 'T', r - 1, c - 1);
  }

  if (c > 1) {
    path(p + 'R', r, c - 1);
  }
}

int main() {
  path("", 3, 3);
  return 0;
}
```

Output:

DDRR
DTR
DRDR
DRT
DRRD
TDR
TT
TRD
RDDR
RDT
RDRD
RTD
RRDD

- **What is the output of the following Java program fragment:**

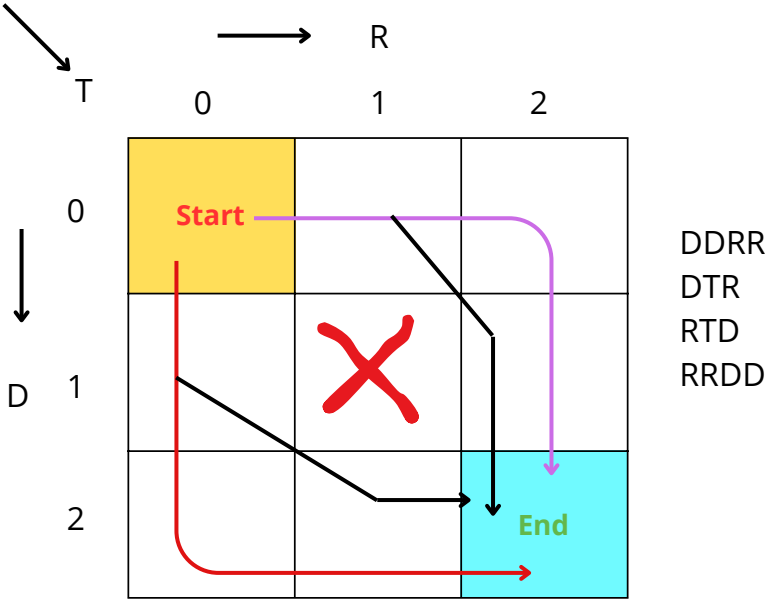
```
#include <iostream>
#include <vector>

using namespace std;

vector<string> pathRetDiagonal(string p, int r, int c)
{
  if (r == 1 && c == 1) {
    vector<string> list;
    list.push_back(p);
    return list;
  }

  vector<string> list;

  if (r > 1) {
    vector<string> downPaths = pathRetDiagonal(p +
'D', r - 1, c);
    list.insert(list.end(), downPaths.begin(),
downPaths.end());
  }

  if (c > 1 && r > 1) {
    vector<string> diagonalPaths =
pathRetDiagonal(p + 'T', r - 1, c - 1);
    list.insert(list.end(), diagonalPaths.begin(),
diagonalPaths.end());
  }
```

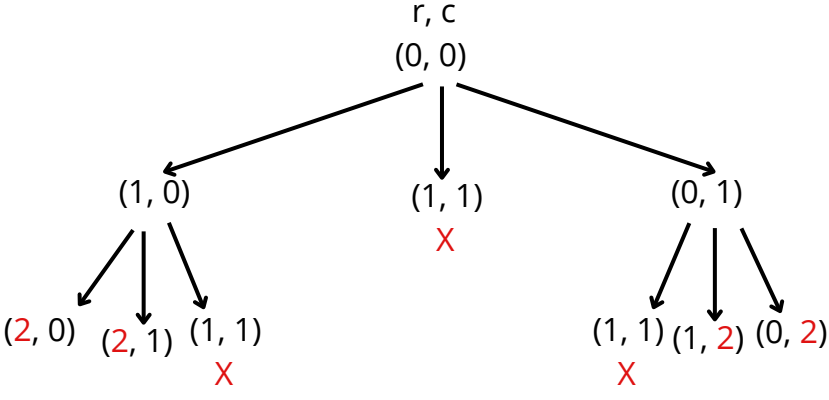
```
 if (c > 1) {
 vector<string> rightPaths = pathRetDiagonal(p +
'R', r, c - 1);
 list.insert(list.end(), rightPaths.begin(),
rightPaths.end());
 }
 return list;
}

int main() {
 vector<string> result = pathRetDiagonal("", 3, 3);

 for (const string& path : result) {
 cout << path << " ";
 }

 return 0;
}
```

Output:

DDRR DTR DRDR DRT DRRD TDR TT TRD RDDR RDT RDRD RTD RRDD

- **Backtracking (Maze Problem) :**





- **What is the output of the following Java program fragment:**

```
#include <stdio.h>

int count_start(int r, int c, int a, int b) {
 if (a == r - 1 || b == c - 1) {
  return 1;
 }

 if (a == 1 && b == 1) {
  return 0;
 }

 int left = count_start(r, c, a + 1, b);
 int middle = count_start(r, c, a + 1, b + 1);
 int right = count_start(r, c, a, b + 1);
 return left + middle + right;
}

int count(int r, int c) {
 return count_start(r, c, 0, 0);
}

int main() {
 int r, c;
 printf("Enter the value of r: ");
 scanf("%d", &r);
 printf("Enter the value of c: ");
 scanf("%d", &c);
```

```
 int result = count(r, c);
 printf("The number of paths is: %d", result);

 return 0;
}
```

Output:

4

- **What is the output of the following Java program fragment:**

```
#include <iostream>
#include <vector>

using namespace std;

void pathRestrictions(string p,
vector<vector<bool>>& maze, int r, int c) {
  if (r == maze.size() - 1 && c == maze[0].size() - 1) {
    cout << p << endl;
    return;
  }

  if (!maze[r][c]) {
    return;
  }

  if (r < maze.size() - 1) {
    pathRestrictions(p + 'D', maze, r + 1, c);
  }

  if (r < maze.size() - 1 && c < maze[0].size() - 1) {
    pathRestrictions(p + 'T', maze, r + 1, c + 1);
  }

  if (c < maze[0].size() - 1) {
    pathRestrictions(p + 'R', maze, r, c + 1);
  }
}
```

```
int main() {
 vector<vector<bool>> board = {
 {true, true, true},
 {true, false, true},
 {true, true, true},
 };

 pathRestrictions("", board, 0, 0);

 return 0;
}
```

Output:

```
DDRR
DTR
RTD
RRDD
```

- **What is the output of the following Java program fragment:**

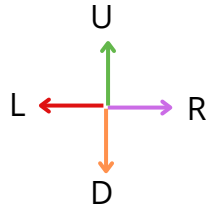
```
#include <iostream>
#include <vector>

using namespace std;

vector<string> pathRestrictions(string p,
vector<vector<bool>>& maze, int r, int c) {
  if (r == maze.size() - 1 && c == maze[0].size() - 1) {
    vector<string> list = {p};
    return list;
  }

  if (!maze[r][c]) {
    vector<string> list;
    return list;
  }

  vector<string> ans; // Initialize an empty vector

  if (r < maze.size() - 1) {
    vector<string> downPaths = pathRestrictions(p +
'D', maze, r + 1, c);
    ans.insert(ans.end(), downPaths.begin(),
downPaths.end());
  }

  if (r < maze.size() - 1 && c < maze[0].size() - 1) {
```

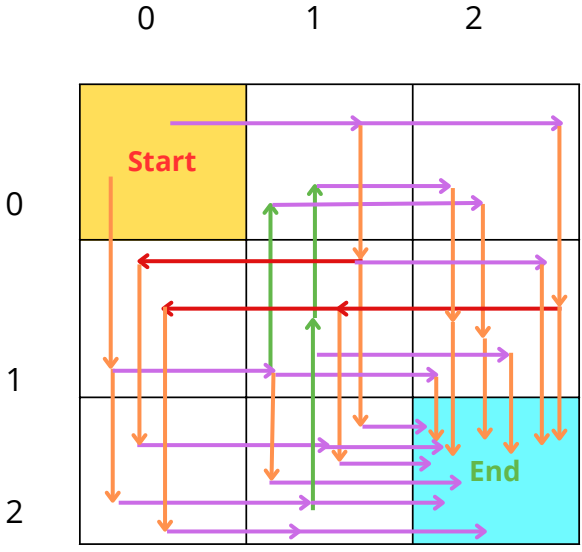
```
 vector<string> diagonalPaths = pathRestrictions(p + 'T',
maze, r + 1, c + 1);
 ans.insert(ans.end(), diagonalPaths.begin(),
diagonalPaths.end());
 }

 if (c < maze[0].size() - 1) {
 vector<string> rightPaths = pathRestrictions(p + 'R', maze,
r, c + 1);
 ans.insert(ans.end(), rightPaths.begin(), rightPaths.end());
 }

 return ans;
}

int main() {
 vector<vector<bool>> board = {
 {true, true, true},
 {true, false, true},
 {true, true, true},
 };

 vector<string> result = pathRestrictions("", board, 0, 0);

