

Striver A2Z

DSA SHEET

Count digits in a number

Brute Force Approach:-

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int countDigits(int n){
    int cnt = 0;
    while(n > 0){
        cnt = cnt + 1;
        n = n / 10;
    }
    // Return the
    // count of digits.
    return cnt;
}
int main() {
    int N = 329823;
    cout << "N: " << N << endl;
    int digits = countDigits(N);
    cout << "Number of Digits in N: " << digits << endl;
    return 0;
}
```

Time Complexity: $O(\log_{10} N + 1)$, where N is the input number. The time complexity is determined by the number of digits in the input integer N. In the worst case when N is a multiple of 10 the number of digits in N is $\log_{10} N + 1$.

1. In the while loop we divide N by 10 until it becomes 0 which takes $\log_{10} N$ iterations.
2. In each iteration of the while loop we perform constant time operations like division and increment the counter.

Space Complexity : $O(1)$, as only a constant amount of additional memory for the counter regardless of size of the input number.

Optimal Approach:-

```
#include <iostream>
#include <cmath>
#include <algorithm>
using namespace std;
int countDigits(int n)
{
    int cnt = (int)(log10(n)+1);
    return cnt;
}
int main() {
    int N = 329823;
    cout << "N: " << N << endl;
    int digits = countDigits(N);
    cout << "Number of Digits in N: " << digits << endl;
    return 0;
}
```

Time Complexity: $O(1)$, as simple arithmetic operations in constant time are computed on integers.

Space Complexity : $O(1)$, as only a constant amount of additional memory for the count variable regardless of size of the input number.

Reverse Digits of A Number

```
#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int reverseNumber(int n) {
        int revNum = 0;
        while (n > 0) {
            int lastDigit = n % 10;
            revNum = revNum * 10 + lastDigit;
            n = n / 10;
        }
    }
};
```

```

    }
    return revNum;
}
};

int main() {
    Solution obj;
    int num = 12345;
    cout << obj.reverseNumber(num) << endl;
    return 0;
}

```

Time Complexity: $O(\log_{10}N)$, The number of iterations in the loop depends on the number of digits in the number N. Since each digit is processed once using constant time operations (modulo, division, multiplication, and addition), the total time taken is proportional to the number of digits, which is $\log_{10}N$.

Space Complexity: $O(1)$, Only a constant number of variables are used regardless of the size of the input number.

Check if a number is Palindrome or Not

```

#include <iostream>

using namespace std;

bool palindrome(int n) {
    int revNum = 0;
    int dup = n;
    while (n > 0) {
        int ld = n % 10;
        revNum = (revNum * 10) + ld
        n = n / 10;
    }
    if (dup == revNum) {
        return true;
    } else {
        return false;
    }
}

int main() {

```

```

    int number = 4554;
    if (palindrome(number)) {
        cout << number << " is a palindrome." <<
endl;
    } else {
        cout << number << " is not a palindrome." <<
endl;
    }
    return 0;
}

```

Time Complexity: $O(\log_{10}N + 1)$, as in the worst case when N is a multiple of 10 the number of digits in N is $\log_{10} N + 1$. In the while loop we divide N by 10 until it becomes 0 which takes $\log_{10}N$ iterations. In each iteration of the while loop we perform constant time operations like modulus and division and pushing elements into the vector.

Space Complexity: $O(1)$, as only a constant amount of additional memory for the reversed number regardless of size of the input number.

Find GCD of two numbers

Brute Force Approach

```

#include <iostream>

#include <algorithm>

using namespace std;

int findGcd(int n1, int n2) {
    int gcd = 1;
    for(int i = 1; i <= min(n1, n2); i++) {
        if(n1 % i == 0 && n2 % i == 0) {
            gcd = i;
        }
    }
    return gcd;
}

int main() {
    int n1 = 20, n2 = 15;
    int gcd = findGcd(n1, n2);
    cout << "GCD of " << n1 << " and " << n2 << "
is: " << gcd << endl;

```

```

    return 0;
}

```

Time Complexity: $O(\min(N_1, N_2))$ where N_1 and N_2 is the input number. The algorithm iterates from 1 to the minimum of N_1 and N_2 and each iteration checks whether both the numbers are divisible by the current number (constant time operations).

Space Complexity: $O(1)$ as the space complexity remains constant and independent of the input size. Only a fixed amount of memory is required to store the integer variables.

Better Approach

```

#include <iostream>

#include <algorithm>

using namespace std;

int findGcd(int n1, int n2) {
    for(int i = min(n1, n2); i > 0; i--) {
        if(n1 % i == 0 && n2 % i == 0) {
            return i;
        }
    }
    return 1;
}

int main() {
    int n1 = 20, n2 = 15;

    int gcd = findGcd(n1, n2);

    cout << "GCD of " << n1 << " and " << n2 << "
    is: " << gcd << endl;

    return 0;
}

```

Time Complexity: $O(\min(N_1, N_2))$ where N_1 and N_2 is the input number. The algorithm iterates from the minimum of N_1 and N_2 to 1 and each iteration checks whether both the numbers are divisible by the current number (constant time operations).

Space Complexity: $O(1)$ as the space complexity remains constant and independent of the input size. Only a fixed amount of memory is required to store the integer variables.

Optimal Approach

```

#include <iostream>

#include <algorithm>

using namespace std;

```

```

int findGcd(int a, int b) {
    while(a > 0 && b > 0) {
        if(a > b) {
            a = a % b;
        }
        else {
            b = b % a;
        }
    }
    if(a == 0) {
        return b;
    }
    return a;
}

int main() {
    int n1 = 20, n2 = 15;

    int gcd = findGcd(n1, n2);

    cout << "GCD of " << n1 << " and " << n2 << "
    is: " << gcd << endl;

    return 0;
}

```

Time Complexity: $O(\min(N_1, N_2))$ where N_1 and N_2 is the input number. The algorithm iterates from the minimum of N_1 and N_2 to 1 and each iteration checks whether both the numbers are divisible by the current number (constant time operations).

Space Complexity: $O(1)$ as the space complexity remains constant and independent of the input size. Only a fixed amount of memory is required to store the integer variable

Check if a number is Armstrong Number or not

```

#include <bits/stdc++.h>

using namespace std;

class ArmstrongChecker {
public:
    static bool isArmstrong(int num) {
        int k = to_string(num).length();
        int sum = 0;

```

```

int n = num;
while (n > 0) {
    int ld = n % 10;
    sum += pow(ld, k);
    n /= 10;
}
return sum == num;
}
};

int main() {
    int number = 153;
    if (ArmstrongChecker::isArmstrong(number)) {
        cout << number << " is an Armstrong
number." << endl;
    } else {
        cout << number << " is not an Armstrong
number." << endl;
    }
    return 0;
}

```

Time Complexity: $O(\log_{10} N + 1)$ where N is the input number. The time complexity is determined by the number of digits in the input integer N . In the worst case when N is a multiple of 10 the number of digits in N is $\log_{10} N + 1$.

Space Complexity: $O(1)$ as only a constant amount of additional memory for the reversed number regardless of size of the input number.

Print all Divisors of a given Number

Brute Force Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    vector<int> getDivisors(int N) {
        vector<int> res;
        for (int i = 1; i <= N; i++) {
            if (N % i == 0) {

```

```

                res.push_back(i);
            }
        }
        return res;
    }
};

int main() {
    Solution sol;
    int N = 36;
    vector<int> result = sol.getDivisors(N);
    cout << "Divisors of " << N << ": ";
    for (int val : result) {
        cout << val << " ";
    }
    cout << endl;
    return 0;
}

```

Time Complexity: $O(N)$, we check for every number from 1 to N .

Space Complexity: $O(N)$, extra space used for storing divisors.

Optimal Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    vector<int> getDivisors(int N) {
        vector<int> res;
        for (int i = 1; i * i <= N; i++) {
            if (N % i == 0) {
                res.push_back(i);
                if (i != N / i) {
                    res.push_back(N / i);
                }
            }
        }
        return res;
    }
}

```

```

    }
};

int main() {
    Solution sol;

    int N = 36;
    vector<int> result = sol.getDivisors(N);

    cout << "Divisors of " << N << ": ";

    for (int val : result) {
        cout << val << " ";
    }

    cout << endl;

    return 0;
}

```

Time Complexity: $O(\sqrt{N})$, we check for every number between 1 and square root of N.

Space Complexity: $O(2*\sqrt{N})$, extra space used for storing divisors.

Check if a number is prime or not

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

bool checkPrime(int n) {
    int cnt = 0;

    for (int i = 1; i <= n; i++) {
        if (n % i == 0) {
            cnt++;
        }
    }

    if (cnt == 2) {
        return true;
    }
    else {
        return false;
    }
}

int main() {
    int n = 1483;

```

```

    bool isPrime = checkPrime(n);

    if (isPrime) {
        cout << n << " is a prime number." << endl;
    }
    else {
        cout << n << " is not a prime number." << endl;
    }

    return 0;
}

```

Time Complexity: $O(N)$, as we iterate from 1 to N performing constant-time operation for each iteration.

Space Complexity : $O(1)$, as the space used by the algorithm does not increase with the size of the input.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

bool checkPrime(int n) {
    int cnt = 0;

    for (int i = 1; i <= sqrt(n); i++) {
        if (n % i == 0) {
            cnt++;

            if (n / i != i) {
                cnt++;
            }
        }
    }

    if (cnt == 2) {
        return true;
    }
    else {
        return false;
    }
}

int main() {
    int n = 1483;

    bool isPrime = checkPrime(n);

    if (isPrime) {
        cout << n << " is a prime number." << endl;
    }
}

```

```

    } else {
        cout << n << " is not a prime number." <<
endl;
    }
    return 0;
}

```

Time Complexity: $O(\sqrt{n})$, as The loop iterates up to the square root of n performing constant time operations at each step.

Space Complexity : $O(1)$, as the space complexity remains constant and independent of the input size. Only a fixed amount of memory is required to store the integer variables.

Print Name N times using Recursion

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void printName(string name, int count, int N) {
        if (count == N)
            return;
        cout << name << "\n";
        printName(name, count + 1, N);
    }
};

int main() {
    Solution sol;
    int N = 5;
    string name = "Ashish";
    sol.printName(name, 0, N);
    return 0;
}

```

Time Complexity: $O(N)$, we print our name exactly N times.

Space Complexity: $O(N)$, stack space used for recursive calls.

Print 1 to N using Recursion

Forward Recursion

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void printNumbers(int current, int n) {
        if (current > n)
            return;
        cout << current << " ";
        printNumbers(current + 1, n);
    }
};

int main() {
    Solution sol;
    int n = 10;
    sol.printNumbers(1, n);
    cout << "\n";
    return 0;
}

```

Time Complexity: $O(N)$, we print every number from 1 to N using recursion

Space Complexity: $O(N)$, stack space used for recursive calls.

Backtracking

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void printNumbers(int current, int n) {
        if (current > n)
            return;
        printNumbers(current + 1, n);
        cout << current << " ";
    }
};

int main() {
    Solution sol;
    int n = 10

```

```

    sol.printNumbers(1, n);

    cout << "\n";

    return 0;
}

```

Time Complexity: $O(N)$, we print every number from 1 to N using recursion

Space Complexity: $O(N)$, stack space used for recursive calls

Print N to 1 using Recursion.

Forward Recursion

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void printNumbers(int current) {
        if (current < 1)
            return;

        cout << current << " ";
        printNumbers(current - 1);
    }
};

int main() {
    Solution sol;

    int n = 10;

    sol.printNumbers(n);

    cout << "\n";

    return 0;
}

```

Time Complexity: $O(N)$, we print every number from N to 1 using recursion

Space Complexity: $O(N)$, stack space used for recursive calls.

Backtracking

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:

```

```

    void printNumbers(int current) {
        if (current < 1)
            return;

        printNumbers(current - 1);

        cout << current << " ";
    }
};

int main() {
    Solution sol;

    int n = 10;

    sol.printNumbers(n);

    cout << "\n";

    return 0;
}

Time Complexity:  $O(N)$ , we print every number from N to 1 using recursion
Space Complexity:  $O(N)$ , stack space used for recursive calls.

```

Sum of first N Natural Numbers

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int sumOfNaturalNumbers(int N) {
        int sum = 0;

        for (int i = 1; i <= N; i++) {
            sum += i;
        }

        return sum;
    }
};

int main() {
    Solution obj;

    int N;

    cin >> N;

    int result = obj.sumOfNaturalNumbers(N);

    cout << result << endl;
}

```

```

    return 0;
}

```

Time Complexity: $O(N)$, We iterate from 1 to N once, performing a constant-time addition operation in each iteration, resulting in linear time complexity.

Space Complexity: $O(1)$, We only use a few variables to store the sum and loop counter, so the space usage remains constant regardless of N.

Using Formula

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int sumOfNaturalNumbers(int N) {
        return (N * (N + 1)) / 2;
    }
};

int main() {
    Solution obj;

    int N;

    cin >> N;

    cout << obj.sumOfNaturalNumbers(N) << endl;

    return 0;
}

```

Time Complexity: $O(1)$

Space Complexity: $O(1)$

Recursive Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int sumOfNaturalNumbers(int N) {
        if (N == 1) {
            return 1;
        }

        return N + sumOfNaturalNumbers(N - 1);
    }
};

```

```

int main() {
    Solution obj;

    int N;

    cin >> N;

    cout << obj.sumOfNaturalNumbers(N) << endl;

    return 0;
}

```

Time Complexity: $O(N)$, The function is called N times, with each call performing $O(1)$ work.

Space Complexity: $O(N)$, Due to recursive function calls being stored on the call stack, which grows linearly with N.

Factorial of a Number : Iterative and Recursive

Iterative Solution

```

#include <iostream>

using namespace std;

int factorial(int X) {
    int ans = 1;

    for (int i = 1; i <= X; i++) {
        ans = ans * i;
    }

    return ans;
}

int main() {
    int X = 5;

    int result = factorial(X);

    cout << "The factorial of " << X << " is " <<
    result;

    return 0;
}

```

Time Complexity: $O(n)$

Space Complexity: $O(1)$

Recursive Solution

```

#include <bits/stdc++.h>

```

```
using namespace std;

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

int main() {
    int n = 3;

    cout << factorial(n) << endl;

    return 0;
}
```

Time Complexity: $O(N)$, Since the function is being called n times, and for each function, we have only one printable line that takes $O(1)$ time, so the cumulative time complexity would be $O(N)$

Space Complexity: $O(N)$, In the worst case, the recursion stack space would be full with all the function calls waiting to get completed and that would make it an $O(N)$ recursion stack space.

Reverse a given Array

Brute Force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> reverseArray(vector<int>& arr) {
        int n = arr.size();
        vector<int> ans(n);
        for (int i = 0; i < n; i++) {
            ans[i] = arr[n - 1 - i];
        }
        return ans;
    }
};

int main() {
    // Input array
    vector<int> arr = {1, 2, 3, 4, 5};
```

```
Solution obj;

vector<int> result = obj.reverseArray(arr);

cout << "Reversed Array: ";

for (int num : result) {
    cout << num << " ";
}

cout << endl;

return 0;
}
```

Time Complexity: $O(n)$ Each element is visited once in a loop, where n is the number of elements in the input array.

Space Complexity: $O(n)$ An additional array of the same size is used to store the reversed elements.

Better Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void reverseArray(vector<int>& arr) {
        int p1 = 0;
        int p2 = arr.size() - 1;
        while (p1 < p2) {
            swap(arr[p1], arr[p2]);
            p1++;
            p2--;
        }
    }
};

int main() {
    Solution sol;
    vector<int> arr = {1, 2, 3, 4, 5};
    sol.reverseArray(arr);
    for (int num : arr) {
        cout << num << " ";
    }
    return 0;
}
```

```
}
```

Time Complexity: $O(n)$ Where n is the number of elements in the array. Each element is visited at most once due to the two-pointer approach.

Space Complexity: $O(1)$ No extra space is used other than a few pointers and variables. The array is reversed in-place.

Built-in Library Function Approach

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Solution {
```

```
public:
```

```
    void reverseArray(vector<int>& arr) {
```

```
        reverse(arr.begin(), arr.end());
```

```
    }
```

```
};
```

```
int main() {
```

```
    vector<int> arr = {1, 2, 3, 4, 5};
```

```
    Solution obj;
```

```
    obj.reverseArray(arr);
```

```
    for (int num : arr) {
```

```
        cout << num << " ";
```

```
    }
```

```
    cout << endl;
```

```
    return 0;
```

```
}
```

Time Complexity: $O(n)$, because each element is visited once and possibly swapped once with its mirror index.

Space Complexity: $O(1)$ for C++, Java, and JavaScript (in-place), but $O(n)$ for Python slicing since it creates a new list and then assigns back (unless using two pointers).

Check if the given String is Palindrome or not

Brute Force Approach

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
bool isPalindrome(string s) {
```

```
    int left = 0, right = s.length() - 1;
```

```
    while (left < right) {
```

```
        if (!isalnum(s[left]))
```

```
            left++;
```

```
        else if (!isalnum(s[right]))
```

```
            right--;
```

```
        else if (tolower(s[left]) != tolower(s[right]))
```

```
            return false;
```

```
        else {
```

```
            left++;
```

```
            right--;
```

```
        }
```

```
    }
```

```
    return true;
```

```
}
```

```
int main() {
```

```
    string str = "ABCDCEBA";
```

```
    bool ans = isPalindrome(str);
```

```
    if (ans == true) {
```

```
        cout << "Palindrome";
```

```
    } else {
```

```
        cout << "Not Palindrome";
```

```
    }
```

```
    return 0;
```

```
}
```

Time Complexity: $O(N)$, where N is the length of the string. Each character is compared at most once till the middle of the string.

Space Complexity: $O(1)$, since no extra space is used apart from a few variables for iteration.

Optimal Approach

```
#include<bits/stdc++.h>
```

```
using namespace std;
```

```
bool palindrome(int i, string& s) {
```

```
    if (i >= s.length() / 2) return true;
```

```
    if (s[i] != s[s.length() - i - 1]) return false;
```

```

        return palindrome(i + 1, s);
    }

int main() {
    string s = "madam";

    cout << palindrome(0, s);

    cout << endl;

    return 0;
}

```

Time Complexity: $O(N)$, where N is the length of the string. Each character is checked once.

Space Complexity: $O(N)$, due to the recursion stack in the worst case (no early termination).

Print Fibonacci Series up to Nth term

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

int main() {
    int n = 5;

    if (n == 0) {
        cout << 0;
    }
    else if (n == 1) {
        cout << 0 << " " << 1 << "\n";
    }
    else {
        int fib[n + 1];
        fib[0] = 0;
        fib[1] = 1;
        for (int i = 2; i <= n; i++) {
            fib[i] = fib[i - 1] + fib[i - 2];
        }

        cout << "The Fibonacci Series up to " << n <<
        "th term:" << endl;

        for (int i = 0; i <= n; i++) {

```

```

            cout << fib[i] << " ";

        }

    }

    return 0;
}

```

Time Complexity: $O(n)+O(n)$, for calculating and printing the Fibonacci series.

Space Complexity: $O(n)$ for storing the fibonacci series.

Better Approach

```

#include <bits/stdc++.h>

using namespace std;

int main() {
    int n = 5;

    if (n == 0) {
        cout << "The Fibonacci Series up to " << n <<
        "th term:" << endl;

        cout << 0;
    }
    else {
        int secondLast = 0;
        int last = 1;

        cout << "The Fibonacci Series up to " << n <<
        "th term:" << endl;

        cout << secondLast << " " << last << " ";

        int cur; // ith term
        for (int i = 2; i <= n; i++) {
            cur = last + secondLast;
            secondLast = last;
            last = cur;
            cout << cur << " ";
        }

        return 0;
    }
}

```

Time Complexity: $O(N)$. As we are iterating over just one for a loop.

Space Complexity: $O(1)$, no extra space used.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

int fibonacci(int N) {
    if (N <= 1) {
        return N;
    }

    int last = fibonacci(N - 1);
    int slast = fibonacci(N - 2);

    return last + slast;
}

int main() {
    int N = 4;

    cout << fibonacci(N) << endl;

    return 0;
}
```

Time Complexity: $O(2^N)$ { This problem involves two function calls for each iteration which further expands to 4 function calls and so on which makes worst-case time complexity to be exponential in nature }.

Space Complexity: $O(N)$ { At maximum there could be N function calls waiting in the recursion stack since we need to calculate the Nth Fibonacci number for which we also need to calculate (N-1) Fibonacci numbers before it }.

Count frequency of each element in the array

Brute Force Approach

```
#include <bits/stdc++.h>

using namespace std;

void countFreq(int arr[], int n) {
    vector<bool> visited(n, false);

    for (int i = 0; i < n; i++) {
        if (visited[i] == true)
            continue;

        int count = 1;

        for (int j = i + 1; j < n; j++) {
            if (arr[i] == arr[j]) {
                visited[j] = true;
```

```
                count++;
            }
        }

        cout << arr[i] << " " << count << endl;
    }
}

int main() {
    int arr[] = {10, 5, 10, 15, 10, 5};

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

    countFreq(arr, n);

    return 0;
}
```

Time Complexity: $O(N^2)$, as for every element we may scan the remaining elements in the array.

Space Complexity: $O(N)$, for the visited array of size N.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

void Frequency(int arr[], int n) {
    unordered_map<int, int> map;

    for (int i = 0; i < n; i++) {
        map[arr[i]]++;
    }

    for (auto x : map) {
        cout << x.first << " " << x.second << endl;
    }
}

int main() {
    int arr[] = {10, 5, 10, 15, 10, 5};

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

    Frequency(arr, n);

    return 0;
}
```

Time Complexity: $O(N)$, where N is the number of elements in the array. Each element is processed once.

Space Complexity: $O(N)$, for storing frequencies of unique elements in the unordered_map.

Find the highest/lowest frequency element

Brute Force Approach

```
#include <bits/stdc++.h>

using namespace std;

class FrequencyCounter {
public:
    void countFreq(int arr[], int n)
    {
        vector<bool> visited(n, false);

        int maxFreq = 0, minFreq = n;
        int maxEle = 0, minEle = 0;

        for (int i = 0; i < n; i++) {
            if (visited[i] == true)
                continue;

            int count = 1;
            for (int j = i + 1; j < n; j++) {
                if (arr[i] == arr[j]) {
                    visited[j] = true;
                    count++;
                }
            }

            if (count > maxFreq) {
                maxEle = arr[i];
                maxFreq = count;
            }

            if (count < minFreq) {
                minEle = arr[i];
                minFreq = count;
            }
        }

        cout << "The highest frequency element is: "
        << maxEle << "\n";
```

```
        cout << "The lowest frequency element is: "
        << minEle << "\n";
    }
};

int main()
{
    FrequencyCounter fc;

    int arr[] = {10, 5, 10, 15, 10, 5};

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

    fc.countFreq(arr, n);

    return 0;
}
```

Time Complexity: $O(N*N)$, where N = size of the array. We are using the nested loop to find the frequency.

Space Complexity: $O(N)$, where N = size of the array. It is for the visited array we are using.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class FrequencyCounter {
public:
    void Frequency(int arr[], int n)
    {
        unordered_map<int, int> map;

        for (int i = 0; i < n; i++)
            map[arr[i]]++;

        int maxFreq = 0, minFreq = n;
        int maxEle = 0, minEle = 0;

        for (auto it : map) {
            int element = it.first;
            int count = it.second;

            if (count > maxFreq) {
                maxFreq = count;
                maxEle = element;
            }

            if (count < minFreq) {
                minFreq = count;
```

```

        minEle = element;
    }
}

cout << "The highest frequency element is: "
<< maxEle << "\n";

cout << "The lowest frequency element is: "
<< minEle << "\n";

}

};

int main()
{
    FrequencyCounter fc;

    int arr[] = {10, 5, 10, 15, 10, 5};

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

    fc.Frequency(arr, n);

    return 0;
}

```

Time Complexity: $O(N)$, where N = size of the array. The insertion and retrieval operation in the map takes $O(1)$ time.

Space Complexity: $O(N)$, where N = size of the array. It is for the map we are using.

Selection Sort Algorithm

```

#include <bits/stdc++.h>

using namespace std;

void selection_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int mini = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[mini]) {
                mini = j;
            }
        }
        int temp = arr[mini];
        arr[mini] = arr[i];
        arr[i] = temp;
    }
}

```

```

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

int main() {
    int arr[] = {13, 46, 24, 52, 20, 9};
    int n = sizeof(arr) / sizeof(arr[0]);

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

    selection_sort(arr, n);

    return 0;
}

```

Time Complexity: $O(N^2)$, Selection sort runs in $O(N^2)$ time in the best, average, and worst cases due to its nested loop structure. It makes approximately $N(N-1)/2$ comparisons, regardless of the array's initial state. Even if no swaps are needed (best case), the number of comparisons remains the same.

Space Complexity: $O(1)$. No extra space used

Bubble Sort Algorithm

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

class BubbleSort {
public:
    void bubble_sort(vector<int>& arr) {
        int n = arr.size();
        for (int i = n - 1; i >= 0; i--) {
            for (int j = 0; j <= i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    swap(arr[j], arr[j + 1]);
                }
            }
        }
    }
}

```

```

    }
}

cout << "After Using Bubble Sort:\n";
for (int num : arr) {
    cout << num << " ";
}

cout << endl;
}

};

int main() {
    vector<int> arr = {13, 46, 24, 52, 20, 9};
    cout << "Before Using Bubble Sort:\n";
    for (int num : arr) {
        cout << num << " ";
    }

    cout << endl;

    BubbleSort sorter;
    sorter.bubble_sort(arr);

    return 0;
}

```

Time Complexity: $O(N^2)$, (where N = size of the array), for the worst, and average cases.

Space Complexity: $O(1)$.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class BubbleSort {
public:
    void bubble_sort(vector<int>& arr) {
        int n = arr.size();
        for (int i = n - 1; i >= 0; i--) {
            int didSwap = 0;
            for (int j = 0; j <= i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    swap(arr[j], arr[j + 1]);
                    didSwap = 1;
                }
            }
            if (didSwap == 0) {
                break;
            }
        }

        cout << "After Using Bubble Sort:\n";
        for (int num : arr) {
            cout << num << " ";
        }

        cout << endl;
    }
};

int main() {
    vector<int> arr = {13, 46, 24, 52, 20, 9};
    cout << "Before Using Bubble Sort:\n";
    for (int num : arr) {
        cout << num << " ";
    }

    cout << endl;

    BubbleSort sorter;
    sorter.bubble_sort(arr);

    return 0;
}

```

```

    }
}

cout << "After Using Bubble Sort:\n";
for (int num : arr) {
    cout << num << " ";
}

cout << endl;
}

};

int main() {
    vector<int> arr = {13, 46, 24, 52, 20, 9};
    cout << "Before Using Bubble Sort:\n";
    for (int num : arr) {
        cout << num << " ";
    }

    cout << endl;

    BubbleSort sorter;
    sorter.bubble_sort(arr);

    return 0;
}

```

Time Complexity: $O(N^2)$ for the worst and average cases and $O(N)$ for the best case. Here, N = size of the array.

Space Complexity: $O(1)$

Insertion Sort Algorithm

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> insertionSort(vector<int>& nums) {
        int n = nums.size(); // Size of the array
    }
}

```

```

    for (int i = 1; i < n; i++) {
        int key = nums[i];
        int j = i - 1;
        while (j >= 0 && nums[j] > key) {
            nums[j + 1] = nums[j];
            j--;
        }
        nums[j + 1] = key; // Insert key at correct
        position
    }
    return nums;
}

int main() {
    Solution solution;
    vector<int> nums = {13, 46, 24, 52, 20, 9};
    cout << "Before Using Insertion Sort: " << endl;
    for (int num : nums) {
        cout << num << " ";
    }
    cout << endl;
    nums = solution.insertionSort(nums);

    cout << "After Using Insertion Sort: " << endl;
    for (int num : nums) {
        cout << num << " ";
    }
    cout << endl;
    return 0;
}

```

Time Complexity: $O(n^2)$, where n is the number of elements in the array. This is because, in the worst case, we may have to compare each element with all the previous elements.

Space Complexity: $O(1)$, as we are sorting the array in place and not using any additional data structures that grow with input size.

Merge Sort Algorithm

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void merge(vector<int>& arr, int low, int mid, int high) {
        vector<int> temp;
        int left = low, right = mid + 1;
        while (left <= mid && right <= high) {
            if (arr[left] <= arr[right])
                temp.push_back(arr[left++]);
            else
                temp.push_back(arr[right++]);
        }
        while (left <= mid)
            temp.push_back(arr[left++]);
        while (right <= high)
            temp.push_back(arr[right++]);
        for (int i = low; i <= high; i++)
            arr[i] = temp[i - low];
    }

    void mergeSort(vector<int>& arr, int low, int high) {
        if (low >= high)
            return;
        int mid = (low + high) / 2;
        mergeSort(arr, low, mid);
        mergeSort(arr, mid + 1, high);
        merge(arr, low, mid, high);
    }
};

int main() {
    vector<int> arr = {5, 2, 8, 4, 1};
    Solution sol;
    sol.mergeSort(arr, 0, arr.size() - 1);
}

```

```

    for (int x : arr)
        cout << x << " ";

    cout << endl;

    return 0;
}

```

Time Complexity: $O(N \cdot \log N)$, merging two arrays take linear time and array is recursively divided into halves ($\log N$ times).

Space Complexity: $O(N)$, we use a temporary array to store elements in sorted order.

Recursive Bubble Sort Algorithm

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

void bubble_sort(int arr[], int n) {
    if (n == 1) return;

    for (int j = 0; j <= n - 2; j++) {
        if (arr[j] > arr[j + 1]) {
            int temp = arr[j + 1];
            arr[j + 1] = arr[j];
            arr[j] = temp;
        }
    }

    bubble_sort(arr, n - 1);
}

int main() {
    int arr[] = {13, 46, 24, 52, 20, 9};

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

    cout << "Before Using Bubble Sort: " << endl;
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    cout << endl;

    bubble_sort(arr, n);

    cout << "After Using Bubble Sort: " << endl;
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}

```

```

    cout << endl;

    return 0;
}

```

Time Complexity: $O(N^2)$, (where N = size of the array), for the worst, and average cases.

Space Complexity: $O(N)$ auxiliary stack space.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

void bubble_sort(int arr[], int n) {
    if (n == 1) return;

    int didSwap = 0;

    for (int j = 0; j <= n - 2; j++) {
        if (arr[j] > arr[j + 1]) {
            int temp = arr[j + 1];
            arr[j + 1] = arr[j];
            arr[j] = temp;
            didSwap = 1;
        }
    }

    if (didSwap == 0) return;

    bubble_sort(arr, n - 1);
}

int main() {
    int arr[] = {13, 46, 24, 52, 20, 9};

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

    cout << "Before Using Bubble Sort: " << endl;
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    cout << endl;

    bubble_sort(arr, n);

    cout << "After Using Bubble Sort: " << endl;
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
}

```

```

    cout << endl;

    return 0;
}

```

Time Complexity: $O(N^2)$ for the worst and average cases and $O(N)$ for the best case. Here, N = size of the array.

Space Complexity: $O(N)$ auxiliary stack space.

Recursive Insertion Sort Algorithm

```

#include <bits/stdc++.h>

using namespace std;

void insertion_sort(int arr[], int i, int n) {
    if (i == n) return;

    int j = i;
    while (j > 0 && arr[j - 1] > arr[j]) {
        int temp = arr[j - 1];
        arr[j - 1] = arr[j];
        arr[j] = temp;
        j--;
    }

    insertion_sort(arr, i + 1, n);
}

int main() {
    int arr[] = {13, 46, 24, 52, 20, 9};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Before Using Insertion Sort: " << endl;
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    cout << endl;
    insertion_sort(arr, 0, n);

    cout << "After Using Insertion Sort: " << endl;
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    cout << endl;
}

```

```

    return 0;
}

```

Time Complexity: $O(N^2)$, (where N = size of the array), for the worst, and average cases.

Space Complexity: $O(N)$ auxiliary stack space.

Quick Sort Algorithm

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void quickSort(vector<int>& arr, int low, int high) {
        if (low < high) {
            int pivotIndex = partition(arr, low, high);
            quickSort(arr, low, pivotIndex - 1);
            quickSort(arr, pivotIndex + 1, high);
        }
    }

    int partition(vector<int>& arr, int low, int high) {
        int pivot = arr[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {
                i++;
                swap(arr[i], arr[j]);
            }
        }

        swap(arr[i + 1], arr[high]);
        return i + 1;
    }
};

int main() {
    // Sample array
    vector<int> arr = {10, 7, 8, 9, 1, 5};

    Solution sol;
}

```

```

sol.quickSort(arr, 0, arr.size() - 1);

for (int num : arr)

    cout << num << " ";

return 0;
}

```

Time Complexity: $O(N \cdot \log N)$, At each step, we divide the whole array, for that we take $\log N$ time and n steps are taken for the partitioning. In worst case i.e. when our pivot is always the greatest or the smallest element of the array, the time complexity can be $O(N^2)$.
Space Complexity: $O(N)$, auxiliary stack space.

Find the Largest element in an array

Brute Force

```

#include<bits/stdc++.h>

using namespace std;

int sortArr(vector<int>& arr) {

    sort(arr.begin(), arr.end());

    return arr[arr.size() - 1];

}

int main() {

    vector<int> arr1 = {2, 5, 1, 3, 0};

    vector<int> arr2 = {8, 10, 5, 7, 9};

    cout << "The Largest element in the array is: "
    << sortArr(arr1) << endl;

    cout << "The Largest element in the array is: "
    << sortArr(arr2);

    return 0;

}

```

Time Complexity: $O(N \log N)$, where N is the size of the array, as we are sorting the array.

Space Complexity: $O(1)$, as we are using a constant

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

int findLargestElement(int arr[], int n) {

    int max = arr[0];

    for (int i = 1; i < n; i++) {

```

```

        if (max < arr[i]) {

            max = arr[i];

        }

    }

    return max;

}

int main() {

    int arr1[] = {2, 5, 1, 3, 0};

    int n = 5;

    int max = findLargestElement(arr1, n);

    cout << "The largest element in the array is: " <<
    max << endl;

    int arr2[] = {8, 10, 5, 7, 9};

    n = 5;

    max = findLargestElement(arr2, n);

    cout << "The largest element in the array is: " <<
    max << endl;

    return 0;

}

```

Time Complexity: $O(N)$, where N is the size of the array, as we are iterating through the array once.

Space Complexity: $O(1)$, as we are using a constant

Find Second Smallest and Second Largest Element in an array

Brute Force

```

#include<bits/stdc++.h>

using namespace std;

void getElements(int arr[], int n)

{

    if(n == 0 || n == 1)

        cout << -1 << " " << -1 << endl;

    sort(arr, arr + n);

    int small = arr[1];

    int large = arr[n - 2];

    cout << "Second smallest is " << small << endl;

    cout << "Second largest is " << large << endl;

```

```

}
int main()
{
    int arr[] = {1, 2, 4, 6, 7, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    getElements(arr, n);
    return 0;
}

```

Time Complexity: $O(N \log N)$, for sorting the array.

Space Complexity: $O(1)$, as we are using a constant amount of space for variables.

Better Approach

```

#include<bits/stdc++.h>

using namespace std;

void getElements(int arr[], int n)
{
    if (n == 0 || n == 1)
        cout << -1 << " " << -1 << endl;

    int small = INT_MAX, second_small = INT_MAX;
    int large = INT_MIN, second_large = INT_MIN;
    int i;
    for (i = 0; i < n; i++) {
        small = min(small, arr[i]);
        large = max(large, arr[i]);
    }
    for (i = 0; i < n; i++) {
        if (arr[i] < second_small && arr[i] != small)
            second_small = arr[i];
        if (arr[i] > second_large && arr[i] != large)
            second_large = arr[i];
    }

    cout << "Second smallest is " << second_small << endl;

    cout << "Second largest is " << second_large << endl;
}

```

```

int main()
{
    int arr[] = {1, 2, 4, 6, 7, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    getElements(arr, n);
    return 0;
}

```

Time Complexity: $O(N)$, we do two linear traversals in our array.

Space Complexity: $O(1)$, as we are using a constant amount of space for variables.

Optimal Approach

```

#include<bits/stdc++.h>

using namespace std;

int secondSmallest(int arr[], int n) {
    if (n < 2)
        return -1;

    int small = INT_MAX;
    int second_small = INT_MAX;
    for (int i = 0; i < n; i++) {
        if (arr[i] < small) {
            second_small = small;
            small = arr[i];
        }
        else if (arr[i] < second_small && arr[i] != small) {
            second_small = arr[i];
        }
    }
    return second_small;
}

int secondLargest(int arr[], int n) {
    if (n < 2)
        return -1;

    int large = INT_MIN, second_large = INT_MIN;
    for (int i = 0; i < n; i++) {
        if (arr[i] > large) {

```

```

        second_large = large;
        large = arr[i];
    }
    else if (arr[i] > second_large && arr[i] !=
large) {
        second_large = arr[i];
    }
}
return second_large;
}

```

```

int main() {
    int arr[] = {1, 2, 4, 7, 7, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    int sS = secondSmallest(arr, n);
    int sL = secondLargest(arr, n);
    cout << "Second smallest is " << sS << endl;
    cout << "Second largest is " << sL << endl;
    return 0;
}

```

Time Complexity: $O(N)$, we do two linear traversals in our array.

Space Complexity: $O(1)$, as we are using a constant

Check if an Array is Sorted

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

bool isSorted(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[i])
                return false;
        }
    }
    return true;
}

```

```

int main() {
    int arr[] = {1, 2, 3, 4, 5}, n = 5;
    bool ans = isSorted(arr, n);
    if (ans) cout << "True" << endl;
    else cout << "False" << endl;
    return 0;
}

```

Time Complexity: $O(N^2)$, as it uses two nested loops to compare every pair of elements in the array.

Space Complexity: $O(1)$, as no extra space is used apart from a few variables.

Optimal Approach

```

#include<bits/stdc++.h>

using namespace std;

bool isSorted(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        if (arr[i] < arr[i - 1])
            return false;
    }
    return true;
}

```

```

int main() {
    int arr[] = {1, 2, 3, 4, 5}, n = 5;
    printf("%s", isSorted(arr, n) ? "True" : "False");
}

```

Time Complexity: $O(N)$, as it checks each adjacent pair once in a single pass through the array.

Space Complexity: $O(1)$, as it uses constant extra space regardless of input size.

Remove Duplicates in-place from Sorted Array

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int removeDuplicates(vector<int>& nums) {

```

```

unordered_set<int> seen;
int index = 0;
for (int num : nums) {
    if (seen.find(num) == seen.end()) {
        seen.insert(num);
        nums[index] = num;
        index++;
    }
}
return index;
}
};

int main() {
    vector<int> nums = {0,0,1,1,1,2,2,3,3,4};
    Solution sol;
    int k = sol.removeDuplicates(nums);
    cout << "k = " << k << "\nArray after removing
duplicates: ";
    for (int i = 0; i < k; i++) {
        cout << nums[i] << " ";
    }
    cout << endl;
}

```

Time Complexity: $O(N)$, We traverse the entire array and insert elements into set.
Space Complexity: $O(N)$, additional space used to store elements in set.

Optimal Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if (nums.empty()) return 0;
        int i = 0;
        for (int j = 1; j < nums.size(); j++) {
            if (nums[j] != nums[i]) {

```

```

                i++;
                nums[i] = nums[j];
            }
        }
        return i + 1;
    }
};

int main() {
    vector<int> nums = {0,0,1,1,1,2,2,3,3,4};
    Solution sol;
    int k = sol.removeDuplicates(nums);
    cout << "Unique count = " << k << "\n";
    cout << "Array after removing duplicates: ";
    for (int x = 0; x < k; x++) {
        cout << nums[x] << " ";
    }
    cout << endl;
}

```

Time Complexity: $O(N)$, We traverse the entire original array only once.

Space Complexity: $O(1)$, constant additional space is used to check unique elements.

Left Rotate the Array by One

Brute Force

```

#include<bits/stdc++.h>
using namespace std;
void solve(int arr[], int n) {
    int temp[n];
    for (int i = 1; i < n; i++) {
        temp[i - 1] = arr[i];
    }
    temp[n - 1] = arr[0];
    for (int i = 0; i < n; i++) {
        cout << temp[i] << " ";
    }
    cout << endl;
}

```

```

}
int main() {
    int n = 5;
    int arr[] = {1, 2, 3, 4, 5};
    solve(arr, n);
    return 0;
}

```

Time Complexity: $O(N)$, where N is the size of the array. This is because we traverse the array once to shift the elements.

Space Complexity: $O(N)$, as we are using a temporary array of the same size as the input array to store the shifted elements.

Optimal Approach

```

#include<bits/stdc++.h>
using namespace std;
class Solution {
public:
    void rotateArrayByOne(vector<int>& nums) {
        int temp = nums[0];
        for (int i = 1; i < nums.size(); ++i) {
            nums[i - 1] = nums[i];
        }
        nums[nums.size() - 1] = temp;
    }
};
int main() {
    Solution solution;
    vector<int> nums = {1, 2, 3, 4, 5};
    solution.rotateArrayByOne(nums);
    for (int num : nums) {
        cout << num << " ";
    }
    return 0;
}

```

Time Complexity: $O(N)$, where N is the size of the input array. This is because we traverse the array once to shift the elements.

Space Complexity: $O(1)$, as we are using only a constant amount of extra space for the temporary variable.

Rotate array by K elements

Brute Force Approach

```

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

class Solution {
public:
    void rotateRight(int arr[], int n, int k) {
        if (n == 0) return;
        k = k % n;
        int temp[k];
        for (int i = n - k; i < n; i++) {
            temp[i - n + k] = arr[i];
        }
        for (int i = n - k - 1; i >= 0; i--) {
            arr[i + k] = arr[i];
        }
        for (int i = 0; i < k; i++) {
            arr[i] = temp[i];
        }
    }

    void rotateLeft(int arr[], int n, int k) {
        if (n == 0) return;
        k = k % n;
        int temp[k];
        for (int i = 0; i < k; i++) {
            temp[i] = arr[i];
        }
        for (int i = k; i < n; i++) {
            arr[i - k] = arr[i];
        }
        for (int i = 0; i < k; i++) {

```

```

        arr[n - k + i] = temp[i];
    }
}
};

int main() {
    Solution sol;
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 2;
    sol.rotateRight(arr, n, k);

    cout << "Array after right rotation by " << k << "
steps:\n";

    for (int i = 0; i < n; i++) cout << arr[i] << " ";
    cout << "\n";
    int arr2[] = {1, 2, 3, 4, 5, 6, 7};
    sol.rotateLeft(arr2, n, k);

    cout << "Array after left rotation by " << k << "
steps:\n";

    for (int i = 0; i < n; i++) cout << arr2[i] << " ";
    return 0;
}

```

Time Complexity: $O(n)$, We are performing a constant number of linear operations copying 'k' elements and shifting up to 'n-k' elements.

Space Complexity: $O(k)$, A temporary array of size 'k' is used to store either the first 'k' or last 'k' elements depending on the direction of rotation.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void reverseArray(vector<int>& nums, int start,
int end) {
        while (start < end) {
            swap(nums[start], nums[end]);
            start++;
            end--;
        }
    }
}

```

```

    }

    vector<int> rotateArray(vector<int>& nums, int
k, string direction) {
        int n = nums.size();
        if (n == 0 || k == 0) return nums;
        k = k % n;
        if (direction == "right") {
            reverseArray(nums, 0, n - 1);
            reverseArray(nums, 0, k - 1);
            reverseArray(nums, k, n - 1);
        }
        else if (direction == "left") {
            reverseArray(nums, 0, k - 1);
            reverseArray(nums, k, n - 1);
            reverseArray(nums, 0, n - 1);
        }
        return nums;
    }
};

```

```

int main() {
    Solution sol;
    vector<int> nums = {1, 2, 3, 4, 5, 6, 7};
    int k = 2;
    string dir = "right";
    vector<int> result = sol.rotateArray(nums, k,
dir);

    for (int num : result) {
        cout << num << " ";
    }

    return 0;
}

```

Time Complexity: $O(N)$, We reverse parts of the array each reverse takes linear time. So total work is $3 \times O(N) = O(N)$.

Space Complexity: $O(1)$ All modifications are done in-place, using only a few temporary variables.

Move all Zeros to the end of the array

Brute Force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> moveZeroes(vector<int>& arr) {
        vector<int> temp(arr.size(), 0);
        int index = 0;
        for (int i = 0; i < arr.size(); i++) {
            if (arr[i] != 0) {
                temp[index] = arr[i];
                index++;
            }
        }
        for (int i = 0; i < arr.size(); i++) {
            arr[i] = temp[i];
        }
        return arr;
    }
};

int main() {
    vector<int> arr = {0, 1, 0, 3, 12};
    Solution sol;
    vector<int> result = sol.moveZeroes(arr);
    cout << "Array after moving zeroes: ";
    for (int num : result) {
        cout << num << " ";
    }
    cout << endl;
    return 0;
}
```

Time Complexity: $O(N)$, we can move all zeroes to end in linear time.

Space Complexity: $O(N)$, additional space used for temporary array.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        // Pointer to the first zero
        int j = -1;
        for (int i = 0; i < nums.size(); i++) {
            if (nums[i] == 0) {
                j = i;
                break;
            }
        }
        if (j == -1) return;
        for (int i = j + 1; i < nums.size(); i++) {
            if (nums[i] != 0) {
                swap(nums[i], nums[j]);
                j++;
            }
        }
    }
};

int main() {
    Solution sol;
    vector<int> nums = {0, 1, 0, 3, 12};
    sol.moveZeroes(nums);
    for (int num : nums) cout << num << " ";
    cout << endl;
    return 0;
}
```

Time Complexity: $O(N)$, we can move all zeroes to end in linear time.

Space Complexity: $O(1)$, constant additional space is used.

Linear Search in C

```

#include<stdio.h>

int search(int arr[], int n, int num)
{
    int i;
    for(i = 0; i < n; i++)
    {
        if(arr[i] == num)
            return i;
    }
    return -1;
}

int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int num = 4;
    int n = sizeof(arr) / sizeof(arr[0]);
    int val = search(arr, n, num);
    printf("%d", val);
    return 0;
}

```

Time Complexity: $O(N)$, where N is the number of elements in the array. This is because we traverse the entire array to find the element.

Space Complexity: $O(1)$, as we are using a constant

Union of Two Sorted Arrays

Approach 1- Using Map

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> FindUnion(int arr1[], int arr2[], int n, int m) {
        map<int, int> freq;
        vector<int> Union;

        for (int i = 0; i < n; i++)
            freq[arr1[i]]++;
    }
}

```

```

        for (int i = 0; i < m; i++)
            freq[arr2[i]]++;

        for (auto &it : freq)
            Union.push_back(it.first);

        return Union;
    }
};

int main() {
    int n = 10;
    int m = 7;
    int arr1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int arr2[] = {2, 3, 4, 4, 5, 11, 12};

    Solution obj;
    vector<int> Union = obj.FindUnion(arr1, arr2, n, m);

    cout << "Union of arr1 and arr2 is " << endl;

    for (auto &val : Union)
        cout << val << " ";

    return 0;
}

```

Time Complexity : $O((m+n)\log(m+n))$. Inserting a key in map takes $\log N$ times, where N is no of elements in map. At max map can store $m+n$ elements **{when there are no common elements and elements in arr, arr2 are distinct}**. So Inserting $m+n$ th element takes $\log(m+n)$ time. Upon approximation across insertion of all elements in worst it would take $O((m+n)\log(m+n))$ time.

Using **HashMap** also takes the same time, On average insertion in `unordered_map` takes $O(1)$ time but sorting the union vector takes $O((m+n)\log(m+n))$ time. Because at max union vector can have $m+n$ elements.

Space Complexity : $O(m+n)$ {If Space of Union ArrayList is considered} , $O(1)$ {If Space of union ArrayList is not considered}

Approach 2- Using Set

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> findUnion(int arr1[], int arr2[], int n, int m) {

```

```

set<int> st;
for (int i = 0; i < n; i++) {
    st.insert(arr1[i]);
}
for (int i = 0; i < m; i++) {
    st.insert(arr2[i]);
}
vector<int> unionArr(st.begin(), st.end());
return unionArr;
}
};

int main() {
    int arr1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int arr2[] = {2, 3, 4, 4, 5, 11, 12};
    int n = 10, m = 7;
    Solution obj;
    vector<int> result = obj.findUnion(arr1, arr2, n,
m);
    cout << "Union of arr1 and arr2 is: ";
    for (int val : result) {
        cout << val << " ";
    }
    return 0;
}

```

Time Complexity : $O((m+n)\log(m+n))$. Inserting an element in a set takes $\log N$ time, where N is no of elements in the set. At max set can store $m+n$ elements **{when there are no common elements and elements in arr, arr2 are distinct}**. So Inserting $m+n$ th element takes $\log(m+n)$ time. Upon approximation across inserting all elements in worst, it would take $O((m+n)\log(m+n))$ time.

Using **HashSet** also takes the same time, On average insertion in unordered_set takes $O(1)$ time but sorting the union vector takes $O((m+n)\log(m+n))$ time. Because at max union vector can have $m+n$ elements.

Space Complexity : $O(m+n)$ {If Space of Union ArrayList is considered} , **$O(1)$** {If Space of union ArrayList is not considered}

Optimal Approach - Two Pointers

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```

class Solution {
public:
    vector<int> findUnion(int arr1[], int arr2[], int n,
int m) {
        vector<int> Union;
        int i = 0, j = 0;
        while (i < n && j < m) {
            if (arr1[i] < arr2[j]) {
                if (Union.empty() || Union.back() !=
arr1[i])
                    Union.push_back(arr1[i]);
                i++;
            }
            else if (arr2[j] < arr1[i]) {
                if (Union.empty() || Union.back() !=
arr2[j])
                    Union.push_back(arr2[j]);
                j++;
            }
            else {
                if (Union.empty() || Union.back() !=
arr1[i])
                    Union.push_back(arr1[i]);
                i++; j++;
            }
        }
        while (i < n) {
            if (Union.empty() || Union.back() != arr1[i])
                Union.push_back(arr1[i]);
            i++;
        }
        while (j < m) {
            if (Union.empty() || Union.back() != arr2[j])
                Union.push_back(arr2[j]);
            j++;
        }
    }
}

```

```

        return Union;
    }
};

int main() {
    int arr1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int arr2[] = {2, 3, 4, 4, 5, 11, 12};
    int n = 10, m = 7;
    Solution obj;
    vector<int> result = obj.findUnion(arr1, arr2, n, m);
    cout << "Union of arr1 and arr2 is: ";
    for (int val : result) cout << val << " ";
    return 0;
}

```

Time Complexity: $O(m+n)$, Because at max i runs for n times and j runs for m times. When there are no common elements in arr1 and arr2 and all elements in arr1, arr2 are distinct.

Space Complexity : $O(m+n)$ {If Space of Union ArrayList is considered} **$O(1)$** {If Space of union ArrayList is not considered}

Find the missing number in an array

Brute Force

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int missingNumber(vector<int>& a, int N) {
        for (int i = 1; i <= N; i++) {
            int flag = 0; // To check if i exists in array
            for (int j = 0; j < N - 1; j++) {
                if (a[j] == i) {
                    flag = 1;
                    break;
                }
            }
        }
    }
}

```

```

        if (flag == 0) return i;
    }
    return -1;
}

int main() {
    int N = 5;
    vector<int> a = {1, 2, 4, 5};
    Solution obj;
    int ans = obj.missingNumber(a, N);
    cout << "The missing number is: " << ans << endl;
    return 0;
}

```

Time Complexity: $O(N*N)$, since nested for loops are used

Space Complexity: $O(1)$. No extra space used

Optimised Approach 1

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int missingNumber(vector<int>& a, int N) {
        int sum = (N * (N + 1)) / 2;
        int s2 = 0;
        for (int i = 0; i < N - 1; i++) {
            s2 += a[i];
        }
        return sum - s2;
    }
};

int main() {
    int N = 5;
    vector<int> a = {1, 2, 4, 5};
    Solution obj;
    int ans = obj.missingNumber(a, N);
}

```

```

    cout << "The missing number is: " << ans <<
endl;

    return 0;
}

```

Time Complexity: $O(N)$. Single loop is used

Space Complexity: $O(3N)$ where 3 is for the stack, left small array and a right small array

Optimised Approach 2

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int missingNumber(vector<int>& a, int N) {
        int xor1 = 0, xor2 = 0;
        for (int i = 0; i < N - 1; i++) {
            xor2 = xor2 ^ a[i];
            xor1 = xor1 ^ (i + 1);
        }
        xor1 = xor1 ^ N;
        return xor1 ^ xor2;
    }
};

```

```

int main() {
    int N = 5;

    vector<int> a = {1, 2, 4, 5};

    Solution obj;

    int ans = obj.missingNumber(a, N);

    cout << "The missing number is: " << ans <<
endl;

    return 0;
}

```

Time Complexity: $O(N)$. For loop is used

Space Complexity: $O(1)$. No extra space used

Count Maximum Consecutive One's in the array

```

#include <bits/stdc++.h>

```

```

using namespace std;

class Solution {
public:
    int findMaxConsecutiveOnes(vector<int>
&nums) {
        int cnt = 0;
        int maxi = 0;
        for (int i = 0; i < nums.size(); i++) {
            if (nums[i] == 1) {
                cnt++;
            } else {
                cnt = 0;
            }
            maxi = max(maxi, cnt);
        }
        return maxi;
    }
};

int main() {
    vector<int> nums = {1, 1, 0, 1, 1, 1};

    Solution obj;

    int ans = obj.findMaxConsecutiveOnes(nums);

    cout << "The maximum consecutive 1's are " <<
ans;

    return 0;
}

```

Time Complexity: $O(N)$, since we scan the array once.

Space Complexity: $O(1)$, as only constant extra variables are used.

Find the number that appears once, and the other numbers twice

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:

```

```

int getSingleElement(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n; i++) {
        int num = arr[i];
        int cnt = 0;
        for (int j = 0; j < n; j++) {
            if (arr[j] == num)
                cnt++;
        }
        if (cnt == 1) return num;
    }
    return -1;
}

int main() {
    vector<int> arr = {4, 1, 2, 1, 2};
    Solution obj;
    int ans = obj.getSingleElement(arr);

    cout << "The single element is: " << ans <<
endl;

    return 0;
}

```

Time Complexity: $O(N*N)$, since nested for loops are used

Space Complexity: $O(1)$. No extra space used

Better Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int getSingleElement(vector<int>& arr) {
        int n = arr.size();
        int maxi = arr[0];
        for (int i = 0; i < n; i++) {
            maxi = max(maxi, arr[i]);
        }
        vector<int> hash(maxi + 1, 0);

```

```

        for (int i = 0; i < n; i++) {
            hash[arr[i]]++;
        }
        for (int i = 0; i < n; i++) {
            if (hash[arr[i]] == 1)
                return arr[i];
        }
        return -1;
    }
};

int main() {
    vector<int> arr = {4, 1, 2, 1, 2};
    Solution obj;
    int ans = obj.getSingleElement(arr);

    cout << "The single element is: " << ans <<
endl;

    return 0;
}

```

Time Complexity: $O(N)+O(N)+O(N)$, where N = size of the array. One $O(N)$ is for finding the maximum, the second one is to hash the elements and the third one is to search the single element in the array.

Space Complexity: $O(\text{maxElement}+1)$ where maxElement = the maximum element of the array.

Optimised Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int getSingleElement(vector<int>& arr) {
        int n = arr.size();
        int xorr = 0;
        for (int i = 0; i < n; i++) {
            xorr = xorr ^ arr[i];
        }
        return xorr;
    }
};

```

```
int main() {
    vector<int> arr = {4, 1, 2, 1, 2};

    Solution obj;

    int ans = obj.getSingleElement(arr);

    cout << "The single element is: " << ans <<
endl;

    return 0;
}
```

Time Complexity: $O(N)$. Where N is the size of the array

Space Complexity: $O(1)$. No extra space used

Longest Subarray with given Sum K(Positives)

Brute Force

```
#include <bits/stdc++.h>

using namespace std;

class Solution
{
public:
    int longestSubarray(vector<int> &nums, int k)
    {
        int n = nums.size();

        int maxLength = 0;

        for (int startIndex = 0; startIndex < n;
startIndex++) {

            for (int endIndex = startIndex; endIndex <
n; endIndex++) {

                int currentSum = 0;

                for (int i = startIndex; i <= endIndex;
i++) {

                    currentSum += nums[i];

                }

                if (currentSum == k)

                    maxLength = max(maxLength,
endIndex - startIndex + 1);

            }

        }

        return maxLength;
    }
}
```

```
};

int main(){

    vector<int> a = { -1, 1, 1 };

    int k = 1;

    Solution solution;

    int len = solution.longestSubarray(a, k);

    cout << "The length of the longest subarray is: "
<< len << "\n";

    return 0;

}
```

Time Complexity: $O(n^3)$, where n is the size of the array. This is because we have three nested loops: one for the starting index, one for the ending index, and one for calculating the sum of the subarray.

Space Complexity: $O(1)$, as we are using a constant amount of space for variables and not using any additional data structures that grow with input size.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution{
public:
    int longestSubarray(vector<int> &nums, int k){

        int n = nums.size();

        int maxLen = 0;

        int left = 0, right = 0;

        int sum = nums[0];

        while(right < n) {

            while(left <= right && sum > k) {

                sum -= nums[left];

                left++;

            }

            if(sum == k) {

                maxLen = max(maxLen, right - left + 1);

            }

            right++;

            if(right < n) sum += nums[right];

        }

    }
}
```

```

    }
    return maxLen;
}
};

int main() {
    vector<int> nums = {10, 5, 2, 7, 1, 9};
    int k = 15;
    Solution sol;
    int ans = sol.longestSubarray(nums, k);
    cout << "The length of longest subarray having
sum k is: " << ans;
    return 0;
}

```

Time Complexity: $O(N)$, where N is the size of the array. The algorithm traverses the array once with two pointers, making it linear in time complexity.

Space Complexity: $O(1)$, as it uses a constant amount of space.

Longest Subarray with sum K [Postives and Negatives]

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

class SubarraySolver {
public:
    int getLongestSubarray(vector<int>& a, int k) {
        int n = a.size();
        int len = 0;
        for (int i = 0; i < n; i++) {
            for (int j = i; j < n; j++) {
                int sum = 0;
                for (int idx = i; idx <= j; idx++) {
                    sum += a[idx];
                }
                if (sum == k) {
                    len = max(len, j - i + 1);
                }
            }
        }
    }
};

```

```

    }
    return len;
}
};

int main() {
    vector<int> a = { -1, 1, 1 };
    int k = 1;
    SubarraySolver solver;
    int len = solver.getLongestSubarray(a, k);
    cout << "The length of the longest subarray is: "
<< len << "\n";
    return 0;
}

```

Time Complexity: $O(N^3)$ approx., where N = size of the array.

Space Complexity: $O(1)$ as we are not using any extra space.

Better Approach

```

#include <bits/stdc++.h>

using namespace std;

class SubarraySolver {
public:
    int getLongestSubarray(vector<int>& a, int k) {
        int n = a.size();
        int len = 0;
        for (int i = 0; i < n; i++) {
            int s = 0;
            for (int j = i; j < n; j++) {
                s += a[j];
                if (s == k) {
                    len = max(len, j - i + 1);
                }
            }
        }
        return len;
    }
};

```

```

    }
};

int main() {
    vector<int> a = { -1, 1, 1 };
    int k = 1;
    SubarraySolver solver;
    int len = solver.getLongestSubarray(a, k);
    cout << "The length of the longest subarray is: "
    << len << "\n";
    return 0;
}

```

Time Complexity: $O(N^2)$ approx., where N = size of the array.

Space Complexity: $O(1)$ as we are not using any extra space.

Optimal Approach

```

#include <bits/stdc++.h>
using namespace std;
class SubarraySolver {
public:
    int getLongestSubarray(vector<int>& a, int k) {
        int n = a.size();
        map<int, int> preSumMap;
        int sum = 0;
        int maxLen = 0;
        for (int i = 0; i < n; i++) {
            sum += a[i];
            if (sum == k) {
                maxLen = max(maxLen, i + 1);
            }
            int rem = sum - k;
            if (preSumMap.find(rem) !=
preSumMap.end()) {
                int len = i - preSumMap[rem];
                maxLen = max(maxLen, len);
            }
            if (preSumMap.find(sum) ==
preSumMap.end()) {

```

```

                preSumMap[sum] = i;
            }
        }
        return maxLen;
    }
};

int main() {
    vector<int> a = { -1, 1, 1 };
    int k = 1;
    SubarraySolver solver;
    int len = solver.getLongestSubarray(a, k);
    cout << "The length of the longest subarray is: "
    << len << "\n";
    return 0;
}

```

Time Complexity: $O(N)$ or $O(N \cdot \log N)$ depending on which map data structure we are using, where N = size of the array.

Space Complexity: $O(N)$ as we are using a map data structure.

Two Sum : Check if a pair with given sum exists in Array

Brute force Approach

```

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

class Solution {
public:
    string twoSumExists(vector<int>& arr, int
target) {
        int n = arr.size();
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (arr[i] + arr[j] == target) {
                    return "YES";
                }
            }
        }
    }
}

```

```

    }
    return "NO";
}

vector<int> twoSumIndices(vector<int>& arr,
int target) {
    int n = arr.size();
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[i] + arr[j] == target) {
                return {i, j};
            }
        }
    }

    return {-1, -1};
}

};

int main() {
    Solution sol;
    vector<int> arr = {2, 6, 5, 8, 11};
    int target = 14;
    cout << sol.twoSumExists(arr, target) << "\n";
    vector<int> res = sol.twoSumIndices(arr, target);
    cout << "[" << res[0] << ", " << res[1] << "]" << "\n";
    return 0;
}

```

Time Complexity: $O(N^2)$ because we use two nested loops to check every possible pair of elements in the array, where N is the size of the array.

Space Complexity: $O(1)$ as we use a constant amount of extra space regardless of input size

Better Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    string twoSumExists(vector<int>& arr, int
target) {
        unordered_map<int, int> mp;

```

```

        int n = arr.size();
        for (int i = 0; i < n; i++) {
            int complement = target - arr[i];
            if (mp.find(complement) != mp.end()) {
                return "YES"; // Pair found
            }
            mp[arr[i]] = i;
        }
        return "NO";
    }

    vector<int> twoSumIndices(vector<int>& arr,
int target) {
        unordered_map<int, int> mp;
        int n = arr.size();
        for (int i = 0; i < n; i++) {
            int complement = target - arr[i];
            if (mp.find(complement) != mp.end()) {
                return {mp[complement], i};
            }
            mp[arr[i]] = i;
        }
        return {-1, -1};
    }
};

int main() {
    Solution sol;
    vector<int> arr = {2, 6, 5, 8, 11};
    int target = 14;
    cout << sol.twoSumExists(arr, target) << "\n";
    vector<int> res = sol.twoSumIndices(arr, target);
    cout << "[" << res[0] << ", " << res[1] << "]" << "\n";
    return 0;
}

```

```

    Solution sol;
    vector<int> arr = {2, 6, 5, 8, 11};
    int target = 14;
    cout << sol.twoSumExists(arr, target) << "\n";
    vector<int> res = sol.twoSumIndices(arr, target);
    cout << "[" << res[0] << ", " << res[1] << "]" << "\n";
    return 0;
}

```

Time Complexity: $O(N)$ because we traverse the array only once, and each lookup or insertion in the hash map takes $O(1)$ on average, where N is the size of the array.

Space Complexity: $O(N)$ since in the worst case we may store all elements of the array in the hash map.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    string twoSumExists(vector<int> arr, int target) {
        int n = arr.size();
        vector<pair<int, int>> numsWithIndex;
        for (int i = 0; i < n; i++) {
            numsWithIndex.push_back({arr[i], i});
        }
        sort(numsWithIndex.begin(),
            numsWithIndex.end());
        int left = 0, right = n - 1;
        while (left < right) {
            int sum = numsWithIndex[left].first +
                numsWithIndex[right].first;
            if (sum == target) {
                return "YES";
            } else if (sum < target) {
                left++;
            } else {
                right--;
            }
        }
        return "NO";
    }

    vector<int> twoSumIndices(vector<int> arr, int
target) {
        int n = arr.size();
        vector<pair<int, int>> numsWithIndex;
        for (int i = 0; i < n; i++) {
            numsWithIndex.push_back({arr[i], i});
        }
        sort(numsWithIndex.begin(),
            numsWithIndex.end());
        int left = 0, right = n - 1;
```

```
        while (left < right) {
            int sum = numsWithIndex[left].first +
                numsWithIndex[right].first;
            if (sum == target) {
                return {numsWithIndex[left].second,
                    numsWithIndex[right].second};
            } else if (sum < target) {
                left++;
            } else {
                right--;
            }
        }
        return {-1, -1};
    }
};

int main() {
    Solution sol;
    vector<int> arr = {2, 6, 5, 8, 11};
    int target = 14;
    cout << sol.twoSumExists(arr, target) << "\n";
    vector<int> res = sol.twoSumIndices(arr, target);
    cout << "[" << res[0] << ", " << res[1] << "]\n";
    return 0;
}
```

Time Complexity: $O(N \log N)$ due to sorting the array initially, where N is the number of elements. The two-pointer traversal runs in $O(N)$.

Space Complexity: $O(N)$ because we store the array elements along with their original indices in a separate list or vector for sorting, maintaining original positions.

Sort an array of 0s, 1s and 2s

Brute force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
```

```

void sortZeroOneTwo(vector<int>& nums) {
    int count0 = 0, count1 = 0, count2 = 0;
    for(int i = 0; i < nums.size(); i++) {
        if(nums[i] == 0) count0++;
        else if(nums[i] == 1) count1++;
        else count2++;
    }
    int index = 0;
    while(count0--) {
        nums[index++] = 0;
    }
    while(count1--) {
        nums[index++] = 1;
    }
    while(count2--) {
        nums[index++] = 2;
    }
}

int main() {
    vector<int> nums = {1, 0, 2, 1, 0};
    Solution obj;
    obj.sortZeroOneTwo(nums);
    for(int x : nums) {
        cout << x << " ";
    }
    return 0;
}

```

Time Complexity: $O(n)$, We traverse the array twice: once to count, once to overwrite. Each operation is $O(n)$.

Space Complexity: $O(1)$, We use only a constant number of counters regardless of the input size. No extra array is used.

Better Approach

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```

class Solution {
public:
    void sortZeroOneTwo(vector<int>& nums) {
        int cnt0 = 0, cnt1 = 0, cnt2 = 0;
        for (int i = 0; i < nums.size(); i++) {
            if (nums[i] == 0) cnt0++;
            else if (nums[i] == 1) cnt1++;
            else cnt2++;
        }
        for (int i = 0; i < cnt0; i++) {
            nums[i] = 0;
        }
        for (int i = cnt0; i < cnt0 + cnt1; i++) {
            nums[i] = 1;
        }
        for (int i = cnt0 + cnt1; i < nums.size(); i++) {
            nums[i] = 2;
        }
    }
};

int main() {
    vector<int> nums = {0, 2, 1, 2, 0, 1};
    Solution sol;
    sol.sortZeroOneTwo(nums);
    cout << "After sorting:" << endl;
    for (int i = 0; i < nums.size(); i++) {
        cout << nums[i] << " ";
    }
    cout << endl;
    return 0;
}

```

Time Complexity: $O(n)$ We make two passes through the array: one for counting and one for updating. So the total time is proportional to the size of the array.

Space Complexity: $O(1)$, Only three integer variables (cnt0, cnt1, cnt2) are used for counting. No extra space is used proportional to the input size.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void sortZeroOneTwo(vector<int>& nums) {
        int low = 0, mid = 0, high = nums.size() - 1;
        while (mid <= high) {
            if (nums[mid] == 0) {
                swap(nums[mid], nums[low]);
                mid++;
                low++;
            }
            else if (nums[mid] == 1) {
                mid++;
            }
            else {
                swap(nums[mid], nums[high]);
                high--;
            }
        }
    }
};

int main() {
    Solution obj;
    vector<int> nums = {2, 0, 2, 1, 1, 0};
    obj.sortZeroOneTwo(nums);

    for (int val : nums)
        cout << val << " ";

    return 0;
}
```

Time Complexity: $O(n)$ The array is traversed only once using the 'mid' pointer. Each element is checked at most once, and swaps are done in constant time.

Space Complexity: $O(1)$ Only a few integer pointers ('low', 'mid', 'high') are used. Sorting is done in-place, requiring no additional space.

Find the Majority Element that occurs more than $N/2$ times

Brute Force

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int n = nums.size();
        for (int i = 0; i < n; i++) {
            int cnt = 0;
            for (int j = 0; j < n; j++) {
                if (nums[j] == nums[i]) {
                    cnt++;
                }
            }
            if (cnt > (n / 2)) {
                return nums[i];
            }
        }
        return -1;
    }
};

int main() {
    vector<int> arr = {2, 2, 1, 1, 1, 2, 2};
    Solution sol;
    int ans = sol.majorityElement(arr);

    cout << "The majority element is: " << ans << endl;

    return 0;
}
```

Time Complexity: $O(N^2)$, where N is the size of the input array. This is because we are using a nested loop to count the occurrences of each element.

Space Complexity: $O(1)$, as we are using a constant amount of space for the counters and indices.

Better Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int n = nums.size();
        unordered_map<int, int> mp;
        for (int num : nums) {
            mp[num]++;
        }
        for (auto& pair : mp) {
            if (pair.second > n / 2) {
                return pair.first;
            }
        }
        return -1;
    }
};

int main() {
    vector<int> arr = {2, 2, 1, 1, 1, 2, 2};
    Solution sol;
    int ans = sol.majorityElement(arr);
    cout << "The majority element is: " << ans <<
endl;

    return 0;
}
```

Time Complexity: $O(N)$, where N is the size of the input array. This is because we are iterating through the array once to count occurrences and then iterating through the hashmap to find the majority element.

Space Complexity: $O(N)$, as we are using a hashmap to store the counts of each element, which can take up to N space in the worst case.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;
```

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int n = nums.size();
        int cnt = 0;
        int el;
        for (int i = 0; i < n; i++) {
            if (cnt == 0) {
                cnt = 1;
                el = nums[i];
            } else if (el == nums[i]) {
                cnt++;
            } else {
                cnt--;
            }
        }
        int cnt1 = 0;
        for (int i = 0; i < n; i++) {
            if (nums[i] == el) {
                cnt1++;
            }
        }
        if (cnt1 > (n / 2)) {
            return el;
        }
        return -1;
    }
};

int main() {
    vector<int> arr = {2, 2, 1, 1, 1, 2, 2};
    Solution sol;
    int ans = sol.majorityElement(arr);
    // Print the majority element found
    cout << "The majority element is: " << ans <<
endl;
```

```

    return 0;
}

```

Time Complexity: $O(N)$, where N is the size of the input array. This is because we are iterating through the array once to find the potential majority element and then again to verify it.

Space Complexity: $O(1)$, as we are using only a constant amount of extra space.

Kadane's Algorithm : Maximum Subarray Sum in an Array

Brute Force

```

#include<bits/stdc++.h>
using namespace std;
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int maxi = INT_MIN;
        for (int i = 0; i < nums.size(); i++) {
            for (int j = i; j < nums.size(); j++) {
                int sum = 0;
                for (int k = i; k <= j; k++) {
                    sum += nums[k];
                }
                maxi = max(maxi, sum);
            }
        }
        return maxi;
    }
};

int main() {
    vector<int> arr = { -2, 1, -3, 4, -1, 2, 1, -5, 4 };
    Solution sol;
    int maxSum = sol.maxSubArray(arr);
}

```

```

    cout << "The maximum subarray sum is: " <<
    maxSum << endl;
}

```

```

    return 0;
}

```

Time Complexity: $O(N^3)$, where N is the size of the array. This is because we have three nested loops: one for the starting index, one for the ending index, and one for calculating the sum of the subarray.

Space Complexity: $O(1)$, as we are using a constant amount of space for variables, regardless of the input size.

Better Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int maxi = INT_MIN;
        for (int i = 0; i < nums.size(); i++) {
            int sum = 0;
            for (int j = i; j < nums.size(); j++) {
                sum += nums[j];
                maxi = max(maxi, sum);
            }
        }
        return maxi;
    }
};

int main() {
    vector<int> arr = { -2, 1, -3, 4, -1, 2, 1, -5, 4 };
    Solution sol;
    int maxSum = sol.maxSubArray(arr);
    cout << "The maximum subarray sum is: " <<
    maxSum << endl;
    return 0;
}

```

Time Complexity: $O(N^2)$, where N is the size of the array. This is because we have two nested loops: one for the starting index and one for the ending index of the subarray.

Space Complexity: $O(1)$, as we are using a constant amount of space for variables, regardless of the input size.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        long long maxi = LLONG_MIN;
        long long sum = 0;
        for (int i = 0; i < nums.size(); i++) {
            sum += nums[i];
            if (sum > maxi) {
                maxi = sum;
            }
            if (sum < 0) {
                sum = 0;
            }
        }
        return maxi;
    }
};
```

```
int main() {
    vector<int> arr = { -2, 1, -3, 4, -1, 2, 1, -5, 4 };
    Solution sol;
    int maxSum = sol.maxSubArray(arr);
    cout << "The maximum subarray sum is: " <<
    maxSum << endl;
    return 0;
}
```

Time Complexity: $O(n)$, where n is the number of elements in the array. We traverse the array only once.

Space Complexity: $O(1)$. We use a constant amount of space for variables.

Stock Buy And Sell

Brute Force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int stockbuySell(vector<int>& prices) {
        int maxProfit = 0;
        for(int i = 0; i < prices.size(); i++) {
            for(int j = i + 1; j < prices.size(); j++) {
                int profit = prices[j] - prices[i];
                maxProfit = max(maxProfit, profit);
            }
        }
        return maxProfit;
    };
int main() {
    Solution sol;
    vector<int> prices = {7, 1, 5, 3, 6, 4};
    cout << "Max Profit: " <<
    sol.stockbuySell(prices) << endl;
    return 0;
}
```

Time Complexity: $O(n^2)$ Because for each element, we are checking every future element nested loops.

Space Complexity: $O(1)$ No extra space used, only variables for storing max profit.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int stockbuySell(vector<int>& prices) {
        int minPrice = INT_MAX;
        int maxProfit = 0;
        for (int price : prices) {
            if (price < minPrice) {
```

```

        minPrice = price;
    }
    else {
        maxProfit = max(maxProfit, price -
minPrice);
    }
}
return maxProfit;
}
};

int main() {
    Solution obj;

    vector<int> prices = {7, 1, 5, 3, 6, 4};

    cout << obj.stockbuySell(prices) << endl;

    return 0;
}

```

Time Complexity: $O(n)$, This is because we are iterating through the array of prices exactly once. There are no nested loops or recursive calls.

Space Complexity: $O(1)$, Only two variables are used to store the minimum price and maximum profit, regardless of the input size.

Rearrange Array Elements by Sign

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class ArrayManipulator {
public:
    vector<int> RearrangeBySign(vector<int>& A,
int n) {
        vector<int> pos;
        vector<int> neg;
        for (int i = 0; i < n; i++) {
            if (A[i] > 0)
                pos.push_back(A[i]); // Add positive to
pos[]

```

```

        else
            neg.push_back(A[i]); // Add negative to
neg[]
        }
        for (int i = 0; i < n / 2; i++) {
            A[2 * i] = pos[i];
            A[2 * i + 1] = neg[i];
        }
        return A;
    }
};

```

```

int main() {
    int n = 4;

    vector<int> A {1, 2, -4, -5};

    ArrayManipulator obj;

    vector<int> ans = obj.RearrangeBySign(A, n);

    for (int i = 0; i < ans.size(); i++) {
        cout << ans[i] << " ";
    }

    return 0;
}

```

Time Complexity: $O(N+N/2)$ { $O(N)$ for traversing the array once for segregating positives and negatives and another $O(N/2)$ for adding those elements alternatively to the array, where N = size of the array A }.

Space Complexity: $O(N/2 + N/2) = O(N)$ { $N/2$ space required for each of the positive and negative element arrays, where N = size of the array A }.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class ArrayManipulator {
public:
    vector<int> rearrangeBySign(vector<int>& A) {
        int n = A.size();
        vector<int> ans(n, 0);
        int posIndex = 0, negIndex = 1;
        for (int i = 0; i < n; i++) {

```

```

        if (A[i] < 0) {
            ans[negIndex] = A[i];
            negIndex += 2;
        } else {
            ans[posIndex] = A[i];
            posIndex += 2;
        }
    }
    return ans;
}
};

int main() {
    vector<int> A = {1, 2, -4, -5};
    ArrayManipulator obj;
    vector<int> result = obj.rearrangeBySign(A);
    for (int num : result) {
        cout << num << " ";
    }
    return 0;
}

```

Time Complexity: $O(N)$ { $O(N)$ for traversing the array once and substituting positives and negatives simultaneously using pointers, where N = size of the array A }.

Space Complexity: $O(N)$ { Extra Space used to store the rearranged elements separately in an array, where N = size of array A }.

next_permutation : find next lexicographically greater permutation

Brute-Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:

```

```

    vector<int> nextPermutation(vector<int>&
nums) {
        vector<vector<int>> all;
        sort(nums.begin(), nums.end());
        do {
            all.push_back(nums);
        } while (next_permutation(nums.begin(),
nums.end()));
        for (int i = 0; i < all.size(); i++) {
            if (all[i] == nums) {
                if (i == all.size() - 1)
                    return all[0];
                return all[i + 1];
            }
        }
        return nums;
    }
};

```

```

int main() {
    Solution sol;
    vector<int> nums = {1, 2, 3};
    vector<int> result = sol.nextPermutation(nums);
    for (int x : result) cout << x << " ";
    cout << endl;
    return 0;
}

```

Time Complexity: $O(N! \cdot N)$, since we are generating all possible permutations, it takes $N!$ time.

Space Complexity: $O(N!)$, storing all permutations.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void nextPermutation(vector<int>& nums) {
        int index = -1;

```

```

for (int i = nums.size() - 2; i >= 0; i--) {
    if (nums[i] < nums[i + 1]) {
        // Store index
        index = i;
        break;
    }
}
if (index == -1) {
    reverse(nums.begin(), nums.end());
    return;
}
for (int i = nums.size() - 1; i > index; i--) {
    if (nums[i] > nums[index]) {
        swap(nums[i], nums[index]);
        break;
    }
}
reverse(nums.begin() + index + 1,
nums.end());
}
};

int main() {
    vector<int> nums = {1, 2, 3};
    Solution sol;
    sol.nextPermutation(nums);
    for (int num : nums) {
        cout << num << " ";
    }
    cout << endl;
    return 0;
}

```

Time Complexity: $O(N)$, we find the breaking point and reverse the subarray in linear time.
 Space Complexity: $O(1)$, constant additional space is used.

Leaders in an Array

Brute Force

```

#include<bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> leaders(vector<int>& nums) {
        vector<int> ans;
        for (int i = 0; i < nums.size(); i++) {
            bool leader = true;
            for (int j = i + 1; j < nums.size(); j++) {
                if (nums[j] >= nums[i]) {
                    leader = false;
                    break;
                }
            }
            if (leader) {
                ans.push_back(nums[i]);
            }
        }
        return ans;
    }
};

int main() {
    vector<int> nums = {1, 2, 5, 3, 1, 2};
    Solution finder;
    vector<int> ans = finder.leaders(nums);
    cout << "Leaders in the array are: ";
    for (int i = 0; i < ans.size(); i++) {
        cout << ans[i] << " ";
    }
    cout << endl;
    return 0;
}

```

Time Complexity: $O(N^2)$, where N is the size of the input array. This is because we have a nested loop where the outer loop runs N times and the inner loop runs up to N times in the worst case.

Space Complexity: $O(1)$, as we are using only a constant amount of extra space for the answer array, which does not depend on the input size.

Optimal Approach

```
#include<bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> leaders(vector<int>& nums) {
        vector<int> ans;
        if(nums.empty()) {
            return ans;
        }
        int max = nums[nums.size() - 1];
        ans.push_back(nums[nums.size() - 1]);
        for (int i = nums.size() - 2; i >= 0; i--) {
            if (nums[i] > max) {
                ans.push_back(nums[i]);
                max = nums[i];
            }
        }
        reverse(ans.begin(), ans.end());
        return ans;
    }
};

int main() {
    vector<int> nums = {10, 22, 12, 3, 0, 6};
    Solution finder;
    vector<int> ans = finder.leaders(nums);
    cout << "Leaders in the array are: ";
    for (int i = 0; i < ans.size(); i++) {
        cout << ans[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Time Complexity: $O(N)$, where N is the size of the input array. This is because we traverse the array only once.
Space Complexity: $O(1)$, as we are using only a constant amount of extra space.

Longest Consecutive Sequence in an Array

Brute Force

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    bool linearSearch(vector<int>& a, int num) {
        int n = a.size();
        for (int i = 0; i < n; i++) {
            if (a[i] == num)
                return true;
        }
        return false;
    }
public:
    int longestConsecutive(vector<int>& nums) {
        if (nums.size() == 0) {
            return 0;
        }
        int n = nums.size();
        int longest = 1;
        for (int i = 0; i < n; i++) {
            int x = nums[i];
            int cnt = 1;
            while (linearSearch(nums, x + 1) == true) {
                x += 1;
                cnt += 1;
            }
            longest = max(longest, cnt);
        }
        return longest;
    }
};

int main() {
```

```

vector<int> a = {100, 4, 200, 1, 3, 2};

Solution solution;

int ans = solution.longestConsecutive(a);

cout << "The longest consecutive sequence is "
<< ans << "\n";

return 0;
}

```

Time Complexity: $O(n^2)$, where n is the number of elements in the array. This is because for each element, we may need to perform a linear search through the entire array to find consecutive numbers.

Space Complexity: $O(1)$, as we are using a constant amount of extra space for variables.

Better Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
        int n = nums.size();
        if (n == 0) return 0;
        sort(nums.begin(), nums.end());
        int lastSmaller = INT_MIN;
        int cnt = 0;
        int longest = 1;
        for (int i = 0; i < n; i++) {
            if (nums[i] - 1 == lastSmaller) {
                cnt += 1;
                lastSmaller = nums[i];
            }
            else if (nums[i] != lastSmaller) {
                cnt = 1;
                lastSmaller = nums[i];
            }
            longest = max(longest, cnt);
        }
        return longest;
    }
}

```

```

};

int main() {
    vector<int> a = {100, 4, 200, 1, 3, 2};
    Solution solution;
    int ans = solution.longestConsecutive(a);
    cout << "The longest consecutive sequence is "
<< ans << "\n";

    return 0;
}

```

Time Complexity: $O(n \log n)$, where n is the number of elements in the array. This is due to the sorting step, which is the most time-consuming operation in this approach.

Space Complexity: $O(1)$, as we are using only a constant amount of extra space.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int longestConsecutive(vector<int>& a) {
        int n = a.size();
        if (n == 0) return 0;
        int longest = 1;
        unordered_set<int> st;
        for (int i = 0; i < n; i++) {
            st.insert(a[i]);
        }
        for (auto it : st) {
            if (st.find(it - 1) == st.end()) {
                int cnt = 1;
                int x = it;
                while (st.find(x + 1) != st.end()) {
                    x = x + 1;
                    cnt = cnt + 1;
                }
                longest = max(longest, cnt);
            }
        }
    }
}

```

```

    }
    return longest;
}
};

int main() {
    vector<int> a = {100, 4, 200, 1, 3, 2};
    Solution solution;
    int ans = solution.longestConsecutive(a);
    cout << "The longest consecutive sequence is "
    << ans << "\n";
    return 0;
}

```

Time Complexity: $O(n)$, where n is the number of elements in the array. This is because we traverse each element once to build the set and then again to find the longest consecutive sequence.

Space Complexity: $O(n)$, as we use a set to store the unique elements of the array, which in the worst case can be equal to the size of the input array.

Set Matrix Zero

Brute force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void setZeroes(vector<vector<int>>& matrix) {
        int m = matrix.size();
        int n = matrix[0].size();
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == 0) {
                    for (int col = 0; col < n; col++) {
                        if (matrix[i][col] != 0)
                            matrix[i][col] = -1;
                    }
                }
            }
            for (int row = 0; row < m; row++) {
                if (matrix[row][i] != 0)
                    matrix[row][i] = -1;
            }
        }
    }
}

```

```

    }
    }
}

for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        if (matrix[i][j] == -1)
            matrix[i][j] = 0;
    }
}
};

int main() {
    vector<vector<int>> matrix =
    {{1,1,1},{1,0,1},{1,1,1}};
    Solution sol;
    sol.setZeroes(matrix);
    for (auto row : matrix) {
        for (auto val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
    return 0;
}

```

Time Complexity: $O(m * n * (m + n))$, We iterate through every cell ($m * n$), and for each zero, we potentially mark its entire row ($O(n)$) and column ($O(m)$), leading to $O(m * n * (m + n))$ overall.

Space Complexity: $O(1)$, We are not using any extra data structures, only modifying the matrix in place (apart from a few variables).

Better Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void setZeroes(vector<vector<int>>& matrix) {
        int m = matrix.size();

```

```

int n = matrix[0].size();
vector<int> row(m, 0);
vector<int> col(n, 0);
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        if (matrix[i][j] == 0) {
            row[i] = 1;
            col[j] = 1;
        }
    }
}
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        if (row[i] == 1 || col[j] == 1) {
            matrix[i][j] = 0;
        }
    }
}
};

int main() {
    vector<vector<int>> matrix =
    {{1,1,1},{1,0,1},{1,1,1}};

    Solution obj;
    obj.setZeroes(matrix);

    for (auto row : matrix) {
        for (auto val : row) {
            cout << val << " ";
        }
        cout << endl;
    }

    return 0;
}

```

Time Complexity: $O(m \times n)$, We make two passes over the matrix. First pass to identify rows and columns to be zeroed ($O(m \times n)$). Second pass to update the matrix using the markers ($O(m \times n)$). Total time is proportional to the number of cells in the matrix, so $O(m \times n)$.
Space Complexity: $O(m + n)$, We store two extra arrays

one for m rows and one for n columns. No other extra space is used besides these arrays.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void setZeroes(vector<vector<int>>& matrix) {
        int m = matrix.size();
        int n = matrix[0].size();
        bool firstRowZero = false;
        bool firstColZero = false;
        for (int j = 0; j < n; j++) {
            if (matrix[0][j] == 0) {
                firstRowZero = true;
                break;
            }
        }
        for (int i = 0; i < m; i++) {
            if (matrix[i][0] == 0) {
                firstColZero = true;
                break;
            }
        }
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                if (matrix[i][j] == 0) {
                    matrix[i][0] = 0;
                    matrix[0][j] = 0;
                }
            }
        }
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                if (matrix[i][0] == 0 || matrix[0][j] == 0) {
                    matrix[i][j] = 0;
                }
            }
        }
    }
};

```

```

        }
    }
}

if (firstRowZero) {
    for (int j = 0; j < n; j++) {
        matrix[0][j] = 0;
    }
}

if (firstColZero) {
    for (int i = 0; i < m; i++) {
        matrix[i][0] = 0;
    }
}
}

};

int main() {
    Solution obj;

    vector<vector<int>> matrix =
    {{0,1,2,0},{3,4,5,2},{1,3,1,5}};

    obj.setZeroes(matrix);

    for (auto row : matrix) {
        for (auto val : row) {
            cout << val << " ";
        }

        cout << endl;
    }

    return 0;
}

```

Time Complexity: $O(m \times n)$, We iterate over the entire matrix a constant number of times (first pass for markers, second pass for zeroing, final pass for first row/col), where m = number of rows and n = number of columns.
 Space Complexity: $O(1)$, No extra space is used apart from a few boolean flags; all marker information is stored within the first row and column of the matrix itself.

Rotate Image by 90 degree

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<vector<int>>
    rotateClockwise(vector<vector<int>>& matrix) {
        int n = matrix.size();

        vector<vector<int>> rotated(n,
        vector<int>(n));

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                rotated[j][n - i - 1] = matrix[i][j];
            }
        }

        return rotated;
    }
};

int main() {
    vector<vector<int>> mat = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    Solution obj;

    vector<vector<int>> rotated =
    obj.rotateClockwise(mat);

    for (auto row : rotated) {
        for (int val : row) cout << val << " ";

        cout << endl;
    }

    return 0;
}

```

Time Complexity: $O(N^2)$, Each element of the matrix is visited exactly once and placed into a new matrix, so the time taken is proportional to the total number of elements.

Space Complexity: $O(N^2)$, We use an additional matrix of the same size to store the rotated result, leading to $O(N^2)$ extra space.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void rotateClockwise(vector<vector<int>>& matrix) {
        int n = matrix.size();
        for (int i = 0; i < n; ++i) {
            for (int j = i + 1; j < n; ++j) {
                swap(matrix[i][j], matrix[j][i]);
            }
        }
        for (int i = 0; i < n; ++i) {
            reverse(matrix[i].begin(), matrix[i].end());
        }
    }
};

int main() {
    vector<vector<int>> matrix = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
    Solution obj;
    obj.rotateClockwise(matrix);
    for (auto row : matrix) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Time Complexity: $O(N^2)$, We traverse every element once during transposition and again during reversal of each row, resulting in a total of $O(N^2)$ time.

Space Complexity: $O(1)$, All operations are done in-place using only temporary variables. No extra matrix is used.

Spiral Traversal of Matrix

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        vector<int> result;
        int top = 0;
        int bottom = matrix.size() - 1;
        int left = 0;
        int right = matrix[0].size() - 1;
        while(top <= bottom && left <= right) {
            for(int i = left; i <= right; i++) {
                result.push_back(matrix[top][i]);
            }
            top++; // Move top boundary down
            for(int i = top; i <= bottom; i++) {
                result.push_back(matrix[i][right]);
            }
            right--; // Move right boundary left
            if(top <= bottom) {
                for(int i = right; i >= left; i--) {
                    result.push_back(matrix[bottom][i]);
                }
                bottom--; // Move bottom boundary up
            }
            if(left <= right) {
                for(int i = bottom; i >= top; i--) {
                    result.push_back(matrix[i][left]);
                }
                left++; // Move left boundary right
            }
        }
    }
}
```

```

    }
    return result;
}
};

int main() {
    Solution obj;
    vector<vector<int>> matrix = {
        { 1, 2, 3, 4 },
        { 5, 6, 7, 8 },
        { 9, 10, 11, 12 },
        { 13, 14, 15, 16 }
    };

    vector<int> result = obj.spiralOrder(matrix);
    for(int val : result) {
        cout << val << " ";
    }

    return 0;
}

```

Time Complexity: $O(m \times n)$, Because we visit each element of the matrix exactly once, where `m` is the number of rows and `n` is the number of columns.

Space Complexity: $O(1)$ We use only a few integer variables to keep track of boundaries (top, bottom, left, right). The `result` vector is part of the output, so it's not counted as extra space.

Count Subarray sum Equals K

Brute force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int subarraySum(vector<int>& arr, int k) {
        int n = arr.size();
        int count = 0;
        for (int i = 0; i < n; i++) {
            for (int j = i; j < n; j++) {

```

```

                int sum = 0;
                for (int m = i; m <= j; m++) {
                    sum += arr[m];
                }
                if (sum == k) {
                    count++;
                }
            }
        }
        return count;
    }
};

int main() {
    vector<int> arr = {3, 1, 2, 4};
    int k = 6;
    Solution sol;
    int result = sol.subarraySum(arr, k);

    cout << "The number of subarrays is: " << result
    << "\n";

    return 0;
}

```

Time Complexity: $O(N^3)$, where N = size of the array. We are using three nested loops here. Though all are not running for exactly N times, the time complexity will be approximately $O(N^3)$.

Space Complexity: $O(1)$ as we are not using any extra space.

Better Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int subarraySum(vector<int>& arr, int k) {
        int n = arr.size();
        int count = 0;
        for (int i = 0; i < n; i++) {

```

```

        int sum = 0;
        for (int j = i; j < n; j++) {
            sum += arr[j];
            if (sum == k) {
                count++;
            }
        }
    }
    return count;
}

int main() {
    vector<int> arr = {3, 1, 2, 4};
    int k = 6;
    Solution sol;
    int result = sol.subarraySum(arr, k);

    cout << "The number of subarrays is: " << result
    << "\n";

    return 0;
}

```

Time Complexity: $O(n^2)$, We use two nested loops to check all subarrays, where n is the size of the array.
Space Complexity: $O(1)$, Only a few extra variables are used, so constant extra space regardless of input size.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int subarraySum(vector<int>& arr, int k) {
        int n = arr.size();
        unordered_map<int, int> prefixSumCount;
        int prefixSum = 0;
        int count = 0;
        prefixSumCount[0] = 1;
        for (int i = 0; i < n; i++) {
            prefixSum += arr[i];

```

```

            int remove = prefixSum - k;
            if (prefixSumCount.find(remove) !=
                prefixSumCount.end()) {
                count += prefixSumCount[remove];
            }
            prefixSumCount[prefixSum]++;
        }
        return count;
    }
};

int main() {
    vector<int> arr = {3, 1, 2, 4};
    int k = 6;
    Solution sol;
    int result = sol.subarraySum(arr, k);

    cout << "The number of subarrays is: " << result
    << "\n";

    return 0;
}

```

Time Complexity: $O(n)$ We traverse the array once, where n is the size of the array. Each prefix sum operation and hashmap lookup is $O(1)$ on average.

Space Complexity: $O(n)$ In the worst case, all prefix sums are distinct and stored in the hashmap, so space grows linearly with input size.

Program to generate Pascal's Triangle

Approach – 1

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<vector<int>> generate(int numRows) {
        vector<vector<int>> triangle;
        for (int i = 0; i < numRows; i++) {
            vector<int> row(i + 1, 1);
            for (int j = 1; j < i; j++) {

```

```

        row[j] = triangle[i - 1][j - 1] + triangle[i - 1][j];
    }
    triangle.push_back(row);
}
return triangle;
}
};

int main() {
    Solution obj;
    int n = 5;
    vector<vector<int>> result = obj.generate(n);
    for (auto &row : result) {
        for (auto &val : row) cout << val << " ";
        cout << endl;
    }
}

```

Time Complexity: $O(N^2)$, we generate all the elements in first N rows sequentially one by one.

Space Complexity: $O(N^2)$, additional space used for storing the entire pascal triangle.

Approach – 2

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    vector<long long> getNthRow(int N) {
        vector<long long> row;
        long long val = 1;
        row.push_back(val);
        for (int k = 1; k < N; k++) {
            val = val * (N - k) / k;
            row.push_back(val);
        }

        return row;
    }
}

```

```

};

int main() {
    int N = 5;
    Solution sol;
    vector<long long> result = sol.getNthRow(N);
    for (auto num : result) {
        cout << num << " ";
    }
    return 0;
}

```

Time Complexity: $O(N)$, we iterate N times to compute each element of the row in $O(1)$ time using the direct relation.

Space Complexity: $O(N)$, additional space used for storing the Nth row.

Approach - 3

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    long long findPascalElement(int r, int c) {
        int n = r - 1;
        int k = c - 1;
        long long result = 1;
        for (int i = 0; i < k; i++) {
            result *= (n - i);
            result /= (i + 1);
        }
        return result;
    }
};

int main() {
    Solution sol;
    int r = 5, c = 3;
    cout << sol.findPascalElement(r, c);
    return 0;
}

```

Time Complexity: $O(\min(c, r-c))$, The loop runs for $\min(c-1, r-c)$ iterations because binomial coefficients are symmetric.

Space Complexity: $O(1)$, constant additional space is used.

Majority Elements(>N/3 times) | Find the elements that appears more than N/3 times in the array

Brute Force

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> majorityElementTwo(vector<int>&
nums) {
        int n = nums.size();
        vector<int> result;
        for (int i = 0; i < n; i++) {
            if (result.size() == 0 || result[0] != nums[i]) {
                int cnt = 0;
                for (int j = 0; j < n; j++) {
                    if (nums[j] == nums[i]) {
                        cnt++;
                    }
                }
                if (cnt > (n / 3))
                    result.push_back(nums[i]);
            }
            if (result.size() == 2) break;
        }
        return result;
    }
};

int main() {
    vector<int> arr = {11, 33, 33, 11, 33, 11};
    Solution sol;
    vector<int> ans = sol.majorityElementTwo(arr)
```

```
cout << "The majority elements are: ";
for (auto it : ans) {
    cout << it << " ";
}

cout << "\n";

return 0;
}
```

Time Complexity: $O(N^2)$, where N is the size of the array. This is because for each element, we are traversing the entire array to count its occurrences.

Space Complexity: $O(1)$, as we are using a constant amount of space for the result array, which can hold at most 2 elements.

Better Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> majorityElementTwo(vector<int>&
nums) {
        int n = nums.size();
        vector<int> result;
        unordered_map<int, int> mpp;
        int mini = int(n / 3) + 1;
        for (int i = 0; i < n; i++) {
            mpp[nums[i]]++;
            if (mpp[nums[i]] == mini) {
                result.push_back(nums[i]);
            }
            if (result.size() == 2) {
                break;
            }
        }
        return result;
    }
};

int main() {
    vector<int> arr = {11, 33, 33, 11, 33, 11};
```

```

Solution sol;

vector<int> ans = sol.majorityElementTwo(arr);

cout << "The majority elements are: ";

for (auto it : ans) {
    cout << it << " ";
}

cout << "\n";

return 0;
}

```

Time Complexity: $O(N * \log N)$, where N is the size of the given array. For using a map data structure, where insertion in the map takes $\log N$ time, and we are doing it for N elements. So, it results in the first term $O(N * \log N)$. On using `unordered_map` instead, the first term will be $O(N)$ for the best and average case, and for the worst case, it will be $O(N^2)$.

Space Complexity: $O(N)$ for using a map data structure. A list that stores a maximum of 2 elements is also used, but that space used is so small that it can be considered constant.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> majorityElementTwo(vector<int>& nums) {
        int n = nums.size();

        int cnt1 = 0, cnt2 = 0;

        int el1 = INT_MIN, el2 = INT_MIN;

        for (int i = 0; i < n; i++) {
            if (cnt1 == 0 && el2 != nums[i]) {
                cnt1 = 1;
                el1 = nums[i];
            }

            else if (cnt2 == 0 && el1 != nums[i]) {
                cnt2 = 1;
                el2 = nums[i];
            }

            else if (nums[i] == el1) {
                cnt1++;
            }
        }
    }
};

```

```

}

else if (nums[i] == el2) {
    cnt2++;
}

else {
    cnt1--;
    cnt2--;
}

}

cnt1 = 0, cnt2 = 0;

for (int i = 0; i < n; i++) {
    if (nums[i] == el1) {
        cnt1++;
    }

    if (nums[i] == el2) {
        cnt2++;
    }
}

int mini = n / 3 + 1;

vector<int> result;

if (cnt1 >= mini) {
    result.push_back(el1);
}

if (cnt2 >= mini && el1 != el2) {
    result.push_back(el2);
}

return result;
}

};

int main() {
    vector<int> arr = {11, 33, 33, 11, 33, 11};

    Solution sol;

    vector<int> ans = sol.majorityElementTwo(arr);

    cout << "The majority elements are: ";

    for (auto it : ans) {
        cout << it << " ";
    }
}

```

```

    }
    cout << "\n";
    return 0;
}

```

Time Complexity: $O(N)$, where N is the size of the input array. We traverse the array twice: once to find potential candidates and once to validate them.

Space Complexity: $O(1)$, as we are using a constant amount of space for the counters and candidate elements, regardless of the input size.

3 Sum : Find triplets that add up to a zero

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& arr,
int n) {
        set<vector<int>> st;
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                for (int k = j + 1; k < n; k++) {
                    if (arr[i] + arr[j] + arr[k] == 0) {
                        vector<int> temp = {arr[i], arr[j],
arr[k]};
                        sort(temp.begin(), temp.end());
                        st.insert(temp);
                    }
                }
            }
        }
        vector<vector<int>> ans(st.begin(), st.end());
        return ans;
    }
}

```

```

};

int main() {
    vector<int> arr = {-1, 0, 1, 2, -1, -4};
    int n = arr.size();

    Solution obj;

    vector<vector<int>> res = obj.threeSum(arr, n);

    for (auto &triplet : res) {
        for (auto &num : triplet) cout << num << " ";
        cout << endl;
    }

    return 0;
}

```

Time Complexity: $O(N^3 * \log(\text{no. of unique triplets}))$, where N = size of the array.

Reason: Here, we are mainly using 3 nested loops. And inserting triplets into the set takes $O(\log(\text{no. of unique triplets}))$ time complexity. But we are not considering the time complexity of sorting as we are just sorting 3 elements every time.

Space Complexity: $O(2 * \text{no. of the unique triplets})$ as we are using a set data structure and a list to store the triplets.

Better Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& arr,
int n) {
        set<vector<int>> ans;

        for (int i = 0; i < n; i++) {
            set<int> hashset;

            for (int j = i + 1; j < n; j++) {
                int third = -(arr[i] + arr[j]);

                if (hashset.find(third) != hashset.end()) {
                    vector<int> temp = {arr[i], arr[j],
third};
                    sort(temp.begin(), temp.end());
                    ans.insert(temp);
                }
            }
        }
    }
}

```

```

        hashset.insert(arr[j]);
    }
}

return vector<vector<int>>(ans.begin(),
ans.end());
}

};

int main() {
    vector<int> arr = {-1, 0, 1, 2, -1, -4};
    int n = arr.size();
    Solution obj;
    vector<vector<int>> res = obj.threeSum(arr, n);
    for (auto &triplet : res) {
        for (auto &num : triplet) cout << num << " ";
        cout << endl;
    }
    return 0;
}

```

Time Complexity: $O(N^2 * \log(\text{no. of unique triplets}))$, as we are mainly using 3 nested loops. And inserting triplets into the set takes $O(\log(\text{no. of unique triplets}))$ time complexity. But we are not considering the time complexity of sorting as we are just sorting 3 elements every time.

Space Complexity: $O(2 * \text{no. of the unique triplets}) + O(N)$ as we are using a set data structure and a list to store the triplets and extra $O(N)$ for storing the array elements in another set.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& arr,
int n) {
        sort(arr.begin(), arr.end());
        vector<vector<int>> ans;
        for (int i = 0; i < n; i++) {
            if (i > 0 && arr[i] == arr[i - 1]) continue;
            int left = i + 1, right = n - 1;

```

```

            while (left < right) {
                int sum = arr[i] + arr[left] + arr[right];
                if (sum == 0) {
                    ans.push_back({arr[i], arr[left],
arr[right]});
                    left++, right--;
                    while (left < right && arr[left] ==
arr[left - 1]) left++;
                    while (left < right && arr[right] ==
arr[right + 1]) right--;
                }
                else if (sum < 0) left++;
                else right--;
            }
        }
        return ans;
    }
};

```

```

int main() {
    vector<int> arr = {-1, 0, 1, 2, -1, -4};
    int n = arr.size();
    Solution obj;
    vector<vector<int>> res = obj.threeSum(arr, n);
    for (auto &triplet : res) {
        for (auto &num : triplet) cout << num << " ";
        cout << endl;
    }
    return 0;
}

```

Time Complexity: $O(N \log N) + O(N^2)$, as The pointer i, is running for approximately N times. And both the pointers j and k combined can run for approximately N times including the operation of skipping duplicates. So the total time complexity will be $O(N^2)$.

Space Complexity: $O(\text{no. of quadruplets})$, **This space is only used to store the answer. We are not using any extra space to solve this problem.** So, from that perspective, space complexity can be written as $O(1)$.

