Contents

Input Buffer

- **String Input :**

- While using scanf to input a string > scanf("%s",string);
  No "&" sign will be used.
- The function fflush(stdin) is used to flush the output buffer of the stream.

  fflush(stdin);

  getchar();

- Variable-length arrays were introduced in the C99 standard but are not a part of the C++ standard. Although some compilers might support it as an extension, it's generally not recommended to use them.

- In C++, input buffers are used to temporarily store data read from input streams, such as std::cin, before being processed. Understanding input buffers is important for handling input correctly and avoiding unexpected behavior. Here are the different kinds of input buffers in C++:

1. **Line Buffering :**

- The default buffering mode for std::cin.
- Reads input line by line, delimited by the newline character ('\n').
- Line buffering occurs when std::cin encounters a newline character or when the buffer is full.
- Can be bypassed using std::cin.ignore() or std::cin.getline().

2. **Stream Buffer:**

- The underlying buffer associated with input/output streams.
- Controlled by the standard library's streambuf class.
- Generally, you don't directly interact with the stream buffer unless you're implementing custom stream behavior.

3. **Input Buffer:**

- A buffer used to hold characters temporarily while reading input.

- Managed by the input stream objects, such as std::cin.
- Can cause issues like incomplete or unexpected input if not handled properly.

➢ **Common techniques to handle input buffer-related issues:**

- std::cin.ignore(): Ignores a specific number of characters or until a specific delimiter is encountered. Useful to skip unwanted characters or consume newline characters.
- std::cin.get(): Reads a single character from the input stream.
- std::cin.getline(): Reads an entire line of input into a character array or std::string. It discards the newline character.
- std::cin >> variable: Reads input until whitespace is encountered, but does not consume the newline character. Consider using std::cin.ignore() to consume the newline if necessary.
- std::cin.clear(): Clears any error flags on the input stream.
- std::cin.sync(): Flushes the input buffer, discarding any unread characters.

Permutation

Swap Function: Create a function to swap two elements in an array. This function will be used to generate permutations by swapping elements at different positions.

Code :

```
void swap(int &a, int &b) {
   int temp = a;
   a = b;
   b = temp;
}
```

Generate Permutations Function: Create a recursive function that generates permutations by swapping elements and recursively processing the remaining part of the array.

Code :

```
void generate_permutations(int arr[], int start, int end) {
   if (start == end) {
      // Print the permutation
      for (int i = 0; i <= end; ++i) {
         cout << arr[i] << " ";
      }
      cout << endl;
   } else {
      for (int i = start; i <= end; ++i) {
         bool should_swap = true;
         // Check if arr[i] is already placed before start
         for (int j = start; j < i; ++j) {
            if (arr[j] == arr[i]) {
```

```
                should_swap = false;
                break;
            }
        }
        if (should_swap) {
            swap(arr[start], arr[i]);  // Swap elements
            generate_permutations(arr, start + 1, end);  // Recurse on the next index
            swap(arr[start], arr[i]);  // Swap back to backtrack
        }
    }
  }
}
```

Main Function: In the main function, create an array of integers and call the generate_permutations function to start generating permutations.

Code :

```
#include <iostream>
using namespace std;
int main() {
  int input_array[] = {1, 2, 3};  // Change this array as needed
  int n = sizeof(input_array) / sizeof(input_array[0]);

  generate_permutations(input_array, 0, n - 1);

  return 0;
      }
```

Main Code :

```
#include <iostream>
using namespace std;
// Swap two elements in an array
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

// Recursive function to generate permutations
void generate_permutations(int arr[], int start, int end) {
    // Base case: When only one element is left to be considered
    if (start == end) {
        // Print the permutation
        for (int i = 0; i <= end; ++i) {
            std::cout << arr[i] << " ";
        }
        std::cout << std::endl;
    } else {
        // Loop through elements to swap and generate permutations
        for (int i = start; i <= end; ++i) {
            bool should_swap = true;

            // Check if arr[i] is already placed before start
```

```
            for (int j = start; j < i; ++j) {
                if (arr[j] == arr[i]) {
                    should_swap = false;
                    break;
                }
            }

            // If not a duplicate, proceed with swapping
            if (should_swap) {
                swap(arr[start], arr[i]);       // Swap elements
                generate_permutations(arr, start + 1, end);  // Recurse on the next index
                swap(arr[start], arr[i]);       // Swap back to backtrack
            }
        }
    }
}

int main() {
    int input_array[] = {1, 2, 2};   // Replace with your input array
    int n = sizeof(input_array) / sizeof(input_array[0]);

    generate_permutations(input_array, 0, n - 1);

    return 0;
}
```

Number of Operations on Array

---------- tbc….