 for (const string& path : result) {
 cout << path << endl;
 }

 return 0;
}
```

Output:

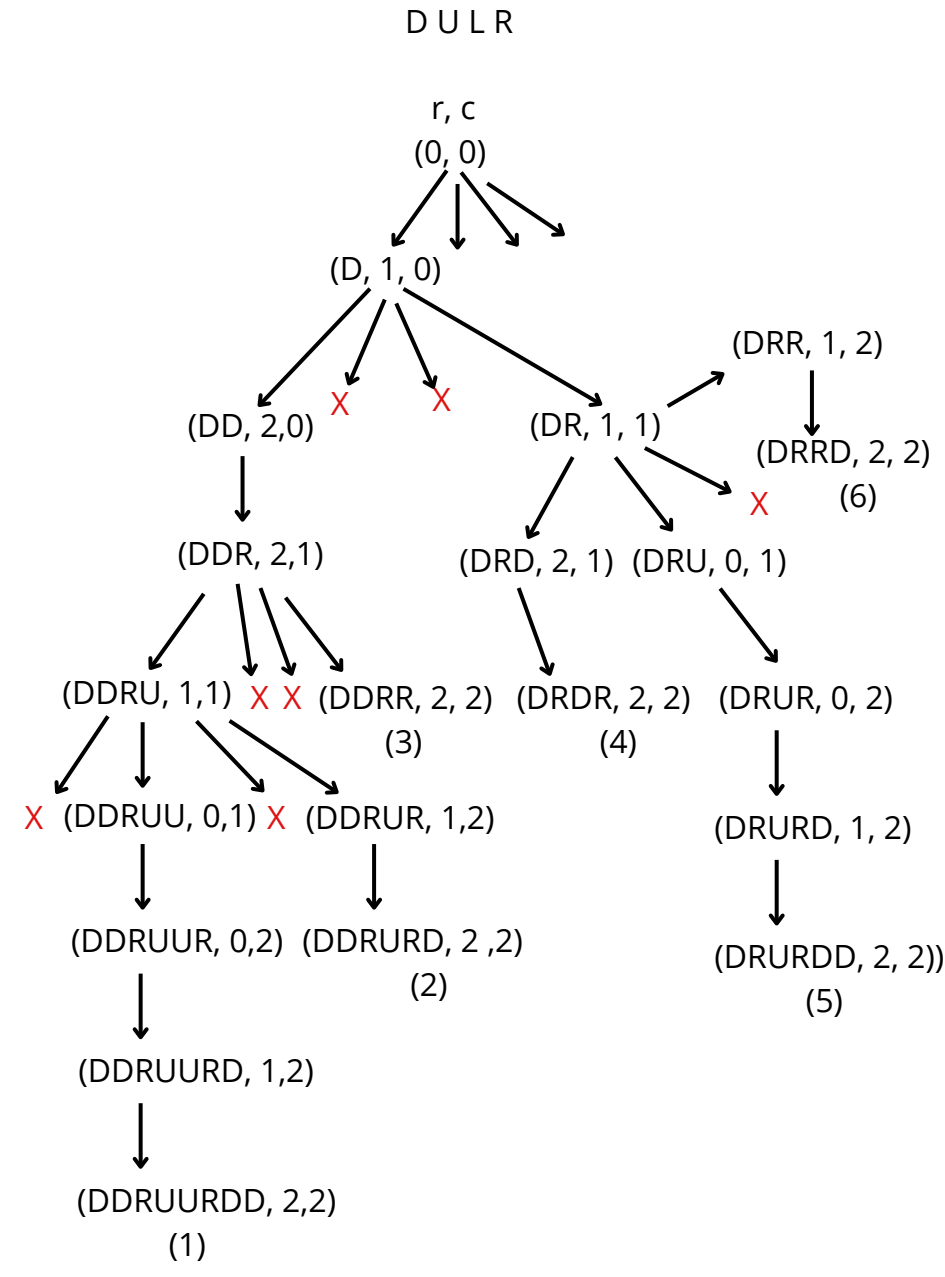
DDRR
DTR
RTD
RRDD

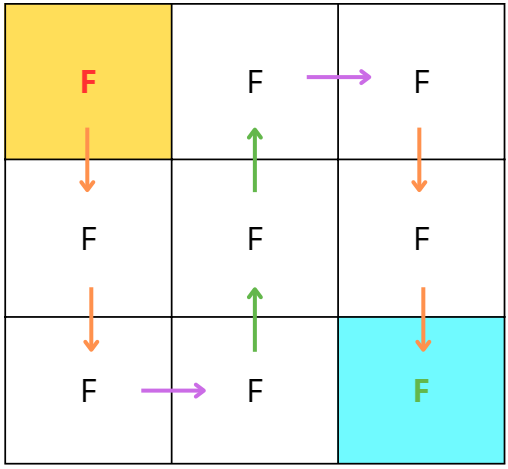
- **Backtracking (Maze Problem--Four Direction) :**





DDRUURDD
DDRURD
DDRR
DRDR
DRURDD
DRRD
RDDR
RDLDRR
RDRD
RRDD
RRDLDR
RRDLLDRR

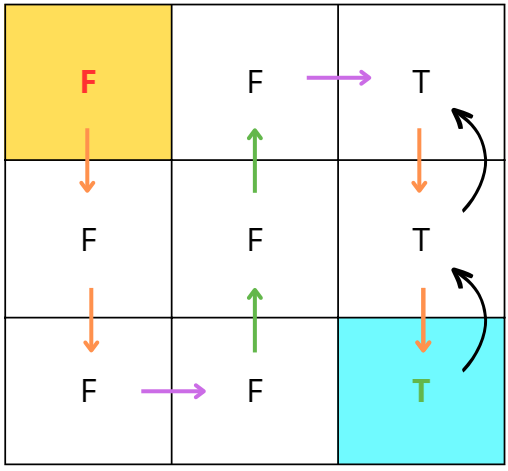

D U L R
r, c
(0, 0)
(D, 1, 0)
X
X
(DD, 2,0)
(DR, 1, 1)
(DRR, 1, 2)
(DRRD, 2, 2)
(6)
X
(DDR, 2,1)
(DRD, 2, 1)
(DRU, 0, 1)
(DDRU, 1,1)
X
X
(DDRR, 2, 2)
(3)
(DRDR, 2, 2)
(4)
(DRUR, 0, 2)
X
(DDRUU, 0,1)
X
(DDRUR, 1,2)
(DRURD, 1, 2)
(DDRUUR, 0,2)
(DDRURD, 2 ,2)
(2)
(DRURDD, 2, 2))
(5)
(DDRUURD, 1,2)
(DDRUURDD, 2,2)
(1)



(DDRUURDD, 2, 2) ---> 1st Path

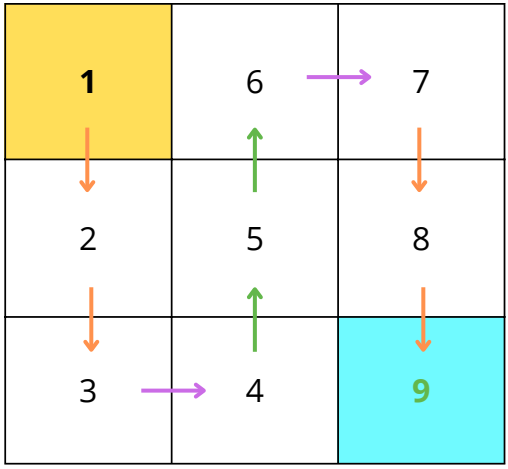
Here,

Making false means I have that cell in my current path. So, when that path is over for example you are in another cell, those cells should not be false.



(DDRUUR, 0, 2) ---> 1st Path

While you are moving back, you restore the maze as it was.



(DDRUURDD, 2, 2) ---> 1st Path

DDRUURDD
[1, 6, 7]
[2, 5, 8]
[3, 4, 9]

- **What is the output of the following Java program fragment:**

```
#include <iostream>
#include <vector>

using namespace std;

int countPath(int n, vector<vector<bool>>& maze,
int r, int c) {
  if (r == maze.size() - 1 && c == maze[0].size() - 1) {
    n++;
    return n;
  }

  if (!maze[r][c]) {
    return n;
  }

  // Consider this block in the path
  maze[r][c] = false;

  if (r < maze.size() - 1) {
    n = countPath(n, maze, r + 1, c);
  }

  if (r > 0) {
    n = countPath(n, maze, r - 1, c);
  }

  if (c > 0) {
```

```
 n = countPath(n, maze, r, c - 1);
 }

 if (c < maze[0].size() - 1) {
 n = countPath(n, maze, r, c + 1);
 }

 // Reset the block after the function call
 maze[r][c] = true;

 return n;
}

int main() {
 vector<vector<bool>> board = {
 {true, true, true},
 {true, true, true},
 {true, true, true},
 };

 cout << countPath(0, board, 0, 0) << endl;

 return 0;
}
```

Output:

12

- **What is the output of the following Java program fragment:**

```
#include <iostream>
#include <vector>

using namespace std;

void path(string p, vector<vector<bool>>& maze,
int r, int c) {
  if (r == maze.size() - 1 && c == maze[0].size() - 1) {
    cout << p << endl;
    return;
  }

  if (!maze[r][c]) {
    return;
  }

  // Consider this block in the path
  maze[r][c] = false;

  if (r < maze.size() - 1) {
    path(p + 'D', maze, r + 1, c);
  }

  if (r > 0) {
    path(p + 'U', maze, r - 1, c);
  }

  if (c > 0) {
```

```
 path(p + 'L', maze, r, c - 1);
 }

 if (c < maze[0].size() - 1) {
 path(p + 'R', maze, r, c + 1);
 }

 // Reset the block after the function call
 maze[r][c] = true;
}

int main() {
 vector<vector<bool>> board = {
 {true, true, true},
 {true, true, true},
 {true, true, true},
 };

 path("", board, 0, 0);

 return 0;
}
```

Output:

DDRUURDD
DDRURD
DDRR
DRDR
DRURDD
DRRD
RDDR
RDLDRR
RDRD
RRDD
RRDLDR
RRDLLDRR

- **What is the output of the following Java program fragment:**

```
#include <iostream>
#include <vector>

using namespace std;

vector<string> path(string p,
vector<vector<bool>>& maze, int r, int c) {
  if (r == maze.size() - 1 && c == maze[0].size() - 1) {
    vector<string> list = {p};
    return list;
  }

  if (!maze[r][c]) {
    vector<string> list;
    return list;
  }

  // Consider this block in the path
  maze[r][c] = false;
  vector<string> ans;

  if (r < maze.size() - 1) {
    vector<string> downPaths = path(p + 'D', maze, r
+ 1, c);
    ans.insert(ans.end(), downPaths.begin(),
downPaths.end());
  }
```

```
 if (r > 0) {
 vector<string> upPaths = path(p + 'U', maze, r - 1, c);
 ans.insert(ans.end(), upPaths.begin(), upPaths.end());
 }

 if (c > 0) {
 vector<string> leftPaths = path(p + 'L', maze, r, c - 1);
 ans.insert(ans.end(), leftPaths.begin(), leftPaths.end());
 }

 if (c < maze[0].size() - 1) {
 vector<string> rightPaths = path(p + 'R', maze, r, c + 1);
 ans.insert(ans.end(), rightPaths.begin(), rightPaths.end());
 }