4 Sum | Find Quads that add up to a target value

Brute Force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& arr,
int target) {
        int n = arr.size();
        set<vector<int>> st;
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                for (int k = j + 1; k < n; k++) {
                    for (int l = k + 1; l < n; l++) {
                        long long sum = (long long)arr[i] +
arr[j] + arr[k] + arr[l];
                        if (sum == target) {
                            vector<int> temp = {arr[i], arr[j],
arr[k], arr[l]};
                            sort(temp.begin(), temp.end());
                            st.insert(temp);
                        }
                    }
                }
            }
        }
        return vector<vector<int>>(st.begin(),
st.end());
    }
};

int main() {
    vector<int> arr = {1, 0, -1, 0, -2, 2};
    int target = 0;
    Solution obj;
    vector<vector<int>> ans = obj.fourSum(arr,
target);
    for (auto quad : ans) {
```

```
        for (int num : quad) cout << num << " ";
        cout << endl;
    }
    return 0;
}
```

Time Complexity: $O(N^3 * \log(\text{no. of unique triplets}))$, where N = size of the array.

Reason: Here, we are mainly using 3 nested loops. And inserting triplets into the set takes $O(\log(\text{no. of unique triplets}))$ time complexity. But we are not considering the time complexity of sorting as we are just sorting 3 elements every time.

Space Complexity: $O(2 * \text{no. of the unique triplets})$ as we are using a set data structure and a list to store the triplets.

Better Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& arr,
int target) {
        int n = arr.size();
        set<vector<int>> st;
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                unordered_set<int> seen;
                for (int k = j + 1; k < n; k++) {
                    long long required = (long long)target -
arr[i] - arr[j] - arr[k];
                    if (seen.count(required)) {
                        vector<int> temp = {arr[i], arr[j],
arr[k], (int)required};
                        sort(temp.begin(), temp.end());
                        st.insert(temp);
                    }
                }
                seen.insert(arr[k]);
            }
        }
    }
}
```

```

        return vector<vector<int>>>(st.begin(),
st.end());
    }
};

int main() {
    vector<int> arr = {1, 0, -1, 0, -2, 2};
    int target = 0;
    Solution obj;
    vector<vector<int>>> ans = obj.fourSum(arr,
target);
    for (auto quad : ans) {
        for (int num : quad) cout << num << " ";
        cout << endl;
    }
    return 0;
}

```

Time Complexity: $O(N^3 \log(M))$, as we are mainly using 3 nested loops, and inside the loops there are some operations on the set data structure which take $\log(M)$ time complexity.

Space Complexity: $O(2 * \text{no. of the quadruplets}) + O(N)$, as we are using a set data structure and a list to store the quads. This results in the first term. And the second space is taken by the set data structure we are using to store the array elements. At most, the set can contain approximately all the array elements and so the space complexity is $O(N)$.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<vector<int>>> fourSum(vector<int>& arr,
int target) {
        int n = arr.size();
        vector<vector<int>>> ans;
        sort(arr.begin(), arr.end());
        for (int i = 0; i < n; i++) {
            if (i > 0 && arr[i] == arr[i - 1]) continue;
            for (int j = i + 1; j < n; j++) {
                if (j > i + 1 && arr[j] == arr[j - 1])
continue;

```

```

                int left = j + 1, right = n - 1;
                while (left < right) {
                    long long sum = (long long)arr[i] +
arr[j] +
                        arr[left] + arr[right];

                    if (sum == target) {
                        ans.push_back({arr[i], arr[j],
                        arr[left], arr[right]});
                        while (left < right && arr[left] ==
arr[left + 1])
                            left++;
                        while (left < right && arr[right] ==
arr[right - 1])
                            right--;
                        left++;
                        right--;
                    }
                    else if (sum < target) left++;
                    else right--;
                }
            }
        }
        return ans;
    }
};

int main() {
    vector<int> arr = {1, 0, -1, 0, -2, 2};
    int target = 0;
    Solution obj;
    vector<vector<int>>> ans = obj.fourSum(arr,
target);
    for (auto quad : ans) {
        for (int num : quad) cout << num << " ";
        cout << endl;
    }
    return 0;
}

```

```
}
```

Time Complexity: $O(N^3)$, as Each of the pointers i and j , is running for approximately N times. And both the pointers k and l combined can run for approximately N times including the operation of skipping duplicates. So the total time complexity will be $O(N^3)$.

Space Complexity: $O(\text{no. of quadruplets})$, as This space is only used to store the answer. We are not using any extra space to solve this problem. So, from that perspective, space complexity can be written as $O(1)$.

Length of the longest subarray with zero Sum

Brute Force

```
#include <bits/stdc++.h>

using namespace std;

int solve(vector<int>& a) {
    int maxLen = 0;
    unordered_map<int, int> sumIndexMap;
    int sum = 0;
    for (int i = 0; i < (int)a.size(); i++) {
        sum += a[i];
        if (sum == 0) {
            maxLen = i + 1;
        }
        else if (sumIndexMap.find(sum) !=
sumIndexMap.end()) {
            maxLen = max(maxLen, i -
sumIndexMap[sum]);
        }
        else {
            sumIndexMap[sum] = i;
        }
    }
    return maxLen;
}

int main() {
    vector<int> a = {9, -3, 3, -1, 6, -5};
```

```
cout << solve(a) << endl;

return 0;
}
```

Time Complexity: $O(N^2)$, where N is the size of the array. This is because we are using two nested loops to check all possible subarrays.

Space Complexity: $O(1)$, as we are not using any additional data structures that grow with input size. We are only using a few variables to store the maximum length and the current sum.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

int maxLen(int A[], int n) {
    unordered_map<int, int> mpp;
    int maxi = 0;
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += A[i];
        if (sum == 0) {
            maxi = i + 1;
        }
        else {
            if (mpp.find(sum) != mpp.end()) {
                maxi = max(maxi, i - mpp[sum]);
            }
            else {
                mpp[sum] = i;
            }
        }
    }
    return maxi;
}

int main() {
    int A[] = {9, -3, 3, -1, 6, -5};
    int n = sizeof(A) / sizeof(A[0]);
    cout << maxLen(A, n) << endl;

    return 0;
}
```

```
}
```

Time Complexity: $O(n)$, where n is the length of the string. This is because we are using a single pass through the string with two pointers, leading to linear time complexity.

Space Complexity: $O(1)$, as we are using a fixed-size hash array of size 256 (for ASCII characters) and not using any additional data structures that grow with input size.

Count the number of subarrays with given xor K

Brute-Force Approach

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Solution {
```

```
public:
```

```
    int countSubarraysXOR(vector<int>& A, int B) {
```

```
        int count = 0;
```

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

```
            int xorVal = 0;
```

```
            for (int j = i; j < A.size(); j++) {
```

```
                xorVal ^= A[j];
```

```
                if (xorVal == B) {
```

```
                    count++;
```

```
                }
```

```
            }
```

```
        }
```

```
        return count;
```

```
    }
```

```
};
```

```
int main() {
```

```
    vector<int> A = {4, 2, 2, 6, 4};
```

```
    int B = 6;
```

```
    Solution sol;
```

```
    cout << sol.countSubarraysXOR(A, B) << endl;
```

```
    return 0;
```

```
}
```

Time Complexity: $O(N^2)$, we generate all possible subarrays to check their XOR.

Space Complexity: $O(1)$, constant amount of extra space is used.

Optimal Approach

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Solution {
```

```
public:
```

```
    int countSubarrays(vector<int>& A, int k) {
```

```
        unordered_map<int, int> freq;
```

```
        freq[0] = 1;
```

```
        int prefixXor = 0;
```

```
        int count = 0;
```

```
        for (int num : A) {
```

```
            prefixXor ^= num;
```

```
            int target = prefixXor ^ k
```

```
            if (freq.find(target) != freq.end()) {
```

```
                count += freq[target];
```

```
            }
```

```
            freq[prefixXor]++;
```

```
        }
```

```
        return count;
```

```
    }
```

```
};
```

```
int main() {
```

```
    vector<int> A = {4, 2, 2, 6, 4};
```

```
    int k = 6;
```

```
    Solution sol;
```

```
    cout << sol.countSubarrays(A, k) << endl;
```

```
    return 0;
```

```
}
```

Time Complexity: $O(N)$, we traverse the entire array once to calculate prefix XOR and subarrays with given XOR.

Space Complexity: $O(N)$, additional amount of extra space is used to store frequencies of prefix XOR in hash map.

Merge Overlapping Sub-intervals

Brute-Force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<vector<int>>
    merge(vector<vector<int>>& intervals) {
        sort(intervals.begin(), intervals.end());
        vector<vector<int>> ans;
        int n = intervals.size();
        for (int i = 0; i < n; ) {
            int start = intervals[i][0];
            int end = intervals[i][1];
            int j = i + 1;
            while (j < n && intervals[j][0] <= end) {
                end = max(end, intervals[j][1]);
                j++;
            }
            ans.push_back({start, end});
            i = j;
        }
        return ans;
    }
};

int main() {
    Solution sol;

    vector<vector<int>> intervals = {{1,3}, {2,6},
    {8,10}, {15,18}};

    vector<vector<int>> result =
    sol.merge(intervals);

    for (auto interval : result) {
        cout << "[" << interval[0] << ", " <<
    interval[1] << "]" << " ";
    }

    return 0;
}
```

Time Complexity: $O(N^2)$, for every interval we check all future intervals.

Space Complexity: $O(N)$, additional space used to store the non-overlapping intervals.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<vector<int>>
    merge(vector<vector<int>>& intervals) {
        sort(intervals.begin(), intervals.end());
        vector<vector<int>> merged;
        for (auto interval : intervals) {
            if (merged.empty() || merged.back()[1] <
            interval[0]) {
                merged.push_back(interval);
            } else {
                merged.back()[1] = max(
                merged.back()[1],
                interval[1]
            );
            }
        }
        return merged;
    }
};

int main() {
    Solution sol;

    vector<vector<int>> intervals = {
    {1, 3}, {2, 6}, {8, 10}, {15, 18}
    };

    vector<vector<int>> result =
    sol.merge(intervals);

    for (auto v : result) {
        cout << "[" << v[0] << ", " << v[1] << "]" << " ";
    }

    return 0;
}
```

```
}
```

Time Complexity: $O(N \cdot \log N) + O(N)$, we sort the entire array and then merge them in a single pass.

Space Complexity: $O(N)$, additional space used to store the non-overlapping intervals.

Merge two Sorted Arrays Without Extra Space

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void merge(vector<int>& nums1, int m,
vector<int>& nums2, int n) {
        int i = m - 1;
        int j = n - 1;
        int k = m + n - 1;
        while (i >= 0 && j >= 0) {
            if (nums1[i] > nums2[j]) {
                nums1[k--] = nums1[i--];
            } else {
                nums1[k--] = nums2[j--];
            }
        }
        while (j >= 0) {
            nums1[k--] = nums2[j--];
        }
    }
};

int main() {
    vector<int> nums1 = {1, 3, 5, 0, 0, 0};
    vector<int> nums2 = {2, 4, 6};
    int m = 3, n = 3;
    Solution().merge(nums1, m, nums2, n);
    for (int num : nums1) cout << num << " ";
    return 0;
}
```

Time Complexity: $O(N+M)$, we traverse both the arrays exactly once.

Space Complexity: $O(1)$, constant extra space is used to store pointers.

Find the repeating and missing numbers

Brute Force

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int>
findMissingRepeatingNumbers(vector<int>&
nums) {
        int n = nums.size();
        int repeating = -1, missing = -1;
        for (int i = 1; i <= n; i++) {
            int cnt = 0;
            for (int j = 0; j < n; j++) {
                if (nums[j] == i) cnt++;
            }
            if (cnt == 2) repeating = i;
            else if (cnt == 0) missing = i;
            if (repeating != -1 && missing != -1)
                break;
        }
        return {repeating, missing};
    }
};

int main() {
    vector<int> nums = {3, 1, 2, 5, 4, 6, 7, 5};
    Solution sol;
    vector<int> result =
sol.findMissingRepeatingNumbers(nums);
    cout << "The repeating and missing numbers are:
" << result[0] << ", " << result[1] << "\n";
    return 0;
}
```

Time Complexity: $O(N^2)$, where N is the size of the array. This is because we are iterating through the array for each integer from 1 to N, leading to a nested loop.
Space Complexity: $O(1)$, as we are using a constant amount of space for the variables 'repeating' and 'missing', regardless of the input size.

Better Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int>
    findMissingRepeatingNumbers(vector<int>&
    nums) {
        int n = nums.size();
        int hash[n + 1] = {0};
        for (int i = 0; i < n; i++) {
            hash[nums[i]]++;
        }
        int repeating = -1, missing = -1;
        for (int i = 1; i <= n; i++) {
            if (hash[i] == 2) {
                repeating = i;
            } else if (hash[i] == 0) {
                missing = i;
            }
        }
        if (repeating != -1 && missing != -1) {
            break;
        }
    }
    return {repeating, missing};
};

int main() {
    vector<int> nums = {3, 1, 2, 5, 4, 6, 7, 5};
    Solution sol;
    vector<int> result =
    sol.findMissingRepeatingNumbers(nums);
    cout << "The repeating and missing numbers are:
    {" << result[0] << ", " << result[1] << "}\n";
```

```
        return 0;
    }
```

Time Complexity: $O(2*N)$, where N is the size of the array. This is because we are iterating through the array once to build the hash array and then iterating through the hash array to find the repeating and missing numbers.
Space Complexity: $O(N)$, as we are using an additional hash array of size N+1 to store the frequency of each element.

Optimal Approach 1

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int>
    findMissingRepeatingNumbers(vector<int>&
    nums) {
        long long n = nums.size();
        long long SN = (n * (n + 1)) / 2;
        long long S2N = (n * (n + 1) * (2 * n + 1)) / 6;
        long long S = 0, S2 = 0;
        for (int i = 0; i < n; i++) {
            S += nums[i];
            S2 += (long long)nums[i] * (long
            long)nums[i];
        }
        long long val1 = S - SN;
        long long val2 = S2 - S2N;
        val2 = val2 / val1;
        long long x = (val1 + val2) / 2;
        long long y = x - val1;
        return {(int)x, (int)y};
    }
};

int main() {
    vector<int> nums = {3, 1, 2, 5, 4, 6, 7, 5};
    Solution sol;
    vector<int> result =
    sol.findMissingRepeatingNumbers(nums);
```

```

    cout << "The repeating and missing numbers are:
    {" << result[0] << ", " << result[1] << "}\n";

    return 0;
}

```

Time Complexity: $O(N)$, where N is the size of the array. This is because we are iterating through the array to calculate the sums and sums of squares.

Space Complexity: $O(1)$, as we are using a constant amount of space for variables, regardless of the input size.

Optimal Approach 2

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int>
    findMissingRepeatingNumbers(vector<int>&
    nums) {
        int n = nums.size();

        int xr = 0;

        for (int i = 0; i < n; i++) {
            xr = xr ^ nums[i];

            // XOR of numbers from 1 to n

            xr = xr ^ (i + 1);
        }

        int number = (xr & ~(xr - 1));

        int zero = 0;

        int one = 0;

        for (int i = 0; i < n; i++) {
            if ((nums[i] & number) != 0) {
                one = one ^ nums[i];
            } else {
                zero = zero ^ nums[i];
            }
        }

        for (int i = 1; i <= n; i++) {
            if ((i & number) != 0) {
                one = one ^ i;
            }
        }
    }
};

```

```

    } else {
        zero = zero ^ i;
    }
}

int cnt = 0;

for (int i = 0; i < n; i++) {
    if (nums[i] == zero) {
        cnt++;
    }
}

if (cnt == 2) {
    return {zero, one};
}

return {one, zero};
}

};

int main() {
    vector<int> nums = {3, 1, 2, 5, 4, 6, 7, 5};

    Solution sol;

    vector<int> result =
    sol.findMissingRepeatingNumbers(nums);

    cout << "The repeating and missing numbers are:
    {" << result[0] << ", " << result[1] << "}\n";

    return 0;
}

```

Time Complexity: $O(N)$, where N is the size of the array. This is because we are iterating through the array to calculate the XOR values.

Space Complexity: $O(1)$, as we are using a constant amount of space for variables, regardless of the input size.

Count inversions in an array

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

int numberOfInversions(vector<int>& a, int n) {
    int cnt = 0; // Initialize inversion count

    for (int i = 0; i < n; i++) {

```

```

        for (int j = i + 1; j < n; j++) {
            if (a[i] > a[j]) cnt++;
        }
    }
    return cnt;
}

int main() {
    vector<int> a = {5, 4, 3, 2, 1};
    int n = a.size();
    int cnt = numberOfInversions(a, n);

    cout << "The number of inversions is: " << cnt
    << endl;

    return 0;
}

```

Time Complexity: $O(N^2)$, as every pair is checked.

Space Complexity: $O(1)$, since no extra space is used apart from variables.

Optimal Approach

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
int merge(vector<int> &arr, int low, int mid, int
high) {
```

```
    vector<int> temp;
```

```
    int left = low;
```

```
    int right = mid + 1;
```

```
    int cnt = 0;
```

```
    while (left <= mid && right <= high) {
```

```
        if (arr[left] <= arr[right]) {
```

```
            temp.push_back(arr[left]);
```

```
            left++;
```

```
        } else {
```

```
            temp.push_back(arr[right]);
```

```
            cnt += (mid - left + 1);
```

```
            right++;
```

```
        }
```

```
    }
```

```
    while (left <= mid) {
```

```
        temp.push_back(arr[left]);
```

```
        left++;
```

```
    }
```

```
    while (right <= high) {
```

```
        temp.push_back(arr[right]);
```

```
        right++;
```

```
    }
```

```
    for (int i = low; i <= high; i++) {
```

```
        arr[i] = temp[i - low];
```

```
    }
```

```
    return cnt;
```

```
}
```

```
int mergeSort(vector<int> &arr, int low, int high) {
```

```
    int cnt = 0;
```

```
    if (low >= high) return cnt;
```

```
    int mid = (low + high) / 2;
```

```
    cnt += mergeSort(arr, low, mid);
```

```
    cnt += mergeSort(arr, mid + 1, high);
```

```
    cnt += merge(arr, low, mid, high);
```

```
    return cnt;
```

```
}
```

```
int numberOfInversions(vector<int> &a, int n) {
```

```
    return mergeSort(a, 0, n - 1);
```

```
}
```

```
int main() {
```

```
    vector<int> a = {5, 4, 3, 2, 1};
```

```
    int n = a.size();
```

```
    int cnt = numberOfInversions(a, n);
```

```
    cout << "The number of inversions are: " << cnt
    << endl;
```

```
    return 0;
```

```
}
```

Time Complexity: $O(N \log N)$, since it is based on merge sort.

Space Complexity: $O(N)$, for the temporary array used during merging.

Count Reverse Pairs

Brute Force Approach

```
#include <bits/stdc++.h>

using namespace std;

int countPairs(vector<int>&a, int n) {
    int cnt = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (a[i] > 2 * a[j]) cnt++;
        }
    }
    return cnt;
}

int team(vector<int> & skill, int n) {
    return countPairs(skill, n);
}

int main(){
    vector<int> a = {4, 1, 2, 3, 1};
    int n = 5;
    int cnt = team(a, n);
    cout << "The number of reverse pair is: "
         << cnt << endl;
    return 0;
}
```

Time Complexity: $O(N^2)$, We are using nested loops here and those two loops roughly run for N times.

Space Complexity: $O(1)$, as we are not using any extra space to solve this problem.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

void merge(vector<int> &arr, int low, int mid, int high) {
    vector<int> temp;

    int left = low;
    int right = mid + 1;

    while (left <= mid && right <= high) {
        if (arr[left] <= arr[right]) {
```

```
            temp.push_back(arr[left]);
            left++;
        }
        else {
            temp.push_back(arr[right]);
            right++;
        }
    }
    while (left <= mid) {
        temp.push_back(arr[left]);
        left++;
    }
    while (right <= high) {
        temp.push_back(arr[right]);
        right++;
    }
    for (int i = low; i <= high; i++) {
        arr[i] = temp[i - low];
    }
}

int countPairs(vector<int> &arr, int low, int mid, int high) {
    int right = mid + 1;
    int cnt = 0;
    for (int i = low; i <= mid; i++) {
        while (right <= high && arr[i] > 2 * arr[right])
            right++;
        cnt += (right - (mid + 1));
    }
    return cnt;
}

int mergeSort(vector<int> &arr, int low, int high) {
    int cnt = 0;
    if (low >= high) return cnt;
    int mid = (low + high) / 2;
    cnt += mergeSort(arr, low, mid);
    cnt += mergeSort(arr, mid + 1, high);
```

```

        cnt += countPairs(arr, low, mid, high);
        merge(arr, low, mid, high);
        return cnt;
    }
}

int team(vector<int> & skill, int n)
{
    return mergeSort(skill, 0, n - 1);
}

int main(){
    vector<int> a = {4, 1, 2, 3, 1};
    int n = 5;
    int cnt = team(a, n);

    cout << "The number of reverse pair is: "
        << cnt << endl;

    return 0;
}

```

Time Complexity: $O(2N \cdot \log N)$, Inside the mergeSort() we call merge() and countPairs() except mergeSort() itself. Now, inside the function countPairs(), though we are running a nested loop, we are actually iterating the left half once and the right half once in total. That is why, the time complexity is $O(N)$. And the merge() function also takes $O(N)$. The mergeSort() takes $O(\log N)$ time complexity. Therefore, the overall time complexity will be $O(\log N * (N+N)) = O(2N \cdot \log N)$.

Space Complexity: $O(N)$, as in the merge sort We use a temporary array to store elements in sorted order.

Maximum Product Subarray in an Array

Brute-Force Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int maxProd = nums[0];
        for (int i = 0; i < nums.size(); i++) {
            int prod = 1;
            for (int j = i; j < nums.size(); j++) {

```

```

                prod *= nums[j];
                maxProd = max(maxProd, prod);
            }
        }

        return maxProd;
    }
};

int main() {
    vector<int> nums = {2, 3, -2, 4};
    Solution sol;
    cout << sol.maxProduct(nums);
    return 0;
}

```

Time Complexity: $O(N^2)$, we check the product of all possible subarrays using two nested loops.

Space Complexity: $O(1)$, No extra space is used.

Optimal Approach – 1

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int maxProductSubArray(vector<int>& arr) {
        int n = arr.size();
        int pre = 1, suff = 1;
        int ans = INT_MIN;
        for (int i = 0; i < n; i++) {
            if (pre == 0) pre = 1;
            if (suff == 0) suff = 1;
            pre *= arr[i];
            suff *= arr[n - i - 1];
            ans = max(ans, max(pre, suff));
        }
        return ans;
    }
};

int main() {

```

```
vector<int> arr = {2, 3, -2, 4};
```

```
Solution obj;
```

```
cout << obj.maxProductSubArray(arr) << endl;
```

```
return 0;
```

```
}
```

Time Complexity: $O(N)$, every element of array is visited once.

Space Complexity: $O(1)$, constant number of variables are used.

Optimal Approach – 2

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Solution {
```

```
public:
```

```
int maxProduct(vector<int>& nums) {
```

```
    int res = nums[0];
```

```
    int maxProd = nums[0];
```

```
    int minProd = nums[0];
```

```
    for (int i = 1; i < nums.size(); i++) {
```

```
        int curr = nums[i];
```

```
        if (curr < 0) swap(maxProd, minProd);
```

```
        maxProd = max(curr, maxProd * curr);
```

```
        minProd = min(curr, minProd * curr);
```

```
        res = max(res, maxProd);
```

```
    }
```

```
    return res;
```

```
}
```

```
};
```

```
int main() {
```

```
    vector<int> nums = {2, 3, -2, 4};
```

```
    Solution sol;
```

```
    cout << sol.maxProduct(nums);
```

```
    return 0;
```

```
}
```

Time Complexity: $O(N)$, every element of array is visited once.

Space Complexity: $O(1)$, only constant variables are used.

Binary Search: Explained

Iterative Implementation

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Solution {
```

```
public:
```

```
int binarySearch(vector& nums, int target) {
```

```
    int n = nums.size(); // size of the array
```

```
    int low = 0, high = n - 1;
```

```
    while (low <= high) {
```

```
        int mid = (low + high) / 2;
```

```
        if (nums[mid] == target) return mid;
```

```
        else if (target > nums[mid]) low = mid + 1;
```

```
        else high = mid - 1;
```

```
    }
```

```
    return -1;
```

```
}
```

```
};
```

```
int main()
```

```
{
```

```
    vector a = {3, 4, 6, 7, 9, 12, 16, 17};
```

```
    int target = 6;
```

```
    Solution obj;
```

```
    int ind = obj.binarySearch(a, target);
```

```
    if (ind == -1) cout << "The target is not present." << endl;
```

```
    else cout << "The target is at index: " << ind << endl;
```

```
    return 0;
```

```
}
```

Time Complexity: In the algorithm, in every step, we are basically dividing the search space into 2 equal halves. This is actually equivalent to dividing the size of the array by 2, every time. After a certain number of divisions, the size will reduce to such an extent that we will not be able to divide that anymore and the process will stop. The number of total divisions will be equal to the time complexity. So the overall time complexity is $O(\log N)$, where N = size of the given array.

Space Complexity: $O(1)$, no extra space being used

Recursive Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int binarySearch(vector& nums, int low, int high,
int target) {
        if (low > high) return -1;
        int mid = (low + high) / 2;
        if (nums[mid] == target) return mid;
        else if (target > nums[mid])
            return binarySearch(nums, mid + 1, high,
target);
        return binarySearch(nums, low, mid - 1,
target);
    }
    int search(vector& nums, int target) {
        return binarySearch(nums, 0, nums.size() - 1,
target);
    }
};

int main()
{
    vector a = {3, 4, 6, 7, 9, 12, 16, 17};
    int target = 6;
    Solution obj;
    int ind = obj.search(a, target);

    if (ind == -1) cout << "The target is not present."
<< endl;

    else cout << "The target is at index: " << ind <<
endl;

    return 0;
}
```

Time Complexity: In the algorithm, in every step, we are basically dividing the search space into 2 equal halves. This is actually equivalent to dividing the size of the array by 2, every time. After a certain number of divisions, the size will reduce to such an extent that we will not be able to divide that anymore and the process will stop. The number of total divisions will be equal to

the time complexity. So the overall time complexity is $O(\log N)$, where N = size of the given array.

Space Complexity: $O(1)$, no extra space being used

Implement Lower Bound

Brute Force Approach

```
#include <bits/stdc++.h>

using namespace std;

class LowerBoundFinder {
public:
    int lowerBound(vector<int> arr, int n, int x) {
        for (int i = 0; i < n; i++) {
            if (arr[i] >= x) {
                return i;
            }
        }
        return n;
    }
};

int main() {
    vector<int> arr = {3, 5, 8, 15, 19};
    int n = arr.size();
    int x = 9;
    LowerBoundFinder finder;
    int ind = finder.lowerBound(arr, n, x);

    cout << "The lower bound is the index: " << ind
<< "\n";

    return 0;
}
```

Time Complexity: $O(N)$, where N = size of the given array.

Space Complexity: $O(1)$, no extra space used.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class LowerBoundFinder {
public:
```

```

int lowerBound(vector<int> arr, int n, int x) {
    int low = 0;
    int high = n - 1;
    int ans = n;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] >= x) {
            ans = mid;
            high = mid - 1;
        } else {
            low = mid + 1;
        }
    }
    return ans;
}

int main() {
    vector<int> arr = {3, 5, 8, 15, 19};
    int n = arr.size();
    int x = 9;

    LowerBoundFinder finder;
    int ind = finder.lowerBound(arr, n, x);

    cout << "The lower bound is the index: " << ind
    << "\n";

    return 0;
}

```

Time Complexity: $O(\log n)$, used for typical binary search

Space Complexity: $O(1)$, no extra space used

Implement Upper Bound

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class UpperBoundFinder {
public:
    int upperBound(vector<int> &arr, int x, int n) {

```

```

        for (int i = 0; i < n; i++) {
            if (arr[i] > x) {
                return i;
            }
        }
        return n;
    }
};

int main() {
    vector<int> arr = {3, 5, 8, 9, 15, 19};
    int n = arr.size();
    int x = 9;

    UpperBoundFinder finder;
    int ind = finder.upperBound(arr, x, n);

    cout << "The upper bound is the index: " << ind
    << "\n";

    return 0;
}

```

Time Complexity: $O(N)$, where N = size of the given array.

Space Complexity: $O(1)$, no extra space used.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class UpperBoundFinder {
public:
    int upperBound(vector<int> &arr, int x, int n) {
        int low = 0, high = n - 1;
        int ans = n;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (arr[mid] > x) {
                ans = mid;
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }
    }
}

```

```

    }
    return ans; // Index of the first element > x
}
};

int main() {
    vector<int> arr = {3, 5, 8, 9, 15, 19};
    int n = arr.size();
    int x = 9;
    UpperBoundFinder finder;
    int ind = finder.upperBound(arr, x, n);
    cout << "The upper bound is the index: " << ind
    << "\n";
    return 0;
}

```

Time Complexity: $O(\log n)$, used for typical binary search

Space Complexity: $O(1)$, no extra space used

Search Insert Position

```

int searchInsert (vector<int>& arr, int x){
    int n = arr.size();
    int low = 0, high = n - 1;
    int ans = n;
    while(low <= high) {
        int mid = (low + high) / 2;
        if(arr[mid] >= x) {
            ans = mid;
            high = mid - 1;
        }
        else {
            low = mid + 1;
        }
    }
    return ans;
}

```

Time Complexity: $O(\log N)$, where N = size of the given array.

Space Complexity: $O(1)$ as we are using no extra space.

Floor and Ceil in Sorted Array

```

#include <bits/stdc++.h>
using namespace std;
class FloorCeilFinder {
public:
    int findFloor(int arr[], int n, int x) {
        int low = 0, high = n - 1;
        int ans = -1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (arr[mid] <= x) {
                ans = arr[mid];
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return ans;
    }
    int findCeil(int arr[], int n, int x) {
        int low = 0, high = n - 1;
        int ans = -1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (arr[mid] >= x) {
                ans = arr[mid];
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }
        return ans;
    }
}

```

```

pair<int, int> getFloorAndCeil(int arr[], int n, int
x) {
    int f = findFloor(arr, n, x);
    int c = findCeil(arr, n, x);
    return make_pair(f, c);
}
};

int main() {
    int arr[] = {3, 4, 4, 7, 8, 10};
    int n = 6, x = 5;
    FloorCeilFinder finder;
    pair<int, int> ans = finder.getFloorAndCeil(arr,
n, x);

    cout << "The floor and ceil are: " << ans.first <<
" " << ans.second << endl;

    return 0;
}

```

Time Complexity: $O(\log N)$, where N = size of the given array. We are using the Binary Search algorithm

Space Complexity: $O(1)$. No extra space used

Last occurrence in a sorted array

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

int solve(int n, int key, vector<int>& v) {
    int res = -1;
    for (int i = n - 1; i >= 0; i--) {
        if (v[i] == key) {
            res = i;
            break;
        }
    }
    return res;
}

int main() {
    int n = 7;

```

```

int key = 13;

vector<int> v = {3, 4, 13, 13, 13, 20, 40};

cout << solve(n, key, v) << "\n";

return 0;
}

```

Time Complexity: $O(N)$, where N is the size of the array. This is because we are traversing the array once to find the last occurrence of the target element.

Space Complexity: $O(1)$, as we are using a constant amount of space for the result variable and the loop index. We are not using any additional data structures that grow with the input size.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

int solve(int n, int key, vector<int>& v) {
    int start = 0;
    int end = n - 1;
    int res = -1;
    while (start <= end) {
        int mid = start + (end - start) / 2;
        if (v[mid] == key) {
            res = mid;
            start = mid + 1;
        }
        else if (key < v[mid]) {
            end = mid - 1;
        }
        else {
            start = mid + 1;
        }
    }
    return res;
}

int main() {
    int n = 7;
    int key = 13;
    vector<int> v = {3, 4, 13, 13, 13, 20, 40};

```

```

    cout << solve(n, key, v) << "\n";

    return 0;
}

```

Time Complexity: $O(\log N)$, where N is the size of the array. This is because we are using binary search, which reduces the search space by half in each iteration.
 Space Complexity: $O(1)$, as we are using a constant amount of space for the result variable and the loop indices. We are not using any additional data structures that grow with the input size.

Count Occurrences in Sorted Array

Brute Force Approach

```

#include<bits/stdc++.h>

using namespace std;

int count(vector<int>& arr, int n, int x) {
    int cnt = 0;
    for (int i = 0; i < n; i++) {
        if (arr[i] == x) cnt++;
    }
    return cnt;
}

int main(){
    vector<int> arr = {2, 4, 6, 8, 8, 8, 11, 13};
    int n = 8, x = 8;
    int ans = count(arr, n, x);
    cout << "The number of occurrences is: "
        << ans << "\n";
    return 0;
}

```

Time Complexity: $O(N)$, We are traversing the whole array.

Space Complexity: $O(1)$, as we are not using any extra space.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

int firstOccurrence(vector<int> &arr, int n, int k) {

```

```

    int low = 0, high = n - 1;
    int first = -1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == k) {
            first = mid;
            high = mid - 1;
        }
        else if (arr[mid] < k) {
            low = mid + 1;
        }
        else {
            high = mid - 1; // look on the left
        }
    }
    return first;
}

int lastOccurrence(vector<int> &arr, int n, int k) {
    int low = 0, high = n - 1;
    int last = -1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == k) {
            last = mid;
            low = mid + 1;
        }
        else if (arr[mid] < k) {
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }
    return last;
}

```

```

pair<int, int> firstAndLastPosition(vector<int>&
arr, int n, int k) {
    int first = firstOccurrence(arr, n, k);
    if (first == -1) return { -1, -1};
    int last = lastOccurrence(arr, n, k);
    return {first, last};
}

int count(vector<int>& arr, int n, int x) {
    pair<int, int> ans = firstAndLastPosition(arr, n,
x);
    if (ans.first == -1) return 0;
    return (ans.second - ans.first + 1);
}

int main(){
    vector<int> arr = {2, 4, 6, 8, 8, 8, 11, 13};
    int n = 8, x = 8;
    int ans = count(arr, n, x);
    cout << "The number of occurrences is: "
        << ans << "\n";
    return 0;
}

```

Time Complexity: $O(2 \cdot \log N)$, We are basically using the binary search algorithm twice.

Space Complexity: $O(1)$, as we are using no extra space.

Search Element in a Rotated Sorted Array

Brute Force Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int search(vector<int>& nums, int target) {
        for (int i = 0; i < nums.size(); i++) {
            if (nums[i] == target) {
                return i;
            }
        }
    }
}

```

```

    }
    return -1;
}
};

int main() {
    vector<int> nums = {4, 5, 6, 7, 0, 1, 2};
    int target = 0;
    Solution obj;
    int index = obj.search(nums, target);
    cout << index << endl;
    return 0;
}

```

Time Complexity: $O(N)$, We may need to check every element in the worst case if the target is not present.

Space Complexity: $O(1)$, No extra space is used; only constant variables.

Optimal Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int low = 0;
        int high = nums.size() - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (nums[mid] == target)
                return mid;
            if (nums[low] <= nums[mid]) {
                if (nums[low] <= target && target <
nums[mid]) {
                    high = mid - 1;
                }
                else {
                    low = mid + 1;
                }
            }
        }
    }
}

```

```

        else {
            if (nums[mid] < target && target <=
nums[high]) {
                low = mid + 1;
            }
            else {
                high = mid - 1;
            }
        }
    }
    return -1;
}
};

```

```

int main() {
    vector<int> nums = {4,5,6,7,0,1,2};
    int target = 0;

    Solution obj;

    int result = obj.search(nums, target);

    cout << result << endl;

    return 0;
}

```

Time Complexity: $O(\log N)$, We eliminate half of the search space in each iteration using binary search.

Space Complexity: $O(1)$, We use only a few variables (low, high, mid) no extra space used.

Search Element in Rotated Sorted Array II

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool
searchInARotatedSortedArrayII(vector<int>& arr,
int k) {
        int n = arr.size(); // size of the array

```

```

        for (int i = 0; i < n; i++) {
            if (arr[i] == k) return true;
        }
        return false;
    }
};

int main() {
    vector<int> arr = {7, 8, 1, 2, 3, 3, 3, 4, 5, 6};
    int k = 3;

    Solution obj;

    bool ans =
obj.searchInARotatedSortedArrayII(arr, k);

    if (!ans)
        cout << "Target is not present.\n";
    else
        cout << "Target is present in the array.\n";

    return 0;
}

```

Time Complexity: $O(N)$, N = size of the given array.

Space Complexity: $O(1)$, no extra space used.

Optimised Approach

```

#include <bits/stdc++.h>

using namespace std;

bool searchInARotatedSortedArrayII(vector<int>& arr, int k) {
    int n = arr.size();

    int low = 0, high = n - 1;

    while (low <= high) {
        int mid = (low + high) / 2;

        if (arr[mid] == k) return true;

        if (arr[low] == arr[mid] && arr[mid] ==
arr[high]) {
            low++;
            high--;
            continue;
        }
    }
}

```

```

        if (arr[low] <= arr[mid]) {
            if (arr[low] <= k && k <= arr[mid]) {
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        } else {
            if (arr[mid] <= k && k <= arr[high]) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
    }
    return false;
}

int main() {
    vector<int> arr = {7, 8, 1, 2, 3, 3, 3, 4, 5, 6};
    int k = 3;

    bool ans = searchInARotatedSortedArrayII(arr,
k);

    if (ans)
        cout << "Target is present in the array.\n";
    else
        cout << "Target is not present.\n";

    return 0;
}

```

Time Complexity: $O(\log N)$ for the best and average case. $O(N/2)$ for the worst case. Here, N = size of the given array.

Space Complexity: $O(1)$, no extra space used

Minimum in Rotated Sorted Array

Brute-Force Approach

```

#include <bits/stdc++.h>

using namespace std;

```

```

class Solution {
public:
    int findMin(vector<int>& nums) {
        int minVal = INT_MAX;

        for (int i = 0; i < nums.size(); i++) {
            minVal = min(minVal, nums[i]);
        }

        return minVal;
    }
};

int main() {
    vector<int> nums = {4, 5, 6, 7, 0, 1, 2};

    Solution sol;

    int result = sol.findMin(nums);

    cout << "Minimum element is " << result <<
endl;

    return 0;
}

```

Time Complexity: $O(N)$, we check every element once.
Space Complexity: $O(1)$, constant additional space is used.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int findMin(vector<int>& nums) {
        int low = 0, high = nums.size() - 1;

        while (low < high) {
            int mid = low + (high - low) / 2;

            if (nums[mid] > nums[high]) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }

        return nums[low];
    }
};

```

```

    }
};

int main() {
    vector<int> nums = {4, 5, 6, 7, 0, 1, 2};

    Solution sol;

    int result = sol.findMin(nums);

    cout << "Minimum element is " << result << endl;

    return 0;
}

```

Time Complexity: $O(\log N)$, at every step the search space is reduced to half using binary search.
 Space Complexity: $O(1)$, constant additional space is used.

Find out how many times the array has been rotated

Brute force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int findRotations(vector<int>& arr) {
        int n = arr.size();
        int minVal = arr[0];
        int minIndex = 0;
        for (int i = 1; i < n; i++) {
            if (arr[i] < minVal) {
                minVal = arr[i];
                minIndex = i;
            }
        }
        return minIndex;
    }
};

int main() {
    Solution obj;

```

```

    vector<int> arr = {4,5,6,7,0,1,2,3};

    int rotations = obj.findRotations(arr);

    cout << rotations << endl;

    return 0;
}

```

Time Complexity: $O(n)$, We scan the entire array once to find the smallest element, where n is the size of the array.
 Space Complexity: $O(1)$, We only use a few extra variables to store the minimum value and its index, so the extra space used is constant.

Better Approach

```

#include <bits/stdc++.h>

using namespace std;

int findRotationCount(vector<int> &arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        if (arr[i] > arr[i + 1]) {
            return i + 1;
        }
    }
    return 0;
}

int main() {
    vector<int> arr = {3, 4, 5, 1, 2};
    int rotations = findRotationCount(arr);
    cout << rotations << endl;
    return 0;
}

```

Time Complexity: $O(n)$, We traverse the array once to find the rotation point, where n is the size of the array.
 Space Complexity: $O(1)$, Only a few extra variables are used regardless of input size, so constant space.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:

```

```

int findRotations(vector<int>& arr) {
    int low = 0;
    int high = arr.size() - 1;
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] > arr[high]) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }
    return low;
}

int main() {
    Solution sol;
    vector<int> arr = {4,5,6,7,0,1,2,3};
    int rotations = sol.findRotations(arr);
    cout << rotations << endl;
    return 0;
}

```

Time Complexity: $O(\log n)$, The binary search halves the search space each iteration, where n is the size of the array.

Space Complexity: $O(1)$, Only a few variables are used regardless of input size, so constant extra space.

Search Single Element in a sorted array

Brute force Approach 1

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int singleNonDuplicate(vector<int>& arr) {
        int n = arr.size();
        if (n == 1) return arr[0];
    }
}

```

```

for (int i = 0; i < n; i++) {
    if (i == 0) {
        if (arr[i] != arr[i + 1])
            return arr[i];
    }
    else if (i == n - 1) {
        if (arr[i] != arr[i - 1])
            return arr[i];
    }
    else {
        if (arr[i] != arr[i - 1] && arr[i] != arr[i + 1])
            return arr[i];
    }
}

return -1;
}

};

```

```

int main() {
    vector<int> arr = {1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6};
    Solution obj;
    int ans = obj.singleNonDuplicate(arr);
    cout << "The single element is: " << ans << "\n";
    return 0;
}

```

Time Complexity: $O(N)$, N = size of the given array. We are traversing the entire array.

Space Complexity: $O(1)$ as we are not using any extra space.

Brute Approach 2

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int singleNonDuplicate(vector<int>& arr) {
        int n = arr.size();
        int ans = 0;
    }
}

```

```

        for (int i = 0; i < n; i++) {
            ans = ans ^ arr[i];
        }
        return ans;
    }
};

int main() {
    vector<int> arr = {1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6};
    Solution obj;
    int ans = obj.singleNonDuplicate(arr);
    cout << "The single element is: " << ans << "\n";
    return 0;
}

```

Time Complexity: $O(N)$, N = size of the given array. We are traversing the entire array.

Space Complexity: $O(1)$ as we are not using any extra space.

Optimal Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int singleNonDuplicate(vector<int>& arr) {
        int n = arr.size();
        if (n == 1) return arr[0];
        if (arr[0] != arr[1]) return arr[0];
        if (arr[n - 1] != arr[n - 2]) return arr[n - 1];
        int low = 1, high = n - 2;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (arr[mid] != arr[mid + 1] && arr[mid] != arr[mid - 1]) {
                return arr[mid];
            }
            if ((mid % 2 == 1 && arr[mid] == arr[mid - 1]) ||
                (mid % 2 == 0 && arr[mid] == arr[mid + 1])) {

```

```

                // Move to the right half
                low = mid + 1;
            }
            else {
                high = mid - 1;
            }
        }
        return -1;
    }
};

int main() {
    vector<int> arr = {1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6};
    Solution obj;
    int ans = obj.singleNonDuplicate(arr);
    cout << "The single element is: " << ans << "\n";
    return 0;
}

```

Time Complexity: $O(\log N)$, N = size of the given array, as we are basically using the Binary Search algorithm.

Space Complexity: $O(1)$ as we are not using any extra space.

Peak element in Array

Brute Force Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        int n = nums.size();
        for (int i = 0; i < n; i++) {
            bool left = (i == 0) || (nums[i] >= nums[i - 1]);
            bool right = (i == n - 1) || (nums[i] >= nums[i + 1]);
            if (left && right) return i;
        }
    }
}

```

```

        return -1;
    }
};

int main() {
    Solution sol;

    vector<int> nums = {1, 3, 20, 4, 1, 0};

    int index = sol.findPeakElement(nums);

    cout << "Peak at index: " << index << " with
value: " << nums[index] << endl;

    return 0;
}

```

Time Complexity: $O(N)$, we traverse the entire array once to find peak element.
Space Complexity: $O(1)$, constant additional space is used.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        int n = nums.size();

        for (int i = 0; i < n; i++) {
            bool left = (i == 0) || (nums[i] >= nums[i - 1]);

            bool right = (i == n - 1) || (nums[i] >= nums[i + 1]);

            if (left && right) return i;
        }

        return -1;
    }
};

int main() {
    Solution sol;

    vector<int> nums = {1, 3, 20, 4, 1, 0};

    int index = sol.findPeakElement(nums);

    cout << "Peak at index: " << index << " with
value: " << nums[index] << endl;

    return 0;
}

```

```

}

```

Time Complexity: $O(N)$, we traverse the entire array once to find peak element.

Space Complexity: $O(1)$, constant additional space is used.

Finding Sqrt of a number using Binary Search

Brute-Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int floorSqrt(int n) {
        int ans = 0;

        for (int i = 1; i <= n; i++) {
            if ((long long)i * i <= n) {
                ans = i;
            } else {
                break;
            }
        }

        return ans;
    }
};

int main() {
    int n = 27;

    Solution sol;

    cout << sol.floorSqrt(n) << endl;

    return 0;
}

```

Time Complexity: $O(N)$, we check for every number from 1 to N .

Space Complexity: $O(1)$, since the algorithm does not use any additional space or data structures.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

```

```

class Solution {
public:
    int mySqrt(int x) {
        if (x < 2) return x;
        int left = 1, right = x / 2, ans = 0;
        while (left <= right) {
            long long mid = left + (right - left) / 2;
            if (mid * mid <= x) {
                ans = mid;
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return ans;
    }
};
int main() {
    Solution s;
    cout << s.mySqrt(8) << endl;
    return 0;
}

```

Time Complexity: $O(\log(N))$, we apply binary search on our search space to reduce it into half at every step.
 Space Complexity: $O(1)$, since the algorithm does not use any additional space or data structures.

Nth Root of a Number using Binary Search

Brute-Force Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int nthRoot(int n, int m) {
        for (int i = 1; i <= m; i++) {
            long long power = pow(i, n);

```

```

            if (power == m) return i;
            if (power > m) break;
        }
        return -1;
    }
};
int main() {
    Solution sol;
    int n = 3, m = 27;
    cout << "Nth Root: " << sol.nthRoot(n, m) << endl;
    return 0;
}

```

Time Complexity: $O(M)$, we search for every possible number from 1 to M to check if it is the Nth root.
 Space Complexity: $O(1)$, constant additional space is used.

Optimal Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int nthRoot(int n, int m) {
        int low = 1, high = m;
        while (low <= high) {
            int mid = (low + high) / 2;
            long long ans = 1;
            for (int i = 0; i < n; i++) {
                ans *= mid;
                if (ans > m) break;
            }
            if (ans == m) return mid;
            if (ans < m) low = mid + 1;
            else high = mid - 1;
        }
        return -1;
    }
}

```

```
};

int main() {
    Solution obj;

    int result = obj.nthRoot(3, 27);

    return 0;
}
```

Time Complexity: $O(\log M)$, we search for every possible number from 1 to M to check if it is the Nth root.
 Space Complexity: $O(1)$, constant additional space is used.

Koko Eating Bananas

Brute-Force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int calculateTotalHours(vector<int>& a, int hourly) {
        int totalHours = 0;
        for (int pile : a) {
            totalHours += (pile + hourly - 1) / hourly;
        }
        return totalHours;
    }

    int minEatingSpeed(vector<int>& a, int h) {
        int maxVal = *max_element(a.begin(), a.end());
        for (int i = 1; i <= maxVal; i++) {
            int hours = calculateTotalHours(a, i);
            if (hours <= h) {
                return i;
            }
        }
        return maxVal;
    }
};

int main()
```

```
vector<int> a = {3, 6, 7, 11};

int h = 8;

Solution obj;

cout << obj.minEatingSpeed(a, h);

return 0;
}
```

Time Complexity: $O(n * \max(a[i]))$, since for each possible speed we go through all the piles.
 Space Complexity: $O(1)$, since the algorithm does not use any additional space or data structures.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int calculateTotalHours(vector<int>& piles, int speed) {
        int totalH = 0;
        for (int bananas : piles) {
            totalH += ceil((double)bananas / speed);
        }
        return totalH;
    }

    int minEatingSpeed(vector<int>& piles, int h) {
        int maxPile = *max_element(piles.begin(), piles.end());
        int low = 1, high = maxPile;
        int ans = maxPile;
        while (low <= high) {
            int mid = (low + high) / 2;
            int totalH = calculateTotalHours(piles, mid);
            if (totalH <= h) {
                ans = mid;
                high = mid - 1;
            }
            else {
                low = mid + 1;
            }
        }
    }
};
```

```

    }
    return ans;
}
};

int main() {
    vector<int> piles = {3, 6, 7, 11};
    int h = 8;

    Solution obj;
    cout << obj.minEatingSpeed(piles, h);

    return 0;
}

```

Time Complexity: $O(N * \log(\max(a[])))$, we apply binary search on our search space to reduce it into half at every step.

Space Complexity: $O(1)$, since the algorithm does not use any additional space or data structures.

Minimum days to make M bouquets

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class RoseGarden {
public:
    bool isPossible(vector<int>& bloomDays, int day, int m, int k) {
        int count = 0;
        int bouquets = 0;
        for (int bloom : bloomDays) {
            if (bloom <= day) {
                count++;
                if (count == k) {
                    bouquets++;
                    count = 0;
                }
            } else {
                count = 0;
            }
        }
    }
}

```

```

        return bouquets >= m;
    }

    int minDaysToMakeBouquets(vector<int>& bloomDays, int m, int k) {
        long long totalFlowers = 1LL * m * k;
        if (totalFlowers > bloomDays.size()) return -1;
        int low = *min_element(bloomDays.begin(), bloomDays.end());
        int high = *max_element(bloomDays.begin(), bloomDays.end());

        for (int day = low; day <= high; ++day) {
            if (isPossible(bloomDays, day, m, k)) {
                return day;
            }
        }

        return -1;
    }
};

```

```

int main() {
    vector<int> bloomDays = {7, 7, 7, 7, 13, 11, 12, 7};
    int k = 3;
    int m = 2;

    RoseGarden garden;

    int result = garden.minDaysToMakeBouquets(bloomDays, m, k);
    if (result == -1)
        cout << "We cannot make m bouquets.\n";
    else
        cout << "We can make bouquets on day " << result << "\n";

    return 0;
}

```

Time Complexity: $O((\max(arr[]) - \min(arr[])) + 1 * N)$, where $\{\max(arr[])\}$ -> maximum element of the array, $\{\min(arr[])\}$ -> minimum element of the array, N = size of the array.

Space Complexity : $O(1)$ as we are not using any extra space to solve this problem.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool possible(vector<int>& arr, int day, int m, int k) {
        int n = arr.size();
        int cnt = 0;
        int bouquets = 0;
        for (int i = 0; i < n; i++) {
            if (arr[i] <= day) {
                cnt++;
                if (cnt == k) {
                    bouquets++;
                    cnt = 0;
                }
            } else {
                cnt = 0;
            }
        }
        return bouquets >= m;
    }

    int roseGarden(vector<int>& arr, int k, int m) {
        long long total = 1LL * k * m;
        if (total > arr.size()) return -1;
        int mini = *min_element(arr.begin(), arr.end());
        int maxi = *max_element(arr.begin(), arr.end());
        int low = mini, high = maxi;
        int result = -1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (possible(arr, mid, m, k)) {
                result = mid;
                high = mid - 1;
            }
        }
    }
};
```

```
        } else {
            low = mid + 1;
        }
    }

    return result;
}

};

int main() {
    vector<int> arr = {7, 7, 7, 7, 13, 11, 12, 7};
    int k = 3;
    int m = 2;
    Solution sol;
    int ans = sol.roseGarden(arr, k, m);
    if (ans == -1)
        cout << "We cannot make m bouquets.\n";
    else
        cout << "We can make bouquets on day " <<
            ans << "\n";
    return 0;
}
```

Time Complexity: $O(1) O(\log(\max(arr[]) - \min(arr[]) + 1) * N)$, where $\{\max(arr[]) \rightarrow \text{maximum element of the array}, \min(arr[]) \rightarrow \text{minimum element of the array}, N = \text{size of the array}\}$.

Space Complexity : $O(h)O(1)$ as we are not using any extra space to solve this problem.

Find the Smallest Divisor Given a Threshold

Brute Force

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int smallestDivisor(vector<int>& arr, int limit) {
        int n = arr.size();
        int maxi = *max_element(arr.begin(), arr.end());
    }
};
```

```

for (int d = 1; d <= maxi; d++) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += ceil(((double)(arr[i]) /
(double)d));
    }
    if (sum <= limit) {
        return d;
    }
}
return -1;
};

int main() {
    vector<int> arr = {1, 2, 3, 4, 5};
    int limit = 8;
    Solution obj;
    int ans = obj.smallestDivisor(arr, limit);
    cout << "The minimum divisor is: " << ans <<
"\n";
    return 0;
}

```

Time Complexity: $O(\max(\text{arr}) * N)$, where $\max(\text{arr})$ = maximum element in the array, N = size of the array. We are using nested loops. The outer loop runs from 1 to $\max(\text{arr})$ and the inner loop runs for N times.

Space Complexity: $O(1)$. No extra space used

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class SmallestDivisorFinder {
public:
    int sumByD(vector<int>& arr, int div) {
        int sum = 0;
        for (int num : arr) {
            sum += ceil((double)num / div);
        }
        return sum;
    }
};

```

```

}

int smallestDivisor(vector<int>& arr, int limit) {
    if (arr.size() > limit) return -1;
    int low = 1;
    int high = *max_element(arr.begin(),
arr.end());
    while (low <= high) {
        int mid = (low + high) / 2;
        if (sumByD(arr, mid) <= limit) {
            high = mid - 1;
        } else {
            low = mid + 1;
        }
    }
    return low;
};

int main() {
    SmallestDivisorFinder solver;
    vector<int> arr = {1, 2, 3, 4, 5};
    int limit = 8;
    int ans = solver.smallestDivisor(arr, limit);
    cout << "The minimum divisor is: " << ans <<
"\n";
    return 0;
}

```

Time Complexity: $O(\log(\max(\text{arr}) * N))$, where $\max(\text{arr})$ = maximum element in the array, N = size of the array. We are applying binary search on our answers that are in the range of $[1, \max(\text{arr})]$. For every possible divisor 'mid', we call the `sumByD()` function. Inside that function, we are traversing the entire array, which results in $O(N)$.

Space Complexity: $O(1)$, no extra space is used.

Capacity to Ship Packages within D Days

Brute Force Approach

```

#include <bits/stdc++.h>

```

```

using namespace std;

class Solution {
public:
    int daysNeeded(vector<int>& weights, int capacity) {
        int days = 1;
        int currentLoad = 0;
        for (int w : weights) {
            if (currentLoad + w > capacity) {
                days++;
                currentLoad = w;
            } else {
                currentLoad += w;
            }
        }
        return days;
    }

    int shipWithinDays(vector<int>& weights, int d)
    {
        int left = *max_element(weights.begin(),
weights.end());
        int right = accumulate(weights.begin(),
weights.end(), 0);

        for (int capacity = left; capacity <= right;
capacity++) {
            int needed = daysNeeded(weights,
capacity);
            if (needed <= d) {
                return capacity;
            }
        }
        return right;
    }
};

int main() {
    vector<int> weights = {5,4,5,2,3,4,5,6};
    int d = 5;
    Solution sol;

```

```

    cout << sol.shipWithinDays(weights, d) << "\n";

    return 0;
}

```

Time Complexity: $O((\text{sum_weights} - \text{max_weight}) * N)$, where N is the number of packages. For each capacity between max weight and total sum, we simulate shipping over N packages.

Space Complexity: $O(1)$, only constant extra space is used.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int daysNeeded(vector<int>& weights, int capacity) {
        int days = 1;
        int currentLoad = 0;
        for (int w : weights) {
            if (currentLoad + w > capacity) {
                days++;
                currentLoad = w;
            } else {
                currentLoad += w;
            }
        }
        return days;
    }

    int shipWithinDays(vector<int>& weights, int d)
    {
        int left = *max_element(weights.begin(),
weights.end());
        int right = accumulate(weights.begin(),
weights.end(), 0);

        while (left < right) {
            int mid = left + (right - left) / 2;
            int needed = daysNeeded(weights, mid);

```

```

        if (needed <= d) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return left;
}
};

int main() {
    vector<int> weights = {5,4,5,2,3,4,5,6};
    int d = 5;

    Solution sol;

    cout << sol.shipWithinDays(weights, d) << "\n";

    return 0;
}

```

Time Complexity: $O(N * \log(S))$, where N is number of packages and S is the search space (sum_weights - max_weight). Each binary search step takes $O(N)$, repeated $O(\log S)$ times.

Space Complexity: $O(1)$, constant extra space used.

Kth Missing Positive Number

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class MissingKFinder {
public:
    int missingK(vector<int> vec, int n, int k) {
        for (int i = 0; i < n; i++) {
            if (vec[i] <= k) {
                k++;
            } else {
            }
        }
        return k;
    }
}

```

```

};

int main() {
    vector<int> vec = {4, 7, 9, 10};
    int n = vec.size();
    int k = 4;

    MissingKFinder finder;

    int ans = finder.missingK(vec, n, k);

    cout << "The missing number is: " << ans <<
    "\n";

    return 0;
}

```

Time Complexity: $O(N)$, where N = size of the given array.

Space Complexity: $O(1)$, no extra space used.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class MissingKFinder {
public:
    int missingK(vector<int> vec, int n, int k) {
        int low = 0, high = n - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            int missing = vec[mid] - (mid + 1);
            if (missing < k) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return k + high + 1;
    }
};

int main() {
    vector<int> vec = {4, 7, 9, 10};
    int n = vec.size();

```

```

int k = 4;
MissingKFinder finder;
int ans = finder.missingK(vec, n, k);
cout << "The missing number is: " << ans <<
"\n";
return 0;
}

```

Time Complexity: $O(\log n)$, used for typical binary search

Space Complexity: $O(1)$, no extra space used

Aggressive Cows : Detailed Solution

Brute Force Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    bool canPlace(vector<int>& stalls, int cows, int d) {
        int count = 1;
        int lastPos = stalls[0];
        for (int i = 1; i < stalls.size(); i++) {
            if (stalls[i] - lastPos >= d) {
                count++;
                lastPos = stalls[i];
            }
            if (count >= cows) return true;
        }
        return false;
    }
    int aggressiveCows(vector<int>& stalls, int cows) {
        sort(stalls.begin(), stalls.end());
        int n = stalls.size();
        int maxDist = stalls[n - 1] - stalls[0];
        int ans = 0;
        for (int d = 1; d <= maxDist; d++) {

```

```

            if (canPlace(stalls, cows, d)) {
                ans = d;
            }
        }
        return ans;
    }
};
int main() {
    vector<int> stalls = {1, 2, 8, 4, 9};
    int cows = 3;
    Solution obj;
    cout << obj.aggressiveCows(stalls, cows) <<
endl;
    return 0;
}

```

Time Complexity: $O(N \log N) + O(N * (\max(stalls[]) - \min(stalls[])))$, where N = size of the array, $\max(stalls[])$ = maximum element in stalls[] array, $\min(stalls[])$ = minimum element in stalls[] array.

Space Complexity: $O(1)$ as we are not using any extra space to solve this problem.

Optimal Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    bool canPlace(vector<int>& stalls, int cows, int d) {
        int count = 1;
        int lastPos = stalls[0];
        for (int i = 1; i < stalls.size(); i++) {
            if (stalls[i] - lastPos >= d) {
                count++;
                lastPos = stalls[i];
            }
            if (count >= cows) return true;
        }
        return false;
    }
};

```

```

    }

    int aggressiveCows(vector<int>& stalls, int
cows) {
        sort(stalls.begin(), stalls.end());

        int low = 1;
        int high = stalls.back() - stalls.front();
        int ans = 0;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (canPlace(stalls, cows, mid)) {
                ans = mid;
                low = mid + 1;
            }
            else {
                high = mid - 1;
            }
        }
        return ans;
    }
};

int main() {
    vector<int> stalls = {1, 2, 8, 4, 9};
    int cows = 3;
    Solution obj;
    // Print result
    cout << obj.aggressiveCows(stalls, cows) <<
endl;

    return 0;
}

```

Time Complexity: $O(N \log N) + O(N * \log(\max(\text{stalls[]}) - \min(\text{stalls[]})))$, where N = size of the array, $\max(\text{stalls[]})$ = maximum element in stalls[] array, $\min(\text{stalls[]})$ = minimum element in stalls[] array.

Space Complexity: $O(1)$ as we are not using any extra space to solve this problem.

Allocate Minimum Number of Pages

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution{
    int countStudents(vector<int> &arr, int pages) {
        int n = arr.size(); //size of array.
        int students = 1;
        long long pagesStudent = 0;
        for (int i = 0; i < n; i++) {
            if (pagesStudent + arr[i] <= pages) {
                //add pages to current student
                pagesStudent += arr[i];
            }
            else {
                students++;
                pagesStudent = arr[i];
            }
        }
        return students;
    }

    int findPages(vector<int>& arr, int n, int m) {
        if (m > n) return -1;
        int low = *max_element(arr.begin(), arr.end());
        int high = accumulate(arr.begin(), arr.end(), 0);
        for (int pages = low; pages <= high; pages++) {
            if (countStudents(arr, pages) == m) {
                return pages;
            }
        }
        return low;
    }

    int main(){
        vector<int> arr = {25, 46, 28, 49, 24};

        int n = 5;
        int m = 4;
        int ans = findPages(arr, n, m);
    }
}

```

```

    cout << "The answer is: " << ans << "\n";

    return 0;
}

```

Time Complexity: $O(N * (\text{sum}(\text{arr}) - \text{max}(\text{arr}) + 1))$, where N = size of the array, $\text{sum}(\text{arr})$ = sum of all array elements, $\text{max}(\text{arr})$

Space Complexity: $O(1)$ as we are not using any extra space to solve this problem.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution{
int countStudents(vector<int> &arr, int pages) {
    int n = arr.size(); //size of array.

    int students = 1;

    long long pagesStudent = 0;

    for (int i = 0; i < n; i++) {
        if (pagesStudent + arr[i] <= pages) {
            pagesStudent += arr[i];
        }
        else {
            students++;
            pagesStudent = arr[i];
        }
    }

    return students;
}

```

```

int findPages(vector<int>& arr, int n, int m) {
    if (m > n) return -1;

    int low = *max_element(arr.begin(), arr.end());
    int high = accumulate(arr.begin(), arr.end(), 0);

    while (low <= high) {
        int mid = (low + high) / 2;

        int students = countStudents(arr, mid);

        if (students > m) {
            low = mid + 1;
        }
    }
}

```

```

    else {
        high = mid - 1;
    }
}

return low;
}

int main(){
    vector<int> arr = {25, 46, 28, 49, 24};

    int n = 5;

    int m = 4;

    int ans = findPages(arr, n, m);

    cout << "The answer is: " << ans << "\n";

    return 0;
}

```

Time Complexity: $O(N * \log(\text{sum}(\text{arr}) - \text{max}(\text{arr}) + 1))$, where N = size of the array, $\text{sum}(\text{arr})$ = sum of all array elements, $\text{max}(\text{arr})$ = maximum of all array elements.

Space Complexity: $O(1)$ as we are not using any extra space to solve this problem.

Split Array - Largest Sum

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

class SubarrayPartitioner {
public:
    int countPartitions(vector<int> &a, int maxSum)
    {
        int n = a.size();

        int partitions = 1;

        long long subarraySum = 0;

        for (int i = 0; i < n; i++) {
            if (subarraySum + a[i] <= maxSum) {
                subarraySum += a[i];
            }
            else {
                partitions++;
            }
        }
    }
}

```

```

        subarraySum = a[i];
    }
}

return partitions;
}

int largestSubarraySumMinimized(vector<int>
&a, int k) {
    int low = *max_element(a.begin(), a.end());
    int high = accumulate(a.begin(), a.end(), 0);
    for (int maxSum = low; maxSum <= high;
maxSum++) {
        if (countPartitions(a, maxSum) == k)
            return maxSum;
    }
    return low;
}
};

int main() {
    vector<int> a = {10, 20, 30, 40};
    int k = 2;

    SubarrayPartitioner sp;

    int ans = sp.largestSubarraySumMinimized(a, k);
    cout << "The answer is: " << ans << "\n";

    return 0;
}

```

Time Complexity: $O(N * (\text{sum}(\text{arr}) - \text{max}(\text{arr}) + 1))$, where N = size of the array, $\text{sum}(\text{arr})$ = sum of all array elements, $\text{max}(\text{arr})$ = maximum of all array elements.

Space Complexity: $O(1)$, no extra space used.

Optimised Approach

Time Complexity: $O(N * \log(\text{sum}(\text{arr}) - \text{max}(\text{arr}) + 1))$, where N = size of the array, $\text{sum}(\text{arr})$ = sum of all array elements, $\text{max}(\text{arr})$ = maximum of all array elements.

Space Complexity: $O(1)$, no extra space used

Painter's Partition Problem

Brute Force Approach

```
#include <bits/stdc++.h>
```

```

using namespace std;

class PainterPartition {
public:
    int countPainters(vector<int> &boards, int time)
    {
        int n = boards.size();
        int painters = 1;
        long long boardsPainter = 0;
        for (int i = 0; i < n; i++) {
            if (boardsPainter + boards[i] <= time) {
                boardsPainter += boards[i];
            } else {
                painters++;
                boardsPainter = boards[i];
            }
        }
        return painters;
    }

    int findLargestMinDistance(vector<int>
&boards, int k) {
        int low = *max_element(boards.begin(),
boards.end());
        int high = accumulate(boards.begin(),
boards.end(), 0);
        for (int time = low; time <= high; time++) {
            if (countPainters(boards, time) <= k) {
                return time;
            }
        }
        return low;
    }
};

int main() {
    vector<int> boards = {10, 20, 30, 40};
    int k = 2;

    PainterPartition obj;

    int ans = obj.findLargestMinDistance(boards, k);
}

```

```

    cout << "The answer is: " << ans << "\n"
    return 0;
}

```

Time Complexity: $O(N * (\text{sum}(\text{arr}) - \text{max}(\text{arr}) + 1))$, where N = size of the array, $\text{sum}(\text{arr})$ = sum of all array elements, $\text{max}(\text{arr})$ = maximum of all array elements.

Space Complexity: $O(1)$, no extra space used.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class PainterPartition {
public:
    int countPainters(vector<int> &boards, int time)
    {
        int painters = 1;
        long long boardsPainter = 0;
        for (int i = 0; i < boards.size(); i++) {
            if (boardsPainter + boards[i] <= time) {
                boardsPainter += boards[i];
            } else {
                painters++;
                boardsPainter = boards[i];
            }
        }
        return painters;
    }

    int findLargestMinDistance(vector<int>
    &boards, int k) {
        int low = *max_element(boards.begin(),
        boards.end());
        int high = accumulate(boards.begin(),
        boards.end(), 0);
        int result = high;
        while (low <= high) {
            int mid = (low + high) / 2;
            int painters = countPainters(boards, mid);
            if (painters > k) {
                low = mid + 1;

```

```

        } else {
            result = mid;
            high = mid - 1;
        }
    }
    return result;
}
};

int main() {
    vector<int> boards = {10, 20, 30, 40};
    int k = 2;
    PainterPartition pp;
    int ans = pp.findLargestMinDistance(boards, k);
    cout << "The answer is: " << ans << "\n"; //
    Expected: 60
    return 0;
}

```

Time Complexity: $O(N * \log(\text{sum}(\text{arr}) - \text{max}(\text{arr}) + 1))$, where N = size of the array, $\text{sum}(\text{arr})$ = sum of all array elements, $\text{max}(\text{arr})$ = maximum of all array elements.

Space Complexity: $O(1)$ since no extra space is required.

Minimise Maximum Distance between Gas Stations

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

class GasStationSolver {
public:
    long double
    minimiseMaxDistance(vector<int>& arr, int k) {
        int n = arr.size();
        vector<int> howMany(n - 1, 0);
        for (int gasStations = 1; gasStations <= k;
        gasStations++) {
            long double maxSection = -1;
            int maxInd = -1;
            for (int i = 0; i < n - 1; i++) {

```

```

        long double diff = arr[i + 1] - arr[i];
        long double sectionLength = diff /
(howMany[i] + 1.0);
        if (sectionLength > maxSection) {
            maxSection = sectionLength;
            maxInd = i;
        }
    }
    howMany[maxInd]++;
}
long double maxAns = -1;
for (int i = 0; i < n - 1; i++) {
    long double diff = arr[i + 1] - arr[i];
    long double sectionLength = diff /
(howMany[i] + 1.0);
    maxAns = max(maxAns, sectionLength);
}
return maxAns;
}
};

int main() {
    vector<int> arr = {1, 2, 3, 4, 5};
    int k = 4;
    GasStationSolver solver;
    long double ans =
solver.minimiseMaxDistance(arr, k);
    cout << "The answer is: " << ans << "\n";
    return 0;
}

```

Time Complexity: $O(k*n) + O(n)$, n = size of the given array, k = no. of gas stations to be placed.

Space Complexity: $O(n-1)$ as we are using an array to keep track of placed gas stations.

Better Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:

```

```

    long double minimiseMaxDistance(vector<int>
&arr, int k) {
        int n = arr.size();
        vector<int> howMany(n - 1, 0);
        priority_queue<pair<long double, int>> pq;
        for (int i = 0; i < n - 1; i++) {
            long double length = arr[i + 1] - arr[i];
            pq.push({length, i});
        }
        for (int gasStations = 1; gasStations <= k;
gasStations++) {
            auto top = pq.top();
            pq.pop();
            int segmentIndex = top.second;
            howMany[segmentIndex]++;
            long double totalDist = arr[segmentIndex +
1] - arr[segmentIndex];
            long double newLen = totalDist /
(howMany[segmentIndex] + 1);
            pq.push({newLen, segmentIndex});
        }
        return pq.top().first;
    }
};

```

```

int main() {
    vector<int> arr = {1, 2, 3, 4, 5};
    int k = 4;
    Solution obj;
    long double ans = obj.minimiseMaxDistance(arr,
k);
    cout << "The answer is: " << ans << "\n";
    return 0;
}

```

Time Complexity: $O(n \log n + k \log n)$, n = size of the given array, k = no. of gas stations to be placed.

Space Complexity: $O(n-1) + O(n-1)$. The first $O(n-1)$ is for the array to keep track of placed gas stations and the second one is for the priority queue..

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class GasStationOptimizer {
public:
    int numberOfGasStationsRequired(long double
dist, vector<int> &arr) {
        int n = arr.size();
        int cnt = 0;

        for (int i = 1; i < n; i++) {
            int numberInBetween = (arr[i] - arr[i - 1]) /
dist;

            if ((arr[i] - arr[i - 1]) == (dist *
numberInBetween)) {
                numberInBetween--;
            }
            cnt += numberInBetween;
        }
        return cnt;
    }

    long double minimiseMaxDistance(vector<int>
&arr, int k) {
        int n = arr.size();
        long double low = 0, high = 0;
        for (int i = 0; i < n - 1; i++) {
            high = max(high, (long double)(arr[i + 1] -
arr[i]));
        }
        long double diff = 1e-6;
        while (high - low > diff) {
            long double mid = (low + high) / 2.0;
            int cnt =
numberOfGasStationsRequired(mid, arr);
            if (cnt > k) low = mid;
            else high = mid;
        }
        return high;
    }
}

```

```

};

int main() {
    vector<int> arr = {1, 2, 3, 4, 5};
    int k = 4;

    GasStationOptimizer optimizer;

    long double ans =
optimizer.minimiseMaxDistance(arr, k);

    cout << "The answer is: " << ans << "\n";

    return 0;
}

```

Time Complexity: $O(n \cdot \log(\text{Len})) + O(n)$, n = size of the given array, Len = length of the answer space.

Space Complexity: $O(1)$, as we are using no extra space to solve this problem.

Median of Two Sorted Arrays of different sizes

Brute-Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    double findMedianSortedArrays(vector<int>&
nums1, vector<int>& nums2) {
        vector<int> merged;
        int i = 0, j = 0;
        while (i < nums1.size() && j < nums2.size())
        {
            if (nums1[i] < nums2[j]) {
                merged.push_back(nums1[i]);
                i++;
            } else {
                merged.push_back(nums2[j]);
                j++;
            }
        }
        while (i < nums1.size()) {
            merged.push_back(nums1[i]);

```

```

        i++;
    }
    while (j < nums2.size()) {
        merged.push_back(nums2[j]);
        j++;
    }
    int n = merged.size();
    if (n % 2 == 1)
        return merged[n / 2];
    else
        return (merged[n / 2 - 1] + merged[n / 2]) /
2.0;
}
};

int main() {
    Solution sol;
    vector<int> nums1 = {1, 3};
    vector<int> nums2 = {2};
    cout << sol.findMedianSortedArrays(nums1,
nums2) << endl;
    return 0;
}

```

Time Complexity: $O(N_1+N_2)$, every element of both array is visited once.
Space Complexity: $O(N_1+N_2)$, to store the merged array.

Better Approach

```

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

double median(vector<int>& a, vector<int>& b) {
    int n1 = a.size(), n2 = b.size();
    int n = n1 + n2;
    int ind2 = n / 2;
    int ind1 = ind2 - 1;
    int cnt = 0, i = 0, j = 0;
    int ind1el = -1, ind2el = -1;
    while (i < n1 && j < n2) {
        if (a[i] < b[j]) {

```

```

            if (cnt == ind1) ind1el = a[i];
            if (cnt == ind2) ind2el = a[i];
            i++;
        } else {
            if (cnt == ind1) ind1el = b[j];
            if (cnt == ind2) ind2el = b[j];
            j++;
        }
        cnt++;
    }
    while (i < n1) {
        if (cnt == ind1) ind1el = a[i];
        if (cnt == ind2) ind2el = a[i];
        cnt++;
        i++;
    }
    while (j < n2) {
        if (cnt == ind1) ind1el = b[j];
        if (cnt == ind2) ind2el = b[j];
        cnt++;
        j++;
    }
    if (n % 2 == 1) return (double)ind2el;
    return (ind1el + ind2el) / 2.0;
}

int main() {
    vector<int> a = {1, 4, 7, 10, 12};
    vector<int> b = {2, 3, 6, 15};
    cout << fixed << setprecision(1);
    cout << "The median is " << median(a, b) <<
"\n";
}

```

Time Complexity: $O(N_1+N_2)$, every element of both the arrays is visited once.
Space Complexity: $O(1)$, constant extra space is used.

Optimal Approach

```

#include <bits/stdc++.h>

```

```

using namespace std;

class Solution {
public:
    double findMedianSortedArrays(vector<int>& a,
vector<int>& b) {

        if (a.size() > b.size()) return
findMedianSortedArrays(b, a);

        int n1 = a.size(), n2 = b.size();

        int low = 0, high = n1;

        while (low <= high) {
            int cut1 = (low + high) / 2;
            int cut2 = (n1 + n2 + 1) / 2 - cut1
            int l1 = (cut1 == 0) ? INT_MIN : a[cut1 -
1];
            int l2 = (cut2 == 0) ? INT_MIN : b[cut2 -
1];
            int r1 = (cut1 == n1) ? INT_MAX : a[cut1];
            int r2 = (cut2 == n2) ? INT_MAX : b[cut2];
            if (l1 <= r2 && l2 <= r1) {
                if ((n1 + n2) % 2 == 0)
                    return (max(l1, l2) + min(r1, r2)) / 2.0;
                else
                    return max(l1, l2);
            }
            else if (l1 > r2) {
                high = cut1 - 1;
            }
            else {
                low = cut1 + 1;
            }
        }
        return 0.0;
    }
};

int main() {
    Solution sol;
    vector<int> a = {1, 3};

```

```

    vector<int> b = {2};

    cout << "Median is: " <<
sol.findMedianSortedArrays(a, b) << endl;

    return 0;
}

```

Time Complexity: $O(\log(\min(n1, n2)))$, we are applying binary search on the range $[0, \min(n1, n2)]$.
Space Complexity: $O(1)$, only constant variables are used.

K-th Element of two sorted arrays

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int kthElement(vector<int>& a, vector<int>& b,
int k) {
        int m = a.size();
        int n = b.size();
        if (m > n) {
            return kthElement(b, a, k);
        }
        int left = k;
        int low = max(0, k - n), high = min(k, m);
        while (low <= high) {
            int mid1 = (low + high) >> 1;
            int mid2 = left - mid1;
            int l1 = (mid1 > 0) ? a[mid1 - 1] :
INT_MIN;
            int l2 = (mid2 > 0) ? b[mid2 - 1] :
INT_MIN;
            int r1 = (mid1 < m) ? a[mid1] : INT_MAX;
            int r2 = (mid2 < n) ? b[mid2] : INT_MAX;
            if (l1 <= r2 && l2 <= r1) {
                return max(l1, l2);
            }
            else if (l1 > r2) {
                high = mid1 - 1;

```

```

    }
    else {
        low = mid1 + 1;
    }
}
return -1;
}
};

int main() {
    vector<int> a = {2, 3, 6, 7, 9};
    vector<int> b = {1, 4, 8, 10};

    int k = 5;
    Solution solution;

    cout << "The " << k << "-th element of two
sorted arrays is: "

    << solution.kthElement(a, b, k) << '\n';

    return 0;
}

```

Time Complexity: $O(\log(\min(M, N)))$, where M and N are the sizes of the two given arrays. As binary search is being applied on the range $[\max(0, k - N2), \min(k, N1)]$, the range length $\leq \min(M, N)$.
Space Complexity: $O(1)$, as no additional space is used.

Find the row with maximum number of 1's

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int rowWithMax1s(vector<vector<int>>
&matrix, int n, int m) {
        int cnt_max = 0;
        int index = -1;
        for (int i = 0; i < n; i++) {
            int cnt_ones = 0;
            for (int j = 0; j < m; j++) {

```

```

                cnt_ones += matrix[i][j];
            }
            if (cnt_ones > cnt_max) {
                cnt_max = cnt_ones;
                index = i;
            }
        }
        return index;
    }
};

int main() {
    vector<vector<int>> matrix = {{1, 1, 1}, {0, 0,
1}, {0, 0, 0}};

    int n = 3, m = 3;
    Solution obj;

    cout << "The row with maximum no. of 1's is: "
<< obj.rowWithMax1s(matrix, n, m) << '\n';

}

```

Time Complexity: $O(n \times m)$, where n = given row number, m = given column number. We are using nested loops running for n and m times respectively.

Space Complexity: $O(1)$. No extra space used

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int lowerBound(vector<int> &arr, int n, int x) {
        int low = 0, high = n - 1;
        int ans = n;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (arr[mid] >= x) {
                ans = mid;
                high = mid - 1;
            } else {
                low = mid + 1;

```

```

    }
}

return ans;

}

int rowWithMax1s(vector<vector<int>>
&matrix, int n, int m) {
    int cnt_max = 0;
    int index = -1;
    for (int i = 0; i < n; i++) {
        int cnt_ones = m - lowerBound(matrix[i],
m, 1);
        if (cnt_ones > cnt_max) {
            cnt_max = cnt_ones;
            index = i;
        }
    }
    return index;
}

};

int main() {
    vector<vector<int>> matrix = {{1, 1, 1}, {0, 0,
1}, {0, 0, 0}};

    int n = 3, m = 3;

    Solution obj;

    cout << "The row with maximum no. of 1's is: "
<< obj.rowWithMax1s(matrix, n, m) << "\n";
}

```

Time Complexity: $O(n \times \log m)$, where n = given row number, m = given column number. We are using a loop running for n times to traverse the rows. Then we are applying binary search on each row with m columns.

Space Complexity: $O(1)$, no extra space is used.

Search in a sorted 2D matrix

Brute force Approach

```
#include <bits/stdc++.h>
```

```

using namespace std;

class Solution {
public:
    bool searchMatrix(vector<vector<int>>&
matrix, int target) {
        int n = matrix.size();
        int m = matrix[0].size();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (matrix[i][j] == target)
                    return true;
            }
        }
        return false;
    }
};

int main() {
    vector<vector<int>> matrix = {{1, 2, 3, 4},
                                {5, 6, 7, 8},
                                {9, 10, 11, 12}};

    Solution obj;

    if (obj.searchMatrix(matrix, 8))
        cout << "true\n";
    else
        cout << "false\n";

    return 0;
}

```

Time Complexity: $O(n \times m)$, We are traversing the entire matrix with ' n ' rows and ' m ' columns. In the worst case, we may end up visiting every cell once if the target is not present. So, the total number of operations is proportional to the number of elements in the matrix.

Space Complexity: $O(1)$, We are not using any additional space. The algorithm uses a constant amount of extra memory regardless of the size of the matrix just loop variables and the target. Therefore, the space complexity is constant.

Better Approach

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```

class Solution {
public:
    bool binarySearch(vector<int>& nums, int target) {
        int n = nums.size();
        int low = 0, high = n - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (nums[mid] == target)
                return true;
            else if (target > nums[mid])
                low = mid + 1;
            else
                high = mid - 1;
        }
        return false;
    }

    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int n = matrix.size();
        int m = matrix[0].size();
        for (int i = 0; i < n; i++) {
            if (matrix[i][0] <= target && target <= matrix[i][m - 1]) {
                return binarySearch(matrix[i], target);
            }
        }
        return false;
    }
};

int main() {
    vector<vector<int>> matrix = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };
    Solution obj;

```

```

        if (obj.searchMatrix(matrix, 8))
            cout << "true\n";
        else
            cout << "false\n";
        return 0;
    }
}

```

Time Complexity: $O(n \times \log m)$, We go through each of the `n` rows once. For any valid row where the target can exist, we apply binary search which takes $O(\log m)$. So overall time = $O(n \times \log m)$.

Space Complexity: $O(1)$, No extra space is used just a few integer variables for looping and binary search. So space complexity is constant.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int n = matrix.size();
        int m = matrix[0].size();
        int low = 0, high = n * m - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            int row = mid / m;
            int col = mid % m;
            if (matrix[row][col] == target)
                return true;
            else if (matrix[row][col] < target)
                low = mid + 1;
            else
                high = mid - 1;
        }
        return false;
    }
};

int main() {

```

```

vector<vector<int>> matrix = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
Solution obj;
if (obj.searchMatrix(matrix, 8))
    cout << "true\n";
else
    cout << "false\n";
return 0;
}

```

Time Complexity: $O(\log(N \times M))$, where N = given row number, M = given column number. We are applying binary search on the imaginary 1D array of size $N \times M$.

Space Complexity: $O(1)$ as we are not using any extra space.

Search in a row and column-wise sorted matrix

Brute Force

```

#include <bits/stdc++.h>
using namespace std;
class MatrixSearch {
private:
    vector<vector<int>> matrix;
public:
    MatrixSearch(vector<vector<int>>& mat) {
        matrix = mat;
    }
    bool searchElement(int target) {
        int n = matrix.size();
        int m = matrix[0].size();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (matrix[i][j] == target) {
                    return true;
                }
            }
        }
        return false;
    }
}

```

```

    }
    }
    return false;
}
};

int main() {
    vector<vector<int>> matrix = {
        {1, 4, 7, 11, 15},
        {2, 5, 8, 12, 19},
        {3, 6, 9, 16, 22},
        {10, 13, 14, 17, 24},
        {18, 21, 23, 26, 30}
    };
}

```

```

MatrixSearch ms(matrix);
bool found = ms.searchElement(8);
cout << (found ? "true\n" : "false\n");
return 0;
}

```

Time Complexity: $O(N \times M)$, where N = given row number, M = given column number in order to traverse the matrix, we are using nested loops running for n and m times respectively.

Space Complexity: $O(1)$ as we are not using any extra space.

Better Approach

```

#include <bits/stdc++.h>
using namespace std;
class MatrixSearch {
private:
    vector<vector<int>> matrix;
    bool binarySearch(vector<int>& nums, int target) {
        int n = nums.size();
        int low = 0, high = n - 1;
        while (low <= high) {
            int mid = (low + high) / 2;

```

```

        if (nums[mid] == target) {
            return true;
        }

        else if (target > nums[mid]) {
            low = mid + 1;
        }

        else {
            high = mid - 1;
        }
    }

    return false;
}

public:
    MatrixSearch(vector<vector<int>>& mat) {
        matrix = mat;
    }

    bool searchElement(int target) {
        int n = matrix.size();
        for (int i = 0; i < n; i++) {
            if (binarySearch(matrix[i], target)) {
                return true;
            }
        }

        return false;
    }
};

int main() {
    vector<vector<int>> matrix = {
        {1, 4, 7, 11, 15},
        {2, 5, 8, 12, 19},
        {3, 6, 9, 16, 22},
        {10, 13, 14, 17, 24},
        {18, 21, 23, 26, 30}
    };

    MatrixSearch ms(matrix);

    bool found = ms.searchElement(8);

```

```

        cout << (found ? "true\n" : "false\n");
    }

```

Time Complexity: $O(N \cdot \log M)$, where N = given row number, M = given column number. We are traversing all rows and it takes $O(N)$ time complexity. And for all rows, we are applying binary search. So, the total time complexity is $O(N \cdot \log M)$.

Space Complexity: $O(1)$ as we are not using any extra space.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class MatrixSearch {
private:
    vector<vector<int>> matrix; // 2D matrix

public:
    MatrixSearch(vector<vector<int>>& mat) {
        matrix = mat;
    }

    bool searchElement(int target) {
        int n = matrix.size();
        int m = matrix[0].size();
        int row = 0;
        int col = m - 1;

        while (row < n && col >= 0) {
            if (matrix[row][col] == target) {
                return true;
            }

            else if (matrix[row][col] < target) {
                row++;
            }

            else {
                col--;
            }
        }

        return false;
    }
};

```

```
int main() {
    vector<vector<int>> matrix = {
        {1, 4, 7, 11, 15},
        {2, 5, 8, 12, 19},
        {3, 6, 9, 16, 22},
        {10, 13, 14, 17, 24},
        {18, 21, 23, 26, 30}
    };
    MatrixSearch ms(matrix);
    bool found = ms.searchElement(8);
    cout << (found ? "true\n" : "false\n");
}
```

Time Complexity: $O(N+M)$, where N = given row number, M = given column number. We are starting traversal from $(0, M-1)$, and at most, we can end up being in the cell $(M-1, 0)$. So, the total distance can be at most $(N+M)$. So, the time complexity is $O(N+M)$.

Space Complexity: $O(1)$ as we are not using any extra space.

Find Peak Element (2D Matrix)

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

class Solution {
public:
    int maxElement(vector<vector<int>>& arr, int col) {
        int n = arr.size();
        int max_val = INT_MIN;
        int index = -1;
        for (int i = 0; i < n; i++) {
            if (arr[i][col] > max_val) {
                max_val = arr[i][col];
                index = i;
            }
        }
        return index;
    }
}
```

```
vector<int>
findPeakGrid(vector<vector<int>>& arr) {
    int n = arr.size();
    int m = arr[0].size();
    int low = 0;
    int high = m - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        int row = maxElement(arr, mid);
        int left = mid - 1 >= 0 ? arr[row][mid - 1] :
INT_MIN;
        int right = mid + 1 < m ? arr[row][mid + 1]
: INT_MIN;
        if (arr[row][mid] > left && arr[row][mid]
> right) {
            return {row, mid};
        }
        else if (left > arr[row][mid]) {
            high = mid - 1;
        }
        else {
            low = mid + 1;
        }
    }
    return {-1, -1};
};
```

```
int main() {
    vector<vector<int>> mat = {
        {4, 2, 5, 1, 4, 5},
        {2, 9, 3, 2, 3, 2},
        {1, 7, 6, 0, 1, 3},
        {3, 6, 2, 3, 7, 2}
    };
    Solution sol;
    vector<int> peak = sol.findPeakGrid(mat);
    cout << "The row of peak element is " <<
peak[0] << " and column of the peak element is "
```

```

        << peak[1] << endl;

    return 0;

}

```

Time Complexity: $O(N * \log M)$, where N is the number of rows in the matrix, M is the number of columns in each row. The complexity arises because binary search is performed on the columns, and for each mid column, a linear search through the rows is executed to find the maximum element.

Space Complexity: $O(1)$ as no additional space is used.

Median of Row Wise Sorted Matrix

Brute-Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int findMedian(vector<vector<int>>& matrix) {
        vector<int> elements;

        for (auto& row : matrix) {
            for (int val : row) {
                elements.push_back(val);
            }
        }

        sort(elements.begin(), elements.end());

        int n = elements.size();

        return elements[n / 2];
    }
};

int main() {
    Solution obj;

    vector<vector<int>> matrix = {
        {1, 3, 5},
        {2, 6, 9},
        {3, 6, 9}
    };

    cout << obj.findMedian(matrix) << endl;
}

```

```

    return 0;

}

```

Time Complexity: $O(M*N*(\log(M*N)))$, sorting the linear array takes time complexity of $O(M*N(\log M*N))$.
Space Complexity: $O(M*N)$, to create a linear array.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int countLessEqual(vector<int>& row, int mid) {
        return upper_bound(row.begin(), row.end(),
            mid) - row.begin();
    }

    int findMedian(vector<vector<int>>& matrix) {
        int rows = matrix.size();
        int cols = matrix[0].size();

        int low = matrix[0][0];
        int high = matrix[0][cols - 1];

        for (int i = 1; i < rows; i++) {
            low = min(low, matrix[i][0]);
            high = max(high, matrix[i][cols - 1]);
        }

        while (low < high) {
            int mid = (low + high) / 2;
            int count = 0;

            for (int i = 0; i < rows; i++) {
                count += countLessEqual(matrix[i], mid);
            }

            if (count < (rows * cols + 1) / 2)
                low = mid + 1;
            else
                high = mid;
        }

        return low;
    }
};

```

```
int main() {
    Solution obj;

    vector<vector<int>> matrix = {
        {1, 3, 5},
        {2, 6, 9},
        {3, 6, 9}
    };

    cout << "Median: " << obj.findMedian(matrix)
    << endl;

    return 0;
}
```

Time Complexity: $O(\text{rows} \times \log(\text{max} - \text{min}) \times \log(\text{cols}))$, Binary search runs on the value space from min to max of the matrix and for each mid in binary search, we count how many numbers are less than or equal to mid
Space Complexity: $O(1)$, constant extra space is used.

Remove Outermost Parentheses

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    string removeOuterParentheses(string s) {
        string result = "";
        int level = 0;
        for (char ch : s) {
            if (ch == '(') {
                if (level > 0) result += ch;
                level++;
            }
            else if (ch == ')') {
                level--;
                if (level > 0) result += ch;
            }
        }
        return result;
    }
};
```

```
int main() {
    string s = "(())(())";

    Solution sol;

    string ans = sol.removeOuterParentheses(s);

    cout << "The result is: " << ans << endl;

    return 0;
}
```

Time Complexity: $O(n)$, since we are performing a single traversal of the string.

Space Complexity: $O(1)$, since we are using a few variables to track the current state.

Reverse Words in a String

Brute Force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    string reverseWords(string s) {
        vector<string> words;
        string word = "";
        for (int i = 0; i < s.size(); i++) {
            if (s[i] != ' ') {
                word += s[i];
            }
            else if (!word.empty()) {
                words.push_back(word);
                word = "";
            }
        }
        if (!word.empty()) {
            words.push_back(word);
        }
        reverse(words.begin(), words.end());
        string result = "";
        for (int i = 0; i < words.size(); i++) {
            result += words[i];
        }
    }
};
```

```

        if (i < words.size() - 1) {
            result += " ";
        }
    }
    return result;
}
};

int main() {
    Solution obj;
    string s = " amazing coding skills ";
    cout << obj.reverseWords(s) << endl;
    return 0;
}

```

Time Complexity: $O(N)$, We traverse the string once to collect words ($O(N)$) and once more to reverse and join them ($O(N)$). Hence total time is $O(N)$.

Space Complexity: $O(N)$, We store all words in a separate list/array, requiring extra space proportional to the number of characters.

Optimal Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    string reverseWords(string s) {
        string result = "";
        int i = s.size() - 1;
        while (i >= 0) {
            while (i >= 0 && s[i] == ' ') {
                i--;
            }
            if (i < 0) break;
            int end = i;
            while (i >= 0 && s[i] != ' ') {
                i--;
            }
            string word = s.substr(i + 1, end - i);
            if (!result.empty()) {

```

```

                result += " ";
            }
            result += word;
        }
        return result;
    }
};

int main() {
    Solution obj;
    string s = " amazing coding skills ";
    cout << obj.reverseWords(s) << endl;
    return 0;
}

```

Time Complexity: $O(N)$, We traverse the string once from right to left and construct the result directly without extra passes.

Space Complexity: $O(1)$, Ignoring the output string, no additional data structures proportional to input size are used.

Largest Odd Number in a String.

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    string largeOddNum(string& s) {
        int ind = -1;
        int i;
        for (i = s.length() - 1; i >= 0; i--) {
            if ((s[i] - '0') % 2 == 1) {
                ind = i;
                break;
            }
        }
        i = 0;
        while (i <= ind && s[i] == '0') i++;
        return s.substr(i, ind - i + 1);
    }
};

```

```

    }
};

int main() {
    Solution solution;
    string num = "504";
    string result = solution.largeOddNum(num);
    cout << "Largest odd number: " << result << endl;
    return 0;
}

Time Complexity: O(N), since the loop runs once through the string of length N.
Space Complexity: O(1), as we are using only a constant amount of extra space.

```

Longest Common Prefix

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    string longestCommonPrefix(vector<string>& str) {
        if (str.empty()) return "";
        sort(str.begin(), str.end());
        string first = str[0];
        string last = str[str.size() - 1];
        string ans = "";
        int minLength = min(first.size(), last.size());
        for (int i = 0; i < minLength; i++) {
            if (first[i] != last[i]) break;
            ans += first[i];
        }
        return ans;
    }
};

int main() {
    Solution solution;

```

```

        vector<string> input = {"interview", "internet",
                                "internal", "interval"};

        string result =
        solution.longestCommonPrefix(input);

        cout << "Longest Common Prefix: " << result
        << endl;

        return 0;
    }

```

Time Complexity: $O(N * \log N + M)$, where N is the number of strings and M is the minimum length of a string. The sorting operation takes $O(N * \log N)$ time, and the comparison of characters in the first and last strings takes $O(M)$ time.

Space Complexity: $O(M)$, as the ans variable can store the length of the prefix which in the worst case will be $O(M)$.

Isomorphic String

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    bool isomorphicString(string s, string t) {
        int m1[256] = {0}, m2[256] = {0};
        int n = s.size();
        for (int i = 0; i < n; ++i) {
            if (m1[s[i]] != m2[t[i]]) return false;
            m1[s[i]] = i + 1;
            m2[t[i]] = i + 1;
        }
        return true;
    }
};

int main() {
    Solution solution;
    string s = "paper";
    string t = "title";
    if (solution.isomorphicString(s, t)) {
        cout << "Strings are isomorphic." << endl;
    } else {

```

```

        cout << "Strings are not isomorphic." <<
endl;
    }
    return 0;
}

```

Time Complexity: $O(N)$ where N is the length of the input strings, due to the single loop iterating through each character.
 Space Complexity: $O(1)$ since the space used by the arrays is constant (256 fixed size) regardless of input size

Check if one string is rotation of another

Brute Force

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    bool rotateString(string& s, string& goal) {
        if (s.length() != goal.length()) {
            return false;
        }
        for (int i = 0; i < s.length(); i++) {
            string rotated = s.substr(i) + s.substr(0, i);
            if (rotated == goal) {
                return true;
            }
        }
        return false;
    }
};

int main() {
    Solution sol;
    string s = "rotation";
    string goal = "tionrota";
    if (sol.rotateString(s, goal)) {
        cout << "true" << endl;
    } else {

```

```

        cout << "false" << endl;
    }
    return 0;
}

```

Time Complexity: $O(N^2)$ since generating N rotations and each comparison takes $O(N)$ time.

Space Complexity: $O(N)$ for the space needed to store each rotated string.

Optimal Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    bool rotateString(string& s, string& goal) {
        if (s.length() != goal.length()) return false;
        string doubledS = s + s;
        return doubledS.find(goal) != string::npos;
    }
};

int main() {
    Solution sol;
    string s = "rotation";
    string goal = "tionrota";
    cout << (sol.rotateString(s, goal) ? "true" :
    "false") << endl;
    return 0;
}

```

Time Complexity: $O(N)$, because checking for a substring in $s + s$ is linear in time.

Space Complexity: $O(N)$ for the space needed to store the concatenated string $s + s$.

Check if two Strings are anagrams of each other

Brute Force Approach

```

#include <bits/stdc++.h>
using namespace std;
bool CheckAnagrams(string str1, string str2) {
    if (str1.length() != str2.length()) {

```

```

        return false;
    }

    sort(str1.begin(), str1.end());
    sort(str2.begin(), str2.end());
    for (int i = 0; i < str1.length(); i++) {
        if (str1[i] != str2[i]) {
            return false;
        }
    }
    return true;
}

int main() {
    string Str1 = "INTEGER";
    string Str2 = "TEGERNI";
    if (CheckAnagrams(Str1, Str2)) {
        cout << "True" << endl;
    } else {
        cout << "False" << endl;
    }

    return 0;
}

```

Time Complexity: $O(N \log N)$, where N is the length of the strings. This is due to the sorting step performed on both strings.

Space Complexity: $O(1)$, as the sorting is done in-place and no extra space proportional to input size is used (excluding the input strings themselves).

Optimal Approach

```

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

bool CheckAnagrams(string str1, string str2) {
    if (str1.length() != str2.length())
        return false;

    int freq[26] = {0};

    for (int i = 0; i < str1.length(); i++) {
        freq[str1[i] - 'A']++;
    }

    for (int i = 0; i < str2.length(); i++) {

```

```

        freq[str2[i] - 'A']--;
    }

    for (int i = 0; i < 26; i++) {
        if (freq[i] != 0)
            return false;
    }

    return true;
}

int main() {
    string Str1 = "INTEGER";
    string Str2 = "TEGERNI";
    if (CheckAnagrams(Str1, Str2))
        cout << "True" << endl;
    else
        cout << "False" << endl;

    return 0;
}

```

Time Complexity: $O(N)$, where N is the length of the strings. Each string is traversed once, and the frequency array is checked in constant time (26 iterations).

Space Complexity: $O(1)$, as a fixed-size array of 26 elements is used regardless of the input size.

Sort characters by frequency

```

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

class Solution {
private:
    static bool comparator(pair<int, char> p1,
pair<int, char> p2) {
        if (p1.first > p2.first) return true;
        if (p1.first < p2.first) return false;
        return p1.second < p2.second;
    }

public:
    vector<char> frequencySort(string& s) {
        pair<int, char> freq[26];
        for (int i = 0; i < 26; i++) {

```

```

        freq[i] = {0, i + 'a'};
    }
    for (char ch : s) {
        freq[ch - 'a'].first++;
    }
    sort(freq, freq + 26, comparator);
    vector<char> ans;
    for (int i = 0; i < 26; i++) {
        if (freq[i].first > 0)
            ans.push_back(freq[i].second);
    }
    return ans;
}
};

int main() {
    Solution sol;
    string s = "tree";
    vector<char> result = sol.frequencySort(s);
    for (char c : result) {
        cout << c << " ";
    }
    return 0;
}

```

Time Complexity: $O(n + k \log k)$, where n is the length of the string and k is the constant 26 for the alphabet.
 Space Complexity: $O(k)$, where k is the constant 26 for the frequency array.

Maximum Nesting Depth of Parenthesis

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int maxDepth(string s) {
        int p = 0;
        int ans = 0;
        for (char x : s) {

```

```

            if (x == '(') p++;
            else if (x == ')') p--;
            ans = max(ans, p);
        }
        return ans;
    }
};

int main() {
    Solution sol;
    string s = "(1+(2*3)+((8)/4))+1";
    int result = sol.maxDepth(s);
    cout << "Max Depth: " << result << endl;
    return 0;
}

```

Time Complexity: $O(n)$, where n is the length of the string.
 Space Complexity: $O(1)$, as only constant extra space is used.

Roman Numerals to Integer

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int romanToInt(string s) {
        int res = 0;
        unordered_map<char, int> roman = {
            {'I', 1}, {'V', 5}, {'X', 10},
            {'L', 50}, {'C', 100}, {'D', 500}, {'M', 1000}
        };
        for (int i = 0; i < s.size() - 1; i++) {
            if (roman[s[i]] < roman[s[i + 1]]) {
                res -= roman[s[i]];
            } else {
                res += roman[s[i]];
            }
        }
    }
}

```

```

        return res + roman[s.back()];
    }
};

int main() {
    Solution sol;
    string s = "MCMXCIV";
    int result = sol.romanToInt(s);
    cout << "Integer value: " << result << endl;
    return 0;
}

```

Time Complexity: $O(n)$, where n is the length of the input string since we traverse the string once.
 Space Complexity: $O(1)$, since we use a fixed-size map for Roman numerals.

Implement Atoi

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int myAtoi(string s) {
        int i = 0, sign = 1;
        long res = 0;
        while (i < s.size() && s[i] == ' ') i++;
        if (i == s.size()) return 0;
        if (s[i] == '-') {
            sign = -1;
            i++;
        } else if (s[i] == '+') {
            i++;
        }
        while (i < s.size() && isdigit(s[i])) {
            res = res * 10 + (s[i] - '0');
            if (sign * res > INT_MAX) return INT_MAX;
            if (sign * res < INT_MIN) return INT_MIN;
            i++;
        }
    }
};

```

```

    }
    return (int)(sign * res);
}

};

int main() {
    Solution sol;
    string input = " -42";
    int result = sol.myAtoi(input);
    cout << "Converted integer: " << result << endl;

    return 0;
}

```

Time Complexity: $O(N)$, where N is the length of the string.
 Space Complexity: $O(1)$, as constant space is used.

Count Number of Substrings

```

#include <bits/stdc++.h>

using namespace std;

int atMostKDistinct(string s, int k) {
    int left = 0, res = 0;
    unordered_map<char, int> freq;
    for (int right = 0; right < s.size(); right++) {
        freq[s[right]]++;
        while (freq.size() > k) {
            freq[s[left]]--;
            if (freq[s[left]] == 0) freq.erase(s[left]);
            left++;
        }
        res += (right - left + 1);
    }
    return res;
}

int countSubstrings(string s, int k) {
    // Exactly k = atMost(k) - atMost(k-1)
    return atMostKDistinct(s, k) -
        atMostKDistinct(s, k - 1);
}

```

```

}
int main() {
    string s = "pqqs";
    int k = 2;

    cout << "Count: " << countSubstrings(s, k) <<
endl; // Output: 7

    return 0;
}

```

Time Complexity: $O(n)$ for each call to `atMostKDistinct`.
Space Complexity: $O(1)$ map size bounded by 26 characters for alphabets.

Longest Palindromic Substring

```

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

class Solution {
public:
    string longestPalindrome(string str) {
        int start = 0, end = 0;

        for (int center = 0; center < str.length();
center++) {
            int lenOdd = expandFromCenter(str, center,
center);

            int lenEven = expandFromCenter(str,
center, center + 1);

            int maxLen = max(lenOdd, lenEven);

            if (maxLen > end - start) {
                start = center - (maxLen - 1) / 2;
                end = center + maxLen / 2;
            }
        }

        return str.substr(start, end - start + 1);
    }

private:
    int expandFromCenter(const string& str, int left,
int right) {

```

```

        while (left >= 0 && right < str.length() &&
str[left] == str[right]) {
            left--;
            right++;
        }

        return right - left - 1;
    }
};

```

```

int main() {
    Solution sol;

    string input = "babad";

    cout << "Longest Palindromic Substring: " <<
sol.longestPalindrome(input) << endl;

    return 0;
}

```

Time Complexity: $O(N^2)$ For each character, expanding could take up to $O(N)$
Space Complexity: $O(1)$ No extra space used.

Sum of Beauty of all substring

```

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

class Solution {
public:
    int beautySum(string s) {
        int n = s.length();

        int sum = 0;

        for (int i = 0; i < n; i++) {
            unordered_map<char, int> freq;

            for (int j = i; j < n; j++) {
                freq[s[j]]++;

                int maxi = INT_MIN;
                int mini = INT_MAX;

                for (auto it : freq) {
                    mini = min(mini, it.second);
                    maxi = max(maxi, it.second);
                }

```

```

        sum += (maxi - mini);
    }
}
return sum;
}
};

int main() {
    Solution sol;
    string s = "xyx";

    cout << "Beauty Sum: " << sol.beautySum(s) <<
endl;

    return 0;
}

```

Time Complexity:

- Outer loop: $O(n)$ (for each starting index)
- Inner loop: $O(n)$ (for each ending index)
- Computing max and min for frequencies: $O(26)$ in the worst case (since only lowercase letters), $O(n^2 * 26) \approx O(n^2)$ because 26 is constant.

Space Complexity:

- Frequency map uses at most 26 characters $\rightarrow O(26) = O(1)$.
- No extra data structures apart from that.

Reverse Every Word in A String

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    string reverseWords(string s) {
        int n = s.length();
        vector<string> words;

        int start, end;

        int i = 0;
        while(i < n) {
            while(i < n && s[i] == ' ') i++;
            if(i >= n) break;

```

```

            start = i;
            while(i < n && s[i] != ' ') i++;
            end = i-1;

            string wordFound = s.substr(start, end-
start+1);

            words.push_back(wordFound);
        }

        string ans = "";
        for(int i = words.size() - 1; i >= 0; i--) {
            ans += words[i];
            if(i != 0) ans.push_back(' ');
        }

        return ans;
    }
};

```

```

int main(){
    string s = " amazing coding skills ";
    Solution sol;

    string ans = sol.reverseWords(s);

    cout << "Input string: " << s << endl;

    cout << "After reversing every word: " << ans
<< endl;

}

```

Time Complexity: $O(n)$ (where n is the length of the input string). The input string is scanned once to extract words, taking $O(n)$ time. Each word is stored in a list and then concatenated in reverse order, which also takes $O(n)$.