STL

- **set <char> a;**
  - This line declares a variable named a, which is of type "set" and holds elements of type char.
  - Stores unique elements in a sorted order.
  - #include<set>
- **a.insert(s[i]);**
  This line inserts a character s[i] into the set a
- **Sort;**
  *String Class:*sort(str.begin(), str.end());

  Array : sort(strArray, strArray + size);
- **Permutaion**
  First sort the array - sort(arr, arr + n);

  Then - next_permutation(arr, arr + n));

  Ex :

```
#include <iostream>
#include <algorithm
using namespace std
int main() {
   int n;
   cin >> n;
   int arr[n];
   for (int i = 0; i < n; ++i) {
      cin >> arr[i];
   }
```

```
    sort(arr, arr + n); // Ensure the array is sorted for permutations
    do {
        for (int i = 0; i < n; ++i) {
            cout << arr[i] << " ";
        }
        cout << endl;
    } while (next_permutation(arr, arr + n));

    return 0;
}

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

int main() {
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; ++i) {
        cin >> arr[i];
    }
    sort(arr, arr + n); // Ensure the array is sorted for permutations

    int num_permutations = 0;
```

```
    // Generate and count permutations using next_permutation
    do {
        num_permutations++;
    } while (next_permutation(arr, arr + n));

    cout << "Number of permutations: " << num_permutations << endl;

    return 0;
}
```

- **Gcd:**

__gcd(a,b);

The maximum GCD occurs when the two numbers have the maximum difference

int maxGCD = n / 2;

- **Vector :**

In programming, a vector typically refers to a dynamic array or a resizable array. In different programming languages, vectors might be called arrays, lists, or dynamic arrays. I'll provide a general overview of vectors in the context of C++.

In C++, the **std::vector** is a part of the Standard Template Library (STL) and provides a dynamic array that automatically adjusts its size. Here are some key aspects of **std::vector**:

Declaration and Initialization:

```
#include <vector>

std::vector<int> myVector; // Declaration of an empty integer vector
std::vector<int> anotherVector = {1, 2, 3}; // Initialization with values
```

Adding Elements:

```
myVector.push_back(42); // Add an element to the end
```

Accessing Elements:

```
int value = myVector[0]; // Access element by index

int size = myVector.size(); // Get the size of the vector
```

Iterating Over Elements:

```
for (int i = 0; i < myVector.size(); ++i)

{

 std::cout << myVector[i] << " ";

}

 // Using C++11 range-based for loop

for (const auto& element : myVector) { std::cout << element << " "; }
```

Removing Elements:

```
myVector.pop_back(); // Remove the last element
```

Clearing the Vector:

```
myVector.clear(); // Remove all elements
```

Other Operations:

```
std::vector<int>

copyVector(anotherVector); // Copy constructor

std::vector<int> thirdVector = anotherVector; // Copy assignment
myVector.insert(myVector.begin() + 1, 99); // Insert 99 at index 1
myVector.erase(myVector.begin() + 2); // Erase element at index 2
```

Memory Management:

Vectors manage their own memory, automatically resizing as needed. The **capacity()** function provides the current allocated capacity, which can be greater than the size.

int capacity = myVector.capacity();

Performance:

Vectors provide O(1) time complexity for random access but may incur O(n) time for insertion or removal in the middle since elements need to be shifted.

Use Cases:

Vectors are useful when the number of elements is unknown in advance, or when frequent insertion or removal is not a primary concern.

Vectors are just one type of dynamic array, and different programming languages may use different terms for similar concepts. Understanding vectors is essential for efficient and flexible array management in many programming scenarios.

Math

- **Typecasting :**
  int number = static_cast<int>(max);

  To convert a string that represents an integer into an actual integer in C++, you can use the **std::stoi** function. Here's an example of how to do it:

```
#include <iostream>
#include <string>
 int main() {
std::string s = "170";
int number = std::stoi(s);
std::cout << "The integer value is: " << number << std::endl;
```

```
return 0;

}
```

In this code, **std::stoi** is used to convert the string **s** into an integer **number**. After the conversion, you can use the **number** variable as an integer in your program.

- In a rectangular table ,
  Number of cells = row * column
  Number of edges = 2 * (r * c) + r + c