 // Reset the block after the function call
 maze[r][c] = true;

 return ans;
}

int main() {
 vector<vector<bool>> board = {
 {true, true, true},
 {true, true, true},
 {true, true, true},
 };

 vector<string> result = path("", board, 0, 0);
```

```
 for (const string& path : result) {
 cout << path << " ";
 }

 return 0;
}
```

Output:

DDRUURDD DDRURD DDRR DRDR DRURDD DRRD
RDDR RDLDRR RDRD RRDD RRDLDR RRDLLDRR

- **What is the output of the following Java program fragment:**

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void path(string p, vector<vector<bool>>& maze,
int r, int c, vector<vector<int>>& pathStep, int step)
{
  if (r == maze.size() - 1 && c == maze[0].size() - 1) {
    cout << p << endl;

    pathStep[r][c] = step;

    for (const vector<int>& arr : pathStep) {
      for (int val : arr) {
        cout << val << " ";
      }
      cout << endl;
    }

    return;
  }

  if (!maze[r][c]) {
    return;
  }
```

```
 // Consider this block in the path
 maze[r][c] = false;
 pathStep[r][c] = step;

 if (r < maze.size() - 1) {
 path(p + 'D', maze, r + 1, c, pathStep, step + 1);
 }

 if (r > 0) {
 path(p + 'U', maze, r - 1, c, pathStep, step + 1);
 }

 if (c > 0) {
 path(p + 'L', maze, r, c - 1, pathStep, step + 1);
 }

 if (c < maze[0].size() - 1) {
 path(p + 'R', maze, r, c + 1, pathStep, step + 1);
 }

 // Reset the block after the function call
 maze[r][c] = true;
 pathStep[r][c] = 0;
}

int main() {
 vector<vector<bool>> board = {
 {true, true, true},
 {true, true, true},
 {true, true, true},
 };
```

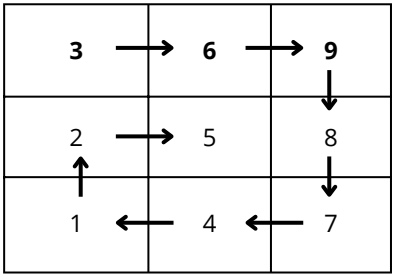
```
 vector<vector<int>> pathStep(board.size(), vector<int>
(board[0].size(), 0));

 path("", board, 0, 0, pathStep, 1);

 return 0;
}
```

Output:

DDRUURDD
1 6 7
2 5 8
3 4 9
DDRURD
1 0 0
2 5 6
3 4 7
DDRR
1 0 0
2 0 0
3 4 5
DRDR
1 0 0
2 3 0
0 4 5
DRURDD
1 4 5
2 3 6
0 0 7
DRRD
1 0 0
2 3 4
0 0 5
RDDR
1 2 0
0 3 0
0 4 5

RDLDRR
1 2 0
4 3 0
5 6 7
RDRD
1 2 0
0 3 4
0 0 5
RRDD
1 2 3
0 0 4
0 0 5
RRDLDR
1 2 3
0 5 4
0 6 7
RRDLLDRR
1 2 3
6 5 4
7 8 9

- **Write a program that takes input of the number of rows and columns of a matrix, followed by the values of the matrix. The program should have to print the matrix in a specific pattern that we provided using recursion.**



Input:
Enter row : 3
Enter column : 3
Enter value:
3 6 9
2 5 8
1 4 7

Output:
3 6 9 8 7 4 1 2 5

```
#include <iostream>

using namespace std;

void printMatrixInPattern(int** matrix, int startRow, int
startCol, int numRows, int numCols) {
  if (numRows <= 0 || numCols <= 0) {
    return;
  }

  for (int j = 0; j < numCols; j++) {
    cout << matrix[startRow][startCol + j] << " ";
  }

  for (int i = 1; i < numRows; i++) {
    cout << matrix[startRow + i][startCol + numCols - 1] << "
";
  }

  if (numRows > 1) {
    for (int j = numCols - 2; j >= 0; j--) {
      cout << matrix[startRow + numRows - 1][startCol + j]
<< " ";
    }
  }

  if (numCols > 1) {
    for (int i = numRows - 2; i > 0; i--) {
      cout << matrix[startRow + i][startCol] << " ";
    }
  }
```

```
 printMatrixInPattern(matrix, startRow + 1, startCol + 1,
numRows - 2, numCols - 2);
}

int main() {
 int rows, cols;

 cout << "Enter the number of rows: ";
 cin >> rows;

 cout << "Enter the number of columns: ";
 cin >> cols;

 int** matrix = new int*[rows];
 for (int i = 0; i < rows; i++) {
 matrix[i] = new int[cols];
 }

 cout << "Enter the values of the matrix:" << endl;

 for (int i = 0; i < rows; i++) {
 for (int j = 0; j < cols; j++) {
 cin >> matrix[i][j];
 }
 }

 printMatrixInPattern(matrix, 0, 0, rows, cols);

 // Deallocate memory
 for (int i = 0; i < rows; i++) {
 delete[] matrix[i];
 }
```

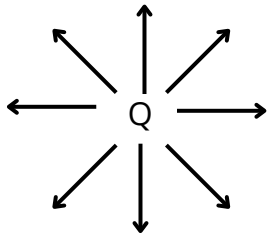
```
 delete[] matrix;

 return 0;
}
```

Output:

```
Enter the number of rows: 3
Enter the number of columns: 3
Enter the values of the matrix:
3 6 9
2 5 8
1 4 7
3 6 9 8 7 4 1 2 5
```

- **N-Queens:**

n = 4
4 Queens

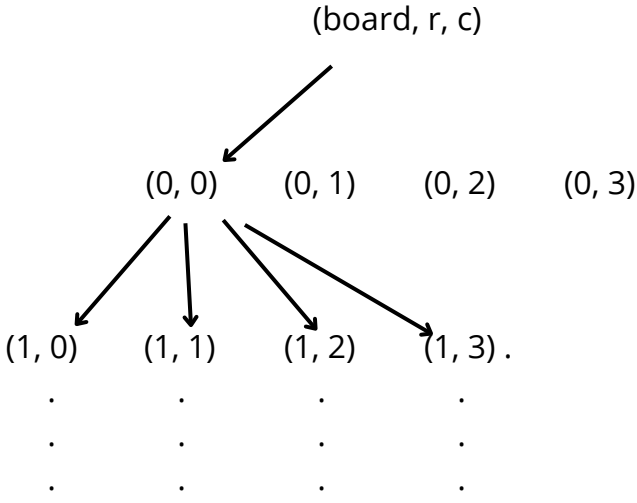


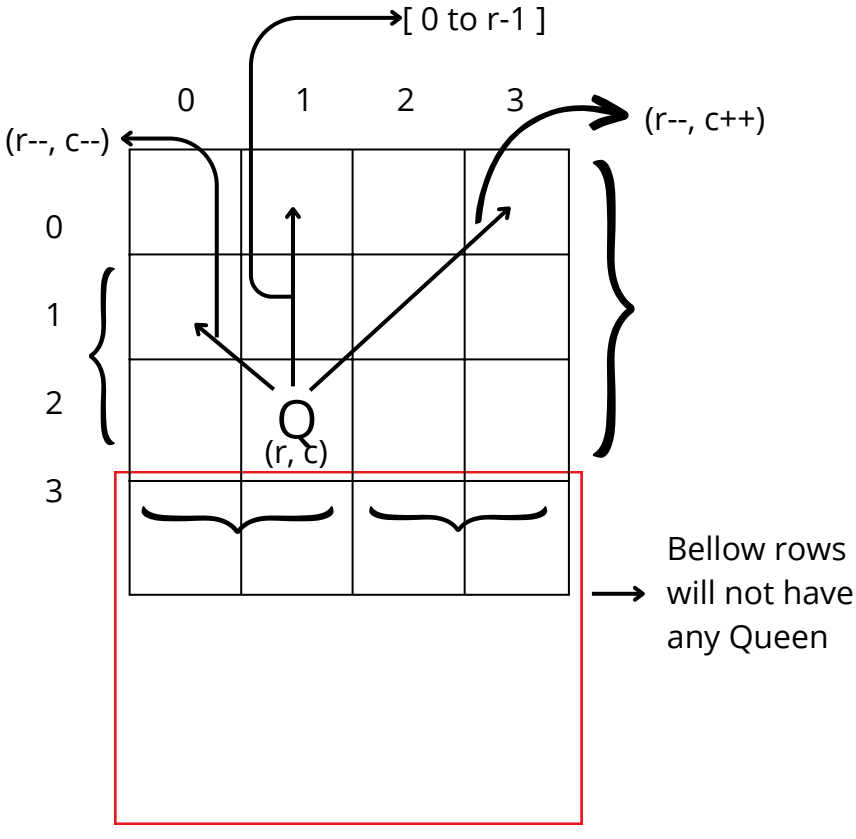
Q			
		Q	

❎

	Q		
			Q
Q			
		Q	

✅


(board, r, c)
(0, 0)
(0, 1)
(0, 2)
(0, 3)
(1, 0)
(1, 1)
(1, 2)
(1, 3)


[0 to r-1]
0
1
2
3
(r--, c--)
(r--, c++)
0
1
2
3
Q
(r, c)
Bellow rows
will not have
any Queen

- **What is the output of the following Java program fragment:**

```
#include <iostream>

using namespace std;

const int N = 4;

void display(bool board[N][N]) {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      if (board[i][j]) {
        cout << "Q ";
      } else {
        cout << "X ";
      }
    }
    cout << endl;
  }
  cout << endl;
}

bool isSafe(bool board[N][N], int row, int col) {
  // Check vertical row
  for (int i = 0; i < row; i++) {
    if (board[i][col]) {
      return false;
    }
  }

  // Diagonal left
  int maxLeft = min(row, col);
```