Space Complexity: $O(n)$ since the words list stores each extracted word, requiring $O(k)$ space, where k is the total number of characters in all words (essentially $O(n)$). The result string requires $O(n)$ space as well.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    void reverseString(string &s, int start, int end) {
        while (start < end) {
            swap(s[start++], s[end--]);

```

```

    }
}

public:
    string reverseWords(string s) {
        int n = s.length();
        reverseString(s, 0, n - 1);
        int i = 0, j = 0, start = 0, end = 0;
        while (j < n) {
            while (j < n && s[j] == ' ') j++;
            if (j == n) break;
            start = i;
            while (j < n && s[j] != ' ') {
                s[i++] = s[j++];
            }
            end = i - 1;
            reverseString(s, start, end);
            if (j < n) {
                s[i++] = ' ';
            }
        }
        if (i > 0 && s[i - 1] == ' ') i--;
        return s.substr(0, i);
    }
};

int main(){
    string s = " amazing coding skills ";
    Solution sol;
    string ans = sol.reverseWords(s);
    cout << "Input string: " << s << endl;
    cout << "After reversing every word: " << ans
    << endl;
}

```

Time Complexity: $O(n)$ (where n is the length of the input string).

- The input string is reversed firstly, taking $O(n)$ time.
- The string is then traversed taking another $O(n)$ time.

- Every word encountered in the string is reversed, taking overall $O(n)$ time.

Space Complexity: $O(1)$ since there is no additional space used. The reversal is done in-place taking $O(1)$ or constant space.

Creating a Linked List

```

#include <bits/stdc++.h>
using namespace std;
class Node {
public:
    int data;
    Node* next;
    Node(int data1, Node* next1) {
        data = data1;
        next = next1;
    }
    Node(int data1) {
        data = data1;
        next = nullptr;
    }
};

int main() {
    vector<int> arr = {2, 5, 8, 7};
    Node* y = new Node(arr[0]);
    cout << y << '\n';
    cout << y->data << '\n';
    return 0;
}

```

Insert at the head of a Linked List

```

#include <bits/stdc++.h>
using namespace std;
class Node {
public:
    int data;
    Node* next;
    Node(int data1, Node* next1) {

```

```

        data = data1;
        next = next1;
    }
    Node(int data1) {
        data = data1;
        next = nullptr;
    }
};

class Solution {
public:
    Node* insertAtHead(Node* head, int newData)
    {
        Node* newNode = new Node(newData, head);
        return newNode;
    }

    void printList(Node* head) {
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    Solution sol;
    Node* head = new Node(2);
    head->next = new Node(3);
    cout << "Original List: ";
    sol.printList(head);
    head = sol.insertAtHead(head, 1);
    cout << "After Insertion at Head: ";
    sol.printList(head);
    return 0;
}

```

Time Complexity: O(1), creating a new node and updating the head takes constant time.

Space Complexity: O(1), only one extra node is created to insert at the head of the linked list.

Delete Last Node of Linked List

```

#include <bits/stdc++.h>

using namespace std;

struct Node {
    int data;
    Node* next;
    Node(int val) {
        data = val;
        next = NULL;
    }
};

class Solution {
public:
    Node* deleteTail(Node* head) {
        if (head == NULL || head->next == NULL) {
            delete head;
            return NULL;
        }
        Node* curr = head;
        while (curr->next->next != NULL) {
            curr = curr->next;
        }
        delete curr->next;
        curr->next = NULL;
        return head;
    }
};

int main() {
    Node* head = new Node(1);
    head->next = new Node(2);
    head->next->next = new Node(3);
    Solution obj;
    head = obj.deleteTail(head);
}

```

```

Node* temp = head;
while (temp) {
    cout << temp->data << " ";
    temp = temp->next;
}
return 0;
}

```

Time Complexity: $O(N)$, we traverse the entire linked list once to delete the tail of the list.

Space Complexity: $O(1)$, constant amount of extra space is used.

Find the Length of a Linked List

```

#include <bits/stdc++.h>

using namespace std;

class Node {
public:
    int data;
    Node* next;
    Node(int data1) {
        data = data1;
        next = nullptr;
    }
};

class Solution {
public:
    int lengthOfLinkedList(Node* head) {
        int count = 0;
        Node* temp = head;
        while (temp != nullptr) {
            count++;
            temp = temp->next;
        }
        return count;
    }
};

int main() {

```

```

Node* head = new Node(10);
head->next = new Node(20);
head->next->next = new Node(30);

Solution obj;

cout << "Length of Linked List: "
    << obj.lengthOfLinkedList(head) << endl;

return 0;
}

```

Time Complexity: $O(N)$, we traverse the entire linked list once to find the total number of nodes.

Space Complexity: $O(1)$, we use fixed number of pointers and variables to find the length of linked list.

Search an element in a Linked List

```

#include <bits/stdc++.h>

using namespace std;

struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(NULL) {}
};

class Solution {
public:
    bool searchValue(Node* head, int key) {
        Node* current = head;
        while (current != NULL) {
            if (current->data == key) {
                return true;
            }
            current = current->next;
        }
        return false;
    }
};

int main() {
    Node* head = new Node(10);
    head->next = new Node(20);

```

```

head->next->next = new Node(30);

Solution obj;

if (obj.searchValue(head, 20))

    cout << "Found\n";

else

    cout << "Not Found\n";

return 0;
}

```

Time Complexity: $O(N)$, we traverse the entire linked list once in worst case to search for the required value.
 Space Complexity: $O(1)$, we use a constant amount of additional space, regardless of the linked list's length to search for an element.

Code for Singly Linked List

```

#include <bits/stdc++.h>

using namespace std;

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

    Node(int data1, Node* next1) {
        data = data1;
        next = next1;
    }

    Node(int data1) {
        data = data1;
        next = nullptr;
    }
};

int main() {
    vector<int> arr = {2, 5, 8, 7};
    Node* y = new Node(arr[0]);
    cout << y << '\n';
    cout << y->data << '\n';
    return 0;
}

```

Code for Doubly Linked List

```

#include <bits/stdc++.h>

using namespace std;

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

    Node(int data1, Node* next1, Node* prev1) {
        data = data1;
        next = next1;
        prev = prev1;
    }

    Node(int data1) {
        data = data1;
        next = nullptr;
        prev = nullptr;
    }
};

int main() {
    vector<int> arr = {2, 5, 8, 7};
    Node* head = new Node(arr[0]);
    cout << head << '\n';
    cout << head->data << '\n';
    return 0;
}

```

Insert at end of Doubly Linked List

```

#include <iostream>
#include <bits/stdc++.h>

using namespace std;

class Node {
public:

```

```

int data;
Node* next;
Node* back;

Node(int data1, Node* next1, Node* back1) {
    data = data1;
    next = next1;
    back = back1;
}

Node(int data1) {
    data = data1;
    next = nullptr;
    back = nullptr;
}

Node* convertArr2DLL(vector<int> arr) {
    Node* head = new Node(arr[0]);
    Node* prev = head;
    for (int i = 1; i < arr.size(); i++) {
        Node* temp = new Node(arr[i], nullptr, prev);
        prev->next = temp;
        prev = temp;
    }
    return head;
}

void print(Node* head) {
    while (head != nullptr) {
        cout << head->data << " ";
        head = head->next;
    }
}

Node* insertAtTail(Node* head, int k) {
    Node* newNode = new Node(k);
    if (head == nullptr) {
        return newNode;
    }
    Node* tail = head;

```

```

    while (tail->next != nullptr) {
        tail = tail->next;
    }
    tail->next = newNode;
    newNode->back = tail;
    return head;
}

int main() {
    vector<int> arr = {12, 5, 8, 7, 4};
    Node* head = convertArr2DLL(arr);
    cout << "Doubly Linked List Initially: " << endl;
    print(head);

    cout << endl << "Doubly Linked List After
    Inserting at the tail with value 10: " << endl;

    head = insertAtTail(head, 10);
    print(head);
    return 0;
}

```

Time Complexity: $O(n)$, where n is the number of nodes in the doubly linked list. This is because we traverse the list to find the tail node before inserting the new node.
 Space Complexity: $O(1)$, as we are only using a constant amount of extra space for the new node, regardless of the size of the list.

Delete Last Node of a Doubly Linked List

```

#include <bits/stdc++.h>

using namespace std;

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

    Node(int val) {
        data = val;
        prev = NULL;
        next = NULL;
    }
}

```

```

};

class Solution {
public:
    Node* deleteTail(Node* head) {
        if (head == NULL) return NULL;
        if (head->next == NULL) {
            delete head;
            return NULL;
        }
        Node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->prev->next = NULL;
        delete temp;
        return head;
    }
};

int main() {
    Node* head = new Node(1);
    head->next = new Node(2);
    head->next->prev = head;
    head->next->next = new Node(3);
    head->next->next->prev = head->next;
    Solution obj;
    head = obj.deleteTail(head);
    Node* curr = head;
    while (curr != NULL) {
        cout << curr->data << " ";
        curr = curr->next;
    }

    return 0;
}

```

Time Complexity: $O(N)$, we traverse the entire linked list once to delete the tail of the list.

Space Complexity: $O(1)$, constant amount of extra space is used.

Reverse a Doubly Linked List

Brute Force Approach

```

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

class Node {
public:
    int data;
    Node* next;
    Node* back;

    Node(int data1, Node* next1, Node* back1) {
        data = data1;
        next = next1;
        back = back1;
    }

    Node(int data1) {
        data = data1;
        next = nullptr;
        back = nullptr;
    }
};

Node* convertArr2DLL(vector<int> arr) {
    Node* head = new Node(arr[0]);
    Node* prev = head;
    for (int i = 1; i < arr.size(); i++) {
        Node* temp = new Node(arr[i], nullptr, prev);
        prev->next = temp;
        prev = temp;
    }
    return head;
}

void print(Node* head) {
    while (head != nullptr) {
        cout << head->data << " ";
        head = head->next;
    }
}

```

```

    }
}

Node* reverseDLL(Node* head) {
    if (head == nullptr || head->next == nullptr) {
        return head;
    }
    stack<int> st;
    Node* temp = head;
    while (temp != nullptr) {
        st.push(temp->data);
        temp = temp->next;
    }
    temp = head;
    while (temp != nullptr) {
        temp->data = st.top();
        st.pop();
        temp = temp->next;
    }
    return head;
}

int main() {
    vector<int> arr = {12, 5, 8, 7, 4};
    Node* head = convertArr2DLL(arr);

    cout << endl << "Doubly Linked List Initially: "
    << endl;

    print(head);

    head = reverseDLL(head);

    cout << endl << "Doubly Linked List After
    Reversing: " << endl;

    print(head);

    return 0;
}

```

Time Complexity : $O(2N)$, During the first traversal, each node's value is pushed into the stack once, which requires $O(N)$ time. Then, during the second iteration, the values are popped from the stack and used to update the nodes.

Space Complexity : $O(N)$, This is because we are using an external stack data structure. At the end of the first

iteration, the stack will hold all N values of the doubly linked list therefore the space required for stack is directly proportional to the size of the input doubly linked list.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Node {
public:
    int data;
    Node* next;
    Node* back;

    Node(int data1, Node* next1, Node* back1) {
        data = data1;
        next = next1;
        back = back1;
    }

    Node(int data1) {
        data = data1;
        next = nullptr;
        back = nullptr;
    }
};

Node* convertArr2DLL(vector<int> arr) {
    Node* head = new Node(arr[0]);
    Node* prev = head;

    for (int i = 1; i < arr.size(); i++) {
        Node* temp = new Node(arr[i], nullptr, prev);
        prev->next = temp;
        prev = temp;
    }

    return head;
}

void print(Node* head) {
    while (head != nullptr) {
        cout << head->data << " ";
        head = head->next;
    }
}

```

```

    }
    cout << endl;
}

Node* reverseDLL(Node* head) {
    if (head == nullptr || head->next == nullptr)
        return head;

    Node* curr = head;
    while (curr != nullptr) {
        Node* temp = curr->next;
        curr->next = curr->back;
        curr->back = temp;
        head = curr;
        curr = temp;
    }
    return head;
}

```

```

int main() {
    vector<int> arr = {10, 20, 30, 40};
    Node* head = convertArr2DLL(arr);
    head = reverseDLL(head);
    print(head);
    return 0;
}

```

Time Complexity : $O(N)$ We only have to traverse the doubly linked list once, hence our time complexity is $O(N)$.

Space Complexity : $O(1)$, as the reversal is done in place.

Find middle element in a Linked List

Brute Force Approach

```

#include <iostream>

#include <bits/stdc++.h>

using namespace std;

class Node {
public:
    int data;

```

```

    Node* next;
    Node(int data1, Node* next1) {
        data = data1;
        next = next1;
    }
    Node(int data1) {
        data = data1;
        next = nullptr;
    }
};

Node* findMiddle(Node* head) {
    if (head == NULL || head->next == NULL) {
        return head;
    }
    Node* temp = head;
    int count = 0;
    while (temp != NULL) {
        count++;
        temp = temp->next;
    }
    int mid = count / 2 + 1;
    temp = head;
    while (temp != null) {
        mid = mid - 1;
        if (mid == 0) {
            break;
        }
        temp = temp->next;
    }
    return temp;
}

```

```

int main() {
    Node* head = new Node(1);
    head->next = new Node(2);
    head->next->next = new Node(3);
    head->next->next->next = new Node(4);

```

```

    head->next->next->next->next = new Node(5);

    Node* middleNode = findMiddle(head);

    cout << "The middle node value is: " <<
middleNode->data << endl;

    return 0;
}

```

Time Complexity: $O(N+N/2)$ The code traverses the entire linked list once and half times and then only half in the second iteration, first to count the number of nodes then then again to get to the middle node. Therefore, the time complexity is linear, $O(N + N/2) \sim O(N)$.

Space Complexity : $O(1)$ There is constant space complexity because it uses a constant amount of extra space regardless of the size of the linked list. We only use a few variables to keep track of the middle position and traverse the list, and the memory required for these variables does not depend on the size of the list.

Optimal Approach

```

#include <iostream>

#include <bits/stdc++.h>

using namespace std;

class Node {
public:
    int data;

    Node* next;

    Node(int data1, Node* next1) {
        data = data1;
        next = next1;
    }

    Node(int data1) {
        data = data1;
        next = nullptr;
    }
};

Node *findMiddle(Node *head) {
    Node *slow = head;
    Node *fast = head;

    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }
}

```

```

    }

    return slow;
}

int main() {
    Node* head = new Node(1);

    head->next = new Node(2);

    head->next->next = new Node(3);

    head->next->next->next = new Node(4);

    head->next->next->next->next = new Node(5);

    Node* middleNode = findMiddle(head);

    cout << "The middle node value is: " <<
middleNode->data << endl;

    return 0;
}

```

Time Complexity: $O(N/2)$ The algorithm requires the 'fast' pointer to reach the end of the list which it does after approximately $N/2$ iterations (where N is the total number of nodes). Therefore, the maximum number of iterations needed to find the middle node is proportional to the number of nodes in the list, making the time complexity linear, or $O(N/2) \sim O(N)$.

Space Complexity : $O(1)$ There is constant space complexity because it uses a constant amount of extra space regardless of the size of the linked list. We only use a few variables to keep track of the middle position and traverse the list, and the memory required for these variables does not depend on the size of the list.

Reverse a Linked List

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

struct ListNode {
    int val;
    ListNode* next;

    ListNode(int x) : val(x), next(NULL) {}
};

class Solution {
public:
    ListNode* reverseList(ListNode* head) {

```

```

stack<int> st;
ListNode* temp = head;
while (temp != NULL) {
    st.push(temp->val);
    temp = temp->next;
}
temp = head;
while (temp != NULL) {
    temp->val = st.top();
    st.pop();
    temp = temp->next;
}
return head;
}
};

int main() {
    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->next = new ListNode(3);
    Solution sol;
    head = sol.reverseList(head);
    while (head != NULL) {
        cout << head->val << " ";
        head = head->next;
    }
    return 0;
}

```

Time Complexity: $O(N)$,We traverse the linked list twice once to push all node values into the stack, and once to reassign values. Each traversal takes $O(N)$ time, where N is the number of nodes.

Space Complexity: $O(N)$, We use an extra stack to store all the node values, which requires $O(N)$ additional space.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class ListNode {

```

```

public:
    int val;
    ListNode* next;
    ListNode(int val) {
        this->val = val;
        this->next = NULL;
    }
};

class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = NULL;
        ListNode* temp = head;
        while (temp != NULL) {
            ListNode* front = temp->next;
            temp->next = prev;
            prev = temp;
            temp = front;
        }
        return prev;
    }
};

int main() {
    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->next = new ListNode(3);
    head->next->next->next = new ListNode(4);
    head->next->next->next->next = new
    ListNode(5);
    Solution sol;
    ListNode* newHead = sol.reverseList(head);
    while (newHead != NULL) {
        cout << newHead->val << " ";
        newHead = newHead->next;
    }
    cout << endl;
}

```

```

    return 0;
}

```

Time Complexity: $O(N)$ Because we are traversing each node of the linked list exactly once. Each pointer reversal is done in constant time.

Space Complexity: $O(1)$ We are not using any additional data structure or recursion. All modifications are done in-place using pointers.

Recursive Approach

```

#include <bits/stdc++.h>

using namespace std;

struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(NULL) {}
};

class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if (head == NULL || head->next == NULL)
            return head;

        ListNode* newHead = reverseList(head->next);
        ListNode* front = head->next;
        front->next = head;
        head->next = NULL;
        return newHead;
    }
};

int main() {
    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->next = new ListNode(3);
    head->next->next->next = new ListNode(4);
    head->next->next->next->next = new
    ListNode(5);

    Solution sol;
    ListNode* reversed = sol.reverseList(head);

```

```

while (reversed != NULL) {
    cout << reversed->val << " ";
    reversed = reversed->next;
}

cout << endl;

return 0;
}

```

Time Complexity: $O(n)$, Each node is visited exactly once during the recursive call, and we do constant-time work for each node (like flipping pointers).

Space Complexity: $O(n)$, The recursion stack goes up to n levels deep (one for each node), which uses extra space on the call stack.

Detect a Cycle in a Linked List

Brute-Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Node {
public:
    int data;
    Node* next;
    Node(int data1, Node* next1) {
        data = data1;
        next = next1;
    }
    Node(int data1) {
        data = data1;
        next = nullptr;
    }
};

class Solution {
public:
    bool detectLoop(Node* head) {
        Node* temp = head;
        unordered_map<Node*, int> nodeMap;
        while (temp != nullptr) {

```

```

        if (nodeMap.find(temp) != nodeMap.end())
        {
            return true;
        }

        nodeMap[temp] = 1;
        temp = temp->next;
    }
    return false;
}

};

int main() {
    Node* head = new Node(1);
    Node* second = new Node(2);
    Node* third = new Node(3);
    Node* fourth = new Node(4);
    Node* fifth = new Node(5);
    head->next = second;
    second->next = third;
    third->next = fourth;
    fourth->next = fifth;
    fifth->next = third;
    Solution obj;
    if (obj.detectLoop(head)) {
        cout << "Loop detected in the linked list." <<
endl;
    } else {
        cout << "No loop detected in the linked list."
<< endl;
    }
    delete head;
    delete second;
    delete third;
    delete fourth;
    delete fifth;
    return 0;
}

```

Time Complexity: $O(N \cdot \log N)$, we traverse the entire linked list once and store and retrieve nodes from the hash map. Map operations have a worst time space complexiy of $O(\log N)$.

Space Complexity: $O(N)$, additional amount of extra space is used to store nodes in a hash map.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Node {
public:
    int data;
    Node* next;
    Node(int data1, Node* next1) {
        data = data1;
        next = next1;
    }
    Node(int data1) {
        data = data1;
        next = nullptr;
    }
};

class Solution {
public:
    bool detectLoop(Node* head) {
        Node* temp = head;
        unordered_map<Node*, int> nodeMap;
        while (temp != nullptr) {
            if (nodeMap.find(temp) != nodeMap.end())
            {
                return true;
            }
            nodeMap[temp] = 1;
            temp = temp->next;
        }
        return false;
    }
};

```

```

int main() {
    Node* head = new Node(1);
    Node* second = new Node(2);
    Node* third = new Node(3);
    Node* fourth = new Node(4);
    Node* fifth = new Node(5);
    head->next = second;
    second->next = third;
    third->next = fourth;
    fourth->next = fifth;
    fifth->next = third;
    Solution obj;
    if (obj.detectLoop(head)) {
        cout << "Loop detected in the linked list." <<
endl;
    } else {
        cout << "No loop detected in the linked list."
<< endl;
    }
    delete head;
    delete second;
    delete third;
    delete fourth;
    delete fifth;
    return 0;
}

```

Time Complexity: $O(N)$, we traverse the entire linked list once. The fast pointer either reaches the end of the list or meets the slow pointer in linear time.
Space Complexity: $O(1)$, constant amount of extra space is used to detect a cycle using Floyd's Cycle Detection Algorithm.

Starting point of loop in a Linked List

Brute Force Approach

```
#include <bits/stdc++.h>
```

```

using namespace std;
class ListNode {
public:
    int val;
    ListNode* next;
    ListNode(int x) {
        val = x;
        next = NULL;
    }
};
class Solution {
public:
    ListNode* detectCycle(ListNode* head) {
        unordered_set<ListNode*> visited;
        while (head != NULL) {
            if (visited.find(head) != visited.end()) {
                return head;
            }
            visited.insert(head);
            head = head->next;
        }
        return NULL;
    }
};
int main() {
    ListNode* head = new ListNode(3);
    head->next = new ListNode(2);
    head->next->next = new ListNode(0);
    head->next->next->next = new ListNode(-4);
    head->next->next->next->next = head->next;
    Solution obj;
    ListNode* startNode = obj.detectCycle(head);
    if (startNode)
        cout << "Cycle starts at node with value: " <<
startNode->val << endl;
    else

```

```

        cout << "No cycle found." << endl;

    return 0;
}

```

Time Complexity: $O(N)$ where N is the number of nodes in the linked list. Each node is visited only once during traversal. Hashing allows $O(1)$ lookup to check for previously visited nodes.

Space Complexity: $O(N)$ due to the additional hash set used to store visited nodes. In the worst case (no cycle), all N nodes will be stored in the hash set.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class ListNode {
public:
    int val;
    ListNode* next;
    ListNode(int x) {
        val = x;
        next = NULL;
    }
};

class Solution {
public:
    ListNode* detectCycle(ListNode* head) {
        ListNode* slow = head;
        ListNode* fast = head;

        while (fast != NULL && fast->next != NULL) {
            slow = slow->next;
            fast = fast->next->next;
            if (slow == fast) {
                slow = head;
                while (slow != fast) {
                    slow = slow->next;
                    fast = fast->next;
                }
                return slow;
            }
        }
    }
};

```

```

    }
}

return NULL;
}

};

int main() {
    ListNode* head = new ListNode(3);
    head->next = new ListNode(2);
    head->next->next = new ListNode(0);
    head->next->next->next = new ListNode(-4);
    head->next->next->next->next = head->next;

    Solution obj;

    ListNode* result = obj.detectCycle(head);
    if (result != NULL)
        cout << "Cycle starts at node with value: " <<
        result->val << endl;
    else
        cout << "No cycle found." << endl;

    return 0;
}

```

Time Complexity: $O(N)$ where N is the number of nodes in the linked list. In the worst case, we traverse the entire list once with the slow and fast pointers, and then again to find the entry point of the loop.

Space Complexity: $O(1)$ constant extra space. No additional data structures are used, only two pointers.

Length of Loop in Linked List

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Node {
public:
    int data;
    Node* next;
    Node(int data1, Node* next1) {
        data = data1;
        next = next1;
    }
};

```

```

    }
    Node(int data1) {
        data = data1;
        next = nullptr;
    }
};

class Solution {
public:
    int lengthOfLoop(Node* head) {
        unordered_map<Node*, int> visitedNodes;
        Node* temp = head;
        int timer = 0;
        while (temp != NULL) {
            if (visitedNodes.find(temp) !=
visitedNodes.end()) {
                int loopLength = timer -
visitedNodes[temp];
                return loopLength;
            }
            visitedNodes[temp] = timer;
            temp = temp->next;
            timer++;
        }
        return 0;
    }
};

int main() {
    Node* head = new Node(1);
    Node* second = new Node(2);
    Node* third = new Node(3);
    Node* fourth = new Node(4);
    Node* fifth = new Node(5);
    head->next = second;
    second->next = third;
    third->next = fourth;
    fourth->next = fifth;
    fifth->next = second;

```

```

    Solution obj;
    int loopLength = obj.lengthOfLoop(head);
    if (loopLength > 0) {
        cout << "Length of the loop: "
            << loopLength << endl;
    } else {
        cout << "No loop found in the linked list."
            << endl;
    }
    return 0;
}

```

Time Complexity: $O(N)$, we traverse the entire linked list atleast once to find the length of the loop.

Space Complexity: $O(N)$, we use a map to store the timers for the nodes in the linked list.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

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

    Node(int data1, Node* next1) {
        data = data1;
        next = next1;
    }
};

Node(int data1) {
    data = data1;
    next = nullptr;
}

class Solution {
public:
    int lengthOfLoop(Node* head) {
        Node* slow = head;
        Node* fast = head;

```

```

while (fast != NULL && fast->next != NULL)
{
    slow = slow->next;
    fast = fast->next->next;
    if (slow == fast) {
        return countLoopLength(slow);
    }
}
return 0;
}

```

```

int countLoopLength(Node* meetingPoint) {
    Node* temp = meetingPoint;
    int length = 1;
    while (temp->next != meetingPoint) {
        temp = temp->next;
        length++;
    }
    return length;
}

```

```

};

int main() {
    Node* head = new Node(1);
    Node* second = new Node(2);
    Node* third = new Node(3);
    Node* fourth = new Node(4);
    Node* fifth = new Node(5);
    head->next = second;
    second->next = third;
    third->next = fourth;
    fourth->next = fifth;
    fifth->next = second;

    Solution obj;
    int loopLength = obj.lengthOfLoop(head);
    if (loopLength > 0) {
        cout << "Length of the loop: "
            << loopLength << endl;
    }
}

```

```

} else {
    cout << "No loop found in the linked list."
        << endl;
}
return 0;
}

```

Time Complexity: $O(N)$, we traverse the entire linked list atleast once to find the length of the loop.

Space Complexity: $O(1)$, we use a constant amount of additional space, regardless of the linked list's length to find the length of the loop.

Check if the given Linked List is Palindrome

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

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

    Node(int data1, Node* next1) {
        data = data1;
        next = next1;
    }

    Node(int data1) {
        data = data1;
        next = nullptr;
    }
};

bool isPalindrome(Node* head) {
    stack<int> st;
    Node* temp = head;
    while (temp != NULL) {
        st.push(temp->data);
        temp = temp->next;
    }
}

```

```

temp = head;
while (temp != NULL) {
    if (temp->data != st.top()) {
        return false;
    }
    st.pop();
    temp = temp->next;
}
return true;
}

void printLinkedList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

int main() {
    Node* head = new Node(1);
    head->next = new Node(5);
    head->next->next = new Node(2);
    head->next->next->next = new Node(5);
    head->next->next->next->next = new Node(1);
    cout << "Original Linked List: ";
    printLinkedList(head);
    if (isPalindrome(head)) {
        cout << "The linked list is a palindrome." <<
endl;
    } else {
        cout << "The linked list is not a palindrome."
<< endl;
    }
    return 0;
}

```

Time Complexity: $O(N)$, we traverse the entire linked list twice, once to push all elements into the stack, and once to compare them with the original list.

Space Complexity: $O(N)$, we use a stack that stores all the elements of the linked list, which takes linear space in the worst case.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

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

    Node(int data1, Node* next1) {
        data = data1;
        next = next1;
    }

    Node(int data1) {
        data = data1;
        next = nullptr;
    }
};

Node* reverseLinkedList(Node* head) {
    if (head == NULL || head->next == NULL) {
        return head;
    }

    Node* newHead = reverseLinkedList(head->next);
    Node* front = head->next;
    front->next = head;
    head->next = NULL;
    return newHead;
}

bool isPalindrome(Node* head) {
    if (head == NULL || head->next == NULL) {
        return true;
    }

    Node* slow = head;
    Node* fast = head;

```

```

while (fast->next != NULL && fast->next->next
!= NULL) {
    slow = slow->next;
    fast = fast->next->next;
}
Node* newHead = reverseLinkedList(slow-
>next);
Node* first = head;
Node* second = newHead;
while (second != NULL) {
    if (first->data != second->data) {
        reverseLinkedList(newHead);
        return false;
    }

    first = first->next;
    second = second->next;
}
reverseLinkedList(newHead);
return true;
}

void printLinkedList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

int main() {
    Node* head = new Node(1);
    head->next = new Node(5);
    head->next->next = new Node(2);
    head->next->next->next = new Node(5);
    head->next->next->next->next = new Node(1);
    cout << "Original Linked List: ";
    printLinkedList(head);

```

```

if (isPalindrome(head)) {
    cout << "The linked list is a palindrome." <<
endl;
} else {
    cout << "The linked list is not a palindrome."
<< endl;
}
return 0;
}

```

Time Complexity: $O(N)$, we traverse the list twice, once to reverse half of it and once to compare, each taking $O(N/2)$, which simplifies to $O(N)$.

Space Complexity: $O(1)$, no extra space is used apart from a few pointers; operations are done in-place.

Segregate even and odd nodes in LinkedList

```

#include <bits/stdc++.h>

using namespace std;

class ListNode {
public:
    int val;
    ListNode* next;
    ListNode(int x) {
        val = x;
        next = nullptr;
    }
};

ListNode* head, *tail;

void PrintList(ListNode* head) {
    ListNode* curr = head;
    for (; curr != nullptr; curr = curr->next)
        cout << curr->val << "-->";
    cout << "null" << endl;
}

void InsertatLast(int value) {

```

```

ListNode* newnode = new ListNode(value);
if (head == nullptr)
    head = newnode, tail = newnode;
else
    tail = tail->next = newnode;
}

ListNode* SegregatetoOddEven() {
    // Creating dummy heads and tails for even and odd lists

    ListNode* oddHead = new ListNode(-1),
    *oddTail = oddHead;

    ListNode* evenHead = new ListNode(-1),
    *evenTail = evenHead;

    ListNode* curr = head, *temp;
    while (curr) {
        temp = curr;
        curr = curr->next;
        temp->next = nullptr;
        if (temp->val & 1) {
            oddTail->next = temp;
            oddTail = temp;
        }
        else {
            evenTail->next = temp;
            evenTail = temp;
        }
    }
    evenTail->next = oddHead->next;
    return evenHead->next;
}

int main() {
    InsertatLast(1);
    InsertatLast(2);
    InsertatLast(3);
    InsertatLast(4);

    cout << "Initial LinkedList : " << endl;
    PrintList(head);

```

```

ListNode* newHead = SegregatetoOddEven();

cout << "LinkedList After Segregation " <<
endl;

PrintList(newHead);

return 0;
}

```

Time Complexity: $O(n)$, We traverse the entire linked list only once to rearrange the nodes. Each node is visited exactly once. No nested traversal or re-traversal occurs. Hence, linear time in terms of the number of nodes n .

Space Complexity: $O(1)$, We do not use any extra data structures

Remove N-th node from the end of a Linked List

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

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

    Node(int data1, Node* next1) {
        data = data1;
        next = next1;
    }

    // (next set to nullptr)
    Node(int data1) {
        data = data1;
        next = nullptr;
    }
};

class Solution {
public:
    void printLL(Node* head) {
        while (head != NULL) {
            cout << head->data << " ";

```

```

        head = head->next;
    }
}

Node* deleteNthNodeFromEnd(Node* head, int N) {
    if (head == NULL) {
        return NULL;
    }

    int cnt = 0;
    Node* temp = head;
    while (temp != NULL) {
        cnt++;
        temp = temp->next;
    }

    if (cnt == N) {
        Node* newHead = head->next;
        // free memory
        delete head;
        return newHead;
    }

    int res = cnt - N;
    temp = head;
    while (temp != NULL) {
        res--;
        if (res == 0) {
            break;
        }
        temp = temp->next;
    }

    Node* delNode = temp->next;
    temp->next = temp->next->next;
    delete delNode; // free memory
    return head;
}

};

int main() {

```

```

    vector<int> arr = {1, 2, 3, 4, 5};
    int N = 3;
    Node* head = new Node(arr[0]);
    head->next = new Node(arr[1]);
    head->next->next = new Node(arr[2]);
    head->next->next->next = new Node(arr[3]);
    head->next->next->next->next = new
Node(arr[4]);

    Solution sol;
    head = sol.deleteNthNodeFromEnd(head, N);
    sol.printLL(head);
}

```

Time Complexity: $O(L) + O(L - N)$, We are calculating the length of the linked list and then iterating up to the $(L - N)$ th node of the linked list, where L is the total length of the list.

Space Complexity: $O(1)$, constant additional space is used.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Node {
public:
    int data;
    Node* next;
    Node(int data1, Node* next1) {
        data = data1;
        next = next1;
    }
    Node(int data1) {
        data = data1;
        next = nullptr;
    }
};

class Solution {
public:
    void printLL(Node* head) {
        while (head != NULL) {

```

```

        cout << head->data << " ";

        head = head->next;
    }
}

Node* deleteNthNodeFromEnd(Node* head, int
N) {
    Node* dummy = new Node(0, head);
    Node* slow = dummy;
    Node* fast = dummy;
    for (int i = 0; i <= N; i++) {
        fast = fast->next;
    }
    while (fast != NULL) {
        slow = slow->next;
        fast = fast->next;
    }
    slow->next = slow->next->next;
    return dummy->next;
}

};

int main() {
    vector<int> arr = {1, 2, 3, 4, 5};
    int N = 3;
    Node* head = new Node(arr[0]);
    head->next = new Node(arr[1]);
    head->next->next = new Node(arr[2]);
    head->next->next->next = new Node(arr[3]);
    head->next->next->next->next = new
Node(arr[4]);
    Solution sol;
    head = sol.deleteNthNodeFromEnd(head, N);
    sol.printLL(head);
}

```

Time Complexity: $O(N)$, since the fast pointer will traverse the entire linked list, where N is the length of the linked list.

Space Complexity: $O(1)$, constant additional space is used to check unique elements.

Delete the Middle Node of the Linked List

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Node {
public:
    int data;
    Node* next;
    Node(int data1, Node* next1) {
        data = data1;
        next = next1;
    }
    Node(int data1) {
        data = data1;
        next = nullptr;
    }
};

class Solution {
public:
    Node* deleteMiddle(Node* head) {
        Node* temp = head;
        int n = 0;
        while (temp != NULL) {
            n++;
            temp = temp->next;
        }
        int res = n / 2;
        temp = head;
        while (temp != NULL) {
            res--;
            if (res == 0) {
                Node* middle = temp->next;
                temp->next = temp->next->next;
                free(middle);
            }
            temp = temp->next;
        }
    }
};

```

```

        break;
    }

    temp = temp->next;
}

return head;
}
};

void printLL(Node* head) {
    Node* temp = head;
    while (temp != NULL) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

int main() {
    Node* head = new Node(1);
    head->next = new Node(2);
    head->next->next = new Node(3);
    head->next->next->next = new Node(4);
    head->next->next->next->next = new Node(5);
    cout << "Original Linked List: ";
    printLL(head);
    Solution obj;
    head = obj.deleteMiddle(head);
    cout << "Updated Linked List: ";
    printLL(head);

    return 0;
}

```

Time Complexity: $O(N + N/2)$, we traverse the entire linked list once to count the number of nodes and then traverse again to delete the middle node.

Space Complexity: $O(1)$, we have fixed number of pointers and variables to delete the Kth node.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

```

```

class Node {
public:
    int data;
    Node* next;
    Node(int data1, Node* next1) {
        data = data1;
        next = next1;
    }
    Node(int data1) {
        data = data1;
        next = nullptr;
    }
};

Node* deleteMiddle(Node* head) {
    if (head == nullptr || head->next == nullptr) {
        delete head;
        return nullptr;
    }
    Node* slow = head;
    Node* fast = head->next->next;
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }
    Node* middle = slow->next;
    slow->next = slow->next->next;
    delete middle;
    return head;
}

void printLL(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

```

```

}
int main() {
    Node* head = new Node(1);
    head->next = new Node(2);
    head->next->next = new Node(3);
    head->next->next->next = new Node(4);
    head->next->next->next->next = new Node(5);
    cout << "Original Linked List: ";
    printLL(head);
    head = deleteMiddle(head);
    cout << "Updated Linked List: ";
    printLL(head);
    return 0;
}

```

Time Complexity: $O(N/2)$, we traverse the entire linked list using slow and fast pointers, effectively covering about half the list before reaching the midpoint.
 Space Complexity: $O(1)$, we have fixed number of pointers and variables to delete the Kth node.

Sort a Linked List

Brute-Force Approach

```

#include <bits/stdc++.h>
using namespace std;
class Node {
public:
    int data;
    Node* next;
    Node(int data1, Node* next1) {
        data = data1;
        next = next1;
    }
    Node(int data1) {
        data = data1;
        next = nullptr;
    }
};

```

```

class Solution {
public:
    Node* sortLL(Node* head) {
        vector<int> arr;
        Node* temp = head;
        while (temp != nullptr) {
            arr.push_back(temp->data);
            temp = temp->next;
        }
        sort(arr.begin(), arr.end());
        temp = head;
        for (int i = 0; i < arr.size(); i++) {
            temp->data = arr[i];
            temp = temp->next;
        }
        return head;
    }
};

void printLinkedList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

int main() {
    Node* head = new Node(3);
    head->next = new Node(2);
    head->next->next = new Node(5);
    head->next->next->next = new Node(4);
    head->next->next->next->next = new Node(1);
    cout << "Original Linked List: ";
    printLinkedList(head);
    Solution obj;
    head = obj.sortLL(head);
}

```

```

    cout << "Sorted Linked List: ";

    printLinkedList(head);

    return 0;
}

```

Time Complexity: $O(2*N + N*\text{Log}N)$, we traverse the linked list, store its elements in an array, sort it, and then copy the sorted values back into the original list.
 Space Complexity: $O(N)$, additional space required to store all the elements of linked list in an array.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Node {
public:
    int data;
    Node* next;
    Node(int data1, Node* next1) {
        data = data1;
        next = next1;
    }
    Node(int data1) {
        data = data1;
        next = nullptr;
    }
};

class Solution {
public:
    Node* mergeTwoSortedLinkedLists(Node* list1,
    Node* list2) {
        Node* dummyNode = new Node(-1);
        Node* temp = dummyNode;
        while (list1 != nullptr && list2 != nullptr) {
            if (list1->data <= list2->data) {
                temp->next = list1;
                list1 = list1->next;
            } else {
                temp->next = list2;
                list2 = list2->next;
            }
        }
    }
};

```

```

    }

    temp = temp->next;
}

if (list1 != nullptr) {
    temp->next = list1;
} else {
    temp->next = list2;
}

return dummyNode->next;
}

Node* findMiddle(Node* head) {
    if (head == nullptr || head->next == nullptr) {
        return head;
    }

    Node* slow = head;
    Node* fast = head->next;

    while (fast != nullptr && fast->next !=
    nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }

    return slow;
}

Node* sortLL(Node* head) {
    if (head == nullptr || head->next == nullptr) {
        return head;
    }

    Node* middle = findMiddle(head);
    Node* right = middle->next;
    middle->next = nullptr;
    Node* left = head;
    left = sortLL(left);
    right = sortLL(right);
    return mergeTwoSortedLinkedLists(left,
    right);
}

};

```

```

void printLinkedList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

int main() {
    Node* head = new Node(3);
    head->next = new Node(2);
    head->next->next = new Node(5);
    head->next->next->next = new Node(4);
    head->next->next->next->next = new Node(1);
    cout << "Original Linked List: ";
    printLinkedList(head);
    Solution obj;
    head = obj.sortLL(head);
    cout << "Sorted Linked List: ";
    printLinkedList(head);
    return 0;
}

```

Time Complexity: $O(N \cdot \log N)$, we recursively divide the linked list into two halves and then merge two sorted halves.

Space Complexity: $O(1)$, constant additional space is required to sort the entire linked list.

Sort a Linked List of 0's 1's and 2's by changing links

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Node {
```

```
public:
```

```
    int data;
```

```
    Node* next;
```

```
    Node(int val) {
```

```
        data = val;
```

```
        next = nullptr;
```

```
    }
```

```
};
```

```
class LinkedList {
```

```
public:
```

```
    Node* head;
```

```
    LinkedList() {
```

```
        head = nullptr;
```

```
    }
```

```
    void insert(int val) {
```

```
        Node* newNode = new Node(val);
```

```
        if (!head) {
```

```
            head = newNode;
```

```
            return;
```

```
        }
```

```
        Node* temp = head;
```

```
        while (temp->next)
```

```
            temp = temp->next;
```

```
        temp->next = newNode;
```

```
    }
```

```
    void print() {
```

```
        Node* temp = head;
```

```
        while (temp) {
```

```
            cout << temp->data;
```

```
            if (temp->next) cout << " -> ";
```

```
            temp = temp->next;
```

```
        }
```

```
        cout << " -> NULL\n";
```

```
    }
```

```
};
```

```
class Solution {
```

```
public:
```

```
    void sortZeroOneTwo(LinkedList& ll) {
```

```
        Node* zeroDummy = new Node(-1);
```

```
        Node* oneDummy = new Node(-1);
```

```
        Node* twoDummy = new Node(-1);
```

```

Node* zeroTail = zeroDummy;
Node* oneTail = oneDummy;
Node* twoTail = twoDummy;
Node* curr = ll.head;
while (curr) {
    if (curr->data == 0) {
        zeroTail->next = curr;
        zeroTail = zeroTail->next;
    } else if (curr->data == 1) {
        oneTail->next = curr;
        oneTail = oneTail->next;
    } else {
        twoTail->next = curr;
        twoTail = twoTail->next;
    }
    curr = curr->next;
}
zeroTail->next = oneDummy->next ?
oneDummy->next : twoDummy->next;
oneTail->next = twoDummy->next;
twoTail->next = nullptr; // end the list
ll.head = zeroDummy->next;
delete zeroDummy;
delete oneDummy;
delete twoDummy;
}
};

int main() {
    LinkedList ll;
    Solution sol;
    ll.insert(1);
    ll.insert(2);
    ll.insert(0);
    ll.insert(1);
    ll.insert(2);
    ll.insert(0);

```

```

cout << "Original List:\n";
ll.print();
sol.sortZeroOneTwo(ll)
cout << "Sorted List:\n";
ll.print();
return 0;
}

```

Time Complexity: $O(n)$, We traverse the entire list once.
 Space Complexity: $O(1)$, Only dummy nodes and pointers are used (constant space).

Find intersection of Two Linked Lists

Brute Force Approach

```

#include<iostream>
using namespace std;
class node {
public:
    int num;
    node* next;
    node(int val) {
        num = val;
        next = NULL;
    }
};

void insertNode(node* &head, int val) {
    node* newNode = new node(val);
    if (head == NULL) {
        head = newNode;
        return;
    }
    node* temp = head;
    while (temp->next != NULL) temp = temp->next;
    temp->next = newNode;
    return;
}

```

```

node* intersectionPresent(node* head1, node*
head2) {
    while (head2 != NULL) {
        node* temp = head1;
        while (temp != NULL) {
            if (temp == head2) return head2;
            temp = temp->next;
        }
        head2 = head2->next;
    }
    return NULL;
}

void printList(node* head) {
    while (head->next != NULL) {
        cout << head->num << "->";
        head = head->next;
    }
    cout << head->num << endl;
}

int main() {
    node* head = NULL;
    insertNode(head, 1);
    insertNode(head, 3);
    insertNode(head, 1);
    insertNode(head, 2);
    insertNode(head, 4);
    node* head1 = head;
    head = head->next->next->next;
    node* headSec = NULL;
    insertNode(headSec, 3);
    node* head2 = headSec;
    headSec->next = head;
    cout << "List1: ";
    printList(head1);
    cout << "List2: ";
    printList(head2);
}

```

```

node* answerNode = intersectionPresent(head1,
head2);
if (answerNode == NULL)
    cout << "No intersection\n";
else
    cout << "The intersection point is " <<
answerNode->num << endl;
    return 0;
}

```

Time Complexity: $O(m \times n)$, For each node in list 2, the entire list 1 is iterated, resulting in nested iterations.

Space Complexity: $O(1)$, No extra space is used; the comparison is done in-place.

Better Approach

```

#include<bits/stdc++.h>
using namespace std;
class node {
public:
    int num;
    node* next;
    node(int val) {
        num = val;
        next = NULL;
    }
};

void insertNode(node* &head, int val) {
    node* newNode = new node(val);
    if (head == NULL) {
        head = newNode;
        return;
    }
    node* temp = head;
    while (temp->next != NULL) temp = temp->next;
    temp->next = newNode;
}

node* intersectionPresent(node* head1, node*
head2) {
}

```

```

unordered_set<node*> st;
while (head1 != NULL) {
    st.insert(head1);
    head1 = head1->next;
}
while (head2 != NULL) {
    if (st.find(head2) != st.end()) return head2;
    head2 = head2->next;
}
return NULL; // Return NULL if no intersection
is found
}

void printList(node* head) {
    while (head->next != NULL) {
        cout << head->num << "->";
        head = head->next;
    }
    cout << head->num << endl;
}

int main() {
    node* head = NULL;
    insertNode(head, 1);
    insertNode(head, 3);
    insertNode(head, 1);
    insertNode(head, 2);
    insertNode(head, 4);
    node* head1 = head;
    head = head->next->next->next;
    node* headSec = NULL;
    insertNode(headSec, 3);
    node* head2 = headSec;
    headSec->next = head;
    cout << "List1: ";
    printList(head1);
    cout << "List2: ";
    printList(head2);

```

```

    node* answerNode = intersectionPresent(head1,
head2);
    if (answerNode == NULL)
        cout << "No intersection\n";
    else
        cout << "The intersection point is " <<
answerNode->num << endl;
    return 0;
}

```

Time Complexity: $O(n + m)$, Iterating through list 1 first takes $O(n)$, then iterating through list 2 takes $O(m)$.

Space Complexity: $O(n)$, Storing list 1 node addresses in an unordered_set.

Optimal Approach 1

```

#include<bits/stdc++.h>
using namespace std;
class node {
public:
    int num;
    node* next;
    node(int val) {
        num = val;
        next = NULL;
    }
};

void insertNode(node* &head, int val) {
    node* newNode = new node(val);
    if (head == NULL) {
        head = newNode;
        return;
    }
    node* temp = head;
    while (temp->next != NULL) temp = temp->next;
    temp->next = newNode;
}

int getDifference(node* head1, node* head2) {
    int len1 = 0, len2 = 0;

```

```

while (head1 != NULL || head2 != NULL) {
    if (head1 != NULL) {
        ++len1;
        head1 = head1->next;
    }
    if (head2 != NULL) {
        ++len2;
        head2 = head2->next;
    }
}
return len1 - len2;
}

node* intersectionPresent(node* head1, node*
head2) {
    int diff = getDifference(head1, head2);
    if (diff < 0)
        while (diff++ != 0) head2 = head2->next;
    else
        while (diff-- != 0) head1 = head1->next;
    while (head1 != NULL) {
        if (head1 == head2) return head1;
        head2 = head2->next;
        head1 = head1->next;
    }
    return head1; // Return NULL if no intersection
}

void printList(node* head) {
    while (head->next != NULL) {
        cout << head->num << "->";
        head = head->next;
    }
    cout << head->num << endl;
}

int main() {
    node* head = NULL;
    insertNode(head, 1);

```

```

insertNode(head, 3);
insertNode(head, 1);
insertNode(head, 2);
insertNode(head, 4);
node* head1 = head;
head = head->next->next->next;
node* headSec = NULL;
insertNode(headSec, 3);
node* head2 = headSec;
headSec->next = head;
cout << "List1: ";
printList(head1);
cout << "List2: ";
printList(head2);
node* answerNode = intersectionPresent(head1,
head2);
if (answerNode == NULL)
    cout << "No intersection\n";
else
    cout << "The intersection point is " <<
answerNode->num << endl;
return 0;
}

```

Time Complexity: $O(2 \times \max(\text{length of list1, length of list2})) + O(\text{abs}(\text{length of list1} - \text{length of list2})) + O(\min(\text{length of list1, length of list2}))$, Finding the length of both lists takes $O(\max)$ time since it's done simultaneously, then moving one pointer by the difference in lengths, and finally searching for the intersection.

Space Complexity: $O(1)$, No extra space is used.

Optimal Approach 2

```

#include<bits/stdc++.h>
using namespace std;
class node {
public:
    int num;
    node* next;
    node(int val) {

```

```

        num = val;
        next = NULL;
    }
};

void insertNode(node* &head, int val) {
    node* newNode = new node(val);
    if (head == NULL) {
        head = newNode;
        return;
    }
    node* temp = head;
    while (temp->next != NULL) temp = temp->next;
    temp->next = newNode;
}

node* intersectionPresent(node* head1, node* head2) {
    node* d1 = head1;
    node* d2 = head2;
    while (d1 != d2) {
        d1 = d1 == NULL ? head2 : d1->next;
        d2 = d2 == NULL ? head1 : d2->next;
    }
    return d1;
}

void printList(node* head) {
    while (head->next != NULL) {
        cout << head->num << "->";
        head = head->next;
    }
    cout << head->num << endl;
}

int main() {
    node* head = NULL;
    insertNode(head, 1);
    insertNode(head, 3);
    insertNode(head, 1);

```

```

    insertNode(head, 2);
    insertNode(head, 4);
    node* head1 = head;
    head = head->next->next->next;
    node* headSec = NULL;
    insertNode(headSec, 3);
    node* head2 = headSec;
    headSec->next = head;
    cout << "List1: ";
    printList(head1);
    cout << "List2: ";
    printList(head2);
    node* answerNode = intersectionPresent(head1, head2);
    if (answerNode == NULL)
        cout << "No intersection\n";
    else
        cout << "The intersection point is " << answerNode->num << endl;
    return 0;
}

```

Time Complexity: $O(2 \times \max(\text{length of list1, length of list2}))$, Uses the same concept of difference of lengths of two lists.

Space Complexity: $O(1)$, No extra data structure is used.

Add 1 to a number represented by LL

Iterative approach

```

#include <bits/stdc++.h>
using namespace std;
class Node {
public:
    int data;
    Node* next;
    Node(int value) {
        data = value;

```

```

        next = nullptr;
    }
};

class LinkedList {
public:
    Node* append(Node* head, int value) {
        Node* newNode = new Node(value);
        if (!head) {
            return newNode;
        }
        Node* current = head;
        while (current->next)
            current = current->next;
        current->next = newNode;
        return head;
    }

    void printList(Node* head) {
        Node* current = head;
        while (current) {
            cout << current->data;
            current = current->next;
        }
        cout << endl;
    }
};

class Solution {
public:
    Node* reverseList(Node* node) {
        Node* prev = nullptr;
        Node* current = node;
        while (current) {
            Node* nextNode = current->next;
            current->next = prev;
            prev = current;
            current = nextNode;
        }
    }
};

```

```

        return prev;
    }

    Node* addOne(Node* head) {
        head = reverseList(head);
        Node* current = head;
        int carry = 1;
        while (current && carry) {
            int sum = current->data + carry;
            current->data = sum % 10;
            carry = sum / 10;
            if (!current->next && carry) {
                current->next = new Node(carry);
                carry = 0;
            }
            current = current->next;
        }
        head = reverseList(head);
        return head;
    }
};

```

```

int main() {
    Node* head = nullptr;
    LinkedList ll;
    Solution sol;
    head = ll.append(head, 1);
    head = ll.append(head, 2);
    head = ll.append(head, 9);
    cout << "Original Number: ";
    ll.printList(head);
    head = sol.addOne(head);
    cout << "After Adding One: ";
    ll.printList(head);
    return 0;
}

```

Time Complexity: $O(n)$, Two reversals + one pass for addition.
 Space Complexity: $O(1)$, Iterative, no extra stack used.

Recursive approach

```
#include <bits/stdc++.h>

using namespace std;

class Node {
public:
    int data;
    Node* next;
    Node(int value) {
        data = value;
        next = nullptr;
    }
};

class LinkedList {
public:
    Node* append(Node* head, int value) {
        Node* newNode = new Node(value);
        if (!head) {
            return newNode;
        }
        Node* current = head;
        while (current->next)
            current = current->next;
        current->next = newNode;
        return head;
    }

    void printList(Node* head) {
        Node* current = head;
        while (current) {
            cout << current->data;
            current = current->next;
        }
        cout << endl;
    }
};

class Solution {
public:
```

```
    int addOneUtil(Node* node) {
        if (!node) return 1;
        int carry = addOneUtil(node->next);
        int sum = node->data + carry;
        node->data = sum % 10;
        // Return new carry
        return sum / 10;
    }

    Node* addOne(Node* head) {
        int carry = addOneUtil(head);
        if (carry) {
            Node* newHead = new Node(carry);
            newHead->next = head;
            head = newHead;
        }
        return head;
    }
};

int main() {
    Node* head = nullptr;
    LinkedList ll;
    Solution sol;
    head = ll.append(head, 1);
    head = ll.append(head, 2);
    head = ll.append(head, 9);
    cout << "Original Number: ";
    ll.printList(head);
    head = sol.addOne(head);
    cout << "After Adding One: ";
    ll.printList(head);
    return 0;
}

Time Complexity: O(n), One pass for addition.
Space Complexity: O(n), Auxiliary stack space.
```

Add two numbers represented as Linked Lists

```
#include <bits/stdc++.h>

using namespace std;

struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};

class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1,
                             ListNode* l2) {
        ListNode *dummy = new ListNode();
        ListNode *temp = dummy;
        int carry = 0;
        while( (l1 != NULL || l2 != NULL) || carry) {
            int sum = 0;
            if(l1 != NULL) {
                sum += l1->val;
                l1 = l1 -> next;
            }
            if(l2 != NULL) {
                sum += l2 -> val;
                l2 = l2 -> next;
            }
            sum += carry;
            carry = sum / 10;
            ListNode *node = new ListNode(sum %
10);
            temp -> next = node;
            temp = temp -> next;
        }
        return dummy -> next;
    }
};
```

```
};

ListNode* createList(vector arr) {
    ListNode *head = new ListNode(arr[0]);
    ListNode *temp = head;
    for (int i = 1; i < arr.size(); i++) {
        temp->next = new ListNode(arr[i]);
        temp = temp->next;
    }
    return head;
}

void printList(ListNode* head) {
    while (head) {
        cout << head->val;
        if (head->next) cout << " -> ";
        head = head->next;
    }
    cout << endl;
}

int main() {
    vector num1 = {2, 4, 3};
    vector num2 = {5, 6, 4};
    ListNode* l1 = createList(num1);
    ListNode* l2 = createList(num2);
    Solution sol;
    ListNode* result = sol.addTwoNumbers(l1, l2);
    printList(result); // Output: 7 -> 0 -> 8
    return 0;
}
```

Time Complexity: $O(\max(m,n))$. Assume that m and n represent the length of $l1$ and $l2$ respectively, the algorithm above iterates at most $\max(m,n)$ times.

Space Complexity: $O(\max(m,n))$. The length of the new list is at most $\max(m,n)+1$.

Delete all occurrences of a key in DLL

```

Node * deleteAllOccurrences (Node* head, int k) {
    Node* temp = head;
    while (temp != NULL) {
        if(temp->data == k) {
            if(temp == head) {
                head = temp->next;
            }
            Node* nextNode = temp->next;
            Node* prevNode = temp->prev;
            if (nextNode != NULL) nextNode->prev = prevNode;
            if (prevNode != NULL) prevNode->next = nextNode;
            free(temp);
            temp = nextNode;
        }
        else {
            temp = temp->next;
        }
    }
    return head;
}

```

Find pairs with given sum in DLL

```

Node* findTail (Node* head) {
    Node* tail = head;
    while(tail->next != NULL) tail = tail->next;
    return tail;
}

vector<pair<int, int>> findPairs (Node* head, int k)
{
    vector<pair<int, int>> ans;
    if (head == NULL) return ans;
    Node* left = head;
    Node* right = findTail (head);
    while(left->data < right->data) {

```

```

        if(left->data + right->data == k) {
            ans.push_back({left->data, right->data});
            left = left->next;
            right = right->prev;
        }
        else if (left->data + right->data < k) {
            left = left->next;
        }
        else right = right->prev;
    }
    return ans;
}

```

Remove duplicates from sorted DLL

```

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

class Node {
public:
    int data;
    Node* prev;
    Node* next;
    Node(int value) {
        data = value;
        prev = nullptr;
        next = nullptr;
    }
};

class Solution {
public:
    Node* head = nullptr;
    void insertAtEnd(int value) {
        Node* newNode = new Node(value);
        if (!head) {
            head = newNode;

```

```

        return;
    }
    Node* current = head;
    while (current->next) {
        current = current->next;
    }
    current->next = newNode;
    newNode->prev = current;
}
Node* removeDuplicates() {
    if (!head) return nullptr;
    Node* current = head;
    while (current != nullptr && current->next !=
nullptr) {
        Node* nextDistinct = current->next;
        while (nextDistinct != nullptr &&
nextDistinct->data == current->data) {
            Node* duplicateNode = nextDistinct;
            nextDistinct = nextDistinct->next;
            delete duplicateNode; // Free memory of
duplicate node
        }
        current->next = nextDistinct;
        if (nextDistinct != nullptr) {
            nextDistinct->prev = current;
        }
        current = current->next;
    }
    return head;
}
void printList() {
    Node* current = head;
    while (current) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << endl;
}

```

```

    }
};

int main() {
    Solution sol;
    vector<int> values = {1, 2, 2, 2, 3, 4, 4, 5, 5, 6};
    for (int value : values) {
        sol.insertAtEnd(value);
    }
    cout << "Original List: ";
    sol.printList();
    sol.removeDuplicates();
    cout << "After Removing Duplicates (keeping 1
occurrence): ";
    sol.printList();
    return 0;
}

```

Time Complexity: $O(n)$, Every node is visited exactly once.
 Space Complexity: $O(1)$, No extra space is used.

Reverse Linked List in groups of Size K

```

#include <bits/stdc++.h>

using namespace std;

class ListNode {
public:
    int val;
    ListNode* next;
    ListNode(int x) {
        val = x;
        next = NULL;
    }
};

class Solution {
public:

```

```

ListNode* reverseKGroup(ListNode* head, int k) {
    ListNode* dummy = new ListNode(0);
    dummy->next = head;
    ListNode* groupPrev = dummy;
    while (true) {
        ListNode* kth = getKthNode(groupPrev, k);
        if (!kth) break;
        ListNode* groupNext = kth->next;
        ListNode* prev = groupNext;
        ListNode* curr = groupPrev->next;
        for (int i = 0; i < k; i++) {
            ListNode* temp = curr->next;
            curr->next = prev;
            prev = curr;
            curr = temp;
        }
        ListNode* temp = groupPrev->next;
        groupPrev->next = kth;
        groupPrev = temp;
    }
    return dummy->next;
}

ListNode* getKthNode(ListNode* curr, int k) {
    while (curr && k > 0) {
        curr = curr->next;
        k--;
    }
    return curr;
}

int main() {
    Solution obj;
    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->next = new ListNode(3);

```

```

    head->next->next->next = new ListNode(4);
    head->next->next->next->next = new
    ListNode(5);

    int k = 2;

    ListNode* result = obj.reverseKGroup(head, k);

    while (result != NULL) {
        cout << result->val << " ";
        result = result->next;
    }

    cout << endl;

    return 0;
}

```

Time Complexity: $O(N)$, We visit each node exactly once during reversal and during group detection (getKthNode). So the total operations are linear with respect to the number of nodes in the list.

Space Complexity: $O(1)$, The algorithm uses a constant amount of extra memory for pointers and dummy node. No additional data structures like arrays or stacks are used.

Rotate a Linked List

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(NULL) {}
};

class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head || !head->next || k == 0) return head;
        for (int i = 0; i < k; i++) {
            ListNode* curr = head;
            ListNode* prev = NULL;
            while (curr->next) {
                prev = curr;

```

```

        curr = curr->next;
    }
    prev->next = NULL;
    curr->next = head;
    head = curr;
}
return head;
}
};

int main() {
    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->next = new ListNode(3);
    head->next->next->next = new ListNode(4);
    head->next->next->next->next = new
ListNode(5);
    Solution obj;
    int k = 2;
    ListNode* result = obj.rotateRight(head, k);
    while (result) {
        cout << result->val << " ";
        result = result->next;
    }
    return 0;
}

```

Time Complexity: $O(k * n)$, We are performing 'k' rotations. In each rotation. We traverse the list to reach the second-last node (this takes $O(n)$ time), then we adjust a few pointers (which is $O(1)$). So the overall time complexity is $O(k * n)$. This approach becomes inefficient if k is large, especially with long lists, because each rotation traverses the full list.

Space Complexity: $O(1)$, We do not use any additional data structures.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

struct ListNode {
    int val;

```

```

    ListNode* next;
    ListNode(int x) {
        val = x;
        next = NULL;
    }
};

class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head || !head->next || k == 0)
            return head;
        int length = 1;
        ListNode* tail = head;
        while (tail->next) {
            tail = tail->next;
            length++;
        }
        tail->next = head;
        k = k % length;
        int stepsToNewTail = length - k;
        ListNode* newTail = head;
        for (int i = 1; i < stepsToNewTail; i++) {
            newTail = newTail->next;
        }
        ListNode* newHead = newTail->next;
        newTail->next = NULL;
        return newHead;
    }
};

int main() {
    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->next = new ListNode(3);
    head->next->next->next = new ListNode(4);
    head->next->next->next->next = new
ListNode(5);

```

```

int k = 2;

Solution obj;

ListNode* newHead = obj.rotateRight(head, k);

while (newHead) {

    cout << newHead->val << " ";

    newHead = newHead->next;

}

cout << endl;

return 0;

}

```

Time Complexity: $O(N)$, We perform a single traversal to calculate the length, another to find the new tail, and one for final breaking all linear operations.

Space Complexity: $O(1)$, No extra space is used; we just adjust pointers in place.

Flattening a Linked List

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

struct ListNode {

    int val;

    ListNode *next;

    ListNode *child;

    ListNode() {

        val = 0;

        next = NULL;

        child = NULL;

    }

    ListNode(int data1) {

        val = data1;

        next = NULL;

        child = NULL;

    }

    ListNode(int data1, ListNode *next1, ListNode*
next2) {

        val = data1;

```

```

        next = next1;

        child = next1;

    }

};

class Solution {

private:

    ListNode*

convertArrToLinkedList(vector<int>& arr) {

        ListNode* dummyNode = new ListNode(-1);

        ListNode* temp = dummyNode;

        for (int i=0; i < arr.size(); i++) {

            temp->child = new ListNode(arr[i]);

            temp = temp->child;

        }

        return dummyNode->child;

    }

public:

    ListNode* flattenLinkedList(ListNode* head) {

        vector<int> arr;

        while (head != nullptr) {

            ListNode* t2 = head;

            while (t2 != nullptr) {

                arr.push_back(t2->val);

                t2 = t2->child;

            }

            head = head->next;

        }

        sort(arr.begin(), arr.end());

        return convertArrToLinkedList(arr);

    }

};

void printLinkedList(ListNode* head) {

    while (head != nullptr) {

        cout << head->val << " ";

        head = head->child;

    }

```

```

        cout << endl;
    }
    void printOriginalLinkedList(ListNode* head, int
depth) {
        while (head != nullptr) {
            cout << head->val;
            if (head->child) {
                cout << " -> ";
                printOriginalLinkedList(head->child, depth
+ 1);
            }
            if (head->next) {
                cout << endl;
                for (int i = 0; i < depth; ++i) {
                    cout << "| ";
                }
            }
            head = head->next;
        }
    }
    int main() {
        ListNode* head = new ListNode(5);
        head->child = new ListNode(14);
        head->next = new ListNode(10);
        head->next->child = new ListNode(4);
        head->next->next = new ListNode(12);
        head->next->next->child = new ListNode(20);
        head->next->next->child->child = new
ListNode(13);
        head->next->next->next = new ListNode(7);
        head->next->next->next->child = new
ListNode(17);
        cout << "Original linked list:" << endl;
        printOriginalLinkedList(head, 0);
        Solution sol;
        ListNode* flattened =
sol.flattenLinkedList(head);
        cout << "\nFlattened linked list: ";

```

```

        printLinkedList(flattened);
        return 0;
    }

```

Time Complexity: $O(N \times M) + O(N \times M \log(N \times M)) + O(N \times M)$, where N is the number of nodes along the next pointers and M is the number of nodes along the child pointers.

$O(N \times M)$ because we traverse through all the nodes, iterating through N nodes along the next pointers and M nodes along the child pointers.

$O(N \times M \log(N \times M))$ because we sort the array containing $N \times M$ total elements.

$O(N \times M)$ because we reconstruct the linked list from the sorted array by iterating over the $N \times M$ elements.

Space Complexity: $O(N \times M) + O(N \times M)$, where N is the number of nodes along the next pointers and M is the number of nodes along the child pointers.

$O(N \times M)$ for storing all the elements in an additional array for sorting.

$O(N \times M)$ to reconstruct the linked list from the array after sorting.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

struct ListNode {
    int val;
    ListNode *next;
    ListNode *child;
    ListNode() {
        val = 0;
        next = NULL;
        child = NULL;
    }
    ListNode(int data1) {
        val = data1;
        next = NULL;
        child = NULL;
    }
    ListNode(int data1, ListNode *next1, ListNode*
next2) {
        val = data1;
        next = next1;
        child = next1;
    }
}

```

```

};

class Solution {
private:
    ListNode* merge(ListNode* list1, ListNode* list2){
        ListNode* dummyNode = new ListNode(-1);
        ListNode* res = dummyNode;
        while(list1 != NULL && list2 != NULL){
            if(list1->val < list2->val){
                res->child = list1;
                res = list1;
                list1 = list1->child;
            }
            else{
                res->child = list2;
                res = list2;
                list2 = list2->child;
            }
            res->next = NULL;
        }
        if(list1){
            res->child = list1;
        } else {
            res->child = list2;
        }
        if(dummyNode->child){
            dummyNode->child->next = NULL;
        }
        return dummyNode->child;
    }
public:
    ListNode* flattenLinkedList(ListNode* head) {
        if(head == NULL || head->next == NULL){
            return head; // Return head
        }
        ListNode* mergedHead =
        flattenLinkedList(head->next);

```

```

        head = merge(head, mergedHead);
        return head;
    }
};

void printLinkedList(ListNode* head) {
    while (head != nullptr) {
        cout << head->val << " ";
        head = head->child;
    }
    cout << endl;
}

void printOriginalLinkedList(ListNode* head, int depth) {
    while (head != nullptr) {
        cout << head->val;
        if (head->child) {
            cout << " -> ";
            printOriginalLinkedList(head->child, depth
+ 1);
        }
        if (head->next) {
            cout << endl;
            for (int i = 0; i < depth; ++i) {
                cout << "| ";
            }
        }
        head = head->next;
    }
}

int main() {
    ListNode* head = new ListNode(5);
    head->child = new ListNode(14);
    head->next = new ListNode(10);
    head->next->child = new ListNode(4);
    head->next->next = new ListNode(12);
    head->next->next->child = new ListNode(20);

```

```

    head->next->next->child->child = new
ListNode(13);

    head->next->next->next = new ListNode(7);

    head->next->next->next->child = new
ListNode(17);

    cout << "Original linked list:" << endl;

    printOriginalLinkedList(head, 0);

    Solution sol;

    ListNode* flattened =
sol.flattenLinkedList(head)

    cout << "\nFlattened linked list: ";

    printLinkedList(flattened);

    return 0;
}

```

Time Complexity: $O(N \times (2M)) \sim O(2N \times M)$, where N is the length of the linked list along the next pointer and M is the length of the linked list along the child pointers. The merge operation in each recursive call takes time complexity proportional to the length of the linked lists being merged, as they have to iterate over the entire lists. Since the vertical depth of the linked lists is assumed to be M , the time complexity for a single merge operation is proportional to $O(2M)$. This operation is performed N number of times (to each and every node along the next pointer list), hence the resultant time complexity becomes $O(N \times 2M)$. Space Complexity: $O(1)$, as this code uses no external space or additional data structures to store values. But a recursive stack uses $O(N)$ space to build the recursive calls for each node along the next pointer list.

Clone Linked List with Random and Next Pointer

Brute Force Approach

```

#include <iostream>

using namespace std;

class Node {

public:

    int data;

    Node *next;

    Node *random;

```

```

    Node() : data(0), next(nullptr),
random(nullptr){};

    Node(int x) : data(x), next(nullptr),
random(nullptr) {}

    Node(int x, Node *nextNode, Node
*randomNode) :

        data(x), next(nextNode),
random(randomNode) {}

};

void insertCopyInBetween(Node* head){

    Node* temp = head;

    while(temp != NULL){

        Node* nextElement = temp->next;

        Node* copy = new Node(temp->data);

        copy->next = nextElement;

        temp->next = copy;

        temp = nextElement;

    }

}

void connectRandomPointers(Node* head){

    Node* temp = head;

    while(temp != NULL){

        Node* copyNode = temp->next;

        if(temp->random){

            copyNode->random = temp->random-
>next;

        }

        else{

            copyNode->random = NULL;

        }

        temp = temp->next->next;

    }

}

Node* getDeepCopyList(Node* head){

    Node* temp = head;

    Node* dummyNode = new Node(-1);

    Node* res = dummyNode;

```

```

while(temp != NULL){
    res->next = temp->next;
    res = res->next;
    temp->next = temp->next->next;
    temp = temp->next;
}
return dummyNode->next;
}

Node *cloneLL(Node *head){
    if(!head) return nullptr;
    insertCopyInBetween(head);
    connectRandomPointers(head);
    return getDeepCopyList(head);
}

void printClonedLinkedList(Node *head) {
    while (head != nullptr) {
        cout << "Data: " << head->data;
        if (head->random != nullptr) {
            cout << ", Random: " << head->random-
>data;
        } else {
            cout << ", Random: nullptr";
        }
        cout << endl;
        // Move to the next node
        head = head->next;
    }
}

int main() {
    Node* head = new Node(7);
    head->next = new Node(14);
    head->next->next = new Node(21);
    head->next->next->next = new Node(28);
    head->random = head->next->next;
    head->next->random = head;
    head->next->next->random = head->next->next-
>next;

```

```

    head->next->next->next->random = head->next;

    cout << "Original Linked List with Random
Pointers:" << endl;

    printClonedLinkedList(head);
    Node* clonedList = cloneLL(head);

    cout << "\nCloned Linked List with Random
Pointers:" << endl;

    printClonedLinkedList(clonedList);

    return 0;
}

```

Time Complexity: $O(2N)$, where N is the number of nodes in the linked list. The linked list is traversed twice, once for creating copies of each node and for the second time to set the next and random pointers for each copied node. The time to access the nodes in the map is $O(1)$ due to hashing.

Space Complexity : $O(N)+O(N)$, where N is the number of nodes in the linked list as all nodes are stored in the map to maintain mappings and the copied linked lists takes $O(N)$ space as well.

Optimal Approach

```

#include <iostream>

using namespace std;

class Node {
public:
    int data;
    Node *next;
    Node *random;

    Node() : data(0), next(nullptr),
random(nullptr){};

    Node(int x) : data(x), next(nullptr),
random(nullptr) {}

    Node(int x, Node *nextNode, Node
*randomNode) :
        data(x), next(nextNode),
random(randomNode) {}
};

void insertCopyInBetween(Node* head){
    Node* temp = head;
    while(temp != NULL){
        Node* nextElement = temp->next;
        Node* copy = new Node(temp->data);

```

```

        copy->next = nextElement;
        temp->next = copy;
        temp = nextElement;
    }
}

void connectRandomPointers(Node* head){
    Node* temp = head;
    while(temp != NULL){
        Node* copyNode = temp->next;
        if(temp->random){
            copyNode->random = temp->random-
>next;
        }
        else{
            copyNode->random = NULL;
        }
        temp = temp->next->next;
    }
}

Node* getDeepCopyList(Node* head){
    Node* temp = head;
    Node* dummyNode = new Node(-1);
    Node* res = dummyNode;
    while(temp != NULL){
        res->next = temp->next;
        res = res->next;
        temp->next = temp->next->next;
        temp = temp->next;
    }
    return dummyNode->next;
}

Node *cloneLL(Node *head){
    if(!head) return nullptr;
    insertCopyInBetween(head);
    connectRandomPointers(head);
    return getDeepCopyList(head);
}

```

```

    }
}

void printClonedLinkedList(Node *head) {
    while (head != nullptr) {
        cout << "Data: " << head->data;
        if (head->random != nullptr) {
            cout << ", Random: " << head->random-
>data;
        } else {
            cout << ", Random: nullptr";
        }
        cout << endl;
        head = head->next;
    }
}

int main() {
    Node* head = new Node(7);
    head->next = new Node(14);
    head->next->next = new Node(21);
    head->next->next->next = new Node(28);
    head->random = head->next->next;
    head->next->random = head;
    head->next->next->random = head->next->next-
>next;
    head->next->next->next->random = head->next;

    cout << "Original Linked List with Random
Pointers:" << endl;

    printClonedLinkedList(head);
    Node* clonedList = cloneLL(head);

    cout << "\nCloned Linked List with Random
Pointers:" << endl;

    printClonedLinkedList(clonedList);

    return 0;
}

```

Time Complexity: $O(3N)$, where N is the number of nodes in the linked list. The algorithm makes three traversals of the linked list, once to create copies and insert them between original nodes, then to set the random pointers of the copied nodes to their appropriate copied nodes and then to separate the copied and original nodes.

Space Complexity : $O(N)$, where N is the number of nodes in the linked list as the only extra additional space allocated it to create the copied list without creating any other additional data structures.

Recursive Implementation of atoi()

```
#include <bits/stdc++.h>

using namespace std;

const int INT_MIN_VAL = -2147483648;
const int INT_MAX_VAL = 2147483647;

int helper(const string &s, int i, long long num, int sign) {
    if (i >= s.size() || !isdigit(s[i]))
        return (int)(sign * num);
    num = num * 10 + (s[i] - '0');
    if (sign * num <= INT_MIN_VAL) return INT_MIN_VAL;
    if (sign * num >= INT_MAX_VAL) return INT_MAX_VAL;
    return helper(s, i + 1, num, sign);
}

int myAtoi(string s, int i = 0) {
    while (i < s.size() && s[i] == ' ') i++;
    int sign = 1;
    if (i < s.size() && (s[i] == '+' || s[i] == '-')) {
        sign = (s[i] == '-') ? -1 : 1;
        i++;
    }
    return helper(s, i, 0, sign);
}

int main() {
    string s = "-12345";
    cout << myAtoi(s) << endl; // Output: -12345
    return 0;
}
```

Time Complexity: $O(n)$ since each character is processed once.

Space Complexity: $O(n)$ since the recursion stack grows up to n calls.

Implement Pow(x,n) | X raised to the power N

Brute Force

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    double myPow(double x, int n) {
        if (n == 0 || x == 1.0) return 1;

        long long temp = n;
        if (n < 0) {
            x = 1 / x;
            temp = -1 * ILL * n;
        }
        double ans = 1;
        for (long long i = 0; i < temp; i++) {
            ans *= x;
        }
        return ans;
    };
};

int main() {
    Solution sol;
    printf("%.4f\n", sol.myPow(2.0000, 10));
    printf("%.4f\n", sol.myPow(2.0000, -2));
    return 0;
}
```

Time Complexity: $O(n)$, where n is the absolute value of the exponent. This is because we multiply the base x , n times.

Space Complexity: $O(1)$, as we are using a constant amount of space for the variables used in the computation.

Optimal Approach

```

#include<bits/stdc++.h>

using namespace std;

class Solution {
private:
    double power(double x, long n) {
        if (n == 0) return 1.0;
        if (n == 1) return x;
        if (n % 2 == 0) {
            return power(x * x, n / 2);
        }
        return x * power(x, n - 1);
    }
public:
    double myPow(double x, int n) {
        int num = n;
        if (num < 0) {
            return (1.0 / power(x, -1 * num));
        }
        return power(x, num);
    }
};

int main() {
    Solution sol;
    double x = 2.0;
    int n = 10;
    double result = sol.myPow(x, n);

    std::cout << x << "^" << n << " = " << result <<
    std::endl;

    return 0;
}

```

Time Complexity: $O(\log n)$, where n is the absolute value of the exponent. This is because we reduce the problem size by half in each recursive call when n is even.

Space Complexity: $O(\log n)$, due to the recursive call stack. In the worst case, the depth of the recursion can go up to $\log(n)$ when n is even.

Count Good numbers

```

#include <bits/stdc++.h>

using namespace std;

const int MOD = 1e9 + 7;

int countGoodNumbers(int index, int n) {
    if (index == n) {
        return 1;
    }
    int result = 0;
    if (index % 2 == 0) {
        for (int digit : {0, 2, 4, 6, 8}) {
            result = (result + countGoodNumbers(index
            + 1, n)) % MOD;
        }
    }
    else {
        for (int digit : {2, 3, 5, 7}) {
            result = (result + countGoodNumbers(index
            + 1, n)) % MOD;
        }
    }
    return result;
}

int main() {
    int n = 1;
    cout << countGoodNumbers(0, n) << endl;

    return 0;
}

```

Time Complexity (TC): $O(2^n)$ because at each index, we check 5 even digits for even indices and 4 prime digits for odd indices, leading to exponential recursion.

Space Complexity (SC): $O(n)$ due to the recursion depth, where each function call adds to the call stack.

Sort a Stack

```

#include <bits/stdc++.h>

using namespace std;

void insert(stack<int>& s, int temp) {
    if (s.empty() || s.top() <= temp) {

```

```

        s.push(temp);
        return;
    }
    int val = s.top();
    s.pop();
    insert(s, temp);
    s.push(val);
}
void sortStack(stack<int>& s) {
    if (!s.empty()) {
        int temp = s.top();
        s.pop();
        sortStack(s);
        insert(s, temp);
    }
}
int main() {
    stack<int> s;
    s.push(4);
    s.push(1);
    s.push(3);
    s.push(2);
    sortStack(s);
    cout << "Sorted stack (descending order): ";
    while (!s.empty()) {
        cout << s.top() << " ";
        s.pop();
    }
    return 0;
}

```

Time Complexity: $O(n^2)$, where n is the number of elements in the stack.
 Space Complexity: $O(n)$, due to the recursion stack.

Reverse a stack using recursion

```
#include <bits/stdc++.h>
```

```

using namespace std;
void insertAtBottom(stack<int> &st, int val) {
    if (st.empty()) {
        st.push(val);
        return;
    }
    int topVal = st.top();
    st.pop();
    insertAtBottom(st, val);
    st.push(topVal);
}
void reverseStack(stack<int> &st) {
    if (st.empty()) return;
    int topVal = st.top();
    st.pop();
    reverseStack(st);
    insertAtBottom(st, topVal);
}
int main() {
    stack<int> st;
    st.push(4);
    st.push(1);
    st.push(3);
    st.push(2);
    reverseStack(st);
    cout << "Reversed Stack: ";
    while (!st.empty()) {
        cout << st.top() << " ";
        st.pop();
    }
    cout << endl;
    return 0;
}

```

Time Complexity: $O(n^2)$, as each element is popped and inserted at the bottom ($O(n)$ per element).
 Space Complexity: $O(n)$, as only the recursion stack is used.

Generate all binary strings

```
#include <bits/stdc++.h>

using namespace std;

void generate(int n, string curr, vector<string>& result) {

    if (curr.length() == n) {
        result.push_back(curr);
        return;
    }

    generate(n, curr + "0", result);
    if (curr.empty() || curr.back() != '1') {
        generate(n, curr + "1", result);
    }
}

int main() {

    int n = 3;

    vector<string> result;

    generate(n, "", result);

    for (string& s : result) {

        cout << s << " ";

    }

    cout << endl;

    return 0;

}
```

Time Complexity: $O(2^n)$, since each position has 2 choices.
Space Complexity: $O(n)$ per recursive path (due to call stack)

Generate Parenthesis

Brute Force

```
#include <bits/stdc++.h>

using namespace std;

bool isValid(string s) {

    int balance = 0;

    for (char c : s) {
```

```
        if (c == '(') balance++;

        else balance--;

        if (balance < 0) return false;

    }

    return balance == 0;

}

void generateAll(string curr, int n, vector<string>& res) {

    if (curr.length() == 2 * n) {

        if (isValid(curr)) res.push_back(curr);

        return;

    }

    generateAll(curr + '(', n, res);

    generateAll(curr + ')', n, res);

}

vector<string> generateParenthesis(int n) {

    vector<string> res;

    generateAll("", n, res);

    return res;

}

int main() {

    int n = 3;

    vector<string> result = generateParenthesis(n);

    for (string s : result) cout << s << endl;

    return 0;

}
```

Time Complexity: $O(2^{(2n)} * n)$ due to the generation and validation of all $2^{(2n)}$ sequences.
Space Complexity: $O(n)$ space required per sequence.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

void backtrack(string curr, int open, int close, int n, vector<string>& res) {

    if (curr.length() == 2 * n) {

        res.push_back(curr);

        return;

    }
```

```

    }
    if (open < n) backtrack(curr + '(', open + 1, close,
n, res);

    if (close < open) backtrack(curr + ')', open, close
+ 1, n, res);
}

vector<string> generateParenthesis(int n) {
    vector<string> res;

    backtrack("", 0, 0, n, res);

    return res;
}

int main() {
    int n = 3;

    vector<string> result = generateParenthesis(n);

    for (string s : result) cout << s << endl;

    return 0;
}

```

Time Complexity: $O(2^n)$ (Catalan number): $C(n) = (2n)! / (n!(n+1)!)$ is the number of valid sequences. Each sequence takes $O(n)$ to build. So, total complexity: $O(C(n) \times n)$
Space Complexity: $O(n)$ recursion depth.
 $O(C(n) \times n)$ to store results.

Power Set: Print all the possible subsequences of the String

Approach – 1

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<string> getSubsequences(string s) {
        int n = s.size();

        int total = 1 << n;

        vector<string> subsequences;

        for (int mask = 0; mask < total; mask++) {
            string subseq = "";

            for (int i = 0; i < n; i++) {
                if (mask & (1 << i)) {

```

```

                    subseq += s[i];
                }
            }

            subsequences.push_back(subseq);
        }

        return subsequences;
    }
};

int main() {
    string s = "abc";

    Solution sol;

    vector<string> subsequences =
sol.getSubsequences(s);

    for (auto &subseq : subsequences) {
        cout << "\"" << subseq << "\"" << endl;
    }

    return 0;
}

```

Time Complexity: $O(n \cdot 2^n)$, for each subsequence, we may check up to n bits to decide inclusion.
Space Complexity: $O(n \cdot 2^n)$, space used to store all possible subsequences.

Approach – 2

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void helper(string &s, int index, string cur,
vector<string> &result) {
        if (index == s.size()) {
            result.push_back(current);

            return;
        }

        helper(s, index + 1, current, result);

        current.push_back(s[index]);

        helper(s, index + 1, current, result);

        current.pop_back();
    }
}

```

```

vector<string> getSubsequences(string s) {
    vector<string> result;
    string current = "";
    helper(s, 0, current, result);
    return result;
}

int main() {
    string s = "abc";
    Solution sol;
    vector<string> subsequences =
sol.getSubsequences(s);
    for (auto &subseq : subsequences) {
        cout << "\"" << subseq << "\"" << endl;
    }
    return 0;
}

```

Time Complexity: $O(n * 2^n)$, for each subsequence, we construct and print the entire subsequence.
 Space Complexity: $O(n * 2^n)$, space used to store all possible subsequences.

Count all subsequences with sum K

```

#include<bits/stdc++.h>
using namespace std;
class Solution {
private:
    int func(int ind, int sum, vector<int> &nums) {
        if (sum == 0) return 1;
        if (sum < 0 || ind == nums.size()) return 0;
        return func(ind + 1, sum - nums[ind], nums) +
func(ind + 1, sum, nums);
    }
public:
    int
countSubsequenceWithTargetSum(vector<int>&
nums, int target) {
        return func(0, target, nums);
    }
}

```

```

}
};
int main() {
    Solution sol;
    vector<int> nums = {1, 2, 3, 4, 5};
    int target = 5;
    cout << "Number of subsequences with target
sum " << target << ": "
    <<
sol.countSubsequenceWithTargetSum(nums, target)
<< endl;
    return 0;
}

```

Time Complexity: $O(2^n)$, where n is the number of elements in the array. This is because each element can either be included or excluded from the subsequence, leading to 2^n possible combinations.
 Space Complexity: $O(n)$, where n is the depth of the recursion stack. In the worst case, the recursion can go as deep as the number of elements in the array.

Check if there exists a subsequence with sum K

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
private:
    bool func(int ind, int sum, std::vector<int>
&nums) {
        if (ind == nums.size()) {
            return sum == 0;
        }
        return func(ind + 1, sum - nums[ind], nums) |
func(ind + 1, sum, nums);
    }
public:
    bool checkSubsequenceSum(std::vector<int>&
nums, int target) {
        return func(0, target, nums); // Start the
recursive process
    }
}

```

```
};

int main() {
    Solution sol;

    std::vector<int> nums = {1, 2, 3, 4};

    int target = 5;

    cout << sol.checkSubsequenceSum(nums,
target); // Expected output: true

    return 0;
}
```

Time Complexity: $O(2^n)$, where n is the number of elements in the input array. This is because each element can either be included or excluded from the subsequence, leading to 2^n possible combinations.

Space Complexity: $O(n)$, where n is the depth of the recursion stack. In the worst case, the recursion can go as deep as n levels, leading to a space complexity of $O(n)$ due to the call stack.

Combination Sum – 1

```
#include<bits/stdc++.h>

using namespace std;

class Solution {
public:
    void findCombination(int ind, int target,
vector<int>& arr, vector<vector<int>>& ans,
vector<int>& ds) {
        if (ind == arr.size()) {
            if (target == 0) {
                ans.push_back(ds);
            }
            return;
        }
        if (arr[ind] <= target) {
            ds.push_back(arr[ind]);
            findCombination(ind, target - arr[ind], arr,
ans, ds);
            ds.pop_back();
        }
        findCombination(ind + 1, target, arr, ans, ds);
    }
}
```

```
public:
    vector<vector<int>>
combinationSum(vector<int>& candidates, int
target) {
        vector<vector<int>> ans;
        vector<int> ds;
        findCombination(0, target, candidates, ans,
ds);
        return ans;
    }
};
```

```
int main() {
    Solution obj;

    vector<int> v {2, 3, 6, 7};

    int target = 7;

    vector<vector<int>> ans =
obj.combinationSum(v, target);

    cout << "Combinations are: " << endl;
    for (int i = 0; i < ans.size(); i++) {
        for (int j = 0; j < ans[i].size(); j++) {
            cout << ans[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Time Complexity: $O(2^t * k)$ due to exploring all combinations up to the target with copying each valid combination of average length k .

Space Complexity: $O(k * x)$ to store all valid combinations, where x is the number of combinations and k is their average length.

Combination Sum II - Find all unique combinations

```
#include<bits/stdc++.h>

using namespace std;
```

```

void findCombination(int ind, int target,
vector<int>& arr, vector<vector<int>>& ans,
vector<int>& ds) {
    if (target == 0) {
        ans.push_back(ds);
        return;
    }
    for (int i = ind; i < arr.size(); i++) {
        if (i > ind && arr[i] == arr[i - 1]) continue;
        if (arr[i] > target) break;
        ds.push_back(arr[i]);
        findCombination(i + 1, target - arr[i], arr, ans,
ds);\
        ds.pop_back();
    }
}\
vector<vector<int>>
combinationSum2(vector<int>& candidates, int
target) {
    sort(candidates.begin(), candidates.end()); \
    vector<vector<int>> ans;
    vector<int> ds;
    findCombination(0, target, candidates, ans, ds);
    return ans;
}
int main() {
    vector<int> v{10, 1, 2, 7, 6, 1, 5};
    vector<vector<int>> comb =
combinationSum2(v, 8);
    cout << "[ ";
    for (int i = 0; i < comb.size(); i++) {
        cout << "[ ";
        for (int j = 0; j < comb[i].size(); j++) {
            cout << comb[i][j] << " ";
        }
        cout << "]";
    }
    cout << " ]";
}

```

```

return 0;
}

```

Time Complexity: $O(2^n * k)$, For each of the 2^n subsequences, storing takes $O(k)$ time where k is the average length of each combination.

Space Complexity: $O(k * x)$, To store all x valid combinations, each of average length k .

Subset Sum : Sum of all Subsets

Bitmasking Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    vector<int> subsetSums(vector<int>& arr) {
        int n = arr.size();
        vector<int> sums;
        for (int mask = 0; mask < (1 << n); mask++) {
            int sum = 0;
            for (int i = 0; i < n; i++) {
                // If ith bit is set, include arr[i] in sum
                if (mask & (1 << i)) {
                    sum += arr[i];
                }
            }
            sums.push_back(sum);
        }
        sort(sums.begin(), sums.end());
        return sums;
    }
};
int main() {
    Solution sol;
    vector<int> arr = {5, 2, 1};
    vector<int> result = sol.subsetSums(arr);
    for (int sum : result) {
        cout << sum << " ";
    }
}

```

```

    }

    cout << endl;

    return 0;
}

```

Time Complexity: $O(2^n * n)$, We generate all possible subsets, which is 2^n . For each subset, we iterate through the n elements to calculate its sum. Hence, total complexity is $O(2^n * n)$. Sorting the resulting sums takes $O(2^n \log(2^n))$, which is smaller than $O(2^n * n)$ for large n , so the overall remains $O(2^n * n)$.

Space Complexity: $O(2^n)$, We store all subset sums in a result array, which requires $O(2^n)$ space. Apart from this, only a few variables are used, so extra space is constant $O(1)$.

Recursive approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void findSums(int index, int currentSum,
vector<int>& arr, vector<int>& sums) {
        if (index == arr.size()) {
            sums.push_back(currentSum);
            return;
        }
        findSums(index + 1, currentSum + arr[index],
arr, sums);
        findSums(index + 1, currentSum, arr, sums);
    }
    vector<int> subsetSums(vector<int>& arr) {
        vector<int> sums;
        findSums(0, 0, arr, sums);

        sort(sums.begin(), sums.end()); // Sort in
increasing order

        return sums;
    }
};

int main() {
    Solution sol;

    vector<int> arr = {5, 2, 1};

    vector<int> result = sol.subsetSums(arr);

```

```

    for (int sum : result) {
        cout << sum << " ";
    }

    cout << endl;

    return 0;
}

```

Time Complexity: $O(2^n)$, Each element has two choices: include or exclude, leading to 2^n subsets. We directly compute sums without iterating over subsets, so complexity is $O(2^n)$. Sorting the sums adds $O(2^n \log(2^n))$, making the total $O(2^n \log(2^n))$.

Space Complexity: $O(2^n)$, The result array holds all subset sums, requiring $O(2^n)$ space. Recursion uses an additional $O(n)$ stack space due to function calls, so total auxiliary space is $O(2^n + n)$.

Subset - II | Print all the Unique Subsets

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void findSubsets(int ind, vector<int>& nums,
vector<int>& ds, set<vector<int>>& result) {
        if (ind == nums.size()) {
            result.insert(ds);
            return;
        }
        ds.push_back(nums[ind]);
        findSubsets(ind + 1, nums, ds, result);
        ds.pop_back();
        findSubsets(ind + 1, nums, ds, result);
    }

    vector<vector<int>>
subsetsWithDup(vector<int>& nums) {
        set<vector<int>> result;

        vector<int> ds;

        sort(nums.begin(), nums.end());

        findSubsets(0, nums, ds, result);
    }
};

```

```

        vector<vector<int>> ans(result.begin(),
result.end());

        return ans;
    }
};

int main() {
    Solution sol;

    vector<int> nums = {1, 2, 2};

    vector<vector<int>> ans =
sol.subsetsWithDup(nums);

    cout << "[ ";

    for (const auto& subset : ans) {
        cout << "[ ";

        for (int num : subset) {
            cout << num << " ";

        }

        cout << "] ";

    }

    cout << "]" << endl;

    return 0;
}

```

Time Complexity: $O(N^2 * 2^N)$. We generate all 2^N possible subsets, and copying each subset into temporary storage costs up to $O(N)$. Additionally, inserting each subset into a balanced BST-based set costs $O(\log(2^N)) = O(N)$, resulting in an extra $O(N)$ factor. Combining these gives $O(N * 2^N + N^2 * 2^N) \approx O(N^2 * 2^N)$.

Space Complexity: $O(N * 2^N)$. We store up to 2^N subsets in the set, each subset storing up to N elements in the worst case. Additionally, $O(N)$ space is used for the recursion stack during subset generation

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void backtrack(int start, vector<int>& nums,
vector<int>& current, vector<vector<int>>&
result) {

        result.push_back(current);

        for (int i = start; i < nums.size(); i++) {

```

```

            if (i > start && nums[i] == nums[i - 1])
continue;

            current.push_back(nums[i]);

            backtrack(i + 1, nums, current, result);

            current.pop_back();

        }

    }

    vector<vector<int>>
subsetsWithDup(vector<int>& nums) {
        sort(nums.begin(), nums.end());

        vector<vector<int>> result;

        vector<int> current;

        backtrack(0, nums, current, result);

        return result;
    }
};

int main() {
    Solution obj;

    int n;

    cin >> n;

    vector<int> nums(n);

    for (int i = 0; i < n; i++) cin >> nums[i];

    vector<vector<int>> ans =
obj.subsetsWithDup(nums);

    for (auto subset : ans) {

        cout << "[ ";

        for (int i = 0; i < subset.size(); i++) {

            cout << subset[i];

            if (i != subset.size() - 1) cout << ", ";

        }

        cout << "] ";

    }

    cout << endl;

    return 0;
}

```

Time Complexity: $O(2^N)$. In the worst case (all unique elements), we generate all possible subsets, which is

2^N . Sorting takes $O(N \log N)$, so total complexity is $O(2^N + N \log N) \approx O(2^N)$.

Space Complexity: $O(N)$, Due to recursion depth and storage of the current subset in the call stack. The output storage is $O(2^N)$ for all subsets.

Combination Sum III

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    void func(int sum, int last, vector<int> &nums,
int k, vector<vector<int>> &ans) {
        if(sum == 0 && nums.size() == k) {
            ans.push_back(nums);
            return;
        }
        if(sum <= 0 || nums.size() > k) return;
        for(int i = last; i <= 9; i++) {
            if(i <= sum) {
                nums.push_back(i);
                func(sum - i, i + 1, nums, k, ans);
                nums.pop_back();
            } else {
                break;
            }
        }
    }
public:
    vector<vector<int>> combinationSum3(int k, int
n) {
        vector<vector<int>> ans;
        vector<int> nums;
        func(n, 1, nums, k, ans);
        return ans;
    }
};

int main() {
```

```
Solution sol;

int k = 3;

int n = 7;

vector<vector<int>> result =
sol.combinationSum3(k, n);

for (const auto& combination : result) {
    for (int num : combination) {
        cout << num << " ";
    }
    cout << endl;
}

return 0;
}
```

Time Complexity: $O(2^9 * k)$, due to the exploration of all subsets of the set $\{1, 2, \dots, 9\}$.

Space Complexity: $O(k)$, where k is the number of elements in the combination. This is due to the space used by the recursive call stack and the storage of valid combinations.

Letter Combinations of a Phone number

```
#include<bits/stdc++.h>

using namespace std;

class Solution {
private:
    void func(int ind, string digits, string s,
vector<string> &ans, string combos[]) {
        if(ind == digits.size()) {
            ans.push_back(s);
            return;
        }
        int digit = digits[ind] - '0';
        for(int i = 0; i < combos[digit].size(); i++) {
            func(ind + 1, digits, s + combos[digit][i],
ans, combos);
        }
    }
public:
```

```

    vector<string> letterCombinations(string digits)
    {
        string combos[] = {"", "", "abc", "def", "ghi",
        "jkl", "mno", "pqrs", "tuv", "wxyz"};

        vector<string> ans; // Vector to store results

        string s = ""; // Temporary string to build
        combinations

        func(0, digits, s, ans, combos);

        return ans; // Return the result
    }
};

int main() {
    Solution solution;

    string digits = "23"; // Input digits

    vector<string> result =
    solution.letterCombinations(digits); // Get
    combinations

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

    return 0;
}

```

Time Complexity: $O(4^N * N)$, where n is the length of the input digits. This is because each digit can map to up to 4 letters, and there are n digits.

Space Complexity: $O(N)$, where n is the length of the input digits. This is due to the recursion stack depth.

Palindrome Partitioning

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool isPalindrome(string &s, int start, int end) {
        while (start < end) {
            if (s[start] != s[end]) return false;
            start++;
            end--;
        }
    }
}

```

```

        return true;
    }

    void backtrack(int index, string &s,
    vector<string> &path, vector<vector<string>>
    &res) {
        if (index == s.length()) {
            res.push_back(path);
            return;
        }

        for (int i = index; i < s.length(); i++) {
            if (isPalindrome(s, index, i)) {
                path.push_back(s.substr(index, i - index
                + 1));

                backtrack(i + 1, s, path, res);

                path.pop_back();
            }
        }
    }

    vector<vector<string>> partition(string s) {
        vector<vector<string>> res;

        vector<string> path;

        backtrack(0, s, path, res);

        return res;
    }
};

int main() {
    Solution sol;

    string s = "aab";

    vector<vector<string>> ans = sol.partition(s);

    for (auto &vec : ans) {
        for (auto &str : vec)
            cout << str << " ";

        cout << endl;
    }

    return 0;
}

```

Time Complexity: $O(2^N * N)$, we create all possible partitions and check whether they are palindrome or not.
 Space Complexity: $O(2^N * N) + O(N)$, additional space used to store all possible results and auxiliary stack space.

Word Search – Leetcode

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool exist(vector<vector<char>>& board, string word) {
        int rows = board.size();
        int cols = board[0].size();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (dfs(board, word, i, j, 0)) {
                    return true;
                }
            }
        }
        return false;
    }

private:
    bool dfs(vector<vector<char>>& board, string& word, int i, int j, int idx) {
        if (idx == word.size()) return true;
        if (i < 0 || j < 0 || i >= board.size() || j >= board[0].size() || board[i][j] != word[idx]) {
            return false;
        }
        char temp = board[i][j];
        board[i][j] = '#';
        bool found = dfs(board, word, i + 1, j, idx + 1) ||
            dfs(board, word, i - 1, j, idx + 1) ||
            dfs(board, word, i, j + 1, idx + 1) ||
            dfs(board, word, i, j - 1, idx + 1);
```

```
        board[i][j] = temp;

        return found;
    }
};

int main() {
    Solution sol;
    vector<vector<char>> board = {
        {'A','B','C','E'},
        {'S','F','C','S'},
        {'A','D','E','E'}
    };

    cout << boolalpha << sol.exist(board, "ABCCED") << endl; // true

    cout << boolalpha << sol.exist(board, "SEE") << endl; // true

    cout << boolalpha << sol.exist(board, "ABCB") << endl; // false
}
```

Time Complexity: $O(m * n * 4^L)$, We may start from each of the $m \times n$ cells, and explore up to 4 directions for each of the L letters in the word.
 Space Complexity: $O(L)$, Recursion depth equals the length of the word; we also modify the board in-place, so no extra space for visited tracking.

N Queen Problem | Return all Distinct Solutions to the N-Queens Puzzle

Brute-Force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool isSafe(int row, int col, vector<vector<char>> &board, int n) {
        for (int j = 0; j < col; j++) {
            if (board[row][j] == 'Q') return false;
        }
        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
```

```

        if (board[i][j] == 'Q') return false;
    }

    for (int i = row, j = col; i < n && j >= 0; i++, j--) {
        if (board[i][j] == 'Q') return false;
    }

    return true;
}

void solve(int col, vector<vector<char>> &board,
           vector<vector<string>> &ans, int n) {
    if (col == n) {
        vector<string> temp;
        for (int i = 0; i < n; i++) {
            string row(board[i].begin(),
                       board[i].end());
            temp.push_back(row);
        }
        ans.push_back(temp);
        return;
    }

    for (int row = 0; row < n; row++) {
        if (isSafe(row, col, board, n)) {
            board[row][col] = 'Q';
            solve(col + 1, board, ans, n);
            board[row][col] = '.';
        }
    }
}

vector<vector<string>> solveNQueens(int n) {
    vector<vector<string>> ans;
    vector<vector<char>> board(n,
                              vector<char>(n, '.'));
    solve(0, board, ans, n);
    return ans;
}
};

```

```

int main() {
    Solution obj;

    int n = 4;

    vector<vector<string>> res =
obj.solveNQueens(n);

    for (auto &board : res) {
        for (auto &row : board) {
            cout << row << "\n";
        }
        cout << "\n";
    }

    return 0;
}

```

Time Complexity: $O(N! \cdot N)$, we try all possible permutations of placing the queens and check for safety.
Space Complexity: $O(N^2 + N)$, additional space used for storing distinct boards and stack space.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void solve(int col, vector<string>& board, int n,
               vector<int>& leftRow, vector<int>&
upperDiagonal, vector<int>& lowerDiagonal,
               vector<vector<string>>& ans) {
        if (col == n) {
            ans.push_back(board);
            return;
        }

        for (int row = 0; row < n; row++) {
            if (leftRow[row] == 0 &&
lowerDiagonal[row + col] == 0 &&
upperDiagonal[n - 1 + col - row] == 0) {
                board[row][col] = 'Q';
                leftRow[row] = 1;
                lowerDiagonal[row + col] = 1;
                upperDiagonal[n - 1 + col - row] = 1;
            }
        }
    }
};

```

```

        solve(col + 1, board, n, leftRow,
upperDiagonal, lowerDiagonal, ans);

        board[row][col] = '.';

        leftRow[row] = 0;

        lowerDiagonal[row + col] = 0;

        upperDiagonal[n - 1 + col - row] = 0;

    }

}

vector<vector<string>>> solveNQueens(int n) {
    vector<vector<string>>> ans;

    vector<string> board(n, string(n, '.'));

    vector<int> leftRow(n, 0), upperDiagonal(2 *
n - 1, 0), lowerDiagonal(2 * n - 1, 0);

    solve(0, board, n, leftRow, upperDiagonal,
lowerDiagonal, ans);

    return ans;
}

};

int main() {
    Solution obj;

    int n = 4;

    vector<vector<string>>> res =
obj.solveNQueens(n);

    for (auto& board : res) {
        for (auto& row : board) {
            cout << row << "\n";
        }

        cout << "\n";
    }

    return 0;
}

```

Time Complexity: $O(N!)$, we try all possible permutations of placing the queens.
Space Complexity: $O(N)$, three boolean arrays are stored to check for safety.