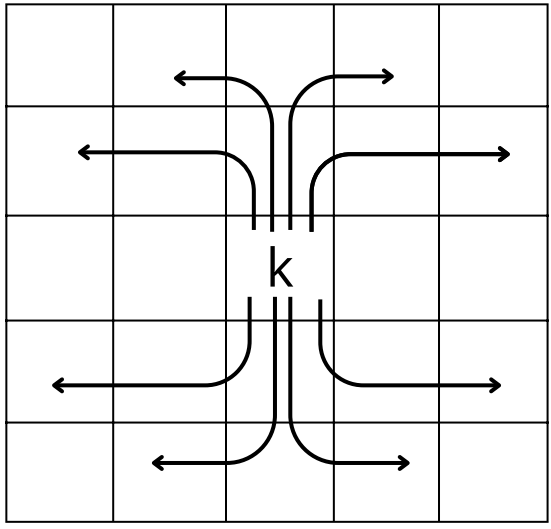
```
 for (int i = 1; i <= maxLeft; i++) {
 if (board[row - i][col - i]) {
 return false;
 }
 }

 // Diagonal right
 int maxRight = min(row, N - col - 1);
 for (int i = 1; i <= maxRight; i++) {
 if (board[row - i][col + i]) {
 return false;
 }
 }

 return true;
}

int queens(bool board[N][N], int row) {
 if (row == N) {
 display(board);
 return 1;
 }

 int count = 0;

 // Placing the queen and checking for every row and
column
 for (int col = 0; col < N; col++) {
 // Place the queen if it is safe
 if (isSafe(board, row, col)) {
 board[row][col] = true;
```

```
 count += queens(board, row + 1);
 board[row][col] = false;
 }
 }
 return count;
}

int main() {
 bool board[N][N] = {false};
 cout << queens(board, 0) << endl;

 return 0;
}
```

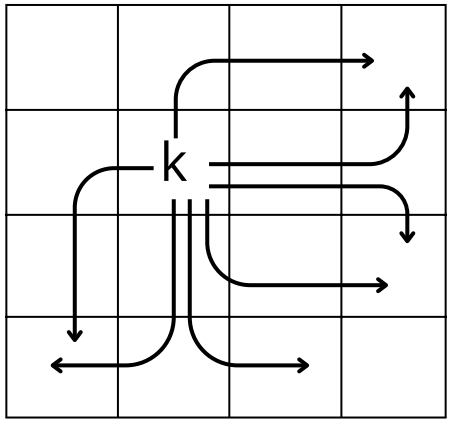
Output:

```
X Q X X
X X X Q
Q X X X
X X Q X

X X Q X
Q X X X
X X X Q
X Q X X

2
```

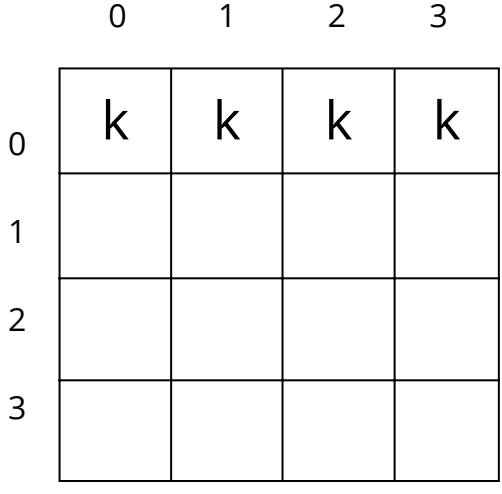
- **N-Knights:**

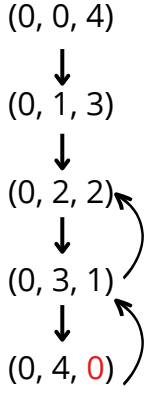


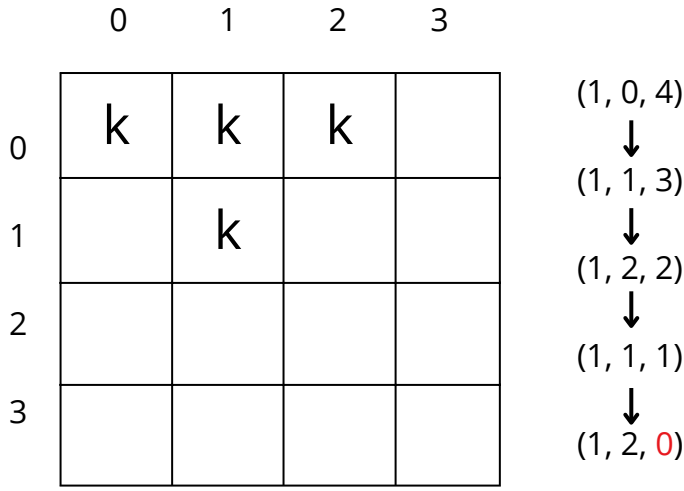
$$\begin{bmatrix} r-2, & c-1 \\ r-2, & c+1 \\ r-1, & c+2 \\ r-1, & c-2 \end{bmatrix}$$



$$\begin{bmatrix} r-2, & c-1 \\ r-2, & c+1 \\ r-1, & c+2 \\ r-1, & c-2 \end{bmatrix}$$






0
1
2
3
k
k
k
k
(1, 0, 4)
(1, 1, 3)
(1, 2, 2)
(1, 1, 1)
(1, 2, 0)

- **What is the output of the following Java program fragment:**

```
//package com.kunal.backtracking;

public class Hamim {
    public static void main(String[] args) {
        int n = 4;
        boolean[][] board = new boolean[n][n];
        knight(board, 0, 0, 4);
    }

    static void knight(boolean[][] board, int row, int col, int
knights) {
        if (knights == 0) {
            display(board);
            System.out.println();
            return;
        }

        if (row == board.length - 1 && col == board.length) {
            return;
        }

        if (col == board.length) {
            knight(board, row + 1, 0, knights);
            return;
        }

        if (isSafe(board, row, col)) {
            board[row][col] = true;
            knight(board, row, col + 1, knights - 1);
            board[row][col] = false;
```

```
    }
      knight(board, row, col + 1, knights);
    }

    private static boolean isSafe(boolean[][] board, int row,
int col) {
        if (isValid(board, row - 2, col - 1)) {
            if (board[row - 2][col - 1]) {
                return false;
            }
        }

        if (isValid(board, row - 1, col - 2)) {
            if (board[row - 1][col - 2]) {
                return false;
            }
        }

        if (isValid(board, row - 2, col + 1)) {
            if (board[row - 2][col + 1]) {
                return false;
            }
        }

        if (isValid(board, row - 1, col + 2)) {
            if (board[row - 1][col + 2]) {
                return false;
            }
        }

        return true;
    }
```

```
    // do not repeat yourself, hence created this function
    static boolean isValid(boolean[][] board, int row, int col) {
        if (row >= 0 && row < board.length && col >= 0 && col
< board.length) {
            return true;
        }
        return false;
    }

    private static void display(boolean[][] board) {
        for(boolean[] row : board) {
            for(boolean element : row) {
                if (element) {
                    System.out.print("K ");
                } else {
                    System.out.print("X ");
                }
            }
            System.out.println();
        }
    }
}
```

Output:

K K K K
X X X X
X X X X
X X X X

K K K X
X K X X
X X X X
X X X X

K K K X
X X X X
X X X X
K X X X

.....
.....
.....

- **Sudoku Solver :**

Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy all of the following rules:

1. Each of the digits 1-9 must occur exactly once in each row.
2. Each of the digits 1-9 must occur exactly once in each column.
3. Each of the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.

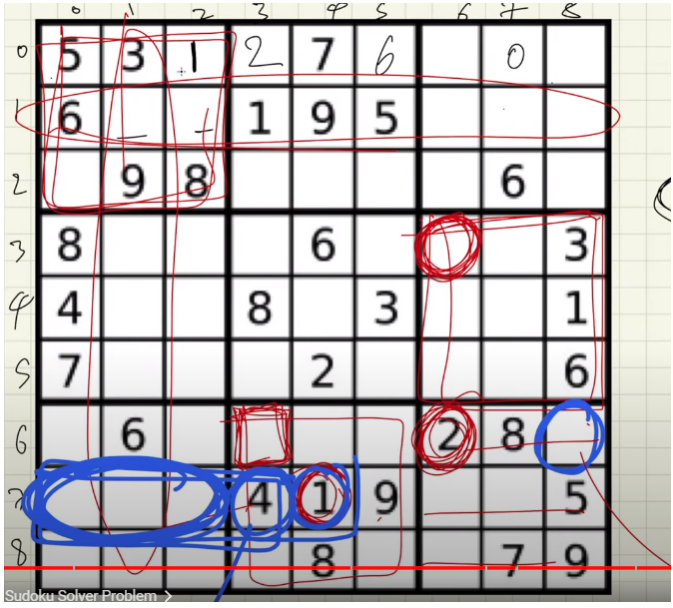
The '.' character indicates empty cells.