Rat in a Maze

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool isSafe(int x, int y, int n,
vector<vector<int>>> &maze,
        vector<vector<int>>> &visited) {
        return (x >= 0 && x < n && y >= 0 && y < n
&&
            maze[x][y] == 1 && visited[x][y] == 0);
    }

    void solve(int x, int y, int n, vector<vector<int>>>
&maze,
        vector<vector<int>>> &visited, string path,
        vector<string> &res) {
        if (x == n - 1 && y == n - 1) {
            res.push_back(path);
            return;
        }

        visited[x][y] = 1;

        if (isSafe(x + 1, y, n, maze, visited)) {
            solve(x + 1, y, n, maze, visited, path + "D",
res);
        }

        if (isSafe(x, y - 1, n, maze, visited)) {
            solve(x, y - 1, n, maze, visited, path + "L",
res);
        }

        if (isSafe(x, y + 1, n, maze, visited)) {
            solve(x, y + 1, n, maze, visited, path + "R",
res);
        }

        if (isSafe(x - 1, y, n, maze, visited)) {
            solve(x - 1, y, n, maze, visited, path + "U",
res);
        }

        visited[x][y] = 0;
    }
}

```

```

vector<string> findPath(vector<vector<int>>
&maze, int n) {
    vector<string> res;
    vector<vector<int>> visited(n, vector<int>(n,
0));
    if (maze[0][0] == 1) {
        solve(0, 0, n, maze, visited, "", res);
    }
    return res;
}
};

int main() {
    vector<vector<int>> maze = {
        {1, 0, 0, 0},
        {1, 1, 0, 1},
        {1, 1, 0, 0},
        {0, 1, 1, 1}
    };
    int n = maze.size();
    Solution obj;
    vector<string> paths = obj.findPath(maze, n);
    for (auto &p : paths) cout << p << " ";
}

```

Time Complexity: $O(4^{(N*N)})$, because on every cell we need to try 4 different directions.
Space Complexity: $O(N*N)$, additional space for visited array and maximum Depth of the recursion tree(auxiliary space).

Word Break

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    bool wordBreak(string s, vector<string>&
wordDict) {
        int n = s.length();
        unordered_set<string> dict(wordDict.begin(),
wordDict.end());

```

```

vector<bool> dp(n + 1, false);
dp[0] = true;
int maxlen = 0;
for (const string& word : wordDict) {
    maxlen = max(maxLen, (int)word.size());
}
for (int i = 1; i <= n; ++i) {
    for (int j = max(0, i - maxlen); j < i; ++j) {
        if (dp[j] && dict.find(s.substr(j, i - j)) !=
dict.end()) {
            dp[i] = true;
            break;
        }
    }
};

int main() {
    Solution obj;
    string s = "leetcode";
    vector<string> wordDict = {"leet", "code"};
    if (obj.wordBreak(s, wordDict)) {
        cout << "True" << endl;
    } else {
        cout << "False" << endl;
    }
    return 0;
}

```

Time Complexity: $O(n * m)$, where n is the length of `s` and m is the average length of words in `wordDict`. The inner loop now runs over a limited range due to the word lengths.

Space Complexity: $O(n)$, for the DP array and the set used for dictionary lookup.

M - Coloring Problem

```

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

```

```

bool isSafe(int node, int color[], bool
graph[101][101], int n, int col) {
    for (int k = 0; k < n; k++) {
        if (k != node && graph[k][node] == 1 &&
color[k] == col) {
            return false;
        }
    }
    return true; // Safe to assign the color
}

bool solve(int node, int color[], int m, int N, bool
graph[101][101]) {
    if (node == N) {
        return true;
    }
    for (int i = 1; i <= m; i++) {
        if (isSafe(node, color, graph, N, i)) {
            color[node] = i;
            if (solve(node + 1, color, m, N, graph))
return true;
            color[node] = 0;
        }
    }
    return false;
}

bool graphColoring(bool graph[101][101], int m,
int N) {
    int color[N] = { 0 };
    if (solve(0, color, m, N, graph)) return true;
    return false;
}

int main() {
    int N = 4;
    int m = 3;
    bool graph[101][101];
    memset(graph, false, sizeof graph);
    graph[0][1] = 1; graph[1][0] = 1;
    graph[1][2] = 1; graph[2][1] = 1;

```

```

graph[2][3] = 1; graph[3][2] = 1;
graph[3][0] = 1; graph[0][3] = 1;
graph[0][2] = 1; graph[2][0] = 1;
cout << graphColoring(graph, m, N);

return 0;
}

```

Time Complexity: $O(m^N)$, where m is the number of colors and N is the number of nodes in the graph. This is because we try to color each node with m different colors, leading to an exponential time complexity in the worst case.

Space Complexity: $O(N)$, where N is the number of nodes in the graph. This is due to the recursion stack and the color array used to store the colors assigned to each node.

Sudoku Solver

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool isValid(vector<vector<char>>> &board, int
row, int col, char c) {
        for (int i = 0; i < 9; i++) {
            if (board[i][col] == c)
                return false;
        }
        for (int j = 0; j < 9; j++) {
            if (board[row][j] == c)
                return false;
        }
        int boxRowStart = 3 * (row / 3);
        int boxColStart = 3 * (col / 3);
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[boxRowStart + i][boxColStart + j]
== c)
                    return false;
            }
        }
        return true;
    }
}

```

```

}
bool solveSudoku(vector<vector<char>> &board)
{
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (board[i][j] == '.') {
                for (char c = '1'; c <= '9'; c++) {
                    if (isValid(board, i, j, c)) {
                        board[i][j] = c;
                        if (solveSudoku(board))
                            return true;
                        board[i][j] = '.';
                    }
                }
                return false;
            }
        }
    }
    return true;
}

int main() {
    vector<vector<char>> board {
        {'9', '5', '7', '.', '1', '3', '.', '8', '4'},
        {'4', '8', '3', '.', '5', '7', '1', '.', '6'},
        {'.', '1', '2', '.', '4', '9', '5', '3', '7'},
        {'1', '7', '.', '3', '.', '4', '9', '.', '2'},
        {'5', '.', '4', '9', '7', '.', '3', '6', '.'},
        {'3', '.', '9', '5', '.', '8', '7', '.', '1'},
        {'8', '4', '5', '7', '9', '.', '6', '1', '3'},
        {'.', '9', '1', '.', '3', '6', '.', '7', '5'},
        {'7', '.', '6', '1', '8', '5', '4', '.', '9'}
    };
    Solution sol;
    sol.solveSudoku(board);
    for (int i = 0; i < 9; i++) {

```

```

        for (int j = 0; j < 9; j++)
            cout << board[i][j] << " ";
        cout << "\n";
    }
    return 0;
}

```

Time Complexity: $O(9^{(n^2)})$, in the worst case, for each cell in the n^2 board, we have 9 possible numbers.

Space Complexity: $O(1)$, since we are refilling the given board itself, there is no extra space required, so constant space complexity.

Expression Add Operators

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void dfs(string& num, int target, int start, long long current_value, long long last_operand, string expression, vector<string>& result) {
        if (start == num.size()) {
            if (current_value == target)
                result.push_back(expression);
            return;
        }
        for (int i = start; i < num.size(); i++) {
            if (i > start && num[start] == '0') return;
            string current_num = num.substr(start, i - start + 1);
            long long current_num_val =
                stoll(current_num);
            if (start == 0) {
                dfs(num, target, i + 1, current_num_val,
                    current_num_val, current_num, result);
            }
            else {

```

```

        dfs(num, target, i + 1, current_value +
current_num_val, current_num_val, expression +
"+"

        + current_num, result);

        dfs(num, target, i + 1, current_value -
current_num_val, -current_num_val, expression +
"_"

        + current_num, result);

        dfs(num, target, i + 1, current_value -
last_operand + last_operand * current_num_val,
last_operand

        * current_num_val, expression + "*" +
current_num, result);
    }
}

vector<string> addOperators(string num, int
target) {
    vector<string> result;
    dfs(num, target, 0, 0, 0, "", result);
    return result;
}

};

int main() {
    string num = "123";
    int target = 6;
    Solution sol;

    vector<string> result = sol.addOperators(num,
target);

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

    return 0;
}

```

Time Complexity: $O(4^n)$, since in each recursive call, we can choose 4 possibilities for each substring (three operators: +, -, *, or no operator in the case of the first number), resulting in an exponential time complexity with a branching factor of 4.

Space Complexity: $O(n)$, since the space complexity is dominated by the recursion depth, which can go as deep as the length of the string. Additionally, we store the result expressions in a list, but this doesn't increase the

space complexity beyond the recursion stack and the input size.

Check if the i-th bit is set or not

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool checkIthBit(int n, int i) {
        string binary = "";
        while (n > 0) {
            binary += (n % 2 == 0 ? '0' : '1');
            n /= 2;
        }
        if (i >= binary.size()) return false;
        return binary[i] == '1';
    }
};

int main() {
    Solution sol;
    int num = 5;
    int bitIndex = 2;
    if (sol.checkIthBit(num, bitIndex)) {
        cout << "The " << bitIndex << "-th bit of " <<
num << " is set (1)." << endl;
    } else {
        cout << "The " << bitIndex << "-th bit of " <<
num << " is not set (0)." << endl;
    }

    return 0;
}

```

Time Complexity: $O(\log n)$, due to integer-to-binary conversion and indexing.

Space Complexity: $O(\log n)$, for the binary string.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

```

```

class Solution {
public:
    bool checkIthBit(int n, int i) {
        return (n & (1 << i)) != 0;
    }
};

int main() {
    Solution sol;

    int num = 5;
    int bitIndex = 2;

    if (sol.checkIthBit(num, bitIndex)) {
        cout << "The " << bitIndex << "-th bit of " <<
num << " is set (1)." << endl;
    } else {
        cout << "The " << bitIndex << "-th bit of " <<
num << " is not set (0)." << endl;
    }

    return 0;
}

```

Time Complexity: $O(1)$, constant time bitwise operation.
Space Complexity: $O(1)$, no additional space used.

check if a number is odd or not

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool isOdd(int n) {
        return (n % 2 != 0);
    }
};

int main() {
    Solution sol;

    int num = 7;

    if (sol.isOdd(num)) {
        cout << num << " is odd." << endl;
    }
}

```

```

    } else {
        cout << num << " is not odd." << endl;
    }

    return 0;
}

```

Time Complexity: $O(1)$ — The modulus operation takes constant time.

Space Complexity: $O(1)$ — No extra space is required.

Check if a number is power of 2 or not

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool isPowerOfTwo(int n) {
        return n > 0 && (n & (n - 1)) == 0; // Check
if n is greater than 0 and has only one bit set
    }
};

int main() {
    Solution sol;

    int num = 8;

    if (sol.isPowerOfTwo(num)) {
        cout << num << " is a power of two." << endl;
    } else {
        cout << num << " is not a power of two." <<
endl;
    }

    return 0;
}

```

Time Complexity: $O(1)$, because bitwise operations take constant time.

Space Complexity: $O(1)$, no extra space used.

Count the number of set bits

Brute Force

```

#include <bits/stdc++.h>

```

```

using namespace std;

class Solution {
public:
    int countSetBits(int n) {
        int count = 0;
        while (n > 0) {
            count += (n & 1);
            n >>= 1;
        }
        return count;
    }
};

int main() {
    int n = 29;
    Solution sol;
    int result = sol.countSetBits(n);
    cout << "The number of set bits is: " << result
    << endl;
    return 0;
}

```

Time Complexity: $O(\log n)$, because each bit of the integer is checked once.
Space Complexity: $O(1)$, only a few variables are used.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int countSetBits(int n) {
        int count = 0;
        while (n) {
            n &= (n - 1);
            count++;
        }
        return count;
    }
};

```

```

int main() {
    int n = 29;
    Solution sol;
    int result = sol.countSetBits(n);
    cout << "The number of set bits is: " << result
    << endl;
    return 0;
}

```

Time Complexity: $O(k)$, where k is the number of set bits (often faster than checking all bits).
Space Complexity: $O(1)$, only a few variables are used.

Set the rightmost bit

```

#include <bits/stdc++.h>

using namespace std;

int setRightmostUnsetBit(int n) {
    return n | (n + 1);
}

int main() {
    int n = 10;
    int result = setRightmostUnsetBit(n);
    cout << "Number after setting rightmost unset
    bit: " << result << endl;
    return 0;
}

```

Time Complexity: $O(1)$ since only one bitwise operation is performed.
Space Complexity: $O(1)$ since no extra space is used.

Swap two numbers

```

#include <bits/stdc++.h>

using namespace std;

void swapXOR(int &a, int &b) {
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
}

int main() {

```

```

int a = 5, b = 10;

swapXOR(a, b);

cout << "a = " << a << ", b = " << b << endl;

return 0;
}

```

Time Complexity: O(1) Constant operations.
Space Complexity: O(1) No extra space used.

Divide two integers without using multiplication, division and mod operator

Brute-Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int divide(int dividend, int divisor) {
        if(dividend == divisor) return 1;

        if(dividend == INT_MIN && divisor == -1)
            return INT_MAX;

        if(divisor == 1) return dividend;

        bool isPositive = true;

        if(dividend >= 0 && divisor < 0)
            isPositive = false;
        else if(dividend < 0 && divisor > 0)
            isPositive = false;

        long long n = dividend;
        long long d = divisor;
        n = abs(n);
        d = abs(d);

        long long ans = 0, sum = 0;
        while(sum + d <= n) {
            ans++;
            sum += d;
        }

        if(ans > INT_MAX && isPositive)
            return INT_MAX;
    }
}

```

```

        if(ans > INT_MAX && !isPositive)
            return INT_MIN;

        return isPositive ? ans : -1*ans;
    }
};

int main() {
    int dividend = 10, divisor = 3;

    Solution sol;

    int ans = sol.divide(dividend, divisor);

    cout << "The result of dividing " << dividend <<
    " and " << divisor << " is " << ans;

    return 0;
}

```

Time Complexity: O(dividend) In the worst case when the divisor is 1, the number of iterations taken will be O(dividend).

Space Complexity: O(1) Using a couple of variables i.e., constant space.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int divide(int dividend, int divisor) {
        if(dividend == divisor) return 1;

        if(dividend == INT_MIN && divisor == -1)
            return INT_MAX;

        if(divisor == 1) return dividend;

        bool isPositive = true;

        if(dividend >= 0 && divisor < 0)
            isPositive = false;
        else if(dividend < 0 && divisor > 0)
            isPositive = false;

        long long n = dividend;
        long long d = divisor;
        n = abs(n);
        d = abs(d);

        long long ans = 0, sum = 0;
    }
}

```

```

while(sum + d <= n) {
    ans++;
    sum += d;
}
if(ans > INT_MAX && isPositive)
    return INT_MAX;
if(ans > INT_MAX && !isPositive)
    return INT_MIN;
return isPositive ? ans : -1*ans;
}
};

int main() {
    int dividend = 10, divisor = 3;
    Solution sol;
    int ans = sol.divide(dividend, divisor);

    cout << "The result of dividing " << dividend <<
" and " << divisor << " is " << ans;

    return 0;
}

```

Time Complexity: $O((\log N)^2)$ – (where N is the absolute value of dividend). The outer loop runs for $O(\log N)$ times. The inner loop runs for $O(\log N)$ (approx.) times as well.
Space Complexity: $O(1)$ – Using a couple of variables i.e., constant space.

Count number of bits to be flipped to convert A to B

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int minBitsFlip(int start, int goal) {
        int num = start ^ goal;
        int count = 0;
        for(int i = 0; i < 32; i++) {
            count += (num & 1);
            num = num >> 1;
        }
    }
};

```

```

}
return count;
}
};

int main() {
    int start = 10, goal = 7;
    Solution sol;
    int ans = sol.minBitsFlip(start, goal);

    cout << "The minimum bit flips to convert
number is: " << ans;

    return 0;
}

```

Time Complexity: $O(1)$, The XOR operation between two integers is performed in constant time, $O(1)$. The for loop iterates over a fixed number of bits (32 bits for a standard integer), which is as good as $O(1)$.

Space Complexity: $O(1)$ – Using a couple of variables i.e., constant space.

Find the number that appears odd number of times

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int singleNumber(vector<int>& nums){
        unordered_map <int, int> mpp;

        for(int i=0; i < nums.size(); i++) {
            mpp[nums[i]]++; //Update the map
        }

        for(auto it : mpp) {
            if(it.second == 1) {
                return it.first;
            }
        }

        return -1;
    }
};

```

```
int main() {
    vector<int> nums = {1, 2, 2, 4, 3, 1, 4};

    Solution sol;

    int ans = sol.singleNumber(nums);

    cout << "The single number in given array is: "
    << ans;

    return 0;
}
```

Time Complexity: $O(N)$ (where N is the size of the array).

- Traversing the array to update the Hash Map: $O(N)$.
- Traversing the map: $O(N)$ (in the worst case).

Space Complexity: $O(N)$, since using a hashmap data structure, and in the worst case (when all elements in the array are unique), it will store N key-value pairs.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int singleNumber(vector<int>& nums){
        int XOR = 0;

        for(int i = 0; i < nums.size(); i++) {
            XOR ^= nums[i];
        }

        return XOR;
    }
};

int main() {
    vector<int> nums = {1, 2, 2, 4, 3, 1, 4};

    Solution sol;

    int ans = sol.singleNumber(nums);

    cout << "The single number in given array is: "
    << ans;

    return 0;
}
```

Time Complexity: $O(N)$ (where N is the size of the array) – Traversing through the array once will result in

$O(N)$ time complexity.

Space Complexity: $O(1)$ – Using constant space.

Power Set | Bit Manipulation

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<vector<int>>> getPowerSet(vector<int>& nums) {
        int n = nums.size();

        int subsets = 1 << n;

        vector<vector<int>>> ans;

        for (int num = 0; num < subsets; num++) {
            vector<int> subset;

            for (int i = 0; i < n; i++) {
                if (num & (1 << i)) {
                    subset.push_back(nums[i]);
                }
            }

            ans.push_back(subset);
        }

        return ans;
    }
};

int main() {
    vector<int> nums = {5, 7, 8};

    Solution obj;

    vector<vector<int>>> subsets =
    obj.getPowerSet(nums);

    cout << "Initial Input Array: ";

    for (auto num : nums) {
        cout << num << " ";
    }

    cout << endl;

    cout << "Subsets: " << endl;

    for (auto subset : subsets) {
```

```

        cout << "[";

        for (auto num : subset) {
            cout << num << " ";
        }

        cout << "]" << endl;
    }

    return 0;
}

```

Time Complexity: $O(N \times 2^N)$ where N is the number of elements in the input array. Iterating through all possible numbers from 0 to $2^N - 1$ where N is the number of elements in the input array requires $O(2^N)$ iterations. For each iteration, we perform $O(N)$ operations to construct the corresponding subset by interpreting the bits of the number.

Space Complexity: $O(N \times 2^N)$ where N is the number of elements in the input array. We store all subsets in a list. Since there are 2^N subsets in the power set, each subset can have at most N elements.

Find XOR of numbers from L to R

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int findRangeXOR(int l, int r){
        int ans = 0;
        for(int i=l; i <= r; i++) {
            ans ^= i;
        }

        return ans;
    }
};

int main() {
    int l = 3, r = 5;

    Solution sol;

    int ans = sol.findRangeXOR(l, r);

    cout << "The XOR of numbers from " << l << "
to " << r << " is: " << ans;

    return 0;
}

```

```

}

```

Time Complexity: $O(N)$ Traversing through all the numbers take $O(N)$ time.

Space Complexity: $O(1)$ Using only a couple of variables, i.e., constant space.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    int XORtillN(int n) {
        if(n % 4 == 1) return 1;
        if(n % 4 == 2) return n+1;
        if(n % 4 == 3) return 0;
        return n;
    }

public:
    int findRangeXOR(int l, int r){

        return XORtillN(l-1) ^ XORtillN(r);
    }
};

int main() {
    int l = 3, r = 5;

    Solution sol;

    int ans = sol.findRangeXOR(l, r);

    cout << "The XOR of numbers from " << l << "
to " << r << " is: " << ans;

    return 0;
}

```

Time Complexity: $O(1)$ Using constant time operations.
Space Complexity: $O(1)$ Using a couple of variables i.e., constant space.

Find the two numbers appearing odd number of times

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

```

```

class Solution {
public:
    vector<int> singleNumber(vector<int>& nums){
        vector<int> ans;
        unordered_map <int, int> mpp;
        for(int i=0; i < nums.size(); i++) {
            mpp[nums[i]]++;
        }
        for(auto it : mpp) {
            if(it.second == 1) {
                ans.push_back(it.first);
            }
        }
        sort(ans.begin(), ans.end());
        return ans;
    }
};

int main() {
    vector<int> nums = {1, 2, 1, 3, 5, 2};
    Solution sol;
    vector<int> ans = sol.singleNumber(nums);
    cout << "The single numbers in given array are:
" << ans[0] << " and " << ans[1];

    return 0;
}

```

Time Complexity: $O(N)$, where N is the size of the array.

- Traversing the array to update the Hash Map: $O(N)$.
- Traversing the map: $O(N)$ (in the worst case).
- Sorting the answer array: $O(2 \cdot \log(2)) \sim O(1)$.

Hence, the overall time complexity is $O(N) + O(N) + O(1) \sim O(N)$.

Space Complexity: $O(N)$, since we are using a hashmap data structure, and in the worst case (when all elements in the array are unique), it will store N key-value pairs.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

```

```

class Solution {
public:
    vector<int> singleNumber(vector<int>& nums){
        int n = nums.size();
        long XOR = 0;
        for(int i=0; i < n; i++) {
            XOR = XOR ^ nums[i];
        }
        int rightmost = (XOR & (XOR - 1)) ^ XOR;
        int XOR1 = 0, XOR2 = 0;
        for(int i=0; i < n; i++) {
            if(nums[i] & rightmost) {
                XOR1 = XOR1 ^ nums[i];
            }
            else {
                XOR2 = XOR2 ^ nums[i];
            }
        }
        if(XOR1 < XOR2) return {XOR1, XOR2};
        return {XOR2, XOR1};
    }
};

int main() {
    vector<int> nums = {1, 2, 1, 3, 5, 2};
    Solution sol;
    vector<int> ans = sol.singleNumber(nums);
    cout << "The single numbers in given array are:
" << ans[0] << " and " << ans[1];

    return 0;
}

```

Time Complexity: $O(N)$, traversing the array twice results in $O(2 \cdot N)$ time complexity.

Space Complexity: $O(1)$, using a couple of variables, i.e., constant space.

Find the two numbers appearing odd number of times

Brute Force

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> singleNumber(vector<int>& nums){
        vector<int> ans;
        unordered_map <int, int> mpp;
        for(int i=0; i < nums.size(); i++) {
            mpp[nums[i]]++;
        }
        for(auto it : mpp) {
            if(it.second == 1) {
                ans.push_back(it.first);
            }
        }
        sort(ans.begin(), ans.end());
        return ans;
    }
};

int main() {
    vector<int> nums = {1, 2, 1, 3, 5, 2};\
    Solution sol;
    vector<int> ans = sol.singleNumber(nums);
    cout << "The single numbers in given array are:
" << ans[0] << " and " << ans[1];
    return 0;
}
```

Time Complexity: $O(N)$, where N is the size of the array.

- Traversing the array to update the Hash Map: $O(N)$.
- Traversing the map: $O(N)$ (in the worst case).
- Sorting the answer array: $O(2 \cdot \log(2)) \sim O(1)$.

Hence, the overall time complexity is $O(N) + O(N) + O(1) \sim O(N)$.

Space Complexity: $O(N)$, since we are using a hashmap data structure, and in the worst case (when all elements in the array are unique), it will store N key-value pairs.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> singleNumber(vector<int>& nums){
        int n = nums.size();
        long XOR = 0;
        for(int i=0; i < n; i++) {
            XOR = XOR ^ nums[i];
        }
        int rightmost = (XOR & (XOR - 1)) ^ XOR;
        int XOR1 = 0, XOR2 = 0;
        for(int i=0; i < n; i++) {
            if(nums[i] & rightmost) {
                XOR1 = XOR1 ^ nums[i];
            }
            else {
                XOR2 = XOR2 ^ nums[i];
            }
        }
        if(XOR1 < XOR2) return {XOR1, XOR2};
        return {XOR2, XOR1};
    }
};

int main() {
    vector<int> nums = {1, 2, 1, 3, 5, 2};
    Solution sol;
    vector<int> ans = sol.singleNumber(nums);
    cout << "The single numbers in given array are:
" << ans[0] << " and " << ans[1];
    return 0;
}
```

Time Complexity: $O(N)$, traversing the array twice results in $O(2 \cdot N)$ time complexity.

Space Complexity: $O(1)$, using a couple of variables, i.e., constant space.

All Divisors of a Number

Brute Force

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> divisors(int n) {
        vector<int> ans;
        for(int i=1; i <= n; i++) {
            if(n % i == 0) {
                ans.push_back(i);
            }
        }
        return ans;
    }
};

int main(){
    int n = 6;
    Solution sol;
    vector<int> ans = sol.divisors(n);
    cout << "The divisors of " << n << " are: ";
    for(int i=0; i < ans.size(); i++) {
        cout << ans[i] << " ";
    }
    return 0;
}
```

Time Complexity: $O(N)$ – Iterating N times, and performing constant time operations in each pass.

Space Complexity: $O(\sqrt{N})$ – A number N can have at max $2\sqrt{N}$ divisors, which are stored in the array.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> divisors(int n) {
        vector<int> ans;
```

```
int sqrtN = sqrt(n);
for(int i=1; i <= sqrtN; i++) {
    if(n % i == 0) {
        ans.push_back(i);
        if(i != n / i) {
            ans.push_back(n/i);
        }
    }
}
sort(ans.begin(), ans.end());
return ans;
};

int main(){
    int n = 6;
    Solution sol;
    vector<int> ans = sol.divisors(n);
    cout << "The divisors of " << n << " are: ";
    for(int i=0; i < ans.size(); i++) {
        cout << ans[i] << " ";
    }
    return 0;
}
```

Time Complexity: $O(\sqrt{N}) + O(K \log K)$ – Iterating \sqrt{N} times, and performing constant time operations in each pass to get all the divisors in the list. Sorting the list of divisors takes $O(K \log K)$ time where K is the number of divisors of the number.

Space Complexity: $O(\sqrt{N})$ – A number N can have at max $2\sqrt{N}$ divisors, which are stored in the array.

Sieve of Eratosthenes

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
```

```

std::vector<int>
primesInRange(std::vector<std::vector<int>>&
queries) {
    if (queries.empty()) {
        return {};
    }
    int maxVal = 0;
    for (const auto& query : queries) {
        maxVal = std::max(maxVal, query[1]);
    }
    std::vector<bool> isPrime(maxVal + 1, true);
    isPrime[0] = isPrime[1] = false;
    for (int p = 2; p * p <= maxVal; ++p) {
        if (isPrime[p]) {
            for (int i = p * p; i <= maxVal; i += p) {
                isPrime[i] = false;
            }
        }
    }
    std::vector<int> primeCount(maxVal + 1, 0);
    for (int i = 1; i <= maxVal; ++i) {
        primeCount[i] = primeCount[i - 1];
        if (isPrime[i]) {
            primeCount[i]++;
        }
    }
    std::vector<int> result;
    for (const auto& query : queries) {
        int start = query[0];
        int end = query[1];
        if (start == 0) {
            result.push_back(primeCount[end]);
        } else {
            result.push_back(primeCount[end] -
primeCount[start - 1]);
        }
    }
}

```

```

return result;
}

};

int main() {
    Solution solution;
    auto result = solution.primesInRange(queries);

    cout << "The number of primes in the given
ranges are: ";

    for (int count : result) {
        cout << count << " ";
    }

    cout << endl;

    return 0;
}

```

Time Complexity: $O(N \log(\log N))$ for the Sieve of Eratosthenes, where n is the maximum value in the queries. The prefix sum computation takes $O(n)$, and each query is processed in $O(1)$ time. Thus, the overall time complexity is dominated by the sieve, resulting in $O(N \log(\log N))$.

Space Complexity: $O(N)$ for storing the prime status array and the prefix sum array, where n is the maximum value in the queries. The space complexity is primarily due to the storage of these arrays, with each requiring space proportional to the maximum value.

Find Prime Factorisation of a Number using Sieve

```

#include <bits/stdc++.h>

using namespace std;

#define MAX_N 100000

vector<int> SPF(MAX_N + 1, 1);

class Solution {
private:
    void sieve() {
        for (int i = 2; i <= MAX_N; i++) {
            if (SPF[i] == 1) {
                for (int j = i; j <= MAX_N; j += i) {
                    if (SPF[j] == 1) {
                        SPF[j] = i;
                    }
                }
            }
        }
    }
}

```

```

    }
    }
}

return;
}

vector<int> primeFact(int n) {
    vector<int> ans;
    while (n != 1) {
        ans.push_back(SPF[n]);
        n = n / SPF[n];
    }
    return ans;
}

public:
    vector<vector<int>> primeFactors(vector<int>& queries){
        sieve();
        vector<vector<int>> ans;
        for(int i=0; i < queries.size(); i++) {
            ans.push_back(primeFact(queries[i]));
        }
        return ans;
    }
};

int main() {
    vector<int> queries = {2, 3, 4, 5, 6};
    Solution sol;
    vector<vector<int>> ans =
sol.primeFactors(queries);

    cout << "The prime factorization of all the
numbers is: " << endl;

    for(int i=0; i < ans.size(); i++) {
        cout << "For " << queries[i] << ": ";
        for(int j=0; j < ans[i].size(); j++) {
            cout << ans[i][j] << " ";
        }
        cout << endl;
    }
}

```

```

    }
    return 0;
}

```

Time Complexity: $O(\max_N \times \log(\log(\max_N))) + O(N \times \log(\text{num}))$, where N represents the number of queries, num represents the average number in the queries, and $\max_N = 10^5$.

The Sieve of Eratosthenes takes $O(\max_N \times \log(\log(\max_N)))$ to precompute the smallest prime factor for all numbers up to \max_N .

In the worst case, finding the prime factors of a number num takes $O(\log(\text{num}))$ time, as the number is divided by its smallest prime factor until it becomes 1.

For N queries, the total time taken will be $O(N \times \log(\text{num}))$.

Space Complexity: $O(\max_N) + O(N \times \log(\text{num}))$, where the SPF array takes $O(\max_N)$ space and the space taken by the list to store the result is $O(N \times \log(\text{num}))$.

Power (n, x)

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    double myPow(double x, int n) {
        if (n == 0 || x == 1.0) return 1;

        long long temp = n;

        if (n < 0) {
            x = 1 / x;
            temp = -1 * 1LL * n;
        }

        double ans = 1;

        for (long long i = 0; i < temp; i++) {
            ans *= x;
        }

        return ans;
    }
};

int main() {
    Solution sol;

    printf("%.4f\n", sol.myPow(2.0000, 10));
}

```

```

    printf("%.4f\n", sol.myPow(2.0000, -2));

    return 0;
}

```

Time Complexity: $O(n)$, where n is the exponent. The loop runs n times to compute the power.

Space Complexity: $O(1)$, as the algorithm uses a constant amount of extra space regardless of the input size.

Optimal Approach

```

#include<bits/stdc++.h>

using namespace std;

class Solution {
private:
    double power(double x, long n) {
        if (n == 0) return 1.0;
        if (n == 1) return x;
        if (n % 2 == 0) {
            return power(x * x, n / 2);
        }
        return x * power(x, n - 1);
    }
public:
    double myPow(double x, int n) {
        int num = n;
        if (num < 0) {
            return (1.0 / power(x, -1 * num));
        }
        return power(x, num);
    }
};

int main() {
    Solution sol;
    double x = 2.0;
    int n = 10;
    double result = sol.myPow(x, n);

    std::cout << x << "^" << n << " = " << result <<
    std::endl;

    return 0;
}

```

```

}

```

Time Complexity: $(\log N)$ due to the halving of n in the even case and linear reduction in the odd case.

Space Complexity: $O(\log n)$ because of the recursive call stack depth.

Implement Stack using Array

```

#include <bits/stdc++.h>

using namespace std;

class ArrayStack {
private:
    int* stackArray;
    int capacity;
    int topIndex;
public:
    ArrayStack(int size = 1000) {
        capacity = size;
        stackArray = new int[capacity];
        topIndex = -1;
    }
    ~ArrayStack() {
        delete[] stackArray;
    }
    void push(int x) {
        if (topIndex >= capacity - 1) {
            cout << "Stack overflow" << endl;
            return;
        }
        stackArray[++topIndex] = x;
    }
    int pop() {
        if (isEmpty()) {
            cout << "Stack is empty" << endl;
            return -1;
        }
        return stackArray[topIndex--];
    }
}

```

```

int top() {
    if (isEmpty()) {
        cout << "Stack is empty" << endl;
        return -1;
    }
    return stackArray[topIndex];
}

bool isEmpty() {
    return topIndex == -1;
}

};

int main() {
    ArrayStack stack;

    vector<string> commands = {"ArrayStack",
    "push", "push", "top", "pop", "isEmpty"};

    vector<vector<int>> inputs = {{}, {5}, {10}, {},
    {}, {}};

    for (size_t i = 0; i < commands.size(); ++i) {
        if (commands[i] == "push") {
            stack.push(inputs[i][0]);
            cout << "null ";
        } else if (commands[i] == "pop") {
            cout << stack.pop() << " ";
        } else if (commands[i] == "top") {
            cout << stack.top() << " ";
        } else if (commands[i] == "isEmpty") {
            cout << (stack.isEmpty() ? "true" : "false")
            << " ";
        } else if (commands[i] == "ArrayStack") {
            cout << "null ";
        }
    }

    return 0;
}

```

Time Complexity: O(1) for all operations (push, pop, top, isEmpty).

Space Complexity: O(N), where N is the maximum capacity of the stack, as we are using an array to store the elements.

Implement Queue Using Array

```

#include <bits/stdc++.h>

using namespace std;

class ArrayQueue {
    int* arr;

    int start, end;

    int currSize, maxSize;

public:
    ArrayQueue() {
        arr = new int[10];

        start = -1;

        end = -1;

        currSize = 0;

        maxSize = 10;
    }

    void push(int x) {
        if (currSize == maxSize) {
            cout << "Queue is full\nExiting..." << endl;
            exit(1);
        }

        if (end == -1) {
            start = 0;

            end = 0;
        }

        else {
            end = (end + 1) % maxSize;
        }

        arr[end] = x;

        currSize++;
    }

    int pop() {
        if (start == -1) {
            cout << "Queue Empty\nExiting..." << endl;
            exit(1);
        }
    }
}

```

```

int popped = arr[start];
if (currSize == 1) {
    start = -1;
    end = -1;
}
else {
    start = (start + 1) % maxSize;
}
currSize--;
return popped;
}

int peek() {
    if (start == -1) {
        cout << "Queue is Empty" << endl;
        exit(1);
    }
    return arr[start];
}

bool isEmpty() {
    return (currSize == 0);
}
};

int main() {
    ArrayQueue queue;
    vector<string> commands = {"ArrayQueue",
    "push", "push",
        "peek", "pop", "isEmpty"};
    vector<vector<int>> inputs = {{}, {5}, {10}, {},
    {}, {}};
    for (int i = 0; i < commands.size(); ++i) {
        if (commands[i] == "push") {
            queue.push(inputs[i][0]);
            cout << "null ";
        } else if (commands[i] == "pop") {
            cout << queue.pop() << " ";
        } else if (commands[i] == "peek") {
            cout << queue.peek() << " ";

```

```

        } else if (commands[i] == "isEmpty") {
            cout << (queue.isEmpty() ? "true" : "false")
            << " ";
        } else if (commands[i] == "ArrayQueue") {
            cout << "null ";
        }
    }
    return 0;
}

```

Time Complexity: O(1) for all operations (push, pop, peek, isEmpty) since they involve simple index manipulations and array accesses.

Space Complexity: O(1) since we are using a fixed-size array and a few variables.

Implement Stack using single Queue

```

#include <bits/stdc++.h>

using namespace std;

class QueueStack {
    queue<int> q;
public:
    void push(int x) {
        int s = q.size();
        q.push(x);
        for (int i = 0; i < s; i++) {
            q.push(q.front());
            q.pop();
        }
    }

    int pop() {
        int n = q.front();
        q.pop();
        return n;
    }

    int top() {
        return q.front();
    }
}

```

```

bool isEmpty() {
    return q.empty();
}

};

int main() {
    QueueStack st;

    vector<string> commands = {"QueueStack",
    "push", "push",
        "pop", "top", "isEmpty"};

    vector<vector<int>> inputs = {{}, {4}, {8}, {},
    {}, {}};

    for (int i = 0; i < commands.size(); ++i) {
        if (commands[i] == "push") {
            st.push(inputs[i][0]);
            cout << "null ";
        } else if (commands[i] == "pop") {
            cout << st.pop() << " ";
        } else if (commands[i] == "top") {
            cout << st.top() << " ";
        } else if (commands[i] == "isEmpty") {
            cout << (st.isEmpty() ? "true" : "false") << "
";
        } else if (commands[i] == "QueueStack") {
            cout << "null ";
        }
    }

    return 0;
}

```

Time Complexity:

Push operation: $O(n)$ (where n is the number of elements in the queue at that time) because every time an element is pushed, all the elements in the queue are popped from the front and pushed in the back again.

Pop operation: $O(1)$ as constant operations are performed.

Top operation: $O(1)$ as constant operations are performed.

IsEmpty operation: $O(1)$ as constant operations are performed.

Space Complexity: $O(k)$ for storing k elements in the queue.

Implement Queue using Stack

Using two Stacks where push operation is $O(N)$

```

#include <bits/stdc++.h>

using namespace std;

class StackQueue {
private:
    stack<int> st1, st2;
public:
    StackQueue () {
    }

    void push(int x) {
        while (!st1.empty()) {
            st2.push(st1.top());
            st1.pop();
        }
        st1.push(x);
        while (!st2.empty()) {
            st1.push(st2.top());
            st2.pop();
        }
    }

    int pop() {
        if (st1.empty()) {
            cout << "Stack is empty";
            return -1;
        }

        int topElement = st1.top();
        st1.pop();
        return topElement;
    }

    int peek() {
        if (st1.empty()) {
            cout << "Stack is empty";
            return -1;
        }
    }
}

```

```

        return st1.top();
    }

    bool isEmpty() {
        return st1.empty();
    }
};

int main() {
    StackQueue q;

    vector<string> commands = {"StackQueue",
    "push", "push",
        "pop", "peek", "isEmpty"};

    vector<vector<int>> inputs = {{}, {4}, {8}, {},
    {}, {}};

    for (int i = 0; i < commands.size(); ++i) {
        if (commands[i] == "push") {
            q.push(inputs[i][0]);
            cout << "null ";
        } else if (commands[i] == "pop") {
            cout << q.pop() << " ";
        } else if (commands[i] == "peek") {
            cout << q.peek() << " ";
        } else if (commands[i] == "isEmpty") {
            cout << (q.isEmpty() ? "true" : "false") << "
";
        } else if (commands[i] == "StackQueue") {
            cout << "null ";
        }
    }

    return 0;
}

```

Time Complexity: $O(n)$ for push operation, $O(1)$ for pop and peek operations.

Space Complexity: $O(n)$ for storing elements in the stacks.

Using Two Stacks Where Push Operation is $O(1)$

```

#include <bits/stdc++.h>

using namespace std;

class StackQueue {

```

```

public:
    stack<int> input, output;
    StackQueue() {}

    void push(int x) {
        input.push(x);
    }

    int pop() {
        if (output.empty()) {
            while (!input.empty()) {
                output.push(input.top());
                input.pop();
            }
        }
        if (output.empty()) {
            cout << "Queue is empty, cannot pop." <<
endl;
            return -1;
        }
        int x = output.top();
        output.pop();
        return x;
    }

    int peek() {
        if (output.empty()) {
            while (!input.empty()) {
                output.push(input.top());
                input.pop();
            }
        }
        if (output.empty()) {
            cout << "Queue is empty, cannot peek." <<
endl;
            return -1;
        }
        return output.top();
    }

    bool isEmpty() {

```

```

        return input.empty() && output.empty();
    }
};

int main() {
    StackQueue q;
    q.push(3);
    q.push(4);
    cout << "The element popped is " << q.pop() << endl;
    q.push(5);
    cout << "The front of the queue is " << q.peek() << endl;
    cout << "Is the queue empty? " << (q.isEmpty() ? "Yes" : "No") << endl;
    cout << "The element popped is " << q.pop() << endl;
    cout << "The element popped is " << q.pop() << endl;
    cout << "Is the queue empty? " << (q.isEmpty() ? "Yes" : "No") << endl;
    return 0;
}

```

Time Complexity: **O(1)** for push operation, **O(n)** for pop and peek operations in the worst case when elements need to be shifted.

Space Complexity: **O(n)** for storing elements in the two stacks.

Implement stack using linked list

```

#include <bits/stdc++.h>

using namespace std;

struct Node {
    int val;
    Node *next;
    Node(int d) {
        val = d;
        next = NULL;
    }
};

class LinkedListStack {

```

```

private:
    Node *head;
    int size;
public:
    LinkedListStack() {
        head = NULL;
        size = 0;
    }
    void push(int x) {
        Node *element = new Node(x);
        element->next = head;
        head = element;
        size++;
    }
    int pop() {
        if (head == NULL) {
            return -1; /
        }
        int value = head->val;
        Node *temp = head;
        head = head->next;
        delete temp;
        size--;
        return value;
    }
    int top() {
        if (head == NULL) {
            return -1;
        }
        return head->val;
    }
    bool isEmpty() {
        return (size == 0);
    }
};

int main() {

```

```

LinkedListStack st;

vector<string> commands = {"LinkedListStack",
"push", "push",

        "pop", "top", "isEmpty"};

vector<vector<int>> inputs = {{}, {3}, {7}, {},
{}, {} };

for (int i = 0; i < commands.size(); ++i) {

    if (commands[i] == "push") {

        st.push(inputs[i][0]);

        cout << "null ";

    } else if (commands[i] == "pop") {

        cout << st.pop() << " ";

    } else if (commands[i] == "top") {

        cout << st.top() << " ";

    } else if (commands[i] == "isEmpty") {

        cout << (st.isEmpty() ? "true" : "false") << "
";

    } else if (commands[i] == "LinkedListStack")

    {

        cout << "null ";

    }

}

return 0;
}

```

Time Complexity: $O(1)$ for push, pop, size, isEmpty, peek operations.

Space Complexity: $O(N)$ because the stack requires space proportional to the number of elements it stores.

Implement Queue using Linked List

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```

struct Node {

    int val;

    Node *next;

    Node(int d) {

        val = d;

        next = NULL;

    }

}

```

```

};

class LinkedListQueue {

private:

    Node *start;

    Node *end;

    int size;

public:

    LinkedListQueue() {

        start = end = NULL;

        size = 0;

    }

    void push(int x) {

        Node *element = new Node(x);

        if(start == NULL) {

            start = end = element;

        }

        else {

            end->next = element;

            end = element;

        }

        size++;

    }

    int pop() {

        if (start == NULL) {

            return -1;

        }

        int value = start->val;

        Node *temp = start;

        start = start->next;

        delete temp;

        size--;

        return value;

    }

    int peek() {

        if (start == NULL) {

            return -1;

        }

    }

}

```

```

    }
    return start->val;
}

bool isEmpty() {
    return (size == 0);
}

};

int main() {
    LinkedListQueue q;

    vector<string> commands =
{"LinkedListQueue", "push", "push",
    "peek", "pop", "isEmpty"};

    vector<vector<int>> inputs = {{}, {3}, {7}, {},
    {}, {}};

    for (int i = 0; i < commands.size(); ++i) {
        if (commands[i] == "push") {
            q.push(inputs[i][0]);
            cout << "null ";
        } else if (commands[i] == "pop") {
            cout << q.pop() << " ";
        } else if (commands[i] == "peek") {
            cout << q.peek() << " ";
        } else if (commands[i] == "isEmpty") {
            cout << (q.isEmpty() ? "true" : "false") << "
";
        } else if (commands[i] == "LinkedListQueue")
        {
            cout << "null ";
        }
    }

    return 0;
}

```

Time Complexity: $O(1)$ for all operations (push, pop, peek, isEmpty) since they involve constant time operations like pointer manipulation and value retrieval.
Space Complexity: $O(n)$ where n is the number of elements in the queue, as we are using a linked list to store the elements. Each element requires space for a node structure.

Check for Balanced Parentheses

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool isValid(string s) {
        stack<char> st;

        for (auto it : s) {
            if (it == '(' || it == '{' || it == '[')
                st.push(it);
            else {
                if (st.empty()) return false;
                char ch = st.top();
                st.pop();
                if ((it == ')' && ch == '(') ||
                    (it == ']' && ch == '[') ||
                    (it == '}' && ch == '{'))
                    continue;
                else
                    return false;
            }
        }

        return st.empty();
    }
};

int main() {
    Solution sol;

    string s = "O[{}O]";

    if (sol.isValid(s))
        cout << "True" << endl;
    else
        cout << "False" << endl;

    return 0;
}

```

Time Complexity: $O(n)$. Single for loop used

Space Complexity: $O(N)$. Stack space

Implement Min Stack : $O(2N)$ and $O(N)$ Space Complexity

Brute Force

```
#include <bits/stdc++.h>

using namespace std;

class MinStack {
private:
    stack <pair<int,int>> st;
public:
    MinStack() {
    }

    void push(int value) {
        if(st.empty()) {
            st.push( {value, value} );
            return;
        }

        int mini = min(getMin(), value);
        st.push({value, mini});
    }

    void pop() {
        st.pop();
    }

    int top() {
        return st.top().first;
    }

    int getMin() {
        return st.top().second;
    }
};

int main() {
    MinStack s;

    s.push(-2);
    s.push(0);
    s.push(-3);

    cout << s.getMin() << " ";
```

```
s.pop();

cout << s.top() << " ";

s.pop();

cout << s.getMin();

return 0;
}
```

Time Complexity: $O(1)$ for all operations (push, pop, top, getMin) as they involve constant time operations on the stack.

Space Complexity: $O(n)$ where n is the number of elements in the stack, as we store pairs of values (element and minimum) in the stack.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class MinStack {
private:
    stack <int> st;
    int mini;
public:
    MinStack() {
    }

    void push(int value) {
        if(st.empty()) {
            mini = value;
            st.push( value );
            return;
        }

        if(value > mini) {
            st.push(value);
        }

        else {
            st.push(2 * value - mini);
            mini = value;
        }
    }

    void pop() {
        if(st.empty()) return;
```

```

int x = st.top();

st.pop();

if(x < mini) {
    mini = 2 * mini - x;
}

}

int top() {
    if(st.empty()) return -1;

    int x = st.top();

    if(mini < x) return x;

    return mini;
}

int getMin() {
    return mini;
}

};

int main() {
    MinStack s;

    s.push(-2);
    s.push(0);
    s.push(-3);

    cout << s.getMin() << " ";

    s.pop();

    cout << s.top() << " ";

    s.pop();

    cout << s.getMin();

    return 0;
}

```

Time Complexity: $O(1)$ for all operations (push, pop, top, getMin) as they involve constant time operations on the stack.

Space Complexity: $O(n)$ where n is the number of elements in the stack, as we store pairs of values (element and minimum) in the stack.

Infix to Postfix

```

#include<bits/stdc++.h>

using namespace std;

```

```

int prec(char c) {
    if (c == '^')
        return 3;
    else if (c == '/' || c == '*')
        return 2;
    else if (c == '+' || c == '-')
        return 1;
    else
        return -1;
}

void infixToPostfix(string s) {
    stack<char> st;
    string result;
    for (int i = 0; i < s.length(); i++) {
        char c = s[i];

        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9'))
            result += c;
        else if (c == '(')
            st.push('(');
        else if (c == ')') {
            while (st.top() != '(') {
                result += st.top();
                st.pop();
            }
            st.pop();
        }
        else {
            while (!st.empty() && prec(s[i]) <= prec(st.top())) {
                result += st.top();
                st.pop();
            }
            st.push(c);
        }
    }

    while (!st.empty()) {

```

```

        result += st.top();

        st.pop();
    }

    cout << "Postfix expression: " << result << endl;
}

int main() {
    string exp = "(p+q)*(m-n)";

    cout << "Infix expression: " << exp << endl;

    infixToPostfix(exp);

    return 0;
}

```

Time Complexity: $O(N)$, where N is the length of the infix expression. Each character in the expression is processed once.

Space Complexity: $O(N)$, where N is the length of the infix expression. The stack can hold at most N operators and parentheses in the worst case.

Prefix to Infix Conversion

```

#include <bits/stdc++.h>

using namespace std;

string prefixToInfix(string prefix) {
    stack<string> s;

    int n = prefix.size();

    for (int i = n - 1; i >= 0; i--) {
        char c = prefix[i];

        if (isalnum(c)) {
            s.push(string(1, c));
        } else {
            string op1 = s.top(); s.pop();

            string op2 = s.top(); s.pop();

            s.push("(" + op1 + c + op2 + ")");
        }
    }
}

```

Time Complexity: $O(n)$ where n is the length of the prefix expression (only one pass).

Space Complexity: $O(n)$ for the stack used to store operands.

Prefix to Postfix Conversion

```

#include <bits/stdc++.h>

using namespace std;

string prefixToPostfix(string prefix) {
    stack<string> s;

    int n = prefix.size();

    for (int i = n - 1; i >= 0; i--) {
        char c = prefix[i];

        if (isalnum(c)) {
            s.push(string(1, c));
        } else {
            string op1 = s.top(); s.pop();

            string op2 = s.top(); s.pop();

            s.push(op1 + op2 + c);
        }
    }

    return s.top();
}

int main() {
    string prefix = "*-A/BC-/AKL";

    cout << "Postfix Expression: " <<
    prefixToPostfix(prefix) << endl;

    return 0;
}

```

Time Complexity: $O(n)$, single pass through the expression.

Space Complexity: $O(n)$, stack space for storing intermediate results.

Postfix to Prefix Conversion

```

#include <bits/stdc++.h>

using namespace std;

string postfixToPrefix(string postfix) {
    stack<string> s;

    int n = postfix.size();

    for (int i = 0; i < n; i++) {
        char c = postfix[i];
    }
}

```

```

        if (isalnum(c)) {
            s.push(string(1, c));
        } else {
            string op2 = s.top(); s.pop();
            string op1 = s.top(); s.pop();
            s.push(c + op1 + op2);
        }
    }
    return s.top();
}

```

```

int main() {
    string postfix = "ABC/-AK/L-*";

    cout << "Prefix Expression: " <<
    postfixToPrefix(postfix) << endl;

    return 0;
}

```

Time Complexity: $O(n)$, as we traverse the expression only once.
Space Complexity: $O(n)$ for the stack to store operands and intermediate results.

Postfix to Infix

```

#include <iosbits/stdc++.h>
using namespace std;
string postfixToInfix(string postfix) {
    stack<string> s;
    int n = postfix.size();
    for (int i = 0; i < n; i++) {
        char c = postfix[i];
        if (isalnum(c)) {
            s.push(string(1, c));
        } else {
            string op2 = s.top(); s.pop();
            string op1 = s.top(); s.pop();
            s.push("(" + op1 + c + op2 + ")");
        }
    }
}

```

```

        return s.top();
    }
    int main() {
        string postfix = "AB*C+";

        cout << "Infix Expression: " <<
        postfixToInfix(postfix) << endl;

        return 0;
    }

```

Time Complexity: $O(n)$, a single pass over the postfix expression.

Space Complexity: $O(n)$, stack space for storing operands.

Infix to Prefix

```

#include <bits/stdc++.h>
using namespace std;
bool isOperator(char c) {
    return (!isalpha(c) && !isdigit(c));
}
int getPriority(char C) {
    if (C == '-' || C == '+')
        return 1;
    else if (C == '*' || C == '/')
        return 2;
    else if (C == '^')
        return 3;
    return 0;
}
string infixToPostfix(string infix) {
    infix = '(' + infix + ')';
    int l = infix.size();
    stack<char> char_stack;
    string output;
    for (int i = 0; i < l; i++) {
        if (isalpha(infix[i]) || isdigit(infix[i]))
            output += infix[i];
        else if (infix[i] == '(')

```

```

        char_stack.push('(');
    else if (infix[i] == ')') {
        while (char_stack.top() != '(') {
            output += char_stack.top();
            char_stack.pop();
        }
        char_stack.pop();
    }
    else {
        if (isOperator(char_stack.top())) {
            if (infix[i] == '^') {
                while (getPriority(infix[i]) <=
getPriority(char_stack.top())) {
                    output += char_stack.top();
                    char_stack.pop();
                }
            } else {
                while (getPriority(infix[i]) <
getPriority(char_stack.top())) {
                    output += char_stack.top();
                    char_stack.pop();
                }
            }
            char_stack.push(infix[i]);
        }
    }
}

while (!char_stack.empty()) {
    output += char_stack.top();
    char_stack.pop();
}

return output;
}

string infixToPrefix(string infix) {
    int l = infix.size();
    reverse(infix.begin(), infix.end());
    for (int i = 0; i < l; i++) {

```

```

        if (infix[i] == '(') {
            infix[i] = ')';
            i++;
        } else if (infix[i] == ')') {
            infix[i] = '(';
            i++;
        }
    }

    string prefix = infixToPostfix(infix);
    reverse(prefix.begin(), prefix.end());
    return prefix;
}

int main() {
    string s = "(p+q)*(c-d)"; // Infix expression
    cout << "Infix expression: " << s << endl;

    cout << "Prefix Expression: " <<
infixToPrefix(s) << endl;

    return 0;
}

```

Time Complexity: $O(N)$, where N is the length of the infix expression. Each character is processed once.

Space Complexity: $O(N)$, where N is the length of the infix expression. The stack can hold at most N characters in the worst case.

Next Greater Element Using Stack

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> nextGreater(vector<int>& nums) {
        // Stack to store elements
        stack<int> st;
        int n = nums.size();
        vector<int> res(n);
        for (int i = n - 1; i >= 0; i--) {
            while (!st.empty() && st.top() <= nums[i])
                st.pop();
            if (st.empty())
                res[i] = -1;
            else
                res[i] = st.top();
            st.push(nums[i]);
        }
        return res;
    }
};

```

```

        st.pop();
    }
    if (st.empty()) res[i] = -1;
    else res[i] = st.top();
    st.push(nums[i]);
}
return res;
}
};

int main() {
    vector<int> nums = {4, 5, 2, 10};
    Solution sol;
    vector<int> ans = sol.nextGreater(nums);
    for (int x : ans) {
        cout << x << " ";
    }
    cout << endl;
    return 0;
}

```

Time Complexity: $O(N)$, we traverse the entire array once and find next greater element in linear time.
Space Complexity: $O(N)$, additional space used for resultant array and stack.

Next Greater Element – 2

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> nextGreaterElements(vector<int>
arr) {
        int n = arr.size();
        vector<int> ans(n, -1);
        for(int i=0; i < n; i++) {
            int currEle = arr[i];
            for(int j=1; j < n; j++) {
                int ind = (j+i) % n;

```

```

                if(arr[ind] > currEle) {
                    ans[i] = arr[ind];
                    break;
                }
            }
        }
        return ans;
    }
};

int main() {
    int n = 6;
    vector<int> arr = {5, 7, 1, 7, 6, 0};
    Solution sol;
    vector<int> ans = sol.nextGreaterElements(arr);
    cout << "The next greater elements are: ";
    for(int i=0; i < n; i++) {
        cout << ans[i] << " ";
    }
    return 0;
}

```

Time Complexity: $O(N^2)$, since we are using two nested for loops to find the next greater elements.
Space Complexity: $O(N)$, The space required to store the answer is $O(N)$.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> nextGreaterElements(vector<int>
arr) {
        int n = arr.size(); //size of array
        vector<int> ans(n);
        stack<int> st;
        for(int i = 2*n-1; i >= 0; i--) {
            int ind = i % n;
            int currEle = arr[ind];

```

```

        while(!st.empty() && st.top() <= currEle) {
            st.pop();
        }
        if(i < n) {
            if(st.empty())
                ans[i] = -1;
            else
                ans[i] = st.top();
        }
        st.push(currEle);
    }
    return ans;
}
};

int main() {
    int n = 6;
    vector<int> arr = {5, 7, 1, 7, 6, 0};
    Solution sol;
    vector<int> ans = sol.nextGreaterElements(arr);
    cout << "The next greater elements are: ";
    for(int i=0; i < n; i++) {
        cout << ans[i] << " ";
    }
    return 0;
}

```

Time Complexity: $O(N)$, since traversing on the array takes $O(N)$ time and traversing the stack will take overall $O(N)$ time as all the elements are pushed in the stack once.

Space Complexity: $O(N)$, since the answer array takes $O(N)$ space and the space used by stack will be $O(N)$ in the worst case.

Next Smaller Element

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

class Solution {

```

```

public:
    vector<int> nextSmallerElement(vector<int>&
arr) {
        int n = arr.size();
        vector<int> ans(n, -1);
        for (int i = 0; i < n; ++i) {
            int currEle = arr[i];
            for (int j = i + 1; j < n; ++j) {
                if (arr[j] < currEle) {
                    ans[i] = arr[j];
                    break;
                }
            }
        }
        return ans;
    }
};

int main() {
    int n = 5;
    vector<int> arr = {4, 8, 5, 2, 25};
    Solution sol;
    vector<int> ans = sol.nextSmallerElement(arr);
    cout << "The next smaller elements are: ";
    for (int i = 0; i < n; ++i) {
        cout << ans[i] << " ";
    }

    return 0;
}

```

Time Complexity: $O(N^2)$, since for each of the N elements, we might need to look at up to $N-1$ elements ahead.

Space Complexity: $O(N)$, since we are using an output array of size N .

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {

```

```

public:
    vector<int> nextSmallerElement(vector<int>&
arr) {
        int n = arr.size();
        stack<int> st;
        vector<int> ans(n, -1);
        for (int i = n - 1; i >= 0; i--) {
            while (!st.empty() && st.top() >= arr[i]) {
                st.pop();
            }
            if (!st.empty()) {
                ans[i] = st.top();
            }
            st.push(arr[i]);
        }
        return ans;
    }
};

int main() {
    vector<int> arr = {1, 3, 2, 4};
    Solution sol;
    vector<int> ans = sol.nextSmallerElement(arr);
    cout << "The next smaller elements are: ";
    for (int val : ans) {
        cout << val << " ";
    }
    return 0;
}

```

Time Complexity: $O(N)$, since each element is pushed and popped at most once.
Space Complexity: $O(N)$, since stack may hold up to N elements in the worst case.

Number of NGEs to the right

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

```

```

class Solution {
public:
    vector<int> nextLargerElement(vector<int> arr)
    {
        int n = arr.size();
        vector<int> ans(n, -1);
        for(int i=0; i < n; i++) {
            int currEle = arr[i];
            for(int j=i+1; j < n; j++) {
                if(arr[j] > currEle) {
                    ans[i] = arr[j];
                    break;
                }
            }
        }
        return ans;
    }
};

int main() {
    int n = 4;
    vector<int> arr = {1, 3, 2, 4};
    Solution sol;
    vector<int> ans = sol.nextLargerElement(arr);
    cout << "The next greater elements are: ";
    for(int i=0; i < n; i++) {
        cout << ans[i] << " ";
    }
    return 0;
}

```

Time Complexity: $O(N^2)$ (where N is the size of the given array). Using two nested for loops to find the next greater elements.

Space Complexity: $O(N)$, The space required to store the answer is $O(N)$.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {

```

public:

```
vector<int> nextLargerElement(vector<int> arr)
{
    int n = arr.size();
    vector<int> ans(n);
    stack<int> st;
    for(int i=n-1; i >= 0; i--) {
        int currEle = arr[i];
        while(!st.empty() && st.top() <= currEle) {
            st.pop();
        }
        if(st.empty())
            ans[i] = -1;
        else
            ans[i] = st.top();
        st.push(currEle);
    }
    return ans;
}

int main() {
    int n = 4;
    vector<int> arr = {1, 3, 2, 4};
    Solution sol;
    vector<int> ans = sol.nextLargerElement(arr);
    cout << "The next greater elements are: ";
    for(int i=0; i < n; i++) {
        cout << ans[i] << " ";
    }
    return 0;
}
```

Time Complexity: $O(N)$ (where N is the size of the array). Traversing on the hypothetical array takes $O(2N)$ time and traversing the stack will take overall $O(2N)$ time as all the elements are pushed in the stack once.

Space Complexity: $O(N)$ since the answer array takes $O(N)$ space and the space used by the stack will be $O(N)$ in the worst case.

Trapping Rainwater

Brute Force Approach

```
#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int trap(vector<int>& height) {
        int n = height.size();
        int totalWater = 0;
        for (int i = 0; i < n; i++) {
            int maxLeft = 0;
            int maxRight = 0;
            for (int j = 0; j <= i; j++) {
                if (height[j] > maxLeft) {
                    maxLeft = height[j];
                }
            }
            for (int j = i; j < n; j++) {
                if (height[j] > maxRight) {
                    maxRight = height[j];
                }
            }
            totalWater += min(maxLeft, maxRight) -
                height[i];
        }
        return totalWater;
    }
};

int main() {
    vector<int> height = {0,1,0,2,1,0,1,3,2,1,2,1};
    Solution sol;
    int result = sol.trap(height);
    cout << "Trapped Rainwater: " << result <<
endl;
    return 0;
}
```

Time Complexity: $O(n^2)$ because for each bar, we scan all bars to its left and right to find the maximum height, resulting in nested loops.

Space Complexity: $O(1)$ as no additional data structures are used proportional to input size, only variables to track max heights and total water.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int trap(vector<int>& height) {
        int n = height.size();
        int left = 0;
        int right = n - 1;
        int maxLeft = 0;
        int maxRight = 0;
        int totalWater = 0;
        while (left <= right) {
            if (height[left] <= height[right]) {
                if (height[left] >= maxLeft) {
                    maxLeft = height[left];
                } else {
                    totalWater += maxLeft - height[left];
                }
                left++;
            } else {
                if (height[right] >= maxRight) {
                    maxRight = height[right];
                } else {
                    totalWater += maxRight -
height[right];
                }
                right--;
            }
        }
        return totalWater;
    }
};
```

```
};

int main() {
    vector<int> height = {0,1,0,2,1,0,1,3,2,1,2,1};
    Solution sol;
    int result = sol.trap(height);
    cout << "Trapped Rainwater: " << result <<
endl;
    return 0;
}
```

Time Complexity: $O(n^2)$ because for each bar, we scan all bars to its left and right to find the maximum height, resulting in nested loops.

Space Complexity: $O(1)$ as no additional data structures are used proportional to input size, only variables to track max heights and total water.

Sum of Subarray Minimums

Brute Force

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int sumSubarrayMins(vector<int> &arr) {
        int n = arr.size();
        int mod = 1e9 + 7;
        int sum = 0;
        for(int i=0; i < n; i++) {
            int mini = arr[i];
            for(int j=i; j < n; j++) {
                mini = min(mini, arr[j]);
                sum = (sum + mini) % mod;
            }
        }
        return sum;
    }
};

int main() {
    vector<int> arr = {3, 1, 2, 5};
```

```

Solution sol;

int ans = sol.sumSubarrayMins(arr);

cout << "The sum of minimum value in each
subarray is: " << ans;

return 0;
}

```

Time Complexity: $O(N^2)$, since we are using two nested loops.

Space Complexity: $O(1)$, as we are not using any extra space except for the input array and a few variables.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    vector<int> findNSE(vector<int> &arr) {
        int n = arr.size();
        vector<int> ans(n);
        stack<int> st;
        for(int i = n - 1; i >= 0; i--) {
            int currEle = arr[i];
            while(!st.empty() && arr[st.top()] >=
arr[i]){
                st.pop();
            }
            ans[i] = !st.empty() ? st.top() : n;
            st.push(i);
        }
        return ans;
    }
    vector<int> findPSEE(vector<int> &arr) {
        int n = arr.size();
        vector<int> ans(n);
        stack<int> st;
        for(int i=0; i < n; i++) {
            int currEle = arr[i];
            while(!st.empty() && arr[st.top()] > arr[i]){
                st.pop();
            }

```

```

        }
        ans[i] = !st.empty() ? st.top() : -1;
        st.push(i);
    }
    return ans;
}

```

public:

```

int sumSubarrayMins(vector<int> &arr) {
    vector<int> nse = findNSE(arr);
    vector<int> psee = findPSEE(arr);
    int n = arr.size();
    int mod = 1e9 + 7;
    int sum = 0;
    for(int i=0; i < n; i++) {
        int left = i - psee[i];
        int right = nse[i] - i;
        long long freq = left*right*1LL;
        int val = (freq*arr[i]*1LL) % mod;
        sum = (sum + val) % mod;
    }
    return sum;
}

```

```

};

int main() {
    vector<int> arr = {3, 1, 2, 5};
    Solution sol;
    int ans = sol.sumSubarrayMins(arr);

    cout << "The sum of minimum value in each
subarray is: " << ans;

    return 0;
}

```

Time Complexity: $O(N)$, since finding the indices of next smaller elements and previous smaller elements take $O(2*N)$ time each and calculating the sum of subarrays minimum takes $O(N)$ time.

Space Complexity: $O(N)$, since finding the indices of the next smaller elements and previous smaller elements takes $O(N)$ space each due to stack space and storing the indices of the next smaller elements and previous smaller elements takes $O(N)$ space each.

Asteroid Collision

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> asteroidCollision(vector<int>
&asteroids){
        int n = asteroids.size();
        vector<int> st;

        for(int i=0; i < n; i++) {
            if(asteroids[i] > 0) {
                st.push_back(asteroids[i]);
            }
            else {
                while(!st.empty() && st.back() > 0 &&
                    st.back() < abs(asteroids[i])) {
                    st.pop_back();
                }
                if(!st.empty() &&
                    st.back() == abs(asteroids[i])) {
                    st.pop_back();
                }
                else if(st.empty() ||
                    st.back() < 0){
                    st.push_back(asteroids[i]);
                }
            }
        }
        return st;
    }
};

int main() {
    vector<int> arr = {10, 20, -10};
    Solution sol;
    vector<int> ans = sol.asteroidCollision(arr);
```

```
    cout << "The state of asteroids after collisions is:
";
    for(int i=0; i < ans.size(); i++) {
        cout << ans[i] << " ";
    }

    return 0;
}
```

Time Complexity: $O(N)$, since traversing all the asteroids takes $O(N)$ time.

Space Complexity: $O(N)$, since in the worst case, all asteroids will be stored in the stack if there are no collisions, leading to a space requirement of $O(N)$.

Sum of Subarray Ranges

Brute Force

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    long long subArrayRanges(vector<int> &arr) {
        int n = arr.size();
        long long sum = 0;
        for(int i=0; i < n; i++) {
            int smallest = arr[i];
            int largest = arr[i];
            for(int j=i; j < n; j++) {
                smallest = min(smallest, arr[j]);
                largest = max(largest, arr[j]);
                sum += (largest - smallest);
            }
        }
        return sum;
    }
};

int main() {
    vector<int> arr = {1, 2, 3};
    Solution sol;
    long long ans = sol.subArrayRanges(arr);
```

```

    cout << "The sum of subarray ranges is: " <<
ans;

    return 0;
}

```

Time Complexity: $O(N^2)$, since we are using two nested loops.

Space Complexity: $O(1)$, since we are using only a couple of variables.

Optimal Approach

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Solution {
```

```
private:
```

```

    vector<int> findNSE(vector<int> &arr) {
        int n = arr.size();
        vector<int> ans(n);
        stack<int> st;
        for(int i = n - 1; i >= 0; i--) {
            int currEle = arr[i];
            while(!st.empty() && arr[st.top()] >=
currEle){
                st.pop();
            }
            ans[i] = !st.empty() ? st.top() : n;
            st.push(i);
        }
        return ans;
    }

    vector<int> findNGE(vector<int> &arr) {
        int n = arr.size();
        vector<int> ans(n);
        stack<int> st;
        for(int i = n - 1; i >= 0; i--) {
            int currEle = arr[i];
            while(!st.empty() && arr[st.top()] <=
currEle){
                st.pop();
            }
        }
    }
}

```

```

        ans[i] = !st.empty() ? st.top() : n;
        st.push(i);
    }

    return ans;
}

vector<int> findPSEE(vector<int> &arr) {
    int n = arr.size();
    vector<int> ans(n);
    stack<int> st;
    for(int i=0; i < n; i++) {
        int currEle = arr[i];
        while(!st.empty() && arr[st.top()] >
currEle){
            st.pop();
        }
        ans[i] = !st.empty() ? st.top() : -1;
        st.push(i);
    }
    return ans;
}

vector<int> findPGEE(vector<int> &arr) {
    int n = arr.size();
    vector<int> ans(n);
    stack<int> st;
    for(int i=0; i < n; i++) {
        int currEle = arr[i];
        while(!st.empty() && arr[st.top()] <
currEle){
            st.pop();
        }
        ans[i] = !st.empty() ? st.top() : -1;
        st.push(i);
    }
    return ans;
}

long long sumSubarrayMins(vector<int> &arr) {
    vector<int> nse = findNSE(arr);
}

```

```

vector<int> psee = findPSEE(arr);
int n = arr.size();
long long sum = 0;
for(int i=0; i < n; i++) {
    int left = i - psee[i];
    int right = nse[i] - i;
    long long freq = left*right*1LL;
    long long val = (freq*arr[i]*1LL);
    sum += val;
}
return sum;
}

long long sumSubarrayMaxs(vector<int> &arr) {
    vector<int> nge = findNGE(arr);
    vector<int> pgee = findPGEE(arr);
    int n = arr.size();
    long long sum = 0;
    for(int i=0; i < n; i++) {
        int left = i - pgee[i];
        int right = nge[i] - i;
        long long freq = left*right*1LL;
        long long val = (freq*arr[i]*1LL);
        sum += val;
    }
    return sum;
}

public:
    long long subArrayRanges(vector<int> &arr) {
        return ( sumSubarrayMaxs(arr) -
                sumSubarrayMins(arr) );
    }
};

int main() {
    vector<int> arr = {1, 2, 3};
    Solution sol;
    long long ans = sol.subArrayRanges(arr);

```

```

    cout << "The sum of subarray ranges is: " <<
    ans;

    return 0;
}

```

Time Complexity: $O(N)$, since calculating the sum of subarray maximums takes $O(N)$ time and calculating the sum of subarray minimums takes $O(N)$ time.

Space Complexity: $O(N)$, since calculating the sum of subarray maximums requires $O(N)$ space and calculating the sum of subarray minimums requires $O(N)$ space.

Remove K Digits

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    string removeKdigits(string nums, int k) {
        stack <char> st;
        for(int i=0; i < nums.size(); i++) {
            char digit = nums[i];
            while(!st.empty() && k > 0 && st.top() >
digit) {
                st.pop();
                k--;
            }
            st.push(digit);
        }
        while(!st.empty() && k > 0) {
            st.pop();
            k--;
        }
        if(st.empty()) return "0";
        string res = "";
        while(!st.empty()) {
            res.push_back(st.top());
            st.pop();
        }
        while(res.size() > 0 && res.back() == '0') {
            res.pop_back();
        }
    }
};

```

```

    }
    reverse(res.begin(), res.end());
    if(res.empty()) return "0";
    return res;
}
};

int main() {
    string nums = "541892";
    int k = 2;

    Solution sol;

    string ans = sol.removeKdigits(nums, k);

    cout << "The smallest possible integer after
removing k digits is: " << ans;

    return 0;
}

```

Time Complexity: $O(N)$, since traversing the given string takes $O(N)$ time, each element is pushed onto and popped from the stack at most once in worst-case taking $o(N)$ time, removing the remaining digits (if $k > 0$) takes $O(k)$ time which can go upto $O(N)$ in worst-case and forming the result, trimming the zeros and reversing the digits takes $O(N)$ time.

Space Complexity: $O(N)$, since we are using a stack to store the digits of the resulting number, in the worst case, the stack can contain all the digits of the input string.

Area of largest rectangle in Histogram

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

int largestarea(int arr[], int n) {
    int maxArea = 0;

    for (int i = 0; i < n; i++) {
        int minHeight = INT_MAX;

        for (int j = i; j < n; j++) {
            minHeight = min(minHeight, arr[j]);

            int width = j - i + 1;

            int area = minHeight * width;

            maxArea = max(maxArea, area);
        }
    }
}

```

```

    }
    reverse(res.begin(), res.end());
    if(res.empty()) return "0";
    return res;
}
};

int main() {
    int arr[] = {2, 1, 5, 6, 2, 3, 1};

    int n = 7;

    cout << "The largest area in the histogram is "
<< largestarea(arr, n) << endl;

    return 0;
}

```

Time Complexity: $O(N*N)$, since nested for loops are used

Space Complexity: $O(1)$. No extra space used

Optimised Approach 1

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int largestRectangleArea(vector<int> &heights)
    {
        int n = heights.size();

        stack<int> st;

        int leftsmall[n], rightsmall[n];

        for (int i = 0; i < n; i++) {
            while (!st.empty() && heights[st.top()] >=
heights[i]) {
                st.pop();
            }
            leftsmall[i] = st.empty() ? 0 : st.top() + 1;
            st.push(i);
        }

        while (!st.empty()) st.pop();

        for (int i = n - 1; i >= 0; i--) {
            while (!st.empty() && heights[st.top()] >=
heights[i]) {
                st.pop();
            }
        }
    }
}

```

```

        rightsmall[i] = st.empty() ? n - 1 : st.top() -
1;
        st.push(i);
    }
    int maxA = 0;
    for (int i = 0; i < n; i++) {
        int width = rightsmall[i] - leftsmall[i] + 1;
        maxA = max(maxA, heights[i] * width);
    }
    return maxA;
}
};

int main() {
    vector<int> heights = {2, 1, 5, 6, 2, 3, 1};
    Solution obj;

    cout << "The largest area in the histogram is "
<< obj.largestRectangleArea(heights) << endl;

    return 0;
}

```

Time Complexity: $O(N)$. Single loop at the end using $O(N)$

Space Complexity: $O(3N)$ where 3 is for the stack, left small array and a right small array

Optimised Approach 2

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int largestRectangleArea(vector<int> &histo) {
        stack<int> st;
        int maxA = 0;
        int n = histo.size();
        for (int i = 0; i <= n; i++) {
            while (!st.empty() && (i == n ||
histo[st.top()] >= histo[i])) {
                int height = histo[st.top()];

```

```

                st.pop();
                int width;
                if (st.empty()) {
                    width = i;
                } else {
                    width = i - st.top() - 1;
                }
                maxA = max(maxA, width * height);
            }
            st.push(i);
        }
        return maxA;
    }
};

```

```

int main() {
    vector<int> histo = {2, 1, 5, 6, 2, 3, 1};
    Solution obj;

    cout << "The largest area in the histogram is "
<< obj.largestRectangleArea(histo) << endl;

    return 0;
}

```

Time Complexity: $O(N) + O(N)$. For loop used along with a while loop

Space Complexity: $O(N)$. Used for stack

Maximal Rectangles

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    int largestRectangleArea(vector<int> &heights)
    {
        int n = heights.size();
        stack<int> st;
        int largestArea = 0;
        int area;

```

```

int nse, pse;
for(int i=0; i < n; i++) {
    while(!st.empty() &&
        heights[st.top()] >= heights[i]){
        int ind = st.top();
        st.pop();
        pse = st.empty() ? -1 : st.top();
        nse = i;
        area = heights[ind] * (nse-pse-1);
        largestArea = max(largestArea, area);
    }
    st.push(i);
}
while(!st.empty()) {
    nse = n;
    int ind = st.top();
    st.pop();
    pse = st.empty() ? -1 : st.top();
    area = heights[ind] * (nse-pse-1);
    largestArea = max(largestArea, area);
}
return largestArea;
}

public:
    int
    maximalAreaOfSubMatrixOfAll1(vector<vector<int>> &matrix){
        int n = matrix.size();
        int m = matrix[0].size();
        vector<vector<int>> prefixSum(n,
            vector<int>(m));
        for(int j=0; j < m; j++) {
            int sum = 0;
            for(int i=0; i < n; i++) {
                sum += matrix[i][j];
                if(matrix[i][j] == 0) {
                    prefixSum[i][j] = 0;

```

```

                sum = 0;
            }
            prefixSum[i][j] = sum;
        }
    }
    int maxArea = 0;
    for(int i = 0; i < n; i++) {
        int area =
        largestRectangleArea(prefixSum[i]);
        maxArea = max(area, maxArea);
    }
    return maxArea;
}

};

int main() {
    vector<vector<int>> matrix = {
        {1, 0, 1, 0, 0},
        {1, 0, 1, 1, 1},
        {1, 1, 1, 1, 1},
        {1, 0, 0, 1, 0}
    };
    Solution sol;
    int ans =
        sol.maximalAreaOfSubMatrixOfAll1(matrix);

    cout << "The largest rectangle area containing all
    1s is: " << ans;

    return 0;
}

```

Time Complexity: $O(N*M)$, since filling the prefix sum matrix takes $O(N*M)$ time, and every row (of length M) is treated as a histogram for which the largest histogram is found in linear($O(2*M)$). Thus, time taking overall is $O(N*M)$.

Space Complexity: $O(N*M)$, since the prefix sum array takes up $O(N*M)$ space, and finding the largest rectangle in each histogram (of length M) takes $O(M)$ space due to stack.

Sliding Window Maximum

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>&
nums, int k) {
        vector<int> result;
        for (int i = 0; i <= nums.size() - k; i++) {
            int maxVal = nums[i];
            for (int j = i; j < i + k; j++) {
                maxVal = max(maxVal, nums[j]);
            }
            result.push_back(maxVal);
        }
        return result;
    }
};

int main() {
    Solution obj;
    vector<int> arr = {4, 0, -1, 3, 5, 3, 6, 8};
    int k = 3;
    vector<int> ans = obj.maxSlidingWindow(arr,
k);

    for (int num : ans) {
        cout << num << " ";
    }

    return 0;
}

```

Time Complexity: $O(n * k)$ Each of the $(n - k + 1)$ windows is scanned completely to find its maximum. In worst-case, each window of size k requires $O(k)$ operations.

Space Complexity: $O(1)$ We are only using output list which does not count as extra space in space complexity analysis. No additional data structures used.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

```

```

class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>&
nums, int k) {
        deque<int> dq;
        vector<int> result;
        for (int i = 0; i < nums.size(); i++) {
            if (!dq.empty() && dq.front() <= i - k) {
                dq.pop_front();
            }
            while (!dq.empty() && nums[dq.back()] <
nums[i]) {
                dq.pop_back();
            }
            dq.push_back(i);
            if (i >= k - 1) {
                result.push_back(nums[dq.front()]);
            }
        }
        return result;
    }
};

int main() {
    Solution obj;
    vector<int> arr = {4, 0, -1, 3, 5, 3, 6, 8};
    int k = 3;
    vector<int> ans = obj.maxSlidingWindow(arr,
k);

    for (int num : ans) {
        cout << num << " ";
    }

    return 0;
}

```

Time Complexity: $O(n)$ Each element is pushed and popped from the deque at most once, so overall traversal is linear.

Space Complexity: $O(k)$ Deque stores at most k elements at any time, one for each index in the window.

Stock span problem

Brute Force

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> stockSpan(vector<int> arr, int n) {
        vector<int> ans(n);
        for(int i=0; i < n; i++) {
            int currSpan = 0;
            for(int j=i; j >= 0; j--) {
                if(arr[j] <= arr[i]) {
                    currSpan++;
                }
                else break;
            }
            ans[i] = currSpan;
        }
        return ans;
    };

    int main() {
        int n = 7;
        vector<int> arr = {120, 100, 60, 80, 90, 110, 115};
        Solution sol;
        vector<int> ans = sol.stockSpan(arr, n);
        cout << "The span of stock prices is: ";
        for(int i=0; i < n; i++) {
            cout << ans[i] << " ";
        }
        return 0;
    }
};
```

Time Complexity: $O(N^2)$, since we are using two nested loops.

Space Complexity: $O(1)$, since we are using only a couple of variables.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    vector<int> findPGE(vector<int> arr) {
        int n = arr.size();
        vector<int> ans(n);
        stack<int> st;
        for(int i=0; i < n; i++) {
            int currEle = arr[i];
            while(!st.empty() && arr[st.top()] <= currEle) {
                st.pop();
            }
            if(st.empty())
                ans[i] = -1;
            else
                ans[i] = st.top();
            st.push(i);
        }
        return ans;
    }

public:
    vector<int> stockSpan(vector<int> arr, int n) {
        vector<int> PGE = findPGE(arr);
        vector<int> ans(n);
        for(int i=0; i < n; i++) {
            ans[i] = i - PGE[i];
        }
        return ans;
    }
};
```

```
int main() {
    int n = 7;
    vector<int> arr = {120, 100, 60, 80, 90, 110, 115};
    Solution sol;
    vector<int> ans = sol.stockSpan(arr, n);
    cout << "The span of stock prices is: ";
    for(int i=0; i < n; i++) {
        cout << ans[i] << " ";
    }
    return 0;
}
```

Time Complexity: $O(N)$, since finding the indices of previous greater elements takes $O(N)$ time and we traverse the array once to compute the stock span, that takes $O(N)$ as well.

Space Complexity: $O(N)$, the stack space used to find the previous greater elements can go up to $O(N)$ in the worst case.

Celebrity Problem

Brute Force

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int celebrity(vector<vector<int>> &M){
        int n = M.size();
        vector<int> knowMe(n, 0);
        vector<int> Iknow(n, 0);
        for(int i=0; i < n; i++) {
            for(int j=0; j < n; j++) {
                if(M[i][j] == 1) {
                    knowMe[j]++;
                    Iknow[i]++;
                }
            }
        }
        for(int i=0; i < n; i++) {
```

```
            if(knowMe[i] == n-1 && Iknow[i] == 0) {
                return i;
            }
        }
        return -1;
    }
};
```

```
int main() {
    vector<vector<int>> M = {
        {0, 1, 1, 0},
        {0, 0, 0, 0},
        {1, 1, 0, 0},
        {0, 1, 1, 0}
    };
    Solution sol;
    int ans = sol.celebrity(M);
    cout << "The index of celebrity is: " << ans;
    return 0;
}
```

Time Complexity: $O(N^2)$, since we are using two nested loops to traverse the square matrix to populate the lists.

Space Complexity: $O(N)$, since we are using two lists of size N to store the count of how many people each person knows and how many people know each person

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int celebrity(vector<vector<int>> &M){
        int n = M.size();
        int top = 0, down = n-1;
        while(top < down) {
            if(M[top][down] == 1) {
                top = top + 1;
            }
            else if(M[down][top] == 1) {
                down = down - 1;
            }
        }
        return top;
    }
};
```

```

    }
    else {
        top++;
        down--;
    }
}
if(top > down) return -1;
for(int i=0; i < n; i++) {
    if(i == top) continue;
    if(M[top][i] == 1 || M[i][top] == 0) {
        return -1;
    }
}
return top;
}
};

int main() {
    vector<vector<int>> M = {
        {0, 1, 1, 0},
        {0, 0, 0, 0},
        {1, 1, 0, 0},
        {0, 1, 1, 0}
    };

    Solution sol;

    int ans = sol.celebrity(M);

    cout << "The index of celebrity is: " << ans;

    return 0;
}

```

Time Complexity: $O(N)$, since eliminating persons and checking if the last candidate is a celebrity both take $O(N)$ time.

Space Complexity: $O(1)$, since we are using only a couple of variables.

Implement LRU Cache

```

#include <bits/stdc++.h>

using namespace std;

```

```

class LRUCache {
public:
    class Node {
    public:
        int key;
        int val;
        Node* next;
        Node* prev;
        Node(int _key, int _val) {
            key = _key;
            val = _val;
        }
    };

    Node* head = new Node(-1, -1);
    Node* tail = new Node(-1, -1);
    int cap;
    unordered_map<int, Node*> m;

    LRUCache(int capacity) {
        cap = capacity;
        head->next = tail;
        tail->prev = head;
    }

    void addNode(Node* newNode) {
        Node* temp = head->next;
        newNode->next = temp;
        newNode->prev = head;
        head->next = newNode;
        temp->prev = newNode;
    }

    void deleteNode(Node* delNode) {
        Node* delPrev = delNode->prev;
        Node* delNext = delNode->next;
        delPrev->next = delNext;
        delNext->prev = delPrev;
    }

    int get(int key_) {

```

```

    if (m.find(key_) != m.end()) {
        Node* resNode = m[key_];

        int res = resNode->val;

        m.erase(key_);

        deleteNode(resNode);

        addNode(resNode);

        m[key_] = head->next;

        return res;
    }

    return -1;
}

void put(int key_, int value) {
    if (m.find(key_) != m.end()) {
        Node* existingNode = m[key_];

        m.erase(key_);

        deleteNode(existingNode);
    }

    if (m.size() == cap) {
        m.erase(tail->prev->key);
        deleteNode(tail->prev);
    }

    addNode(new Node(key_, value));

    m[key_] = head->next;
}

};

int main() {
    LRUCache cache(2);

    cache.put(1, 1);

    cache.put(2, 2);

    cout << cache.get(1) << endl;

    cache.put(3, 3);

    cout << cache.get(2) << endl;

    cache.put(4, 4);

    cout << cache.get(1) << endl;

    cout << cache.get(3) << endl;

    cout << cache.get(4) << endl;
}

```

```

        return 0;
    }
}

```

Time Complexity: get() function: $O(1)$, accessing a value in the HashMap is $O(1)$ on average.
 put() function: $O(1)$, checking and removing from the map as well as inserting into the doubly linked list is $O(1)$.

Space Complexity: $O(\text{capacity})$, for storing up to capacity nodes in the doubly linked list and map.

LFU Cache

```

#include <bits/stdc++.h>

using namespace std;

struct Node {
    int key, value, cnt;

    Node *next;

    Node *prev;

    Node(int _key, int _value) {
        key = _key;
        value = _value;
        cnt = 1;
    }
};

struct List {
    int size;

    Node *head;

    Node *tail;

    List() {
        head = new Node(0, 0);

        tail = new Node(0, 0);

        head->next = tail;

        tail->prev = head;

        size = 0;
    }

    void addFront(Node *node) {

```

```

Node* temp = head->next;
node->next = temp;
node->prev = head;
head->next = node;
temp->prev = node;
size++;
}
void removeNode(Node* delnode) {
    Node* prevNode = delnode->prev;
    Node* nextNode = delnode->next;
    prevNode->next = nextNode;
    nextNode->prev = prevNode;
    size--;
}
};
class LFUCache {
private:
    map<int, Node*> keyNode;
    map<int, List*> freqListMap;
    int maxSizeCache; // Max size of cache
    int minFreq;
    int curSize;
public:
    LFUCache(int capacity) {
        maxSizeCache = capacity;
        minFreq = 0;
        curSize = 0;
    }
    void updateFreqListMap(Node *node) {
        keyNode.erase(node->key);
        freqListMap[node->cnt]->removeNode(node);
        if((node->cnt == minFreq &&
            freqListMap[node->cnt]->size == 0) {
            minFreq++;
        }
        List* nextHigherFreqList = new List();
        if(freqListMap.find(node->cnt + 1) !=
            freqListMap.end()) {
            nextHigherFreqList = freqListMap[node-
                >cnt + 1];
        }
        node->cnt += 1;
        nextHigherFreqList->addFront(node);
        freqListMap[node->cnt] = nextHigherFreqList;
        keyNode[node->key] = node;
    }
    int get(int key) {
        if(keyNode.find(key) != keyNode.end()) {
            Node* node = keyNode[key];
            int val = node->value;
            updateFreqListMap(node);
            return val;
        }
        return -1;
    }
    void put(int key, int value) {
        if (maxSizeCache == 0) {
            return;
        }
        if(keyNode.find(key) != keyNode.end()) {
            Node* node = keyNode[key];
            node->value = value;
            updateFreqListMap(node);
        }
        else {
            if(curSize == maxSizeCache) {
                List* list = freqListMap[minFreq];
                keyNode.erase(list->tail->prev->key);
                freqListMap[minFreq]->removeNode(
                    list->tail->prev
                );
                curSize--;
            }

```

```

    }
    curSize++;
    minFreq = 1;
    List* listFreq = new List();
    if(freqListMap.find(minFreq) !=
        freqListMap.end()) {
        listFreq = freqListMap[minFreq];
    }
    Node* node = new Node(key, value);
    listFreq->addFront(node);
    keyNode[key] = node;
    freqListMap[minFreq] = listFreq;
}
};

int main() {
    LFUCache cache(2);
    cache.put(1, 1);
    cache.put(2, 2);
    cout << cache.get(1) << " ";
    cache.put(3, 3);
    cout << cache.get(2) << " ";
    cout << cache.get(3) << " ";
    cache.put(4, 4);
    cout << cache.get(1) << " ";
    cout << cache.get(3) << " ";
    cout << cache.get(4) << " ";
    return 0;
}

```

Time Complexity: $O(N)$, where N is the number of queries on the LFU cache. Each get and put method takes an average of constant time, making the overall complexity $O(N)$.

Space Complexity: $O(\text{cap})$, where cap is the capacity of the LFU cache. The cache can store a maximum of cap data items, taking $O(\text{cap})$ space.

Length of Longest Substring without any Repeating Character

Brute Force

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int longestNonRepeatingSubstring(string &s) {
        int n = s.size();
        int maxLen = 0;
        for (int i = 0; i < n; i++) {
            vector<int> hash(256, 0);
            for (int j = i; j < n; j++) {
                if (hash[s[j]] == 1) break;
                hash[s[j]] = 1;
                int len = j - i + 1;
                maxLen = max(maxLen, len);
            }
        }
        return maxLen;
    }
};

int main() {
    string input = "cadbzabed";
    Solution sol;

    int length =
    sol.longestNonRepeatingSubstring(input);

    cout << "Length of longest substring without
    repeating characters: " << length << endl;

    return 0;
}

```

Time Complexity: $O(n^2)$, where n is the length of the string. This is because we are using a nested loop to check all possible substrings, leading to a quadratic time complexity.

Space Complexity: $O(1)$, as we are using a fixed-size hash array of size 256 (for extended ASCII characters) and not using any additional data structures that grow with input size.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

```

```

class Solution {
public:
    int longestNonRepeatingSubstring(string& s) {
        int n = s.size();
        int HashLen = 256;
        int hash[HashLen];
        for (int i = 0; i < HashLen; ++i) {
            hash[i] = -1;
        }
        int l = 0, r = 0, maxLen = 0;
        while (r < n) {
            if (hash[s[r]] != -1) {
                l = max(hash[s[r]] + 1, l);
            }
            int len = r - l + 1;
            maxLen = max(len, maxLen);
            hash[s[r]] = r;
            r++;
        }
        return maxLen;
    }
};

int main() {
    string s = "cadbzabed";
    Solution sol;
    int result =
sol.longestNonRepeatingSubstring(s);
    cout << "The maximum length is:\n";
    cout << result << endl;
    return 0;
}

```

Time Complexity: $O(n)$, where n is the length of the string. This is because we are using a single pass through the string with two pointers, leading to linear time complexity.

Space Complexity: $O(1)$, as we are using a fixed-size hash array of size 256 (for ASCII characters) and not using any additional data structures that grow with input size.

Max Consecutive Ones III

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int longestOnes(vector<int>& nums, int k) {
        int maxLen = 0;
        for (int i = 0; i < nums.size(); i++) {
            int zeros = 0;
            for (int j = i; j < nums.size(); j++) {
                if (nums[j] == 0) {
                    zeros++;
                }
                if (zeros > k) {
                    break;
                }
                maxLen = max(maxLen, j - i + 1);
            }
        }
        return maxLen;
    }
};

```

```

int main() {
    Solution sol;
    vector<int> nums = {1,1,1,0,0,0,1,1,1,1,0};
    int k = 2;
    cout << sol.longestOnes(nums, k) << endl;
    return 0;
}

```

Time complexity: $O(n^2)$, as we are using two nested loops.

Space complexity: $O(1)$, no extra space used.

Better Approach

```

#include <bits/stdc++.h>

```

```

using namespace std;

class Solution {
public:
    int longestOnes(vector<int>& nums, int k) {
        int left = 0;
        int zeros = 0;
        int maxLen = 0;
        for (int right = 0; right < nums.size(); right++)
        {
            if (nums[right] == 0) {
                zeros++;
            }
            while (zeros > k) {
                if (nums[left] == 0) {
                    zeros--;
                }
                left++;
            }
            maxLen = max(maxLen, right - left + 1);
        }
        return maxLen;
    }
};

int main() {
    Solution sol;
    vector<int> nums = {1,1,1,0,0,0,1,1,1,1,0};
    int k = 2;
    cout << sol.longestOnes(nums, k) << endl;
    return 0;
}

```

Time Complexity: O(N), We traverse the array only once using two pointers (left and right). Each element is visited at most twice once by the right pointer, and once by the left pointer when shrinking the window.

Space Complexity: O(1), Only a few integer variables are used to track the window and counters, so the space usage is constant.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int longestOnes(vector<int>& nums, int k) {
        int left = 0;
        int zerocount = 0;
        int maxlen = 0;
        for (int right = 0; right < nums.size(); right++)
        {
            if (nums[right] == 0) {
                zerocount++;
            }
            if (zerocount > k) {
                if (nums[left] == 0) {
                    zerocount--;
                }
                left++;
            }
            maxlen = max(maxlen, right - left + 1);
        }
        return maxlen;
    }
};

int main() {
    Solution sol;
    vector<int> nums = {1,1,1,0,0,0,1,1,1,1,0};
    int k = 2;
    cout << sol.longestOnes(nums, k) << endl;
    return 0;
}

```

Time Complexity: O(N), Each element is processed at most once by the right pointer and once by the left pointer. There's no nested iteration, so the traversal is strictly linear.

Space Complexity: O(1), We use only a fixed number of integer variables (left, right, zerocount, maxlen), regardless of input size, so space usage remains constant.

Fruit Into Baskets

Brute force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int totalFruit(vector<int>& fruits) {
        int maxFruits = 0;
        for (int start = 0; start < fruits.size(); ++start) {
            unordered_map<int, int> basket;
            int currentCount = 0;
            for (int end = start; end < fruits.size(); ++end) {
                basket[fruits[end]]++;
                if (basket.size() > 2) {
                    break;
                }
                currentCount++;
            }
            maxFruits = max(maxFruits, currentCount);
        }
        return maxFruits;
    }
};

int main() {
    Solution obj;
    vector<int> fruits = {1, 2, 1};
    cout << obj.totalFruit(fruits) << endl; // Output:
3
    return 0;
}
```

Time Complexity: $O(N^2)$, where N is the number of trees (length of the input array). We check every possible starting index and extend the window to the right until we encounter a third type of fruit.

Space Complexity: $O(1)$, because we only store a frequency map for at most 3 types of fruits at a time (2 allowed + 1 breaking the rule), and the size of this map does not grow with the input size.

Better Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int totalFruit(vector<int>& fruits) {
        unordered_map<int, int> basket;
        int maxFruits = 0;
        int left = 0;
        for (int right = 0; right < fruits.size(); right++)
        {
            basket[fruits[right]]++;
            while (basket.size() > 2) {
                basket[fruits[left]]--;
                if (basket[fruits[left]] == 0) {
                    basket.erase(fruits[left]);
                }
                left++;
            }
            maxFruits = max(maxFruits, right - left + 1);
        }
        return maxFruits;
    }
};

int main() {
    Solution obj;
    vector<int> fruits = {1, 2, 1, 2, 3};
    cout << obj.totalFruit(fruits) << endl;
    return 0;
}
```

Time Complexity: $O(n)$, where n is the length of the input array. The sliding window expands and contracts over the array. Each element is processed at most twice, once when the right pointer includes it in the window and possibly again when the left pointer removes it. Hence, the overall traversal is linear in time.

Space Complexity: $O(1)$, constant auxiliary space. Although we use a hash map to keep track of the count of fruit types in the current window, it holds at most two

keys (since we're allowed only two types of fruits). Therefore, the space usage remains constant and does not scale with input size.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int totalFruit(vector<int>& fruits) {
        int maxlen = 0;
        int lastfruit = -1, secondlastfruit = -1;
        int currcount = 0, lastfruitstreak = 0;
        for (int fruit : fruits)
            if (fruit == lastfruit || fruit ==
                secondlastfruit) {
                currcount++;
            } else {
                currcount = lastfruitstreak + 1;
            }
            if (fruit == lastfruit) {
                lastfruitstreak++;
            } else {
                lastfruitstreak = 1;
                secondlastfruit = lastfruit;
                lastfruit = fruit;
            }
            maxlen = max(maxlen, currcount);
        }
        return maxlen;
    }
};

int main() {
    Solution sol;
    vector<int> fruits = {1,2,1,2,3};
    cout << sol.totalFruit(fruits) << endl;
    return 0;
}
```

Time Complexity: $O(n)$, where n is the total number of elements in the input array.

Space Complexity: $O(1)$, constant auxiliary space. Only a fixed number of integer variables are maintained.

Longest repeating character replacement

Brute force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int characterReplacement(string s, int k) {
        int maxLength = 0;
        for (int i = 0; i < s.length(); i++) {
            vector<int> freq(26, 0);
            int maxFreq = 0;
            for (int j = i; j < s.length(); j++) {
                freq[s[j] - 'A']++;
                maxFreq = max(maxFreq, freq[s[j] -
                    'A']);
                int windowLength = j - i + 1;
                int replace = windowLength - maxFreq;
                if (replace <= k) {
                    maxLength = max(maxLength,
                        windowLength);
                }
            }
        }
        return maxLength;
    }
};

int main() {
    Solution sol;
    string s = "AABABBA";
    int k = 1;
    cout << sol.characterReplacement(s, k) << endl;
}
```

```

    return 0;
}

```

Time Complexity: $O(n^2 \times 26)$, where n is the length of the input string. This is because for every possible substring (which takes $O(n^2)$ time), we compute the frequency of each character (which takes $O(26) = O(1)$ time since there are only 26 uppercase English letters). So total time complexity becomes $O(n^2 \times 26)$, which simplifies to $O(n^2)$.

Space Complexity: $O(1)$, constant space. We use a fixed-size array of size 26 to store character frequencies for each substring. No additional space is used that grows with input size.

Better Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int characterReplacement(string s, int k) {
        unordered_map<char, int> freq;

        int left = 0;
        int max_freq = 0;
        int max_len = 0;
        for (int right = 0; right < s.length(); right++) {
            freq[s[right]]++;
            max_freq = max(max_freq, freq[s[right]]);
            while ((right - left + 1) - max_freq > k) {
                freq[s[left]]--;
                left++;
            }
            max_len = max(max_len, right - left + 1);
        }
        return max_len;
    }
};

int main() {
    Solution sol;
    string s = "AABABBA";
    int k = 1;
    cout << sol.characterReplacement(s, k) << endl;
}

```

```

    return 0;
}

```

Time Complexity: $O(N)$, We iterate through the entire string once using a sliding window. Each character is added and removed from the window at most once, resulting in linear time complexity relative to the length of the string (N).

Space Complexity: $O(26)$, We use a fixed-size frequency array or hashmap to store counts of uppercase English letters (which are 26 in total), so the space used remains constant regardless of the input size.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int characterReplacement(string s, int k) {
        vector<int> freq(26, 0);
        int left = 0, right = 0;
        int maxCount = 0;
        int maxLength = 0;
        while (right < s.size()) {
            freq[s[right] - 'A']++;
            maxCount = max(maxCount, freq[s[right] - 'A']);
            while ((right - left + 1) - maxCount > k) {
                freq[s[left] - 'A']--;
                left++;
            }
            maxLength = max(maxLength, right - left + 1);
            right++;
        }
        return maxLength;
    }
};

int main() {
    Solution sol;
    string s = "AABABBA";
    int k = 1;
}

```

```

// Output: 4

cout << sol.characterReplacement(s, k) << endl;

return 0;
}

```

Time Complexity: $O(n)$, where n is the length of the string, each character is processed at most twice once by the right pointer, once by the left. All operations inside the loop run in constant time.

Space Complexity: $O(1)$, constant space. Only a fixed-size frequency array (26 letters) is used, regardless of input size.

Binary subarray with sum

Brute force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int numSubarraysWithSum(vector<int>& nums,
int goal) {
        int count = 0;
        for (int start = 0; start < nums.size(); ++start) {
            int sum = 0;
            for (int end = start; end < nums.size();
++end) {
                sum += nums[end];
                if (sum == goal) {
                    count++;
                }
            }
        }
        return count;
    }
};

int main() {
    Solution obj;

    vector<int> nums = {1, 0, 1, 0, 1};
    int goal = 2;

```

```

        cout << obj.numSubarraysWithSum(nums, goal)
<< endl;

        return 0;
    }
}

```

Time Complexity: $O(n^2)$, where n is the length of the array. We are using two nested loops to explore all possible subarrays. Each subarray takes $O(1)$ time to compute the sum cumulatively, so overall $O(n^2)$ pairs are checked.

Space Complexity: $O(1)$, constant space. We only use integer variables to store counts and intermediate sums.

Better Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int numSubarraysWithSum(vector<int>& nums,
int goal) {
        unordered_map<int, int> prefixSumCount;

        int count = 0, sum = 0;

        prefixSumCount[0] = 1;

        for (int num : nums) {
            sum += num;

            if (prefixSumCount.find(sum - goal) !=
prefixSumCount.end()) {
                count += prefixSumCount[sum - goal];
            }

            prefixSumCount[sum]++;
        }

        return count;
    }
};

int main() {
    Solution sol;

    vector<int> nums = {1, 0, 1, 0, 1};

    int goal = 2;

    cout << sol.numSubarraysWithSum(nums, goal)
<< endl;

    return 0;
}

```

Time Complexity: $O(n)$, where n is the length of the input array. Each element is visited exactly once during the single-pass traversal.

Space Complexity: $O(n)$, where n is the length of the input array. In the worst case, all cumulative sums are distinct, so the hash map can store up to n unique keys. Thus, the space required grows linearly with the input size.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int numSubarraysWithSum(vector<int>& nums,
int goal) {
        return atMost(nums, goal) - atMost(nums,
goal - 1);
    }
private:
    int atMost(vector<int>& nums, int k) {
        if (k < 0) return 0;
        int left = 0;
        int sum = 0;
        int count = 0;
        for (int right = 0; right < nums.size(); right++)
        {
            sum += nums[right];
            while (sum > k) {
                sum -= nums[left];
                left++;
            }
            count += (right - left + 1);
        }
        return count;
    }
};

int main() {
    Solution sol;
    vector<int> nums = {1, 0, 1, 0, 1};
    int goal = 2;
```

```
    cout << sol.numSubarraysWithSum(nums, goal)
<< endl;

    return 0;
}
```

Time Complexity: $O(n)$, where n is the size of the input array. This is because the algorithm uses the sliding window technique twice (in the two calls to `atMost`). Each `atMost` function runs in linear time, the left and right pointers only move forward, never backward, so the total number of operations is at most $2n$.

Space Complexity: $O(1)$, constant extra space. Only a few integer variables are used.

Count number of nice subarrays

Brute force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int numberOfSubarrays(vector<int>& nums, int
k) {
        int count = 0;
        for (int start = 0; start < nums.size(); start++) {
            int oddCount = 0;
            for (int end = start; end < nums.size();
end++) {
                if (nums[end] % 2 != 0)
                    oddCount++;
                if (oddCount > k) break;
                if (oddCount == k)
                    count++;
            }
        }
        return count;
    }
};

int main() {
    Solution sol;
    vector<int> nums = {1, 1, 2, 1, 1};
    int k = 3;
```

```

        cout << sol.numberOfSubarrays(nums, k) <<
endl;

        return 0;
}

```

Time Complexity: $O(N^2)$,We use two nested loops to check all possible subarrays. For each subarray, we count the number of odd elements. The outer loop runs from index 0 to N-1, and the inner loop also runs up to N in the worst case. So total iterations can be approximately $N * N = O(N^2)$.

Space Complexity: $O(1)$, No extra space used.

Better Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int numberOfSubarrays(vector<int>& nums, int k) {
        unordered_map<int, int> freq;
        freq[0] = 1;
        int oddCount = 0;
        int result = 0;
        for (int num : nums) {
            if (num % 2 == 1) oddCount++;
            if (freq.count(oddCount - k)) {
                result += freq[oddCount - k];
            }
            freq[oddCount]++;
        }
        return result;
    }
};

int main() {
    vector<int> nums = {1, 1, 2, 1, 1};
    int k = 3;
    Solution sol;
    cout << sol.numberOfSubarrays(nums, k) <<
endl;
    return 0;
}

```

```

}

```

Time Complexity: $O(N)$,We traverse the array once and each operation (map lookup, insertion, and update) takes constant time. So the total time complexity is linear in terms of the number of elements.

Space Complexity: $O(N)$,We use a hashmap to store the frequency of prefix odd counts. In the worst case, all prefixes have different odd counts, leading to $O(n)$ extra space.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int countAtMost(vector<int>& nums, int k) {
        int left = 0, res = 0;
        for (int right = 0; right < nums.size(); right++)
        {
            if (nums[right] % 2 != 0)
                k--;
            while (k < 0) {
                if (nums[left] % 2 != 0)
                    k++;
                left++;
            }
            res += (right - left + 1);
        }
        return res;
    }

    int numberOfSubarrays(vector<int>& nums, int k) {
        return countAtMost(nums, k) -
countAtMost(nums, k - 1);
    }
};

int main() {
    Solution sol;
    vector<int> nums = {1, 1, 2, 1, 1};
    int k = 3;
}

```

```

        cout << sol.numberOfSubarrays(nums, k) <<
endl;

        return 0;
}

```

Time Complexity: $O(n)$, We scan the array two times using the sliding window helper. Each scan processes every element at most once, making it linear in size of input.

Space Complexity: $O(1)$, No additional space is used except a few integer variables for tracking window bounds and counts. So, constant space.

Number of substring containing all three characters

Brute force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int numberOfSubstrings(string s) {
        int count = 0;
        int n = s.length();
        for (int i = 0; i < n; i++) {
            vector<int> freq(3, 0);
            for (int j = i; j < n; j++) {
                freq[s[j] - 'a']++;
                if (freq[0] > 0 && freq[1] > 0 && freq[2]
> 0) {
                    count++;
                }
            }
        }
        return count;
    }
};

int main() {
    Solution sol;
    string s = "abcabc";
    cout << sol.numberOfSubstrings(s) << endl;
}

```

```

        return 0;
}

```

Time Complexity: $O(n^2)$, where n is the length of the input string. We iterate through all possible starting indices from 0 to n-1, and for each starting index, we traverse the substring until we find a valid one (containing at least one 'a', 'b', and 'c'). In the worst case, the inner loop can run up to n times for each outer loop iteration, leading to a total of $O(n^2)$ operations.

Space Complexity: $O(1)$, constant space. We use a frequency map of fixed size (only for characters 'a', 'b', and 'c'). Regardless of input size, the space used remains constant. Hence, space complexity is $O(1)$.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int numberOfSubstrings(string s) {
        vector<int> freq(3, 0);
        int res = 0;
        int left = 0;
        for (int right = 0; right < s.length(); right++) {
            freq[s[right] - 'a']++;
            while (freq[0] > 0 && freq[1] > 0 &&
freq[2] > 0) {
                res += (s.length() - right);
                freq[s[left] - 'a']--;
                left++;
            }
        }
        return res;
    }
};

// Driver code
int main() {
    Solution sol;
    string s = "abcabc";
    cout << sol.numberOfSubstrings(s) << endl;
    return 0;
}

```

```
}
```

Time Complexity: $O(n)$, We traverse the string once with the right pointer and adjust the left pointer in a linear pass. Each character is processed at most twice (once by the right pointer and once by the left), resulting in linear time complexity.

Space Complexity: $O(1)$, We only use a constant-size frequency array for three characters ('a', 'b', 'c'), hence the space usage does not grow with input size.

Maximum point you can obtain from cards

Brute Force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int maxScore(vector<int>& cardPoints, int k) {
        int n = cardPoints.size();
        int maxSum = 0;
        for (int i = 0; i <= k; i++) {
            int tempSum = 0;
            for (int j = 0; j < i; j++) {
                tempSum += cardPoints[j];
            }
            for (int j = 0; j < k - i; j++) {
                tempSum += cardPoints[n - 1 - j];
            }
            maxSum = max(maxSum, tempSum);
        }
        return maxSum;
    }
};

int main() {
    Solution sol;
    vector<int> cards = {1, 2, 3, 4, 5, 6, 1};
    int k = 3;
    cout << sol.maxScore(cards, k) << endl;
    return 0;
}
```

```
}
```

Time Complexity: $O(k)$, We try all combinations of taking cards from the front and back such that the total is exactly k cards. For each combination, we perform constant-time calculations, leading to a total of $O(k)$ iterations.

Space Complexity: $O(1)$, Only a fixed number of variables are used to store temporary sums and results, regardless of input size.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int maxScore(vector<int>& cardPoints, int k) {
        int n = cardPoints.size();
        int total = 0;
        for (int i = 0; i < k; ++i) {
            total += cardPoints[i];
        }
        int maxPoints = total;
        for (int i = 0; i < k; ++i) {
            total -= cardPoints[k - 1 - i];
            total += cardPoints[n - 1 - i];
            maxPoints = max(maxPoints, total);
        }
        return maxPoints;
    }
};

int main() {
    vector<int> cards = {1, 2, 3, 4, 5, 6, 1};
    int k = 3;
    Solution sol;
    cout << sol.maxScore(cards, k) << endl;
    return 0;
}
```

Time Complexity: $O(k)$, We calculate the initial sum of the first k cards, $O(k)$. Then we slide the window k times, $O(k)$. So overall: $O(k + k) = O(k)$

Space Complexity: $O(1)$, We only use a few variables (total, maxPoints, loop counters), no extra space used.

Longest Substring with At Most K Distinct Characters

Brute force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int lengthOfLongestSubstringKDistinct(string s,
int k) {
        int maxLength = 0;
        for (int i = 0; i < s.size(); i++) {
            unordered_map<char, int> freq;
            for (int j = i; j < s.size(); j++) {
                freq[s[j]]++;
                if (freq.size() > k) break;
                maxLength = max(maxLength, j - i + 1);
            }
        }
        return maxLength;
    }
};

int main() {
    Solution sol;
    string s = "eceba";
    int k = 2;
    cout <<
sol.lengthOfLongestSubstringKDistinct(s, k) <<
endl;
    return 0;
}
```

Time Complexity: $O(n^2)$, We are checking every possible substring which takes, and for each substring, we count distinct characters using a map/set which takes up to $O(n)$ in the worst case. But since we break early when distinct characters exceed K, the inner loop doesn't always go to the end. Hence worst-case complexity remains $O(n^2)$.

Space Complexity: $O(k)$, We use a hash map to store character frequencies for each substring, and in the worst case, it stores at most k distinct characters.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int lengthOfLongestSubstringKDistinct(string s,
int k) {
        if (k == 0 || s.empty()) return 0;
        unordered_map<char, int> freq;
        int left = 0;
        int maxLen = 0;
        for (int right = 0; right < s.length(); right++) {
            freq[s[right]]++;
            while (freq.size() > k) {
                freq[s[left]]--;
                if (freq[s[left]] == 0) {
                    freq.erase(s[left]);
                }
                left++;
            }
            maxLen = max(maxLen, right - left + 1);
        }
        return maxLen;
    }
};

int main() {
    Solution sol;
    string s = "eceba";
    int k = 2;
    cout <<
sol.lengthOfLongestSubstringKDistinct(s, k) <<
endl;
    return 0;
}
```

Time Complexity: $O(n)$, We iterate through the string once, and each character is added and removed from the map at most once. So the overall time complexity is linear.

Space Complexity: $O(k)$, We store at most k characters in the frequency map at any given time, so space used is proportional to k .

Subarray with k different integers

Brute force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int subarraysWithKDistinct(vector<int>& nums,
int k) {
        int n = nums.size();
        int count = 0;
        for (int i = 0; i < n; i++) {
            unordered_map<int, int> freq;
            for (int j = i; j < n; j++) {
                freq[nums[j]]++;
                if (freq.size() == k)
                    count++;
                if (freq.size() > k)
                    break;
            }
        }
        return count;
    }
};

int main() {
    Solution obj;
    vector<int> nums = {1,2,1,2,3};
    int k = 2;
    cout << obj.subarraysWithKDistinct(nums, k);
}
```

Time Complexity: $O(N^2 * K)$, We check all possible subarrays by iterating over all start and end indices. For

each subarray, we count the number of distinct integers using a HashSet or frequency map, which can take up to $O(K)$ time per check. So overall it becomes $O(N^2 * K)$ where N is the size of the array and K is the number of unique elements allowed.

Space Complexity: $O(K)$, For each subarray, we maintain a HashSet or HashMap to store the distinct elements in it. In the worst case, this set can grow to size K .

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int atMostK(vector<int>& nums, int K) {
        unordered_map<int, int> freq;
        int left = 0, count = 0;
        for (int right = 0; right < nums.size(); right++)
        {
            if (freq[nums[right]] == 0) {
                K--;
            }
            freq[nums[right]]++;
            while (K < 0) {
                freq[nums[left]]--;
                if (freq[nums[left]] == 0) {
                    K++;
                }
                left++;
            }
            count += (right - left + 1);
        }
        return count;
    }

    int subarraysWithKDistinct(vector<int>& nums,
int k) {
        return atMostK(nums, k) - atMostK(nums, k -
1);
    }
};
```

```
int main() {
    Solution sol;
    vector<int> nums = {1, 2, 1, 2, 3};
    int k = 2;
    cout << sol.subarraysWithKDistinct(nums, k) <<
endl;
    return 0;
}
```

Time Complexity: $O(N)$, where n is the length of the array. Both calls to `atMostK()` are linear.

Space Complexity: $O(K)$, where k is the number of distinct elements in the current window. We use a hash map to store frequency counts, which in the worst case could grow to the number of unique elements in the array.

Check if an array represents a min heap

```
#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    bool isMinHeap(vector<int>& nums) {
        int n = nums.size();
        for (int i = 0; i <= (n / 2) - 1; i++) {
            int left = 2 * i + 1;
            if (left < n && nums[i] > nums[left]) {
                return false;
            }
            int right = 2 * i + 2;
            if (right < n && nums[i] > nums[right]) {
                return false;
            }
        }
        return true;
    }
};
int main() {
    Solution obj;
```

```
    vector<int> nums = {10, 20, 30, 21, 23};
    cout << (obj.isMinHeap(nums) ? "true" : "false")
<< endl;
    return 0;
}
```

Time Complexity: $O(n)$, where n is the number of elements in the array. We only iterate through the non-leaf nodes, which are at most $n/2$, so it's still $O(n)$ in terms of asymptotic complexity. Each comparison (at most two per node) is constant time.

Space Complexity: $O(1)$, no extra space is used. We only use a few variables.

Kth largest/smallest element in an array

Brute Force

```
#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int kthLargestElement(vector<int>& nums, int k)
    {
        priority_queue<int, vector<int>,
greater<int>> pq;
        for(int i = 0; i < k; i++) pq.push(nums[i]);
        for(int i = k; i < nums.size(); i++) {
            if(nums[i] > pq.top()) {
                pq.pop();
                pq.push(nums[i]);
            }
        }
        return pq.top();
    }
};
int main() {
    vector<int> nums = {-5, 4, 1, 2, -3};
    int k = 5;
    Solution sol;
    int ans = sol.kthLargestElement(nums, k);
```

```
    cout << "The Kth largest element in the array is:
" << ans << endl;
```

```
    return 0;
}
```

Time Complexity: $O(N * \log K)$, where N is the size of the given input array. Traversing the array takes $O(N)$ time, and for each element, in the worst case, we perform heap operations which take $O(\log K)$ time. Note that K can be equal to N in the worst case, making the worst-case time complexity as $O(N * \log N)$.

Space Complexity: $O(K)$, as a Min-heap data structure of size K is used to store the K largest elements

Optimal Solution

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Solution {
```

```
public:
```

```
    int kthLargestElement(vector<int>& nums, int k)
    {
```

```
        if(k > nums.size()) return -1;
```

```
        int left = 0, right = nums.size() - 1;
```

```
        while(true) {
```

```
            int pivotIndex = randomIndex(left, right);
```

```
            pivotIndex =
partitionAndReturnIndex(nums, pivotIndex, left,
right);
```

```
            if(pivotIndex == k-1) return
nums[pivotIndex];
```

```
            else if(pivotIndex > k-1) right = pivotIndex
- 1;
```

```
            else left = pivotIndex + 1;
```

```
        }
```

```
        return -1;
```

```
    }
```

```
private:
```

```
    int randomIndex(int &left, int &right) {
```

```
        int len = right - left + 1;
```

```
        return (rand() % len) + left;
```

```
    }
```

```
    int partitionAndReturnIndex(vector<int>
&nums, int pivotIndex, int left, int right) {
```

```
        int pivot = nums[pivotIndex]; // Get the pivot
element
```

```
        swap(nums[left], nums[pivotIndex]);
```

```
        int ind = left + 1; // Index to mark the start of
right portion
```

```
        for(int i = left + 1; i <= right; i++) {
```

```
            if(nums[i] > pivot) {
```

```
                swap(nums[ind], nums[i]);
```

```
                ind++;
```

```
            }
```

```
        }
```

```
        swap(nums[left], nums[ind-1]);
```

```
        return ind-1;
```

```
    }
```

```
};
```

```
int main() {
```

```
    vector<int> nums = {-5, 4, 1, 2, -3};
```

```
    int k = 5;
```

```
    Solution sol;
```

```
    int ans = sol.kthLargestElement(nums, k);
```

```
    cout << "The Kth largest element in the array is:
" << ans << endl;
```

```
    return 0;
```

```
}
```

Time Complexity: $O(N)$, where N is the size of the given array.

In the average case (when the pivot is chosen randomly): Assuming the array gets divided into two equal parts, with every partitioning step, the search range is reduced by half. Thus, the time complexity is $O(N + N/2 + N/4 + \dots + 1) = O(N)$.

In the worst-case scenario (when the element at the left or right index is chosen as the pivot):

In such cases, the array is divided into two unequal halves, and the search range is reduced by one element with every partitioning step. Thus, the time complexity is $O(N + N-1 + N-2 + \dots + 1) = O(N^2)$. However, the probability of this worst-case scenario is negligible.

Space Complexity: $O(1)$, as we are modifying the input array in place and using only a constant amount of extra space.

Kth largest/smallest element in an array

Brute Force

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int kthLargestElement(vector<int>& nums, int k)
    {
        priority_queue<int, vector<int>,
        greater<int>> pq;
        for(int i = 0; i < k; i++) pq.push(nums[i]);
        for(int i = k; i < nums.size(); i++) {
            if(nums[i] > pq.top()) {
                pq.pop();
                pq.push(nums[i]);
            }
        }
        return pq.top();
    }
};

int main() {
    vector<int> nums = {-5, 4, 1, 2, -3};
    int k = 5;
    Solution sol;
    int ans = sol.kthLargestElement(nums, k);

    cout << "The Kth largest element in the array is:
" << ans << endl;

    return 0;
}
```

Time Complexity: $O(N * \log K)$, where N is the size of the given input array. Traversing the array takes $O(N)$ time, and for each element, in the worst case, we perform heap operations which take $O(\log K)$ time. Note that K can be equal to N in the worst case, making the worst-case time complexity as $O(N * \log N)$.

Space Complexity: $O(K)$, as a Min-heap data structure of size K is used to store the K largest elements

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
```

```
public:
    int kthLargestElement(vector<int>& nums, int k)
    {
        if(k > nums.size()) return -1;
        int left = 0, right = nums.size() - 1;
        while(true) {
            int pivotIndex = randomIndex(left, right);

            pivotIndex =
            partitionAndReturnIndex(nums, pivotIndex, left,
            right);

            if(pivotIndex == k-1) return
            nums[pivotIndex];
            else if(pivotIndex > k-1) right = pivotIndex
            - 1;
            else left = pivotIndex + 1;
        }
        return -1;
    }
};

private:
    int randomIndex(int &left, int &right) {
        int len = right - left + 1;
        return (rand() % len) + left;
    }

    int partitionAndReturnIndex(vector<int>
    &nums, int pivotIndex, int left, int right) {
        int pivot = nums[pivotIndex];
        swap(nums[left], nums[pivotIndex]);
        int ind = left + 1;
        for(int i = left + 1; i <= right; i++) {
            if(nums[i] > pivot) {
                swap(nums[ind], nums[i]);
                ind++;
            }
        }
        swap(nums[left], nums[ind-1]);
        return ind-1;
    }
};
```

```
int main() {
    vector<int> nums = {-5, 4, 1, 2, -3};

    int k = 5;

    Solution sol;

    int ans = sol.kthLargestElement(nums, k);

    cout << "The Kth largest element in the array is:
" << ans << endl;

    return 0;
}
```

Time Complexity: $O(N)$, where N is the size of the given array.

In the average case (when the pivot is chosen randomly): Assuming the array gets divided into two equal parts, with every partitioning step, the search range is reduced by half. Thus, the time complexity is $O(N + N/2 + N/4 + \dots + 1) = O(N)$.

In the worst-case scenario (when the element at the left or right index is chosen as the pivot):

In such cases, the array is divided into two unequal halves, and the search range is reduced by one element with every partitioning step. Thus, the time complexity is $O(N + N-1 + N-2 + \dots + 1) = O(N^2)$. However, the probability of this worst-case scenario is negligible.

Space Complexity: $O(1)$, as we are modifying the input array in place and using only a constant amount of extra space.

Sort K sorted array

Brute Force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int>
    sortNearlySortedArray(vector<int>& arr, int k) {
        sort(arr.begin(), arr.end());

        return arr;
    }
};

int main() {
    vector<int> arr = {6, 5, 3, 2, 8, 10, 9};

    int k = 3;

    Solution obj;
```

```
    vector<int> result =
    obj.sortNearlySortedArray(arr, k);

    for (int num : result) {
        cout << num << " ";
    }

    return 0;
}
```

Time Complexity: $O(n \log n)$, This is because we sort the entire array of size n , using a general-purpose sorting algorithm.

Space Complexity: $O(1)$ for in-place sorting (if using in-place sort) However, some built-in sorts may use $O(\log n)$ stack space for recursion or $O(n)$ if not in-place.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int>
    sortNearlySortedArray(vector<int>& arr, int k) {
        priority_queue<int, vector<int>, greater<int>>
        minHeap;

        vector<int> result;

        for (int i = 0; i <= k && i < arr.size(); i++) {
            minHeap.push(arr[i]);
        }

        for (int i = k + 1; i < arr.size(); i++) {
            result.push_back(minHeap.top());
            minHeap.pop();
            minHeap.push(arr[i]);
        }

        while (!minHeap.empty()) {
            result.push_back(minHeap.top());
            minHeap.pop();
        }

        return result;
    }
};

int main() {
```

```

vector<int> arr = {6, 5, 3, 2, 8, 10, 9};

int k = 3;

Solution obj;

vector<int> sortedArr =
obj.sortNearlySortedArray(arr, k);

for (int num : sortedArr) {

    cout << num << " ";

}

return 0;
}

```

Time Complexity: $O(n \log k)$, We insert n elements into a min heap of size $k + 1$. Each insertion/removal from heap costs $O(\log k)$. So total time = $O(n \log k)$.

Space Complexity: $O(k)$, We maintain a min heap of size at most $k + 1$ at all times.

Merge M sorted Lists

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

struct ListNode {

    int val;

    ListNode *next;

    ListNode(int x) : val(x), next(NULL) {}

};

class Solution {

public:

    ListNode*

mergeKSortedLists(vector<ListNode*>& lists) {

        vector<int> allValues;

        for (auto list : lists) {

            while (list != NULL) {

                allValues.push_back(list->val);

                list = list->next;

            }

        }

        sort(allValues.begin(), allValues.end());

        ListNode* dummy = new ListNode(0);

```

```

        ListNode* curr = dummy;

        for (int val : allValues) {

            curr->next = new ListNode(val);

            curr = curr->next;

        }

        return dummy->next;

    }

};

int main() {

    ListNode* a = new ListNode(1);

    a->next = new ListNode(4);

    a->next->next = new ListNode(5);

    ListNode* b = new ListNode(1);

    b->next = new ListNode(3);

    b->next->next = new ListNode(4);

    ListNode* c = new ListNode(2);

    c->next = new ListNode(6);

    vector<ListNode*> lists = {a, b, c};

    Solution obj;

    ListNode* result = obj.mergeKSortedLists(lists);

    while (result) {

        cout << result->val << " ";

        result = result->next;

    }

    return 0;

}

```

Time Complexity: $O(N \log N)$, We traverse all the nodes once to collect their values, which takes $O(N)$ where N is the total number of nodes across all lists. Sorting these N values takes $O(N \log N)$. Building the new linked list from the sorted array takes $O(N)$. So the overall time complexity is $O(N \log N)$.

Space Complexity: $O(N)$, We use an additional array to store all the node values, which requires $O(N)$ space. Plus, the new merged linked list also takes $O(N)$ space, but since this is the required output, only the temporary storage is considered extra.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

```

```

struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(NULL) {}
};

class Compare {
public:
    bool operator()(ListNode* a, ListNode* b) {
        return a->val > b->val;
    }
};

class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>&
lists) {
        priority_queue<ListNode*,
vector<ListNode*>, Compare> pq;

        for (auto list : lists) {
            if (list != NULL)
                pq.push(list);
        }

        ListNode* dummy = new ListNode(0);
        ListNode* tail = dummy;

        while (!pq.empty()) {
            ListNode* smallest = pq.top();
            pq.pop();
            tail->next = smallest;
            tail = tail->next;
            if (smallest->next != NULL)
                pq.push(smallest->next);
        }

        return dummy->next;
    }
};

int main() {
    Solution sol;
    ListNode* list1 = new ListNode(1);

```

```

list1->next = new ListNode(4);
list1->next->next = new ListNode(5);
ListNode* list2 = new ListNode(1);
list2->next = new ListNode(3);
list2->next->next = new ListNode(4);
ListNode* list3 = new ListNode(2);
list3->next = new ListNode(6);
vector<ListNode*> lists = {list1, list2, list3};
ListNode* result = sol.mergeKLists(lists);
while (result != NULL) {
    cout << result->val << " ";
    result = result->next;
}
return 0;
}

```

Time Complexity: $O(N * \log K)$, Where N is the total number of nodes across all K linked lists. Each insertion and extraction operation on the min-heap takes $O(\log K)$ time, and we perform such operations N times (once per node). Hence, the overall time complexity becomes $O(N * \log K)$.

Space Complexity: $O(K)$, The heap stores at most K nodes at any time one from each of the K linked lists. Therefore, the auxiliary space used by the priority queue is $O(K)$. The final merged linked list uses $O(1)$ extra space if we disregard the output list, as the nodes are rearranged rather than copied.

Replace elements by its rank in the array

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> replaceWithRank(vector<int>& arr)
    {
        vector<int> rankArr;

        for (int i = 0; i < arr.size(); i++) {
            unordered_set<int> smaller;

            for (int j = 0; j < arr.size(); j++) {

```

```

        if (arr[j] < arr[i]) {
            smaller.insert(arr[j]);
        }
    }

    int rank = smaller.size() + 1;
    rankArr.push_back(rank);
}

return rankArr;
}

};

int main() {
    vector<int> arr = {20, 15, 26, 2, 98, 6};
    Solution sol;
    vector<int> result = sol.replaceWithRank(arr);
    for (int r : result) {
        cout << r << " ";
    }
    cout << endl;
    return 0;
}

```

Time Complexity: $O(N^2)$, Because for each element (N times), we iterate the full array again (N), making it N^2 overall.

Space Complexity: $O(N)$, Each iteration uses a set for unique elements which in worst case can hold up to N elements. Also, final result array uses $O(N)$

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> replaceWithRank(vector<int>& arr)
    {
        vector<int> sortedArr = arr;
        sort(sortedArr.begin(), sortedArr.end());

        unordered_map<int, int> rankMap;
        int rank = 1;
        for (int num : sortedArr) {

```

```

            if (rankMap.find(num) == rankMap.end()) {
                rankMap[num] = rank;
                rank++;
            }
        }

        vector<int> result;
        for (int num : arr) {
            result.push_back(rankMap[num]);
        }

        return result;
    }
};

int main() {
    Solution obj;
    vector<int> arr = {1, 5, 8, 15, 8, 25, 9};
    vector<int> res = obj.replaceWithRank(arr);
    for (int x : res) {
        cout << x << " ";
    }
    return 0;
}

```

Time Complexity: $O(N \log N)$, We sort the array which takes $O(N \log N)$, then loop over the array multiple times all $O(N)$ operations.

Space Complexity: $O(N)$, We store an extra hash map for ranks and copy of the array.

Task Scheduler

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int leastInterval(vector<char>& tasks, int n) {
        unordered_map<char, int> freq;
        for (char task : tasks) {
            freq[task]++;
        }

```

```

priority_queue<int> maxHeap;
for (auto& entry : freq) {
    maxHeap.push(entry.second);
}
int time = 0;
while (!maxHeap.empty()) {
    vector<int> temp;
    int cycle = n + 1;
    int i = 0;
    while (i < cycle && !maxHeap.empty()) {
        int cnt = maxHeap.top();
        maxHeap.pop();
        cnt--;
        if (cnt > 0) {
            temp.push_back(cnt);
        }
        time++;
        i++;
    }
    for (int val : temp) {
        maxHeap.push(val);
    }
    if (maxHeap.empty()) break;
    time += (cycle - i);
}
return time;
}
};

int main() {
    Solution obj;
    vector<char> tasks = {'A','A','A','B','B','B'};
    int n = 2;
    cout << obj.leastInterval(tasks, n) << endl;
    return 0;
}

```

Time Complexity: $O(N \log K)$, We count frequencies in $O(N)$ and use a max heap of K unique tasks. Each task can be pushed and popped from the heap multiple times resulting $(\log K)$ per operation.

Space Complexity: $O(K)$, We store task frequencies in a hashmap and use a max heap, both taking $O(K)$ space where K is the number of unique tasks.

Hands of Straight

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    bool isNStraightHand(vector<int>& hand, int
groupSize) {
        if (hand.size() % groupSize != 0) return false;
        map<int, int> freq;
        for (int card : hand) {
            freq[card]++;
        }
        auto it = freq.begin();
        while (it != freq.end()) {
            if (it->second == 0) {
                ++it;
                continue;
            }
            int start = it->first;
            int count = it->second;
            for (int i = 0; i < groupSize; ++i) {
                if (freq[start + i] < count) return false;
                freq[start + i] -= count;
            }
            ++it;
        }
        return true;
    }
}

```

```
};

int main() {
    Solution sol;

    vector<int> hand1 = {1,2,3,6,2,3,4,7,8};

    int groupSize1 = 3;

    cout << sol.isNStraightHand(hand1, groupSize1)
    << endl;

    vector<int> hand2 = {1,2,3,4,5};

    int groupSize2 = 4;

    cout << sol.isNStraightHand(hand2, groupSize2)
    << endl;

    return 0;
}
```

Time Complexity: $O(N \log N)$, where N is the number of cards. We insert each card into a map which takes $\log N$ time per insertion. Then, for each unique card, we attempt to form a group of size `groupSize`, and this check involves accessing up to `groupSize` elements in the map, each in $\log N$ time.

Space Complexity: $O(N)$, because we store the frequency of each unique card in a map. In the worst case, all cards are unique, so the map can hold up to N entries.

Design Twitter

```
#include <bits/stdc++.h>

using namespace std;

class Twitter {
private:
    unordered_map<int, vector<pair<int, int>>>
    tweets;

    unordered_map<int, unordered_set<int>>
    following;

    int time;

public:
    Twitter() {
        time = 0;
    }

    void postTweet(int userId, int tweetId) {
        tweets[userId].push_back({time++, tweetId});
    }
};
```

```
vector<int> getNewsFeed(int userId) {
    priority_queue<pair<int, int>, vector<pair<int,
    int>>, greater<>> pq;

    for (auto& t : tweets[userId]) {
        pq.push(t);
        if (pq.size() > 10)
            pq.pop();
    }

    for (int followee : following[userId]) {
        for (auto& t : tweets[followee]) {
            pq.push(t);
            if (pq.size() > 10)
                pq.pop();
        }
    }

    vector<int> res;

    while (!pq.empty()) {
        res.push_back(pq.top().second);
        pq.pop();
    }

    reverse(res.begin(), res.end());

    return res;
}

void follow(int followerId, int followeeId) {
    following[followerId].insert(followeeId);
}

void unfollow(int followerId, int followeeId) {
    following[followerId].erase(followeeId);
}

};

int main() {
    Twitter twitter;

    twitter.postTweet(1, 2);
    twitter.postTweet(2, 6);

    vector<int> res1 = twitter.getNewsFeed(1);

    for (int i : res1) cout << i << " ";
```

```

cout << endl;
twitter.follow(1, 2);
vector<int> res2 = twitter.getNewsFeed(1);
for (int i : res2) cout << i << " ";
cout << endl;
twitter.unfollow(1, 2);
twitter.postTweet(1, 7);
vector<int> res3 = twitter.getNewsFeed(1);
for (int i : res3) cout << i << " ";
cout << endl;
return 0;
}

```

Time Complexity: postTweet, follow, and unfollow run in $O(1)$. getNewsFeed takes $O(\log k)$ where k is the number of users followed (including self), since we maintain a min-heap with at most one tweet per user and extract up to 10 tweets.

Space Complexity: Storing tweets takes $O(n)$, where n is total tweets. Follow relationships take $O(u^2)$ in worst case (if everyone follows everyone). Min-heap in getNewsFeed uses $O(k)$ space, where k is number of users followed.

Kth largest element in a stream of running integers

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
    priority_queue<int, vector<int>, greater<int>>
minHeap;

    int size;

public:
    Solution(int k, vector<int>& nums) {
        size = k;

        for (int num : nums) {
            minHeap.push(num);

            if (minHeap.size() > k) {
                minHeap.pop();
            }
        }
    }
}

```

```

}

int add(int val) {
    minHeap.push(val);
    if (minHeap.size() > size) {
        minHeap.pop();
    }

    return minHeap.top();
}

};

int main() {
    vector<int> nums = {4, 5, 8, 2};
    Solution kthLargest(3, nums);

    cout << kthLargest.add(3) << endl; // Output: 4
    cout << kthLargest.add(5) << endl; // Output: 5
    cout << kthLargest.add(10) << endl; // Output: 5
    cout << kthLargest.add(9) << endl; // Output: 8
    cout << kthLargest.add(4) << endl; // Output: 8

    return 0;
}

```

Time Complexity: $O(\log k)$, per insertion each insertion into the min-heap takes $O(\log k)$ time. Since we maintain a heap of at most k elements, both inserting a new number and removing the smallest (if needed) are $\log k$ operations.

Space Complexity: $O(k)$, The min-heap stores only the top k largest elements at any time. So space usage is proportional to k .

Maximum Sum Combination

Brute force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> maxCombinations(vector<int>&
nums1, vector<int>& nums2, int k) {
        vector<int> allSums;

        for (int i = 0; i < nums1.size(); i++) {

```

```

        for (int j = 0; j < nums2.size(); j++) {
            allSums.push_back(nums1[i] +
nums2[j]);
        }
    }

    sort(allSums.begin(), allSums.end(),
greater<int>());

    vector<int> result(allSums.begin(),
allSums.begin() + k);

    return result;
}
};

```

```

int main() {
    Solution obj;

    vector<int> nums1 = {7, 3};
    vector<int> nums2 = {1, 6};
    int k = 2;

    vector<int> result =
obj.maxCombinations(nums1, nums2, k);

    for (int val : result) {
        cout << val << " ";
    }

    cout << endl;

    return 0;
}

```

Time Complexity: $O(n * m + nm \log(nm))$, We compute all $n * m$ pairwise sums. Sorting takes $O(nm \log nm)$. Extracting top k is $O(k)$.

Space Complexity: $O(n * m)$, We store all pairwise sums explicitly before sorting and slicing.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> maxCombinations(vector<int>&
nums1, vector<int>& nums2, int k) {

        sort(nums1.begin(), nums1.end(),
greater<int>());
    }
}

```

```

        sort(nums2.begin(), nums2.end(),
greater<int>());

        priority_queue<tuple<int, int, int>> maxHeap;
        set<pair<int, int>> visited;

        maxHeap.push({nums1[0] + nums2[0], 0, 0});
        visited.insert({0, 0});

        vector<int> result;

        while(k-- && !maxHeap.empty()) {
            auto [sum, i, j] = maxHeap.top();
            maxHeap.pop();

            result.push_back(sum);

            if(i + 1 < nums1.size() && !visited.count({i
+ 1, j})) {
                maxHeap.push({nums1[i + 1] +
nums2[j], i + 1, j});
                visited.insert({i + 1, j});
            }

            if(j + 1 < nums2.size() &&
!visited.count({i, j + 1})) {
                maxHeap.push({nums1[i] + nums2[j +
1], i, j + 1});
                visited.insert({i, j + 1});
            }
        }

        return result;
    }
};

```

```

int main() {
    Solution sol;

    vector<int> nums1 = {7, 3};
    vector<int> nums2 = {1, 6};
    int k = 2;

    vector<int> result =
sol.maxCombinations(nums1, nums2, k);

    for(int val : result) cout << val << " ";

    return 0;
}

```

Time Complexity: $O(k * \log k)$, The loop runs k times, and in each iteration, we perform heap insertions and

deletions, which cost $O(\log k)$. In total, this results in $O(k \log k)$ time.

Space Complexity: $O(k)$, The heap and visited set can grow to size k in the worst case.

Find Median from Data Stream

Brute Force Approach

```
#include <bits/stdc++.h>

using namespace std;

class MedianFinder {
private:
    vector<int> nums;
public:
    MedianFinder() {
    }

    void addNum(int num) {
        nums.push_back(num);
    }

    double findMedian() {
        sort(nums.begin(), nums.end());
        int n = nums.size();
        if (n % 2 == 1) {
            return nums[n / 2];
        }
        return (nums[n / 2 - 1] + nums[n / 2]) / 2.0;
    }
};

int main() {
    MedianFinder mf;
    mf.addNum(1);
    mf.addNum(2);
    mf.addNum(3);

    cout << mf.findMedian() << endl; // Output: 2

    return 0;
}
```

Time Complexity: $O(N \log N)$, Every call to `findMedian()` sorts the list of N numbers, which takes

$O(N \log N)$ time. Adding a number is $O(1)$. Hence, the median query is the bottleneck.

Space Complexity: $O(N)$, We store all elements in a list, which uses linear space.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class MedianFinder {
private:
    priority_queue<int> maxHeap;
    priority_queue<int, vector<int>, greater<int>>
minHeap;
public:
    MedianFinder() {
    }

    void addNum(int num) {
        maxHeap.push(num);
        minHeap.push(maxHeap.top());
        maxHeap.pop();
        if (minHeap.size() > maxHeap.size()) {
            maxHeap.push(minHeap.top());
            minHeap.pop();
        }
    }

    double findMedian() {
        if (maxHeap.size() == minHeap.size()) {
            return (maxHeap.top() + minHeap.top()) /
2.0;
        }
        return maxHeap.top();
    }
};

int main() {
    MedianFinder mf;
    mf.addNum(1);
    mf.addNum(2);

    cout << mf.findMedian() << endl; // Output: 1.5
}
```

```

mf.addNum(3);

cout << mf.findMedian() << endl; // Output: 2

return 0;
}

```

Time Complexity: $O(N \log N + M)$, Because each call to addNum() takes $O(\log N)$, and each call to findMedian() is $O(1)$.

Space Complexity: $O(N)$, We store all the numbers across two heaps. The max-heap stores the smaller half of numbers (up to $N/2$) and the min-heap stores the larger half (up to $N/2$) So, the total space used is $O(N)$, where N is the number of elements added to the data structure.

Assign Cookies

Memoization

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int findContentChildren(vector<int>& student,
vector<int>& cookie) {

        sort(student.begin(), student.end());

        sort(cookie.begin(), cookie.end());

        vector<vector<int>> memo(student.size(),
vector<int>(cookie.size(), -1));

        return helper(0, 0, student, cookie, memo);

    }

private:
    int helper(int studentIndex, int cookieIndex,
vector<int>& student, vector<int>& cookie,
vector<vector<int>>& memo) {

        if (studentIndex >= student.size() ||
cookieIndex >= cookie.size())

            return 0;

        if (memo[studentIndex][cookieIndex] != -1)

            return memo[studentIndex][cookieIndex];

        int result = 0;

        if (cookie[cookieIndex] >=
student[studentIndex]) {

            result = max(result, 1 + helper(studentIndex
+ 1, cookieIndex + 1, student, cookie, memo));

```

```

        }

        result = max(result, helper(studentIndex,
cookieIndex + 1, student, cookie, memo));

        return memo[studentIndex][cookieIndex] =
result;

    }

};

int main() {

    vector<int> student = {1, 2, 3};

    vector<int> cookie = {1, 1};

    Solution solver;

    int result = solver.findContentChildren(student,
cookie);

    cout << "Maximum number of content students:
" << result << endl;

    return 0;

}

```

Time Complexity: $O(n*m)$, every pair of student and cookie is checked exactly once.

Space Complexity: $O(n*m) + O(n+m)$, A 2D memoization table is used to store result of subproblems and an additional $O(n+m)$ stack space is used.

Tabulation

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int findContentChildren(vector<int>& student,
vector<int>& cookie) {

        int n = student.size();

        int m = cookie.size();

        sort(student.begin(), student.end());

        sort(cookie.begin(), cookie.end());

        vector<vector<int>> dp(n + 1, vector<int>(m
+ 1, 0));

        for (int i = n - 1; i >= 0; --i) {

            for (int j = m - 1; j >= 0; --j) {

                int skip = dp[i][j + 1];

                int take = 0;

                if (cookie[j] >= student[i]) {

```

```

        take = 1 + dp[i + 1][j + 1];
    }
    dp[i][j] = max(skip, take);
}
}
return dp[0][0];
}
};

int main() {
    vector<int> student = {1, 2};
    vector<int> cookie = {1, 2, 3};
    Solution solver;

    int result = solver.findContentChildren(student,
    cookie);

    cout << "Maximum number of content students:
    " << result << endl;

    return 0;
}

```

Time Complexity: $O(n*m)$, every pair of student and cookie is checked exactly once.
Space Complexity: $O(n*m)$, A 2D memoization table is used to store result of subproblems.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int findContentChildren(vector<int>& student,
    vector<int>& cookie) {
        sort(student.begin(), student.end());
        sort(cookie.begin(), cookie.end());

        int studentIndex = 0;
        int cookieIndex = 0;

        while (studentIndex < student.size() &&
        cookieIndex < cookie.size()) {
            if (cookie[cookieIndex] >=
            student[studentIndex]) {
                studentIndex++;
            }
        }
    }
}

```

```

        cookieIndex++;
    }
    return studentIndex;
}
};

int main() {
    vector<int> student = {1, 2, 3};
    vector<int> cookie = {1, 1};
    Solution solver;

    int result = solver.findContentChildren(student,
    cookie);

    cout << "Maximum number of content students:
    " << result << endl;

    return 0;
}

```

Time Complexity: $O(n*\log n + m*\log m)$, Both the arrays are sorted in increasing order.

Space Complexity: $O(1)$, No extra space is used.

Fractional Knapsack Problem : Greedy Approach

```

#include<bits/stdc++.h>

using namespace std;

struct Item {
    int value;
    int weight;
};

class Solution {
public:
    bool static comp(Item a, Item b) {
        double r1 = (double) a.value / (double)
        a.weight;
        double r2 = (double) b.value / (double)
        b.weight;
        return r1 > r2;
    }

    double fractionalKnapsack(int W, Item arr[], int
    n) {
        sort(arr, arr + n, comp);

        int curWeight = 0;
    }
}

```

```

double finalvalue = 0.0;
for (int i = 0; i < n; i++) {
    if (curWeight + arr[i].weight <= W) {
        curWeight += arr[i].weight;
        finalvalue += arr[i].value;
    } else {
        int remain = W - curWeight;
        finalvalue += (arr[i].value / (double)
arr[i].weight) * (double) remain;
        break;
    }
}
return finalvalue;
}
};

int main() {
    int n = 3, weight = 50;
    Item arr[n] = { {100,20},{60,10},{120,30} };
    Solution obj;
    double ans = obj.fractionalKnapsack(weight, arr,
n);
    cout << "The maximum value is " <<
setprecision(2) << fixed << ans;
    return 0;
}

```

Time Complexity: $O(n \log n + n)$. $O(n \log n)$ to sort the items and $O(n)$ to iterate through all the items for calculating the answer.
Space Complexity: $O(1)$, no additional data structure has been used.

Find minimum number of coins

```

#include <bits/stdc++.h>

using namespace std;

vector<int> minCoins(int V) {
    int coins[] = {1, 2, 5, 10, 20, 50, 100, 500,
1000};
    int n = 9;
    vector<int> ans;

```

```

    for (int i = n - 1; i >= 0; i--) {
        while (V >= coins[i]) {
            V -= coins[i];
            ans.push_back(coins[i]);
        }
    }
    return ans;
}

int main() {
    int V = 49;
    vector<int> ans = minCoins(V);
    cout << "The minimum number of coins is " <<
ans.size() << endl;
    cout << "The coins are " << endl;
    for (int coin : ans) {
        cout << coin << " ";
    }
    cout << endl;
    return 0;
}

```

Time Complexity: $O(V)$ as in the worst case, we may add one coin per unit of value.
Space Complexity: $O(V)$ since result array may store up to V coins (all 1s).

Lemonade Change

```

#include <vector>
#include <iostream>

using namespace std;

class LemonadeStand {
public:
    bool lemonadeChange(vector<int>& bills) {
        int five = 0;
        int ten = 0;
        for (int bill : bills) {
            if (bill == 5) {
                five++;

```

```

    }
    else if (bill == 10) {
        if (five > 0) {
            five--;
            ten++;
        } else {
            return false;
        }
    }
}

else {
    if (five > 0 && ten > 0) {
        five--;
        ten--;
    }
    else if (five >= 3) {
        five -= 3;
    }
    else {
        return false;
    }
}

return true;
}

};

int main() {
    vector<int> bills = {5, 5, 5, 10, 20};
    cout << "Queue of customers: ";
    for (int bill : bills) cout << bill << " ";
    cout << endl;

    LemonadeStand stand;

    bool ans = stand.lemonadeChange(bills);

    if (ans)
        cout << "It is possible to provide change for all customers." << endl;
    else

```

```

        cout << "It is not possible to provide change for all customers." << endl;

```

```

        return 0;

```

```

    }

```

Time Complexity: $O(N)$ where N is the number of people in queue/ bills we will deal with. We iterate through each customer's bills exactly once. The loop runs for N iterations and at each iteration the operations performed are constant time.

Space Complexity: $O(1)$ as the algorithm uses only a constant amount of extra space regardless of the size of the input array. It does not require any additional data structures that scale with the input size.

Valid Paranthesis Checker

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool isValid(string& s, int i, int open) {
        if (open < 0) return false;
        if (i == s.length()) return open == 0;
        if (s[i] == '(') {
            return isValid(s, i + 1, open + 1);
        }
        else if (s[i] == ')') {
            return isValid(s, i + 1, open - 1);
        }
        else {
            return isValid(s, i + 1, open) ||
                isValid(s, i + 1, open + 1) ||
                isValid(s, i + 1, open - 1);
        }
    }
};

int main() {
    string s;
    cout << "Enter the string: ";

```

```

cin >> s;

Solution sol;

if (sol.isValid(s, 0, 0)) {

    cout << "Valid parenthesis string\n";

} else {

    cout << "Invalid parenthesis string\n";

}

return 0;

}

```

Time Complexity: $O(3^n)$ the worst case, every '*' can be replaced with '(', ')' or an empty string. For each '*', we have 3 choices, so with k '*' characters, we make 3^k recursive calls. If the input string has length n, and all are '*', the time complexity becomes exponential.

Space Complexity: $O(n)$, This is due to the maximum depth of the recursive call stack. At most, there are n recursive calls at any time (one for each character).

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

    bool isValid(string s) {

        int minOpen = 0, maxOpen = 0;

        for (char c : s) {

            if (c == '(') {

                minOpen++;

                maxOpen++;

            } else if (c == ')') {

                minOpen--;

                maxOpen--;

            } else {

                minOpen--;

                maxOpen++;

            }

            if (maxOpen < 0) return false;

            minOpen = max(minOpen, 0);

        }

        return minOpen == 0;

    }

};

```

```

}

};

int main() {

    string s;

    cout << "Enter the string: ";

    cin >> s;

    Solution sol;

    if (sol.isValid(s)) {

        cout << "Valid parenthesis string" << endl;

    } else {

        cout << "Invalid parenthesis string" << endl;

    }

    return 0;

}

```

Time Complexity: $O(N)$, where N is the length of the input string. We traverse the string once in a single pass, updating min and max possible open brackets at each character.

Space Complexity: $O(1)$, constant space. We use only a few integer variables (minOpen and maxOpen) regardless of the input size.

N meetings in one room

```

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

    vector<int> maxMeetings(vector<int>& start,
vector<int>& end) {

        vector<tuple<int, int, int>> meetings;

        for (int i = 0; i < start.size(); i++) {

            meetings.push_back({end[i], start[i], i +
1});

        }

        sort(meetings.begin(), meetings.end());

        vector<int> result;

        int lastEnd = -1;

        for (auto& m : meetings) {

            int e = get<0>(m);

```

```

        int s = get<1>(m);
        int idx = get<2>(m);
        if (s > lastEnd) {
            result.push_back(idx);
            lastEnd = e;
        }
    }
    return result;
}
};

int main() {
    vector<int> start = {1, 3, 0, 5, 8, 5};
    vector<int> end = {2, 4, 6, 7, 9, 9};
    Solution sol;
    vector<int> res = sol.maxMeetings(start, end);
    for (int idx : res) cout << idx << " ";
}

```

Time Complexity: $O(N \cdot \log N) + O(N)$, We sort the entire start and end time array and then iterate over every interval one by one.

Space Complexity: $O(N)$, additional space used to store tuple of start time, end time and index.

Jump Game – I

```

#include <vector>

using namespace std;

class JumpGame {
public:
    bool canJump(vector<int>& nums) {
        int maxIndex = 0;
        for (int i = 0; i < nums.size(); i++) {
            if (i > maxIndex) {
                return false;
            }
            maxIndex = max(maxIndex, i + nums[i]);
        }
        return true;
    }
};

```

```

    }
};

int main() {
    vector<int> nums = {4, 3, 7, 1, 2};

    cout << "Array representing maximum jump
from each index: ";

    for (int val : nums) cout << val << " ";

    cout << endl;

    JumpGame game;

    bool ans = game.canJump(nums);

    if (ans)
        cout << "It is possible to reach the last index."
        << endl;
    else
        cout << "It is not possible to reach the last
index." << endl;

    return 0;
}

```

Time Complexity: $O(N)$ where N is the length of the input array. We iterate through the input array exactly once and at each element perform constant time operations.

Space Complexity: $O(1)$ as the algorithm uses only a constant amount of extra space regardless of the size of the input array. It does not require any additional data structures that scale with the input size.

Jump Game 2

Brute force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int jump(vector<int>& nums) {
        return minJumps(nums, 0);
    }

private:
    int minJumps(vector<int>& nums, int position) {
        if (position >= nums.size() - 1) return 0;
    }
};

```

```

        if (nums[position] == 0) return INT_MAX;

        int minStep = INT_MAX;

        for (int jump = 1; jump <= nums[position];
            ++jump) {
            int subResult = minJumps(nums, position +
jump);
            if (subResult != INT_MAX)
                minStep = min(minStep, 1 + subResult);
        }

        return minStep;
    }
};

int main() {
    vector<int> nums = {2, 3, 1, 1, 4};

    Solution sol;

    cout << "Minimum number of jumps: " <<
sol.jump(nums) << endl;

    return 0;
}

```

Time Complexity: $O(2^N)$, where N is the number of elements in the array. This is because, from each index, the function recursively explores all possible jump lengths, leading to an exponential number of recursive calls.

Space Complexity: $O(N)$, due to the maximum depth of the recursion stack in the worst case. No extra data structures are used except the recursive call stack.

Better Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int jump(vector<int>& nums) {
        int n = nums.size();
        vector<int> dp(n, INT_MAX);
        dp[0] = 0;
        for (int i = 0; i < n; ++i) {
            for (int j = 1; j <= nums[i] && i + j < n;
                ++j) {
                dp[i + j] = min(dp[i + j], dp[i] + 1);
            }
        }
    }
};

```

```

    }
}

return dp[n - 1];
}

};

int main() {
    Solution sol;

    vector<int> nums = {2, 3, 1, 1, 4};

    cout << "Minimum jumps: " << sol.jump(nums)
<< endl;

    return 0;
}

```

Time Complexity: $O(N^2)$, We use two nested loops where outer loop runs for N elements and inner can go up to N in worst case.

Space Complexity: $O(N)$, We use an extra DP array of size N to store the minimum jumps to reach each index.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int jump(vector<int>& nums) {
        int jumps = 0, currentEnd = 0, farthest = 0;
        for (int i = 0; i < nums.size() - 1; ++i) {
            farthest = max(farthest, i + nums[i]);
            if (i == currentEnd) {
                jumps++;
                currentEnd = farthest;
            }
        }
        return jumps;
    }
};

int main() {
    Solution sol;

    vector<int> nums = {2, 3, 1, 1, 4};
}

```

```

    cout << "Minimum jumps required: " <<
    sol.jump(nums) << endl;

    return 0;
}

```

Time Complexity: $O(N)$, We traverse the entire array once from start to end.

Space Complexity: $O(1)$, No extra space is used apart from a few integer variables.

Minimum number of platforms required for a railway

Brute-Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int countPlatforms(int n, int arr[], int dep[]) {
        int ans = 1;
        for (int i = 0; i < n; i++) {
            int count = 1;
            for (int j = i + 1; j < n; j++) {
                if ((arr[i] >= arr[j] && arr[i] <= dep[j]) ||
                    (arr[j] >= arr[i] && arr[j] <= dep[i])) {
                    count++;
                }
            }
            ans = max(ans, count);
        }
        return ans;
    }
};

```

```

int main() {
    int arr[] = {900, 945, 955, 1100, 1500, 1800};
    int dep[] = {920, 1200, 1130, 1150, 1900, 2000};
    int n = sizeof(arr) / sizeof(arr[0]);
}

```

```

Solution obj;

cout << "Minimum number of Platforms
required "

    << obj.countPlatforms(n, arr, dep) << endl;

    return 0;
}

```

Time Complexity: $O(N^2)$, for every interval we check other intervals for overlap.

Space Complexity: $O(1)$, constant additional space is used.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int countPlatforms(int n, int arr[], int dep[]) {
        sort(arr, arr + n);
        sort(dep, dep + n);
        int platforms = 1;
        int result = 1;
        int i = 1, j = 0;
        while (i < n && j < n) {
            if (arr[i] <= dep[j]) {
                platforms++;
                i++;
            } else {
                platforms--;
                j++;
            }
            result = max(result, platforms);
        }
        return result;
    }
};

```

```

int main() {
    int arr[] = {900, 945, 955, 1100, 1500, 1800};
}

```

```

int dep[] = {920, 1200, 1130, 1150, 1900, 2000};
int n = sizeof(arr) / sizeof(arr[0]);

Solution obj;

cout << "Minimum number of Platforms
required "

    << obj.countPlatforms(n, arr, dep) << endl;

return 0;
}

```

Time Complexity: $O(N \log N)$, we sort both the arrival and departure arrays.

Space Complexity: $O(1)$, constant additional space is used.

Job Sequencing Problem

```

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

struct Job {
    int id;

    int dead;

    int profit;
};

class Solution {
public:
    bool static comparison(Job a, Job b) {
        return (a.profit > b.profit);
    }

    pair < int, int > JobScheduling(Job arr[], int n) {
        sort(arr, arr + n, comparison);

        int maxi = arr[0].dead;

        for (int i = 1; i < n; i++) {
            maxi = max(maxi, arr[i].dead);
        }

        int slot[maxi + 1];

        for (int i = 0; i <= maxi; i++)

            slot[i] = -1;
    }
}

```

```

int countJobs = 0, jobProfit = 0;

for (int i = 0; i < n; i++) {
    for (int j = arr[i].dead; j > 0; j--) {
        if (slot[j] == -1) {
            slot[j] = i;

            countJobs++;

            jobProfit += arr[i].profit;

            break;
        }
    }
}

return make_pair(countJobs, jobProfit);
}
};

```

```

int main() {
    int n = 4;

    Job arr[n] =
    {{1,4,20},{2,1,10},{3,2,40},{4,2,30}};

    Solution obj;

    pair < int, int > ans = obj.JobScheduling(arr, n);

    cout << ans.first << " " << ans.second << endl;

    return 0;
}

```

Time Complexity: $O(N \log N) + O(N * M)$, $O(N \log N)$ for sorting the jobs in decreasing order of profit. $O(N * M)$ since we are iterating through all N jobs and for every job, we are checking from the last deadline, say M deadlines in the worst case.

Space Complexity: $O(M)$, for an array that keeps track of which day each job is performed if M is the maximum deadline available.

Candy

Brute force Approach

```

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

class Solution {
public:
    int candy(vector<int>& ratings) {
    }
}

```

```

int n = ratings.size();
vector<int> candies(n, 1);
bool updated = true;
while (updated) {
    updated = false;
    for (int i = 1; i < n; ++i) {
        if (ratings[i] > ratings[i - 1] &&
            candies[i] <= candies[i - 1]) {
            candies[i] = candies[i - 1] + 1;
            updated = true;
        }
    }
    for (int i = n - 2; i >= 0; --i) {
        if (ratings[i] > ratings[i + 1] &&
            candies[i] <= candies[i + 1]) {
            candies[i] = candies[i + 1] + 1;
            updated = true;
        }
    }
}
return accumulate(candies.begin(),
candies.end(), 0);
};

int main() {
    Solution obj;

    vector<int> ratings = {1, 0, 5};

    cout << "Minimum candies required: " <<
obj.candy(ratings) << endl;

    return 0;
}

```

Time Complexity: $O(N^2)$ In the worst case, the algorithm can keep updating the candy distribution multiple times. Each full pass (left-to-right and right-to-left) takes $O(N)$ time, where N is the number of children.

Space Complexity: $O(N)$ We use an extra array to store the number of candies given to each child.

Better Approach

```
#include <bits/stdc++.h>
```

```

using namespace std;

class Solution {
public:
    int candy(vector<int>& ratings) {
        int n = ratings.size();
        vector<int> candies(n, 1);
        for (int i = 1; i < n; ++i) {
            if (ratings[i] > ratings[i - 1])
                candies[i] = candies[i - 1] + 1;
        }
        for (int i = n - 2; i >= 0; --i) {
            if (ratings[i] > ratings[i + 1])
                candies[i] = max(candies[i], candies[i +
1] + 1);
        }
        return accumulate(candies.begin(),
candies.end(), 0);
    }
};

int main() {
    Solution obj;

    vector<int> ratings = {1, 0, 5};

    cout << "Minimum candies: " <<
obj.candy(ratings) << endl;

    return 0;
}

```

Time Complexity: $O(n)$ The algorithm makes two passes over the ratings array (left to right and right to left), and a final pass to sum up the candies. Each pass is linear in time.

Space Complexity: $O(n)$ We use an auxiliary array to store the number of candies assigned to each child, which requires $O(n)$ space.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int candy(vector<int>& ratings) {

```

```

int n = ratings.size();
int candies = n;
int i = 1;
while (i < n) {
    if (ratings[i] == ratings[i - 1]) {
        i++;
        continue;
    }
    int peak = 0;
    while (i < n && ratings[i] > ratings[i - 1]) {
        peak++;
        candies += peak;
        i++;
    }
    int valley = 0;
    while (i < n && ratings[i] < ratings[i - 1]) {
        valley++;
        candies += valley;
        i++;
    }
    candies -= min(peak, valley);
}
return candies;
}
};

int main() {
    Solution sol;

    vector<int> ratings = {1, 3, 6, 8, 9, 5, 3};

    cout << "Minimum candies required: " <<
    sol.candy(ratings) << endl;

    return 0;
}

```

Time Complexity: $O(n)$, We iterate through the ratings array only once using a single while loop with two inner while loops. Each index is processed at most twice: once during an increasing slope and once during a decreasing slope. Hence, the total operations are linear with respect to the size of the input.

Space Complexity: $O(1)$, No extra space is used apart from a few integer variables like peak, valley, i, and candies. No auxiliary arrays or data structures are used, making the space constant regardless of input size.

Shortest Job First (or SJF) CPU Scheduling

```

#include <vector>

using namespace std;

class ShortestJobFirst {
public:
    float calculateAverageWaitTime(vector<int>&
    jobs) {
        sort(jobs.begin(), jobs.end());

        float waitTime = 0;
        int totalTime = 0;
        int n = jobs.size();
        for (int i = 0; i < n; i++) {
            waitTime += totalTime;
            totalTime += jobs[i];
        }
        return waitTime / n;
    }
};

int main() {
    vector<int> jobs = {4, 3, 7, 1, 2};

    cout << "Array Representing Job Durations: ";
    for (int job : jobs) cout << job << " ";
    cout << endl;

    ShortestJobFirst sjf;

    float ans = sjf.calculateAverageWaitTime(jobs);

    cout << "Average waiting time: " << ans <<
    endl;

    return 0;
}

```

Time Complexity: $O(N \log N + N)$ where N is the length of the jobs array. We sort the jobs array giving complexity $O(N \log N)$ and to calculate the waiting time

we iterate through the sorted array taking $O(N)$ complexity.

Space Complexity: $O(1)$ as the algorithm uses only a constant amount of extra space regardless of the size of the input array. It does not require any additional data structures that scale with the input size.

Program for Least Recently Used (LRU) Page Replacement Algorithm

```
#include <bits/stdc++.h>

using namespace std;

class LRUCache {
public:
    class Node {
    public:
        int key;
        int val;
        Node* next;
        Node* prev;
        Node(int _key, int _val) {
            key = _key;
            val = _val;
        }
    };

    Node* head = new Node(-1, -1);
    Node* tail = new Node(-1, -1);
    int cap;
    unordered_map<int, Node*> m;
    LRUCache(int capacity) {
        cap = capacity;
        head->next = tail;
        tail->prev = head;
    }
    void addNode(Node* newNode) {
        Node* temp = head->next;
        newNode->next = temp;
        newNode->prev = head;
```

```
        head->next = newNode;
        temp->prev = newNode;
    }
    void deleteNode(Node* delNode) {
        Node* delPrev = delNode->prev;
        Node* delNext = delNode->next;
        delPrev->next = delNext;
        delNext->prev = delPrev;
    }
    int get(int key_) {
        if (m.find(key_) != m.end()) {
            Node* resNode = m[key_];
            int res = resNode->val;
            m.erase(key_);
            deleteNode(resNode);
            addNode(resNode);
            m[key_] = head->next;
            return res;
        }
        return -1;
    }
    void put(int key_, int value) {
        if (m.find(key_) != m.end()) {
            Node* existingNode = m[key_];
            m.erase(key_);
            deleteNode(existingNode);
        }
        if (m.size() == cap) {
            m.erase(tail->prev->key);
            deleteNode(tail->prev);
        }
        addNode(new Node(key_, value));
        m[key_] = head->next;
    }
};

int main() {
```

```

LRUCache cache(2);

cache.put(1, 1);

cache.put(2, 2);

cout << cache.get(1) << endl;

cache.put(3, 3);

cout << cache.get(2) << endl;

cache.put(4, 4);

cout << cache.get(1) << endl;

cout << cache.get(3) << endl;

cout << cache.get(4) << endl;

return 0;
}

```

Time Complexity: get() function: $O(1)$, accessing a value in the HashMap is $O(1)$ on average.
put() function: $O(1)$, checking and removing from the map as well as inserting into the doubly linked list is $O(1)$.

Space Complexity: $O(\text{capacity})$, for storing up to capacity nodes in the doubly linked list and map.

Merge Overlapping Sub-intervals

Brute-Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<vector<int>>>
    merge(vector<vector<int>>& intervals) {
        sort(intervals.begin(), intervals.end());

        vector<vector<int>>> ans;

        int n = intervals.size();

        for (int i = 0; i < n; ) {
            int start = intervals[i][0];
            int end = intervals[i][1];
            int j = i + 1;
            while (j < n && intervals[j][0] <= end) {
                end = max(end, intervals[j][1]);
                j++;
            }

```

```

        ans.push_back({start, end});

        i = j;
    }

    return ans;
}

};

int main() {
    Solution sol;

    vector<vector<int>>> intervals = {{1,3}, {2,6},
    {8,10}, {15,18}};

    vector<vector<int>>> result =
    sol.merge(intervals);

    for (auto interval : result) {
        cout << "[" << interval[0] << ", " <<
        interval[1] << "] ";
    }

    return 0;
}

```

Time Complexity: $O(N^2)$, for every interval we check all future intervals.

Space Complexity: $O(N)$, additional space used to store the non-overlapping intervals.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<vector<int>>>
    merge(vector<vector<int>>& intervals) {
        sort(intervals.begin(), intervals.end());

        vector<vector<int>>> merged;

        for (auto interval : intervals) {
            if (merged.empty() || merged.back()[1] <
            interval[0]) {
                merged.push_back(interval);
            } else {
                merged.back()[1] = max(
                merged.back()[1],
                interval[1]
            );

```

```

        );
    }
}

return merged;
}

};

int main() {
    Solution sol;

    vector<vector<int>> intervals = {
        {1, 3}, {2, 6}, {8, 10}, {15, 18}
    };

    vector<vector<int>> result =
sol.merge(intervals);

    for (auto v : result) {
        cout << "[" << v[0] << ", " << v[1] << "]" ";
    }

    return 0;
}

```

Time Complexity: $O(N \cdot \log N) + O(N)$, we sort the entire array and then merge them in a single pass.

Space Complexity: $O(N)$, additional space used to store the non-overlapping intervals.

Non-overlapping Intervals

Brute force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int eraseOverlapIntervals(vector<vector<int>>&
intervals) {
        int n = intervals.size();

        int maxValid = 0;

        for (int mask = 0; mask < (1 << n); ++mask) {
            vector<vector<int>> subset;

            for (int i = 0; i < n; ++i) {
                if (mask & (1 << i))
                    subset.push_back(intervals[i]);
            }
        }
    }
}

```

```

    }

    sort(subset.begin(), subset.end());

    bool valid = true;

    for (int i = 1; i < subset.size(); ++i) {
        if (subset[i][0] < subset[i-1][1]) {
            valid = false;
            break;
        }
    }

    if (valid)
        maxValid = max(maxValid,
(int)subset.size());
    }

    return n - maxValid;
}

};

int main() {
    Solution sol;

    vector<vector<int>> intervals = {{1,2}, {2,3},
{3,4}, {1,3}};

    cout << "Minimum intervals to remove: " <<
sol.eraseOverlapIntervals(intervals) << endl;

    return 0;
}

```

Time Complexity: $O(2^N \times N \log N)$ The total number of subsets of N intervals is 2^N , and for each subset, We sort the subset, which takes $O(N \log N)$. We check if the intervals in the subset are non-overlapping, which takes $O(N)$.

Space Complexity: $O(N)$, We use extra space to store subsets of intervals, which can hold up to N intervals per subset.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int eraseOverlapIntervals(vector<vector<int>>&
intervals) {
        sort(intervals.begin(), intervals.end(), [](auto&
a, auto& b) {

```

```

        return a[1] < b[1];
    });
    int count = 0;
    int prevEnd = intervals[0][1];
    for (int i = 1; i < intervals.size(); i++) {
        if (intervals[i][0] < prevEnd) {
            count++;
        } else {
            prevEnd = intervals[i][1];
        }
    }
    return count;
}
};

int main() {
    Solution sol;

    vector<vector<int>> intervals = {{1, 3}, {2, 4},
    {3, 5}, {1, 2}};

    cout << "Minimum number of intervals to
    remove: " << sol.eraseOverlapIntervals(intervals)
    << endl;

    return 0;
}

```

Time Complexity: $O(n \log n)$, This is because we sort the intervals by their end time, which takes $O(n \log n)$ time. Then we traverse the sorted intervals once, which takes $O(n)$ time. So overall time complexity is $O(n \log n)$.

Space Complexity: $O(1)$, We are not using any extra data structures apart from a few variables. The sorting is done in-place (or with constant space depending on language), so the space complexity is constant.

Preorder Traversal of Binary Tree

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```

struct Node {
    int data;

    Node* left;

    Node* right;
}

```

```

    Node(int val) : data(val), left(nullptr),
    right(nullptr) {}
};

class Solution {
public:
    void preorder(Node* root, vector<int> &arr){
        if(root == nullptr){
            return;
        }
        arr.push_back(root->data);
        preorder(root->left, arr);
        preorder(root->right, arr);
    }

    vector<int> preOrder(Node* root){
        vector<int> arr;

        preorder(root, arr);

        return arr;
    }
};

int main()
{
    Node* root = new Node(1);

    root->left = new Node(2);

    root->right = new Node(3);

    root->left->left = new Node(4);

    root->left->right = new Node(5);

    Solution sol;

    vector<int> result = sol.preOrder(root);

    cout << "Preorder Traversal: ";

    for(int val : result) {
        cout << val << " ";
    }

    cout << endl;

    return 0;
}

```

Time Complexity: $O(N)$, we process each node once for traversal.

Space Complexity: $O(N)$, extra space used for storing preorder traversal and recursive stack.

Inorder Traversal of Binary Tree

Recursive Approach

```
#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int val) : data(val), left(nullptr),
right(nullptr) {}
};

class Solution{
    private:
        void recursiveInorder(TreeNode* root,
vector<int> &arr){
            if(root == nullptr){
                return;
            }
            recursiveInorder(root->left, arr);
            arr.push_back(root->data);
            recursiveInorder(root->right, arr);
        }
    public:
        vector<int> inorder(TreeNode* root){
            vector<int> arr;
            recursiveInorder(root, arr);
            return arr;
        }
};

int main()
{
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
```

```
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    Solution sol = Solution();
    cout << "Inorder Traversal: ";
    for(int val : result) {
        cout << val << " ";
    }
    cout << endl;
    return 0;
}
```

Time Complexity: $O(N)$, where N is the number of nodes in the binary tree. Each node is visited exactly once during the traversal.

Space Complexity: $O(N)$, where N is the number of nodes in the binary tree. The space is used for the recursion stack and the vector to store the inorder traversal result.

Iterative Approach

```
#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : data(x), left(nullptr),
right(nullptr) {}
};

class Solution {
    public:
        vector<int> inorder(TreeNode* root){
            stack<TreeNode*> st;
            TreeNode* node = root;
            vector<int> inorder;
            while(true){
                if(node != NULL){
                    st.push(node);
                    node = node->left;
                }
```

```

else{
    if(st.empty()){
        break;
    }
    node = st.top();
    st.pop();
    inorder.push_back(node->data);
    node = node->right;
}
}
return inorder;
}
};

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    Solution sol;
    vector<int> result = sol.inorder(root)
    cout << "Inorder Traversal: ";
    for (int val : result) {
        cout << val << " ";
    }
    cout << endl;
    return 0;
}

```

Time Complexity: $O(n)$, where n is the number of nodes in the binary tree. Each node is visited exactly once.

Space Complexity: $O(h)$, where h is the height of the binary tree. This is the space required for the stack to store the nodes during traversal.

Post-Order Traversal Of Binary Tree

```
#include <bits/stdc++.h>
```

```

using namespace std;
struct TreeNode {
    int val;
    TreeNode *left, *right;
    TreeNode(int x) : val(x), left(NULL),
right(NULL) {}
};
class Solution {
public:
    void postorderTraversal(TreeNode* root,
vector<int>& result) {
        if (!root) return;
        postorderTraversal(root->left, result);
        postorderTraversal(root->right, result);
        result.push_back(root->val);
    }
};
int main() {

```

```

    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    Solution sol;
    vector<int> result;
    sol.postorderTraversal(root, result);
    for (int val : result) cout << val << " ";
    return 0;
}

```

Time Complexity: $O(N)$, we process each node once in traversal.

Space Complexity: $O(N)$, extra space used for storing post order traversal and recursion stack space.

Level Order Traversal of a Binary Tree

```

#include <bits/stdc++.h>
using namespace std;
struct TreeNode {
    int data;

```

```

TreeNode* left;
TreeNode* right;
TreeNode(int val) : data(val), left(nullptr),
right(nullptr) {}
};
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode*
root) {
        vector<vector<int>> ans;
        if (root == nullptr) {
            return ans;
        }
        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            int size = q.size();
            vector<int> level;
            for (int i = 0; i < size; i++) {
                TreeNode* node = q.front();
                q.pop();
                level.push_back(node->data);
                if (node->left != nullptr) {
                    q.push(node->left);
                }
                if (node->right != nullptr) {
                    q.push(node->right);
                }
            }
            ans.push_back(level);
        }
        return ans;
    }
};
void printVector(const vector<int>& vec) {
    for (int num : vec) {
        cout << num << " ";
    }
}

```

```

    }
    cout << endl;
}
int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    Solution solution;
    vector<vector<int>> result =
solution.levelOrder(root);
    cout << "Level Order Traversal of Tree: "<<
endl;
    for (const vector<int>& level : result) {
        printVector(level);
    }
    return 0;
}

```

Time Complexity: $O(N)$, where N is the number of nodes in the binary tree. Each node is visited once during the level-order traversal.

Space Complexity: $O(N)$, where N is the number of nodes in the binary tree. The space is used by the queue to store nodes at each level, and in the worst case, it can hold all nodes at the last level.

Iterative Preorder Traversal of Binary Tree

```

#include <bits/stdc++.h>
using namespace std;
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}
};
class Solution {
public:

```

```

vector<int> preorderTraversal(TreeNode* root) {
    vector<int> preorder;
    if(root == nullptr) {
        return preorder;
    }
    stack<TreeNode*> st;
    st.push(root);
    while(!st.empty()) {
        root = st.top();
        st.pop();
        preorder.push_back(root->val);
        if(root->right != nullptr) {
            st.push(root->right);
        }
        if(root->left != nullptr) {
            st.push(root->left);
        }
    }
    return preorder;
}

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    Solution sol;
    vector<int> result = sol.preorderTraversal(root);
    cout << "Preorder Traversal: ";
    for (int val : result) {
        cout << val << " ";
    }
    cout << endl;
    return 0;
}

```

Time Complexity: $O(N)$, where N is the number of nodes in the binary tree. Each node is visited once during the traversal.

Space Complexity: $O(H)$, where H is the height of the binary tree. The space is used by the stack to store nodes during traversal.

Inorder Traversal of Binary Tree

Recursive Approach

```

#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int val) : data(val), left(nullptr),
right(nullptr) {}
};

class Solution{
private:
    void recursiveInorder(TreeNode* root,
vector<int> &arr){
        if(root == nullptr){
            return;
        }
        recursiveInorder(root->left, arr)
        arr.push_back(root->data);
        recursiveInorder(root->right, arr);
    }

public:
    vector<int> inorder(TreeNode* root){
        vector<int> arr;
        recursiveInorder(root, arr);
        return arr;
    }
};

int main()
{

```

```

TreeNode* root = new TreeNode(1);
root->left = new TreeNode(2);
root->right = new TreeNode(3);
root->left->left = new TreeNode(4);
root->left->right = new TreeNode(5);
Solution sol = Solution();
vector<int> result = sol.inorder(root);
cout << "Inorder Traversal: ";
for(int val : result) {
    cout << val << " ";
}
cout << endl;
return 0;
}

```

Time Complexity: $O(N)$, where N is the number of nodes in the binary tree. Each node is visited exactly once during the traversal.

Space Complexity: $O(N)$, where N is the number of nodes in the binary tree. The space is used for the recursion stack and the vector to store the inorder traversal result.

Iterative Approach

```

#include <bits/stdc++.h>
using namespace std;
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : data(x), left(nullptr),
right(nullptr) {}
};
class Solution {
public:
vector<int> inorder(TreeNode* root){
    stack<TreeNode*> st;
    TreeNode* node = root;
    vector<int> inorder;
    while(true){
        if(node != NULL){

```

```

            st.push(node);
            node = node->left;
        }
        else{
            if(st.empty()){
                break;
            }
            node = st.top();
            st.pop();
            inorder.push_back(node->data);
            node = node->right;
        }
    }
    return inorder;
}
};

```

```

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    Solution sol;
    vector<int> result = sol.inorder(root);
    cout << "Inorder Traversal: ";
    for (int val : result) {
        cout << val << " ";
    }
    cout << endl;
    return 0;
}

```

Time Complexity: $O(n)$, where n is the number of nodes in the binary tree. Each node is visited exactly once.

Space Complexity: $O(h)$, where h is the height of the binary tree. This is the space required for the stack to store the nodes during traversal.

Iterative Postorder Traversal of Binary Tree Using 2 Stacks

```
#include <bits/stdc++.h>

using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(nullptr),
right(nullptr) {}
};

vector<int> postOrder(Node* root) {
    vector<int> postorder;
    if(root == NULL){
        return postorder;
    }
    stack<Node*> st1, st2;
    st1.push(root);
    while(!st1.empty()){
        root = st1.top();
        st1.pop();
        st2.push(root);
        if(root->left != NULL){
            st1.push(root->left);
        }
        if(root->right != NULL){
            st1.push(root->right);
        }
    }
    while(!st2.empty()){
        postorder.push_back(st2.top()->data);
        st2.pop();
    }
    return postorder;
}
```

```

}

void printVector(const vector<int>& vec) {
    for (int num : vec) {
        cout << num << " ";
    }
    cout << endl;
}

int main()
{
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    vector<int> result = postOrder(root);
    cout << "Postorder traversal: ";
    printVector(result);
    return 0;
}
```

Time Complexity: $O(N)$, where N is the number of nodes in the binary tree. Each node is visited once during the traversal.

Space Complexity: $O(H)$, where H is the height of the binary tree. The space is used by the stack to store nodes during traversal.

Post-Order Traversal Of Binary Tree

```
#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode *left, *right;

    TreeNode(int x) : val(x), left(NULL),
right(NULL) {}
};

class Solution {
public:
```

```

    void postorderTraversal(TreeNode* root,
vector<int>& result) {
        if (!root) return;
        postorderTraversal(root->left, result);
        postorderTraversal(root->right, result);
        result.push_back(root->val);
    }
};

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    Solution sol;
    vector<int> result;
    sol.postorderTraversal(root, result);
    for (int val : result) cout << val << " ";
    return 0;
}

```

Time Complexity: $O(N)$, we process each node once in traversal.

Space Complexity: $O(N)$, extra space used for storing post order traversal and recursion stack space.

Preorder Inorder Postorder Traversals in One Traversal

```

#include <bits/stdc++.h>

using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    vector<vector<int>> preInPostTraversal(Node*
root) {

```

```

vector<int> pre, in, post;
if (root == NULL) {
    return {};
}
stack<pair<Node*, int>> st;
st.push({root, 1});
while (!st.empty()) {
    auto it = st.top();
    st.pop();
    if (it.second == 1) {
        pre.push_back(it.first->data);
        it.second = 2;
        st.push(it);
        if (it.first->left != NULL) {
            st.push({it.first->left, 1});
        }
    }
    else if (it.second == 2) {
        in.push_back(it.first->data);
        it.second = 3;
        st.push(it);
        if (it.first->right != NULL) {
            st.push({it.first->right, 1});
        }
    }
    else {
        post.push_back(it.first->data);
    }
}

vector<vector<int>> result;
result.push_back(pre);
result.push_back(in);
result.push_back(post);
return result;
}
};

```

```

int main()
{
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    Solution sol;
    vector<int> pre, in, post;
    vector<vector<int>> traversals =
sol.preInPostTraversal(root);

    pre = traversals[0];
    in = traversals[1];
    post = traversals[2];
    cout << "Preorder traversal: ";
    for (int val : pre) {
        cout << val << " ";
    }
    cout << endl;
    cout << "Inorder traversal: ";
    for (int val : in) {
        cout << val << " ";
    }
    cout << endl;
    cout << "Postorder traversal: ";
    for (int val : post) {
        cout << val << " ";
    }
    cout << endl;
    return 0;
}

```

Time Complexity: $O(3*N)$, we process each node thrice, once for every traversal.

Space Complexity: $O(4*N)$, extra space used for storing postorder, inorder, preorder traversal and stack.

Maximum depth of a Binary Tree

```

#include <bits/stdc++.h>
using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) : data(val), left(nullptr),
right(nullptr) {}
};
class Solution{
public:
    int maxDepth(Node* root){
        if(root == NULL){
            return 0;
        }
        int lh = maxDepth(root->left);
        int rh = maxDepth(root->right);
        return 1 + max(lh, rh);
    }
};
int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->left->right->right = new Node(6);
    root->left->right->right->right = new Node(7);
    Solution solution;
    int depth = solution.maxDepth(root);
    cout << "Maximum depth of the binary tree: "
<< depth << endl;
    return 0;
}

```

Time Complexity: $O(N)$, each node is processed once in Level Order Traversal.

Space Complexity: $O(N)$, in worst case, a maximum of $N/2$ nodes can be present in queue.

Check if the Binary Tree is Balanced Binary Tree

Brute Force Approach

```
#include <bits/stdc++.h>

using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    bool isBalanced(Node* root) {
        if (root == nullptr) {
            return true;
        }

        int leftHeight = getHeight(root->left);
        int rightHeight = getHeight(root->right);
        if (abs(leftHeight - rightHeight) <= 1 &&
            isBalanced(root->left) &&
            isBalanced(root->right)) {
            return true;
        }

        return false;
    }

    int getHeight(Node* root) {
        if (root == nullptr) {
            return 0;
        }

        int leftHeight = getHeight(root->left);
        int rightHeight = getHeight(root->right);
        return max(leftHeight, rightHeight) + 1;
    }
};
```

```
};

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->left->right->right = new Node(6);
    root->left->right->right->right = new Node(7);

    Solution solution;
    if (solution.isBalanced(root)) {
        cout << "The tree is balanced." << endl;
    } else {
        cout << "The tree is not balanced." << endl;
    }

    return 0;
}
```

Time Complexity: $O(N^2)$, where N is the number of nodes in the binary tree. For each node, we calculate the height of its left and right subtrees, and height calculation takes $O(N)$ in the worst case, leading to an overall $O(N \times N) = O(N^2)$.

Space Complexity: $O(H)$, where H is the height of the tree. This space is used by the recursive call stack of the `getHeight` function. In the worst case (skewed tree), $H = N$, and in the best case (balanced tree), $H = \log N$. No additional data structures are used, so auxiliary space remains constant.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
```

```

bool isBalanced(Node* root) {
    return dfsHeight(root) != -1;
}

int dfsHeight(Node* root) {
    if (root == NULL) return 0;
    int leftHeight = dfsHeight(root->left);
    if (leftHeight == -1)
        return -1;
    int rightHeight = dfsHeight(root->right);
    if (rightHeight == -1)
        return -1;
    if (abs(leftHeight - rightHeight) > 1)
        return -1;
    return max(leftHeight, rightHeight) + 1;
}

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->left->right->right = new Node(6);
    root->left->right->right->right = new Node(7);
    Solution solution;
    if (solution.isBalanced(root)) {
        cout << "The tree is balanced." << endl;
    } else {
        cout << "The tree is not balanced." << endl;
    }
    return 0;
}

```

Time Complexity: $O(N)$, where N is the number of nodes in the Binary Tree. Each node is visited exactly once during the postorder traversal.

Space Complexity: $O(1)$, since no extra data structures are used that grow with input size. However, **$O(H)$** auxiliary space is used by the recursion

stack, where H is the height of the tree. In the best case (balanced tree), $H = \log_2 N$; in the worst case (skewed tree), $H = N$.

Calculate the Diameter of a Binary Tree

Brute Force Approach

```

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

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    int diameter = 0;

    int calculateHeight(Node* node) {
        if (node == nullptr) {
            return 0;
        }
        int leftHeight = calculateHeight(node->left);
        int rightHeight = calculateHeight(node->right);
        diameter = max(diameter, leftHeight + rightHeight);
        return 1 + max(leftHeight, rightHeight);
    }

    int diameterOfBinaryTree(Node* root) {
        calculateHeight(root);
        return diameter;
    }
};

int main() {
    Node* root = new Node(1);

```

```

root->left = new Node(2);
root->right = new Node(3);
root->left->left = new Node(4);
root->left->right = new Node(5);
root->left->right->right = new Node(6);
root->left->right->right->right = new Node(7);

Solution solution;

int diameter =
solution.diameterOfBinaryTree(root);

cout << "The diameter of the binary tree is: " <<
diameter << endl;

return 0;
}

```

Time Complexity: $O(N*N)$ where N is the number of nodes in the Binary Tree.

Space Complexity : $O(1)$ as no additional data structures or memory is allocated.

Optimal Approach

```

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

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    int diameterOfBinaryTree(Node* root) {
        int diameter = 0;

        height(root, diameter);

        return diameter;
    }

private:
    int height(Node* node, int& diameter) {
        if (!node) {

```

```

return 0;
        }

        int lh = height(node->left, diameter);
        int rh = height(node->right, diameter);
        diameter = max(diameter, lh + rh);
        return 1 + max(lh, rh);
    }
};

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->left->right->right = new Node(6);
    root->left->right->right->right = new Node(7);

    Solution solution;

    int diameter =
solution.diameterOfBinaryTree(root);

    cout << "The diameter of the binary tree is: " <<
diameter << endl;

    return 0;
}

```

Time Complexity: $O(N)$ where N is the number of nodes in the Binary Tree. This complexity arises from visiting each node exactly once during the postorder traversal.

Space Complexity : $O(1)$ as no additional space or data structures is created that is proportional to the input size of the tree.

Maximum Sum Path in Binary Tree

```

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

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;

```

```

TreeNode(int x) {
    val = x;
    left = NULL;
    right = NULL;
}
};

class Solution {
public:
    int maxPathSum(TreeNode* root) {
        int maxSum = INT_MIN;
        dfs(root, maxSum);
        return maxSum;
    }

    int dfs(TreeNode* node, int &maxSum) {
        if (!node) return 0;
        int left = max(0, dfs(node->left, maxSum));
        int right = max(0, dfs(node->right, maxSum));
        maxSum = max(maxSum, left + right + node->val);
        return max(left, right) + node->val;
    }
};

int main() {
    TreeNode* root = new TreeNode(-10);
    root->left = new TreeNode(9);
    root->right = new TreeNode(20);
    root->right->left = new TreeNode(15);
    root->right->right = new TreeNode(7);
    Solution obj;
    cout << "Maximum Path Sum: "
         << obj.maxPathSum(root) << endl;
    return 0;
}

```

Time Complexity: $O(N)$, each node is processed once in DFS Traversal.

Space Complexity: $O(H)$, auxiliary stack space, where H is height of Binary Tree.

Check if two trees are identical

```

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

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    bool isIdentical(Node* node1, Node* node2) {
        if (node1 == NULL && node2 == NULL) {
            return true;
        }
        if (node1 == NULL || node2 == NULL) {
            return false;
        }
        return ((node1->data == node2->data)
                && isIdentical(node1->left, node2->left)
                && isIdentical(node1->right, node2->right));
    }
};

int main() {
    Node* root1 = new Node(1);
    root1->left = new Node(2);
    root1->right = new Node(3);
    root1->left->left = new Node(4);

    Node* root2 = new Node(1);
    root2->left = new Node(2);
    root2->right = new Node(3);
    root2->left->left = new Node(4);

    Solution solution;
    if (solution.isIdentical(root1, root2)) {

```

```

        cout << "The binary trees are identical." <<
endl;
    } else {
        cout << "The binary trees are not identical."
<< endl;
    }
    return 0;
}

```

Time Complexity: $O(N + M)$ where N is the number of nodes in the first Binary Tree and M is the number of nodes in the second Binary Tree. This complexity arises from visiting each node of both trees during their comparison.

Space Complexity: $O(1)$ as no additional space or data structures are created that are proportional to the input size of the tree.

Zig Zag Traversal Of Binary Tree

```

#include <bits/stdc++.h>
using namespace std;
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}
};
class Solution {
public:
    vector<vector<int>>
zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> result;
        if (!root) return result;
        queue<TreeNode*> q;
        q.push(root);
        bool leftToRight = true;
        while (!q.empty()) {
            int size = q.size();
            vector<int> level(size);
            for (int i = 0; i < size; i++) {

```

```

                TreeNode* node = q.front();
                q.pop();
                int index = leftToRight ? i : size - 1 - i;
                level[index] = node->val;
                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }
            leftToRight = !leftToRight;
            result.push_back(level);
        }
        return result;
    }
};

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->right->right = new TreeNode(6);
    Solution sol;
    vector<vector<int>> ans =
sol.zigzagLevelOrder(root);
    cout << "[";
    for (auto &level : ans) {
        cout << "[";
        for (int i = 0; i < level.size(); i++) {
            cout << level[i];
            if (i != level.size() - 1) cout << ", ";
        }
        cout << "]";
    }
    cout << "]" << endl;
    return 0;
}

```

Time Complexity: $O(N)$ where N is the number of nodes in the binary tree. Each node of the binary tree is

enqueued and dequeued exactly once, hence all nodes need to be processed and visited. Processing each node takes constant time operations which contributes to the overall linear time complexity.

Space Complexity: $O(N)$ where N is the number of nodes in the binary tree. In the worst case, the queue has to hold all the nodes of the last level of the binary tree, the last level could at most hold $N/2$ nodes hence the space complexity of the queue is proportional to $O(N)$. The resultant vector answer also stores the values of the nodes level by level and hence contains all the nodes of the tree contributing to $O(N)$ space as well.

Boundary Traversal of a Binary Tree

```
#include <iostream>

#include <vector>

using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    bool isLeaf(Node* root) {
        return !root->left && !root->right;
    }

    void addLeftBoundary(Node* root,
vector<int>& res) {
        Node* curr = root->left;
        while (curr) {
            if (!isLeaf(curr)) {
                res.push_back(curr->data);
            }
            if (curr->left) {
                curr = curr->left;
            } else {
                curr = curr->right;
            }
        }
    }
};
```

```
    }
}

void addRightBoundary(Node* root,
vector<int>& res) {
    Node* curr = root->right;
    vector<int> temp;
    while (curr) {
        if (!isLeaf(curr)) {
            temp.push_back(curr->data);
        }
        if (curr->right) {
            curr = curr->right;
        } else {
            curr = curr->left;
        }
    }
    for (int i = temp.size() - 1; i >= 0; --i) {
        res.push_back(temp[i]);
    }
}

void addLeaves(Node* root, vector<int>& res) {
    if (isLeaf(root)) {
        res.push_back(root->data);
        return;
    }
    if (root->left) {
        addLeaves(root->left, res);
    }
    if (root->right) {
        addLeaves(root->right, res);
    }
}

vector<int> printBoundary(Node* root) {
    vector<int> res;
    if (!root) {
        return res;
    }
}
```

```

    }
    if (!isLeaf(root)) {
        res.push_back(root->data);
    }
    addLeftBoundary(root, res);
    addLeaves(root, res);
    addRightBoundary(root, res);
    return res;
}
};

void printResult(const vector<int>& result) {
    for (int val : result) {
        cout << val << " ";
    }
    cout << endl;
}

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->right->left = new Node(6);
    root->right->right = new Node(7);

    Solution solution;

    vector<int> result =
solution.printBoundary(root);

    cout << "Boundary Traversal: ";
    printResult(result);

    return 0;
}

```

Time Complexity: $O(N)$ where N is the number of nodes in the Binary Tree.

Space Complexity: $O(N)$ where N is the number of nodes in the Binary Tree to store the boundary nodes of the tree. $O(H)$ or $O(\log 2N)$ Recursive stack space while traversing the tree. In the worst case scenario the tree is skewed and the auxiliary recursion stack space would be stacked up to the maximum depth of the tree, resulting in an $O(N)$ auxiliary space complexity.

Vertical Order Traversal of Binary Tree

```

#include <bits/stdc++.h>

using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    vector<vector<int>> findVertical(Node* root) {
        map<int, map<int, multiset<int>>> nodes;
        queue<pair<Node*, pair<int, int>>> todo;
        todo.push({root, {0, 0}});

        while (!todo.empty()) {
            auto p = todo.front();
            todo.pop();
            Node* temp = p.first;
            int x = p.second.first;
            int y = p.second.second;
            nodes[x][y].insert(temp->data);

            if (temp->left) {
                todo.push({temp->left, {x - 1, y + 1}});
            }

            if (temp->right) {
                todo.push({temp->right, {x + 1, y + 1}});
            }
        }

        vector<vector<int>> ans;

        for (auto p : nodes) {
            vector<int> col;

            for (auto q : p.second) {

```

```

        col.insert(col.end(), q.second.begin(),
q.second.end());
    }
    ans.push_back(col);
}
return ans;
}
};

void printResult(const vector<vector<int>>&
result) {
    for (auto level : result) {
        for (auto node : level) {
            cout << node << " ";
        }
        cout << endl;
    }
    cout << endl;
}

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->left->left = new Node(4);
    root->left->right = new Node(10);
    root->left->left->right = new Node(5);
    root->left->left->right->right = new Node(6);
    root->right = new Node(3);
    root->right->right = new Node(10);
    root->right->left = new Node(9);
    Solution solution;
    vector<vector<int>> verticalTraversal =
solution.findVertical(root);
    cout << "Vertical Traversal:" << endl;
    printResult(verticalTraversal);
    return 0;
}

```

Time Complexity: $O(N * \log^2 N * \log^2 N * \log^2 N)$, where N represents the number of nodes in the Binary Tree. Postorder traversal is performed using BFS with a time

complexity of $O(N)$, since each node is visited exactly once. Multiset operations for inserting overlapping nodes at specific vertical and horizontal levels take $O(\log^2 N)$ time. Map operations involve insertion and retrieval of nodes using vertical and level as keys. Since there are two nested maps, the total complexity becomes $O(\log^2 N * \log^2 N)$.

Space Complexity: $O(N + N/2)$, where N represents the number of nodes in the Binary Tree. The map storing nodes based on vertical and level information occupies $O(N)$ space, as it stores all N nodes of the tree. The queue for BFS traversal occupies space proportional to the maximum number of nodes at any level, which can be $O(N/2)$ in the worst case for a balanced tree.

Top view of a Binary Tree

```

#include <bits/stdc++.h>

using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) : data(val), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    vector<int> topView(Node* root) {
        vector<int> ans;
        if (root == NULL) {
            return ans;
        }
        map<int, int> mpp;
        queue<pair<Node*, int>> q;
        q.push({root, 0});
        while (!q.empty()) {
            auto it = q.front();
            q.pop();
            Node* node = it.first;
            int line = it.second;
            if (mpp.find(line) == mpp.end()) {
                mpp[line] = node->data;
            }
        }
    }
};

```

```

    }
    if (node->left != NULL) {
        q.push({node->left, line - 1});
    }
    if (node->right != NULL) {
        q.push({node->right, line + 1});
    }
}
for (auto it : mpp) {
    ans.push_back(it.second);
}
return ans;
}
};

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->left->left = new Node(4);
    root->left->right = new Node(10);
    root->left->left->right = new Node(5);
    root->left->left->right->right = new Node(6);
    root->right = new Node(3);
    root->right->right = new Node(10);
    root->right->left = new Node(9);
    Solution solution;
    vector<int> result = solution.topView(root);
    cout << "Top View Traversal: ";
    for (auto node : result) {
        cout << node << " ";
    }
    return 0;
}

```

Time Complexity: $O(N)$ where N is the number of nodes in the Binary Tree. This complexity arises from visiting each node exactly once during the BFS traversal.

Space Complexity: $O(N/2 + N/2)$ where N represents the number of nodes in the Binary Tree. The main space consuming data structure is the queue used for BFS

traversal. It acquires space proportional to the number of nodes in the level it is exploring hence in the worst case of a balanced binary tree, the queue will have at most $N/2$ nodes which is the maximum width. Additionally, the map is used to store the top view nodes based on their vertical positions hence its complexity will also be proportional to the greatest width level. In the worst case, it may have $N/2$ entries as well.

Bottom view of a Binary Tree

```

#include <iostream>

#include <vector>

#include <set>

#include <queue>

#include <map>

using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(nullptr),
right(nullptr) {}
};

class Solution{
public:
    vector<int> bottomView(Node* root){
        vector<int> ans;
        if(root == NULL){
            return ans;
        }
        map<int, int> mpp;
        queue<pair<Node*, int>> q;
        q.push({root, 0});
        while(!q.empty()){
            auto it = q.front();
            q.pop();
            Node* node = it.first;
            int line = it.second;
            mpp[line] = node->data;

```

```

        if(node->left != NULL){
            q.push({node->left, line - 1});
        }
        if(node->right != NULL){
            q.push({node->right, line + 1});
        }
    }
    for(auto it : mpp){
        ans.push_back(it.second);
    }
    return ans;
}
};

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->left->left = new Node(4);
    root->left->right = new Node(10);
    root->left->left->right = new Node(5);
    root->left->left->right->right = new Node(6);
    root->right = new Node(3);
    root->right->right = new Node(10);
    root->right->left = new Node(9);
    Solution solution;
    vector<int> bottomView =
        solution.bottomView(root);
    cout << "Bottom View Traversal: "<< endl;
    for(auto node: bottomView){
        cout << node << " ";
    }
    return 0;
}

```

Time Complexity: $O(N)$ where N is the number of nodes in the Binary Tree. This complexity arises from visiting each node exactly once during the BFS traversal.

Space Complexity: $O(N/2 + N/2)$ where N represents the number of nodes in the Binary Tree.

Right/Left view of binary tree

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) {
        data = val;
        left = right = NULL;
    }
};

class Solution {
public:
    vector<vector<int>>> levelOrder(Node* root) {
        vector<vector<int>>> ans;
        if (!root) return ans;
        queue<Node*> q;
        q.push(root);
        while (!q.empty()) {
            int size = q.size();
            vector<int> level;
            for (int i = 0; i < size; i++) {
                Node* node = q.front();
                q.pop();
                level.push_back(node->data);
                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }
            ans.push_back(level);
        }
        return ans;
    }
}

```

```

vector<int> leftView(Node* root) {
    vector<vector<int>> levels = levelOrder(root);
    vector<int> left;
    for (auto& level : levels) {
        left.push_back(level[0]);
    }
    return left;
}

vector<int> rightView(Node* root) {
    vector<vector<int>> levels = levelOrder(root);
    vector<int> right;
    for (auto& level : levels) {
        right.push_back(level.back());
    }
    return right;
}
};

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->right->right = new Node(5);
    Solution obj;
    vector<vector<int>> levelOrder =
obj.levelOrder(root);
    cout << "Level Order Traversal:\n";
    for (auto& level : levelOrder) {
        for (int val : level) cout << val << " ";
        cout << "\n";
    }
    vector<int> left = obj.leftView(root);
    cout << "Left View: ";
    for (int val : left) cout << val << " ";
    cout << "\n";
    vector<int> right = obj.rightView(root);

```

```

    cout << "Right View: ";
    for (int val : right) cout << val << " ";
    cout << "\n";
    return 0;
}

```

Time Complexity: $O(N)$ where N is the number of nodes in the binary tree. Each node of the binary tree is enqueued and dequeued exactly once, hence all nodes need to be processed and visited. Processing each node takes constant time operations which contributes to the overall linear time complexity.

Space Complexity : $O(N)$ where N is the number of nodes in the binary tree. In the worst case, the queue has to hold all the nodes of the last level of the binary tree, the last level could at most hold $N/2$ nodes hence the space complexity of the queue is proportional to $O(N)$. The resultant vector answer also stores the values of the nodes level by level and hence contains all the nodes of the tree contributing to $O(N)$ space as well

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int v) : val(v), left(NULL),
right(NULL) {}
};

class Solution {
public:
    void leftDFS(TreeNode* node, int level,
vector<int>& res) {
        if (!node) return;
        if (res.size() == level)
            res.push_back(node->val);
        leftDFS(node->left, level + 1, res);
        leftDFS(node->right, level + 1, res);
    }

    void rightDFS(TreeNode* node, int level,
vector<int>& res) {
        if (!node) return;
        if (res.size() == level)

```

```

        res.push_back(node->val);
        rightDFS(node->right, level + 1, res);
        rightDFS(node->left, level + 1, res);
    }
    vector<int> leftView(TreeNode* root) {
        vector<int> res;
        leftDFS(root, 0, res);
        return res;
    }
    vector<int> rightView(TreeNode* root) {
        vector<int> res;
        rightDFS(root, 0, res);
        return res;
    }
};

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->right = new TreeNode(4);
    root->left->right->right = new TreeNode(5);
    root->left->right->right->right = new
TreeNode(6);

    Solution sol;
    vector<int> left = sol.leftView(root);
    vector<int> right = sol.rightView(root);
    cout << "Left View: ";
    for (int val : left) cout << val << " ";
    cout << "\nRight View: ";
    for (int val : right) cout << val << " ";
    return 0;
}

```

Time Complexity: $O(N)$ In the worst case, we may visit every node in the binary tree exactly once. This happens when the tree is skewed (i.e., every node has only one child), effectively forming a linear structure. Hence, the time complexity becomes $O(N)$, where N is the total number of nodes in the tree.

Space Complexity: $O(H)$, The space complexity depends on the height (H) of the binary tree due to the recursion stack in depth-first traversal (like preorder, inorder, postorder). In a balanced binary tree, the height is $\log_2 N$, leading to $O(\log N)$ space. However, in the worst case (a skewed tree), the height is N , resulting in $O(N)$ space. So the space complexity is $O(H)$, where H is the height of the tree

Check for Symmetrical Binary Tree

```

#include <iostream>

using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(nullptr),
right(nullptr) {}
};

class Solution {
private:
    bool isSymmetricUtil(Node* root1, Node*
root2) {
        if (root1 == NULL || root2 == NULL) {
            return root1 == root2;
        }
        return (root1->data == root2->data)
            && isSymmetricUtil(root1->left, root2-
>right)
            && isSymmetricUtil(root1->right, root2-
>left);
    }
public:
    bool isSymmetric(Node* root) {
        if (!root) {
            return true;
        }
        return isSymmetricUtil(root->left, root-
>right);
    }
}

```

```

};

void printInorder(Node* root){
    if(!root){
        return;
    }
    printInorder(root->left);
    cout << root->data << " ";
    printInorder(root->right);
}

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(2);
    root->left->left = new Node(3);
    root->right->right = new Node(3);
    root->left->right = new Node(4);
    root->right->left = new Node(4);
    Solution solution;
    cout << "Binary Tree (Inorder): ";
    printInorder(root);
    cout << endl;
    bool res;
    res = solution.isSymmetric(root);
    if(res){
        cout << "This Tree is Symmetrical" << endl;
    }
    else{
        cout << "This Tree is NOT Symmetrical" <<
endl;
    }
    return 0;
}

```

Time Complexity: $O(N)$ where N is the number of nodes in the Binary Tree. This complexity arises from visiting each node exactly once during the traversal and the function compares the nodes in a symmetric manner.

Space Complexity: $O(1)$ as no additional data structures or memory is allocated.

Print Root to Node Path in a Binary Tree

```

#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    bool getPath(TreeNode* root, vector<int>& arr,
int x) {
        if (!root) {
            return false;
        }
        arr.push_back(root->val);
        if (root->val == x) {
            return true;
        }
        if (getPath(root->left, arr, x) ||
            getPath(root->right, arr, x)) {
            return true;
        }
        arr.pop_back();
        return false;
    }

    vector<int> solve(TreeNode* A, int B) {
        vector<int> arr;
        if (A == NULL) {
            return arr;
        }
        getPath(A, arr, B);
    }
}

```

```

        return arr;
    }
};

int main() {
    TreeNode* root = new TreeNode(3);
    root->left = new TreeNode(5);
    root->right = new TreeNode(1);
    root->left->left = new TreeNode(6);
    root->left->right = new TreeNode(2);
    root->right->left = new TreeNode(0);
    root->right->right = new TreeNode(8);
    root->left->right->left = new TreeNode(7);
    root->left->right->right = new TreeNode(4);
    Solution sol;
    int targetLeafValue = 7;
    vector<int> path = sol.solve(root,
targetLeafValue);
    cout << "Path from root to node with value ";
    cout << targetLeafValue << ": ";
    for (int i = 0; i < path.size(); ++i) {
        cout << path[i];
        if (i < path.size() - 1) {
            cout << " -> ";
        }
    }
    return 0;
}

```

Time Complexity: $O(N)$, where N is the number of nodes in the binary tree as each node of the binary tree is visited exactly once in the inorder traversal.

Space Complexity: $O(N)$, additional stack space used for recursion and space for storing the path.

Print Root to Node Path in a Binary Tree

```

#include <bits/stdc++.h>

using namespace std;

struct TreeNode {

```

```

    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    bool getPath(TreeNode* root, vector<int>& arr,
int x) {
        if (!root) {
            return false;
        }
        arr.push_back(root->val);
        if (root->val == x) {
            return true;
        }
        if (getPath(root->left, arr, x) ||
            getPath(root->right, arr, x)) {
            return true;
        }
        arr.pop_back();
        return false;
    }

    vector<int> solve(TreeNode* A, int B) {
        vector<int> arr;
        if (A == NULL) {
            return arr;
        }
        getPath(A, arr, B);
        return arr;
    }
};

int main() {
    TreeNode* root = new TreeNode(3);
    root->left = new TreeNode(5);
    root->right = new TreeNode(1);

```

```

root->left->left = new TreeNode(6);
root->left->right = new TreeNode(2);
root->right->left = new TreeNode(0);
root->right->right = new TreeNode(8);
root->left->right->left = new TreeNode(7);
root->left->right->right = new TreeNode(4);

Solution sol;

int targetLeafValue = 7;

vector<int> path = sol.solve(root,
targetLeafValue);

cout << "Path from root to node with value ";

cout << targetLeafValue << ": ";

for (int i = 0; i < path.size(); ++i) {

    cout << path[i];

    if (i < path.size() - 1) {

        cout << " -> ";

    }

}

return 0;
}

```

Time Complexity: $O(N)$, where N is the number of nodes in the binary tree as each node of the binary tree is visited exactly once in the inorder traversal.

Space Complexity: $O(N)$, additional stack space used for recursion and space for storing the path.