Input:

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

Output:

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



- **What is the output of the following Java program fragment:**

```
#include <iostream>
#include <cmath>

using namespace std;

const int N = 9;

void display(int board[N][N]) {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      cout << board[i][j] << " ";
    }
    cout << endl;
  }
}

bool isSafe(int board[N][N], int row, int col, int num) {
  // Check the row
  for (int i = 0; i < N; i++) {
    if (board[row][i] == num) {
      return false;
    }
  }

  // Check the column
  for (int i = 0; i < N; i++) {
    if (board[i][col] == num) {
      return false;
    }
  }
```

```
 // Check the 3x3 box
 int sqrtN = sqrt(N);
 int boxStartRow = row - row % sqrtN;
 int boxStartCol = col - col % sqrtN;

 for (int i = 0; i < sqrtN; i++) {
 for (int j = 0; j < sqrtN; j++) {
 if (board[boxStartRow + i][boxStartCol + j] == num) {
 return false;
 }
 }
 }

 return true;
}

bool solve(int board[N][N]) {
 int row = -1;
 int col = -1;

 bool emptyLeft = true;

 // Find the first empty cell
 for (int i = 0; i < N; i++) {
 for (int j = 0; j < N; j++) {
 if (board[i][j] == 0) {
 row = i;
 col = j;
 emptyLeft = false;
 break;
 }
 }
```

```
 if (!emptyLeft) {
 break;
 }
 }

 // If there is no empty cell, the Sudoku is solved
 if (emptyLeft) {
 return true;
 }

 // Backtrack to find the solution
 for (int num = 1; num <= 9; num++) {
 if (isSafe(board, row, col, num)) {
 board[row][col] = num;

 // Recursive call to solve the rest of the Sudoku
 if (solve(board)) {
 return true; // Found the solution
 }

 // If the current configuration doesn't lead to a
solution, backtrack
 board[row][col] = 0;
 }
 }

 return false; // No solution found
}
```

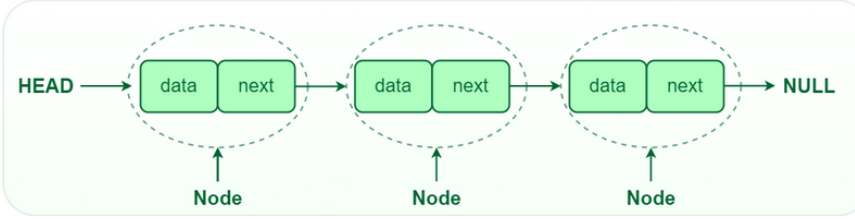
```
int main() {
 int board[N][N] = {
 {3, 0, 6, 5, 0, 8, 4, 0, 0},
 {5, 2, 0, 0, 0, 0, 0, 0, 0},
 {0, 8, 7, 0, 0, 0, 0, 3, 1},
 {0, 0, 3, 0, 1, 0, 0, 8, 0},
 {9, 0, 0, 8, 6, 3, 0, 0, 5},
 {0, 5, 0, 0, 9, 0, 6, 0, 0},
 {1, 3, 0, 0, 0, 0, 2, 5, 0},
 {0, 0, 0, 0, 0, 0, 0, 7, 4},
 {0, 0, 5, 2, 0, 6, 3, 0, 0}
 };

 if (solve(board)) {
 display(board);
 } else {
 cout << "Cannot solve" << endl;
 }

 return 0;
}
```

Output:

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

3. LinkedList

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

- **Node Structure:** A node in a linked list typically consists of two components:
- **Data:** It holds the actual value or data associated with the node.
- **Next Pointer:** It stores the memory address (reference) of the next node in the sequence.
- **Head and Tail:** The linked list is accessed through the head node, which points to the first node in the list. The last node in the list points to NULL or nullptr, indicating the end of the list. This node is known as the tail node.

Why linked list data structure needed?

Here are a few advantages of a linked list that is listed below, it will help you understand why it is necessary to know.

- **Dynamic Data structure:** The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.
- **Ease of Insertion/Deletion:** The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.
- **Efficient Memory Utilization:** As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.
- **Implementation:** Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

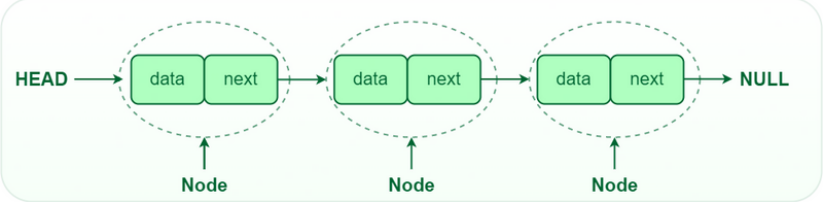
Types of linked lists:

There are mainly three types of linked lists:

- Single-linked list
- Double linked list
- Circular linked list

1. Single-linked list:

In a singly linked list, each node contains a reference to the next node in the sequence. Traversing a singly linked list is done in a forward direction.

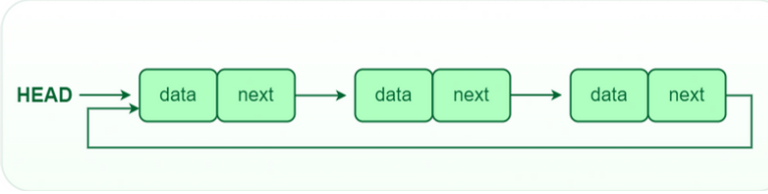


2. Double-linked list:

In a doubly linked list, each node contains references to both the next and previous nodes. This allows for traversal in both forward and backward directions, but it requires additional memory for the backward reference.

3. Circular linked list:

In a circular linked list, the last node points back to the head node, creating a circular structure. It can be either singly or doubly linked.



Operations on Linked Lists

1. **Insertion:** Adding a new node to a linked list involves adjusting the pointers of the existing nodes to maintain the proper sequence. Insertion can be performed at the beginning, end, or any position within the list
2. **Deletion:** Removing a node from a linked list requires adjusting the pointers of the neighboring nodes to bridge the gap left by the deleted node. Deletion can be performed at the beginning, end, or any position within the list.
3. **Searching:** Searching for a specific value in a linked list involves traversing the list from the head node until the value is found or the end of the list is reached.

Advantages of Linked Lists:

- **Dynamic Size:** Linked lists can grow or shrink dynamically, as memory allocation is done at runtime.
- **Insertion and Deletion:** Adding or removing elements from a linked list is efficient, especially for large lists.
- **Flexibility:** Linked lists can be easily reorganized and modified without requiring a contiguous block of memory.

Disadvantages of Linked Lists:

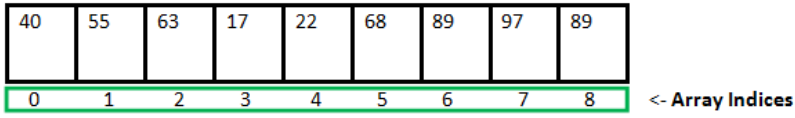
- **Random Access:** Unlike arrays, linked lists do not allow direct access to elements by index. Traversal is required to reach a specific node.
- **Extra Memory:** Linked lists require additional memory for storing the pointers, compared to arrays.

Linked lists are versatile data structures that provide dynamic memory allocation and efficient insertion and deletion operations. Understanding the basics of linked lists is essential for any programmer or computer science enthusiast. With this knowledge, you can implement linked lists to solve various problems and expand your understanding of data structures and algorithms.

Linked List vs Array:

Array:

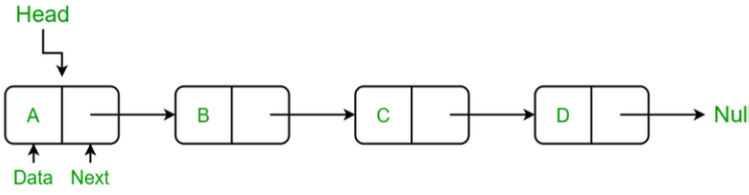
Arrays store elements in contiguous memory locations, resulting in easily calculable addresses for the elements stored and this allows faster access to an element at a specific index.



Array Length = 9
First Index = 0
Last Index = 8

Linked List:

Linked lists are less rigid in their storage structure and elements are usually not stored in contiguous locations, hence they need to be stored with additional tags giving a reference to the next element.



Major differences between array and linked-list are listed below:

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

Basic Operations:

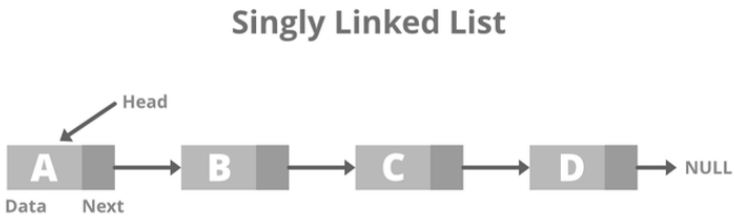
- Linked List Insertion
- Search an element in a Linked List (Iterative and Recursive)
- Find Length of a Linked List (Iterative and Recursive)
- Reverse a linked list
- Linked List Deletion (Deleting a given key)
- Linked List Deletion (Deleting a key at given position)
- Write a function to delete a Linked List
- Write a function to get Nth node in a Linked List
- Nth node from the end of a Linked List

Types Of Linked List:

1. Singly Linked List:

It is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.

The node contains a pointer to the next node means that the node stores the address of the next node in the sequence. A single linked list allows the traversal of data only in one way. Below is the image for the same:



- **Singly LinkedList implementation in C :**

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

struct Node {
   int data;
   struct Node* next;
};

typedef struct Node Node;

Node* createNode(Node* temp, int data) {
   Node* newNode = (Node*)malloc(sizeof(Node));
   newNode->data = data;
   newNode->next = NULL;
   return newNode;
}

void insertAtBegin(Node** head, int data) {
   Node* firstNode = (Node*)malloc(sizeof(Node));
   firstNode->data = data;
   firstNode->next=*head;
   *head=firstNode;

}
int insertAtEnd(Node* temp, int data) {
   Node* lastNode = (Node*)malloc(sizeof(Node));
   lastNode->data = data;
   lastNode->next = NULL;
```

```
    while(temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = lastNode;
}

void printList(Node* temp) {
    while(temp != NULL) {
        printf("%i->", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main()
{
    Node* head = NULL;
    head = createNode(head, 5);
    printf("List is: ");
    printList(head);
    insertAtEnd(head,6);
    insertAtEnd(head,7);
    insertAtEnd(head,8);
    printf("List is: ");
    printList(head);

    insertAtBegin(&head,40);
    printf("List is: ");
    printList(head);

    return 0;
}
```

Output:

List is: 5->NULL
List is: 5->6->7->8->NULL
List is: 40->5->6->7->8->NULL

- **Singly LinkedList implementation in C :**

```
// Singly LinkedList
#include<stdio.h>
#include<stdlib.h>

struct Node{
  int data;
  struct Node *next;
};

typedef struct Node Node;
Node *head = NULL;
int size = 0;