Lowest Common Ancestor for two given Nodes

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```

struct TreeNode {

    int data;

    TreeNode* left;

    TreeNode* right;

    TreeNode(int val) : data(val), left(NULL),
right(NULL) {}

};

class Solution {

```

```
public:
```

```

    TreeNode* lowestCommonAncestor(TreeNode*
root, TreeNode* p, TreeNode* q) {

        if (root == NULL || root == p || root == q) {

            return root;

        }

        TreeNode* left =
lowestCommonAncestor(root->left, p, q);

        TreeNode* right =
lowestCommonAncestor(root->right, p, q);

        if (left == NULL) {

            return right;

        } else if (right == NULL) {

            return left;

        } else {

            return root;

        }

    }

};

```

```

int main() {

    TreeNode* root = new TreeNode(3);

    root->left = new TreeNode(5);

    root->right = new TreeNode(1);

    root->left->left = new TreeNode(6);

    root->left->right = new TreeNode(2);

    root->right->left = new TreeNode(0);

    root->right->right = new TreeNode(8);

    Solution solution;

    TreeNode* p = root->left;

    TreeNode* q = root->right;

    TreeNode* lca =
solution.lowestCommonAncestor(root, p, q);

    cout << "Lowest Common Ancestor: " << lca-
>data << endl;

    return 0;

}

```

Time Complexity: $O(N)$, where N is the number of nodes in the binary tree. In the worst case, we may need to traverse all nodes to find the LCA.

Space Complexity: $O(H)$, where H is the height of the binary tree. This is due to the recursive stack space used during the traversal. In the worst case, for a skewed tree, H can be equal to N , but for a balanced tree, H will be $\log(N)$.

Maximum Width of a Binary Tree

```
#include <bits/stdc++.h>

using namespace std;

class TreeNode {
public:
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) {
        val = x;
        left = nullptr;
        right = nullptr;
    }
};

class Solution {
public:
    int widthOfBinaryTree(TreeNode* root) {
        if (!root)
            return 0;
        int maxWidth = 0;
        queue<pair<TreeNode*, int>> q;
        q.push({root, 0});
        while (!q.empty()) {
            int size = q.size();
            int minIndex = q.front().second;
            int first = 0;
            int last = 0;
            for (int i = 0; i < size; i++) {
                int currIndex = q.front().second - minIndex;
                TreeNode* node = q.front().first;
                q.pop();
```

```
                if (i == 0)
                    first = currIndex;
                if (i == size - 1)
                    last = currIndex;
                if (node->left)
                    q.push({node->left, 2 * currIndex + 1});
                if (node->right)
                    q.push({node->right, 2 * currIndex + 2});
            }
            maxWidth = max(maxWidth, last - first + 1);
        }
        return maxWidth;
    }
};

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(3);
    root->right = new TreeNode(2);
    root->left->left = new TreeNode(5);
    root->left->right = new TreeNode(3);
    root->right->right = new TreeNode(9);
    Solution sol;
    cout << "Maximum width: " <<
    sol.widthOfBinaryTree(root) << endl;

    return 0;
}
```

Time Complexity: $O(N)$, each node is processed once in level order traversal.

Space Complexity: $O(N)$, in worst case, the queue holds all the nodes of the last level i.e. $N/2$ nodes.

Check for Children Sum Property in a Binary Tree

```
#include <iostream>

using namespace std;
```

```

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    void changeTree(TreeNode* root) {
        if (root == NULL) {
            return;
        }
        int child = 0;
        if (root->left) {
            child += root->left->val;
        }
        if (root->right) {
            child += root->right->val;
        }
        if (child >= root->val) {
            root->val = child;
        } else {
            if (root->left) {
                root->left->val = root->val;
            } else if (root->right) {
                root->right->val = root->val;
            }
        }
        changeTree(root->left);
        changeTree(root->right);
        int tot = 0;
        if (root->left) {
            tot += root->left->val;
        }
        if (root->right) {
            tot += root->right->val;
        }
        if (root->left or root->right) {
            root->val = tot;
        }
    }
};

void inorderTraversal(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    inorderTraversal(root->left);
    cout << root->val << " ";
    inorderTraversal(root->right);
}

int main() {
    TreeNode* root = new TreeNode(3);
    root->left = new TreeNode(5);
    root->right = new TreeNode(1);
    root->left->left = new TreeNode(6);
    root->left->right = new TreeNode(2);
    root->right->left = new TreeNode(0);
    root->right->right = new TreeNode(8);
    root->left->right->left = new TreeNode(7);
    root->left->right->right = new TreeNode(4);
    Solution sol;
    cout << "Binary Tree before modification: ";
    inorderTraversal(root);
    cout << endl;
    sol.changeTree(root);
    cout << "Binary Tree after Children Sum
Property: " ;
    inorderTraversal(root);
    cout << endl;
    return 0;
}

```

Time Complexity: $O(N)$, where N is the number of nodes in the binary tree. This is because the algorithm traverses each node exactly once, performing constant-time operations at each node.

Space Complexity: $O(N)$, where N is the number of nodes in the Binary Tree.

Print all the Nodes at a distance of K in a Binary Tree

```
#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL),
right(NULL) {}
};

class Solution {
public:
    vector<int> distanceK(TreeNode* root,
TreeNode* target, int k) {
        if (!root) return {};
        unordered_map<TreeNode*, TreeNode*>
parentMap;
        mapParentNodes(root, parentMap);
        return bfsFromTarget(target, parentMap, k);
    }

private:
    void mapParentNodes(TreeNode* root,
unordered_map<TreeNode*, TreeNode*>&
parentMap) {
        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            TreeNode* node = q.front();
            q.pop();
            if (node->left) {
                parentMap[node->left] = node;
                q.push(node->left);
            }
```

```
        }
        if (node->right) {
            parentMap[node->right] = node;
            q.push(node->right);
        }
    }
}

vector<int> bfsFromTarget(TreeNode* target,
unordered_map<TreeNode*, TreeNode*>&
parentMap, int k) {
    queue<TreeNode*> q;
    unordered_set<TreeNode*> visited;
    q.push(target);
    visited.insert(target);
    int currentLevel = 0;
    while (!q.empty()) {
        int size = q.size();
        if (currentLevel++ == k) break;
        for (int i = 0; i < size; ++i) {
            TreeNode* node = q.front();
            q.pop();
            if (node->left && visited.find(node->left)
== visited.end()) {
                visited.insert(node->left);
                q.push(node->left);
            }
            if (node->right && visited.find(node-
>right) == visited.end()) {
                visited.insert(node->right);
                q.push(node->right);
            }
            if (parentMap.count(node) &&
visited.find(parentMap[node]) == visited.end()) {
                visited.insert(parentMap[node]);
                q.push(parentMap[node]);
            }
        }
    }
}
```

```

vector<int> result;
while (!q.empty()) {
    result.push_back(q.front()->val);
    q.pop();
}
return result;
}
};

int main() {
    TreeNode* root = new TreeNode(3);
    root->left = new TreeNode(5);
    root->right = new TreeNode(1);
    root->left->left = new TreeNode(6);
    root->left->right = new TreeNode(2);
    root->left->right->left = new TreeNode(7);
    root->left->right->right = new TreeNode(4);
    root->right->left = new TreeNode(0);
    root->right->right = new TreeNode(8);
    TreeNode* target = root->left;
    int k = 2;
    Solution sol;
    vector<int> result = sol.distanceK(root, target,
k);
    for (int val : result) {
        cout << val << " ";
    }
    return 0;
}

```

Time Complexity: $O(N)$,We visit each node exactly once when building the parent map using BFS , $O(N)$. We again visit each node at most once during the second BFS traversal from the target, $O(N)$. Hence, the total time complexity is $O(N)$, where N is the number of nodes in the binary tree.

Space Complexity: $O(N)$, The parent map stores one entry per node, $O(N)$. The queue and visited set used in BFS also take up to $O(N)$ space in the worst case. Therefore, the total space complexity is $O(N)$.

Minimum time taken to BURN the Binary Tree from a Node

```

#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(NULL),
right(NULL) {}
};

class Solution {
public:
    int minTime(TreeNode* root, int target) {
        unordered_map<int, vector<int>> graph;
        buildGraph(root, nullptr, graph);
        unordered_set<int> visited;
        queue<int> q;
        q.push(target);
        visited.insert(target);
        int time = 0;
        while (!q.empty()) {
            int size = q.size();
            bool burned = false;
            for (int i = 0; i < size; i++) {
                int node = q.front();
                q.pop();
                for (int neighbor : graph[node]) {
                    if (!visited.count(neighbor)) {
                        visited.insert(neighbor);
                        q.push(neighbor);
                        burned = true;
                    }
                }
            }
            if (burned) time++;
        }
    }
};

```

```

    }
    return time;
}

private:
    void buildGraph(TreeNode* node, TreeNode*
parent, unordered_map<int, vector<int>>& graph)
    {
        if (!node) return;
        if (parent) {
            graph[node->val].push_back(parent->val);
            graph[parent->val].push_back(node->val);
        }
        buildGraph(node->left, node, graph);
        buildGraph(node->right, node, graph);
    }
};

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->right->left = new TreeNode(5);
    root->right->right = new TreeNode(6);
    root->left->left->right = new TreeNode(7);

    Solution sol;

    int target = 1;

    cout << "Minimum time to burn the tree: " <<
sol.minTime(root, target) << endl;

    return 0;
}

```

Time Complexity: $O(N)$, We perform two full traversals of the tree. The first is a DFS to build the graph, which visits each node once, taking $O(N)$ time. The second is a BFS starting from the target node to simulate the fire spreading, which also takes $O(N)$ time since each node is visited once. Therefore, the total time complexity is $O(N)$.

Space Complexity: $O(N)$, We use an adjacency list to store the graph, which takes $O(N)$ space. A visited set is used to track which nodes are burned, also requiring $O(N)$ space. Additionally, the BFS queue can hold up to

$O(N)$ nodes in the worst case. So, the overall space complexity is $O(N)$.

Count Number of Nodes in a Binary Tree

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    void inorder(TreeNode* root, int &count) {
        if (root == NULL) {
            return;
        }
        count++;
        inorder(root->left, count);
        inorder(root->right, count);
    }

    int countNodes(TreeNode* root) {
        if (root == NULL) {
            return 0;
        }
        int count = 0;
        inorder(root, count);

        return count;
    }
};

int main() {
    TreeNode* root = new TreeNode(1);

```

```

root->left = new TreeNode(2);
root->right = new TreeNode(3);
root->left->left = new TreeNode(4);
root->left->right = new TreeNode(5);
root->right->left = new TreeNode(6);
Solution sol;

int totalNodes = sol.countNodes(root);

cout << "Total number of nodes in the Complete
Binary Tree: " << totalNodes << endl;

return 0;
}

```

Time Complexity: $O(N)$ where N is the number of nodes in the binary tree as each node of the binary tree is visited exactly once.

Space Complexity : $O(N)$ where N is the number of nodes in the binary tree. This is because the recursive stack uses an auxiliary space which can potentially hold all nodes in the tree when dealing with a skewed tree (all nodes have only one child), consuming space proportional to the number of nodes. In the average case or for a balanced tree, the maximum number of nodes that could be in the stack at any given time would be roughly the height of the tree hence $O(\log 2N)$.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    int countNodes(TreeNode* root) {
        if (root == NULL) {
            return 0;
        }
        int lh = findHeightLeft(root);

```

```

        int rh = findHeightRight(root);
        if (lh == rh) {
            return (1 << lh) - 1;
        }
        return 1 + countNodes(root->left) +
countNodes(root->right);
    }
    int findHeightLeft(TreeNode* node) {
        int height = 0;
        while (node) {
            height++;
            node = node->left;
        }
        return height;
    }
    int findHeightRight(TreeNode* node) {
        int height = 0;
        while (node) {
            height++;
            node = node->right;
        }
        return height;
    }
};

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->right->left = new TreeNode(6);
    Solution sol;
    int totalNodes = sol.countNodes(root);
    cout << "Total number of nodes in the Complete
Binary Tree: " << totalNodes << endl;
    return 0;
}

```

Time Complexity: $O(\log N * \log N)$ where N is the number of nodes in the Binary Tree. The calculation of leftHeight and rightHeight takes $O(\log N)$ time. In the worst case, when encountering the second case (leftHeight != rightHeight), the recursive calls are made at most $\log N$ times (the height of the tree). Therefore, the total time complexity is $O(\log N * \log N)$.

Space Complexity : $O(H) \sim O(N)$ where N is the number of nodes in the Binary Tree. The space complexity is determined by the maximum depth of the recursion stack, which is equal to the height of the binary tree. Since the given tree is a complete binary tree, the height will always be $\log N$. Therefore, the space complexity is $O(\log N)$.

Construct A Binary Tree from Inorder and Preorder Traversal

```
#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    TreeNode* buildTree(vector<int>& preorder,
vector<int>& inorder) {
        map<int, int> inMap;

        for (int i = 0; i < inorder.size(); i++) {
            inMap[inorder[i]] = i;
        }

        return build(preorder, 0, preorder.size() - 1,
inorder, 0, inorder.size() - 1, inMap);
    }

private:
    TreeNode* build(vector<int>& preorder, int
preStart, int preEnd,

        vector<int>& inorder, int inStart, int
inEnd, map<int, int>& inMap) {
        if (preStart > preEnd || inStart > inEnd) return
nullptr;
```

```
        TreeNode* root = new
TreeNode(preorder[preStart]);

        int inRoot = inMap[root->val];

        int numsLeft = inRoot - inStart;

        root->left = build(preorder, preStart + 1,
preStart + numsLeft,

            inorder, inStart, inRoot - 1,
inMap);

        root->right = build(preorder, preStart +
numsLeft + 1, preEnd,

            inorder, inRoot + 1, inEnd,
inMap);

        return root;
    }
};

void printInorder(TreeNode* root) {
    if (!root) return;
    printInorder(root->left);
    cout << root->val << " ";
    printInorder(root->right);
}

int main() {
    vector<int> inorder = {9, 3, 15, 20, 7};
    vector<int> preorder = {3, 9, 20, 15, 7};

    Solution sol;

    TreeNode* root = sol.buildTree(preorder,
inorder);

    cout << "Inorder of Unique Binary Tree
Created:\n";

    printInorder(root);

    cout << endl;

    return 0;
}
```

Time Complexity: $O(N)$, as each node is visited once.

Space Complexity: $O(N)$, for the hashmap and recursion stack (worst case when tree is skewed).

Construct Binary Tree from Inorder and PostOrder Traversal

```

#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode *left, *right;
    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    TreeNode* buildTree(vector<int>& inorder,
vector<int>& postorder) {
        if (inorder.size() != postorder.size()) return
nullptr;

        map<int, int> hm;
        for (int i = 0; i < inorder.size(); i++) {
            hm[inorder[i]] = i;
        }

        return build(inorder, 0, inorder.size() - 1,
                    postorder, 0, postorder.size() - 1, hm);
    }

private:
    TreeNode* build(vector<int>& inorder, int is, int
ie,
                    vector<int>& postorder, int ps, int pe,
map<int, int>& hm) {
        if (ps > pe || is > ie) return nullptr;

        TreeNode* root = new
TreeNode(postorder[pe]);

        int inRoot = hm[postorder[pe]];
        int numsLeft = inRoot - is;

        root->left = build(inorder, is, inRoot - 1,
                        postorder, ps, ps + numsLeft - 1,
hm)
        root->right = build(inorder, inRoot + 1, ie,
                        postorder, ps + numsLeft, pe - 1,
hm);

        return root;
    }
}

```

```

};

void printInorder(TreeNode* root) {
    if (!root) return;
    printInorder(root->left);
    cout << root->val << " ";
    printInorder(root->right);
}

int main() {
    vector<int> inorder = {40, 20, 50, 10, 60, 30};
    vector<int> postorder = {40, 50, 20, 60, 30, 10};

    cout << "Inorder Vector: ";
    for (int x : inorder) cout << x << " ";
    cout << "\nPostorder Vector: ";
    for (int x : postorder) cout << x << " ";
    cout << endl;

    Solution sol;

    TreeNode* root = sol.buildTree(inorder,
postorder);

    cout << "Inorder of Unique Binary Tree
Created:\n";

    printInorder(root);

    cout << endl;

    return 0;
}

```

Time Complexity: O(N), Every node is visited once.

Space Complexity: O(N), Due to hashmap and recursion stack (up to tree height).

Serialize And Deserialize a Binary Tree

```

#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) {

```

```

        val = x;
        left = nullptr;
        right = nullptr;
    }
};

void inorder(TreeNode* root) {
    if (!root) {
        return;
    }
    inorder(root->left);
    cout << root->val << " "
    inorder(root->right);
}

class Solution {
public:
    string serialize(TreeNode* root) {
        if (!root) {
            return "";
        }
        string s = "";
        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            TreeNode* curNode = q.front();
            q.pop();
            if (curNode == nullptr) {
                s += "#,";
            }
            else {
                s += to_string(curNode->val) + ",";
                q.push(curNode->left);
                q.push(curNode->right);
            }
        }
        return s;
    }
};

```

```

TreeNode* deserialize(string data) {
    if (data.empty()) {
        return nullptr;
    }
    stringstream s(data);
    string str;
    getline(s, str, ',');
    TreeNode* root = new TreeNode(stoi(str));
    queue<TreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        TreeNode* node = q.front();
        q.pop();
        getline(s, str, ',');
        if (str != "#") {
            TreeNode* leftNode = new
TreeNode(stoi(str));
            node->left = leftNode;
            q.push(leftNode);
        }
        getline(s, str, ',');
        if (str != "#") {
            TreeNode* rightNode = new
TreeNode(stoi(str));
            node->right = rightNode;
            q.push(rightNode);
        }
    }
    return root;
};

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->right->left = new TreeNode(4);
    root->right->right = new TreeNode(5);
}

```

```

Solution solution;

cout << "Original Tree: ";

inorder(root);

cout << endl;

string serialized = solution.serialize(root);

cout << "Serialized: " << serialized << endl;

TreeNode* deserialized =
solution.deserialize(serialized);

cout << "Tree after deserialisation: ";

inorder(deserialized);

cout << endl;

return 0;
}

```

Time Complexity: $O(N)$

1. serialize function: $O(N)$, where N is the number of nodes in the tree. This is because the function performs a level-order traversal of the tree, visiting each node once.
2. deserialize function: $O(N)$, where N is the number of nodes in the tree. Similar to the serialize function, it processes each node once while reconstructing the tree.

Space Complexity: $O(N)$

1. serialize function: $O(N)$, where N is the maximum number of nodes at any level in the tree. In the worst case, the queue can hold all nodes at the last level of the tree.
2. deserialize function: $O(N)$, where N is the maximum number of nodes at any level in the tree. The queue is used to store nodes during the reconstruction process, and in the worst case, it may hold all nodes at the last level.

Morris Preorder Traversal of a Binary Tree

```

#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;

    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}

```

```

};

class Solution {
public:
    vector<int> getPreorder(TreeNode* root) {
        vector<int> preorder;
        TreeNode* cur = root;
        while (cur != NULL) {
            if (cur->left == NULL) {
                preorder.push_back(cur->val);
                cur = cur->right;
            } else {
                TreeNode* prev = cur->left;
                while (prev->right && prev->right !=
cur) {
                    prev = prev->right;
                }
                if (prev->right == NULL) {
                    prev->right = cur;
                    cur = cur->left;
                } else {
                    prev->right = NULL;
                    preorder.push_back(cur->val);
                    cur = cur->right;
                }
            }
        }
        return preorder;
    }
};

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->left->right->right = new TreeNode(6);

```

```

Solution sol;
vector<int> preorder = sol.getPreorder(root);
cout << "Binary Tree Morris Preorder Traversal:
";

for(int i = 0; i < preorder.size(); i++){
    cout << preorder[i] << " ";
}

cout << endl;

return 0;
}

```

Time Complexity: $O(2N)$, the time complexity is linear, as each node is visited at most twice (once for establishing the temporary link and once for reverting it).
Space Complexity: $O(1)$, the space complexity is constant, as the algorithm uses only a constant amount of extra space irrespective of the input size.

Morris Inorder Traversal of a Binary tree

```

#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    vector<int> getInorder(TreeNode* root) {
        vector<int> inorder;
        TreeNode* cur = root;
        while (cur != NULL) {
            if (cur->left == NULL) {
                inorder.push_back(cur->val);
                cur = cur->right;
            }
        }
    }
};

```

```

        } else {
            TreeNode* prev = cur->left;
            while (prev->right && prev->right !=
cur) {
                prev = prev->right;
            }
            if (prev->right == NULL) {
                prev->right = cur;
                cur = cur->left;
            } else {
                prev->right = NULL;
                inorder.push_back(cur->val);
                cur = cur->right;
            }
        }
    }

    return inorder;
}

};

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->left->right->right = new TreeNode(6);
    Solution sol;
    vector<int> inorder = sol.getInorder(root);
    cout << "Binary Tree Morris Inorder Traversal:
";
    for(int i = 0; i < inorder.size(); i++){
        cout << inorder[i] << " ";
    }
    cout << endl;
    return 0;
}

```

Time Complexity: $O(2N)$, the time complexity is linear, as each node is visited at most twice (once for establishing the temporary link and once for reverting it).
Space Complexity: $O(1)$, the space complexity is constant, as the algorithm uses only a constant amount of extra space irrespective of the input size.

Flatten Binary Tree to Linked List

Brute Force

```
#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    TreeNode* prev = nullptr;
    void flatten(TreeNode* root) {
        if (root == nullptr) return;
        flatten(root->right);
        flatten(root->left);
        root->right = prev;
        root->left = nullptr;
        prev = root;
    }
};

void printPreorder(TreeNode* root) {
    if (!root) return;
    cout << root->val << " ";
    printPreorder(root->left);
    printPreorder(root->right);
}

void printFlattenTree(TreeNode* root) {
    if (!root) return;
```

```
        cout << root->val << " ";
        printFlattenTree(root->right);
    }

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->left->right->right = new TreeNode(6);
    root->right->right = new TreeNode(7);
    root->right->left = new TreeNode(8);

    Solution sol;

    cout << "Binary Tree Preorder: ";
    printPreorder(root);
    cout << endl;
    sol.flatten(root);
    cout << "Binary Tree After Flatten: ";
    printFlattenTree(root);
    cout << endl;
    return 0;
}
```

Time Complexity: $O(n)$, where n is the number of nodes in the binary tree. Each node is visited once during the flattening process.

Space Complexity: $O(\log_2 N)$, where N is the number of nodes in the Binary Tree. There are no additional data structures or space used but the auxiliary stack space is used during recursion. Since the recursion depth can be at most equal to the height to the Binary Tree, the space complexity is $O(H)$ where H is the height of the Binary Tree. In the ideal case, $H = \log_2 N$ and in the worst case $H = N$ (skewed tree).

Better Approach

```
#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
```

```

    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    TreeNode* prev = nullptr;
    void flatten(TreeNode* root) {
        if (root == nullptr) return;
        stack<TreeNode*> st;
        st.push(root);
        while (!st.empty()) {
            TreeNode* cur = st.top();
            st.pop();
            if (cur->right != nullptr) st.push(cur->right);
            if (cur->left != nullptr) st.push(cur->left);
            if (!st.empty()) cur->right = st.top();
            cur->left = nullptr;
        }
    }
};

void printPreorder(TreeNode* root) {
    if (!root) return;
    cout << root->val << " ";
    printPreorder(root->left);
    printPreorder(root->right);
}

void printFlattenTree(TreeNode* root) {
    if (!root) return;
    cout << root->val << " ";
    printFlattenTree(root->right);
}

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);

```

```

    root->left->right = new TreeNode(5);
    root->left->right->right = new TreeNode(6);
    root->right->right = new TreeNode(7);
    root->right->left = new TreeNode(8);

    Solution sol;

    cout << "Binary Tree Preorder: ";
    printPreorder(root);
    cout << endl;
    sol.flatten(root);
    cout << "Binary Tree After Flatten: ";
    printFlattenTree(root);
    cout << endl;

    return 0;
}

```

Time Complexity: $O(n)$, where n is the number of nodes in the binary tree. Each node is visited once during the flattening process.

Space Complexity: $O(\log_2 N)$ where N is the number of nodes in the Binary Tree. There are no additional data structures or space used but the auxiliary stack space is used during recursion. Since the recursion depth can be at most equal to the height to the Binary Tree, the space complexity is $O(H)$ where H is the height of the Binary Tree. In the ideal case, $H = \log_2 N$ and in the worst case $H = N$ (skewed tree).

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;

    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    void flatten(TreeNode* root) {
        TreeNode* curr = root;
        while (curr) {
            if (curr->left) {

```

```

        TreeNode* pre = curr->left;
        while (pre->right) {
            pre = pre->right;
        }
        pre->right = curr->right;
        curr->right = curr->left;
        curr->left = NULL;
    }
    curr = curr->right;
}
};

void printPreorder(TreeNode* root){
    if(!root){
        return;
    }
    cout << root->val << " ";
    printPreorder(root->left);
    printPreorder(root->right);
}

void printFlattenTree(TreeNode* root){
    if(!root){
        return;
    }
    cout << root->val << " ";
    printFlattenTree(root->right);
}

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->left->right->right = new TreeNode(6);
    root->right->right = new TreeNode(7);

```

```

    root->right->left = new TreeNode(8);
    Solution sol;
    cout << "Binary Tree Preorder: ";
    printPreorder(root);
    cout << endl;
    sol.flatten(root);
    cout << "Binary Tree After Flatten: ";
    printFlattenTree(root);
    cout << endl;
    return 0;
}

```

Time Complexity: $O(2N)$ where N is the number of nodes in the Binary Tree. The time complexity is linear, as each node is visited at most twice (once for establishing the temporary link and once for reverting it). In each step, we perform constant-time operations, such as moving to the left or right child and updating pointers.

Space Complexity: $O(1)$ The space complexity is constant, as the algorithm uses only a constant amount of extra space irrespective of the input size. Morris Traversal does not use any additional data structures like stacks or recursion, making it an in-place algorithm. The only space utilised is for a few auxiliary variables, such as pointers to current and in-order predecessor nodes.

Search in a Binary Search Tree

```

#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int data) {
        val = data;
        left = right = nullptr;
    }
};

class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int
target) {

```

```

        while (root != nullptr && root->val != target)
        {
            if (target < root->val) {
                root = root->left;
            }
            else {
                root = root->right;
            }
        }
        return root;
    }
};

```

```

int main() {
    TreeNode* root = new TreeNode(4);
    root->left = new TreeNode(2);
    root->right = new TreeNode(7);
    root->left->left = new TreeNode(1);
    root->left->right = new TreeNode(3);
    Solution obj;
    TreeNode* result = obj.searchBST(root, 2);
    if (result)
        cout << "Node found: " << result->val << endl;
    else
        cout << "Node not found" << endl;
    return 0;
}

```

Time Complexity: $O(\log N)$, Each step eliminates half of the tree, just like binary search. However, in the worst case (unbalanced tree), it could be $O(N)$.

Space Complexity: $O(1)$, Iterative solution uses constant space as no recursion stack is involved.

Ceil in a Binary Search Tree

```

#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;

```

```

    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    int findCeil(TreeNode* root, int key){
        int ceil = -1;
        while(root){
            if(root->val == key){
                ceil = root->val;
                return ceil;
            }
            if(key > root->val){
                root = root->right;
            }
            else{
                ceil = root->val;
                root = root->left;
            }
        }
        return ceil;
    }
};

void printInOrder(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    printInOrder(root->left);
    cout << root->val << " ";
    printInOrder(root->right);
}

int main() {
    TreeNode* root = new TreeNode(10);
    root->left = new TreeNode(5);

```

```

root->right = new TreeNode(13);
root->left->left = new TreeNode(3);
root->left->left->left = new TreeNode(2);
root->left->left->right = new TreeNode(4);
root->left->right = new TreeNode(6);
root->left->right->right = new TreeNode(9);
root->right->left = new TreeNode(11);
root->right->right = new TreeNode(14);
cout << "Binary Search Tree: " << endl;
printInOrder(root);
cout << endl;
Solution solution;
int target = 8;
int ceil = solution.findCeil(root, target);
if(ceil != -1){
    cout << "Ceiling of " << target << " is: " <<
ceil << endl;
}
else{
    cout << "No ceiling found!";
}
return 0;
}

```

Time Complexity: $O(\log_2 N)$, where N is the number of nodes in the Binary Search Tree. The complexity is equivalent to the height of the tree.

Space Complexity: $O(1)$, since the algorithm does not use any additional space or data structures.

Floor in a Binary Search Tree

```

#include <bits/stdc++.h>
using namespace std;
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}

```

```

};
class Solution {
public:
    int floorInBST(TreeNode* root, int key){
        int floor = -1;
        while(root){
            if(root->val == key){
                floor = root->val;
                return floor;
            }
            if(key > root->val){
                floor = root->val;
                root = root->right;
            }
            else{
                root = root->left;
            }
        }
        return floor;
    }
};
void printInOrder(TreeNode* root) {
    if (root == nullptr) return;
    printInOrder(root->left);
    cout << root->val << " ";
    printInOrder(root->right);
}
int main() {
    TreeNode* root = new TreeNode(10);
    root->left = new TreeNode(5);
    root->right = new TreeNode(13);
    root->left->left = new TreeNode(3);
    root->left->left->left = new TreeNode(2);
    root->left->left->right = new TreeNode(4);
    root->left->right = new TreeNode(6);
    root->left->right->right = new TreeNode(9);

```

```

root->right->left = new TreeNode(11);
root->right->right = new TreeNode(14);
cout << "Binary Search Tree (Inorder): " << endl;
printInOrder(root);
cout << endl;
Solution solution;
int target = 8;
int floorVal = solution.floorInBST(root, target);
if(floorVal != -1){
    cout << "Floor of " << target << " is: " <<
floorVal << endl;
}
else{
    cout << "No floor found!" << endl;
}
return 0;
}

```

Time Complexity: $O(\log_2 N)$, where N is the number of nodes in the Binary Search Tree. The complexity is equivalent to the height of the tree.
Space Complexity: $O(1)$, since the algorithm does not use any additional space or data structures.

Insert a given Node in Binary Search Tree

```

class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* root, int
val) {
        if(root == NULL) return new
TreeNode(val);
        TreeNode *cur = root;
        while(true) {
            if(cur->val <= val) {
                if(cur->right != NULL) cur = cur-
>right;
            }
            else {
                cur->right = new TreeNode(val);
                break;
            }
        }
    }
}

```

```

    }
    } else {
        if(cur->left != NULL) cur = cur-
>left;
        else {
            cur->left = new TreeNode(val);
            break;
        }
    }
    return root;
};

```

Delete a Node in Binary Search Tree

```

class Solution {
public:
    TreeNode* deleteNode(TreeNode* root, int
key) {
        if (root == NULL) {
            return NULL;
        }
        if (root->val == key) {
            return helper(root);
        }
        TreeNode *dummy = root;
        while (root != NULL) {
            if (root->val > key) {
                if (root->left != NULL && root-
>left->val == key) {
                    root->left = helper(root->left);
                    break;
                }
            }
            else {
                root = root->left;
            }
        }
        return dummy;
    }
};

```

```

        if (root->right != NULL &&
root->right->val key) {
            root->right helper(root-
>right);

            break;
        } else {
            root root->right;
        }
    }
}

return dummy;
}

TreeNode* helper (TreeNode* root) {
    if (root->left == NULL) {
        return root->right;
    }
    else if (root->right == NULL) {
        return root->left;
    }
    TreeNode* rightChild root->right;
    TreeNode* lastRight
findLastRight(root->left);
    lastRight->right rightChild;
    return root->left;
}

TreeNode* findLastRight (TreeNode* root) {
    if (root->right == NULL) {
        return root;
    }
    return findlastRight(root->right);
}
};

```

Kth largest/smallest element in Binary Search Tree

Brute Force

```
#include <bits/stdc++.h>
```

```

using namespace std;

struct TreeNode {
    int data;
    TreeNode *left;
    TreeNode *right;

    TreeNode(int val) : data(val), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    void inorderTraversal(TreeNode* node,
vector<int>& values) {
        if (node) {
            inorderTraversal(node->left, values);
            values.push_back(node->data);
            inorderTraversal(node->right, values);
        }
    }

    vector<int> kLargesSmall(TreeNode* root, int
k) {
        vector<int> values;
        inorderTraversal(root, values);
        int kth_smallest = values[k - 1];
        int kth_largest = values[values.size() - k];
        return {kth_smallest, kth_largest};
    }
};

int main() {
    TreeNode* root = new TreeNode(3);
    root->left = new TreeNode(1);
    root->left->right = new TreeNode(2);
    root->right = new TreeNode(4);
    Solution solution;

    int k = 1;
    vector<int> result = solution.kLargesSmall(root,
k);
    cout << "[" << result[0] << ", " << result[1] <<
 "]" << endl; // Output: [1, 4]

```

```

    return 0;
}

```

Time Complexity: $O(N)$, where N is the number of nodes in the BST. Because the code performs an in-order traversal of the BST, which requires $O(N)$ time.

Space Complexity: $O(N)$, since the code stores all the node values in a list.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int data;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int val) : data(val), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        this->k = k;
        this->result = -1;
        inorder(root);
        return result;
    }
    int kthLargest(TreeNode* root, int k) {
        this->k = k;
        this->result = -1;
        reverse_inorder(root);
        return result;
    }
    vector<int> kLargesSmall(TreeNode* root, int k)
{
    return {kthSmallest(root, k), kthLargest(root,
k)};
}

private:
    int k;
    int result;

```

```

void inorder(TreeNode* node) {
    if (node != nullptr) {
        inorder(node->left);
        if (--k == 0) {
            result = node->data;
            return;
        }
        inorder(node->right);
    }
}

void reverse_inorder(TreeNode* node) {
    if (node != nullptr) {
        reverse_inorder(node->right);
        if (--k == 0) {
            result = node->data;
            return;
        }
        reverse_inorder(node->left);
    }
};

int main() {
    TreeNode* root = new TreeNode(3);
    root->left = new TreeNode(1);
    root->left->right = new TreeNode(2);
    root->right = new TreeNode(4);
    Solution solution;
    int k = 1;
    vector<int> result = solution.kLargesSmall(root,
k);
    cout << "[" << result[0] << ", " << result[1] <<
"]" << endl;
    return 0;
}

```

Time Complexity: $O(N)$, where N is the number of nodes in the binary tree. The reason is that in the worst-case scenario, the inorder and reverse inorder traversals visit each node exactly once.

Space Complexity: $O(H)$, where H is the height of the binary tree.

Check if a tree is a BST or BT

```
class Solution {
    public boolean isValidBST(TreeNode root) {
        return isValidBST(root,
            Long.MIN_VALUE, Long.MAX_VALUE);
    }

    public boolean isValidBST(TreeNode root,
        long minVal, long maxVal) {
        if (root == null) return true;
        if (root.val >= maxVal || root.val <=
            minVal) return false;
        return isValidBST(root.left, minVal,
            root.val) && isValidBST(root.right, root.val,
            maxVal);
    }
}
```

LCA in Binary Search Tree

```
class Solution {
    public TreeNode lowestCommonAncestor
        (TreeNode root, TreeNode p, TreeNode q) {
        if (root == null) return null;
        int curr = root.val;
        if (curr > p.val && curr > q.val) {
            return lowestCommonAncestor
                (root.right, p, q);
        }
        if (curr < p.val && curr < q.val) {
            return lowestCommonAncestor
                (root.left, p, q);
        }
        return root;
    }
}
```

Construct a BST from a preorder traversal

```
class Solution {
public:
    TreeNode bstFromPreorder (vector<int>& A)
    {
        int i = 0;
        return build(A, i, INT_MAX);
    }

    TreeNode* build(vector<int>& A, int& i, int
        bound) {
        if (i == A.size() || A[i] > bound) return
            NULL;
        TreeNode* root = new
            TreeNode(A[i++]);
        root->left = build(A, i, root->val);
        root->right = build(A, i, bound);
        return root;
    }
};
```

Inorder Successor/Predecessor in BST

Brute Force

```
#include <bits/stdc++.h>
using namespace std;
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr),
        right(nullptr) {}
};
class Solution {
public:
    TreeNode* inorderSuccessor(TreeNode* root,
        TreeNode* p) {
        vector<int> inorder;
```

```

inorderTraversal(root, inorder);
int idx = binarySearch(inorder, p->val);
if (idx == (int)inorder.size() - 1 || idx == -1) {
    return nullptr;
}
return new TreeNode(inorder[idx + 1]);
}

void inorderTraversal(TreeNode* root,
vector<int>& inorder) {
    if (root == nullptr) return;
    inorderTraversal(root->left, inorder);
    inorder.push_back(root->val);
    inorderTraversal(root->right, inorder);
}

int binarySearch(vector<int>& arr, int target) {
    int left = 0, right = (int)arr.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid;
        else if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return left == (int)arr.size() ? -1 : left;
}

};

void printInOrder(TreeNode* root) {
    if (root == nullptr) return;
    printInOrder(root->left);
    cout << root->val << " ";
    printInOrder(root->right);
}

int main() {
    TreeNode* root = new TreeNode(5);
    root->left = new TreeNode(3);
    root->right = new TreeNode(6);
    root->left->left = new TreeNode(2);

```

```

root->left->right = new TreeNode(4);
root->right->right = new TreeNode(7);
cout << "BST: ";
printInOrder(root);
cout << endl;
TreeNode* p = root->left->right;
Solution solution;
TreeNode* successor =
solution.inorderSuccessor(root, p);
if (successor != nullptr) {
    cout << "Inorder Successor of " << p->val <<
" is: " << successor->val << endl;
} else {
    cout << "Inorder Successor of " << p->val <<
" does not exist." << endl;
}
return 0;
}

```

Time Complexity: $O(N + \log N)$ where N is the number of nodes of the binary search tree. $O(N)$ to traverse all nodes and store them in an inorder array and $O(\log N)$ for the binary search.

Space Complexity: $O(N)$ as an array of size N is used to store the inorder traversal of the binary search tree

Better Approach

```

#include <bits/stdc++.h>
using namespace std;
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}
};
class Solution {
public:
    TreeNode* inorderSuccessor(TreeNode* root,
TreeNode* p) {
        TreeNode* successor = nullptr;
        while (root != nullptr) {

```

```

        if (root->val > p->val) {
            successor = root;
            root = root->left;
        }
        else {
            root = root->right;
        }
    }
    return successor;
}

};

void printInOrder(TreeNode* root) {
    if (root == nullptr) return;
    printInOrder(root->left);
    cout << root->val << " ";
    printInOrder(root->right);
}

int main() {
    TreeNode* root = new TreeNode(5);
    root->left = new TreeNode(3);
    root->right = new TreeNode(6);
    root->left->left = new TreeNode(2);
    root->left->right = new TreeNode(4);
    root->right->right = new TreeNode(7);
    cout << "BST: ";
    printInOrder(root);
    cout << endl;
    TreeNode* p = root->left->right;
    Solution solution;
    TreeNode* successor =
    solution.inorderSuccessor(root, p);

    if (successor != nullptr) {
        cout << "Inorder Successor of " << p->val <<
" is: " << successor->val << endl;
    } else {
        cout << "Inorder Successor of " << p->val <<
" does not exist." << endl;
    }
}

```

```

    }
    return 0;
}

```

Time Complexity: $O(N)$ where N is the number of nodes in the binary search tree. This complexity arises from the fact that we have to traverse all nodes in an inorder fashion to get to the inorder successor.
Space Complexity: $O(1)$ as no additional data structure or memory allocation is done during the traversal and algorithm. Only a value comparison at each node.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}
};

class Solution {
public:
    TreeNode* inorderSuccessor(TreeNode* root,
TreeNode* p) {
        TreeNode* successor = nullptr;
        while (root != nullptr) {
            if (p->val >= root->val) {
                root = root->right;
            }
            else {
                successor = root;
                root = root->left;
            }
        }
        return successor;
    }
};

void printInOrder(TreeNode* root) {
    if (root == nullptr) return;
}

```

```

    printInOrder(root->left);
    cout << root->val << " ";
    printInOrder(root->right);
}

int main() {
    TreeNode* root = new TreeNode(5);
    root->left = new TreeNode(3);
    root->right = new TreeNode(6);
    root->left->left = new TreeNode(2);
    root->left->right = new TreeNode(4);
    root->right->right = new TreeNode(7);
    cout << "BST: ";
    printInOrder(root);
    cout << endl;
    TreeNode* p = root->left->right;
    Solution solution;

    TreeNode* successor =
    solution.inorderSuccessor(root, p);

    if (successor != nullptr) {
        cout << "Inorder Successor of " << p->val <<
        " is: " << successor->val << endl;
    } else {
        cout << "Inorder Successor of " << p->val <<
        " does not exist." << endl;
    }

    return 0;
}

```

Time Complexity: $O(H)$ where H is the height of the binary search tree as we are traversing along the height of the tree.

Space Complexity: $O(1)$ as no additional data structure or memory allocation is done during the traversal and algorithm.

Merge 2 BSTs

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

struct Node {

```

```

    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(NULL),
    right(NULL) {}
};

class Solution {
public:
    void traverse(Node* root, vector<int>&
    elements) {
        if (!root) return;
        traverse(root->left, elements);
        elements.push_back(root->data);
        traverse(root->right, elements);
    }

    vector<int> mergeBSTs(Node* root1, Node*
    root2) {
        vector<int> elements;
        traverse(root1, elements);
        traverse(root2, elements);
        sort(elements.begin(), elements.end());
        return elements;
    }
};

int main() {
    Node* root1 = new Node(2);
    root1->left = new Node(1);
    root1->right = new Node(4);
    Node* root2 = new Node(3);
    root2->left = new Node(0);
    root2->right = new Node(5);

    Solution sol;

    vector<int> result = sol.mergeBSTs(root1,
    root2);

    for (int val : result) {
        cout << val << " ";
    }
}

```

```
}
```

Time Complexity: $O((n+m) \cdot \log(n+m))$, we traverse both the BSTs and sort all the elements.

Space Complexity: $O(m+n)$, additional space required for storing elements of the two BSTs.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(NULL),
right(NULL) {}
};

class Solution {
public:
    void inorderTraversal(Node* root, vector<int>&
arr) {
        if (!root) return;
        inorderTraversal(root->left, arr);
        arr.push_back(root->data);
        inorderTraversal(root->right, arr);
    }

    vector<int> mergeArrays(vector<int>& arr1,
vector<int>& arr2) {
        vector<int> merged;
        int i = 0, j = 0;
        while (i < arr1.size() && j < arr2.size()) {
            if (arr1[i] < arr2[j])
merged.push_back(arr1[i++]);
            else merged.push_back(arr2[j++]);
        }
        while (i < arr1.size())
merged.push_back(arr1[i++]);
        while (j < arr2.size())
merged.push_back(arr2[j++]);
        return merged;
    }
}
```

```
vector<int> mergeBSTs(Node* root1, Node*
root2) {
```

```
    vector<int> arr1, arr2;
    inorderTraversal(root1, arr1);
    inorderTraversal(root2, arr2);
    return mergeArrays(arr1, arr2);
}
```

```
};
```

```
int main() {
    Node* root1 = new Node(3);
    root1->left = new Node(1);
    root1->right = new Node(5);
    Node* root2 = new Node(4);
    root2->left = new Node(2);
    root2->right = new Node(6);
    Solution sol;
    vector<int> result = sol.mergeBSTs(root1,
root2);
    for (int val : result) cout << val << " ";
    return 0;
}
```

Time Complexity: $O(n+m)$, we traverse both the BSTs and merge two sorted lists.

Space Complexity: $O(m+n)$, additional space required for storing elements of the two BSTs.

Two Sum In BST | Check if there exists a pair with Sum K

Brute Force Approach

```
#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}
};
```

```

class Solution {
public:
    bool findTarget(TreeNode* root, int k) {
        vector<int> inorder;
        inorderTraversal(root, inorder);
        int left = 0;
        int right = inorder.size() - 1;
        while (left < right) {
            int sum = inorder[left] + inorder[right];
            if (sum == k) {
                return true;
            }
            else if (sum < k) {
                left++;
            }
            else {
                right--;
            }
        }
        return false;
    }
private:
    void inorderTraversal(TreeNode* root,
        vector<int>& inorder) {
        if (!root) return;
        inorderTraversal(root->left, inorder);
        inorder.push_back(root->val);
        inorderTraversal(root->right, inorder);
    }
};

void printInOrder(TreeNode* root) {
    if (!root) return;
    printInOrder(root->left);
    cout << root->val << " ";
    printInOrder(root->right);
}

```

```

int main() {
    TreeNode* root = new TreeNode(7);
    root->left = new TreeNode(3);
    root->right = new TreeNode(15);
    root->right->left = new TreeNode(9);
    root->right->right = new TreeNode(20);
    cout << "Tree Initialized: ";
    printInOrder(root);
    cout << endl;
    Solution solution;
    int target = 22;
    bool exists = solution.findTarget(root, target);
    if (exists) cout << "Pair with sum " << target <<
        " exists." << endl;
    else cout << "Pair with sum " << target << " does
        not exist." << endl;
    return 0;
}

```

Time Complexity: $O(N+N)$ where N is the number of nodes in the Binary Search Tree. To create the array that will store the inorder sequence, we have to traverse the entire BST, hence $O(N)$ and to apply the two pointer approach and get the pair equal to sum again requires $O(N)$ hence $O(N+N) \sim O(2N) \sim O(N)$.

Space Complexity : $O(N)$ where N is the number of nodes in the BST, as we have to store all the nodes in an additional data structure array. The two pointer approach does not use any additional space hence the space complexity is $O(N)$.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr),
        right(nullptr) {}
};

class BSTIterator {

```

private:

```
stack<TreeNode*> myStack;
```

```
bool reverse;
```

public:

```
BSTIterator(TreeNode* root, bool isReverse) :  
reverse(isReverse) {
```

```
    pushAll(root);
```

```
}
```

```
bool hasNext() {
```

```
    return !myStack.empty();
```

```
}
```

```
int next() {
```

```
    TreeNode* tmpNode = myStack.top();
```

```
    myStack.pop();
```

```
    if (!reverse) {
```

```
        pushAll(tmpNode->right);
```

```
    }
```

```
    else {
```

```
        pushAll(tmpNode->left);
```

```
    }
```

```
    return tmpNode->val;
```

```
}
```

private:

```
void pushAll(TreeNode* node) {
```

```
    while (node != nullptr) {
```

```
        myStack.push(node);
```

```
        if (reverse) {
```

```
            node = node->right;
```

```
        }
```

```
        else {
```

```
            node = node->left;
```

```
        }
```

```
    }
```

```
}
```

```
};
```

```
class Solution {
```

public:

```
bool findTarget(TreeNode* root, int k) {
```

```
    if (!root) return false;
```

```
    BSTIterator l(root, false);
```

```
    BSTIterator r(root, true);
```

```
    int i = l.next();
```

```
    int j = r.next();
```

```
    while (i < j) {
```

```
        if (i + j == k) return true;
```

```
        else if (i + j < k) i = l.next();
```

```
        else j = r.next();
```

```
    }
```

```
    return false;
```

```
}
```

```
};
```

```
void printInOrder(TreeNode* root) {
```

```
    if (!root) return;
```

```
    printInOrder(root->left);
```

```
    cout << root->val << " ";
```

```
    printInOrder(root->right);
```

```
}
```

```
int main() {
```

```
    TreeNode* root = new TreeNode(7);
```

```
    root->left = new TreeNode(3);
```

```
    root->right = new TreeNode(15);
```

```
    root->right->left = new TreeNode(9);
```

```
    root->right->right = new TreeNode(20);
```

```
    cout << "Tree Initialized: ";
```

```
    printInOrder(root);
```

```
    cout << endl;
```

```
    Solution solution;
```

```
    int target = 22;
```

```
    bool exists = solution.findTarget(root, target);
```

```
    if (exists) cout << "Pair with sum " << target <<  
" exists." << endl;
```

```
    else cout << "Pair with sum " << target << " does  
not exist." << endl;
```

```

    return 0;
}

```

Time Complexity: $O(N)$ where N is the number of nodes in the BST as we have to traverse all the nodes using the i and j pointers to find the pair with sum 'k'.

Space Complexity : $O(H)$ where H is the height of the Binary Search Tree as the BSTIterator class uses a stack to store the nodes. At maximum the size of such a stack will be equal to the height of the Binary Tree.

Recover BST | Correct BST with two nodes swapped

```

class Solution {
private:
    TreeNode* first;
    TreeNode* prev;
    TreeNode* middle;
    TreeNode* last;
private:
    void inorder (TreeNode* root) {
        if(root == NULL) return;
        inorder(root->left);
        if (prev != NULL && (root->val < prev->val))
        {
            if (first == NULL){
                first = prev;
                middle = root;
            }
            else
                last = root;
        }
        prev = root;
        inorder(root->right);
    }
public:
    void recoverTree (TreeNode* root) {
        first = middle last = NULL;
        prev = new TreeNode(INT_MIN);

```

```

        inorder(root);
        if(first && last) swap(first->val, last->val);
        else if(first && middle) swap(first->val, middle->val);
    }
};

```

Largest BST in Binary Tree

```

class NodeValue {
public:
    int maxNode, minNode, maxSize;
    NodeValue(int minNode, int maxNode, int maxSize) {
        this->maxNode = maxNode;
        this->minNode = minNode;
        this->maxSize = maxSize;
    }
};
class Solution {
private:
    NodeValue largestBSTSubtreeHelper
(TreeNode* root) {
        if (!root) {
            return NodeValue(INT_MAX, INT_MIN, 0);
        }
        auto left = largestBSTSubtreeHelper
(root->left);
        auto right = largestBSTSubtreeHelper
(root->right);
        if (left.maxNode < root->val && root->val < right.minNode) {
            return NodeValue(min(root->val, left.minNode), max(root->val, right.maxNode),
                left.maxSize
                right.maxSize + 1);
        }
        return NodeValue(INT_MIN, INT_MAX, max(left.maxSize, right.maxSize));
    }
};

```

```

}
public:
int largestBSTSubtree (TreeNode* root) {
    return
largestBSTSubtreeHelper(root).maxSize;
    }
};

```

Breadth First Search (BFS): Level Order Traversal

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    vector<int> bfsOfGraph(int V, vector<int> adj[])
    {
        int vis[V] = {0};
        vis[0] = 1;
        queue<int> q;
        q.push(0);
        vector<int> bfs;
        while(!q.empty()) {
            int node = q.front();
            q.pop();
            bfs.push_back(node);
            for(auto it : adj[node]) {
                if(!vis[it]) {
                    vis[it] = 1;
                    q.push(it);
                }
            }
        }
        return bfs;
    }
};
void addEdge(vector<int> adj[], int u, int v) {

```

```

adj[u].push_back(v);
adj[v].push_back(u);
}
void printAns(vector<int> &ans) {
    for (int i = 0; i < ans.size(); i++) {
        cout << ans[i] << " ";
    }
}
int main() {
    vector<int> adj[6];
    addEdge(adj, 0, 1);
    addEdge(adj, 1, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 0, 4);
    Solution obj;
    vector<int> ans = obj.bfsOfGraph(5, adj);
    printAns(ans);
    return 0;
}

```

Depth First Search (DFS)

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    void dfs(int v, vector<int> adj[],
            vector<int>& visited,
            vector<int>& result) {
        visited[v] = 1;
        result.push_back(v);
        for (int u : adj[v]) {
            if (!visited[u]) {
                dfs(u, adj, visited, result);
            }
        }
    }
}

```

```

    }
};

int main() {
    int V = 5;
    vector<int> adj[V];
    adj[0] = {1, 2};
    adj[1] = {0, 3};
    adj[2] = {0, 4};
    adj[3] = {1};
    adj[4] = {2};
    vector<int> visited(V, 0);
    vector<int> result;
    Solution sol;
    sol.dfs(0, adj, visited, result);
    for (int x : result) cout << x << " ";
    cout << endl;
    return 0;
}

```

Time Complexity: $O(V+E)$, each vertex is visited once and every edge is checked once in the adjacency list.
Space Complexity: $O(V)$, additional amount of space required for recursion stack.

Number of Provinces

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    void dfs(int node, vector<int> adjList[], int visited[]) {
        visited[node] = 1;
        for (auto neighbor : adjList[node]) {
            if (!visited[neighbor]) {
                dfs(neighbor, adjList, visited);
            }
        }
    }
}

```

```

public:
    int numProvinces(vector<vector<int>> adj, int V) {
        vector<int> adjList[V];
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (adj[i][j] == 1 && i != j) {
                    adjList[i].push_back(j);
                    adjList[j].push_back(i);
                }
            }
        }
        int visited[V] = {0};
        int count = 0;
        for (int i = 0; i < V; i++) {
            if (!visited[i]) {
                count++;
                dfs(i, adjList, visited);
            }
        }
        return count;
    }
};

int main() {
    vector<vector<int>> adj = {
        {1, 0, 1},
        {0, 1, 0},
        {1, 0, 1}
    };
    int V = 3;
    Solution obj;
    cout << obj.numProvinces(adj, V) << endl;
    return 0;
}

```

Time Complexity: $O(V+E)$, we visit every node and for every node we visit all of its neighbours in the DFS traversal.

Space Complexity: $O(N)$, for storing visited array and auxiliary stack space.

Connected Components

```
#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int countComponents(int V,
        vector<vector<int>>& edges) {
        vector<vector<int>> adj(V);
        for (auto &e : edges) {
            adj[e[0]].push_back(e[1]);
            adj[e[1]].push_back(e[0]);
        }
        vector<int> visited(V, 0);
        int components = 0;
        for (int i = 0; i < V; ++i) {
            if (!visited[i]) {
                components++;
                queue<int> q;
                q.push(i);
                visited[i] = 1;
                while (!q.empty()) {
                    int node = q.front();
                    q.pop();
                    for (auto &nbr : adj[node]) {
                        if (!visited[nbr]) {
                            visited[nbr] = 1;
                            q.push(nbr);
                        }
                    }
                }
            }
        }
        return components;
    }
};
```

```
    }
};
int main() {
    int V = 5;
    vector<vector<int>> edges =
        {{0,1},{1,2},{3,4}};
    Solution sol;
    cout << "Number of Connected Components: "
        << sol.countComponents(V, edges) << endl;
    return 0;
}
```

Time Complexity: $O(V+E)$, Each vertex is visited exactly once, and each edge is processed at most twice (once from each end).

Space Complexity: $O(V+E)$, To build Adjacency List.

Rotten Oranges : Min time to rot all oranges : BFS

```
#include <bits/stdc++.h>
using namespace std;
int orangesRotting(vector<vector<int>>& grid) {
    if (grid.empty()) return 0;
    int m = grid.size();
    int n = grid[0].size();
    int days = 0;
    int tot = 0;
    int cnt = 0;
    queue<pair<int, int>> rotten;
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (grid[i][j] != 0)
                tot++;
            if (grid[i][j] == 2)
                rotten.push({i, j});
        }
    }
    int dx[4] = {0, 0, 1, -1};
```

```

int dy[4] = {1, -1, 0, 0};
while (!rotten.empty()) {
    int k = rotten.size();
    cnt += k;
    while (k-- > 0) {
        int x = rotten.front().first;
        int y = rotten.front().second;
        rotten.pop();
        for (int i = 0; i < 4; ++i) {
            int nx = x + dx[i];
            int ny = y + dy[i];
            if (nx < 0 || ny < 0 || nx >= m || ny >= n ||
                grid[nx][ny] != 1)
                continue;
            grid[nx][ny] = 2;
            rotten.push({nx, ny});
        }
    }
    if (!rotten.empty())
        days++;
}
return tot == cnt ? days : -1;
}

int main() {
    vector<vector<int>> v{{2, 1, 1},
                        {1, 1, 0},
                        {0, 1, 1}};

    int rotting = orangesRotting(v);

    cout << "Minimum Number of Minutes
    Required " << rotting << endl;

    return 0;
}

```

Time Complexity: $O(n \times n \times 4)$ In the worst case, every cell in the grid may contain an orange, and for each rotten orange we explore 4 directions (up, down, left, right). So we iterate through all cells ($n \times n$) and perform 4 operations per orange. Hence, the complexity becomes $O(n \times n \times 4)$, which simplifies to $O(n^2)$.

Space Complexity: $O(n \times n)$ In the worst case, all the oranges might be rotten and will be stored in the queue simultaneously. The maximum size of the queue can be equal to the total number of oranges in the grid, i.e., $n \times n$. Therefore, the space complexity is $O(n^2)$.

Flood Fill Algorithm – Graphs

```

#include<bits/stdc++.h>

using namespace std;

class Solution {
private:
    void dfs(int row, int col,
            vector<vector<int>>&ans,
            vector<vector<int>>& image, int newColor, int
            delRow[], int delCol[],
            int iniColor) {
        ans[row][col] = newColor;
        int n = image.size();
        int m = image[0].size();
        for(int i = 0; i < 4; i++) {
            int nrow = row + delRow[i];
            int ncol = col + delCol[i];
            if(nrow >= 0 && nrow < n && ncol >= 0 &&
                ncol < m &&
                image[nrow][ncol] == iniColor &&
                ans[nrow][ncol] != newColor) {
                dfs(nrow, ncol, ans, image, newColor,
                    delRow, delCol, iniColor);
            }
        }
    }

public:
    vector<vector<int>>
    floodFill(vector<vector<int>>& image,
            int sr, int sc, int newColor) {
        int iniColor = image[sr][sc];
        vector<vector<int>> ans = image;
        int delRow[] = {-1, 0, +1, 0};
        int delCol[] = {0, +1, 0, -1};
    }
}

```

```

        dfs(sr, sc, ans, image, newColor, delRow,
delCol, iniColor);

        return ans;
    }
};

int main(){
    vector<vector<int>>>image{
        {1,1,1},
        {1,1,0},
        {1,0,1}
    };

    Solution obj;

    vector<vector<int>>> ans = obj.floodFill(image,
1, 1, 2);

    for(auto i: ans){
        for(auto j: i)
            cout << j << " ";

            cout << "\n";
        }

        return 0;
    }

```

Time Complexity: $O(N \times M + N \times M \times 4) \sim O(N \times M)$, For the worst case, all of the pixels will have the same colour, so DFS function will be called for $(N \times M)$ nodes and for every node we are traversing for 4 neighbours, so it will take $O(N \times M \times 4)$ time.

Space Complexity: $O(N \times M) + O(N \times M)$, $O(N \times M)$ for copied input array and recursive stack space takes up $N \times M$ locations at max.

Detect Cycle in an Undirected Graph (using BFS)

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    bool detect(int src, vector<int> adj[], int vis[]) {
        vis[src] = 1;

        queue<pair<int,int>> q;

```

```

        q.push({src, -1});

        while(!q.empty()) {
            int node = q.front().first;
            int parent = q.front().second;

            q.pop();

            for(auto adjacentNode: adj[node]) {
                if(!vis[adjacentNode]) {
                    vis[adjacentNode] = 1;
                    q.push({adjacentNode, node});
                }
                else if(parent != adjacentNode) {
                    return true;
                }
            }
        }

        return false;
    }

public:
    bool isCycle(int V, vector<int> adj[]) {
        int vis[V] = {0};

        for(int i = 0; i < V; i++) {
            if(!vis[i]) {
                if(detect(i, adj, vis)) return true;
            }
        }

        return false;
    }
};

int main() {
    vector<int> adj[4] = {{}, {2}, {1, 3}, {2}};

    Solution obj;

    bool ans = obj.isCycle(4, adj);

    if (ans)
        cout << "1\n";
    else
        cout << "0\n";
}

```

```

    return 0;
}

```

Detect Cycle in an Undirected Graph (using DFS)

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    bool dfs(int node, int parent, vector<int> adj[],
vector<int>& visited) {
        visited[node] = 1;
        for (int neighbor : adj[node]) {
            if (!visited[neighbor]) {
                if (dfs(neighbor, node, adj, visited))
                    return true;
            }
            else if (neighbor != parent) {
                return true;
            }
        }
        return false;
    }
    bool isCycle(int V, vector<int> adj[]) {
        vector<int> visited(V, 0);
        for (int i = 0; i < V; i++) {
            if (!visited[i]) {
                if (dfs(i, -1, adj, visited)) return true;
            }
        }
        return false;
    }
};
int main() {
    int V = 5;
    vector<int> adj[V];

```

```

    adj[0].push_back(1);
    adj[1].push_back(0);
    adj[1].push_back(2);
    adj[2].push_back(1);
    adj[2].push_back(3);
    adj[3].push_back(2);
    adj[3].push_back(4);
    adj[4].push_back(3);
    adj[4].push_back(1);

    Solution sol;
    if (sol.isCycle(V, adj))
        cout << "Cycle detected\n";
    else
        cout << "No cycle found\n";
    return 0;
}

```

Time Complexity: $O(V+E)$, we build the adjacency list and explore all the edges and visit all the vertices once during DFS traversal.

Space Complexity: $O(V+E)$, additional space is used to store adjacency list, visited array and recursive call stack for DFS traversal.

Distance of Nearest Cell having 1

```

#include<bits/stdc++.h>
using namespace std;
class Solution
{
public:
    vector<vector<int>>nearest(vector<vector<int>>gr
id){
        int n = grid.size();
        int m = grid[0].size();
        vector<vector<int>> vis(n, vector<int>(m,
0));
        vector<vector<int>> dist(n, vector<int>(m,
0));
        queue<pair<pair<int,int>, int>> q;
        for(int i = 0;i<n;i++) {

```

```

        for(int j = 0;j<m;j++) {
            if(grid[i][j] == 1) {
                q.push({ {i,j}, 0});
                vis[i][j] = 1;
            }
            else {
                vis[i][j] = 0;
            }
        }
        if(grid[i][j] == 1) {
            q.push({ {i,j}, 0});
            vis[i][j] = 1;
        }
        else {
            vis[i][j] = 0;
        }
    }
}

int delrow[] = {-1, 0, +1, 0};
int delcol[] = {0, +1, 0, -1};
while(!q.empty()) {
    int row = q.front().first.first;
    int col = q.front().first.second;
    int steps = q.front().second;

    q.pop();
    dist[row][col] = steps;
    for(int i = 0;i<4;i++) {
        int nrow = row + delrow[i];
        int ncol = col + delcol[i];

        if(nrow >= 0 && nrow < n && ncol >=
0 && ncol < m
        && vis[nrow][ncol] == 0) {
            vis[nrow][ncol] = 1;
            q.push({ {nrow, ncol}, steps+1});
        }
    }
}

```

```

    }
}

return dist;
}

};

int main(){
    vector<vector<int>>>grid{
        {0,1,1,0},
        {1,1,0,0},
        {0,0,1,1}
    };

    Solution obj;
    vector<vector<int>>> ans = obj.nearest(grid);

    for(auto i: ans){
        for(auto j: i){
            cout << j << " ";
            cout << "\n";
        }

        return 0;
    }
}

```

Time Complexity: $O(N \times M + N \times M \times 4) \sim O(N \times M)$, the BFS function will be called for $(N \times M)$ nodes, and for every node, we are traversing for 4 neighbors, so it will take $O(N \times M \times 4)$ time.

Space Complexity: $O(N \times M) + O(N \times M) + O(N \times M) \sim O(N \times M)$, for the visited array, distance matrix, and queue space takes up $N \times M$ locations at max.

Surrounded Regions | Replace O's with X's

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    void dfs(int row, int col, vector<vector<int>>& vis, vector<vector<char>>& mat, int delrow[], int delcol[]) {
        vis[row][col] = 1;

        int n = mat.size(), m = mat[0].size();
    }
}

```

```

        for (int k = 0; k < 4; k++) {
            int nrow = row + delrow[k], ncol = col +
delcol[k];

            if (nrow >= 0 && nrow < n && ncol >= 0
&& ncol < m && !vis[nrow][ncol] &&
mat[nrow][ncol] == 'O') {

                dfs(nrow, ncol, vis, mat, delrow, delcol);

            }

        }

    }

public:

    vector<vector<char>> fill(int n, int m,
vector<vector<char>> mat) {

        if (n == 0 || m == 0) return mat;

        int delrow[4] = {-1, 0, 1, 0};

        int delcol[4] = {0, 1, 0, -1};

        vector<vector<int>> vis(n, vector<int>(m,
0));

        for (int j = 0; j < m; j++) {

            if (!vis[0][j] && mat[0][j] == 'O') dfs(0, j,
vis, mat, delrow, delcol);

            if (!vis[n - 1][j] && mat[n - 1][j] == 'O')
dfs(n - 1, j, vis, mat, delrow, delcol);

        }

        for (int i = 0; i < n; i++) {

            if (!vis[i][0] && mat[i][0] == 'O') dfs(i, 0,
vis, mat, delrow, delcol);

            if (!vis[i][m - 1] && mat[i][m - 1] == 'O')
dfs(i, m - 1, vis, mat, delrow, delcol);

        }

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < m; j++) {

                if (!vis[i][j] && mat[i][j] == 'O') mat[i][j]
= 'X';

            }

        }

        return mat;

    }

};

int main() {

```

```

vector<vector<char>> mat{

    {'X','X','X','X'},

    {'X','O','X','X'},

    {'X','O','O','X'},

    {'X','O','X','X'},

    {'X','X','O','O'}

};

Solution ob;

vector<vector<char>> ans =
ob.fill((int)mat.size(), (int)mat[0].size(), mat);

for (int i = 0; i < (int)ans.size(); i++) {

    for (int j = 0; j < (int)ans[0].size(); j++) {

        cout << ans[i][j] << " ";

    }

    cout << "\n";

}

return 0;

}

```

Time Complexity: $O(N \times M)$, since each cell is visited at most once during DFS and once during the final traversal.

Space Complexity: $O(N \times M)$, due to the visited matrix and the recursion stack in the worst case.

Number of Enclaves

```

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

    int numberOfEnclaves(vector<vector<int>>&
grid) {

        if (grid.empty() || grid[0].empty()) return 0;

        int n = (int)grid.size();

        int m = (int)grid[0].size();

        vector<vector<int>> vis(n, vector<int>(m,
0));

        queue<pair<int,int>> q;

        for (int i = 0; i < n; i++) {

```

```

        for (int j = 0; j < m; j++) {
            if (i == 0 || j == 0 || i == n - 1 || j == m - 1)
            {
                if (grid[i][j] == 1) {
                    vis[i][j] = 1;
                    q.push({i, j});
                }
            }
        }
    }

    int delrow[] = {-1, 0, +1, 0};
    int delcol[] = {0, +1, 0, -1};

    while (!q.empty()) {
        auto [row, col] = q.front();

        q.pop();

        for (int k = 0; k < 4; k++) {
            int nrow = row + delrow[k];
            int ncol = col + delcol[k];

            if (nrow >= 0 && nrow < n && ncol >=
0 && ncol < m
                && !vis[nrow][ncol] &&
grid[nrow][ncol] == 1) {
                vis[nrow][ncol] = 1;
                q.push({nrow, ncol});
            }
        }
    }

    int cnt = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (grid[i][j] == 1 && vis[i][j] == 0)
cnt++;
        }
    }

    return cnt;
}
};

```

```

int main() {
    vector<vector<int>>> grid{
        {0, 0, 0, 0},
        {1, 0, 1, 0},
        {0, 1, 1, 0},
        {0, 0, 0, 0}
    };

    Solution obj;

    cout << obj.numberOfEnclaves(grid) << endl; //
Expected: 3

    return 0;
}

```

Time Complexity: $O(n \times m)$, where n is the number of rows and m is the number of columns. Each cell is processed at most once.

Space Complexity: $O(n \times m)$, due to the vis array and BFS queue in the worst case.

Word Ladder - I : G-29

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int wordLadderLength(string startWord, string
targetWord, vector<string>& wordList) {
        queue<pair<string, int>> q;

        q.push({startWord, 1});

        unordered_set<string> st(wordList.begin(),
wordList.end());

        st.erase(startWord);

        while (!q.empty()) {
            string word = q.front().first;
            int steps = q.front().second;

            q.pop();

            if (word == targetWord) return steps;

            for (int i = 0; i < word.size(); i++) {
                char original = word[i];

                for (char ch = 'a'; ch <= 'z'; ch++) {

```

```

        word[i] = ch;
        if (st.find(word) != st.end()) {
            st.erase(word);
            q.push({word, steps + 1});
        }
    }
    word[i] = original;
}
}
return 0;
}
};

int main() {
    vector<string> wordList = {"des", "der", "dfr",
    "dgt", "dfs"};

    string startWord = "der", targetWord = "dfs";

    Solution obj;

    cout << obj.wordLadderLength(startWord,
    targetWord, wordList) << endl;

    return 0;
}

```

Time Complexity: $O(N * L * 26)$, where N is the number of words in the list and L is the length of each word. For each word, we attempt to change each of its L characters to 26 possible letters.

Space Complexity: $O(N * L)$, for the set storing all words and the queue used for BFS.

G-30 : Word Ladder-II

```

#include <bits/stdc++.h>

using namespace std;

class Solution{
public:
    vector<vector<string>>> findSequences(string
    beginWord, string endWord,

                                vector<string>
    &wordList)
    {
        unordered_set<string> st(wordList.begin(),
    wordList.end());
    }
}

```

```

queue<vector<string>>> q;
q.push({beginWord});
vector<string> usedOnLevel;
usedOnLevel.push_back(beginWord);
int level = 0;
vector<vector<string>>> ans;
while (!q.empty())
{
    vector<string> vec = q.front();
    q.pop();
    if (vec.size() > level)
    {
        level++;
        for (auto it : usedOnLevel)
        {
            st.erase(it);
        }
    }
    string word = vec.back();
    if (word == endWord) {
        if (ans.size() == 0) {
            ans.push_back(vec);
        }
        else if (ans[0].size() == vec.size()){
            ans.push_back(vec);
        }
    }
    for (int i = 0; i < word.size(); i++){
        char original = word[i];
        for (char c = 'a'; c <= 'z'; c++){
            {
                word[i] = c;
                if (st.count(word) > 0)
                {
                    vec.push_back(word);
                    q.push(vec);
                }
            }
        }
    }
}

```

```

        usedOnLevel.push_back(word);
        vec.pop_back();
    }
}
word[i] = original;
}
}
return ans;
}
};
bool comp(vector<string> a, vector<string> b)
{
    string x = "", y = "";
    for (string i : a)
        x += i;
    for (string i : b)
        y += i;
    return x < y;
}
int main() {
    vector<string> wordList = {"des", "der", "dfr",
    "dgt", "dfs"};
    string startWord = "der", targetWord = "dfs";
    Solution obj;
    vector<vector<string>> ans =
obj.findSequences(startWord, targetWord,
wordList);
    if (ans.size() == 0)
        cout << -1 << endl;
    else {
        sort(ans.begin(), ans.end(), comp);
        for (int i = 0; i < ans.size(); i++) {
            for (int j = 0; j < ans[i].size(); j++) {
                cout << ans[i][j] << " ";
            }
            cout << endl;
        }
    }
}

```

```

    }
    return 0;
}

```

Time Complexity: $O(N \times L \times 26 + S \times L) \rightarrow$ dominated by generating all transformations (N = words, L = word length, S = number of shortest sequences).

Space Complexity: $O(N \times L + S \times L) \rightarrow$ for queue storing paths, set for unused words, and final sequences.

Number of Islands

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void bfs(int row, int col, vector<vector<int>>
    &vis, vector<vector<char>> &grid) {
        queue<pair<int, int>> q;
        q.push({row, col});
        vis[row][col] = 1;
        int delRow[] = {-1, -1, -1, 0, 1, 1, 1, 0};
        int delCol[] = {-1, 0, 1, 1, 1, 0, -1, -1};
        while (!q.empty()) {
            int r = q.front().first;
            int c = q.front().second;
            q.pop();
            for (int i = 0; i < 8; i++) {
                int nrow = r + delRow[i];
                int ncol = c + delCol[i];
                if (nrow >= 0 && nrow < grid.size() &&
                    ncol >= 0 && ncol < grid[0].size() &&
                    !vis[nrow][ncol] && grid[nrow][ncol]
                    == '1') {
                    vis[nrow][ncol] = 1;
                    q.push({nrow, ncol});
                }
            }
        }
    }
}

```

```

int numIslands(vector<vector<char>>& grid) {
    int n = grid.size();
    int m = grid[0].size();
    vector<vector<int>> vis(n, vector<int>(m,
0));
    int count = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (!vis[i][j] && grid[i][j] == '1') {
                count++;
                bfs(i, j, vis, grid);
            }
        }
    }
    return count;
}
};

int main() {
    vector<vector<char>> grid = {
        {'1','1','0','0','0'},
        {'1','1','0','0','0'},
        {'0','0','1','0','0'},
        {'0','0','0','1','1'}
    };
    Solution obj;
    cout << obj.numIslands(grid) << endl;
    return 0;
}

```

Time Complexity: $O(N \cdot M)$, where N is the number of rows and M is the number of columns in the grid. Every cell is visited atleast once..

Space Complexity: $O(N \cdot M)$, additional space used for storing visited matrix and queue.

Bipartite Graph | DFS Implementation

```

#include<bits/stdc++.h>

using namespace std;

```

```

class Solution {
private:
    bool dfs(int node, int col, int color[], vector<int>
adj[]) {
        color[node] = col;
        for(auto it : adj[node]) {
            if(color[it] == -1) {
                if(dfs(it, !col, color, adj) == false) return
false;
            }
            else if(color[it] == col) {
                return false;
            }
        }
        return true;
    }
public:
    bool isBipartite(int V, vector<int>adj[]){
        int color[V];
        for(int i = 0;i<V;i++) color[i] = -1;
        for(int i = 0;i<V;i++) {
            if(color[i] == -1) {
                if(dfs(i, 0, color, adj) == false)
                    return false;
            }
        }
        return true;
    }
};

void addEdge(vector <int> adj[], int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}

int main(){
    vector<int>adj[4];
    addEdge(adj, 0, 2);
    addEdge(adj, 0, 3);
}

```

```

addEdge(adj, 2, 3);
addEdge(adj, 3, 1);
Solution obj;
bool ans = obj.isBipartite(4, adj);
if(ans)cout << "1\n";
else cout << "0\n";
return 0;
}

```

Detect cycle in a directed graph (using DFS) : G 19

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
private:
bool dfsCheck(int node, vector<int> adj[], int vis[],
int pathVis[]) {
    vis[node] = 1;
    pathVis[node] = 1;
    for (auto it : adj[node]) {
        if (!vis[it]) {
            if (dfsCheck(it, adj, vis, pathVis) == true)
                return true;
        }
        else if (pathVis[it]) {
            return true;
        }
    }
    pathVis[node] = 0;
    return false;
}
public:
bool isCyclic(int V, vector<int> adj[]) {
    int vis[V] = {0};
    int pathVis[V] = {0};
    for (int i = 0; i < V; i++) {

```

```

        if (!vis[i]) {
            if (dfsCheck(i, adj, vis, pathVis) ==
true) return true;
        }
    }
    return false;
}
};
int main() {
    vector<int> adj[11] = {{}, {2}, {3}, {4, 7},
{5}, {6}, {}, {5}, {9}, {10}, {8}};
    int V = 11;
    Solution obj;
    bool ans = obj.isCyclic(V, adj);
    if (ans)
        cout << "True\n";
    else
        cout << "False\n";
    return 0;
}

```

Topological Sort Algorithm | DFS: G-21

Using DFS (Depth first Search)

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    void dfs(int node, vector<int> adj[],
vector<int>& vis, stack<int>& st) {
        vis[node] = 1;
        for (auto it : adj[node]) {
            if (!vis[it]) {
                dfs(it, adj, vis, st);
            }
        }
        st.push(node);
    }

```

```

}
vector<int> topoSort(int V, vector<int> adj[]) {
    vector<int> vis(V, 0);
    stack<int> st;
    for (int i = 0; i < V; i++) {
        if (!vis[i]) {
            dfs(i, adj, vis, st);
        }
    }
    vector<int> ans;
    while (!st.empty()) {
        ans.push_back(st.top());
        st.pop();
    }
    return ans;
}
};

int main() {
    int V = 6, E = 6;
    vector<int> adj[V];
    adj[5].push_back(0);
    adj[5].push_back(2);
    adj[4].push_back(0);
    adj[4].push_back(1);
    adj[2].push_back(3);
    adj[3].push_back(1);
    Solution obj;
    vector<int> res = obj.topoSort(V, adj);
    cout << "Topological Sort: ";
    for (auto it : res) {
        cout << it << " ";
    }
    cout << endl;
    return 0;
}

```

Time Complexity: $O(V + E)$, This is because in a topological sort using DFS, each vertex is visited exactly once, and each directed edge is explored exactly once during the DFS traversal. The cost of visiting all vertices is $O(V)$ and the cost of exploring all edges is $O(E)$. Since both happen sequentially and not nested, the total time complexity is $O(V + E)$. For example, if $V = 6$ and $E = 6$, the DFS will make exactly 6 vertex visits and 6 edge explorations, leading to a total of $O(6 + 6) = O(12)$, which simplifies to $O(V + E)$.

Space Complexity: $O(V + E)$, The space complexity comes from three parts: the adjacency list (which stores all vertices and edges, taking $O(V + E)$ space), the visited array ($O(V)$ space), and the recursion stack ($O(V)$ in the worst case for a DFS if the graph is like a chain). The stack used to store the topological order will also take $O(V)$ space. Therefore, the dominant space usage is $O(V + E)$. For example, if $V = 6$ and $E = 6$, the adjacency list will store all 6 vertices and 6 edges ($O(12)$ space), the visited array takes $O(6)$, and the recursion stack may take up to $O(6)$ in the worst case, keeping the total within $O(V + E)$.

Using BFS (Breadth First Search)

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> topologicalSort(int V, vector<int> adj[]) {
        vector<int> indegree(V, 0);
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                indegree[it]++;
            }
        }
        queue<int> q;
        for (int i = 0; i < V; i++) {
            if (indegree[i] == 0) {
                q.push(i);
            }
        }
        vector<int> topo;
        while (!q.empty()) {
            int node = q.front();
            q.pop();

```

```

        topo.push_back(node);
    for (auto it : adj[node]) {
        indegree[it]--;
        if (indegree[it] == 0) {
            q.push(it);
        }
    }
}

return topo;
}

};

int main() {
    int V = 6, E = 6;
    vector<int> adj[V];
    adj[5].push_back(0);
    adj[5].push_back(2);
    adj[4].push_back(0);
    adj[4].push_back(1);
    adj[2].push_back(3);
    adj[3].push_back(1);
    Solution obj;
    vector<int> ans = obj.topologicalSort(V, adj);
    for (auto it : ans) {
        cout << it << " ";
    }
    cout << endl;
    return 0;
}

```

Time Complexity: $O(V + E)$ because we visit each vertex exactly once and process all outgoing edges from each vertex exactly once. The in-degree calculation takes $O(E)$, and each vertex is enqueued and dequeued exactly once in $O(V)$. Thus, total time is linear in the sum of vertices and edges.

Space Complexity: $O(V + E)$ because we store the adjacency list which takes $O(E)$ space, the in-degree array which takes $O(V)$, the queue which can store up to $O(V)$ vertices at a time, and the topological order array which takes $O(V)$. Overall, the space requirement is proportional to the size of the graph.

Kahn's Algorithm | Topological Sort Algorithm | BFS: G-22

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> topoSort(int V, vector<int> adj[]) {
        vector<int> ans;
        vector<int> inDegree(V, 0);
        for(int i=0; i<V; i++) {
            for(auto it : adj[i]) inDegree[it]++;
            queue<int> q;
            for(int i=0; i<V; i++) {
                if(inDegree[i] == 0) q.push(i);
            }
            while(!q.empty()) {
                int node = q.front();
                ans.push_back(node);
                q.pop();
                for(auto it : adj[node]) {
                    inDegree[it]--;
                    if(inDegree[it] == 0) q.push(it);
                }
            }
            return ans;
        }
    };

    int main() {
        int V = 6;
        vector<int> adj[V] = {
            {},
            {},
            {3},
            {1},
            {0,1},

```

```

        {0,2}
    };

    Solution sol;

    vector<int> ans = sol.topoSort(V, adj);

    cout << "The topological sorting of the given
graph is: \n";

    for(int i=0; i < V; i++) {

        cout << ans[i] << " ";

    }

    return 0;
}

```

Time Complexity: $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph. We traverse each vertex and edge once.

Space Complexity: $O(V)$, where V is the number of vertices. We use an array to store the in-degrees and a queue for BFS.

Detect a cycle in a directed graph

```

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

    bool isCyclic(int V, vector<vector<int>>& adj) {

        vector<int> indegree(V, 0);

        for (int i = 0; i < V; i++) {

            for (auto &nbr : adj[i]) {

                indegree[nbr]++;

            }

        }

        queue<int> q;

        for (int i = 0; i < V; i++) {

            if (indegree[i] == 0)

                q.push(i);

        }

        int count = 0;

        while (!q.empty()) {

            int node = q.front();

```

```

                q.pop();

                count++;

                for (auto &nbr : adj[node]) {

                    indegree[nbr]--;

                    if (indegree[nbr] == 0)

                        q.push(nbr);

                }

            }

            return count != V;

        }

    };

    int main() {

        int V = 4;

        vector<vector<int>> adj = {

            {1}, {2}, {3}, {1}

        };

        Solution sol;

        cout << (sol.isCyclic(V, adj) ? "Graph contains a
cycle" : "No cycle") << endl;

        return 0;

    }
}

```

Time Complexity: $O(V+E)$, Each vertex and edge is processed exactly once while calculating in-degrees and during the BFS traversal.

Space Complexity: $O(V+E)$, We store the adjacency list, an in-degree array and a queue.

Course Schedule I and II | Pre-requisite Tasks | Topological Sort: G-24

Course Schedule I

```

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

    bool canFinish(int numCourses,

        vector<vector<int>>& prerequisites) {

        vector<vector<int>> adj(numCourses);

```

```

vector<int> inDegree(numCourses, 0);
for (auto& pre : prerequisites) {
    int a = pre[0], b = pre[1];
    adj[b].push_back(a);
    inDegree[a]++;
}
queue<int> q;
for (int i = 0; i < numCourses; i++) {
    if (inDegree[i] == 0) {
        q.push(i);
    }
}
int count = 0;
while (!q.empty()) {
    int node = q.front();
    q.pop();
    count++;
    for (int nei : adj[node]) {
        inDegree[nei]--;
        if (inDegree[nei] == 0) {
            q.push(nei);
        }
    }
}
return count == numCourses;
}
};

int main() {
    Solution sol;
    vector<vector<int>> prerequisites = {{1, 0}, {0, 1}};
    int numCourses = 2;
    cout << (sol.canFinish(numCourses, prerequisites)
        ? "true" : "false");
    return 0;
}

```

Time Complexity: $O(V+E)$, each course and prerequisite edge is processed once.
Space Complexity: $O(V+E)$, additional space is used for storing adjacency list and queue.

Course Schedule II

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> findOrder(int numCourses,
        vector<vector<int>>& prerequisites) {
        vector<vector<int>> adj(numCourses);
        vector<int> inDegree(numCourses, 0);
        for (auto& pre : prerequisites) {
            int a = pre[0], b = pre[1];
            adj[b].push_back(a);
            inDegree[a]++;
        }
        queue<int> q;
        for (int i = 0; i < numCourses; i++) {
            if (inDegree[i] == 0) {
                q.push(i);
            }
        }
        vector<int> order;
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            order.push_back(node);
            for (int nei : adj[node]) {
                inDegree[nei]--;
                if (inDegree[nei] == 0) {
                    q.push(nei);
                }
            }
        }
    }
}

```

```

        if ((int)order.size() == numCourses) {
            return order;
        }
        return {};
    }
};

int main() {
    Solution sol;

    vector<vector<int>> prerequisites = {{1, 0}, {2, 0}, {3, 1}, {3, 2}};

    int numCourses = 4;

    vector<int> ans = sol.findOrder(numCourses, prerequisites);

    for (int x : ans) cout << x << " ";

    return 0;
}

```

Time Complexity: $O(V+E)$, each course and prerequisite edge is processed once.

Space Complexity: $O(V+E)$, additional space is used for storing adjacency list, queue and ordering array.

Course Schedule I and II | Pre-requisite Tasks | Topological Sort: G-24

Course Schedule I

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool canFinish(int numCourses,
        vector<vector<int>>& prerequisites) {

        vector<vector<int>> adj(numCourses);

        vector<int> inDegree(numCourses, 0);

        for (auto& pre : prerequisites) {
            int a = pre[0], b = pre[1];

            adj[b].push_back(a);

            inDegree[a]++;
        }
    }
}

```

```

queue<int> q;

for (int i = 0; i < numCourses; i++) {
    if (inDegree[i] == 0) {
        q.push(i);
    }
}

int count = 0;

while (!q.empty()) {
    int node = q.front();
    q.pop();
    count++;

    for (int nei : adj[node]) {
        inDegree[nei]--;

        if (inDegree[nei] == 0) {
            q.push(nei);
        }
    }
}

return count == numCourses;
}

};

int main() {
    Solution sol;

    vector<vector<int>> prerequisites = {{1, 0}, {0, 1}};

    int numCourses = 2;

    cout << (sol.canFinish(numCourses, prerequisites)
        ? "true" : "false");

    return 0;
}

```

Time Complexity: $O(V+E)$, each course and prerequisite edge is processed once.

Space Complexity: $O(V+E)$, additional space is used for storing adjacency list and queue.

Course Schedule II

```

#include <bits/stdc++.h>

using namespace std;

```

```

class Solution {
public:
    vector<int> findOrder(int numCourses,
                        vector<vector<int>>&
prerequisites) {
        vector<vector<int>> adj(numCourses);
        vector<int> inDegree(numCourses, 0);
        for (auto& pre : prerequisites) {
            int a = pre[0], b = pre[1];
            adj[b].push_back(a);
            inDegree[a]++;
        }
        queue<int> q;
        for (int i = 0; i < numCourses; i++) {
            if (inDegree[i] == 0) {
                q.push(i);
            }
        }
        vector<int> order;
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            order.push_back(node);
            for (int nei : adj[node]) {
                inDegree[nei]--;
                if (inDegree[nei] == 0) {
                    q.push(nei);
                }
            }
        }
        if ((int)order.size() == numCourses) {
            return order;
        }
        return {};
    }
};

```

```

int main() {
    Solution sol;
    vector<vector<int>> prerequisites = {{1, 0}, {2,
0}, {3, 1}, {3, 2}};
    int numCourses = 4;
    vector<int> ans = sol.findOrder(numCourses,
prerequisites);
    for (int x : ans) cout << x << " ";
    return 0;
}

```

Time Complexity: $O(V+E)$, each course and pre-requisite edge is processed once.

Space Complexity: $O(V+E)$, additional space is used for storing adjacency list, queue and ordering array.

Find Eventual Safe States - BFS - Topological Sort: G-25

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    vector<int> eventualSafeNodes(int V,
vector<int> adj[]) {
        vector<int> adjRev[V];
        int indegree[V] = {0};
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                adjRev[it].push_back(i);
                indegree[i]++;
            }
        }
        queue<int> q;
        vector<int> safeNodes;
        for (int i = 0; i < V; i++) {
            if (indegree[i] == 0) {
                q.push(i);
            }
        }
    }
}

```

```

while (!q.empty()) {
    int node = q.front();

    q.pop();
    safeNodes.push_back(node);
    for (auto it : adjRev[node]) {
        indegree[it]--;
        if (indegree[it] == 0) {
            q.push(it);
        }
    }
}

sort(safeNodes.begin(), safeNodes.end());
return safeNodes;
}

};

int main() {
    vector<int> adj[12] = {{1}, {2}, {3, 4}, {4, 5},
        {6}, {6}, {7}, {}, {1, 9}, {10},
        {8}, {9}};

    int V = 12;

    Solution obj;

    vector<int> safeNodes =
obj.eventualSafeNodes(V, adj);

    for (auto node : safeNodes) {
        cout << node << " ";
    }

    cout << endl;

    return 0;
}

```

Time Complexity: $O(V + E) + O(N \log N)$,

- $O(V + E)$ for performing BFS on the graph (where V = number of vertices/nodes, E = number of edges).
- $O(N \log N)$ for sorting the safeNodes array (where N is the number of safe nodes).

Space Complexity: $O(3N) \sim O(N)$,

- $O(N)$ for the indegree array.
- $O(N)$ for the queue used during BFS traversal.

- $O(N)$ for the reversed adjacency list representation of the graph.

Alien Dictionary - Topological Sort: G-26

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    vector<int> topoSort(int V, vector<int> adj[]) {
        vector<int> indegree(V, 0);
        for (int i = 0; i < V; i++) {
            for (auto neighbor : adj[i]) {
                indegree[neighbor]++;
            }
        }

        queue<int> q;
        for (int i = 0; i < V; i++) {
            if (indegree[i] == 0) {
                q.push(i);
            }
        }

        vector<int> topo;
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            topo.push_back(node);
            for (auto neighbor : adj[node]) {
                indegree[neighbor]--;
                if (indegree[neighbor] == 0) {
                    q.push(neighbor);
                }
            }
        }

        return topo;
    }
}

```

```

public:

string findOrder(string dict[], int N, int K) {
    vector<int> adj[K];

    for (int i = 0; i < N - 1; i++) {
        string s1 = dict[i];
        string s2 = dict[i + 1];

        int len = min(s1.size(), s2.size());
        for (int ptr = 0; ptr < len; ptr++) {
            if (s1[ptr] != s2[ptr]) {
                adj[s1[ptr] - 'a'].push_back(s2[ptr] -
'a');
                break;
            }
        }
    }
    vector<int> topo = topoSort(K, adj);
    string ans = "";
    for (auto node : topo) {
        ans += char(node + 'a');
    }
    return ans;
}

};

int main() {
    int N = 5, K = 4;

    string dict[] = {"baa", "abcd", "abca", "cab",
"cad"};

    Solution obj;
    string ans = obj.findOrder(dict, N, K);

    for (auto ch : ans) {
        cout << ch << " ";
    }

    cout << endl;

    return 0;
}

```

Time Complexity: $O(N \cdot \text{len}) + O(K + E)$, where N is the number of words in the dictionary, 'len' is the length up

to the index where the first inequality occurs, K = no. of nodes, and E = no. of edges.

Space Complexity: $O(K) + O(K) + O(K) + O(K) \sim O(4K)$, $O(K)$ for the indegree array, and $O(K)$ for the queue data structure used in BFS (where K = no. of nodes), $O(K)$ for the answer array and $O(K)$ for the adjacency list used in the algorithm.) for the answer array and $O(K)$ for the adjacency list used in the algorithm.

Shortest Path in Undirected Graph with unit distance: G-28

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> shortestPath(vector<vector<int>>&
edges, int N, int M, int src) {
        vector<int> adj[N];

        for (auto it : edges) {
            adj[it[0]].push_back(it[1]);
            adj[it[1]].push_back(it[0]);
        }

        int dist[N];

        for (int i = 0; i < N; i++)
            dist[i] = 1e9;

        dist[src] = 0;

        queue<int> q;

        q.push(src);

        while (!q.empty()) {
            int node = q.front();
            q.pop();

            for (auto it : adj[node]) {
                if (dist[node] + 1 < dist[it]) {
                    dist[it] = 1 + dist[node];
                    q.push(it);
                }
            }
        }

        vector<int> ans(N, -1);

```

```

        for (int i = 0; i < N; i++) {
            if (dist[i] != 1e9) {
                ans[i] = dist[i];
            }
        }
        return ans;
    }
};

int main() {
    int N = 9, M = 10;
    vector<vector<int>> edges = {
        {0, 1}, {0, 3}, {3, 4}, {4, 5}, {5, 6},
        {1, 2}, {2, 6}, {6, 7}, {7, 8}, {6, 8}
    };
    Solution obj;
    vector<int> ans = obj.shortestPath(edges, N, M,
0);
    for (int i = 0; i < ans.size(); i++) {
        cout << ans[i] << " ";
    }
    return 0;
}

```

Time Complexity: $O(M)$ { for creating the adjacency list from given list 'edges' } + $O(N + 2M)$ { for the BFS Algorithm } + $O(N)$ { for adding the final values of the shortest path in the resultant array } $\sim O(N+2M)$. Where N = number of vertices and M = number of edges.

Space Complexity: $O(N)$ {for the stack storing the BFS} + $O(N)$ {for the resultant array} + $O(N)$ {for the dist array storing updated shortest paths} + $O(N+2M)$ {for the adjacency list} $\sim O(N+M)$, Where N = number of vertices and M = number of edges.

Shortest Path in Directed Acyclic Graph Topological Sort: G-27

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Solution {
```

```
private:
```

```

    void topoSort(int node, vector<pair<int, int>>
adj[], int vis[], stack<int> &st) {
        vis[node] = 1;
        for (auto it : adj[node]) {
            int v = it.first;
            if (!vis[v]) {
                topoSort(v, adj, vis, st);
            }
        }
        st.push(node);
    }
public:
    vector<int> shortestPath(int N, int M,
vector<vector<int>> &edges) {
        vector<pair<int, int>> adj[N];
        for (int i = 0; i < M; i++) {
            int u = edges[i][0];
            int v = edges[i][1];
            int wt = edges[i][2];
            adj[u].push_back({v, wt});
        }
        int vis[N] = {0};
        stack<int> st;
        for (int i = 0; i < N; i++) {
            if (!vis[i]) {
                topoSort(i, adj, vis, st);
            }
        }
        vector<int> dist(N);
        for (int i = 0; i < N; i++) {
            dist[i] = 1e9;
        }
        dist[0] = 0;
        while (!st.empty()) {
            int node = st.top();
            st.pop();
            for (auto it : adj[node]) {

```

```

    int v = it.first;

    int wt = it.second;

    if (dist[node] + wt < dist[v]) {

        dist[v] = wt + dist[node];

    }

}

}

for (int i = 0; i < N; i++) {

    if (dist[i] == 1e9) {

        dist[i] = -1;

    }

}

return dist;

}

};

int main() {

    int N = 6, M = 7;

    vector<vector<int>>> edges = {

        {0, 1, 2},

        {0, 4, 1},

        {4, 5, 4},

        {4, 2, 2},

        {1, 2, 3},

        {2, 3, 6},

        {5, 3, 1}

    };

    Solution obj;

    vector<int> ans = obj.shortestPath(N, M, edges);

    for (int i = 0; i < ans.size(); i++) {

        cout << ans[i] << " ";

    }

    return 0;

}

Time Complexity:  $O(N+M)$  {for the topological sort} +  $O(N+M)$  {for relaxation of vertices, each node and its adjacent nodes get traversed}  $\sim O(N+M)$ , where  $N$ = number of vertices and  $M$ = number of edges.

```

Space Complexity: $O(N)$ {for the stack storing the topological sort} + $O(N)$ {for storing the shortest distance for each node} + $O(N)$ {for the visited array} + $O(N+2M)$ {for the adjacency list} $\sim O(N+M)$.

Dijkstra's Algorithm - Using Set : G-33

```

#include <bits/stdc++.h>

using namespace std;

class Solution{

public:

    vector<int> dijkstra(int V,
vector<vector<int>>> adj[], int S) {

        set<pair<int, int>> st;

        vector<int> dist(V, 1e9);

        st.insert({0, S});

        dist[S] = 0;

        while(!st.empty()) {

            auto it = *(st.begin());

            int node = it.second;

            int dis = it.first;

            st.erase(it);

            for(auto it : adj[node]) {

                int adjNode = it[0];

                int edgW = it[1];

                if(dis + edgW < dist[adjNode]){

                    if(dist[adjNode] != 1e9)

                        st.erase({dist[adjNode],

adjNode});

                    dist[adjNode] = dis + edgW;

                    st.insert({dist[adjNode], adjNode});

                }

            }

        }

        return dist;

    }

};

int main(){

```

```

int V = 3, E = 3, S = 2;
vector<vector<int>> adj[V];

vector<int> v1 {1, 1}, v2 {2, 6}, v3 {2, 3}, v4 {0, 1}, v5 {1, 3}, v6 {0, 6};
adj[0].push_back(v1);
adj[0].push_back(v2);
adj[1].push_back(v3);
adj[1].push_back(v4);
adj[2].push_back(v5);
adj[2].push_back(v6);
Solution obj;
vector<int> res = obj.dijkstra(V, adj, S);

for (int i = 0; i < V; i++){
    cout << res[i] << " ";
}

cout << endl;

return 0;
}

```

Time Complexity: $O(E \log V)$, as each edge leads to at most one insertion in the priority queue, which takes $\log V$ time.

Space Complexity: $O(V + E)$, due to the distance array and adjacency list storing all vertices and edges.

Dijkstra's Algorithm - Using Priority Queue : G-32

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> dijkstra(int V,
vector<vector<pair<int,int>>>& adj, int src) {
        vector<int> dist(V, 1e9);

        priority_queue<pair<int,int>,
vector<pair<int,int>>,
            greater<pair<int,int>>> pq;

        dist[src] = 0;

        pq.push({0, src});

```

```

while (!pq.empty()) {
    int d = pq.top().first;
    int node = pq.top().second;
    pq.pop();
    if (d > dist[node]) continue;
    for (auto it : adj[node]) {
        int next = it.first;
        int wt = it.second;
        if (dist[node] + wt < dist[next]) {
            dist[next] = dist[node] + wt;
            pq.push({dist[next], next});
        }
    }
}

return dist;
}

};

int main() {
    int V = 5;

    vector<vector<pair<int,int>>> adj(V);
    adj[0].push_back({1, 2});
    adj[0].push_back({2, 4});
    adj[1].push_back({2, 1});
    adj[1].push_back({3, 7});
    adj[2].push_back({4, 3});
    adj[3].push_back({4, 2});

    Solution obj;

    vector<int> dist = obj.dijkstra(V, adj, 0);

    for (int i = 0; i < V; i++) {
        cout << "Distance from 0 to " << i << " = " <<
dist[i] << endl;
    }
}

```

Time Complexity: $O((V+E)\log V)$, each edge is relaxed at most once, and for each relaxation we may push into the priority queue, giving $O(E)$ pushes with $O(\log V)$ heap operations.

Space Complexity: $O(V+E)$, additional space is used to store adjacency list, distance array and priority queue.

G-36: Shortest Distance in a Binary Maze

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int shortestPath(vector<vector<int>> &grid,
pair<int, int> source, pair<int, int> destination) {
        if (source.first == destination.first &&
source.second == destination.second)
            return 0;

        queue<pair<int, pair<int, int>>> q;

        int n = grid.size();
        int m = grid[0].size();

        vector<vector<int>> dist(n, vector<int>(m,
1e9));

        dist[source.first][source.second] = 0;

        q.push({0, {source.first, source.second}});

        int dr[] = {-1, 0, 1, 0};
        int dc[] = {0, 1, 0, -1};

        while (!q.empty()) {
            auto it = q.front();

            q.pop();

            int dis = it.first;

            int r = it.second.first;

            int c = it.second.second;

            for (int i = 0; i < 4; i++) {
                int newr = r + dr[i];

                int newc = c + dc[i];

                if (newr >= 0 && newr < n && newc >=
0 && newc < m && grid[newr][newc] == 1 &&
dis + 1 < dist[newr][newc]) {
                    dist[newr][newc] = 1 + dis;

                    if (newr == destination.first && newc
== destination.second)
                        return dis + 1;

                    q.push({1 + dis, {newr, newc}});
                }
            }
        }
    }
};
```

```
    }
}

return -1;
}

};

int main() {
    pair<int, int> source = {0, 1};
    pair<int, int> destination = {2, 2};
    vector<vector<int>> grid = {{1, 1, 1, 1},
                                {1, 1, 0, 1},
                                {1, 1, 1, 1},
                                {1, 1, 0, 0},
                                {1, 0, 0, 1}};

    Solution obj;

    int res = obj.shortestPath(grid, source,
destination);

    cout << res << endl;

    return 0;
}
```

Time Complexity: $O(4 * N * M)$, where $N * M$ are the total cells, and for each cell, we check 4 adjacent nodes for the shortest path length. Where N = No. of rows of the binary maze and M = No. of columns of the binary maze.

Space Complexity: $O(N * M)$, where N = No. of rows of the binary maze and M = No. of columns of the binary maze.

G-37: Path With Minimum Effort

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int MinimumEffort(vector<vector<int>>
&heights) {
        priority_queue<pair<int, pair<int, int>>,
vector<pair<int, pair<int, int>>>,
greater<pair<int, pair<int, int>>>>
pq;
```

```

int n = heights.size();
int m = heights[0].size();
vector<vector<int>> dist(n, vector<int>(m,
1e9));
dist[0][0] = 0;
pq.push({0, {0, 0}});
int dr[] = {-1, 0, 1, 0};
int dc[] = {0, 1, 0, -1};
while (!pq.empty()) {
    auto it = pq.top();
    pq.pop();
    int diff = it.first;
    int row = it.second.first;
    int col = it.second.second;
    if (row == n - 1 && col == m - 1)
        return diff;
    for (int i = 0; i < 4; i++) {
        int newr = row + dr[i];
        int newc = col + dc[i];
        if (newr >= 0 && newc >= 0 && newr <
n && newc < m) {
            int newEffort =
max(abs(heights[row][col] - heights[newr][newc]),
diff);
            if (newEffort < dist[newr][newc]) {
                dist[newr][newc] = newEffort;
                pq.push({newEffort, {newr,
newc}});
            }
        }
    }
}
return 0;
}
};
int main() {
    vector<vector<int>> heights = {{1, 2, 2}, {3, 8,
2}, {5, 3, 5}};

```

```

Solution obj;
int ans = obj.MinimumEffort(heights);
cout << ans << endl;
return 0;
}

```

Time Complexity: $O(4 * N * M * \log(N * M))$, where $N * M$ are the total cells, for each of which we check 4 adjacent nodes for the minimum effort and an additional $\log(N * M)$ for insertion-deletion operations in a priority queue. Where N = No. of rows of the binary maze and M = No. of columns of the binary maze.

Space Complexity: $O(N * M)$, where $N * M$ is the distance matrix containing $N * M$ cells, plus the priority queue in the worst case containing all the nodes ($N * M$). Where N = No. of rows of the binary maze and M = No. of columns of the binary maze.

G-38: Cheapest Flights Within K Stops

```

#include <bits/stdc++.h>
using namespace std;
class Solution{
public:
    int CheapestFLight(int n, vector<vector<int>>
&flights, int src, int dst, int K){
        vector<pair<int, int>> adj[n];
        for (auto it : flights)
        {
            adj[it[0]].push_back({it[1], it[2]});
        }
        queue<pair<int, pair<int, int>>> q;
        q.push({0, {src, 0}});
        vector<int> dist(n, 1e9);
        dist[src] = 0;
        while (!q.empty())
        {
            auto it = q.front();
            q.pop();
            int stops = it.first;
            int node = it.second.first;
            int cost = it.second.second;

```

```

        if (stops > K)
            continue;
        for (auto iter : adj[node])
        {
            int adjNode = iter.first;
            int edW = iter.second;
            if (cost + edW < dist[adjNode] && stops
<= K)
            {
                dist[adjNode] = cost + edW;
                q.push({stops + 1, {adjNode, cost +
edW}});
            }
        }
        if (dist[dst] == 1e9)
            return -1;
        return dist[dst];
    }
};

int main(){
    int n = 4, src = 0, dst = 3, K = 1;

    vector<vector<int>> flights = {{0, 1, 100}, {1,
2, 100}, {2, 0, 100}, {1, 3, 600},
{2, 3, 200}};

    Solution obj;

    int ans = obj.CheapestFLight(n, flights, src, dst,
K);

    cout << ans << endl;

    return 0;
}

```

Time Complexity: $O(N)$, where the additional $\log(N)$ time is eliminated by using a simple queue rather than a priority queue, which is usually used in Dijkstra's Algorithm. Where N = Number of flights / Number of edges.

Space Complexity: $O(|E| + |V|)$, for the adjacency list, priority queue, and the dist array. Where E = Number of edges (`flights.size()`) and V = Number of airports.

Network Delay Time

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        vector<vector<pair<int, int>>> adj(n + 1);

        for (auto& time : times) {
            int u = time[0], v = time[1], w = time[2];
            adj[u].push_back({v, w});
        }

        priority_queue<pair<int, int>, vector<pair<int,
int>>, greater<>> pq;

        pq.push({0, k});

        vector<int> dist(n + 1, INT_MAX);

        dist[k] = 0;

        while (!pq.empty()) {
            int time = pq.top().first;
            int node = pq.top().second;
            pq.pop();

            for (auto& [nbr, wt] : adj[node]) {
                if (dist[nbr] > time + wt) {
                    dist[nbr] = time + wt;
                    pq.push({dist[nbr], nbr});
                }
            }
        }

        int ans = *max_element(dist.begin() + 1,
dist.end());

        return ans == INT_MAX ? -1 : ans;
    }
};

int main() {
    Solution sol;

    vector<vector<int>> times =
{{2,1,1},{2,3,1},{3,4,1}};

```

```
int n = 4, k = 2;

cout << sol.networkDelayTime(times, n, k) <<
endl;
}

Time Complexity:  $O((E + V) * \log V)$ , Each edge is
relaxed atmost once. Priority queue operations take  $\log V$ 
time.
Space Complexity:  $O(V+E)$ , We store the adjacency list,
an in-degree array and a priority queue in order to find
the minimum time required for every node in the network
to receive the signal.
```

G-40: Number of Ways to Arrive at Destination

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:

    int CheapestFLight(int n, vector<vector<int>>>
    &flights, int src, int dst, int K) {

        vector<pair<int, int>> adj[n];

        for (auto it : flights) {

            adj[it[0]].push_back({it[1], it[2]});

            adj[it[1]].push_back({it[0], it[2]});

        }

        priority_queue<pair<int, int>,

            vector<pair<int, int>>,

            greater<pair<int, int>>> pq;

        vector<int> dist(n, INT_MAX), ways(n, 0);

        dist[src] = 0;

        ways[src] = 1;

        pq.push({0, src});

        int mod = (int)(1e9 + 7);

        while (!pq.empty()) {

            int dis = pq.top().first;

            int node = pq.top().second;

            pq.pop();

            for (auto it : adj[node]) {

                int adjNode = it.first;

                int edW = it.second;
```

```
                if (dis + edW < dist[adjNode]) {

                    dist[adjNode] = dis + edW;

                    pq.push({dis + edW, adjNode});

                    ways[adjNode] = ways[node];

                }

                else if (dis + edW == dist[adjNode]) {

                    ways[adjNode] = (ways[adjNode] +

                    ways[node]) % mod;

                }

            }

        }

        return ways[dst] % mod;

    }

};

int main() {

    int n = 7;

    vector<vector<int>>> edges = {{0, 6, 7}, {0, 1,

    2}, {1, 2, 3}, {1, 3, 3}, {6, 3, 3},

    {3, 5, 1}, {6, 5, 1}, {2, 5, 1},

    {0, 4, 5}, {4, 6, 2}};

    Solution obj;

    int ans = obj.CheapestFLight(n, edges, 0, 3, 1);

    cout << ans << endl;

    return 0;

}
```

Time Complexity: $O(E * \log(V))$, as we are using simple Dijkstra's algorithm here, the time complexity will be of the order $E * \log(V)$. Where E = Number of edges and V = Number of vertices.
Space Complexity: $O(N)$, for the dist array, ways array, and approximate complexity for the priority queue. Where N = Number of nodes.