Node *createNode(int data){
  Node *newNode = (Node*)
malloc(sizeof(Node));
  if(newNode == NULL){
    printf("Error!!! Memory allocation failed\n");
    exit(EXIT_FAILURE);
  }
  //(*newNode).data = data;
  newNode->data = data;
  newNode->next = NULL;
  return newNode;
}

void insertAtFirst(int data){
  Node *newNode = createNode(data);
  if(head == NULL){
```

```
    head = newNode;
  }else{
    newNode->next = head;
    head = newNode;
  }
  size++;
}

void insertAtEnd(int data){
  Node *newNode = createNode(data);
  if(head == NULL){
    head = newNode;
  }else{
    Node *current = head;

    while(current->next != NULL){
      current = current->next;
    }

    current->next = newNode;
  }
  size++;
}

int isEmpty(){
  if(head == NULL){
    return 1;
  }
  return 0;
}
```

```
int getSize(){
  return size;
}

void insertAtIndex(int index, int data){
  Node *newtNode = createNode(data);
  Node* current = head;
  int i=0;
  if(isEmpty()){
    printf("LinkedList is empty!!!\n");
  }else if(index > -1 && index < getSize()+1){
    Node *temp=NULL;
    while(i<index){
      temp=current;
      current = current->next;
      i++;
    }
    newtNode->next = current;
    temp->next = newtNode;
    size++;
  }else{
    printf("Index out of bounds!!!\nYou can insert
in 0 to %d\n", getSize());
  }
}

void printList(){
  Node *current = head;
  printf("[");
  while(current != NULL){
```

```
    printf("%d, ", current->data);
    current = current-> next;
  }
  printf("NULL]\n");
}
void printrev(Node *current){

  if(current == NULL){
    return;
  }
  printrev(current->next);
  printf("%d, ", current->data);
}

void printReverse(){
  Node *current = head;
  printf("[");
  printrev(current);
  printf("NULL]\n");
}

int elementAt(int index){
  Node* current = head;
  int i=0;
  if(isEmpty()){
    printf("LinkedList is empty!!!\n");
  }else if(index > -1 && index < getSize()){
    Node *temp=NULL;
    while(i<=index){
      temp=current;
```

```
      current = current->next;
      i++;
    }
    return temp->data;
  }else{
    printf("Index out of bounds!!!\n");
  }
  return i;
}

void deleteAtFirst(){
  Node *current = head;
  head = current->next;
  free(current);
  size--;
}

void deleteAtEnd(){
  Node *current = head;
  while(current->next->next!=NULL){
    current = current->next;
  }
  free(current->next);
  current->next = NULL;
  size--;
}

void deleteAtIndex(int index){
  Node* current = head;
  int i=0;
```

```
  if(isEmpty()){
    printf("LinkedList is empty!!!\n");
  }else if(index > -1 && index < getSize()){
    Node *temp=NULL;
    while(i<index){
      temp=current;
      current = current->next;
      i++;
    }
    temp->next = current->next;
    free(current);
    size--;
  }else{
    printf("Index out of bounds!!!\nYou can delete
in 0 to %d\n", getSize()-1);
  }
}

int main(){
  insertAtIndex(4, 13);
  printf("Size of LinkedList: %d\n",getSize());
  printList();
  insertAtEnd(1);
  printList();
  insertAtEnd(2);
  printList();
  insertAtFirst(7);
  printList();
  insertAtFirst(8);
  printList();
```

```
  printReverse();
  printf("Size of LinkedList: %d\n",getSize());
  insertAtIndex(4, 13);
  printList();
  insertAtIndex(4, 12);
  printList();
  insertAtIndex(2, 12);
  printList();
  printf("%d is at 4\n", elementAt(4));
  deleteAtEnd();
  printList();
  deleteAtEnd();
  printList();
  deleteAtFirst();
  printList();
  deleteAtIndex(1);
  printList();
  deleteAtIndex(-1);
  printList();
}
```

Output:

```
LinkedList is empty!!!
Size of LinkedList: 0
[NULL]
[1, NULL]
[1, 2, NULL]
[7, 1, 2, NULL]
[8, 7, 1, 2, NULL]
[2, 1, 7, 8, NULL]
Size of LinkedList: 4
[8, 7, 1, 2, 13, NULL]
[8, 7, 1, 2, 12, 13, NULL]
[8, 7, 12, 1, 2, 12, 13, NULL]
2 is at 4
[8, 7, 12, 1, 2, 12, NULL]
[8, 7, 12, 1, 2, NULL]
[7, 12, 1, 2, NULL]
[7, 1, 2, NULL]
Index out of bounds!!!
You can delete in 0 to 2
[7, 1, 2, NULL]
```

- **Singly LinkedList implementation in C :**

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

typedef struct Node {
  int data;
  struct Node* next;
} Node;

Node* createNode(int data) {
  Node* newNode = (Node*)malloc(sizeof(Node));
  if (newNode == NULL) {
    perror("Memory allocation failed");
    exit(EXIT_FAILURE);
  }
  newNode->data = data;
  newNode->next = NULL;
  return newNode;
}

void append(Node** head, int data) {
  Node* newNode = createNode(data);
  if (*head == NULL) {
    *head = newNode;
  } else {
    Node* current = *head;
    while (current->next != NULL) {
      current = current->next;
    }
    current->next = newNode;
```

```
    }
}

void printList(Node* head) {
    Node* current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

int searchIterative(Node* head, int key) {
    Node* current = head;
    int position = 0;
    while (current != NULL) {
        if (current->data == key) {
            return position;
        }
        current = current->next;
        position++;
    }
    return -1;
}

int searchRecursive(Node* head, int key) {
    if (head == NULL) {
        return -1;
    }
    if (head->data == key) {
```

```
        return 0;
    }
    int position = searchRecursive(head->next, key);
    if (position == -1) {
        return -1;
    }
    return position + 1;
}

int findLengthIterative(Node* head) {
    int length = 0;
    Node* current = head;
    while (current != NULL) {
        length++;
        current = current->next;
    }
    return length;
}


int findLengthRecursive(Node* head) {
  if (head == NULL) {
    return 0;
  }
  return 1 + findLengthRecursive(head->next);
}

void reverseList(Node** head) {
  Node* prev = NULL;
  Node* current = *head;
  Node* next = NULL;
```

```
  while (current != NULL) {
    next = current->next;
    current->next = prev;
    prev = current;
    current = next;
  }

  *head = prev;
}

void deleteKey(Node** head, int key) {
  Node* current = *head;
  Node* prev = NULL;

  if (current != NULL && current->data == key) {
    *head = current->next;
    free(current);
    return;
  }

  while (current != NULL && current->data != key)
{
    prev = current;
    current = current->next;
  }

  if (current == NULL) {
    return;
  }
```

```
  prev->next = current->next;
  free(current);
}

void deletePosition(Node** head, int position) {
  if (*head == NULL) {
    return;
  }

  Node* current = *head;

  if (position == 0) {
    *head = current->next;
    free(current);
    return;
  }

  for (int i = 0; current != NULL && i < position - 1;
i++) {
    current = current->next;
  }

  if (current == NULL || current->next == NULL) {
    return;
  }

  Node* next = current->next->next;
  free(current->next);
  current->next = next;
}
```

```
void deleteList(Node** head) {
    Node* current = *head;
    Node* next;

    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }

    *head = NULL;
}

int main() {
    Node* head = NULL;

    append(&head, 1);
    append(&head, 2);
    append(&head, 3);
    printf("Initial Linked List: ");
    printList(head);

    int searchKey = 2;
    int searchPositionIterative = searchIterative(head,
searchKey);
if (searchPositionIterative != -1) {
    printf("%d found at position %d (Iterative)\n",
searchKey, searchPositionIterative);
  } else {
    printf("%d not found (Iterative)\n", searchKey);
  }
```

```
int searchPositionRecursive = searchRecursive(head,
searchKey);
    if (searchPositionRecursive != -1) {
        printf("%d found at position %d (Recursive)\n",
searchKey, searchPositionRecursive);
    } else {
        printf("%d not found (Recursive)\n", searchKey);
    }

    int lengthIterative = findLengthIterative(head);
    printf("Length of the Linked List (Iterative): %d\n",
lengthIterative);

    int lengthRecursive = findLengthRecursive(head);
    printf("Length of the Linked List (Recursive): %d\n",
lengthRecursive);

    reverseList(&head);
    printf("Reversed Linked List: ");
    printList(head);

    int keyToDelete = 2;
    deleteKey(&head, keyToDelete);
    printf("Linked List after deleting %d: ", keyToDelete);
    printList(head);

    int positionToDelete = 1;
    deletePosition(&head, positionToDelete);
    printf("Linked List after deleting node at position %d:
", positionToDelete);
    printList(head);
```

```
 deleteList(&head);
 printf("Linked List deleted.\n");

 return 0;
}
```

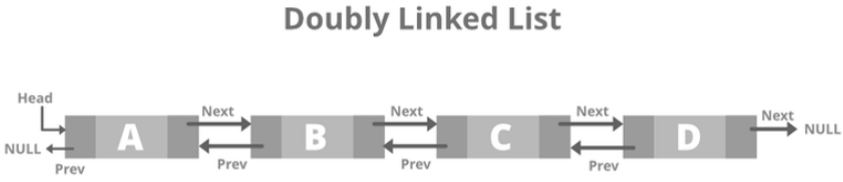
Output:

```
Initial Linked List: 1 -> 2 -> 3 -> NULL
2 found at position 1 (Iterative)
2 found at position 1 (Recursive)
Length of the Linked List (Iterative): 3
Length of the Linked List (Recursive): 3
Reversed Linked List: 3 -> 2 -> 1 -> NULL
Linked List after deleting 2: 3 -> 1 -> NULL
Linked List after deleting node at position 1: 3 -> NULL
Linked List deleted.
```

2. Doubly Linked List:

A doubly linked list or a two-way linked list is a more complex type of linked list that contains a pointer to the next as well as the previous node in sequence.

Therefore, it contains three parts of data, a pointer to the next node, and a pointer to the previous node. This would enable us to traverse the list in the backward direction as well. Below is the image for the same:



- **Doubly LinkedList implementation in C :**

```
/* Doubly LinkedList implementation (part-2)*/

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

struct Node {
  int data;
  struct Node* next;
  struct Node* prev;
};

struct Node* head; // global variable - pointer to
head node.

//Creates a new Node and returns pointer to it.
struct Node* GetNewNode(int x) {
  struct Node* newNode = (struct
Node*)malloc(sizeof(struct Node));
  newNode->data = x;
  newNode->prev = NULL;
  newNode->next = NULL;
  return newNode;
}

//Inserts a Node at head of doubly linked list
void InsertAtHead(int x) {
  struct Node* newNode = GetNewNode(x);
  if(head == NULL) {
    head = newNode;
```

```
    return;
  }
  head->prev = newNode;
  newNode->next = head;
  head = newNode;
}

//Inserts a Node at tail of Doubly linked list
void InsertAtTail(int x) {
  struct Node* temp = head;
  struct Node* newNode = GetNewNode(x);
  if(head == NULL) {
    head = newNode;
    return;
  }
  while(temp->next != NULL) temp = temp->next; // Go
To last Node
  temp->next = newNode;
  newNode->prev = temp;
}

//Prints all the elements in linked list in forward
traversal order
void Print() {
  struct Node* temp = head;
  printf("Forward: ");
  while(temp != NULL) {
    printf("%d ",temp->data);
    temp = temp->next;
  }
  printf("\n");
}
```

```
//Prints all elements in linked list in reverse traversal
order.
void ReversePrint() {
  struct Node* temp = head;
  if(temp == NULL) return; // empty list, exit
  // Going to last Node
  while(temp->next != NULL) {
    temp = temp->next;
  }
  // Traversing backward using prev pointer
  printf("Reverse: ");
  while(temp != NULL) {
    printf("%d ",temp->data);
    temp = temp->prev;
  }
  printf("\n");
}

int main() {

  //Driver code to test the implementation
  head = NULL; // empty list. set head as NULL.

  // Calling an Insert and printing list both in forward
as well as reverse direction.
  InsertAtTail(2); Print(); ReversePrint();
  InsertAtTail(4); Print(); ReversePrint();
  InsertAtHead(6); Print(); ReversePrint();
  InsertAtTail(8); Print(); ReversePrint();

}
```

Output:

Forward: 2
Reverse: 2
Forward: 2 4
Reverse: 4 2
Forward: 6 2 4
Reverse: 4 2 6
Forward: 6 2 4 8
Reverse: 8 4 2 6

- **Doubly LinkedList implementation in C :**

```
/* Doubly LinkedList implementation (part-1)*/

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

typedef struct Student {
   char name[20];
   int age;
   float height;
}Student;

typedef struct Node {
   Student student;
   struct Node* next;
   struct Node* prev;
}Node;

Node* head = NULL;

Node* getNewNode(Student x)  {
   Node* newNode = (Node*)malloc(sizeof(Node));
   newNode->student = x;
   newNode->next = NULL;
   newNode->prev = NULL;
   return newNode;
}

void insertAtBegin(Student x) {
   Node* newNode = getNewNode(x);
```

```
   if(head == NULL) {
      head = newNode;
      return;
   }

   newNode->next = head;
   head->prev = newNode;
   head = newNode;
}


void insertAtEnd(Student x) {
   Node* newNode = getNewNode(x);
   if(head == NULL) {
      head = newNode;
      return;
   }

   Node* temp = head;
   while(temp->next != NULL) {
      temp=temp->next;
   }
   temp->next = newNode;
   newNode->prev = temp;
}

void print(){
   if(head == NULL) {
      printf("Linked List is Empty\n");
      return;
   }
```

```
    Node* temp = head;
    printf("\nStudent Information (Forward):\n");
    printf("Name\tAge\tHeight\n");
    while(temp != NULL) {
        printf("%s\t%i\t%.2f\n", temp->student.name,
temp->student.age, temp->student.height);
        temp = temp->next;
    }
    printf("NULL\n");
}

void reversePrint(){
    if(head == NULL) {
        printf("Linked List is Empty\n");
        return;
    }

    Node* temp = head;
    while(temp->next != NULL)
        temp = temp->next;

    printf("\nStudent Information (Reverse):\n");
    printf("Name\tAge\tHeight\n");
    while(temp != NULL) {
        printf("%s\t%i\t%.2f\n", temp->student.name,
temp->student.age, temp->student.height);
        temp = temp->prev;
    }
    printf("NULL\n");
}
```

```
int main()
{
    int n;
    Student student;
    printf("Enter number of students: ");
    scanf("%d", &n);

    for(int i = 0; i < n; i++) {
        printf("Input information for student no
%d\n",i+1);
        printf("Name: ");
        scanf(" %[^\n]", student.name);
        printf("Age: ");
        scanf("%d", &student.age);
        printf("Height: ");
        scanf("%f", &student.height);
        insertAtEnd(student);
    }

    print();
    reversePrint();

    return 0;
}
```

Output:

```
Enter number of students: 3
Input information for student no 1
Name: Hamim Talukder
Age: 23
Height: 5.9
Input information for student no 2
Name: Hridi Chowdhury
Age: 24
Height: 5.5
Input information for student no 3
Name: Jim Talukder
Age: 22
Height: 5.5

Student Information (Forward):
Name    Age     Height
Hamim Talukder  23      5.90
Hridi Chowdhury 24      5.50
Jim Talukder    22      5.50
NULL

Student Information (Reverse):
Name    Age     Height
Jim Talukder    22      5.50
Hridi Chowdhury 24      5.50
Hamim Talukder  23      5.90
NULL
```

- **Doubly LinkedList implementation in C++ :**

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
 int data;
 Node* next;
 Node* prev;
};

void push(Node** head_ref, int new_data)
{
 Node* new_node = new Node();

 new_node->data = new_data;

 new_node->next = (*head_ref);
 new_node->prev = NULL;

 if ((*head_ref) != NULL)
  (*head_ref)->prev = new_node;

 (*head_ref) = new_node;
}

void printList(Node* node)
{
 Node* last;
```

```
 cout << "\nTraversal in forward"
  << " direction \n";
 while (node != NULL) {

  cout << " " << node->data << " ";
  last = node;
  node = node->next;
 }

 cout << "\nTraversal in reverse"
  << " direction \n";
 while (last != NULL) {

  cout << " " << last->data << " ";
  last = last->prev;
 }
}

int main()
{
 Node* head = NULL;

 push(&head, 6);

 push(&head, 7);

 push(&head, 1);

 cout << "Created DLL is: ";
 printList(head);
 return 0;
}
```

Output:

```
Created DLL is:
Traversal in forward direction
 1  7  6
Traversal in reverse direction
 6  7  1
```

Time Complexity:

The time complexity of the push() function is O(1) as it performs constant-time operations to insert a new node at the beginning of the doubly linked list. The time complexity of the printList() function is O(n) where n is the number of nodes in the doubly linked list. This is because it traverses the entire list twice, once in the forward direction and once in the backward direction. Therefore, the overall time complexity of the program is O(n).

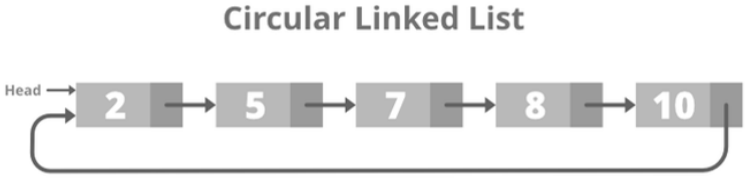
Space Complexity:

The space complexity of the program is O(n) as it uses a doubly linked list to store the data, which requires n nodes. Additionally, it uses a constant amount of auxiliary space to create a new node in the push() function. Therefore, the overall space complexity of the program is O(n).

3. Circular Linked List:

A circular linked list is that in which the last node contains the pointer to the first node of the list.

While traversing a circular linked list, we can begin at any node and traverse the list in any direction forward and backward until we reach the same node we started. Thus, a circular linked list has no beginning and no end. Below is the image for the same:



- **Circular LinkedList implementation in C++ :**

```
#include <bits/stdc++.h>
using namespace std;

class Node
{
public:
 int data;
 Node *next;
};

void push(Node **head_ref, int data)
{
 Node *ptr1 = new Node();
 Node *temp = *head_ref;
 ptr1->data = data;
 ptr1->next = *head_ref;

 if (*head_ref != NULL)
 {
  while (temp->next != *head_ref)
  {
   temp = temp->next;
  }
  temp->next = ptr1;
 }

 else
  ptr1->next = ptr1;
```

```
 *head_ref = ptr1;
}

void printList(Node *head)
{
 Node *temp = head;
 if (head != NULL)
 {
  do
  {

   cout << temp->data << " ";
   temp = temp->next;
  } while (temp != head);
 }
}

int main()
{
 Node *head = NULL;

 push(&head, 12);
 push(&head, 56);
 push(&head, 2);
 push(&head, 11);

 cout << "Contents of Circular"<< " Linked List\n ";

 printList(head);