G-39: Minimum Multiplications to Reach End

```
#include <bits/stdc++.h>

using namespace std;

class Solution{
public:

    int minimumMultiplications(vector<int> &arr,
```

```

        int start, int end)
    {
        queue<pair<int, int>> q;
        q.push({start, 0});
        vector<int> dist(100000, 1e9);
        dist[start] = 0;
        int mod = 100000;
        while (!q.empty())
        {
            int node = q.front().first;
            int steps = q.front().second;
            q.pop();
            for (auto it : arr)
            {
                int num = (it * node) % mod;
                if (steps + 1 < dist[num])
                {
                    dist[num] = steps + 1;
                    if (num == end)
                        return steps + 1;
                    q.push({num, steps + 1});
                }
            }
        }
        return -1;
    }
};

int main(){
    int start = 3, end = 30;
    vector<int> arr = {2, 5, 7};
    Solution obj;
    int ans = obj.minimumMultiplications(arr, start,
end);
    cout << ans;
    cout << endl;
    return 0;
}

```

```

}

```

Time Complexity : $O(100000 * N)$, Where '100000' are the total possible numbers generated by multiplication (hypothetical) and N = size of the array with numbers of which each node could be multiplied.

Space Complexity : $O(100000 * N)$, Where '100000' are the total possible numbers generated by multiplication (hypothetical) and N = size of the array with numbers of which each node could be multiplied. $100000 * N$ is the max possible queue size. The space complexity of the dist array is constant.

Bellman Ford Algorithm: G-41

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> bellman_ford(int V,
vector<vector<int>>& edges, int S) {
        vector<int> dist(V, 1e8);
        dist[S] = 0;
        for (int i = 0; i < V - 1; i++) {
            for (auto it : edges) {
                int u = it[0];
                int v = it[1];
                int wt = it[2];
                if (dist[u] != 1e8 && dist[u] + wt <
dist[v]) {
                    dist[v] = dist[u] + wt;
                }
            }
        }
        for (auto it : edges) {
            int u = it[0];
            int v = it[1];
            int wt = it[2];
            if (dist[u] != 1e8 && dist[u] + wt <
dist[v]) {
                return { -1 };
            }
        }
        return dist;
    }
}

```

```

    }
};

int main() {
    int V = 6;
    vector<vector<int>> edges(7, vector<int>(3));
    edges[0] = {3, 2, 6};
    edges[1] = {5, 3, 1};
    edges[2] = {0, 1, 5};
    edges[3] = {1, 5, -3};
    edges[4] = {1, 2, -2};
    edges[5] = {3, 4, -2};
    edges[6] = {2, 4, 3};
    int S = 0;
    Solution obj;
    vector<int> dist = obj.bellman_ford(V, edges,
S);
    for (auto d : dist) {
        cout << d << " ";
    }
    cout << endl;
    return 0;
}

```

Time Complexity: $O(V \cdot E)$, where V = no. of vertices and E = no. of Edges.

Space Complexity: $O(V)$ for the distance array which stores the minimized distances.

Floyd Warshall Algorithm: G-42

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    void shortest_distance(vector<vector<int>>
&matrix){
        int n = matrix.size();
        for(int k=0; k<n; k++) {
            for(int i=0; i<n; i++) {

```

```

                for(int j=0; j<n; j++) {
                    if(matrix[i][k] == -1 || matrix[k][j] == -
1)
                        continue;
                    if(matrix[i][j] == -1) {
                        matrix[i][j] = matrix[i][k] +
matrix[k][j];
                    }else {
                        matrix[i][j] = min(matrix[i][j] ,
matrix[i][k] + matrix[k][j]);
                    }
                }
            }
        }
    };
    int main() {
        vector<vector<int>> matrix ={
            {0, 2, -1, -1},
            {1, 0, 3, -1},
            {-1, -1, 0, -1},
            {3, 5, 4, 0}
        };
        Solution sol;
        sol.shortest_distance(matrix);
        int n = matrix.size();
        cout << "The shortest distance matrix is:\n";
        for(int i=0; i < n; i++) {
            for(int j=0; j < n; j++) {
                cout << matrix[i][j] << " ";
            }
            cout << endl;
        }
        return 0;
    }
}

```

Time Complexity: $O(V^3)$, where V is the number of vertices in the graph. This is because we have three nested loops iterating over the vertices.

Space Complexity: $O(V^2)$, where V is the number of vertices in the graph. This is due to the storage of the distance matrix, which is of size $V \times V$.

Find the City With the Smallest Number of Neighbours at a Threshold Distance: G-43

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int findCity(int n, int m, vector<vector<int>>& edges, int distanceThreshold) {
        vector<vector<int>> dist(n, vector<int>(n, INT_MAX));

        for (auto it : edges) {
            dist[it[0]][it[1]] = it[2];
            dist[it[1]][it[0]] = it[2];
        }

        for (int i = 0; i < n; i++) dist[i][i] = 0;

        for (int k = 0; k < n; k++) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    if (dist[i][k] == INT_MAX || dist[k][j] == INT_MAX)
                        continue;
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }

        int cntCity = n;
        int cityNo = -1;

        for (int city = 0; city < n; city++) {
            int cnt = 0;

            for (int adjCity = 0; adjCity < n; adjCity++)
            {
                if (dist[city][adjCity] <= distanceThreshold)
```

```
                cnt++;
            }

            if (cnt <= cntCity) {
                cntCity = cnt;
                cityNo = city;
            }
        }

        return cityNo;
    }
};

int main() {
    int n = 4;
    int m = 4;

    vector<vector<int>> edges = {{0, 1, 3}, {1, 2, 1}, {1, 3, 4}, {2, 3, 1}};

    int distanceThreshold = 4;

    Solution obj;

    int cityNo = obj.findCity(n, m, edges, distanceThreshold);

    cout << "The answer is node: " << cityNo << endl;

    return 0;
}
```

Time Complexity: $O(V^3)$, as we have three nested loops each running for V times, where V = number of vertices.

Space Complexity: $O(V^2)$, where V = number of vertices. This space complexity is due to storing the adjacency matrix of the given graph.

Prim's Algorithm - Minimum Spanning Tree

```
#include <bits/stdc++.h>

using namespace std;

class Solution{
public:
    int spanningTree(int V, vector<vector<int>> adj[]){
        priority_queue<pair<int, int>,
```

```

        vector<pair<int, int> >, greater<pair<int,
int>>>> pq;

        vector<int> vis(V, 0);
        pq.push({0, 0});
        int sum = 0;
        while (!pq.empty()) {
            auto it = pq.top();
            pq.pop();
            int node = it.second;
            int wt = it.first;
            if (vis[node] == 1) continue;
            vis[node] = 1;
            sum += wt;
            for (auto it : adj[node]) {
                int adjNode = it[0];
                int edW = it[1];
                if (!vis[adjNode]){
                    pq.push({edW, adjNode});
                }
            }
        }
        return sum;
    }
};

int main(){
    int V = 5;

    vector<vector<int>> edges = {{0, 1, 2}, {0, 2,
1}, {1, 2, 1}, {2, 3, 2}, {3, 4, 1}, {4, 2, 2}};

    vector<vector<int>> adj[V];
    for (auto it : edges){
        vector<int> tmp(2);
        tmp[0] = it[1];
        tmp[1] = it[2];
        adj[it[0]].push_back(tmp);
        tmp[0] = it[0];
        tmp[1] = it[2];
        adj[it[1]].push_back(tmp);
    }
}

```

```

    }
    Solution obj;
    int sum = obj.spanningTree(V, adj);
    cout << "The sum of all the edge weights: " <<
sum << endl;

    return 0;
}

```

Disjoint Set | Union by Rank | Union by Size | Path Compression: G-46

```

#include <bits/stdc++.h>

using namespace std;

class DisjointSet {
    vector<int> rank, parent;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {

```

```

        parent[ulp_v] = ulp_u;
    }
    else {
        parent[ulp_v] = ulp_u;
        rank[ulp_u]++;
    }
}
};

int main() {
    DisjointSet ds(7);
    ds.unionByRank(1, 2);
    ds.unionByRank(2, 3);
    ds.unionByRank(4, 5);
    ds.unionByRank(6, 7);
    ds.unionByRank(5, 6);
    // if 3 and 7 same or not
    if (ds.findUPar(3) == ds.findUPar(7)) {
        cout << "Same\n";
    }
    else cout << "Not same\n";
    ds.unionByRank(3, 7);
    if (ds.findUPar(3) == ds.findUPar(7)) {
        cout << "Same\n";
    }
    else cout << "Not same\n";
    return 0; }

```

Kruskal's Algorithm - Minimum Spanning Tree : G-47

```

#include <bits/stdc++.h>
using namespace std;
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);

```

```

        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }
    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }
    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }
    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;

```

```

        size[ulp_u] += size[ulp_v];
    }
}
};

class Solution{
public:
    int spanningTree(int V, vector<vector<int>>>
adj[]) {
        vector<pair<int, pair<int, int>>> edges;
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                int v = it[0]; // Node v
                int wt = it[1]; // edge weight
                int u = i; // Node u
                edges.push_back({wt, {u, v}});
            }
        }
        DisjointSet ds(V);
        sort(edges.begin(), edges.end());
        int sum = 0;
        for (auto it : edges) {
            int wt = it.first;
            int u = it.second.first;
            int v = it.second.second;
            if (ds.findUPar(u) != ds.findUPar(v)) {
                sum += wt;
                ds.unionBySize(u, v);
            }
        }
        return sum;
    }
};

int main() {
    int V = 4;
    vector<vector<int>>> edges = {
        {0, 1, 1},

```

```

        {1, 2, 2},
        {2, 3, 3},
        {0, 3, 4}
    };
    vector<vector<int>>> adj[4];
    for(auto it : edges) {
        int u = it[0];
        int v = it[1];
        int wt = it[2];
        adj[u].push_back({v, wt});
        adj[v].push_back({u, wt});
    }
    Solution sol;
    int ans = sol.spanningTree(V, adj);

    cout << "The sum of weights of edges in MST is:
" << ans;

    return 0;
}

```

Time Complexity: $O(N + E) + O(E \log E) + O(E * 4 * 2)$, where N = number of nodes and E = number of edges. $O(N + E)$ for extracting edge information from the adjacency list. $O(E \log E)$ for sorting the array consisting of the edge tuples. Finally, we are using disjoint set operations inside a loop. The loop will continue for E times. Inside that loop, there are two disjoint set operations like findUPar() and UnionBySize(), each taking 4, so it will result in $4 * 2$. That is why the last term $O(E * 4 * 2)$ is added.

Space Complexity: $O(N) + O(N) + O(E)$, where E = number of edges and N = number of nodes. $O(E)$ space is taken by the array that we are using to store the edge information. In the disjoint set data structure, we are using two N -sized arrays (i.e., a parent and a size array, as we are using unionBySize() function; otherwise, a rank array of the same size if unionByRank() is used), which result in the first two terms $O(N)$.

Number of Operations to Make Network Connected - DSU: G-49.

```

#include <bits/stdc++.h>

using namespace std;

class DSU {
public:
    vector<int> parent, rank;

```

```

DSU(int n) {
    parent.resize(n);
    rank.resize(n, 0);
    for(int i = 0; i < n; i++) {
        parent[i] = i;
    }
}

int find(int x) {
    if(parent[x] != x)
        parent[x] = find(parent[x]);
    return parent[x];
}

void unite(int x, int y) {
    int px = find(x);
    int py = find(y);

    if(px == py) return;
    if(rank[px] < rank[py])
        parent[px] = py;
    else if(rank[px] > rank[py])
        parent[py] = px;
    else {
        parent[py] = px;
        rank[px]++;
    }
}

};

class Solution {
public:
    int makeConnected(int n, vector<vector<int>>&
connections) {
        if(connections.size() < n - 1) return -1;

        DSU dsu(n);
        for(auto& edge : connections) {
            dsu.unite(edge[0], edge[1]);
        }
    }
};

```

```

unordered_set<int> components;

for(int i = 0; i < n; i++) {
    components.insert(dsu.find(i));
}

return components.size() - 1;
}

};

int main() {
    int n = 6;

    vector<vector<int>> connections =
    {{0,1},{0,2},{0,3},{1,4}};

    Solution sol;

    cout << sol.makeConnected(n, connections) <<
endl;

    return 0;
}

```

Time Complexity: $O(N + M \times \alpha(N))$, we visit every node and edge once to find number of components.
Space Complexity: $O(N)$, for storing parent and rank/size vector for DSU.

Most Stones Removed with Same Row or Column - DSU: G-53

```

#include <bits/stdc++.h>

using namespace std;

class DSU {
public:
    unordered_map<int, int> parent;

    int find(int x) {
        if (parent.find(x) == parent.end())
            parent[x] = x;
        if (x != parent[x])
            parent[x] = find(parent[x]);
        return parent[x];
    }

    void unite(int x, int y) {
        parent[find(x)] = find(y);
    }
}

```

```

};

class Solution {
public:
    int removeStones(vector<vector<int>>& stones)
    {
        DSU dsu;
        for (auto& stone : stones) {
            dsu.unite(stone[0], stone[1] + 10001);
        }
        unordered_set<int> components;
        for (auto& stone : stones) {
            components.insert(dsu.find(stone[0]));
        }
        return stones.size() - components.size();
    }
};

int main() {
    vector<vector<int>> stones = {
        {0, 0}, {0, 1}, {1, 0}, {1, 2}, {2, 1}, {2, 2}
    };
    Solution obj;
    cout << obj.removeStones(stones) << endl;
    return 0;
}

```

Time Complexity: $O(N * \alpha(N))$, where N is number of stones. We visit every stone once to unite it with its row and column.

Space Complexity: $O(N)$, for the parent map inside the Disjoint Set data structure.

Accounts Merge - DSU: G-50

```

#include <bits/stdc++.h>

using namespace std;

class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);

```

```

        parent.resize(n + 1);
        size.resize(n + 1);
        // Initialize every node
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        parent[node] = findUPar(parent[node]);
        return parent[node];
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];

```

```

    }
    else {
        parent[ulp_v] = ulp_u;
        size[ulp_u] += size[ulp_v];
    }
}
};

class Solution {
public:
    vector<vector<string>> accountsMerge(
        vector<vector<string>> &details)
    {
        int n = details.size();
        DisjointSet ds(n);
        unordered_map<string, int> mapMailNode;
        for (int i = 0; i < n; i++) {
            for (int j = 1; j < details[i].size(); j++) {
                string mail = details[i][j];
                if (mapMailNode.find(mail) ==
mapMailNode.end()) {
                    mapMailNode[mail] = i;
                }
                else {
                    ds.unionBySize(i,
mapMailNode[mail]);
                }
            }
        }
        vector<string> mergedMail[n];
        for (auto it : mapMailNode) {
            string mail = it.first;
            int node = ds.findUPar(it.second);
            mergedMail[node].push_back(mail);
        }
        vector<vector<string>> ans;
        for (int i = 0; i < n; i++) {
            if (mergedMail[i].empty()) continue;

```

```

            sort(mergedMail[i].begin(),
mergedMail[i].end());

            vector<string> temp;
            temp.push_back(details[i][0]);
            for (auto &mail : mergedMail[i]) {
                temp.push_back(mail);
            }
            ans.push_back(temp);
        }
        sort(ans.begin(), ans.end());
        return ans;
    }
};

int main() {
    vector<vector<string>> accounts = {
        {"John", "j1@com", "j2@com", "j3@com"},
        {"John", "j4@com"},
        {"Raj", "r1@com", "r2@com"},
        {"John", "j1@com", "j5@com"},
        {"Raj", "r2@com", "r3@com"},
        {"Mary", "m1@com"}
    };
    Solution obj;
    vector<vector<string>> ans =
obj.accountsMerge(accounts);
    for (auto &acc : ans) {
        cout << acc[0] << ":-";
        for (int i = 1; i < acc.size(); i++) {
            cout << acc[i] << " ";
        }
        cout << endl;
    }
    return 0;
}

```

Time Complexity: $O(N+E) + O(E*4\alpha) + O(N*(E\log E + E))$ where N = no. of indices or nodes and E = no. of emails. The first term is for visiting all the emails. The second term is for merging the accounts. And the third

term is for sorting the emails and storing them in the answer array.

Space Complexity: $O(N) + O(N) + O(2N) \sim O(N)$ where N = no. of nodes/indices. The first and second space is for the 'mergedMail' and the 'ans' array. The last term is for the parent and size array used inside the Disjoint set data structure.

Number of Islands - II - Online Queries - DSU: G-51

```
#include <bits/stdc++.h>

using namespace std;

class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1, 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }
    // Path compression

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        } else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        } else {
            parent[ulp_v] = ulp_u;
```

```
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        } else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};

class Solution {
private:
    bool isValid(int adjr, int adjc, int n, int m) {
        return adjr >= 0 && adjr < n && adjc >= 0
        && adjc < m;
    }
public:
    vector<int> numOfIslands(int n, int m,
    vector<vector<int>>& operators) {
        DisjointSet ds(n * m); // Disjoint set to
        manage connected land cells

        int vis[n][m];
        memset(vis, 0, sizeof vis);
        int cnt = 0;
        vector<int> ans;
        for (auto it : operators) {
            int row = it[0], col = it[1];
            if (vis[row][col] == 1) {
                ans.push_back(cnt);
                continue;
            }
        }
```

```

        vis[row][col] = 1;

        cnt++;

        int dr[] = {-1, 0, 1, 0};
        int dc[] = {0, 1, 0, -1};

        for (int ind = 0; ind < 4; ind++) {
            int adjr = row + dr[ind], adjc = col +
dc[ind];

            if (isValid(adjr, adjc, n, m)) {
                if (vis[adjr][adjc] == 1) {
                    int nodeNo = row * m + col;
                    int adjNodeNo = adjr * m + adjc;

                    if (ds.findUPar(nodeNo) !=
ds.findUPar(adjNodeNo)) {
                        cnt--;

                        ds.unionBySize(nodeNo,
adjNodeNo);
                    }
                }
            }
        }

        ans.push_back(cnt);
    }

    return ans;
}

};

int main() {
    int n = 4, m = 5;

    vector<vector<int>> operators = {{0, 0}, {0, 0},
{1, 1}, {1, 0}, {0, 1},
    {0, 3}, {1, 3}, {0, 4}, {3, 2}, {2, 2}, {1, 2},
{0, 2}
    };

    Solution obj;

    vector<int> ans = obj.numOfIslands(n, m,
operators);

    for (auto res : ans) {
        cout << res << " ";
    }
}

```

```

        cout << endl;

        return 0;
    }
}

```

Time Complexity: $O(N \times M)$, Each cell is visited at most once in BFS/DFS traversal.

Space Complexity: $O(N \times M)$, Visited array and BFS queue may store all cells in the worst case.

Making a Large Island - DSU: G-52

```

#include <bits/stdc++.h>

using namespace std;

class DisjointSet {
public:
    vector<int> rank, parent, size;

    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);

        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;

        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);

        if (ulp_u == ulp_v) return;

        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
    }
}

```

```

else if (rank[ulp_v] < rank[ulp_u]) {
    parent[ulp_v] = ulp_u;
}
else {
    parent[ulp_v] = ulp_u;
    rank[ulp_u]++;
}
}

void unionBySize(int u, int v) {
    int ulp_u = findUPar(u);
    int ulp_v = findUPar(v);
    if (ulp_u == ulp_v) return;
    if (size[ulp_u] < size[ulp_v]) {
        parent[ulp_u] = ulp_v;
        size[ulp_v] += size[ulp_u];
    }
    else {
        parent[ulp_v] = ulp_u;
        size[ulp_u] += size[ulp_v];
    }
}

};

class Solution{
private:
    vector<int> delRow = {-1, 0, 1, 0};
    vector<int> delCol = {0, 1, 0, -1};
    bool isValid(int &i, int &j, int &n) {
        if(i < 0 || i >= n) return false;
        if(j < 0 || j >= n) return false;
        return true;
    }

    void addInitialIslands(vector<vector<int>>> grid,
        DisjointSet &ds, int n) {
        for (int row = 0; row < n ; row++) {
            for (int col = 0; col < n ; col++) {
                if (grid[row][col] == 0) continue;

                for (int ind = 0; ind < 4; ind++) {
                    int newRow = row + delRow[ind];
                    int newCol = col + delCol[ind];
                    if (isValid(newRow, newCol, n) &&
                        grid[newRow][newCol] == 1) {
                        int nodeNo = row * n + col;
                        int adjNodeNo = newRow * n +
                            newCol;
                        ds.unionBySize(nodeNo,
                            adjNodeNo);
                    }
                }
            }
        }
    }

public:
    int largestIsland(vector<vector<int>>> &grid) {
        int n = grid.size();
        DisjointSet ds(n*n);
        addInitialIslands(grid, ds, n);
        int ans = 0;
        for (int row = 0; row < n; row++) {
            for (int col = 0; col < n; col++) {
                if (grid[row][col] == 1) continue;
                set<int> components;
                for (int ind = 0; ind < 4; ind++) {
                    int newRow = row + delRow[ind];
                    int newCol = col + delCol[ind];
                    if (isValid(newRow, newCol, n) &&
                        grid[newRow][newCol] == 1) {
                        int nodeNumber = newRow * n +
                            newCol;
                        components.insert(ds.findUPar(nodeNumber));
                    }
                }
                int sizeTotal = 0;

```

```

        for (auto it : components)
            sizeTotal += ds.size[it];
    }
    ans = max(ans, sizeTotal + 1);
}
}

for (int cellNo = 0; cellNo < n * n; cellNo++)
{
    ans = max(ans,
ds.size[ds.findUPar(cellNo)]);
}
return ans;
}
};

int main() {
    vector<vector<int>> grid = {
        {1,0},
        {0,1}
    };

    Solution sol;

    int ans = sol.largestIsland(grid);

    cout << "The size of the largest island is: " <<
ans;

    return 0;
}

```

Time Complexity: $O(N^2)$ using nested loops, and within the loops, all the operations take constant time.

Space Complexity: $O(N^2)$, as the Disjoint set storing N^2 nodes (cells) will take up $2 * N^2$ space due to parent and size arrays.

Swim in Rising Water

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int swimInWater(vector<vector<int>>& grid) {
        int n = grid.size();

```

```

        priority_queue<vector<int>,
vector<vector<int>>, greater<vector<int>>>
minHeap;

        vector<vector<int>> visited(n, vector<int>(n,
0));

        minHeap.push({grid[0][0], 0, 0});

        visited[0][0] = 1;

        vector<pair<int, int>> dirs = {{0,1}, {1,0},
{0,-1}, {-1,0}};

        while (!minHeap.empty()) {
            auto curr = minHeap.top(); minHeap.pop();

            int elevation = curr[0], r = curr[1], c =
curr[2];

            if (r == n - 1 && c == n - 1) return
elevation;

            for (auto& dir : dirs) {
                int nr = r + dir.first;

                int nc = c + dir.second;

                if (nr >= 0 && nc >= 0 && nr < n && nc
< n && !visited[nr][nc]) {
                    visited[nr][nc] = 1;

                    minHeap.push({max(elevation,
grid[nr][nc]), nr, nc});
                }
            }
        }

        return -1;
    }
};

```

```

int main() {
    vector<vector<int>> grid = {
        {0,2}, {1,3}
    };

    Solution sol;

    cout << "Minimum time to reach destination: "
<< sol.swimInWater(grid) << endl;

    return 0;
}

```

Time Complexity: $O(N^2 * \log(N))$, Each cell is inserted into minHeap at most once. Priority queue

operations take $\log V$ time.

Space Complexity: $O(N^2)$, We use additional space to store the visited matrix and minHeap.

Bridges in Graph - Using Tarjan's Algorithm of time in and low time: G-55

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Solution {
```

```
private:
```

```
    int timer = 1;
```

```
    void dfs(int node, int parent, vector<int> &vis,
```

```
            vector<int> adj[], int tin[], int low[],
```

```
            vector<vector<int>> &bridges) {
```

```
        vis[node] = 1;
```

```
        tin[node] = low[node] = timer;
```

```
        timer++;
```

```
        for (auto it : adj[node]) {
```

```
            if (it == parent) continue;
```

```
            if (vis[it] == 0) {
```

```
                dfs(it, node, vis, adj, tin, low, bridges)
```

```
                low[node] = min(low[node], low[it]);
```

```
                if (low[it] > tin[node]) {
```

```
                    bridges.push_back({it, node});
```

```
                }
```

```
            } else {
```

```
                low[node] = min(low[node], low[it]);
```

```
            }
```

```
        }
```

```
    }
```

```
public:
```

```
    vector<vector<int>> criticalConnections(int n,
```

```
    vector<vector<int>> &connections) {
```

```
        vector<int> adj[n];
```

```
        for (auto it : connections) {
```

```
            int u = it[0], v = it[1];
```

```
            adj[u].push_back(v);
```

```
            adj[v].push_back(u);
```

```
        }
```

```
        vector<int> vis(n, 0);
```

```
        int tin[n];
```

```
        int low[n];
```

```
        vector<vector<int>> bridges;
```

```
        dfs(0, -1, vis, adj, tin, low, bridges);
```

```
        return bridges;
```

```
    }
```

```
};
```

```
int main() {
```

```
    int n = 4;
```

```
    vector<vector<int>> connections = {
```

```
        {0, 1}, {1, 2}, {2, 0}, {1, 3}
```

```
    };
```

```
    Solution obj;
```

```
    vector<vector<int>> bridges =
```

```
    obj.criticalConnections(n, connections);
```

```
    cout << "Critical Connections (Bridges): ";
```

```
    for (auto it : bridges) {
```

```
        cout << "[" << it[0] << ", " << it[1] << "] ";
```

```
    }
```

```
    cout << endl;
```

```
    return 0;
```

```
}
```

Time Complexity: $O(V+2E)$, where V = no. of vertices, E = no. of edges. It is because the algorithm is just a simple DFS traversal.

Space Complexity: $O(V+2E) + O(3V)$, where V = no. of vertices, E = no. of edges. $O(V+2E)$ to store the graph in an adjacency list and $O(3V)$ for the three arrays i.e. tin, low, and vis, each of size V .

Articulation Point in Graph: G-56

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```

class Solution {
private:
    int timer = 1;

    void dfs(int node, int parent, vector<int> &vis,
int tin[], int low[],
        vector<int> &mark, vector<int> adj[]) {
        vis[node] = 1;
        tin[node] = low[node] = timer++; // set
discovery and low time
        int child = 0;
        for (auto it : adj[node]) {
            if (it == parent) continue;
            if (!vis[it]) {
                dfs(it, node, vis, tin, low, mark, adj);
                low[node] = min(low[node], low[it]);
                if (low[it] >= tin[node] && parent != -1)
{
                    mark[node] = 1;
                }
                child++;
            }
            else {
                low[node] = min(low[node], tin[it]);
            }
        }
        if (parent == -1 && child > 1) {
            mark[node] = 1;
        }
    }
public:
    vector<int> articulationPoints(int n, vector<int>
adj[]) {
        vector<int> vis(n, 0), mark(n, 0);
        int tin[n], low[n];
        for (int i = 0; i < n; i++) {
            if (!vis[i]) {
                dfs(i, -1, vis, tin, low, mark, adj);

```

```

            }
        }
        vector<int> ans;
        for (int i = 0; i < n; i++) {
            if (mark[i]) ans.push_back(i);
        }
        return ans.empty() ? vector<int>{-1} : ans;
    }
};

int main() {
    int n = 5;
    vector<vector<int>> edges = {
        {0, 1}, {1, 4}, {2, 4}, {2, 3}, {3, 4}
    };
    vector<int> adj[n];
    for (auto e : edges) {
        adj[e[0]].push_back(e[1]);
        adj[e[1]].push_back(e[0]);
    }
    Solution sol;
    vector<int> res = sol.articulationPoints(n, adj);
    for (int v : res) cout << v << " ";
    cout << endl;
    return 0;
}

```

Time Complexity: $O(V+2E)$, where V = no. of vertices, E = no. of edges. It is because the algorithm is just a simple DFS traversal.

Space Complexity: $O(3V)$, where V = no. of vertices. $O(3V)$ is for the three arrays i.e. tin, low, and vis, each of size V .

Strongly Connected Components - Kosaraju's Algorithm: G-54

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
private:

```

```

    void dfs(int node, vector<int> &vis, vector<int>
adj[], stack<int> &st) {
        vis[node] = 1;
        for (auto it : adj[node]) {
            if (!vis[it]) {
                dfs(it, vis, adj, st);
            }
        }
        st.push(node);
    }

    void dfs3(int node, vector<int> &vis,
vector<int> adjT[]) {
        vis[node] = 1;
        for (auto it : adjT[node]) {
            if (!vis[it]) {
                dfs3(it, vis, adjT);
            }
        }
    }

public:
    int kosaraju(int V, vector<int> adj[]) {
        vector<int> vis(V, 0);
        stack<int> st;
        for (int i = 0; i < V; i++) {
            if (!vis[i]) {
                dfs(i, vis, adj, st);
            }
        }
        vector<int> adjT[V];
        for (int i = 0; i < V; i++) {
            vis[i] = 0; // reset visited
            for (auto it : adj[i]) {
                adjT[it].push_back(i);
            }
        }
        int scc = 0;
        while (!st.empty()) {

```

```

            int node = st.top();
            st.pop();
            if (!vis[node]) {
                scc++;
                dfs3(node, vis, adjT);
            }
        }
        return scc;
    }
};

int main() {
    int n = 5;
    int edges[5][2] = {
        {1, 0}, {0, 2},
        {2, 1}, {0, 3},
        {3, 4}
    };
    vector<int> adj[n];
    for (int i = 0; i < 5; i++) {
        adj[edges[i][0]].push_back(edges[i][1]);
    }
    Solution obj;
    int ans = obj.kosaraju(n, adj);
    cout << "The number of strongly connected
components is: " << ans << endl;
    return 0;
}

```

Time Complexity: $O(V + E)$, since each node and edge is visited at most twice (once in the original graph, once in the transposed graph).

Space Complexity: $O(V + E)$, due to the adjacency list, visited array, and recursion stack.

Dynamic Programming

Part - 1: Memoization

```

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

```

```

class Solution {
public:
    int fib(int n, vector<int>& dp) {
        if (n <= 1) return n;
        if (dp[n] != -1) return dp[n];
        dp[n] = fib(n - 1, dp) + fib(n - 2, dp);
        return dp[n];
    }
};

int main() {
    int n = 10;
    vector<int> dp(n + 1, -1);
    Solution sol;
    cout << sol.fib(n, dp);
    return 0;
}

```

Part -2: Tabulation

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int fib(int n) {
        if (n <= 1) return n;
        vector<int> dp(n + 1, 0);
        dp[0] = 0;
        dp[1] = 1;
        for (int i = 2; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
        return dp[n];
    }
};

int main() {
    int n = 10;
    Solution sol;
    cout << sol.fib(n);
}

```

```

return 0;
}

```

Part 3: Space Optimization

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int fib(int n) {
        if(n == 0) return 0;
        if(n == 1) return 1;
        int prev2 = 0;
        int prev = 1;
        int curr;
        for(int i = 2; i <= n; i++) {
            curr = prev + prev2;
            prev2 = prev;
            prev = curr;
        }
        return prev;
    }
};

int main() {
    Solution s;
    int n = 10;
    cout << s.fib(n);
    return 0;
}

```

Dynamic Programming : Climbing Stairs

```

#include <bits/stdc++.h>
using namespace std;
int main() {
    int n = 3;
    vector<int> dp(n + 1, -1);
    dp[0] = 1;
}

```

```

dp[1] = 1;
for (int i = 2; i <= n; i++) {
    dp[i] = dp[i - 1] + dp[i - 2];
}

cout << dp[n];
return 0;
}

```

Time Complexity: $O(n)$, since we compute each Fibonacci number from 2 to n exactly once.

Space Complexity: $O(n)$, for storing the DP array of size $n+1$.

Dynamic Programming : Frog Jump (DP 3)

Memoization Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int solve(int ind, const vector<int>& height,
vector<int>& dp) {
        if (ind == 0) return 0;
        if (dp[ind] != -1) return dp[ind];
        int jumpTwo = INT_MAX;
        int jumpOne = solve(ind - 1, height, dp) +
abs(height[ind] - height[ind - 1]);
        if (ind > 1) {
            jumpTwo = solve(ind - 2, height, dp) +
abs(height[ind] - height[ind - 2]);
        }
        return dp[ind] = min(jumpOne, jumpTwo);
    }
    int frogJump(const vector<int>& height) {
        if (height.empty()) return 0;
        int n = (int)height.size();
        vector<int> dp(n, -1);
        return solve(n - 1, height, dp);
    }
}

```

```

};
int main() {
    vector<int> height{30, 10, 60, 10, 60, 50};
    Solution sol;
    cout << sol.frogJump(height) << endl; //
Expected: 40
    return 0;
}

```

Time Complexity: $O(n)$, since each state (index) is computed once and stored in the DP array.

Space Complexity: $O(n)$ for the DP array + $O(n)$ recursion stack, leading to $O(n)$ overall.

Tabulation Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int frogJump(const vector<int>& height) {
        if (height.empty()) return 0;
        int n = (int)height.size();
        vector<int> dp(n, INT_MAX);
        dp[0] = 0;
        for (int ind = 1; ind < n; ind++) {
            int jumpOne = dp[ind - 1] + abs(height[ind]
- height[ind - 1]);
            int jumpTwo = INT_MAX;
            if (ind > 1) {
                jumpTwo = dp[ind - 2] + abs(height[ind]
- height[ind - 2]);
            }
            dp[ind] = min(jumpOne, jumpTwo);
        }
        return dp[n - 1];
    }
};
int main() {
    vector<int> height{30, 10, 60, 10, 60, 50};
    Solution sol;
}

```

```

    cout << sol.frogJump(height) << endl; //
Expected: 40

    return 0;
}

```

Time Complexity: $O(n)$, since we fill the DP array once with at most two computations per index.

Space Complexity: $O(n)$ for the DP array.

Space Optimization Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int frogJump(const vector<int>& height) {
        if (height.empty()) return 0;

        int n = (int)height.size();
        if (n == 1) return 0;

        int prev = 0;
        int prev2 = 0;
        for (int i = 1; i < n; i++) {
            int jumpTwo = INT_MAX;
            int jumpOne = prev + abs(height[i] -
height[i - 1]);
            if (i > 1) {
                jumpTwo = prev2 + abs(height[i] -
height[i - 2]);
            }
            int cur_i = min(jumpOne, jumpTwo);
            prev2 = prev;
            prev = cur_i;
        }
        return prev;
    }
};

int main() {
    vector<int> height{30, 10, 60, 10, 60, 50};
    Solution sol;

```

```

    cout << sol.frogJump(height) << endl; //
Expected: 40

    return 0;
}

```

Time Complexity: $O(n)$, since we iterate through the stones once.

Space Complexity: $O(1)$, because we only store two variables instead of an entire DP array.

Dynamic Programming: Frog Jump with k Distances (DP 4)

Memoization Approach

```

#include <bits/stdc++.h>

using namespace std;

int solveUtil(int ind, vector<int>& height,
vector<int>& dp, int k) {
    if (ind == 0) return 0;
    if (dp[ind] != -1) return dp[ind];
    int mmSteps = INT_MAX;
    for (int j = 1; j <= k; j++) {
        if (ind - j >= 0) {
            int jump = solveUtil(ind - j, height, dp, k) +
abs(height[ind] - height[ind - j]);
            mmSteps = min(jump, mmSteps);
        }
    }
    return dp[ind] = mmSteps;
}

int solve(int n, vector<int>& height, int k) {
    vector<int> dp(n, -1);
    return solveUtil(n - 1, height, dp, k);
}

int main() {
    vector<int> height{30, 10, 60, 10, 60, 50};
    int n = height.size();
    int k = 2;
    cout << solve(n, height, k) << endl;
}

```

```

    return 0;
}

```

Time Complexity: $O(n * k)$, where n is the number of stairs and k is the maximum jump allowed. For each index, we compute up to k recursive calls, each memoized.

Space Complexity: $O(n)$, due to the memoization array dp storing results for all indices from 0 to $n-1$.

Tabulation Approach

```

#include <bits/stdc++.h>

using namespace std;

int solveUtil(int n, vector<int>& height,
vector<int>& dp, int k) {
    dp[0] = 0;
    for (int i = 1; i < n; i++) {
        int mmSteps = INT_MAX;
        for (int j = 1; j <= k; j++) {
            if (i - j >= 0) {
                int jump = dp[i - j] + abs(height[i] -
height[i - j]);
                mmSteps = min(mmSteps, jump);
            }
        }
        dp[i] = mmSteps;
    }
    return dp[n - 1];
}

int solve(int n, vector<int>& height, int k) {
    vector<int> dp(n, -1);
    return solveUtil(n, height, dp, k);
}

int main() {
    vector<int> height{30, 10, 60, 10, 60, 50};
    int n = height.size();
    int k = 2;
    cout << solve(n, height, k) << endl;
    return 0;
}

```

Time Complexity: $O(n \times k)$, where n is the number of stones and k is the maximum number of jumps. We process n elements and for each we check up to k previous jumps.

Space Complexity: $O(n)$, for the dp array that stores the minimum cost to reach each stone

Maximum sum of non-adjacent elements (DP 5)

Memoization

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int solve(vector<int>& arr, int i, vector<int>&
dp) {
        if (i < 0) return 0;
        if (i == 0) return arr[0];
        if (dp[i] != -1) return dp[i];
        int pick = arr[i] + solve(arr, i - 2, dp);
        int notPick = solve(arr, i - 1, dp);
        return dp[i] = max(pick, notPick);
    }

    int maximumNonAdjacentSum(vector<int>&
arr) {
        int n = arr.size();
        vector<int> dp(n, -1);
        return solve(arr, n - 1, dp);
    }
};

int main() {
    vector<int> arr = {2, 1, 4, 9};
    Solution obj;
    cout << obj.maximumNonAdjacentSum(arr);
    return 0;
}

```

Time Complexity: $O(N)$, where N = total no. of elements in array. The overlapping subproblems will return the answer in constant time $O(1)$.

Space Complexity: $O(N+N)$, extra space used for memoization and auxiliary stack space.

Tabulation

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int maximumNonAdjacentSum(vector<int>& arr) {
        int n = arr.size();
        if (n == 1) return arr[0];
        vector<int> dp(n);
        dp[0] = arr[0];
        dp[1] = max(arr[0], arr[1]);
        for (int i = 2; i < n; i++) {
            dp[i] = max(arr[i] + dp[i - 2], dp[i - 1]);
        }
        return dp[n - 1];
    }
};

int main() {
    vector<int> arr = {2, 1, 4, 9};
    Solution obj;
    cout << obj.maximumNonAdjacentSum(arr);
    return 0;
}
```

Time Complexity: $O(N)$, every element of array is processed once.

Space Complexity: $O(N)$, extra space used to store DP array.

Space Optimization

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int maxSum(vector<int>& nums) {
        if (nums.empty()) return 0;
```

```
        int prev2 = 0;
        int prev = nums[0];
        for (int i = 1; i < nums.size(); i++) {
            int include = nums[i] + prev2;
            int exclude = prev;
            int curr = max(include, exclude);
            prev2 = prev;
            prev = curr;
        }
        return prev;
    }
};

int main() {
    vector<int> arr = {3, 2, 5, 10, 7};
    Solution obj;
    cout << obj.maxSum(arr) << endl;
    return 0;
}
```

Time Complexity: $O(N)$, every element of array is processed once.

Space Complexity: $O(1)$, only constant variables are used.

Dynamic Programming: House Robber (DP 6)

```
#include <bits/stdc++.h>

using namespace std;

long long int solve(vector<int>& arr) {
    int n = arr.size();
    if (n == 1) return arr[0];
    long long int prev = arr[0];
    long long int prev2 = 0;
    for (int i = 1; i < n; i++) {
        long long int pick = arr[i];
        if (i > 1) pick += prev2;
        long long int nonPick = prev;
        long long int cur_i = max(pick, nonPick);
```

```

    prev2 = prev;
    prev = cur_i;
}
return prev;
}

long long int robStreet(int n, vector<int> &arr) {
    if (n == 0) return 0;
    if (n == 1) return arr[0];
    vector<int> arr1, arr2;
    for (int i = 0; i < n; i++) {
        if (i != 0) arr1.push_back(arr[i]);
        if (i != n - 1) arr2.push_back(arr[i]);
    }
    long long int ans1 = solve(arr1);
    long long int ans2 = solve(arr2);
    return max(ans1, ans2);
}

int main() {
    vector<int> arr{1, 5, 1, 2, 6};
    int n = arr.size();
    cout << robStreet(n, arr);
    return 0;
}

```

Time Complexity: $O(n)$, where n is the number of houses. We process each house once in two separate passes (excluding first and last), but each pass is linear.

Space Complexity: $O(1)$, since only constant extra space is used for variables like `prev` and `prev2`; no additional DP array is maintained.

Dynamic Programming: Ninja's Training (DP 7)

Memoization Approach

```

#include <bits/stdc++.h>

using namespace std;

int f(int day, int last, vector<vector<int>> &points,
vector<vector<int>> &dp) {
    if (dp[day][last] != -1) return dp[day][last];

```

```

    if (day == 0) {
        int maxi = 0;
        for (int i = 0; i <= 2; i++) {
            if (i != last)
                maxi = max(maxi, points[0][i]);
        }
        return dp[day][last] = maxi;
    }

    int maxi = 0;
    for (int i = 0; i <= 2; i++) {
        if (i != last) {
            int activity = points[day][i] + f(day - 1, i,
points, dp);
            maxi = max(maxi, activity);
        }
    }

    return dp[day][last] = maxi;
}

int ninjaTraining(int n, vector<vector<int>>
&points) {
    vector<vector<int>> dp(n, vector<int>(4, -1));

    return f(n - 1, 3, points, dp);
}

int main() {
    vector<vector<int>> points = {{10, 40, 70},
                                {20, 50, 80},
                                {30, 60, 90}};

    int n = points.size();
    cout << ninjaTraining(n, points);
}

```

Time Complexity: $O(n \times 4 \times 3) = O(n)$

- There are n days and 4 possible values for `last` (0, 1, 2, 3).
- For each state, we loop through 3 possible activities.

Space Complexity: $O(n \times 4) + O(n)$ for recursion stack

- $O(n \times 4)$ for the DP table

- $O(n)$ for recursion call stack in the worst case

Tabulation Approach

```
#include <bits/stdc++.h>

using namespace std;

int ninjaTraining(int n, vector<vector<int>>& points) {
    vector<vector<int>> dp(n, vector<int>(4, 0));
    dp[0][0] = max(points[0][1], points[0][2]);
    dp[0][1] = max(points[0][0], points[0][2]);
    dp[0][2] = max(points[0][0], points[0][1]);
    dp[0][3] = max(points[0][0], max(points[0][1], points[0][2]));
    for (int day = 1; day < n; day++) {
        for (int last = 0; last < 4; last++) {
            dp[day][last] = 0;
            for (int task = 0; task <= 2; task++) {
                if (task != last) {
                    int activity = points[day][task] + dp[day - 1][task];
                    dp[day][last] = max(dp[day][last], activity);
                }
            }
        }
    }
    return dp[n - 1][3];
}

int main() {
    vector<vector<int>> points = {{10, 40, 70},
                                {20, 50, 80},
                                {30, 60, 90}};

    int n = points.size();
    cout << ninjaTraining(n, points);
}
```

Time Complexity: $O(n \times 4 \times 3) = O(n)$

- For each day, we loop through 4 values of last and 3 values of task.

Space Complexity: $O(n \times 4)$

- DP table of size $n \times 4$ is maintained to store intermediate results.

Space Optimization Approach

```
#include <bits/stdc++.h>

using namespace std;

int ninjaTraining(int n, vector<vector<int>>& points) {
    vector<int> prev(4, 0);
    prev[0] = max(points[0][1], points[0][2]);
    prev[1] = max(points[0][0], points[0][2]);
    prev[2] = max(points[0][0], points[0][1]);
    prev[3] = max(points[0][0], max(points[0][1], points[0][2]));
    for (int day = 1; day < n; day++) {
        vector<int> temp(4, 0);
        for (int last = 0; last < 4; last++) {
            temp[last] = 0;
            for (int task = 0; task <= 2; task++) {
                if (task != last) {
                    temp[last] = max(temp[last], points[day][task] + prev[task]);
                }
            }
        }
        prev = temp;
    }
    return prev[3];
}

int main() {
    vector<vector<int>> points = {{10, 40, 70},
                                {20, 50, 80},
                                {30, 60, 90}};

    int n = points.size();
    cout << ninjaTraining(n, points);
}
```

Time Complexity: $O(n \times 4 \times 3) = O(n)$

- For each day (n), we loop through 4 values of last and for each last, try 3 task options.

Space Complexity: $O(4) = O(1)$

- Only two arrays of size 4 are used (prev and temp), making it constant space.

Grid Unique Paths : DP on Grids (DP8)

Memoization Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    int func(int i, int j, vector<vector<int>>& dp){
        if (i == 0 && j == 0) return 1;
        if (i < 0 || j < 0) return 0;
        if (dp[i][j] != -1) return dp[i][j];
        int up = func(i - 1, j, dp);
        int left = func(i, j - 1, dp);
        return dp[i][j] = up + left;
    }
public:
    int uniquePaths(int m, int n) {
        vector<vector<int>> dp(m, vector<int>(n, -1));
        return func(m-1, n-1, dp);
    }
};

int main() {
    int m = 3;
    int n = 2;

    Solution sol;

    cout << "Number of ways: " <<
    sol.uniquePaths(m, n) << endl;

    return 0;
}
```

Time Complexity: $O(m * n)$, where m is the number of rows and n is the number of columns in the grid. This is because we are filling up a 2D DP array of size m x n, and each cell takes constant time to compute.

Space Complexity: $O(m * n)$, due to the storage of the

DP array which keeps track of the number of unique paths to each cell in the grid.

Tabulation Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    int func(int m, int n, vector<vector<int>>& dp){
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (i == 0 && j == 0) {
                    dp[i][j] = 1;
                    continue;
                }
                int up = 0;
                int left = 0;
                if (i > 0)
                    up = dp[i - 1][j];
                if (j > 0)
                    left = dp[i][j - 1];
                dp[i][j] = up + left;
            }
        }
        return dp[m - 1][n - 1];
    }
public:
    int uniquePaths(int m, int n) {
        vector<vector<int>> dp(m, vector<int>(n, -1));
        return func(m, n, dp);
    }
};

int main() {
    int m = 3;
    int n = 2;

    Solution sol;
```

```

        cout << "Number of ways: " <<
sol.uniquePaths(m, n) << endl;

        return 0;
    }

```

Time Complexity: $O(m * n)$, where m is the number of rows and n is the number of columns in the grid. This is because we are filling up a 2D DP array of size $m \times n$, and each cell takes constant time to compute.

Space Complexity: $O(m * n)$, as we are using a 2D array to store the results of subproblems. However, we can optimize this to $O(n)$ by using a 1D array and updating it in place

Space Optimized Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    int func(int m, int n){
        vector<int> prev(n, 0);
        for (int i = 0; i < m; i++) {
            vector<int> temp(n, 0);
            for (int j = 0; j < n; j++) {
                if (i == 0 && j == 0) {
                    temp[j] = 1;
                    continue;
                }
                int up = 0;
                int left = 0;
                if (i > 0)
                    up = prev[j];
                if (j > 0)
                    left = temp[j - 1];
                temp[j] = up + left;
            }
            prev = temp;
        }
        return prev[n - 1];
    }
public:

```

```

    int uniquePaths(int m, int n) {
        return func(m, n);
    }
};

int main() {
    int m = 3;
    int n = 2;

    Solution sol;

    cout << "Number of ways: " <<
sol.uniquePaths(m, n) << endl;

    return 0;
}

```

Time Complexity: $O(m * n)$, where m is the number of rows and n is the number of columns in the grid. This is because we are iterating through each cell in the grid once.

Space Complexity: $O(n)$, as we are using two arrays of size n to store the current and previous rows. This is an optimization from the previous approaches that used a 2D array.

Grid Unique Paths 2 (DP 9)

Memoization Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    int func(int i, int j, vector<vector<int>>& matrix,
vector<vector<int>> &dp){
        if (i < 0 || j < 0 || matrix[i][j] == 1) return 0;
        else if(i == 0 && j == 0) return 1;
        if(dp[i][j] != -1) return dp[i][j];
        int up = func(i - 1, j, matrix, dp);
        int left = func(i, j - 1, matrix, dp);
        return dp[i][j] = up + left;
    }
public:
    int
uniquePathsWithObstacles(vector<vector<int>>&
matrix) {

```

```

        int m = matrix.size();
        int n = matrix[0].size();
        vector<vector<int>>> dp(m, vector<int>(n, -
1));
        return func(m-1, n-1, matrix, dp);
    }
};

int main() {
    vector<vector<int>>> maze{
        {0, 0, 0},
        {0, 1, 0},
        {0, 0, 0}
    };
    Solution sol;

    cout << "Number of paths with obstacles: " <<
sol.uniquePathsWithObstacles(maze) << endl;

    return 0;
}

```

Time Complexity: $O(m * n)$, where m is the number of rows and n is the number of columns in the matrix. This is because we are filling up a 2D DP array of size $m \times n$, and each cell takes constant time to compute.

Space Complexity: $O((N-1)+(M-1)) + O(M*N)$ We are using a recursion stack space: $O((N-1)+(M-1))$, here $(N-1)+(M-1)$ is the path length and an external DP Array of size ' $M*N$ '.

Tabulation Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    int func(int m, int n, vector<vector<int>>&
matrix, vector<vector<int>>& dp) {
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == 1) {
                    dp[i][j] = 0;
                    continue;
                }
                if (i == 0 && j == 0) {

```

```

                    dp[i][j] = 1;
                    continue;
                }
                int up = 0;
                int left = 0;
                if (i > 0)
                    up = dp[i - 1][j];
                if (j > 0)
                    left = dp[i][j - 1];
                dp[i][j] = up + left;
            }
        }
        return dp[m - 1][n - 1];
    }

public:
    int
uniquePathsWithObstacles(vector<vector<int>>&
matrix) {
        int m = matrix.size();
        int n = matrix[0].size();

        vector<vector<int>>> dp(m, vector<int>(n, 0));

        return func(m, n, matrix, dp);
    }
};

int main() {
    vector<vector<int>>> maze{
        {0, 0, 0},
        {0, 1, 0},
        {0, 0, 0}
    };
    Solution sol;

    cout << "Number of paths with obstacles: " <<
sol.uniquePathsWithObstacles(maze) << endl;

    return 0;
}

```

Time Complexity: $O(m * n)$, where m is the number of rows and n is the number of columns in the matrix. This is because we are filling up a 2D DP array of size $m \times n$,

and each cell takes constant time to compute.

Space Complexity: $O(m * n)$, where m is the number of rows and n is the number of columns in the matrix. This is due to the use of a 2D DP array to store intermediate results.

Space Optimized Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    int func(int m, int n, vector<vector<int>>& matrix) {
        vector<int> prev(n, 0), curr(n, 0);
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == 1) {
                    curr[j] = 0;
                    continue;
                }
                if (i == 0 && j == 0) {
                    curr[j] = 1;
                    continue;
                }
                int up = 0;
                int left = 0;
                if (i > 0)
                    up = prev[j];
                if (j > 0)
                    left = curr[j - 1];
                curr[j] = up + left;
            }
            prev = curr;
        }
        return prev[n-1];
    }
public:
```

```
    int
    uniquePathsWithObstacles(vector<vector<int>>& matrix) {
        int m = matrix.size();
        int n = matrix[0].size();
        return func(m, n, matrix);
    }
};

int main() {
    vector<vector<int>> maze{
        {0, 0, 0},
        {0, 1, 0},
        {0, 0, 0}
    };
    Solution sol;
    cout << "Number of paths with obstacles: " <<
    sol.uniquePathsWithObstacles(maze) << endl;
    return 0;
}
```

Time Complexity: $O(m * n)$, where m is the number of rows and n is the number of columns in the matrix. This is because we are filling up a 2D DP array of size m x n, and each cell takes constant time to compute.

Space Complexity: $O(n)$, where n is the number of columns in the matrix. This is due to the use of two 1D arrays to store intermediate results, instead of a full 2D DP array.

Minimum Path Sum In a Grid (DP 10)

Memoization Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int minPath(int i, int j,
        vector<vector<int>> &grid,
        vector<vector<int>> &dp) {
        if (i == 0 && j == 0)
            return grid[0][0];
```

```

    if (i < 0 || j < 0)
        return 1e9;
    if (dp[i][j] != -1)
        return dp[i][j];
    int up = grid[i][j] +
        minPath(i - 1, j, grid, dp);
    int left = grid[i][j] +
        minPath(i, j - 1, grid, dp);
    return dp[i][j] = min(up, left);
}

int minPathSum(vector<vector<int>> &grid) {
    int n = grid.size();
    int m = grid[0].size();
    vector<vector<int>> dp(n,
        vector<int>(m, -1));
    return minPath(n - 1, m - 1, grid, dp);
}

};

int main() {
    vector<vector<int>> grid = {
        {5, 9, 6},
        {11, 5, 2}
    };

    Solution obj;

    cout << "Minimum sum path: "
        << obj.minPathSum(grid) << endl;

    return 0;
}

```

Time Complexity: $O(N*M)$, at max, there will be $N*M$ calls of recursion..

Space Complexity: $O(M+N) + O(N*M)$, additional space for recursion stack and memo table

Tabulation Approach

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Solution {
```

```
public:
```

```

    int minPathSum(vector<vector<int>> &matrix)
    {
        int n = matrix.size();
        int m = matrix[0].size();
        vector<vector<int>> dp(n, vector<int>(m, 0));
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (i == 0 && j == 0)
                    dp[i][j] = matrix[i][j];
                else {
                    int up = matrix[i][j];
                    if (i > 0) up += dp[i - 1][j];
                    else up += 1e9;
                    int left = matrix[i][j];
                    if (j > 0) left += dp[i][j - 1];
                    else left += 1e9;
                    dp[i][j] = min(up, left);
                }
            }
        }
        return dp[n - 1][m - 1];
    }

};

int main() {
    vector<vector<int>> matrix{
        {5, 9, 6},
        {11, 5, 2}
    };

    Solution obj;

    cout << "Minimum sum path: " <<
        obj.minPathSum(matrix) << endl;

    return 0;
}

```

Time Complexity: $O(N*M)$, entire grid is visited atleast once.

Space Complexity: $O(N*M)$, space used for DP array.

Space Optimization Approach

```

#include <bits/stdc++.h>

using namespace std;

class solution {
public:
    int minSumPath(int n, int m,
vector<vector<int>>> &matrix) {
        vector<int> prev(m, 0);
        for (int i = 0; i < n; i++) {
            vector<int> temp(m, 0);
            for (int j = 0; j < m; j++) {
                if (i == 0 && j == 0)
                    temp[j] = matrix[i][j];
                else {
                    int up = matrix[i][j];
                    if (i > 0)
                        up += prev[j];
                    else
                        up += 1e9;
                    int left = matrix[i][j];
                    if (j > 0)
                        left += temp[j - 1];
                    else
                        left += 1e9;
                    temp[j] = min(up, left);
                }
            }
            prev = temp;
        }
        return prev[m - 1];
    }
};

int main() {
    vector<vector<int>>> matrix{
        {5, 9, 6},
        {11, 5, 2}
    };

```

```

        int n = matrix.size();
        int m = matrix[0].size();
        Solution obj;
        cout << "Minimum sum path: " <<
obj.minSumPath(n, m, matrix) << endl;
        return 0;
    }

```

Time Complexity: $O(N \cdot M)$, every element of grid is visited atleast once.

Space Complexity: $O(N)$, we only use one array for storing rows.

Minimum path sum in Triangular Grid (DP 11)

Memoization Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int solve(int i, int j, vector<vector<int>>>
&triangle, int n, vector<vector<int>>> &dp) {
        if (dp[i][j] != -1)
            return dp[i][j];
        if (i == n - 1)
            return triangle[i][j];
        int down = triangle[i][j] + solve(i + 1, j,
triangle, n, dp);
        int diag = triangle[i][j] + solve(i + 1, j + 1,
triangle, n, dp);
        return dp[i][j] = min(down, diag);
    }

    int minimumPathSum(vector<vector<int>>>
&triangle) {
        int n = triangle.size();
        vector<vector<int>>> dp(n, vector<int>(n, -1));
        return solve(0, 0, triangle, n, dp);
    }
};

int main() {

```

```

Solution obj;

vector<vector<int>> triangle{

    {1},

    {2, 3},

    {3, 6, 7},

    {8, 9, 6, 10}

};

cout << obj.minimumPathSum(triangle);

return 0;

}

```

Time Complexity: $O(N*N)$, we fill our complete 2D DP table one by one.

Space Complexity: $O(N) + O(N*N)$, additional space for recursion stack and 2D DP table.

Tabulation Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

    int minimumPathSum(vector<vector<int>>
    &triangle, int n) {

        vector<vector<int>> dp(n, vector<int>(n, 0));

        for (int j = 0; j < n; j++) {

            dp[n - 1][j] = triangle[n - 1][j];

        }

        for (int i = n - 2; i >= 0; i--) {

            for (int j = i; j >= 0; j--) {

                int down = triangle[i][j] + dp[i + 1][j];

                int diag = triangle[i][j] + dp[i + 1][j + 1];

                dp[i][j] = min(down, diag);

            }

        }

        return dp[0][0];

    }

};

int main() {

    vector<vector<int>> triangle{

```

```

        {1},

        {2, 3},

        {3, 6, 7},

        {8, 9, 6, 10}

    };

    int n = triangle.size();

    Solution solver;

    cout << solver.minimumPathSum(triangle, n);

    return 0;

}

```

Time Complexity: $O(N*N)$, entire triangular grid is visited atleast once.

Space Complexity: $O(N*N)$, space used for 2D DP array.

Space Optimization Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

    int minimumPathSum(vector<vector<int>>
    &triangle, int n) {

        vector<int> front(n, 0);

        vector<int> cur(n, 0);

        for (int j = 0; j < n; j++) {

            front[j] = triangle[n - 1][j];

        }

        for (int i = n - 2; i >= 0; i--) {

            for (int j = i; j >= 0; j--) {

                int down = triangle[i][j] + front[j];

                int diagonal = triangle[i][j] + front[j + 1];

                cur[j] = min(down, diagonal);

            }

            front = cur;

        }

        return front[0];

    }

};

```

```
int main() {
    vector<vector<int>> triangle{
        {1},
        {2, 3},
        {3, 6, 7},
        {8, 9, 6, 10}
    };
    int n = triangle.size();
    Solution obj;
    cout << obj.minimumPathSum(triangle, n);
    return 0;
}
```

Time Complexity: $O(N*N)$, every element of triangular grid is visited atleast once.

Space Complexity: $O(N)$, we only use one array for storing rows.

Minimum/Maximum Falling Path Sum (DP-12)

Memoization

```
#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int dfs(int row, int col, vector<vector<int>>& matrix, vector<vector<int>>& dp) {
        if (col < 0 || col >= matrix[0].size()) {
            return 1e9;
        }
        if (row == matrix.size() - 1) {
            return matrix[row][col];
        }
        if (dp[row][col] != -1) {
            return dp[row][col];
        }
        int down = dfs(row + 1, col, matrix, dp);
        int downLeft = dfs(row + 1, col - 1, matrix, dp);
```

```
        int downRight = dfs(row + 1, col + 1, matrix, dp);
        int ans = matrix[row][col] + min({down, downLeft, downRight});
        return dp[row][col] = ans;
    }
    int minFallingPathSum(vector<vector<int>>& matrix) {
        int n = matrix.size();
        int m = matrix[0].size();
        vector<vector<int>> dp(n, vector<int>(m, -1));
        int minSum = 1e9;
        for (int col = 0; col < m; col++) {
            minSum = min(minSum, dfs(0, col, matrix, dp));
        }
        return minSum;
    }
};
```

```
int main() {
    vector<vector<int>> matrix = {
        {1, 4, 3, 1},
        {2, 3, -1, -1},
        {1, 1, -1, 8}
    };
    Solution sol;
    cout << "Minimum Falling Path Sum: " << sol.minFallingPathSum(matrix) << endl;
    return 0;
}
```

Time Complexity: $O(N*M)$, we call our recursive function for every element in the grid.

Space Complexity: $O(N*M) + O(N)$, space used to store memo table and recursive stack space.

Tabulation

```
#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
```

```

int minFallingPathSum(vector<vector<int>>&
matrix) {
    int n = matrix.size();
    int m = matrix[0].size();
    vector<vector<int>> dp(n, vector<int>(m, 0));
    for (int col = 0; col < m; col++) {
        dp[n - 1][col] = matrix[n - 1][col];
    }
    for (int row = n - 2; row >= 0; row--) {
        for (int col = 0; col < m; col++) {
            int down = dp[row + 1][col];
            int downLeft = (col > 0) ? dp[row + 1][col - 1] : 1e9;
            int downRight = (col < m - 1) ? dp[row + 1][col + 1] : 1e9;
            dp[row][col] = matrix[row][col] +
min({down, downLeft, downRight});
        }
    }
    int minSum = *min_element(dp[0].begin(),
dp[0].end());
    return minSum;
}

int main() {
    vector<vector<int>> matrix = {
        {1, 4, 3, 1},
        {2, 3, -1, -1},
        {1, 1, -1, 8}
    };
    Solution sol;
    cout << "Minimum Falling Path Sum: " <<
sol.minFallingPathSum(matrix) << endl;
    return 0;
}

```

Time Complexity: $O(N \times M)$, we call our recursive function for every element in the grid.
Space Complexity: $O(N \times M)$, space used to store the 2D DP array.

Space Optimization

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int minFallingPathSum(vector<vector<int>>&
matrix) {
        int n = matrix.size();
        int m = matrix[0].size();
        vector<int> dp(matrix[n - 1]);
        for (int row = n - 2; row >= 0; row--) {
            vector<int> curr(m, 0);
            for (int col = 0; col < m; col++) {
                int down = dp[col];
                int downLeft = (col > 0) ? dp[col - 1] :
1e9;
                int downRight = (col < m - 1) ? dp[col + 1] :
1e9;
                curr[col] = matrix[row][col] +
min({down, downLeft, downRight});
            }
            dp = curr; // update dp for next iteration
        }
    }
}

int main() {
    vector<vector<int>> matrix = {
        {1, 4, 3, 1},
        {2, 3, -1, -1},
        {1, 1, -1, 8}
    };
    Solution sol;
    cout << "Minimum Falling Path Sum: " <<
sol.minFallingPathSum(matrix) << endl;
    return 0;
}

```

Time Complexity: $O(N*M)$, we call our recursive function for every element in the grid.
Space Complexity: $O(N)$, space used to store the 2 arrays having values of current and previous rows.

3-d DP : Ninja and his friends (DP-13)

Memoization Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int solve(int i, int j1, int j2, int n, int m,
              vector<vector<int>>& grid,
              vector<vector<vector<int>>>& dp) {
        if (j1 < 0 || j1 >= m || j2 < 0 || j2 >= m)
            return -1e9;
        if (i == n - 1) {
            if (j1 == j2) return grid[i][j1];
            else return grid[i][j1] + grid[i][j2];
        }
        if (dp[i][j1][j2] != -1) return dp[i][j1][j2];
        int maxi = -1e9;
        int curr = (j1 == j2) ? grid[i][j1] : grid[i][j1] +
            grid[i][j2];
        for (int dj1 = -1; dj1 <= 1; dj1++) {
            for (int dj2 = -1; dj2 <= 1; dj2++) {
                int ans = curr + solve(i + 1, j1 + dj1, j2 +
                    dj2,
                    n, m, grid, dp);
                maxi = max(maxi, ans);
            }
        }
        return dp[i][j1][j2] = maxi;
    }

    int maximumChocolates(int n, int m,
        vector<vector<int>>& grid) {
        vector<vector<vector<int>>> dp(n,
```

```
        vector<vector<int>>(m, vector<int>(m, -
            1)));
        return solve(0, 0, m - 1, n, m, grid, dp);
    }
};

int main() {
    vector<vector<int>> grid = {
        {2, 3, 1, 2},
        {3, 4, 2, 2},
        {5, 6, 3, 5}
    };
    int n = grid.size(), m = grid[0].size();
    Solution obj;
    cout << obj.maximumChocolates(n, m, grid) <<
        endl;
    return 0;
}
```

Time Complexity: $O(N*M*M) * 9$, At max, there will be $N*M*M$ calls of recursion to solve a new problem and in every call, two nested loops together run for 9 times.

Space Complexity: $O(N) + O(N*M*M)$, We are using a recursion stack space: $O(N)$, where N is the path length and an external DP Array of size ' $N*M*M$ '

Tabulation Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int maximumChocolates(int n, int m,
        vector<vector<int>>& grid) {
        vector<vector<vector<int>>> dp(n,
            vector<vector<int>>(m, vector<int>(m,
                0)));
        for (int j1 = 0; j1 < m; j1++) {
            for (int j2 = 0; j2 < m; j2++) {
                if (j1 == j2) dp[n-1][j1][j2] = grid[n-
                    1][j1];
                else dp[n-1][j1][j2] = grid[n-1][j1] +
                    grid[n-1][j2];
            }
        }
```

```

    }
    for (int i = n - 2; i >= 0; i--) {
        for (int j1 = 0; j1 < m; j1++) {
            for (int j2 = 0; j2 < m; j2++) {
                int maxi = -1e9;
                int curr = (j1 == j2) ? grid[i][j1]
                    : grid[i][j1] + grid[i][j2];
                for (int dj1 = -1; dj1 <= 1; dj1++) {
                    for (int dj2 = -1; dj2 <= 1; dj2++) {
                        int newJ1 = j1 + dj1;
                        int newJ2 = j2 + dj2;
                        if (newJ1 >= 0 && newJ1 < m
&&
                            newJ2 >= 0 && newJ2 < m) {
                            maxi = max(maxi, curr +
                                dp[i+1][newJ1][newJ2]);
                        } else {
                            maxi = max(maxi, (int)-1e9);
                        }
                    }
                }
                dp[i][j1][j2] = maxi;
            }
        }
    }
    return dp[0][0][m-1];
}
};

int main() {
    vector<vector<int>>> grid = {
        {2, 3, 1, 2},
        {3, 4, 2, 2},
        {5, 6, 3, 5}
    };
    int n = grid.size(), m = grid[0].size();
    Solution obj;

```

```

        cout << obj.maximumChocolates(n, m, grid) <<
endl;

        return 0;
    }

```

Time Complexity: $O(N*M*M)*9$, The outer nested loops run for $(N*M*M)$ times and the inner two nested loops run for 9 times.

Space Complexity: $O(N*M*M)$, We are using an external array of size ' $N*M*M$ '. The stack space will be eliminated

Space Optimization Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int maximumChocolates(int n, int m,
        vector<vector<int>>>& grid) {

        vector<vector<int>>> next(m, vector<int>(m,
0));
        vector<vector<int>>> curr(m, vector<int>(m,
0));

        for (int j1 = 0; j1 < m; j1++) {
            for (int j2 = 0; j2 < m; j2++) {
                if (j1 == j2) next[j1][j2] = grid[n-1][j1];
                else next[j1][j2] = grid[n-1][j1] + grid[n-
1][j2];
            }
        }

        for (int i = n - 2; i >= 0; i--) {
            for (int j1 = 0; j1 < m; j1++) {
                for (int j2 = 0; j2 < m; j2++) {
                    int maxi = -1e9;
                    int currChoco = (j1 == j2) ? grid[i][j1]
                        : grid[i][j1] +
grid[i][j2];
                    for (int dj1 = -1; dj1 <= 1; dj1++) {
                        for (int dj2 = -1; dj2 <= 1; dj2++) {
                            int newJ1 = j1 + dj1;
                            int newJ2 = j2 + dj2;

```

```

        if (newJ1 >= 0 && newJ1 < m
&&
            newJ2 >= 0 && newJ2 < m) {
            maxi = max(maxi, currChoco +
                next[newJ1][newJ2]);
        } else {
            maxi = max(maxi, (int)-1e9);
        }
    }
    curr[j1][j2] = maxi;
}
next = curr;
}
return next[0][m-1];
}
};

int main() {
    vector<vector<int>> grid = {
        {2, 3, 1, 2},
        {3, 4, 2, 2},
        {5, 6, 3, 5}
    };
    int n = grid.size(), m = grid[0].size();
    Solution obj;
    cout << obj.maximumChocolates(n, m, grid) <<
endl;
    return 0;
}

```

Time Complexity: $O(N*M*M)*9$, The outer nested loops run for $(N*M*M)$ times and the inner two nested loops run for 9 times.

Space Complexity: $O(M*M)$, We are using an external array of size ' $M*M$ '.

Subset sum equal to target (DP-14)

Memorization Approach

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    bool subsetSumUtil(int ind, int target,
vector<int>& arr, vector<vector<int>>& dp) {
        if (target == 0) return true;
        if (ind == 0) return arr[0] == target;
        if (dp[ind][target] != -1) return dp[ind][target];
        bool notTaken = subsetSumUtil(ind - 1, target,
arr, dp);
        bool taken = false;
        if (arr[ind] <= target) {
            taken = subsetSumUtil(ind - 1, target -
arr[ind], arr, dp);
        }
        return dp[ind][target] = notTaken || taken;
    }
    bool subsetSumToK(int n, int k, vector<int>&
arr) {
        vector<vector<int>> dp(n, vector<int>(k + 1,
-1));
        return subsetSumUtil(n - 1, k, arr, dp);
    }
};

int main() {
    vector<int> arr = {1, 2, 3, 4};
    int k = 4;
    int n = arr.size();
    Solution sol;
    if (sol.subsetSumToK(n, k, arr))
        cout << "Subset with the given target found";
    else
        cout << "Subset with the given target not
found";
    return 0;
}

```

Time Complexity: $O(N \cdot K)$, There are $N \cdot K$ states therefore at max ' $N \cdot K$ ' new problems will be solved.

Space Complexity: $O(N \cdot K) + O(N)$, We are using a recursion stack space($O(N)$) and a 2D array ($O(N \cdot K)$).

Tabulation Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool subsetSumToK(int n, int k, vector<int>
&arr) {
        vector<vector<bool>> dp(n, vector<bool>(k +
1, false));

        for (int i = 0; i < n; i++) {
            dp[i][0] = true;
        }

        if (arr[0] <= k) {
            dp[0][arr[0]] = true;
        }

        for (int ind = 1; ind < n; ind++) {
            for (int target = 1; target <= k; target++) {
                bool notTaken = dp[ind - 1][target];

                bool taken = false;

                if (arr[ind] <= target) {
                    taken = dp[ind - 1][target - arr[ind]];
                }

                dp[ind][target] = notTaken || taken;
            }
        }

        return dp[n - 1][k];
    }
};

int main() {
    vector<int> arr = {1, 2, 3, 4};

    int k = 4;

    int n = arr.size();

    Solution sol;
```

```
    if (sol.subsetSumToK(n, k, arr))

        cout << "Subset with the given target found";

    else

        cout << "Subset with the given target not
found";

    return 0;
}
```

Time Complexity: $O(N \cdot K)$, There are two nested loops

Space Complexity: $O(N \cdot K)$, We are using an external array of size ' $N \cdot K$ '. Stack Space is eliminated.

Space Optimization Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool subsetSumToK(int n, int k, vector<int>
&arr) {
        vector<bool> prev(k + 1, false);

        prev[0] = true;

        if (arr[0] <= k) {
            prev[arr[0]] = true;
        }

        for (int ind = 1; ind < n; ind++) {
            vector<bool> cur(k + 1, false);

            cur[0] = true;

            for (int target = 1; target <= k; target++) {
                bool notTaken = prev[target];

                bool taken = false;

                if (arr[ind] <= target) {
                    taken = prev[target - arr[ind]];
                }

                cur[target] = notTaken || taken;
            }

            prev = cur;
        }

        return prev[k];
    }
}
```

```

};

int main() {
    vector<int> arr = {1, 2, 3, 4};
    int k = 4;
    int n = arr.size();
    Solution sol;
    if (sol.subsetSumToK(n, k, arr))
        cout << "Subset with the given target found";
    else
        cout << "Subset with the given target not found";
    return 0;
}

```

Time Complexity: $O(N \cdot K)$, There are three nested loops

Space Complexity: $O(K)$, We are using an external array of size 'K+1' to store only one row.

Partition Equal Subset Sum (DP-15)

Memoization

```

#include <bits/stdc++.h>

using namespace std;

bool subsetSumUtil(int ind, int target,
vector<int>& arr, vector<vector<int>>& dp) {
    if (target == 0)
        return true;
    if (ind == 0)
        return arr[0] == target;
    if (dp[ind][target] != -1)
        return dp[ind][target];
    bool notTaken = subsetSumUtil(ind - 1, target,
arr, dp);
    bool taken = false;
    if (arr[ind] <= target)
        taken = subsetSumUtil(ind - 1, target -
arr[ind], arr, dp);
    return dp[ind][target] = notTaken || taken;
}

```

```

bool canPartition(int n, vector<int>& arr) {
    int totSum = 0;
    for (int i = 0; i < n; i++) {
        totSum += arr[i];
    }
    if (totSum % 2 == 1)
        return false;
    else {
        int k = totSum / 2;
        vector<vector<int>> dp(n, vector<int>(k + 1,
-1));
        return subsetSumUtil(n - 1, k, arr, dp);
    }
}

int main() {
    vector<int> arr = {2, 3, 3, 3, 4, 5};
    int n = arr.size();
    if (canPartition(n, arr))
        cout << "The Array can be partitioned into
two equal subsets";
    else
        cout << "The Array cannot be partitioned into
two equal subsets";
    return 0;
}

```

Time Complexity: $O(N \cdot K)$, there are total $N \cdot K$ states, where N is the length of array and K is the target sum i.e. Half of the sum of all elements of the array.

Space Complexity: $O(N \cdot K) + O(N)$, we use a memo table to avoid recomputation. Extra auxiliary stack space is used for recursion.

Tabulation

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool canPartition(int n, vector<int>& arr) {
        int totalSum = 0;
        for (int i = 0; i < n; i++) {

```

```

        totalSum += arr[i];
    }
    if (totalSum % 2 != 0) {
        return false;
    }
    int targetSum = totalSum / 2;
    vector<vector<bool>> dp(n,
vector<bool>(targetSum + 1, false));
    for (int i = 0; i < n; i++) {
        dp[i][0] = true;
    }
    if (arr[0] <= targetSum) {
        dp[0][arr[0]] = true;
    }
    for (int index = 1; index < n; index++) {
        for (int target = 1; target <= targetSum;
target++) {
            bool notTaken = dp[index - 1][target];
            bool taken = false;
            if (arr[index] <= target) {
                taken = dp[index - 1][target -
arr[index]];
            }
            dp[index][target] = notTaken || taken;
        }
    }
    return dp[n - 1][targetSum];
}
};

int main() {
    vector<int> arr = {2, 3, 3, 3, 4, 5};
    int n = arr.size();
    Solution solver;
    if (solver.canPartition(n, arr)) {
        cout << "The array can be partitioned into two
equal subsets";
    } else {

```

```

        cout << "The array cannot be partitioned into
two equal subsets";
    }
    return 0;
}

```

Time Complexity: $O(N \cdot K)$, there are total $N \cdot K$ states, where N is the length of array and K is the target sum i.e. Half of the sum of all elements of the array.
Space Complexity: $O(N \cdot K)$, we use a 2D DP array to avoid recomputation.

Space Optimization

```

#include <bits/stdc++.h>

using namespace std;

bool canPartition(int n, vector<int>& arr) {
    int totSum = 0;
    for (int i = 0; i < n; i++) {
        totSum += arr[i];
    }
    if (totSum % 2 == 1)
        return false;
    else {
        int k = totSum / 2;
        vector<bool> prev(k + 1, false);
        prev[0] = true;
        if (arr[0] <= k)
            prev[arr[0]] = true;
        for (int ind = 1; ind < n; ind++) {
            vector<bool> cur(k + 1, false);
            cur[0] = true;
            for (int target = 1; target <= k; target++) {
                bool notTaken = prev[target];
                bool taken = false;
                if (arr[ind] <= target)
                    taken = prev[target - arr[ind]];
                cur[target] = notTaken || taken;
            }
            prev = cur;

```

```

    }
    return prev[k];
}

int main() {
    vector<int> arr = {2, 3, 3, 3, 4, 5};
    int n = arr.size();
    if (canPartition(n, arr))
        cout << "The Array can be partitioned into
two equal subsets";
    else
        cout << "The Array cannot be partitioned into
two equal subsets";
    return 0;
}

```

Time Complexity: $O(N \cdot K)$, there are total $N \cdot K$ states, where N is the length of array and K is the target sum i.e. Half of the sum of all elements of the array.
Space Complexity: $O(N)$, we use two 1D arrays to store value of previous row and current row.

Partition Set Into 2 Subsets With Min Absolute Sum Diff (DP- 16)

Memoization

```

#include <bits/stdc++.h>

using namespace std;

bool subsetSumUtil(int ind, int target,
vector<int>& arr, vector<vector<int>>& dp) {
    if (target == 0)
        return dp[ind][target] = true;
    if (ind == 0)
        return dp[ind][target] = (arr[0] == target);
    if (dp[ind][target] != -1)
        return dp[ind][target];
    bool notTaken = subsetSumUtil(ind - 1, target,
arr, dp);
    bool taken = false;
    if (arr[ind] <= target)

```

```

        taken = subsetSumUtil(ind - 1, target -
arr[ind], arr, dp);
        return dp[ind][target] = notTaken || taken;
    }
}

int minSubsetSumDifference(vector<int>& arr, int
n) {
    int totSum = 0;
    for (int i = 0; i < n; i++) {
        totSum += arr[i];
    }
    vector<vector<int>> dp(n, vector<int>(totSum +
1, -1));
    for (int i = 0; i <= totSum; i++) {
        bool dummy = subsetSumUtil(n - 1, i, arr, dp);
    }
    int mini = 1e9;
    for (int i = 0; i <= totSum; i++) {
        if (dp[n - 1][i] == true) {
            int diff = abs(i - (totSum - i));
            mini = min(mini, diff);
        }
    }
    return mini;
}

```

```

int main() {
    vector<int> arr = {1, 2, 3, 4};
    int n = arr.size();
    cout << "The minimum absolute difference is: "
<< minSubsetSumDifference(arr, n);
    return 0;
}

```

Time Complexity: $O(N \cdot K)$, there are total $N \cdot K$ states, where N is the length of array and K is the total sum of the array.

Space Complexity: $O(N \cdot K) + O(N)$, we use a memo table to avoid recomputation. Extra auxiliary stack space is used for recursion.

Tabulation

```

#include <bits/stdc++.h>

using namespace std;

```

```

class Solution {
public:
    int minSubsetSumDifference(vector<int>& arr,
int n) {
        int totSum = 0;
        for (int i = 0; i < n; i++) {
            totSum += arr[i];
        }
        vector<vector<bool>>> dp(n,
vector<bool>(totSum + 1, false));
        for (int i = 0; i < n; i++) {
            dp[i][0] = true;
        }
        if (arr[0] <= totSum)
            dp[0][arr[0]] = true;
        for (int ind = 1; ind < n; ind++) {
            for (int target = 1; target <= totSum;
target++) {
                bool notTaken = dp[ind - 1][target]
                bool taken = false;
                if (arr[ind] <= target)
                    taken = dp[ind - 1][target - arr[ind]];
                dp[ind][target] = notTaken || taken;
            }
        }
        int mini = 1e9;
        for (int i = 0; i <= totSum; i++) {
            if (dp[n - 1][i] == true) {
                int diff = abs(i - (totSum - i));
                mini = min(mini, diff);
            }
        }
        return mini;
    }
};

int main() {
    vector<int> arr = {1, 2, 3, 4};

```

```

        int n = arr.size();
        Solution sol;
        cout << "The minimum absolute difference is: "
<< sol.minSubsetSumDifference(arr, n);
        return 0;
    }

```

Time Complexity: $O(N \cdot K)$, there are total $N \cdot K$ states, where N is the length of array and K is the total sum of the array.

Space Complexity: $O(N \cdot K)$, we use a 2D DP array to avoid recomputation.

Space Optimization

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int minSubsetSumDifference(vector<int>& arr,
int n) {
        int totSum = 0;
        for (int i = 0; i < n; i++) {
            totSum += arr[i];
        }
        vector<bool> prev(totSum + 1, false);
        prev[0] = true;
        prev[arr[0]] = true;
        for (int ind = 1; ind < n; ind++) {
            vector<bool> cur(totSum + 1, false);
            cur[0] = true;
            for (int target = 1; target <= totSum;
target++) {
                bool notTaken = prev[target];
                bool taken = false;
                if (arr[ind] <= target)
                    taken = prev[target - arr[ind]];
                cur[target] = notTaken || taken;
            }
            prev = cur;
        }
        int mini = INT_MAX;

```

```

    for (int i = 0; i <= totSum; i++) {
        if (prev[i]) {
            int diff = abs(i - (totSum - i));
            mini = min(mini, diff);
        }
    }
    return mini;
}

};

int main() {
    vector<int> arr = {1, 2, 3, 4};
    int n = arr.size();
    Solution sol;

    cout << "The minimum absolute difference is: "
    << sol.minSubsetSumDifference(arr, n) << "\n";

    return 0;
}

```

Time Complexity: $O(N \cdot K)$, there are total $N \cdot K$ states, where N is the length of array and K is the total sum of the array.

Space Complexity: $O(N)$, we use two 1D arrays to store value of previous row and current row.

Count Subsets with Sum K (DP - 17)

Memoization

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int countSubsets(vector<int>& nums, int target)
    {
        vector<vector<int>> dp(nums.size(),
        vector<int>(target + 1, -1));

        return solve(nums.size() - 1, target, nums, dp);
    }

private:
    int solve(int index, int target, vector<int>&
    nums, vector<vector<int>>& dp) {

```

```

        if (target == 0) return 1;

        if (index == 0) return (nums[0] == target ? 1 :
        0);

        if (dp[index][target] != -1) return
        dp[index][target];

        int notTake = solve(index - 1, target, nums,
        dp);

        int take = 0;

        if (nums[index] <= target) {
            take = solve(index - 1, target - nums[index],
            nums, dp);
        }

        return dp[index][target] = take + notTake;
    }
};

int main() {
    vector<int> nums = {1, 2, 3, 3};
    int target = 6;
    Solution obj;

    cout << obj.countSubsets(nums, target) << endl;

    return 0;
}

```

Time Complexity: $O(N \cdot K)$, each state defined by index and target is computed once.

Space Complexity: $O(N \cdot K)$, extra space is used for the dp table and recursion stack.

Tabulation

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int countSubsets(vector<int>& arr, int K) {
        int n = arr.size();

        vector<vector<int>> dp(n, vector<int>(K + 1,
        0));

        dp[0][0] = 1;

        if (arr[0] <= K) dp[0][arr[0]] = 1;

        for (int i = 1; i < n; i++) {
            for (int target = 0; target <= K; target++) {

```

```

        int notTake = dp[i - 1][target];
        int take = 0;
        if (arr[i] <= target) take = dp[i - 1][target
- arr[i]];
        dp[i][target] = notTake + take;
    }
}
return dp[n - 1][K];
}
};

int main() {
    Solution obj;
    vector<int> arr = {1, 2, 3, 3};
    int K = 6;
    cout << obj.countSubsets(arr, K) << endl;
    return 0;
}

```

Time Complexity: $O(N * K)$, each state defined by index and target is computed once.
Space Complexity: $O(N * K)$, extra space is used for the dp table.

Space Optimization

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int countSubsets(vector<int>& arr, int K) {
        vector<int> dp(K + 1, 0);
        dp[0] = 1;
        if (arr[0] <= K) dp[arr[0]] += 1;
        for (int i = 1; i < arr.size(); i++) {
            vector<int> curr(K + 1, 0);
            curr[0] = 1;
            for (int t = 0; t <= K; t++) {
                int notTake = dp[t];
                int take = 0;
                if (arr[i] <= t) {

```

```

                    take = dp[t - arr[i]];
                }
                curr[t] = take + notTake;
            }
            dp = curr;
        }
        return dp[K];
    }
};

int main() {
    Solution sol;
    vector<int> arr = {1, 2, 3};
    int K = 4;
    cout << sol.countSubsets(arr, K) << endl;
    return 0;
}

```

Time Complexity: $O(N * K)$, each state defined by index and target is computed once.
Space Complexity: $O(K)$, extra space is used for storing the dp array.

Count Partitions with Given Difference (DP - 18)

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int countPartitions(vector<int>& arr, int d) {
        int totalSum = accumulate(arr.begin(),
arr.end(), 0);
        if (((totalSum + d) % 2 != 0 || d > totalSum)
return 0;
        int K = (totalSum + d) / 2;
        vector<int> dp(K + 1, 0);
        dp[0] = 1;
        if (arr[0] <= K) dp[arr[0]] += 1;
        for (int i = 1; i < arr.size(); i++) {
            vector<int> curr(K + 1, 0);

```

```

curr[0] = 1;

for (int t = 0; t <= K; t++) {
    int notTake = dp[t];
    int take = 0;
    if (arr[i] <= t) {
        take = dp[t - arr[i]];
    }
    curr[t] = take + notTake;
}
dp = curr;
return dp[K];
}
};

int main() {
    Solution sol;
    vector<int> arr = {1, 2, 3, 4};
    int d = 1;
    cout << sol.countPartitions(arr, d) << endl;
    return 0;
}

```

Time Complexity: $O(N * K)$, each state defined by index and target is computed once.

Space Complexity: $O(K)$, extra space is used for storing the dp array.

Assign Cookies

Memoization

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int findContentChildren(vector<int>& student,
        vector<int>& cookie) {
        sort(student.begin(), student.end());
        sort(cookie.begin(), cookie.end());
    }
}

```

```

vector<vector<int>>> memo(student.size(),
    vector<int>(cookie.size(), -1));

return helper(0, 0, student, cookie, memo);
}

private:
    int helper(int studentIndex, int cookieIndex,
        vector<int>& student, vector<int>& cookie,
        vector<vector<int>>>& memo) {
        if (studentIndex >= student.size() ||
            cookieIndex >= cookie.size())
            return 0;
        if (memo[studentIndex][cookieIndex] != -1)
            return memo[studentIndex][cookieIndex];
        int result = 0;
        if (cookie[cookieIndex] >=
            student[studentIndex]) {
            result = max(result, 1 + helper(studentIndex
                + 1, cookieIndex + 1, student, cookie, memo));
        }
        result = max(result, helper(studentIndex,
            cookieIndex + 1, student, cookie, memo));
        return memo[studentIndex][cookieIndex] =
            result;
    }
};

int main() {
    vector<int> student = {1, 2, 3};
    vector<int> cookie = {1, 1};

    Solution solver;

    int result = solver.findContentChildren(student,
        cookie);

    cout << "Maximum number of content students:
    " << result << endl;

    return 0;
}

```

Time Complexity: $O(n*m)$, every pair of student and cookie is checked exactly once.

Space Complexity: $O(n*m) + O(n+m)$, A 2D memoization table is used to store result of subproblems and an additional $O(n+m)$ stack space is used.

Tabulation

```

#include <bits/stdc++.h>

```

```

using namespace std;

class Solution {
public:
    int findContentChildren(vector<int>& student,
vector<int>& cookie) {
        int n = student.size();
        int m = cookie.size();

        sort(student.begin(), student.end());
        sort(cookie.begin(), cookie.end());

        vector<vector<int>> dp(n + 1, vector<int>(m
+ 1, 0));
        for (int i = n - 1; i >= 0; --i) {
            for (int j = m - 1; j >= 0; --j) {
                // Skip current cookie
                int skip = dp[i][j + 1];
                int take = 0;
                if (cookie[j] >= student[i]) {
                    take = 1 + dp[i + 1][j + 1];
                }
                dp[i][j] = max(skip, take);
            }
        }
        return dp[0][0];
    }
};

int main() {
    vector<int> student = {1, 2};
    vector<int> cookie = {1, 2, 3};
    Solution solver;

    int result = solver.findContentChildren(student,
cookie);

    cout << "Maximum number of content students:
" << result << endl;

    return 0;
}

```

Time Complexity: $O(n*m)$, every pair of student and cookie is checked exactly once.
Space Complexity: $O(n*m)$, A 2D memoization table is used to store result of subproblems.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int findContentChildren(vector<int>& student,
vector<int>& cookie) {
        sort(student.begin(), student.end());
        sort(cookie.begin(), cookie.end());

        int studentIndex = 0;
        int cookieIndex = 0;

        while (studentIndex < student.size() &&
cookieIndex < cookie.size()) {
            if (cookie[cookieIndex] >=
student[studentIndex]) {
                studentIndex++;
                cookieIndex++;
            }
            return studentIndex;
        }
    };

    int main() {
        vector<int> student = {1, 2, 3};
        vector<int> cookie = {1, 1};

        Solution solver;

        int result = solver.findContentChildren(student,
cookie);

        cout << "Maximum number of content students:
" << result << endl;

        return 0;
    }
}

```

Time Complexity: $O(n*\log n + m*\log m)$, Both the arrays are sorted in increasing order.
Space Complexity: $O(1)$, No extra space is used.

Minimum Coins (DP - 20)

Memoization Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int coinChange(vector<int>& coins, int amount)
    {
        vector<int> dp(amount + 1, -2);

        return helper(coins, amount, dp);
    }

private:
    int helper(vector<int>& coins, int rem,
vector<int>& dp) {
        if (rem == 0) return 0;
        if (rem < 0) return -1;
        if (dp[rem] != -2) return dp[rem];
        int mini = INT_MAX;
        for (int coin : coins) {
            int res = helper(coins, rem - coin, dp);
            if (res >= 0 && res < mini)
                mini = 1 + res;
        }
        dp[rem] = (mini == INT_MAX) ? -1 : mini;
        return dp[rem];
    }
};

int main() {
    vector<int> coins = {1, 2, 5};
    int amount = 11;

    Solution obj;

    cout << obj.coinChange(coins, amount) << endl;

    return 0;
}

```

Time Complexity: $O(N \cdot T)$, there are total of $N \cdot T$ states.

Space Complexity: $O(N \cdot T) + O(N)$, additional space used to for memo table and recursion stack.

Tabulation Approach

```

#include <bits/stdc++.h>

```

```

using namespace std;

class Solution {
public:
    int coinChange(vector<int>& coins, int amount)
    {
        vector<int> dp(amount + 1, INT_MAX);
        dp[0] = 0;
        for (int i = 1; i <= amount; i++) {
            for (int coin : coins) {
                if (i - coin >= 0 && dp[i - coin] !=
INT_MAX) {
                    dp[i] = min(dp[i], 1 + dp[i - coin]);
                }
            }
        }
        return dp[amount] == INT_MAX ? -1 :
dp[amount];
    }
};

int main() {
    vector<int> coins = {1, 2, 5};
    int amount = 11;

    Solution obj;

    cout << obj.coinChange(coins, amount) << endl;

    return 0;
}

```

Time Complexity: $O(N \cdot T)$, there are total of $N \cdot T$ states.

Space Complexity: $O(N \cdot T)$, additional space used to for memo table.

Space Optimization Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int minimumElements(vector<int>& arr, int T) {
        int n = arr.size();
        vector<int> prev(T + 1, 0);

```

```

vector<int> cur(T + 1, 0);
for (int i = 0; i <= T; i++) {
    if (i % arr[0] == 0)
        prev[i] = i / arr[0];
    else
        prev[i] = 1e9;
}
for (int ind = 1; ind < n; ind++) {
    for (int target = 0; target <= T; target++) {
        int notTake = prev[target];
        int take = 1e9;
        if (arr[ind] <= target)
            take = 1 + cur[target - arr[ind]];
        cur[target] = min(notTake, take);
    }
    prev = cur;
}
int ans = prev[T];
if (ans >= 1e9)
    return -1;
return ans;
}
};

int main() {
    vector<int> arr = {1, 2, 3};
    int T = 7;
    Solution sol;
    int result = sol.minimumElements(arr, T);
    cout << "The minimum number of coins required
to form the target sum is "
    << result << endl;
    return 0;
}

Time Complexity:  $O(N \cdot T)$ , there are total of  $N \cdot T$ 
states.
Space Complexity:  $O(T)$ , additional space used for
storing rows.

```

Target Sum (DP - 21)

Memorization Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int
target) {
        int totalSum = accumulate(nums.begin(),
nums.end(), 0);
        if ((totalSum - target) < 0 || (totalSum - target)
% 2 != 0)
            return 0;
        int subsetSum = (totalSum - target) / 2;
        vector<vector<int>> dp(nums.size(),
vector<int>(subsetSum + 1, -1));
        return countSubsets(nums, nums.size() - 1,
subsetSum, dp);
    }
private:
    int countSubsets(vector<int>& nums, int ind, int
target, vector<vector<int>>& dp) {
        if (ind == 0) {
            if (target == 0 && nums[0] == 0) return 2;
            // {pick or not pick 0}
            if (target == 0 || target == nums[0]) return 1;
            // Either pick or skip
            return 0;
        }
        if (dp[ind][target] != -1) return dp[ind][target];
        int notPick = countSubsets(nums, ind - 1,
target, dp);
        int pick = 0;
        if (nums[ind] <= target)
            pick = countSubsets(nums, ind - 1, target -
nums[ind], dp);
        return dp[ind][target] = pick + notPick;
    }
};

```

```
int main() {
    Solution sol;

    vector<int> nums = {1,1,1,1,1};

    int target = 3;

    cout << sol.findTargetSumWays(nums, target)
    << endl; // Output: 5

    return 0;
}
```

Time Complexity: $O(N \cdot K)$, There are $N \cdot K$ states therefore at max ' $N \cdot K$ ' new problems will be solved.

Space Complexity: $O(N \cdot K) + O(N)$, We are using a recursion stack space($O(N)$) and a 2D array ($O(N \cdot K)$).

Tabulation Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int
    target) {
        int n = nums.size();

        int totalSum = accumulate(nums.begin(),
        nums.end(), 0);

        if ((totalSum + target) % 2 != 0 || abs(target) >
        totalSum) return 0;

        int newTarget = (totalSum + target) / 2;

        vector<vector<int>> dp(n + 1,
        vector<int>(newTarget + 1, 0));

        for (int i = 0; i <= n; i++) dp[i][0] = 1;
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j <= newTarget; j++) {
                dp[i][j] = dp[i - 1][j];
                if (nums[i - 1] <= j) {
                    dp[i][j] += dp[i - 1][j - nums[i - 1]];
                }
            }
        }

        return dp[n][newTarget];
    }
};
```

```
int main() {
    Solution sol;

    vector<int> nums = {1, 1, 1, 1, 1};

    int target = 3;

    cout << sol.findTargetSumWays(nums, target)
    << endl; // Output: 5

    return 0;
}
```

Time Complexity: $O(N \cdot K)$, There are two nested loops

Space Complexity: $O(N \cdot K)$, We are using an external array of size ' $N \cdot K$ '. Stack Space is eliminated.

Space Optimization Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int
    target) {
        int total = accumulate(nums.begin(),
        nums.end(), 0);

        if ((total + target) % 2 != 0 || abs(target) >
        total) return 0;

        int newTarget = (total + target) / 2;

        vector<int> dp(newTarget + 1, 0);

        dp[0] = 1;

        for (int num : nums) {
            for (int j = newTarget; j >= num; j--) {
                dp[j] += dp[j - num];
            }
        }

        return dp[newTarget];
    }
};

int main() {
    Solution sol;

    vector<int> nums = {1,1,1,1,1};

    int target = 3;
```

```

        cout << sol.findTargetSumWays(nums, target)
        << endl;

        return 0;
    }

```

Time Complexity: $O(N \times K)$, There are three nested loops

Space Complexity: $O(K)$, We are using an external array of size 'K+1' to store only one row.

Coin Change 2 (DP - 22)

Memoization Approach

```

#include <bits/stdc++.h>

using namespace std;

long countWaysToMakeChangeUtil(vector<int>& arr, int ind, int T, vector<vector<long>>& dp) {

    if (ind == 0) {

        return (T % arr[0] == 0);

    }

    if (dp[ind][T] != -1)

        return dp[ind][T];

    long notTaken =

countWaysToMakeChangeUtil(arr, ind - 1, T, dp);

    long taken = 0;

    if (arr[ind] <= T)

        taken = countWaysToMakeChangeUtil(arr,

ind, T - arr[ind], dp);

    return dp[ind][T] = notTaken + taken;

}

long countWaysToMakeChange(vector<int>& arr,

int n, int T) {

    vector<vector<long>> dp(n, vector<long>(T + 1,

-1));

    return countWaysToMakeChangeUtil(arr, n - 1,

T, dp);

}

int main() {

    vector<int> arr = {1, 2, 3};

    int target = 4;

    int n = arr.size();

```

```

        cout << "The total number of ways is " <<

countWaysToMakeChange(arr, n, target) << endl;

        return 0;

    }

```

Time Complexity: $O(N \times \text{Target})$, since each state (ind, T) is computed once.

Space Complexity: $O(N \times \text{Target})$ for the DP table, plus $O(\text{Target})$ recursion stack in the worst case.

Tabulation Approach

```

#include <bits/stdc++.h>

using namespace std;

long countWaysToMakeChange(vector<int>& arr,

int n, int T) {

    vector<vector<long>> dp(n, vector<long>(T + 1,

0)); // Create a DP table

    for (int i = 0; i <= T; i++) {

        if (i % arr[0] == 0)

            dp[0][i] = 1;

    }

    for (int ind = 1; ind < n; ind++) {

        for (int target = 0; target <= T; target++) {

            long notTaken = dp[ind - 1][target];

            long taken = 0;

            if (arr[ind] <= target)

                taken = dp[ind][target - arr[ind]];

            dp[ind][target] = notTaken + taken;

        }

    }

    return dp[n - 1][T];

}

int main() {

    vector<int> arr = {1, 2, 3};

    int target = 4;

    int n = arr.size();

    cout << "The total number of ways is " <<

countWaysToMakeChange(arr, n, target) << endl;

    return 0; // Return 0 to indicate successful

program execution

}

```

Time Complexity: $O(N*T)$, as There are two nested loops.

Space Complexity: $O(N*T)$, as We are using an external array of size ' $N*T$ '. Stack Space is eliminated.

Space Optimization Approach

```
#include <bits/stdc++.h>

using namespace std;

long countWaysToMakeChange(vector<int>& arr,
int n, int T) {
    vector<long> prev(T + 1, 0);
    for (int i = 0; i <= T; i++) {
        if (i % arr[0] == 0)
            prev[i] = 1;
    }
    for (int ind = 1; ind < n; ind++) {
        vector<long> cur(T + 1, 0);
        for (int target = 0; target <= T; target++) {
            long notTaken = prev[target];
            long taken = 0;
            if (arr[ind] <= target)
                taken = cur[target - arr[ind]];
            cur[target] = notTaken + taken;
        }
        prev = cur;
    }
    return prev[T];
}

int main() {
    vector<int> arr = {1, 2, 3};
    int target = 4;
    int n = arr.size();

    cout << "The total number of ways is " <<
countWaysToMakeChange(arr, n, target) << endl;

    return 0;
}
```

Time Complexity: $O(N*T)$, as There are two nested loops.

Space Complexity: $O(T)$, as We are using an external array of size ' $T+1$ ' to store two rows only

Unbounded Knapsack (DP-23)

Memorization Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int knapsackUtil(vector<int>& wt, vector<int>&
val, int ind, int W, vector<vector<int>>& dp) {
        if (ind == 0) {
            return (W / wt[0]) * val[0];
        }
        if (dp[ind][W] != -1)
            return dp[ind][W];
        int notTaken = knapsackUtil(wt, val, ind - 1,
W, dp);
        int taken = INT_MIN;
        if (wt[ind] <= W)
            taken = val[ind] + knapsackUtil(wt, val,
ind, W - wt[ind], dp);
        return dp[ind][W] = max(notTaken, taken);
    }

    int unboundedKnapsack(int n, int W,
vector<int>& val, vector<int>& wt) {
        vector<vector<int>> dp(n, vector<int>(W + 1,
-1));
        return knapsackUtil(wt, val, n - 1, W, dp);
    }
};

int main() {
    vector<int> wt = {2, 4, 6};
    vector<int> val = {5, 11, 13};
    int W = 10;
    int n = wt.size();

    Solution obj;

    cout << "The Maximum value of items the thief
can steal is "
        << obj.unboundedKnapsack(n, W, val, wt) <<
endl;
```

```

    return 0;
}

```

Time Complexity: $O(N*W)$, There are $N*W$ states therefore at max ' $N*W$ ' new problems will be solved.

Space Complexity: $O(N*W) + O(N)$, We are using a recursion stack space ($O(N)$) and a 2D array ($O(N*W)$).

Tabulation Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int unboundedKnapsack(int n, int W,
        vector<int>& val, vector<int>& wt) {
        vector<vector<int>> dp(n, vector<int>(W + 1,
0));
        for (int i = wt[0]; i <= W; i++) {
            dp[0][i] = (i / wt[0]) * val[0];
        }
        for (int ind = 1; ind < n; ind++) {
            for (int cap = 0; cap <= W; cap++) {
                int notTaken = dp[ind - 1][cap];
                int taken = INT_MIN;
                if (wt[ind] <= cap)
                    taken = val[ind] + dp[ind][cap -
wt[ind]];
                dp[ind][cap] = max(notTaken, taken);
            }
        }
        return dp[n - 1][W];
    }
};

int main() {
    vector<int> wt = {2, 4, 6};
    vector<int> val = {5, 11, 13};
    int W = 10;
    int n = wt.size();
    Solution obj;

```

```

    cout << "The Maximum value of items the thief
can steal is "

```

```

    << obj.unboundedKnapsack(n, W, val, wt) <<
endl;

```

```

    return 0;
}

```

Time Complexity: $O(N*W)$, There are two nested loops

Space Complexity: $O(N*W)$, We are using an external array of size ' $N*W$ '. Stack Space is eliminated.

Space Optimization Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int unboundedKnapsack(int n, int W,
        vector<int>& val, vector<int>& wt) {
        vector<int> cur(W + 1, 0);
        for (int i = wt[0]; i <= W; i++) {
            cur[i] = (i / wt[0]) * val[0];
        }
        for (int ind = 1; ind < n; ind++) {
            for (int cap = 0; cap <= W; cap++) {
                int notTaken = cur[cap];
                int taken = INT_MIN;
                if (wt[ind] <= cap) {
                    taken = val[ind] + cur[cap - wt[ind]];
                }
                cur[cap] = max(notTaken, taken);
            }
        }
        return cur[W];
    }
};

int main() {
    vector<int> wt = {2, 4, 6};
    vector<int> val = {5, 11, 13};
    int W = 10;

```

```

int n = wt.size();

Solution obj;

cout << "The Maximum value of items the thief
can steal is "

    << obj.unboundedKnapsack(n, W, val, wt) <<
endl;

return 0;
}

```

Time Complexity: $O(N*W)$, There are two nested loops.

Space Complexity: $O(W)$, We are using an external array of size 'W+1' to store only one row.

Rod Cutting Problem | (DP - 24)

Memoization Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int rodCutting(vector<int> price, int n) {
        vector<vector<int>> dp(n, vector<int>(n + 1,
0));
        for(int length = 0; length <= n; length++) {
            dp[0][length] = length * price[0];
        }
        for(int i = 1; i < n; i++) {
            for(int length = 0; length <= n; length++) {
                int notTake = dp[i - 1][length];
                int take = INT_MIN;
                int rodLength = i + 1;
                if(rodLength <= length) {
                    take = price[i] + dp[i][length -
rodLength];
                }
                dp[i][length] = max(take, notTake);
            }
        }
        return dp[n - 1][n];
    }
}

```

```

};

int main() {
    int n = 8;
    vector<int> price = {1, 5, 8, 9, 10, 17, 17, 20};
    Solution obj;
    int maxValue = obj.rodCutting(price, n);

    cout << "The maximum obtainable value is: " <<
maxValue << endl;

    return 0;
}

```

Time Complexity: $O(n \times n)$ Each subproblem (i, length) is computed once.

Space Complexity: $O(n \times n)$, We use a 2D DP table for memoization.