 return 0;
}
```

Output:
Contents of Circular Linked List
11 2 56 12

Time Complexity:

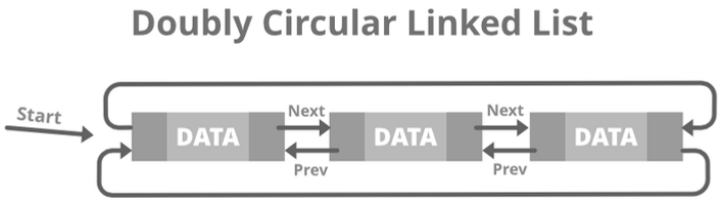
Insertion at the beginning of the circular linked list takes O(1) time complexity.
Traversing and printing all nodes in the circular linked list takes O(n) time complexity where n is the number of nodes in the linked list.
Therefore, the overall time complexity of the program is O(n).

Auxiliary Space:

The space required by the program depends on the number of nodes in the circular linked list.
In the worst-case scenario, when there are n nodes, the space complexity of the program will be O(n) as n new nodes will be created to store the data.
Additionally, some extra space is required for the temporary variables and the function calls.
Therefore, the auxiliary space complexity of the program is O(n).

4. Doubly Circular linked list:

A Doubly Circular linked list or a circular two-way linked list is a more complex type of linked list that contains a pointer to the next as well as the previous node in the sequence. The difference between the doubly linked and circular doubly list is the same as that between a singly linked list and a circular linked list. The circular doubly linked list does not contain null in the previous field of the first node. Below is the image for the same:



- **Doubly Circular LinkedList implementation in C++ :**

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
 int data;
 struct Node* next;
 struct Node* prev;
};

void insertBegin(struct Node** start, int value)
{
 if (*start == NULL) {
  struct Node* new_node = new Node;
  new_node->data = value;
  new_node->next = new_node->prev = new_node;
  *start = new_node;
  return;
 }

 struct Node* last = (*start)->prev;

 struct Node* new_node = new Node;

 new_node->data = value;

 new_node->next = *start;
 new_node->prev = last;
```

```
 last->next = (*start)->prev = new_node;

 *start = new_node;
}

void display(struct Node* start)
{
 struct Node* temp = start;

 printf("\nTraversal in"
  " forward direction \n");
 while (temp->next != start) {
  printf("%d ", temp->data);
  temp = temp->next;
 }
 printf("%d ", temp->data);

 printf("\nTraversal in "
  "reverse direction \n");
 Node* last = start->prev;
 temp = last;

 while (temp->prev != last) {

  printf("%d ", temp->data);
  temp = temp->prev;
 }
 printf("%d ", temp->data);
}
```

```
int main()
{
 struct Node* start = NULL;

 insertBegin(&start, 5);

 insertBegin(&start, 4);

 insertBegin(&start, 7);

 printf("Created circular doubly linked list is: ");
 display(start);

 return 0;
}
```

Output:
Created circular doubly linked list is:
Traversal in forward direction
7 4 5
Traversal in reverse direction
5 4 7

Time Complexity:

Insertion at the beginning of a doubly circular linked list takes O(1) time complexity.
Traversing the entire doubly circular linked list takes O(n) time complexity, where n is the number of nodes in the linked list.
Therefore, the overall time complexity of the program is O(n).

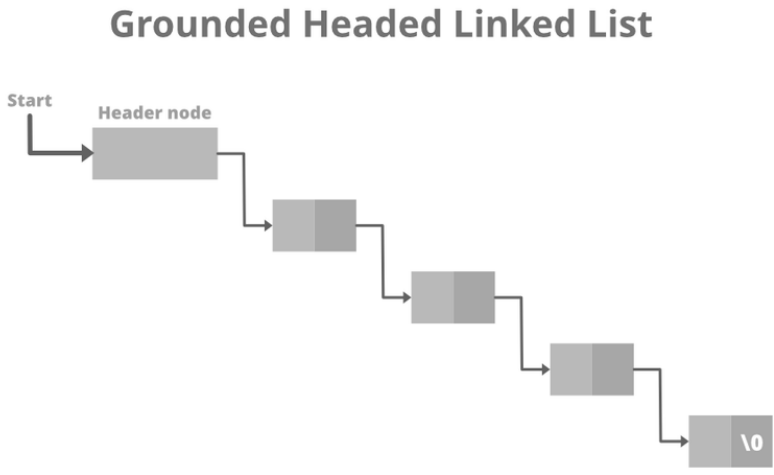
Auxiliary space:

The program uses a constant amount of auxiliary space, i.e., O(1), to create and traverse the doubly circular linked list.
The space required to store the linked list grows linearly with the number of nodes in the linked list.
Therefore, the overall auxiliary space complexity of the program is O(1).

5. Header Linked List:

A header linked list is a special type of linked list that contains a header node at the beginning of the list.

So, in a header linked list START will not point to the first node of the list but START will contain the address of the header node. Below is the image for Grounded Header Linked List:



- **Header LinkedList implementation in C++ :**

```
#include <bits/stdc++.h>

struct link {
 int info;
 struct link* next;
};

struct link* start = NULL;

struct link* create_header_list(int data)
{

 struct link *new_node, *node;
 new_node = (struct link*)malloc(sizeof(struct link));
 new_node->info = data;
 new_node->next = NULL;

 if (start == NULL) {

  start = (struct link*)malloc(sizeof(struct link));
  start->next = new_node;
 }
 else {

  node = start;
  while (node->next != NULL) {
   node = node->next;
  }
```

```
  node->next = new_node;
 }
 return start;
}

struct link* display()
{
 struct link* node;
 node = start;
 node = node->next;

 while (node != NULL) {

  printf("%d ", node->info);
  node = node->next;
 }
 printf("\n");

 return start;
}

int main()
{
 create_header_list(11);
 create_header_list(12);
 create_header_list(13);

 printf("List After inserting 3 elements:\n");
 display();
 create_header_list(14);
 create_header_list(15);
```

```
 printf("List After inserting 2 more elements:\n");
 display();

 return 0;
}
```

Output:

List After inserting 3 elements:
11 12 13
List After inserting 2 more elements:
11 12 13 14 15

Time Complexity:

The time complexity of creating a new node and inserting it at the end of the linked list is O(1).
The time complexity of traversing the linked list to display its contents is O(n), where n is the number of nodes in the list.
Therefore, the overall time complexity of creating and traversing a header linked list is O(n).

Auxiliary Space:

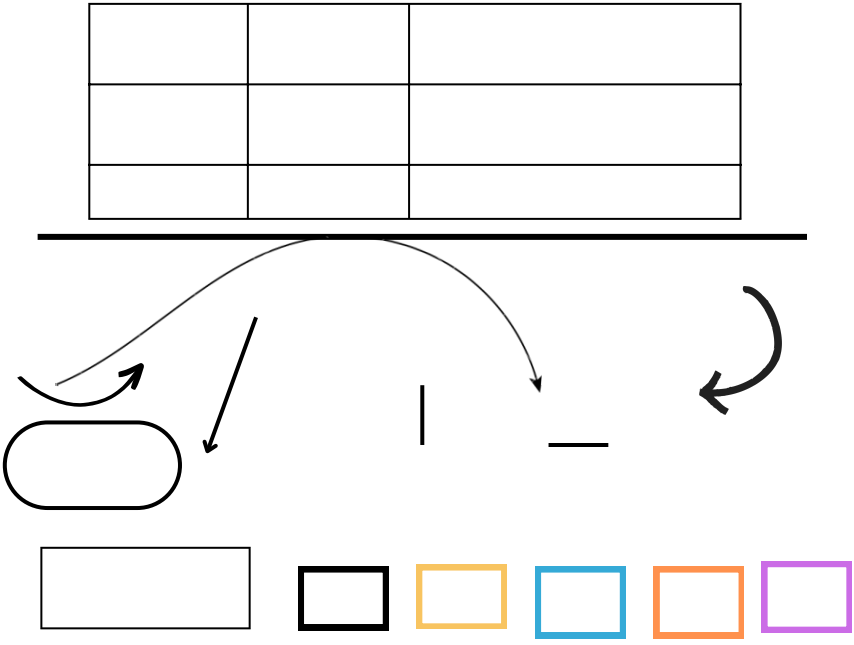
The space complexity of the program is O(n), where n is the number of nodes in the linked list. This is because we are creating n nodes, each with a fixed amount of space required for storing the node information and a pointer to the next node.
Therefore, the overall auxiliary space complexity of the program is O(n).

- **Additional Types:**

Multiply Linked List:
Multiply Linked List is a data structure in which each node of the list contains multiple pointers. It is a type of linked list which has multiple linked lists in one list. Each node has multiple pointers which can point to different nodes in the list and can also point to nodes outside the list. The data stored in a Multiply Linked List can be easily accessed and modified, making it a very efficient data structure. The nodes in a Multiply Linked List can be accessed in any order, making it suitable for applications such as graphs, trees, and cyclic lists.

Operator Precedence and Associativity:

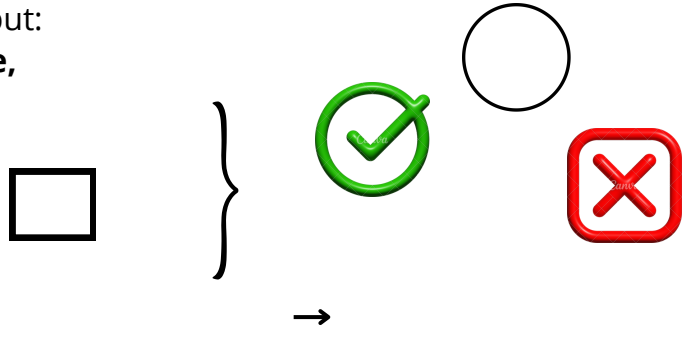
- **Make a program that will show String palindrome.**



Output:

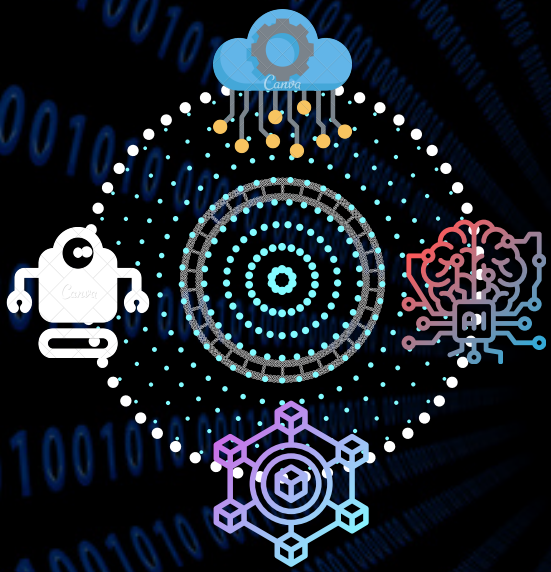
Here,

Jim



- **What is the output of the following Java program fragment:**

Operator



JAVA
(FOURTH PART)

T.I.M. HAMIM

ABC
PROKASHONI