Tabulation Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int rodCutting(vector<int>& price, int n) {
        vector<vector<int>> dp(n, vector<int>(n + 1,
0));
        for(int length = 0; length <= n; length++){
            dp[0][length] = price[0]*length;
        }
        for (int ind = 1; ind < n; ++ind) {
            for (int length = 1; length <= n; ++length) {
                int notTaken = 0+dp[ind - 1][length];
                int taken = INT_MIN;
                int rodLength = ind + 1;
                if (rodLength <= length) {
                    taken = price[ind] + dp[ind][length -
rodLength];
                }
                dp[ind][length] = max(notTaken, taken);
            }
        }
        return dp[n - 1][n];
    }
}

```

```

    }
};

int main() {
    vector<int> price = {2, 4, 6, 8};

    int n = price.size();

    Solution sol;

    cout << "The Maximum value is " <<
    sol.rodCutting(price, n) << endl;

    return 0;
}

```

Time Complexity: $O(N * W)$, Because we have a nested loop iterating through all 'n' items and all 'W' capacities.

Space Complexity: $O(N * W)$, We are using a 2D DP table of size 'n' by 'W+1' to store intermediate results, and we eliminated recursion stack space.

Space Optimization Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int rodCutting(vector<int>& price, int n) {
        vector<int> prev(n+1, 0), cur(n+1, 0);

        for(int length = 0; length <= n; length++){
            prev[length] = price[0]*length;
        }

        for (int ind = 1; ind < n; ++ind) {
            for (int length = 1; length <= n; ++length) {
                int notTaken = 0+prev[length];

                int taken = INT_MIN;

                int rodLength = ind + 1;

                if (rodLength <= length) {
                    taken = price[ind] + cur[length -
                    rodLength];
                }

                cur[length] = max(notTaken, taken);
            }

            prev = cur;
        }
    }
};

```

```

        return prev[n];
    }
};

int main() {
    vector<int> price = {2, 4, 6, 8};

    int n = price.size();

    Solution sol;

    cout << "The Maximum value is " <<
    sol.rodCutting(price, n) << endl;

    return 0;
}

```

Time Complexity: $O(N * W)$, We iterate through N items, and for each item we process W weight capacities.

Space Complexity: $O(W)$, We only use a 1D array of size W to store intermediate results, eliminating the need for 2D DP or recursion stack.

Longest Common Subsequence | (DP - 25)

Memoization Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution{
private:
    int func(string& s1, string& s2, int ind1, int ind2,
    vector<vector<int>>& dp) {
        if (ind1 < 0 || ind2 < 0)
            return 0;

        if (dp[ind1][ind2] != -1)
            return dp[ind1][ind2];

        if (s1[ind1] == s2[ind2])
            return dp[ind1][ind2] = 1 + func(s1, s2,
            ind1 - 1, ind2 - 1, dp);

        else
            return dp[ind1][ind2] = max(func(s1, s2,
            ind1, ind2 - 1, dp), func(s1, s2, ind1 - 1, ind2, dp));
    }

public:

```

```

int lcs(string str1, string str2) {
    int n = str1.size();
    int m = str2.size();
    vector<vector<int>> dp(n, vector<int>(m, -1));
    return func(str1, str2, n - 1, m - 1, dp);
}

int main() {
    string s1 = "acd";
    string s2 = "ced";
    Solution sol;

    cout << "The Length of Longest Common Subsequence is " << sol.lcs(s1, s2) << endl;

    return 0;
}

```

Time Complexity: $O(n * m)$, where n is the length of $str1$ and m is the length of $str2$. This is because we are using a 2D DP array to store results for all pairs of indices.
 Space Complexity: $O(N * M) + O(N + M)$, We are using an auxiliary recursion stack space($O(N + M)$) and a 2D array ($O(N * M)$).

Tabulation Approach

```

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

class Solution{
public:
    int lcs(string str1, string str2) {
        int n = str1.size();
        int m = str2.size();

        vector<vector<int>> dp(n + 1, vector<int>(m + 1, -1));

        for (int i = 0; i <= n; i++) {
            dp[i][0] = 0;
        }

        for (int i = 0; i <= m; i++) {
            dp[0][i] = 0;
        }

        for (int ind1 = 1; ind1 <= n; ind1++) {

```

```

            for (int ind2 = 1; ind2 <= m; ind2++) {
                if (str1[ind1 - 1] == str2[ind2 - 1])
                    dp[ind1][ind2] = 1 + dp[ind1 - 1][ind2 - 1];
                else
                    dp[ind1][ind2] = max(dp[ind1 - 1][ind2], dp[ind1][ind2 - 1]);
            }
        }

        return dp[n][m];
    }
};

int main() {
    string s1 = "acd";
    string s2 = "ced";
    Solution sol;

    cout << "The Length of Longest Common Subsequence is " << sol.lcs(s1, s2) << endl;

    return 0;
}

```

Time Complexity: $O(n * m)$, where n is the length of $str1$ and m is the length of $str2$. This is because we are filling a 2D DP array of size $(n+1) \times (m+1)$.

Space Complexity: $O(n * m)$, as we are using a 2D DP array to store results for all pairs of indices.

Space Optimized Approach

```

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

class Solution{
public:
    int lcs(string str1, string str2) {
        int n = str1.size();
        int m = str2.size();

        vector<int> prev(m + 1, 0), cur(m + 1, 0);

        for (int ind1 = 1; ind1 <= n; ind1++) {
            for (int ind2 = 1; ind2 <= m; ind2++) {
                if (str1[ind1 - 1] == str2[ind2 - 1])
                    cur[ind2] = 1 + prev[ind2 - 1];
                else

```

```

        cur[ind2] = max(prev[ind2], cur[ind2 -
1]);
    }
    prev = cur;
}
return prev[m];
}
};

int main() {
    string s1 = "acd";
    string s2 = "ced";

    Solution sol;

    cout << "The Length of Longest Common
Subsequence is " << sol.lcs(s1, s2) << endl;

    return 0;
}

```

Time Complexity: $O(n * m)$, where n is the length of $str1$ and m is the length of $str2$. This is because we are still filling a 2D DP table, but now using only two rows.

Space Complexity: $O(m)$, where m is the length of $str2$. We are using only two arrays of size $m+1$ to store the current and previous rows of the DP table.

Print Longest Common Subsequence | (DP - 26)

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    string longestCommonSubsequence(string
&text1, string &text2) {
        int n = text1.size();
        int m = text2.size();

        vector<vector<int>>> dp(n + 1, vector<int>(m
+ 1, 0));

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                if (text1[i - 1] == text2[j - 1]) {
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {

```

```

                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }

        int i = n, j = m;
        string lcs = "";
        while (i > 0 && j > 0) {
            if (text1[i - 1] == text2[j - 1]) {
                lcs += text1[i - 1];
                i--;
                j--;
            } else if (dp[i - 1][j] > dp[i][j - 1]) {
                i--;
            } else {
                j--;
            }
        }

        reverse(lcs.begin(), lcs.end());

        return lcs;
    }
};

int main() {
    string s1 = "abcde";
    string s2 = "ace";

    Solution sol;

    cout << "LCS: " <<
sol.longestCommonSubsequence(s1, s2) << endl;

    return 0;
}

```

Time Complexity: $O(N * M) + O(N + M)$, we first build the entire DP table and then traverse it to reconstruct the LCS.

Space Complexity: $O(N * M)$, space used to store DP table.

Longest Common Substring | (DP - 27)

Tabulation Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int longestCommonSubstr(string str1, string str2)
    {
        int n = str1.size();
        int m = str2.size();

        vector<vector<int>>> dp(n + 1, vector<int>(m + 1, 0));

        int ans = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                if (str1[i - 1] == str2[j - 1]) {
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                    ans = max(ans, dp[i][j]);
                } else {
                    dp[i][j] = 0;
                }
            }
        }

        return ans;
    }
};

int main() {
    string s1 = "abcjklp";
    string s2 = "acjkp";

    Solution sol;

    cout << "The Length of Longest Common Substring is " << sol.longestCommonSubstr(s1, s2) << endl;

    return 0;
}

```

Time Complexity: $O(n * m)$, where n is the length of $str1$ and m is the length of $str2$. This is because we are filling a 2D DP table of size $(n+1) \times (m+1)$.

Space Complexity: $O(n * m)$, where n is the length of $str1$ and m is the length of $str2$. This is due to the storage of the DP table, which requires $O(n * m)$ space.

Space Optimized Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int longestCommonSubstr(string str1, string str2)
    {
        int n = str1.size();
        int m = str2.size();

        vector<int> prev(m+1, 0);
        vector<int> cur(m+1, 0);

        int ans = 0;
        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= m; j++){
                if(str1[i-1] == str2[j-1]){
                    int val = 1 + prev[j-1];
                    cur[j] = val;
                    ans = max(ans, val);
                }
                else{
                    cur[j] = 0;
                }
            }
            prev = cur;
        }

        return ans;
    }
};

int main() {
    string s1 = "abcjklp";
    string s2 = "acjkp";

    Solution sol;

    cout << "The Length of Longest Common Substring is " << sol.longestCommonSubstr(s1, s2) << endl;

    return 0;
}

```

Time Complexity: $O(n * m)$, where n is the length of `str1` and m is the length of `str2`. This is because we are iterating through both strings and filling a 2D DP table.
Space Complexity: $O(m)$, where m is the length of `str2`. We are using two arrays of size $m+1$ to store the current and previous row values.

Longest Palindromic Subsequence | (DP-28)

Tabulation Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution{
private:
    int func(string s1, string s2) {
        int n = s1.size();
        int m = s2.size();

        vector<vector<int>>> dp(n + 1, vector<int>(m + 1, -1));

        for (int i = 0; i <= n; i++) {
            dp[i][0] = 0;
        }

        for (int i = 0; i <= m; i++) {
            dp[0][i] = 0;
        }

        for (int ind1 = 1; ind1 <= n; ind1++) {
            for (int ind2 = 1; ind2 <= m; ind2++) {
                if (s1[ind1 - 1] == s2[ind2 - 1])
                    dp[ind1][ind2] = 1 + dp[ind1 - 1][ind2 - 1];
                else
                    dp[ind1][ind2] = max(dp[ind1 - 1][ind2], dp[ind1][ind2 - 1]);
            }
        }

        return dp[n][m];
    }

public:
    int longestPalinSubseq(string s){
```

```
        string t = s;

        reverse(s.begin(), s.end());

        return lcs(s, t);
    }
};

int main() {
    string s = "bbabcbcab";

    Solution sol;

    cout << "The Length of Longest Palindromic Subsequence is " << sol.longestPalinSubseq(s);

    return 0;
}
```

Time Complexity: $O(N * M)$, where N is the length of the string and M is the length of its reverse.

Space Complexity: $O(N * M)$ for the DP array used to store the lengths of the longest common subsequences.

Space Optimized Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution{
private:
    int lcs(string s1, string s2) {
        int n = s1.size();
        int m = s2.size();

        vector<int> prev(m + 1, 0), cur(m + 1, 0);

        for (int ind1 = 1; ind1 <= n; ind1++) {
            for (int ind2 = 1; ind2 <= m; ind2++) {
                if (s1[ind1 - 1] == s2[ind2 - 1])
                    cur[ind2] = 1 + prev[ind2 - 1];
                else
                    cur[ind2] = max(prev[ind2], cur[ind2 - 1]);
            }

            prev = cur;
        }

        return prev[m];
    }

public:
```

```

int longestPalinSubseq(string s){
    string t = s;
    reverse(s.begin(), s.end());
    return lcs(s, t);
}
};

int main() {
    string s = "bbabcbcab";
    Solution sol;

    cout << "The Length of Longest Palindromic
    Subsequence is " << sol.longestPalinSubseq(s);

    return 0;
}

```

Time Complexity: $O(N * M)$, where N is the length of the string and M is the length of its reverse.

Space Complexity: $O(M)$ for the two arrays used to store the lengths of the longest common subsequences, where M is the length of the second string.

Minimum insertions to make string palindrome | DP-29

Tabulation Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int lcs(string s1, string s2) {
        int n = s1.size();
        int m = s2.size();
        vector<vector<int>>> dp(n + 1, vector<int>(m + 1, -1));

        for (int i = 0; i <= n; i++) {
            dp[i][0] = 0;
        }

        for (int i = 0; i <= m; i++) {
            dp[0][i] = 0;
        }

        for (int ind1 = 1; ind1 <= n; ind1++) {

```

```

        for (int ind2 = 1; ind2 <= m; ind2++) {
            if (s1[ind1 - 1] == s2[ind2 - 1])
                dp[ind1][ind2] =
                    1 + dp[ind1 - 1][ind2 - 1];
            else
                dp[ind1][ind2] =
                    max(dp[ind1 - 1][ind2],
                        dp[ind1][ind2 - 1]);
        }
    }

    return dp[n][m];
}

int longestPalindromeSubsequence(string s) {
    string t = s;
    reverse(s.begin(), s.end());
    return lcs(s, t);
}

int minInsertion(string s) {
    int n = s.size();
    int k = longestPalindromeSubsequence(s);
    return n - k;
}

};

int main() {
    Solution obj;
    string s = "abcaa";

    cout << "The Minimum insertions required to
    make string palindrome: "
        << obj.minInsertion(s);

    return 0;
}

```

Time Complexity: $O(N*N)$, we fill the entire DP array of size $N*N$ one by one.

Space Complexity: $O(N*N)$, additional space used to store DP array.

Space Optimized Approach

```

#include <bits/stdc++.h>

```

```

using namespace std;

class Solution {
public:
    int lcs(string s1, string s2) {
        int n = s1.size();
        int m = s2.size();
        vector<int> prev(m + 1, 0), cur(m + 1, 0);
        for (int ind1 = 1; ind1 <= n; ind1++) {
            for (int ind2 = 1; ind2 <= m; ind2++) {
                if (s1[ind1 - 1] == s2[ind2 - 1])
                    cur[ind2] = 1 + prev[ind2 - 1];
                else
                    cur[ind2] = max(prev[ind2], cur[ind2 - 1]);
            }
            prev = cur;
        }
        return prev[m];
    }

    int longestPalindromeSubsequence(string s) {
        string t = s;
        reverse(t.begin(), t.end());
        return lcs(s, t);
    }

    int minInsertion(string s) {
        int n = s.size();
        int k = longestPalindromeSubsequence(s);
        return n - k;
    }
};

int main() {
    Solution sol;
    string s = "abcaa";
    cout << "The Minimum insertions required to make string palindrome: "
        << sol.minInsertion(s) << endl;
    return 0;
}

```

```

}

```

Time Complexity: $O(N*N)$, we fill the entire DP array of size $N*N$ one by one.

Space Complexity: $O(N)$, we are using an external array of size ' $N+1$ ' to store only two rows.

Minimum Insertions/Deletions to Convert String | (DP- 30)

Tabulation Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int minOperations(string s1, string s2) {
        int n = s1.length();
        int m = s2.length();
        vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                if (s1[i - 1] == s2[j - 1])
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                else
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
        int lcs = dp[n][m];
        return (n - lcs) + (m - lcs);
    }
};

int main() {
    Solution sol;
    string s1 = "heap", s2 = "pea";
    cout << "Minimum operations: " <<
        sol.minOperations(s1, s2) << endl;
    return 0;
}

```

Time Complexity: $O(N*M)$, we fill the entire DP array of size $N*M$ one by one.
Space Complexity: $O(N*M)$, additional space used to store DP array.

Space Optimized Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int minOperations(string s1, string s2) {
        int n = s1.size(), m = s2.size();
        vector<int> prev(m + 1, 0), cur(m + 1, 0);
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                if (s1[i - 1] == s2[j - 1])
                    cur[j] = 1 + prev[j - 1];
                else
                    cur[j] = max(prev[j], cur[j - 1]);
            }
            prev = cur;
        }
        int lcs = prev[m];
        return (n - lcs) + (m - lcs);
    }
};

int main() {
    string s1 = "heap", s2 = "pea";
    Solution obj;
    cout << obj.minOperations(s1, s2) << endl;
    return 0;
}
```

Time Complexity: $O(N*M)$, we fill the entire DP array of size $N*M$ one by one.
Space Complexity: $O(M)$, we are using an external array of size ' $M+1$ ' to store only two rows.

Shortest Common Supersequence | (DP - 31)

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    string shortestSupersequence(string s1, string s2)
    {
        int n = s1.size();
        int m = s2.size();
        vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
        for (int i = 0; i <= n; i++) {
            dp[i][0] = 0;
        }
        for (int i = 0; i <= m; i++) {
            dp[0][i] = 0;
        }
        for (int ind1 = 1; ind1 <= n; ind1++) {
            for (int ind2 = 1; ind2 <= m; ind2++) {
                if (s1[ind1 - 1] == s2[ind2 - 1])
                    dp[ind1][ind2] = 1 + dp[ind1 - 1][ind2 - 1];
                else
                    dp[ind1][ind2] = max(dp[ind1 - 1][ind2], dp[ind1][ind2 - 1]);
            }
        }
        int i = n;
        int j = m;
        string ans = "";
        while (i > 0 && j > 0) {
            if (s1[i - 1] == s2[j - 1]) {
                ans += s1[i - 1];
                i--;
                j--;
            }
            else if (dp[i - 1][j] > dp[i][j - 1]) {
                ans += s1[i - 1];
            }
        }
    }
};
```

```

        i--;
    } else {
        ans += s2[j - 1];
        j--;
    }
}
while (i > 0) {
    ans += s1[i - 1];
    i--;
}
while (j > 0) {
    ans += s2[j - 1];
    j--;
}
reverse(ans.begin(), ans.end());
return ans;
}
};

int main() {
    string s1 = "brute";
    string s2 = "groot";
    Solution obj;

    cout << "The Shortest Common Supersequence
is " << obj.shortestSupersequence(s1, s2);

    return 0;
}

```

Time Complexity: $O(n * m)$, Filling the DP table takes $O(n * m)$, and backtracking adds $O(n + m)$, which is negligible compared to the main DP.

Space Complexity: $O(n * m)$, A 2D DP array of size $(n+1) \times (m+1)$ is used.

Distinct Subsequences| (DP-32)

Memoization

```

#include <bits/stdc++.h>

using namespace std;

class Solution {

```

```

public:

    int helper(int i, int j, string &s, string &t,
        vector<vector<int>> &dp) {
        if (j == t.size()) return 1;
        if (i == s.size()) return 0;
        if (dp[i][j] != -1) return dp[i][j];
        if (s[i] == t[j]) {
            int take = helper(i + 1, j + 1, s, t, dp);
            int notTake = helper(i + 1, j, s, t, dp);
            return dp[i][j] = take + notTake;
        } else {
            return dp[i][j] = helper(i + 1, j, s, t, dp);
        }
    }

    int numDistinct(string s, string t) {
        vector<vector<int>> dp(s.size(),
            vector<int>(t.size(), -1));

        return helper(0, 0, s, t, dp);
    }
};

int main() {
    Solution sol;

    string s = "babgbag";
    string t = "bag";

    cout << sol.numDistinct(s, t) << endl;

    return 0;
}

```

Time Complexity: $O(N * M)$, each state defined by character of string s and character of string t is computed once.

Space Complexity: $O(N * M) + O(N)$, extra space is used for storing the dp array and recursion stack space.

Tabulation

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:

```

```

int numDistinct(string s, string t) {
    int n = s.size();
    int m = t.size();

    vector<vector<long long>> dp(n + 1,
vector<long long>(m + 1, 0));

    for (int i = 0; i <= n; i++) {
        dp[i][m] = 1;
    }

    for (int i = n - 1; i >= 0; i--) {
        for (int j = m - 1; j >= 0; j--) {
            if (s[i] == t[j]) {
                dp[i][j] = dp[i + 1][j + 1] + dp[i + 1][j];
            } else {
                dp[i][j] = dp[i + 1][j];
            }
        }
    }

    return dp[0][0];
}

};

int main() {
    Solution sol;

    string s = "babgbag";
    string t = "bag";

    cout << sol.numDistinct(s, t) << endl;

    return 0;
}

```

Time Complexity: $O(N * M)$, each state defined by character of string s and character of string t is computed once.

Space Complexity: $O(N * M)$, extra space is used for storing the dp array.

Space Optimization

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:

```

```

int numDistinct(string s, string t) {
    int n = s.size();
    int m = t.size();

    vector<long long> dp(m + 1, 0);
    dp[m] = 1;

    for (int i = n - 1; i >= 0; i--) {
        vector<long long> curr = dp;

        for (int j = m - 1; j >= 0; j--) {
            if (s[i] == t[j]) {
                curr[j] = dp[j + 1] + dp[j];
            }
            else {
                curr[j] = dp[j];
            }
        }

        dp = curr;
    }

    return (int)dp[0];
}

};

int main() {
    Solution sol;

    string s = "rabbbit";
    string t = "rabbit";

    cout << sol.numDistinct(s, t) << endl;

    return 0;
}

```

Time Complexity: $O(N * M)$, each state defined by character of string s and character of string t is computed once.

Space Complexity: $O(M)$, extra space is used for storing the 1D array.

Edit Distance | (DP-33)

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

```

```

int editDistanceUtil(string& S1, string& S2, int i,
int j, vector<vector<int>>& dp) {
    if (i < 0)
        return j + 1;
    if (j < 0)
        return i + 1;
    if (dp[i][j] != -1)
        return dp[i][j];
    if (S1[i] == S2[j])
        return dp[i][j] = 0 + editDistanceUtil(S1, S2, i
- 1, j - 1, dp);
    else
        return dp[i][j] = 1 + min(editDistanceUtil(S1,
S2, i - 1, j - 1, dp),
                                min(editDistanceUtil(S1, S2, i
- 1, j, dp),
                                editDistanceUtil(S1, S2, i, j
- 1, dp)));
}

int editDistance(string& S1, string& S2) {
    int n = S1.size();
    int m = S2.size();
    vector<vector<int>> dp(n, vector<int>(m, -1));
    return editDistanceUtil(S1, S2, n - 1, m - 1, dp);
}

int main() {
    string s1 = "horse";
    string s2 = "ros";
    cout << "The minimum number of operations
required is: " << editDistance(s1, s2);
    return 0;
}

```

Time Complexity: $O(N \times M)$, as There are at most $N \times M$ distinct states in the DP table (one for each pair of indices), so we solve at most that many subproblems.

Space Complexity: $O(N \times M) + O(N + M)$, as $O(N \times M)$ for the 2D dp array used for memoization, and $O(N + M)$ for the recursion call stack in the worst case.

Better Approach

```

#include <bits/stdc++.h>
using namespace std;
int editDistance(string& S1, string& S2) {
    int n = S1.size();
    int m = S2.size();
    vector<vector<int>> dp(n + 1, vector<int>(m +
1, 0));
    for (int i = 0; i <= n; i++) {
        dp[i][0] = i;
    }
    for (int j = 0; j <= m; j++) {
        dp[0][j] = j;
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (S1[i - 1] == S2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1];
            } else {
                dp[i][j] = 1 + min(dp[i - 1][j - 1],
min(dp[i - 1][j], dp[i][j - 1]));
            }
        }
    }
    return dp[n][m];
}

int main() {
    string s1 = "horse";
    string s2 = "ros";
    cout << "The minimum number of operations
required is: " << editDistance(s1, s2);
    return 0;
}

```

Time Complexity: $O(N \times M)$, as There are two nested loops running for N and M respectively, so the total operations are $N \times M$

Space Complexity: $O(N \times M)$, as A 2D DP array of size $N \times M$ is used for memoization. No recursion, so stack space is eliminated.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

int editDistance(string& S1, string& S2) {
    int n = S1.size();
    int m = S2.size();
    vector<int> prev(m + 1, 0);
    vector<int> cur(m + 1, 0);
    for (int j = 0; j <= m; j++) {
        prev[j] = j;
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (S1[i - 1] == S2[j - 1]) {
                cur[j] = prev[j - 1];
            } else {
                cur[j] = 1 + min(prev[j - 1], min(prev[j],
cur[j - 1]));
            }
        }
        prev = cur;
    }
    return cur[m];
}

int main() {
    string s1 = "horse";
    string s2 = "ros";

    cout << "The minimum number of operations
required is: " << editDistance(s1, s2);

    return 0;
}
```

Time Complexity: $O(N \times M)$, as There are two nested loops.

Space Complexity: $O(M)$, as We are using an external array of size 'M+1' to store two rows.

Wildcard Matching | (DP-34)

Memoization Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool isAllStars(string &S1, int i) {
        for (int j = 0; j <= i; j++) {
            if (S1[j] != '*')
                return false;
        }
        return true;
    }

    bool wildcardMatchingUtil(string &S1, string
&S2, int i, int j, vector<vector<int>>& dp) {
        if (i < 0 && j < 0)
            return true;
        if (i < 0 && j >= 0)
            return false;
        if (j < 0 && i >= 0)
            return isAllStars(S1, i);
        if (dp[i][j] != -1)
            return dp[i][j];
        if (S1[i] == S2[j] || S1[i] == '?')
            return dp[i][j] = wildcardMatchingUtil(S1,
S2, i - 1, j - 1, dp);
        if (S1[i] == '*')
            return dp[i][j] = wildcardMatchingUtil(S1,
S2, i - 1, j, dp) ||
wildcardMatchingUtil(S1, S2, i,
j - 1, dp);
        return dp[i][j] = false;
    }

    bool wildcardMatching(string &S1, string &S2)
    {
        int n = S1.size();
```

```

        int m = S2.size();
        vector<vector<int>> dp(n, vector<int>(m, -
1));
        return wildcardMatchingUtil(S1, S2, n - 1, m -
1, dp);
    }
};

int main() {
    string S1 = "ab*cd";
    string S2 = "abdefcd";
    Solution obj;
    if (obj.wildcardMatching(S1, S2))
        cout << "String S1 and S2 do match";
    else
        cout << "String S1 and S2 do not match";
    return 0;
}

```

Time Complexity: $O(N*M)$, There are $N*M$ states therefore at max ' $N*M$ ' new problems will be solved.

Space Complexity: $O(N*M) + O(N+M)$, We are using a recursion stack space($O(N+M)$) and a 2D array ($O(N*M)$).

Tabulation Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool isAllStars(string &S1, int i) {
        for (int j = 1; j <= i; j++) {
            if (S1[j - 1] != '*')
                return false;
        }
        return true;
    }

    bool wildcardMatching(string &S1, string &S2)
    {
        int n = S1.size();
        int m = S2.size();

```

```

        vector<vector<bool>> dp(n + 1,
vector<bool>(m + 1, false));

        dp[0][0] = true;
        for (int j = 1; j <= m; j++) {
            dp[0][j] = false;
        }
        for (int i = 1; i <= n; i++) {
            dp[i][0] = isAllStars(S1, i);
        }
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                if (S1[i - 1] == S2[j - 1] || S1[i - 1] == '?')
                {
                    dp[i][j] = dp[i - 1][j - 1];
                }
                else if (S1[i - 1] == '*') {
                    dp[i][j] = dp[i - 1][j] || dp[i][j - 1];
                }
            }
        }
        return dp[n][m];
    }
};

int main() {
    Solution sol;
    string S1 = "ab*cd";
    string S2 = "abdefcd";
    bool result = sol.wildcardMatching(S1, S2);
    if (result)
        cout << "String S1 and S2 do match";
    else
        cout << "String S1 and S2 do not match";
    return 0;
}

```

Time Complexity: $O(N*M)$, There are two nested loops

Space Complexity: $O(N*M)$, We are using an external array of size ' $N*M$ '. Stack Space is eliminated.

Space Optimization Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool isAllStars(string &S1, int i) {
        for (int j = 1; j <= i; j++) {
            if (S1[j - 1] != '*')
                return false;
        }
        return true;
    }

    bool wildcardMatching(string &S1, string &S2)
    {
        int n = S1.size();
        int m = S2.size();

        vector<bool> prev(m + 1, false);
        vector<bool> cur(m + 1, false);

        prev[0] = true;
        for (int i = 1; i <= n; i++) {
            cur[0] = isAllStars(S1, i);
            for (int j = 1; j <= m; j++) {
                if (S1[i - 1] == S2[j - 1] || S1[i - 1] == '?')
                {
                    cur[j] = prev[j - 1];
                }
                else if (S1[i - 1] == '*') {
                    cur[j] = prev[j] || cur[j - 1];
                }
                else {
                    cur[j] = false;
                }
            }
            prev = cur;
        }
    }
};
```

```
    }
    return prev[m];
}

};

int main() {
    Solution obj;
    string S1 = "ab*cd";
    string S2 = "abdefcd";
    bool result = obj.wildcardMatching(S1, S2);
    if (result)
        cout << "String S1 and S2 do match";
    else
        cout << "String S1 and S2 do not match";
    return 0;
}
```

Time Complexity: $O(N*M)$, There are two nested loops.

Space Complexity: $O(M)$, We are using an external array of size ' $M+1$ ' to store two rows.

Stock Buy and Sell | (DP-35)

Brute Force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int maxProfit = 0;
        for (int i = 0; i < prices.size(); i++) {
            for (int j = i + 1; j < prices.size(); j++) {
                int profit = prices[j] - prices[i];
                maxProfit = max(maxProfit, profit);
            }
        }
        return maxProfit;
    }
};
```

```
int main() {
    Solution obj;

    vector<int> prices = {7, 1, 5, 3, 6, 4};

    cout << obj.maxProfit(prices) << endl; //
Output: 5

    return 0;
}
```

Time Complexity: $O(n^2)$, We iterate over all possible pairs of days using two nested loops. For each day 'i', we check every future day 'j', making the total number of comparisons equal to $n(n-1)/2$, which is quadratic in nature.

Space Complexity: $O(1)$, We use only constant extra space (for variables like 'maxProfit', 'profit'). No additional data structures are used.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int minPrice = INT_MAX;
        int maxProfit = 0;
        for (int i = 0; i < prices.size(); i++) {
            if (prices[i] < minPrice)
                minPrice = prices[i];
            int profit = prices[i] - minPrice;
            if (profit > maxProfit)
                maxProfit = profit;
        }
        return maxProfit;
    }
};

int main() {
    Solution obj;

    vector<int> prices = {7, 1, 5, 3, 6, 4};

    cout << "Maximum Profit: " <<
obj.maxProfit(prices) << endl;

    return 0;
}
```

```
}
```

Time Complexity: $O(n)$, We only iterate through the prices array once, making it linear time.

Space Complexity: $O(1)$, We use only a few variables (minPrice, maxProfit) for tracking state; no extra space used.

Buy and Sell Stock - II | (DP -36)

Memoization Approach

```
#include <bits/stdc++.h>

using namespace std;

class StockProfit{
long getAns(long *Arr, int ind, int buy, int n,
vector<vector<long>> &dp) {
    if (ind == n) {
        return 0;
    }
    if (dp[ind][buy] != -1) {
        return dp[ind][buy];
    }
    long profit = 0;
    if (buy == 0) {
        profit = max(0 + getAns(Arr, ind + 1, 0, n,
dp), -Arr[ind] + getAns(Arr, ind + 1, 1, n, dp));
    }
    if (buy == 1) {
        profit = max(0 + getAns(Arr, ind + 1, 1, n,
dp), Arr[ind] + getAns(Arr, ind + 1, 0, n, dp));
    }
    return dp[ind][buy] = profit;
}

long getMaximumProfit(long *Arr, int n) {
    vector<vector<long>> dp(n, vector<long>(2, -
1));
    if (n == 0) {
        return 0;
    }
    long ans = getAns(Arr, 0, 0, n, dp);
}
```

```

        return ans;
    }

    int main() {
        int n = 6;

        long Arr[n] = {7, 1, 5, 3, 6, 4};

        return 0;
    }

```

Time Complexity: $O(N^2)$. There are N^2 states therefore at max ' N^2 ' new problems will be solved and we are running a for loop for ' N ' times to calculate the total sum

Space Complexity: $O(N^2) + O(N)$. We are using a recursion stack space($O(N)$) and a 2D array ($O(N^2)$).

Tabulation Approach

```

#include <bits/stdc++.h>

using namespace std;

class StockProfit {
    long getMaximumProfit(long *Arr, int n) {
        vector<vector<long>>> dp(n + 1, vector<long>(2, -1));

        dp[n][0] = dp[n][1] = 0;

        long profit;

        for (int ind = n - 1; ind >= 0; ind--) {
            for (int buy = 0; buy <= 1; buy++) {
                if (buy == 0) {
                    profit = max(0 + dp[ind + 1][0], -Arr[ind] + dp[ind + 1][1]);
                }
                if (buy == 1) {
                    profit = max(0 + dp[ind + 1][1], Arr[ind] + dp[ind + 1][0]);
                }
                dp[ind][buy] = profit;
            }
        }

        return dp[0][0];
    }
}

int main() {
    int n = 6;

    long Arr[n] = {7, 1, 5, 3, 6, 4};

```

```

        cout << "The maximum profit that can be generated is " << getMaximumProfit(Arr, n);

        return 0;
    }

```

Time Complexity: $O(N^2)$. Reason: There are two nested loops that account for $O(N^2)$ complexity.

Space Complexity: $O(N^2)$. We are using an external array of size ' N^2 '. Stack Space is eliminated.

Space Optimization Approach

```

#include <bits/stdc++.h>

using namespace std;

class StockProfit {
    long getMaximumProfit(long *Arr, int n) {
        vector<long> ahead(2, 0);
        vector<long> cur(2, 0);
        ahead[0] = ahead[1] = 0;

        long profit;

        for (int ind = n - 1; ind >= 0; ind--) {
            for (int buy = 0; buy <= 1; buy++) {
                if (buy == 0) {
                    profit = max(0 + ahead[0], -Arr[ind] + ahead[1]);
                }
                if (buy == 1) {
                    profit = max(0 + ahead[1], Arr[ind] + ahead[0]);
                }
                cur[buy] = profit;
            }
            ahead = cur;
        }

        return cur[0];
    }
}

int main() {
    int n = 6;

    long Arr[n] = {7, 1, 5, 3, 6, 4};

    cout << "The maximum profit that can be generated is " << getMaximumProfit(Arr, n);

```

```

    return 0;
}

```

Time Complexity: $O(N^2)$. Reason: There are two nested loops that account for $O(N^2)$ complexity

Space Complexity: $O(1)$. Reason: We are using an external array of size '2'.

Buy and Sell Stock - III | (DP - 37)

Memoization Approach

```

#include <bits/stdc++.h>

using namespace std;

class StockProfit{

int getAns(vector<int>& Arr, int n, int ind, int buy,
int cap, vector<vector<vector<int>>>& dp) {

    if (ind == n || cap == 0)

        return 0;

    if (dp[ind][buy][cap] != -1)

        return dp[ind][buy][cap];

    int profit;

    if (buy == 0) {

        profit = max(0 + getAns(Arr, n, ind + 1, 0,
cap, dp),

                    -Arr[ind] + getAns(Arr, n, ind + 1, 1,
cap, dp));

    }

    if (buy == 1) {

        profit = max(0 + getAns(Arr, n, ind + 1, 1,
cap, dp),

                    Arr[ind] + getAns(Arr, n, ind + 1, 0,
cap - 1, dp));

    }

    return dp[ind][buy][cap] = profit;

}

int maxProfit(vector<int>& prices, int n) {

    vector<vector<vector<int>>> dp(n,
vector<vector<int>>(2, vector<int>(3, -1)));

    return getAns(prices, n, 0, 0, 2, dp);

}

int main() {

```

```

vector<int> prices = {3, 3, 5, 0, 0, 3, 1, 4};

int n = prices.size();

```

```

cout << "The maximum profit that can be
generated is " << maxProfit(prices, n);

```

```

    return 0;

}

```

Time Complexity: $O(N^2*3)$, There are N^2*3 states therefore at max ' N^2*3 ' new problems will be solved.

Space Complexity: $O(N^2*3) + O(N)$, We are using a recursion stack space($O(N)$) and a 3D array ($O(N^2*3)$).

Tabulation Approach

```

#include <bits/stdc++.h>

using namespace std;

class StockProfit{

int maxProfit(vector<int>& Arr, int n) {

    vector<vector<vector<int>>> dp(n + 1,
vector<vector<int>>(2, vector<int>(3, 0)));

    for (int ind = n - 1; ind >= 0; ind--) {

        for (int buy = 0; buy <= 1; buy++) {

            for (int cap = 1; cap <= 2; cap++) {

                if (buy == 0) { // We can buy the stock

                    dp[ind][buy][cap] = max(0 + dp[ind +
1][0][cap],

                                            -Arr[ind] + dp[ind +
1][1][cap]);

                }

                if (buy == 1) { // We can sell the stock

                    dp[ind][buy][cap] = max(0 + dp[ind +
1][1][cap],

                                            Arr[ind] + dp[ind +
1][0][cap - 1]);

                }

            }

        }

    }

    return dp[0][0][2];

}

int main() {

    vector<int> prices = {3, 3, 5, 0, 0, 3, 1, 4};

```

```

int n = prices.size();

cout << "The maximum profit that can be
generated is " << maxProfit(prices, n);

return 0;
}

```

Time Complexity: $O(N*2*3)$ There are three nested loops that account for $O(N*2*3)$ complexity.

Space Complexity: $O(N*2*3)$, We are using an external array of size ' $N*2*3$ '. Stack Space is eliminated.

Space Optimization Approach

```

#include <bits/stdc++.h>

using namespace std;

class StockProfit{
int maxProfit(vector<int>& Arr, int n) {
    vector<vector<int>> ahead(2, vector<int>(3, 0));
    vector<vector<int>> cur(2, vector<int>(3, 0));
    for (int ind = n - 1; ind >= 0; ind--) {
        for (int buy = 0; buy <= 1; buy++) {
            for (int cap = 1; cap <= 2; cap++) {
                if (buy == 0) { // We can buy the stock
                    cur[buy][cap] = max(0 +
ahead[0][cap],
                                -Arr[ind] +
ahead[1][cap]);
                }
                if (buy == 1) {
                    cur[buy][cap] = max(0 +
                                Arr[ind] + ahead[0][cap -
1]);
                }
            }
        }
        ahead = cur;
    }
    return ahead[0][2];
}

int main() {
    vector<int> prices = {3, 3, 5, 0, 0, 3, 1, 4};

```

```

int n = prices.size();

cout << "The maximum profit that can be
generated is " << maxProfit(prices, n);

return 0;
}

```

Time Complexity: $O(N*2*3)$. There are three nested loops that account for $O(N*2*3)$ complexity

Space Complexity: $O(1)$, We are using two external arrays of size ' $2*3$ '.

Buy and Sell Stock - IV | (DP - 38)

Memoization Approach

```

#include <bits/stdc++.h>

using namespace std;

class StockBuySell{
int getAns(vector& Arr, int n, int ind, int buy, int
cap, vector>>& dp) {
    if (ind == n || cap == 0) return 0;
    if (dp[ind][buy][cap] != -1)
        return dp[ind][buy][cap];
    int profit;
    if (buy == 0) {
        profit = max(0 + getAns(Arr, n, ind + 1, 0,
cap, dp),
                    -Arr[ind] + getAns(Arr, n, ind + 1, 1,
cap, dp));
    }
    if (buy == 1) {
        profit = max(0 + getAns(Arr, n, ind + 1, 1,
cap, dp),
                    Arr[ind] + getAns(Arr, n, ind + 1, 0,
cap - 1, dp));
    }
    return dp[ind][buy][cap] = profit;
}

int maximumProfit(vector& prices, int n, int k) {
    vector>> dp(n, vector>(2, vector(k + 1, -1)));
    return getAns(prices, n, 0, 0, k, dp);
}

```

```
int main() {
    vector prices = {3, 3, 5, 0, 0, 3, 1, 4};
    int n = prices.size();
    int k = 2;

    cout << "The maximum profit that can be
generated is " << maximumProfit(prices, n, k) <<
endl;

    return 0;
}
```

Time Complexity: $O(N^2 \cdot 3)$,There are $N^2 \cdot 3$ states therefore at max ' $N^2 \cdot 3$ ' new problems will be solved.

Space Complexity: $O(N^2 \cdot 3) + O(N)$, We are using a recursion stack space($O(N)$) and a 3D array ($O(N^2 \cdot 3)$).

Tabulation Approach

```
#include <bits/stdc++.h>

using namespace std;

class StockBuySell{
int maximumProfit(vector& Arr, int n, int k) {
    vector>> dp(n + 1, vector>(2, vector(k + 1, 0)));
    for (int ind = n - 1; ind >= 0; ind--) {
        for (int buy = 0; buy <= 1; buy++) {
            for (int cap = 1; cap <= k; cap++) {
                if (buy == 0) { // We can buy the stock
                    dp[ind][buy][cap] = max(0 + dp[ind +
1][0][cap],
                    -Arr[ind] + dp[ind +
1][1][cap]);
                }
                if (buy == 1) {
                    dp[ind][buy][cap] = max(0 + dp[ind +
1][1][cap],
                    Arr[ind] + dp[ind + 1][0][cap -
1]);
                }
            }
        }
    }
    return dp[0][0][k];
}
```

```
int main() {
    vector prices = {3, 3, 5, 0, 0, 3, 1, 4};
    int n = prices.size();
    int k = 2;

    cout << "The maximum profit that can be
generated is " << maximumProfit(prices, n, k) <<
endl;

    return 0;
}
```

Time Complexity: $O(N^2 \cdot 3)$, There are three nested loops that account for $O(N^2 \cdot 3)$ complexity.

Space Complexity: $O(N^2 \cdot 3)$, We are using an external array of size ' $N^2 \cdot 3$ '. Stack Space is eliminated.

Space Optimization Approach

```
#include <bits/stdc++.h>

using namespace std;

class StockBuySell{
int maximumProfit(vector& Arr, int n, int k) {
    vector>> dp(n + 1,
    vector>(2, vector(k + 1, 0)));
    for (int ind = n - 1; ind >= 0; ind--) {
        for (int buy = 0; buy <= 1; buy++) {
            for (int cap = 1; cap <= k; cap++) {
                if (buy == 0) { // We can buy the stock
                    dp[ind][buy][cap] = max(0 + dp[ind +
1][0][cap],
                    -Arr[ind] + dp[ind +
1][1][cap]);
                }
                if (buy == 1) { // We can sell the stock
                    dp[ind][buy][cap] = max(0 + dp[ind +
1][1][cap],
                    Arr[ind] + dp[ind + 1][0][cap -
1]);
                }
            }
        }
    }
    return dp[0][0][k];
}
```

```

}

int main() {
    vector prices = {3, 3, 5, 0, 0, 3, 1, 4};
    int n = prices.size();
    int k = 2;

    cout << "The maximum profit that can be
generated is " << maximumProfit(prices, n, k) <<
endl;

    return 0;
}

```

Time Complexity: $O(N^2 \cdot 3)$, There are three nested loops that account for $O(N^2 \cdot 3)$ complexity

Space Complexity: $O(1)$, We are using two external arrays of size ' $2 \cdot 3$ '.

Buy and Sell Stocks With Cooldown | (DP - 39)

Memoization Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution{
int getAns(vector<int> Arr, int ind, int buy, int n,
vector<vector<int>> &dp) {
    if (ind >= n) return 0;
    if (dp[ind][buy] != -1)
        return dp[ind][buy];
    int profit;
    if (buy == 0) {
        profit = max(0 + getAns(Arr, ind + 1, 0, n,
dp), -Arr[ind] + getAns(Arr, ind + 1, 1, n, dp));
    }
    if (buy == 1) {
        profit = max(0 + getAns(Arr, ind + 1, 1, n,
dp), Arr[ind] + getAns(Arr, ind + 2, 0, n, dp));
    }
    return dp[ind][buy] = profit;
}

int stockProfit(vector<int> &Arr) {
    int n = Arr.size();

```

```

    vector<vector<int>> dp(n, vector<int>(2, -1));

    int ans = getAns(Arr, 0, 0, n, dp);

    return ans;
}

int main() {
    vector<int> prices {4, 9, 0, 4, 10};

    cout << "The maximum profit that can be
generated is " << stockProfit(prices) << endl;

    return 0;
}

```

Time Complexity: $O(N^2)$. There are N^2 states therefore at max ' N^2 ' new problems will be solved and we are running a for loop for ' N ' times to calculate the total sum.

Space Complexity: $O(N^2) + O(N)$. We are using a recursion stack space $O(N)$ and a 2D array $O(N^2)$.

Tabulation Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution{
int stockProfit(vector<int> &Arr) {
    int n = Arr.size();
    vector<vector<int>> dp(n + 2, vector<int>(2,
0));
    for (int ind = n - 1; ind >= 0; ind--) {
        for (int buy = 0; buy <= 1; buy++) {
            int profit;
            if (buy == 0) {
                profit = max(0 + dp[ind + 1][0], -Arr[ind]
+ dp[ind + 1][1]);
            }
            if (buy == 1) {
                profit = max(0 + dp[ind + 1][1], Arr[ind]
+ dp[ind + 2][0]);
            }
            dp[ind][buy] = profit;
        }
    }
    return dp[0][0];
}

```

```

}

int main() {
    vector<int> prices {4, 9, 0, 4, 10};

    cout << "The maximum profit that can be
generated is " << stockProfit(prices) << endl;

    return 0;
}

```

Time Complexity: $O(N^2)$. There are two nested loops that account for $O(N^2)$ complexity.

Space Complexity: $O(N^2)$. We are using an external array of size ' N^2 '. Stack Space is eliminated.

Space Optimization Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution{
int stockProfit(vector<int> &Arr) {
    int n = Arr.size();

    vector<int> cur(2, 0);
    vector<int> front1(2, 0);
    vector<int> front2(2, 0);

    for (int ind = n - 1; ind >= 0; ind--) {
        for (int buy = 0; buy <= 1; buy++) {
            int profit;

            if (buy == 0) {
                profit = max(0 + front1[0], -Arr[ind] +
front1[1]);
            }

            if (buy == 1) {
                profit = max(0 + front1[1], Arr[ind] +
front2[0]);
            }

            cur[buy] = profit;
        }

        front2 = front1;
        front1 = cur;
    }

    return cur[0];
}

```

```

int main() {
    vector<int> prices {4, 9, 0, 4, 10};

    cout << "The maximum profit that can be
generated is " << stockProfit(prices) << endl;

    return 0;
}

```

Time Complexity: $O(N^2)$. There are two nested loops that account for $O(N^2)$ complexity

Space Complexity: $O(1)$. We are using three external arrays of size '2'.

Buy and Sell Stocks With Transaction Fees | (DP - 40)

Memoization Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution{
int getAns(vector<int> &Arr, int ind, int buy, int n,
int fee, vector<vector<int>> &dp) {
    if (ind == n) return 0;

    if (dp[ind][buy] != -1)
        return dp[ind][buy];

    int profit;

    if (buy == 0) {
        profit = max(0 + getAns(Arr, ind + 1, 0, n, fee,
dp), -Arr[ind] + getAns(Arr, ind + 1, 1, n, fee, dp));
    }

    if (buy == 1) {
        profit = max(0 + getAns(Arr, ind + 1, 1, n, fee,
dp), Arr[ind] - fee + getAns(Arr, ind + 1, 0, n, fee,
dp));
    }

    return dp[ind][buy] = profit;
}

int maximumProfit(int n, int fee, vector<int> &Arr)
{
    vector<vector<int>> dp(n, vector<int>(2, -1));

    if (n == 0) return 0;

    int ans = getAns(Arr, 0, 0, n, fee, dp);
}

```

```

        return ans;
    }

    int main() {
        vector<int> prices = {1, 3, 2, 8, 4, 9};
        int n = prices.size();
        int fee = 2;

        cout << "The maximum profit that can be
generated is " << maximumProfit(n, fee, prices) <<
endl;

        return 0;
    }

```

Time Complexity: $O(N^2)$. There are N^2 states therefore at max ' N^2 ' new problems will be solved and we are running a for loop for ' N ' times to calculate the total sum.

Space Complexity: $O(N^2) + O(N)$. We are using a recursion stack space $O(N)$ and a 2D array $O(N^2)$.

Tabulation Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution{
    int maximumProfit(int n, int fee, vector<int>& Arr)
    {
        if (n == 0) return 0;

        vector<vector<int>> dp(n + 1, vector<int>(2,
0));

        for (int ind = n - 1; ind >= 0; ind--) {
            for (int buy = 0; buy <= 1; buy++) {
                int profit;
                if (buy == 0) {
                    profit = max(0 + dp[ind + 1][0], -Arr[ind]
+ dp[ind + 1][1]);
                }
                if (buy == 1) {
                    profit = max(0 + dp[ind + 1][1], Arr[ind]
- fee + dp[ind + 1][0]);
                }
                dp[ind][buy] = profit;
            }
        }
    }
}

```

```

        return dp[0][0];
    }

    int main() {
        vector<int> prices = {1, 3, 2, 8, 4, 9};
        int n = prices.size();
        int fee = 2;

        cout << "The maximum profit that can be
generated is " << maximumProfit(n, fee, prices) <<
endl;

        return 0;
    }

```

Time Complexity: $O(N^2)$. There are two nested loops that account for $O(N^2)$ complexity.

Space Complexity: $O(N^2)$. We are using an external array of size ' N^2 '. Stack Space is eliminated.

Space Optimization Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution{
    int maximumProfit(int n, int fee, vector<int>& Arr)
    {
        if (n == 0) return 0;

        vector<long> ahead(2, 0);
        vector<long> cur(2, 0);
        ahead[0] = ahead[1] = 0;

        long profit;

        for (int ind = n - 1; ind >= 0; ind--) {
            for (int buy = 0; buy <= 1; buy++) {
                if (buy == 0) {
                    profit = max(0 + ahead[0], -Arr[ind] +
ahead[1]);
                }
                if (buy == 1) {
                    profit = max(0 + ahead[1], Arr[ind] - fee
+ ahead[0]);
                }
                cur[buy] = profit;
            }
            ahead = cur;
        }
    }
}

```

```

    }
    return cur[0];
}

int main() {
    vector<int> prices = {1, 3, 2, 8, 4, 9};
    int n = prices.size();
    int fee = 2;

    cout << "The maximum profit that can be
generated is " << maximumProfit(n, fee, prices) <<
endl;

    return 0;
}

```

Time Complexity: $O(N*2)$. There are two nested loops that account for $O(N*2)$ complexity

Space Complexity: $O(1)$. We are using an external array of size '2'.

Longest Increasing Subsequence | (DP-41)

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    int func(int i, int prevInd, vector<int> &arr,
vector<vector<int>> &dp) {
        if(i == arr.size() - 1) {
            if(prevInd == -1 || arr[prevInd] < arr[i])
return 1;

            return 0;
        }

        if(dp[i][prevInd + 1] != -1) return
dp[i][prevInd + 1];

        int notTake = func(i+1, prevInd, arr, dp);
        int take = 0;

        if(prevInd == -1)
            take = func(i+1, i, arr, dp) + 1;
        else if(arr[i] > arr[prevInd])
            take = func(i+1, i, arr, dp) + 1

```

```

        return dp[i][prevInd + 1] = max(take,
notTake);
    }

public:
    int LIS(vector<int>& nums) {
        int n = nums.size();
        vector<vector<int>> dp(n, vector<int>(n+1, -
1));

        return func(0, -1, nums, dp);
    }
};

int main() {
    vector<int> nums = {10, 9, 2, 5, 3, 7, 101, 18};

    Solution sol;

    int lengthOfLIS = sol.LIS(nums);

    cout << "The length of the LIS for the given
array is: " << lengthOfLIS << endl;

    return 0;
}

```

Time Complexity: $O(n^2)$, where n is the length of the input array. This is because we are using a 2D DP array of size $n \times n$, and each subproblem takes $O(1)$ time to compute.

Space Complexity: $O(n^2)$, due to the 2D DP array used to store the results of subproblems. Additionally, the recursion stack can go up to $O(n)$ in depth, but it is dominated by the DP array.

Printing Longest Increasing Subsequence | (DP-42)

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int>
longestIncreasingSubsequence(vector<int>
&nums) {
        int n = nums.size();
        vector<int> prev(n, -1);

        for (int i = 1; i < n; i++) {
            for (int j = 0; j < i; j++) {

```

```

        if (nums[j] < nums[i] && dp[j] + 1 >
dp[i]) {
            dp[i] = dp[j] + 1;
            prev[i] = j;
        }
    }
}

int maxLen = 0, maxIndex = 0;
for (int i = 0; i < n; i++) {
    if (dp[i] > maxLen) {
        maxLen = dp[i];
        maxIndex = i;
    }
}

vector<int> lisSeq;
int curr = maxIndex;
while (curr != -1) {
    lisSeq.push_back(nums[curr]);
    curr = prev[curr];
}

reverse(lisSeq.begin(), lisSeq.end())
return lisSeq;
}

};

int main() {
    vector<int> nums = {10, 9, 2, 5, 3, 7, 101, 18};
    Solution sol;
    vector<int> lis =
sol.longestIncreasingSubsequence(nums);

    cout << "LIS: ";
    for (int x : lis) cout << x << " ";
    cout << endl;

    return 0;
}

```

Time Complexity: $O(N^2)$, nested loops are used to compute dp array and prev array.
Space Complexity: $O(N)$, space used to store DP array and previous array.

Longest Increasing Subsequence | Binary Search | (DP-43)

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int LIS(vector<int>& nums) {
        int n = nums.size();
        vector<int> temp;
        temp.push_back(nums[0]);
        for(int i=1; i < n; i++) {
            if(nums[i] > temp.back())
                temp.push_back(nums[i]);
            else {
                int ind = lower_bound(temp.begin(),
temp.end(), nums[i]) -
temp.begin();
                temp[ind] = nums[i];
            }
        }
        return temp.size();
    }
};

int main() {
    vector<int> nums = {10, 9, 2, 5, 3, 7, 101, 18};
    Solution sol;
    int lengthOfLIS = sol.LIS(nums);

    cout << "The length of the LIS for the given
array is: " << lengthOfLIS << endl;

    return 0;
}

```

Time Complexity: $O(N \log N)$, where N is the number of elements in the input array. This is because we iterate through each element and perform a binary search on the temporary array.

Space Complexity: $O(N)$, where N is the number of elements in the input array. This is due to the temporary array used to store the longest increasing subsequence.

Longest Divisible Subset | (DP-44)

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> largestDivisibleSubset(vector<int>
nums) {
        int n = nums.size();
        sort(nums.begin(), nums.end());
        vector<int> ans;
        vector<int> dp(n, 1);
        vector<int> parent(n);
        int lastIndex = 0;
        int maxLen = 0;
        for(int i = 0; i < n; i++) {
            parent[i] = i;
            for(int prevInd = 0; prevInd < i; prevInd++)
            {
                if(nums[i] % nums[prevInd] == 0 &&
dp[i] < dp[prevInd] + 1) {
                    dp[i] = dp[prevInd] + 1;
                    parent[i] = prevInd;
                }
            }
            if(dp[i] > maxLen) {
                lastIndex = i;
                maxLen = dp[i];
            }
        }
        int i = lastIndex;
        while(parent[i] != i) {
            ans.push_back(nums[i]);
            i = parent[i];
        }
        ans.push_back(nums[i]);
```

```
        return ans;
    }
};

int main() {
    vector<int> nums = {3, 5, 10, 20};
    Solution sol;
    vector<int> ans =
sol.largestDivisibleSubset(nums);
    cout << "The largest divisible subset is: ";
    for(int x : ans) cout << x << " ";
    return 0;
}
```

Time Complexity: $O(n^2)$, where n is the number of elements in the input array. This is due to the nested loops used to fill the DP and parent arrays.

Space Complexity: $O(n)$, where n is the number of elements in the input array. This is due to the storage of the DP and parent arrays.

Longest String Chain | (DP- 45)

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int longestStringChain(vector<string>& words)
    {
        int n = words.size();
        sort(words.begin(), words.end(), compare);
        vector<int> dp(n, 1);
        int maxLen = 0;
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < i; j++) {
                if(checkPossible(words[i], words[j]) &&
dp[i] < dp[j] + 1) {
                    dp[i] = dp[j] + 1;
                }
            }
            if(dp[i] > maxLen) maxLen = dp[i];
        }
    }
```

```

        return maxLen;
    }
private:
    bool static compare(string &s, string &t) {
        return s.size() < t.size();
    }
    bool checkPossible(string &s, string &t) {
        if(s.size() != t.size() + 1) return false;
        int i = 0, j = 0;
        while(i < s.size()) {
            if(j < t.size() && s[i] == t[j]) {
                i++, j++;
            }
            else {
                i++;
            }
        }
        if(i == s.size() && j == t.size()) return true;
        return false;
    }
};

int main() {
    vector<string> words = {"a", "ab", "abc",
        "abcd", "abcde"};

    Solution sol;

    int lengthOfLongestStringChain =
        sol.longestStrChain(words);

    cout << "The length of the Longest String Chain
is: " << lengthOfLongestStringChain << endl;

    return 0;
}

```

Time Complexity: $O(n^2 * m)$, where n is the number of words and m is the average length of the words. This is because we are checking each word against all previous words to see if it can be formed by adding one character.

Space Complexity: $O(n)$, where n is the number of words. This is due to the storage of the DP array which keeps track of the longest chain length for each word.

Longest Bitonic Subsequence | (DP-46)

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int LongestBitonicSequence(vector<int>& arr) {
        int n = arr.size();
        vector<int> LIS_dp(n, 1);
        int maxLen = 0;
        for(int i = 0; i < n; i++) {
            for(int prev = 0; prev < i; prev++) {
                if(arr[prev] < arr[i] && LIS_dp[i] <
                    LIS_dp[prev] + 1) {
                    LIS_dp[i] = LIS_dp[prev] + 1; //
                    Update the DP value
                }
            }
        }

        vector<int> LDS_dp(n, 1);
        for(int i = n-1; i >= 0; i--) {
            for(int prev = n-1; prev > i; prev--) {
                if(arr[prev] < arr[i] && LDS_dp[i] <
                    LDS_dp[prev] + 1) {
                    LDS_dp[i] = LDS_dp[prev] + 1;
                }
            }
            maxLen = max(maxLen, LIS_dp[i] +
                LDS_dp[i] - 1);
        }

        return maxLen;
    }
};

int main() {
    vector<int> arr = {5, 1, 4, 2, 3, 6, 8, 7};

    Solution sol;

    int lengthOfLongestBitonicSequence =
        sol.LongestBitonicSequence(arr);
}

```

```

    cout << "The length of the Longest Bitonic
Sequence is: " <<
lengthOfLongestBitonicSequence << endl;

    return 0;
}

```

Time Complexity: $O(n^2)$, where n is the number of elements in the array. This is because we are using two nested loops to calculate the LIS and LDS for each element.

Space Complexity: $O(n)$, where n is the number of elements in the array. This is due to the storage of the LIS and LDS arrays, which each require $O(n)$ space.

Number of Longest Increasing Subsequences | (DP-47)

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int findNumberOfLIS(vector<int>& arr) {
        int n = arr.size();
        vector<int> dp(n, 1);
        vector<int> ct(n, 1);
        int maxi = 1;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < i; j++) {
                if (arr[j] < arr[i] && dp[j] + 1 > dp[i]) {
                    dp[i] = dp[j] + 1;
                    ct[i] = ct[j];
                }
                else if (arr[j] < arr[i] && dp[j] + 1 ==
dp[i]) {
                    ct[i] = ct[i] + ct[j];
                }
            }
            maxi = max(maxi, dp[i]);
        }
        int countLIS = 0;
        for (int i = 0; i < n; i++) {
            if (dp[i] == maxi) {

```

```

                countLIS += ct[i];
            }
        }
        return countLIS;
    }
};

int main() {
    vector<int> arr = {1, 5, 4, 3, 2, 6, 7, 2};
    Solution sol;

    cout << "The count of Longest Increasing
Subsequences (LIS) is "
    << sol.findNumberOfLIS(arr) << endl;

    return 0;
}

```

Time Complexity: $O(N^2)$, for each element, we compare it with all previous elements to check if it can extend an increasing subsequence.

Space Complexity: $O(N)$, space used to store length array and count array array.

Matrix Chain Multiplication | (DP-48)

Recursive Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int matrixChainOrder(vector<int>& arr, int i, int
j) {
        if (i == j) return 0;
        int minCost = INT_MAX;
        for (int k = i; k < j; k++) {
            int cost1 = matrixChainOrder(arr, i, k);

            int cost2 = matrixChainOrder(arr, k + 1, j);
            int costMultiply = arr[i - 1] * arr[k] * arr[j];
            int total = cost1 + cost2 + costMultiply;
            minCost = min(minCost, total);

```

```

    }
    return minCost;
}
};

int main() {
    Solution sol;
    vector<int> arr = {40, 20, 30, 10, 30};
    int n = arr.size();
    cout << "Minimum number of multiplications is:
"
        << sol.matrixChainOrder(arr, 1, n - 1) <<
endl;
    return 0;
}

```

Time Complexity: $O(2^n)$. This is because we try every possible parenthesization, which leads to exponential recursion.

Space Complexity: $O(n)$. This comes from the recursion call stack depth in the worst case.

Memoization Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int solve(vector<int>& arr, int i, int j,
vector<vector<int>>& dp) {
        if (i == j) return 0;
        if (dp[i][j] != -1) return dp[i][j];
        int minCost = INT_MAX;
        for (int k = i; k < j; k++) {
            int cost1 = solve(arr, i, k, dp);
            int cost2 = solve(arr, k + 1, j, dp);
            int costMultiply = arr[i - 1] * arr[k] * arr[j];
            int total = cost1 + cost2 + costMultiply;
            minCost = min(minCost, total);
        }
        return dp[i][j] = minCost;
    }
}

```

```

int matrixChainOrder(vector<int>& arr) {
    int n = arr.size();
    vector<vector<int>> dp(n, vector<int>(n, -1));
    return solve(arr, 1, n - 1, dp);
}

int main() {
    Solution sol;
    vector<int> arr = {40, 20, 30, 10, 30};
    cout << "Minimum number of multiplications is:
"
        << sol.matrixChainOrder(arr) << endl;
    return 0;
}

```

Time Complexity: $O(n^3)$. There are $O(n^2)$ subproblems (for each i, j), and for each we try $O(n)$ partitions.

Space Complexity: $O(n^2)$. We use a 2D dp table of size $n \times n$, plus recursion stack $O(n)$.

Matrix Chain Multiplication | Tabulation Method | (DP-49)

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int matrixMultiplication(vector<int>& nums) {
        int n = nums.size();
        vector<vector<int>> dp(n, vector<int>(n, INT_MAX));
        for (int i = 1; i < n; ++i) {
            dp[i][i] = 0;
        }
        for (int length = 2; length < n; ++length) { //
length of the chain
            for (int i = 1; i <= n - length; ++i) {
                int j = i + length - 1;
                for (int k = i; k < j; ++k) {

```

```

        int cost = dp[i][k] + dp[k + 1][j] +
nums[i - 1] * nums[k] * nums[j];
        if (cost < dp[i][j]) {
            dp[i][j] = cost;
        }
    }
}
}

return dp[1][n - 1];
}
};

int main() {
    Solution sol;

    vector<int> nums = {10, 15, 20, 25};

    cout << sol.matrixMultiplication(nums) << endl;

    return 0;
}

```

Time Complexity: $O(N^3)$, where N is the number of matrices. We have three nested loops: one for the length of the chain, one for the starting index, and one for the split point.

Space Complexity: $O(N^2)$, for the dp array used to store the minimum multiplication costs.

Minimum cost to cut the stick| (DP-50)

Recursion

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int findMinimumCost(int i, int j, vector<int>
&cuts) {
        if (i > j) {
            return 0;
        }

        int mini = INT_MAX;

        for (int ind = i; ind <= j; ind++) {
            int ans = cuts[j + 1] - cuts[i - 1] +

```

```

        findMinimumCost(i, ind - 1, cuts) +
        findMinimumCost(ind + 1, j, cuts);

            mini = min(mini, ans);
        }

        return mini;
    }
}

int minimumCost(int n, int c, vector<int> &cuts)
{
    cuts.push_back(n);

    cuts.insert(cuts.begin(), 0);

    sort(cuts.begin(), cuts.end());

    return findMinimumCost(1, c, cuts);
}

};

```

```

int main() {
    vector<int> cuts = {3, 5, 1, 4};

    int c = cuts.size();

    int n = 7;

    Solution sol;

    cout << "The minimum cost incurred is: " <<
sol.minimumCost(n, c, cuts) << endl;

    return 0;
}

```

Time Complexity: $O(2^C)$, as we try all possible cuts between the current boundary, the time complexity is exponential.

Space Complexity: $O(C)$, additional space used for recursive stack.

Memoization

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int findMinimumCost(int i, int j, vector<int>
&cuts, vector<vector<int>> &dp) {
        if (i > j) {
            return 0;
        }

        if (dp[i][j] != -1) {

```

```

        return dp[i][j];
    }

    int mini = INT_MAX;
    for (int ind = i; ind <= j; ind++) {
        int ans = cuts[j + 1] - cuts[i - 1] +
            findMinimumCost(i, ind - 1, cuts, dp)
+
            findMinimumCost(ind + 1, j, cuts,
dp);

        mini = min(mini, ans);
    }
    return dp[i][j] = mini;
}

int minimumCost(int n, int c, vector<int> &cuts)
{
    cuts.push_back(n);
    cuts.insert(cuts.begin(), 0);
    sort(cuts.begin(), cuts.end());
    vector<vector<int>> dp(c + 1, vector<int>(c +
1, -1));
    return findMinimumCost(1, c, cuts, dp);
}

};

int main() {
    vector<int> cuts = {3, 5, 1, 4};
    int c = cuts.size();
    int n = 7;
    Solution sol;

    cout << "The minimum cost incurred is: " <<
sol.minimumCost(n, c, cuts) << endl;

    return 0;
}

```

Time Complexity: $O(C^3)$, for every state in our dp table i.e. c^2 states, we try all possible cuts between i and j.
 Space Complexity: $O(C^2)$, additional space used for dp array and recursion stack.

Tabulation

```

#include <bits/stdc++.h>

using namespace std;

```

```

class Solution {
public:
    int minimumCost(int n, int c, vector<int> &cuts)
    {
        cuts.push_back(n);
        cuts.insert(cuts.begin(), 0);
        sort(cuts.begin(), cuts.end());

        vector<vector<int>> dp(c + 2, vector<int>(c +
2, 0));

        for (int i = c; i >= 1; i--) {
            for (int j = i; j <= c; j++) {
                int mini = INT_MAX;

                for (int ind = i; ind <= j; ind++) {
                    int ans = cuts[j + 1] - cuts[i - 1] +
dp[i][ind - 1] + dp[ind + 1][j];

                    mini = min(mini, ans);
                }

                dp[i][j] = mini;
            }
        }

        return dp[1][c];
    }
};

int main() {
    vector<int> cuts = {3, 5, 1, 4};
    int c = cuts.size();
    int n = 7;
    Solution sol;

    cout << "The minimum cost incurred is: " <<
sol.minimumCost(n, c, cuts) << endl;

    return 0;
}

```

Time Complexity: $O(C^3)$, for every state in our dp table i.e. c^2 states, we try all possible cuts between i and j.
 Space Complexity: $O(C^2)$, additional space used for dp array.

Burst Balloons | Partition DP | DP 51

Recursive Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution{
int maxCoinsHelper(int i, int j, vector<int> &nums)
{
    if (i > j) return 0;

    int maxCoins = INT_MIN;

    for (int k = i; k <= j; k++) {
        int coins = nums[i - 1] * nums[k] * nums[j + 1];

        int remainingCoins = maxCoinsHelper(i, k - 1,
nums) + maxCoinsHelper(k + 1, j, nums);

        maxCoins = max(maxCoins, coins +
remainingCoins);
    }

    return maxCoins;
}

int maxCoins(vector<int> &nums) {

    int n = nums.size();

    nums.insert(nums.begin(), 1);

    nums.push_back(1);

    return maxCoinsHelper(1, n, nums);
}

int main() {

    vector<int> nums = {3, 1, 5, 8};

    int maxCoinsResult = maxCoins(nums);

    cout << "Maximum coins obtained: " <<
maxCoinsResult << "\n";

    return 0;
}
```

Time Complexity: Exponential

Space Complexity: $O(1)$. No extra space used

Memoization Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution{
```

```
int maxCoinsHelper(int i, int j, vector<int> &nums,
vector<vector<int>> &dp) {

    if (i > j) return 0;

    if (dp[i][j] != -1) return dp[i][j];

    int maxCoins = INT_MIN;

    for (int k = i; k <= j; k++) {

        int coins = nums[i - 1] * nums[k] * nums[j + 1];

        int remainingCoins = maxCoinsHelper(i, k - 1,
nums, dp) + maxCoinsHelper(k + 1, j, nums, dp);

        maxCoins = max(maxCoins, coins +
remainingCoins);
    }

    return dp[i][j] = maxCoins;
}

int maxCoins(vector<int> &nums) {

    int n = nums.size();

    nums.insert(nums.begin(), 1);

    nums.push_back(1);

    vector<vector<int>> dp(n + 2, vector<int>(n + 2,
-1));

    return maxCoinsHelper(1, n, nums, dp);
}
```

Time Complexity: $O(N^3)$, There are total N^2 no. of states. And for each state, we are running a partitioning loop roughly for N times.

Space Complexity: $O(N^2)$ + Auxiliary stack space of $O(N)$, N^2 for the dp array we are using.

Tabulation Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution{

int maxCoins(vector<int> &nums) {

    int n = nums.size();

    nums.insert(nums.begin(), 1);

    nums.push_back(1);

    vector<vector<int>> dp(n + 2, vector<int>(n + 2,
0));

    for (int i = n; i >= 1; i--) {

        for (int j = 1; j <= n; j++) {
```

```

        if (i > j) continue;

        int maxi = INT_MIN;

        for (int ind = i; ind <= j; ind++) {

            int coins = nums[i - 1] * nums[ind] *
            nums[j + 1];

            int remainingCoins = dp[i][ind - 1] +
            dp[ind + 1][j];

            maxi = max(maxi, coins +
            remainingCoins);

        }

        dp[i][j] = maxi;

    }

}

return dp[1][n];

}

int main() {

    vector<int> nums = {3, 1, 5, 8};

    int maxCoinsResult = maxCoins(nums);

    cout << "Maximum coins obtained: " <<
    maxCoinsResult << "\n";

    return 0;

}

```

Time Complexity: $O(N^3)$, There are total N^2 no. of states. And for each state, we are running a partitioning loop roughly for N times.

Space Complexity: $O(N^2)$, N^2 for the dp array we are using.

Evaluate Boolean Expression to True | Partition DP: DP 52

Brute Force Approach

```

#include <bits/stdc++.h>

using namespace std;

#define ll long long

const int mod = 1000000007;

int f(int i, int j, int isTrue, string &exp) {

    if (i > j) return 0;

    if (i == j) {

        if (isTrue == 1) return exp[i] == 'T' ? 1 : 0;

```

```

        else return exp[i] == 'F' ? 1 : 0;

    }

    ll ways = 0;

    for (int ind = i + 1; ind <= j - 1; ind += 2) {

        ll iT = f(i, ind - 1, 1, exp);

        ll lF = f(i, ind - 1, 0, exp);

        ll rT = f(ind + 1, j, 1, exp);

        ll rF = f(ind + 1, j, 0, exp);

        if (exp[ind] == '&') {

            if (isTrue) ways = (ways + (iT * rT) % mod)
            % mod;

            else ways = (ways + (lF * rT) % mod + (iT
            * rF) % mod + (lF * rF) % mod) % mod;

        }

        else if (exp[ind] == '|') {

            if (isTrue) ways = (ways + (lF * rT) % mod
            + (iT * rF) % mod + (iT * rT) % mod) % mod;

            else ways = (ways + (lF * rF) % mod) %
            mod;

        }

        else {

            if (isTrue) ways = (ways + (lF * rT) % mod
            + (iT * rF) % mod) % mod;

            else ways = (ways + (lF * rF) % mod + (iT
            * rT) % mod) % mod;

        }

    }

    return ways;

}

int evaluateExp(string &exp) {

    int n = exp.size();

    return f(0, n - 1, 1, exp);

}

int main() {

    string exp = "F|T^F";

    int ways = evaluateExp(exp);

    cout << "The total number of ways: " << ways
    << "\n";

    return 0;
}

```

```
}
```

Time Complexity: $O(3^N)$, as for each operator we recursively compute the number of ways to evaluate all possible left and right subexpressions as True or False. With approximately $N/2$ operators, the recursion explodes exponentially in the worst case.

Space Complexity: $O(N)$, as the recursion stack depth can go up to N in the worst case where N is the length of the expression string.

Better Approach

```
#include <bits/stdc++.h>

using namespace std;

#define ll long long

const int mod = 1000000007;

int f(int i, int j, int isTrue, string &exp,
vector<vector<vector<ll>>> &dp) {

    if (i > j) return 0;

    if (i == j) {
        if (isTrue == 1) return exp[i] == 'T' ? 1 : 0;
        else return exp[i] == 'F' ? 1 : 0;
    }

    if (dp[i][j][isTrue] != -1) return dp[i][j][isTrue];

    ll ways = 0;

    for (int ind = i + 1; ind <= j - 1; ind += 2) {

        ll IT = f(i, ind - 1, 1, exp, dp);

        ll IF = f(i, ind - 1, 0, exp, dp);

        ll rT = f(ind + 1, j, 1, exp, dp);

        ll rF = f(ind + 1, j, 0, exp, dp);

        if (exp[ind] == '&') {

            if (isTrue) ways = (ways + (IT * rT) % mod)
% mod;

            else ways = (ways + (IF * rT) % mod + (IT
* rF) % mod + (IF * rF) % mod) % mod;

        }

        else if (exp[ind] == '|') {

            if (isTrue) ways = (ways + (IF * rT) % mod
+ (IT * rF) % mod + (IT * rT) % mod) % mod;

            else ways = (ways + (IF * rF) % mod) %
mod;

        }

    }

    else {
```

```
        if (isTrue) ways = (ways + (IF * rT) % mod
+ (IT * rF) % mod) % mod;
```

```
        else ways = (ways + (IF * rF) % mod + (IT
* rT) % mod) % mod;
```

```
    }
```

```
}
```

```
return dp[i][j][isTrue] = ways;
```

```
}
```

```
int evaluateExp(string &exp) {
```

```
    int n = exp.size();
```

```
    vector<vector<vector<ll>>> dp(n,
vector<vector<ll>>>(n, vector<ll>(2, -1))); // DP
table initialization
```

```
    return f(0, n - 1, 1, exp, dp); // Start evaluation
with isTrue set to true.
```

```
}
```

```
int main() {
```

```
    string exp = "F|T^F";
```

```
    int ways = evaluateExp(exp);
```

```
    cout << "The total number of ways: " << ways
<< "\n";
```

```
    return 0;
```

```
}
```

Time Complexity: $O(N \times N \times 2 \times N) = O(N^3)$, There are a total of $2 \times N^2$ states (for each i, j , and boolean $isTrue$). For each state, we iterate through the partition points, which takes $O(N)$ time.

Space Complexity: $O(2 \times N^2) + O(N)$ auxiliary stack space, The DP array takes $O(2 \times N^2)$ space, and the recursion stack (in worst case) adds $O(N)$ more.

Optimal Approach

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
#define ll long long
```

```
const int mod = 1000000007;
```

```
int evaluateExp(string &exp) {
```

```
    int n = exp.size();
```

```
    vector<vector<vector<ll>>> dp(n,
vector<vector<ll>>>(n, vector<ll>(2, 0)));
```

```
    for (int i = n - 1; i >= 0; i--) {
```

```

        for (int j = 0; j <= n - 1; j++) {
            if (i > j) continue;
            for (int isTrue = 0; isTrue <= 1; isTrue++) {
                if (i == j) {
                    if (isTrue == 1) dp[i][j][isTrue] =
exp[i] == 'T';
                    else dp[i][j][isTrue] = exp[i] == 'F';
                    continue;
                }
                ll ways = 0;
                for (int ind = i + 1; ind <= j - 1; ind += 2)
{
                    ll lT = dp[i][ind - 1][1];
                    ll lF = dp[i][ind - 1][0];
                    ll rT = dp[ind + 1][j][1];
                    ll rF = dp[ind + 1][j][0];
                    if (exp[ind] == '&') {
                        if (isTrue) ways = (ways + (lT * rT)
% mod) % mod;
                        else ways = (ways + (lF * rT) %
mod + (lT * rF) % mod + (lF * rF) % mod) % mod;
                    }
                    else if (exp[ind] == '|') {
                        if (isTrue) ways = (ways + (lF * rT)
% mod + (lT * rF) % mod + (lT * rT) % mod) %
mod;
                        else ways = (ways + (lF * rF) %
mod) % mod;
                    }
                    else {
                        if (isTrue) ways = (ways + (lF * rT)
% mod + (lT * rF) % mod) % mod;
                        else ways = (ways + (lF * rF) %
mod + (lT * rT) % mod) % mod;
                    }
                }
                dp[i][j][isTrue] = ways;
            }
        }
    }
}

```

```

    }
    return dp[0][n - 1][1];
}

int main() {
    string exp = "F|T^F";
    int ways = evaluateExp(exp);
    cout << "The total number of ways: " << ways
<< "\n";
    return 0;
}

```

Time Complexity: $O(N * N^2 * N) \sim O(N^3)$, There are a total of $2 * N^2$ no. of states. And for each state, we are running a partitioning loop roughly for N times.

Space Complexity: $O(2 * N^2)$, $2 * N^2$ for the dp array we are using.

Palindrome Partitioning - II | Front Partition : DP 53

Memoization

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
private:
    bool isPalindrome(const string& s, int start, int
end) {
        while (start < end) {
            if (s[start] != s[end])
                return false;
            start++;
            end--;
        }
        return true;
    }
    int minCutsHelper(const string& s, int start,
vector<int>& memo) {
        int n = (int)s.size();
        if (start == n || isPalindrome(s, start, n - 1))
            return 0;
    }
}

```

```

        if (memo[start] != -1)
            return memo[start];
        int minCuts = INT_MAX;
        for (int end = start; end < n; end++) {
            if (isPalindrome(s, start, end)) {
                int cuts = 1 + minCutsHelper(s, end + 1,
memo);
                minCuts = min(minCuts, cuts);
            }
        }
        return memo[start] = minCuts;
    }
public:
    int minCut(string s) {
        int n = (int)s.size();
        vector<int> memo(n, -1);
        return minCutsHelper(s, 0, memo);
    }
};

int main() {
    string s = "aab";
    Solution sol;

    cout << "Minimum cuts needed: " <<
sol.minCut(s) << "\n";

    return 0;
}

```

Time Complexity: $O(N^2)$, there are total N states, where N is the length of array and for every state we check whether substring is a palindrome or not.
Space Complexity: $O(N) + O(N)$, we use a DP array to avoid recomputation. Extra auxiliary stack space is used for recursion.

Tabulation

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    bool isPalindrome(const string& s, int i, int j) {

```

```

        while (i < j) {
            if (s[i] != s[j])
                return false;

            i++;
            j--;
        }
        return true;
    }
public:
    int minCut(string s) {
        int n = (int)s.size();
        vector<int> dp(n + 1, 0);
        dp[n] = -1;
        for (int i = n - 1; i >= 0; i--) {
            int minCuts = INT_MAX;
            for (int j = i; j < n; j++) {
                if (isPalindrome(s, i, j)) {
                    minCuts = min(minCuts, 1 + dp[j +
1]);
                }
            }
            dp[i] = minCuts;
        }
        return dp[0];
    }
};

int main() {

```

```

    string s = "aab";
    Solution sol;

    cout << "Minimum cuts needed: " <<
sol.minCut(s) << "\n";

    return 0;
}

```

Time Complexity: $O(N^2)$, there are total N states and for every state we check whether the substring is a palindrome or not.

Space Complexity: $O(N)$, we use a DP array to store previously calculated results and avoid recomputation.

Partition Array for Maximum Sum | Front Partition : DP 54

Memoization

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
private:
    int helper(const vector<int>& arr, int k, int start,
vector<int>& memo) {
        int n = (int)arr.size();
        if (start == n) return 0;
        if (memo[start] != -1) return memo[start];
        int maxSum = 0;
        int maxElem = 0;
        for (int length = 1; length <= k && start +
length <= n; length++) {
            maxElem = max(maxElem, arr[start +
length - 1]);
            int currentSum = maxElem * length +
helper(arr, k, start + length, memo);
            maxSum = max(maxSum, currentSum);
        }
        return memo[start] = maxSum;
    }
public:
    int maxSumAfterPartitioning(vector<int>& arr,
int k) {
        int n = (int)arr.size();
        vector<int> memo(n, -1);
        return helper(arr, k, 0, memo);
    }
};

int main() {
    vector<int> arr = {1, 15, 7, 9, 2, 5, 10};
    int k = 3;
    Solution sol;
```

```
    cout << "Maximum sum after partitioning: " <<
sol.maxSumAfterPartitioning(arr, k) << "\n";
```

```
    return 0;
```

```
}
```

Time Complexity: $O(N \cdot K)$, there are total N states, where N is the length of array and for every state we check all subarrays of length K .

Space Complexity: $O(N) + O(N)$, we use a DP array to avoid recomputation. Extra auxiliary stack space is used for recursion.

Tabulation

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int maxSumAfterPartitioning(vector<int>& arr,
int k) {
        int n = (int)arr.size();
        vector<int> dp(n + 1, 0);
        for (int i = n - 1; i >= 0; i--) {
            int maxElem = 0;
            int maxSum = 0;
            for (int length = 1; length <= k && i +
length <= n; length++) {
                maxElem = max(maxElem, arr[i + length
- 1]);
                int currentSum = maxElem * length +
dp[i + length];
                maxSum = max(maxSum, currentSum);
            }
            dp[i] = maxSum;
        }
        return dp[0];
    }
};

int main() {
    vector<int> arr = {1, 15, 7, 9, 2, 5, 10};
    int k = 3;
    Solution sol;
```

```

    cout << "Maximum sum after partitioning: " <<
sol.maxSumAfterPartitioning(arr, k) << "\n";

    return 0;
}

```

Time Complexity: $O(N*K)$, there are total N states and for every state we check all subarrays of length K .
 Space Complexity: $O(N)$, we use a DP array to store previously calculated results and avoid recomputation.

Maximum Rectangle Area with all 1's | DP on Rectangles: DP 55

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int largestRectangleArea(vector<int>& heights)
    {
        stack<int> st;

        int maxArea = 0;

        heights.push_back(0);

        for (int i = 0; i < heights.size(); i++) {
            while (!st.empty() && heights[i] <
heights[st.top()]) {
                int height = heights[st.top()];

                st.pop();

                int width = st.empty() ? i : i - st.top() - 1;

                maxArea = max(maxArea, height *
width);
            }
            st.push(i);
        }

        return maxArea;
    }

    int maximalRectangle(vector<vector<char>>&
matrix) {
        if (matrix.empty()) return 0;

        int m = matrix[0].size();

        vector<int> height(m, 0);

        int maxArea = 0;

        for (auto& row : matrix) {

```

```

            for (int i = 0; i < m; i++) {
                if (row[i] == '1') height[i]++;

                else height[i] = 0;
            }

            maxArea = max(maxArea,
largestRectangleArea(height));
        }

        return maxArea;
    }
};

int main() {
    vector<vector<char>> matrix = {
        {'1','0','1','0','0'},
        {'1','0','1','1','1'},
        {'1','1','1','1','1'},
        {'1','0','0','1','0'}
    };

    Solution sol;

    cout << sol.maximalRectangle(matrix) << endl;

    return 0;
}

```

Time Complexity: $O(N*(M+M))$, where N = total no. of rows and M = total no. of columns.
 Space Complexity: $O(M)$, used for storing heights in array and stack.

Count Square Submatrices with All 1s | DP on Rectangles : DP 56

```

#include <bits/stdc++.h>

using namespace std;

int countSquares(int n, int m, vector<vector<int>>
&arr) {
    vector<vector<int>> dp(n, vector<int>(m, 0));

    for (int j = 0; j < m; j++)
        dp[0][j] = arr[0][j];

    for (int i = 0; i < n; i++)
        dp[i][0] = arr[i][0];

    for (int i = 1; i < n; i++) {

```

```

    for (int j = 1; j < m; j++) {
        if (arr[i][j] == 0)
            dp[i][j] = 0;
        else {
            dp[i][j] = 1 + min(dp[i - 1][j],
                               min(dp[i - 1][j - 1], dp[i][j -
1]));
        }
    }
}

int sum = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        sum += dp[i][j];
    }
}

return sum;
}

int main() {
    vector<vector<int>> arr = {
        {0, 1, 1, 1},
        {1, 1, 1, 1},
        {0, 1, 1, 1}
    };

    int n = 3, m = 4;

    int squares = countSquares(n, m, arr);

    cout << "The number of squares: " << squares
    << "\n";

    return 0;
}

```

Time Complexity: $O(n \times m) \rightarrow$ each cell is processed once.

Space Complexity: $O(n \times m) \rightarrow$ DP table used.

Implement Trie – 1

```

#include <bits/stdc++.h>

using namespace std;

```

```

struct Node {
    Node* links[26] = {nullptr};

    bool flag = false;

    bool containsKey(char ch) {
        return links[ch - 'a'] != nullptr;
    }

    void put(char ch, Node* node) {
        links[ch - 'a'] = node;
    }

    Node* get(char ch) {
        return links[ch - 'a'];
    }

    void setEnd() {
        flag = true;
    }

    bool isEnd() {
        return flag;
    }

    ~Node() {
        for (int i = 0; i < 26; i++) {
            if (links[i] != nullptr) {
                delete links[i];
                links[i] = nullptr;
            }
        }
    }
};

class Trie {
private:
    Node* root;

public:
    Trie() {
        root = new Node();
    }

    ~Trie() {
        delete root;
    }
}

```

```

}
void insert(string word) {
    Node* node = root;
    for (char ch : word) {
        if (!node->containsKey(ch)) {
            node->put(ch, new Node());
        }
        node = node->get(ch);
    }
    node->setEnd();
}
bool search(string word) {
    Node* node = root;
    for (char ch : word) {
        if (!node->containsKey(ch)) {
            return false;
        }
        // Move to the next node
        node = node->get(ch);
    }
    return node->isEnd();
}
bool startsWith(string prefix) {
    Node* node = root;
    for (char ch : prefix) {
        if (!node->containsKey(ch)) {
            return false;
        }
        node = node->get(ch);
    }
    return true;
}
};

int main() {
    Trie* trie = new Trie();

```

```

        vector<string> operations = {"Trie", "insert",
        "search", "search", "startsWith", "insert", "search"};

        vector<vector<string>> arguments = { {},
        {"apple"}, {"apple"}, {"app"}, {"app"}, {"app"},
        {"app"} };

        vector<string> output;

        for (int i = 0; i < operations.size(); i++) {
            if (operations[i] == "Trie") {
                output.push_back("null");
            } else if (operations[i] == "insert") {
                trie->insert(arguments[i][0]);
                output.push_back("null");
            } else if (operations[i] == "search") {
                bool result = trie->search(arguments[i][0]);
                output.push_back(result ? "true" : "false");
            } else if (operations[i] == "startsWith") {
                bool result = trie-
                >startsWith(arguments[i][0]);
                output.push_back(result ? "true" : "false");
            }
        }

        for (string res : output) {
            cout << res << endl;
        }

        delete trie;
        return 0;
    }
}

```

Time Complexity:

Insertion: $O(N)$, where N is the length of the word being inserted. This is because we need to go through each letter of the word to either find its corresponding node or create a new node as needed.

Search: $O(N)$, where N is the length of the word being searched for. During a Trie search, we traverse each letter of the word starting from the root, checking if the current node has a child node at the position of the next letter. This continues until we either reach the end of the word or find a missing letter node.

Prefix Search: $O(N)$, where N is the length of the prefix being searched for. Just like in word search, we go through each letter of the prefix to find its corresponding node.

Space Complexity: $O(N)$, where N is the total number of characters across all unique words inserted into the Trie. For each character in a word, a new node might need to

be created, resulting in space usage that is proportional to the total number of characters.

Implement Trie – II

```
#include <bits/stdc++.h>

using namespace std;

struct Node {
    Node* links[26];
    int cntEndWith = 0;
    int cntPrefix = 0;
    bool containsKey(char ch) {
        return (links[ch - 'a'] != NULL);
    }
    Node* get(char ch) {
        return links[ch - 'a'];
    }
    void put(char ch, Node* node) {
        links[ch - 'a'] = node;
    }
    void increaseEnd() {
        cntEndWith++;
    }
    void increasePrefix() {
        cntPrefix++;
    }
    void deleteEnd() {
        cntEndWith--;
    }
    void reducePrefix() {
        cntPrefix--;
    }
};

class Trie {
private:
    Node* root;
public:
```

```
Trie() {
    root = new Node();
}

void insert(string word) {
    Node* node = root;
    for (int i = 0; i < word.size(); i++) {
        if (!node->containsKey(word[i])) {
            node->put(word[i], new Node());
        }
        node = node->get(word[i]);
        node->increasePrefix();
    }
    node->increaseEnd();
}

int countWordsEqualTo(string word) {
    Node* node = root;
    for (int i = 0; i < word.size(); i++) {
        if (node->containsKey(word[i])) {
            node = node->get(word[i]);
        } else {
            return 0;
        }
    }
    return node->cntEndWith;
}

int countWordsStartingWith(string word) {
    Node* node = root;
    for (int i = 0; i < word.size(); i++) {
        if (node->containsKey(word[i])) {
            node = node->get(word[i]);
        } else {
            return 0;
        }
    }
    return node->cntPrefix;
}
```

```

void erase(string word) {
    Node* node = root;

    for (int i = 0; i < word.size(); i++) {
        if (node->containsKey(word[i])) {
            node = node->get(word[i]);
            node->reducePrefix();
        } else {
            return;
        }
    }

    node->deleteEnd();
}

};

int main() {
    Trie trie;

    trie.insert("apple");
    trie.insert("apple");

    cout << "Inserting strings 'apple' twice into Trie"
    << endl;

    cout << "Count Words Equal to 'apple': ";
    cout << trie.countWordsEqualTo("apple") <<
    endl;

    cout << "Count Words Starting With 'app': ";
    cout << trie.countWordsStartingWith("app") <<
    endl;

    cout << "Erasing word 'apple' from trie" << endl;
    trie.erase("apple");

    cout << "Count Words Equal to 'apple': ";
    cout << trie.countWordsEqualTo("apple") <<
    endl;

    cout << "Count Words Starting With 'app': ";
    cout << trie.countWordsStartingWith("app") <<
    endl;

    cout << "Erasing word 'apple' from trie" << endl;
    trie.erase("apple");

    cout << "Count Words Starting With 'app': ";
    cout << trie.countWordsStartingWith("app") <<
    endl;
}

```

```

return 0;
}

```

Time Complexity: $O(N)$, where N is the length of the word or prefix being processed. Each method (inserting a word, counting words equal to a given word, counting words starting with a prefix, and erasing a word) involves traversing the Trie for each character of the input word or prefix. Thus, the time complexity is linear relative to the length of the word or prefix being processed.

Space Complexity: $O(N)$, where N is the total number of characters across all words inserted into the Trie. The space complexity depends on the number of unique words added to the Trie and the average length of these words

Number of Distinct Substrings in a String Using Trie

Brute-Force Approach

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    set<string> countDistinctSubstrings(const
    string& s) {
        set<string> st;

        int n = s.length();

        for (int i = 0; i < n; i++) {
            string str = "";

            for (int j = i; j < n; j++) {
                str += s[j];
                st.insert(str);
            }
        }

        return st;
    }
};

int main() {
    string s = "striver";

    cout << "Given String: " << s << endl;

    Solution sol;

    set<string> substrings =
    sol.countDistinctSubstrings(s);
}

```

```

int count = 0;

cout << "Distinct Substrings:" << endl;

for (const auto& substr : substrings) {

    cout << substr << endl;

    count++;

}

cout << "Number of distinct substrings: " <<
count + 1 << endl;

return 0;

}

```

Time Complexity: $O(N^2)$., we generate all possible substrings starting from all possible indices.

Space Complexity: $O(N^2)$, maximum number of substrings that the set can hold.

Optimal Approach

```

#include <bits/stdc++.h>

using namespace std;

struct Node {

    Node* links[26];

    bool flag = false;

    bool containsKey(char ch) {

        return (links[ch - 'a'] != NULL);

    }

    Node* get(char ch) {

        return links[ch - 'a'];

    }

    void put(char ch, Node* node) {

        links[ch - 'a'] = node;

    }

    void setEnd() {

        flag = true;

    }

    bool isEnd() {

        return flag;

    }

};

class Solution {

```

```

public:

    int countDistinctSubstrings(string &s) {

        Node* root = new Node();

        int cnt = 0;

        int n = s.size();

        for (int i = 0; i < n; i++) {

            Node* node = root;

            for (int j = i; j < n; j++) {

                if (!node->containsKey(s[j])) {

                    node->put(s[j], new Node());

                    cnt++;

                }

                node = node->get(s[j]);

            }

            return cnt + 1;

        }

    };

    int main() {

        string s = "striver";

        cout << "Current String: " << s << endl;

        Solution sol;

        cout << "Number of distinct substrings: " <<
sol.countDistinctSubstrings(s) << endl;

        return 0;

    }
}

```

Time Complexity: $O(N^2)$, We consider all substrings of a string with length n . For each substring, inserting characters into the Trie takes up to $O(n)$ in the worst case. Space Complexity: $O(N^2)$, additional space used for storing total number of substring in the trie.

Maximum XOR of Two Numbers in an Array

```

#include <bits/stdc++.h>

using namespace std;

class Node {

public:

```

```

Node* links[2];
bool containsKey(int bit) {
    return links[bit] != NULL;
}
Node* get(int bit) {
    return links[bit];
}
void put(int bit, Node* node) {
    links[bit] = node;
}
};

class Solution {
public:
    Node* root;
    Solution() {
        root = new Node();
    }
    void insert(int num) {
        Node* node = root;
        for (int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            if (!node->containsKey(bit)) {
                node->put(bit, new Node());
            }
            node = node->get(bit);
        }
    }
    int getMaxXOR(int num) {
        Node* node = root;
        int maxXor = 0;
        for (int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            if (node->containsKey(1 - bit)) {
                maxXor |= (1 << i);
                node = node->get(1 - bit);
            } else {

```

```

                node = node->get(bit);
            }
        }
        return maxXor;
    }
    int findMaximumXOR(vector<int>& nums) {
        for (int num : nums) {
            insert(num);
        }
        int maxResult = 0;
        for (int num : nums) {
            maxResult = max(maxResult,
getMaxXOR(num));
        }
        return maxResult;
    }
};

int main() {
    vector<int> nums = {3, 10, 5, 25, 2, 8};
    Solution sol;
    cout << sol.findMaximumXOR(nums) << endl;
    return 0;
}

```

Time Complexity: O(N), each number is inserted and queried in the Trie in constant time (32 bits).
Space Complexity: O(N), Trie stores up to 32 bits for each number, resulting in linear space in worst case.

Maximum Xor Queries | Trie

```

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

struct Node {
    Node *links[2];
    bool containsKey(int ind) {
        return (links[ind] != NULL);
    }
    Node* get(int ind) {

```

```

        return links[ind];
    }

    void put(int ind, Node* node) {
        links[ind] = node;
    }
};

class Trie {
private:
    Node* root;
public:
    Trie() {
        root = new Node();
    }

    void insert(int num) {
        Node* node = root;
        for (int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            if (!node->containsKey(bit)) {
                node->put(bit, new Node());
            }
            node = node->get(bit);
        }
    }

    int findMax(int num) {
        Node* node = root;
        int maxNum = 0;
        for (int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            if (node->containsKey(!bit)) {
                maxNum = maxNum | (1 << i);
                node = node->get(!bit);
            }
            else {
                node = node->get(bit);
            }
        }
    }

    return links[ind];
}

return maxNum;
}

};

class Solution {
public:
    vector<int> maxXorQueries(vector<int> &arr,
                             vector<vector<int>>> &queries) {
        vector<int> ans(queries.size(), 0);
        vector<pair<int, pair<int,int>>>
        offlineQueries;

        sort(arr.begin(), arr.end());

        int index = 0;

        for (auto &it: queries) {
            offlineQueries.push_back({it[1], {it[0],
            index++}});
        }

        sort(offlineQueries.begin(),
        offlineQueries.end());

        int i = 0;
        int n = arr.size();

        Trie trie;

        for (auto &it : offlineQueries) {
            while (i < n && arr[i] <= it.first) {
                trie.insert(arr[i]);
                i++;
            }

            if (i != 0)
                ans[it.second.second] =
                trie.findMax(it.second.first);
            else
                ans[it.second.second] = -1;
        }

        return ans;
    }
};

int main() {
    vector<int> arr = {3, 10, 5, 25, 2, 8};

```

```

cout << "Given Array: ";
for (int i = 0; i < arr.size(); i++) {
    cout << arr[i] << " ";
}
cout << endl;
vector<vector<int>> queries = {
    {0, 1}, {1, 2}, {0, 3}, {3, 3}
};
cout << "Queries: ";
for (auto query: queries) {
    cout << query[0] << " " << query[1] << ", ";
}
cout << endl;
Solution obj;
vector<int> result = obj.maxXorQueries(arr,
queries);
cout << "Result of Max XOR Queries:" << endl;
for (int i = 0; i < result.size(); ++i) {
    cout << "Query " << i+1 << ": " << result[i]
<< endl;
}
return 0;
}

```

Time Complexity: $O(32*N + Q(\log Q) + 32*Q)$, for every number in array and query we traverse all its 32 bits to store in the trie. We also sort the queries in ascending order of their end points.

Space Complexity: $O(32*N + Q)$, the space complexity of the Trie depends on the number of bits required to represent the numbers in the input array.

Minimum number of bracket reversals needed to make an expression balanced

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    int minReversalsToBalance(string expression) {
        int openBrackets = 0;

```

```

        int closeBrackets = 0;
        for (char ch : expression) {
            if (ch == '(') {
                openBrackets++;
            } else {
                if (openBrackets > 0) {
                    openBrackets--;
                } else {
                    closeBrackets++;
                }
            }
        }
        if ((openBrackets + closeBrackets) % 2 != 0)
            return -1;
        return (openBrackets + 1) / 2 + (closeBrackets + 1) / 2;
    }
};
int main() {
    string expression = "(()))(";
    Solution solver;
    int result =
solver.minReversalsToBalance(expression);
    cout << "Minimum reversals required: " <<
result << endl;
    return 0;
}

```

Time Complexity: $O(N)$, we traverse the string once processing each bracket in constant time.
Space Complexity: $O(1)$, constant extra space is used.

Count and Say

```

#include <bits/stdc++.h>
using namespace std;
class Solution {
public:
    string countAndSay(int n) {
        string result = "1";

```

```

for (int i = 1; i < n; ++i) {
    string current = "";
    int count = 1;
    for (int j = 1; j < result.size(); ++j) {
        if (result[j] == result[j - 1]) {
            count++;
        } else {
            current += to_string(count) + result[j -
1];
            count = 1;
        }
    }
    current += to_string(count) + result.back();
    result = current;
}
return result;
}
};

int main() {
    Solution solver;

    int n = 5;

    cout << "Count and Say term " << n << ": " <<
solver.countAndSay(n) << endl;

    return 0;
}

```

Time Complexity: $O(n * 2^n)$, there can be n iterations, with string length of upto 2^n characters.
Space Complexity: $O(2^n)$, Final string length can reach upto 2^n

Z Function

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> computeZArray(const string& s) {
        int n = s.length();
        vector<int> z(n, 0);
    }
}

```

```

int left = 0, right = 0;
for (int i = 1; i < n; i++) {
    if (i <= right)
        z[i] = min(right - i + 1, z[i - left]);
    while (i + z[i] < n && s[z[i]] == s[i + z[i]])
        z[i]++;
    if (i + z[i] - 1 > right) {
        left = i;
        right = i + z[i] - 1;
    }
}
return z;
}

vector<int> zFunctionSearch(const string& text,
const string& pattern) {
    string combined = pattern + "$" + text;
    vector<int> z = computeZArray(combined);
    vector<int> result;

    for (int i = pattern.length() + 1; i <
combined.length(); i++) {
        if (z[i] == pattern.length()) {
            result.push_back(i - pattern.length() - 1);
        }
    }
    return result;
}

};

int main() {
    string text = "ababcababc";
    string pattern = "ab";

    Solution sol;

    vector<int> indices = sol.zFunctionSearch(text,
pattern);

    cout << "Pattern found at indices: ";
    for (int index : indices) {
        cout << index << " ";
    }
}

```

```

    cout << endl;

    return 0;
}

```

Time Complexity: $O(n + m)$, where n is the length of the text string and m is the length of the pattern string. The Z-array is computed in linear time using a window technique that avoids unnecessary comparisons.
 Space Complexity: $O(n + m)$, to store the combined string and the Z-array used for processing.

KMP Algorithm or LPS Array

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    vector<int> computeLPS(string pattern) {
        vector<int> lps(pattern.length(), 0);
        int len = 0;
        int i = 1;
        while (i < pattern.length()) {
            if (pattern[i] == pattern[len]) {
                len++;
                lps[i] = len;
                i++;
            } else {
                if (len != 0) {
                    len = lps[len - 1];
                } else {
                    lps[i] = 0;
                    i++;
                }
            }
        }
        return lps;
    }

    vector<int> KMP(string text, string pattern) {
        vector<int> lps = computeLPS(pattern);
        vector<int> result;
    }
}

```

```

int i = 0;
int j = 0;
while (i < text.length()) {
    if (text[i] == pattern[j]) {
        i++;
        j++;
    }

    if (j == pattern.length()) {
        result.push_back(i - j);
        j = lps[j - 1];
    }

    else if (i < text.length() && text[i] !=
pattern[j]) {
        if (j != 0) {
            j = lps[j - 1];
        } else {
            i++;
        }
    }
}

return result;
};

int main() {
    string text = "ababcbababcabc";
    string pattern = "abc";
    Solution sol;

    vector<int> matches = sol.KMP(text, pattern);
    cout << "Pattern found at indices: ";

    for (int idx : matches) {
        cout << idx << " ";
    }

    cout << endl;

    return 0;
}

```

Time Complexity: $O(n + m)$, where n is the length of the text and m is the length of the pattern. This is because both the LPS preprocessing and the pattern searching phases take linear time.
 Space Complexity: $O(m)$, used for storing the LPS array of the pattern.

Longest Happy Prefix

Brute-Force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    string longestPrefix(string s) {
        int n = s.size();
        for (int len = n - 1; len > 0; len--) {
            if (s.substr(0, len) == s.substr(n - len, len)) {
                return s.substr(0, len);
            }
        }
        return "";
    }
};

int main() {
    Solution sol;
    string s = "levellevel";
    cout << sol.longestPrefix(s) << endl;
    return 0;
}
```

Time Complexity: $O(n^2)$, for each of the n possible lengths we compare up to n characters.
 Space Complexity: $O(n)$, for storing substrings during comparison.

Optimal Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    string longestPrefix(string s) {
```

```
        vector<int> lps(s.length(), 0);

        int len = 0;
        for (int i = 1; i < s.length(); i++) {
            if (s[i] == s[len]) {
                len++;
                lps[i] = len;
            }
            else if (len != 0) {
                len = lps[len - 1];
                i--; // retry with same i
            }
        }

        return s.substr(0, lps[s.length() - 1]);
    }
};

int main() {
    Solution sol;
    string s = "levellevel";
    cout << sol.longestPrefix(s) << endl;
    return 0;
}
```

Time Complexity: $O(n)$, we traverse the string once to compute the LPS array and use it to get the result.
 Space Complexity: $O(n)$, extra space is used for the LPS array of length n .

Count palindromic subsequence in given string

Brute Force Approach

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    bool isPalindrome(string s) {
        int left = 0, right = s.size() - 1;
        while (left < right) {
            if (s[left++] != s[right--]) return false;
```

```

    }
    return true;
}

int count(string s, string curr, int index) {
    if (index == s.length()) {
        if (!curr.empty() && isPalindrome(curr))
            return 1;
        return 0;
    }
    int include = count(s, curr + s[index], index + 1);
    int exclude = count(s, curr, index + 1);
    return include + exclude;
}

int countPalindromicSubsequences(string s) {
    return count(s, "", 0);
}
};

int main() {
    Solution sol;
    string s = "abcb";

    cout << sol.countPalindromicSubsequences(s)
    << endl;

    return 0;
}

```

Time Complexity: $O(2^n \cdot n)$, all subsequences are generated and each is checked for being a palindrome.

Space Complexity: $O(n)$, due to recursion stack or temporary string storage.

Optimal Approach – Memoization

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int countPalindromicSubsequences(string s) {
        int n = s.size();
        vector<vector<int>> dp(n, vector<int>(n, -1));

```

```

        return count(s, 0, n - 1, dp);
    }
private:
    int count(string &s, int i, int j,
        vector<vector<int>> &dp) {
        if (i > j) return 0;
        if (i == j) return 1;
        if (dp[i][j] != -1) return dp[i][j];
        if (s[i] == s[j])
            dp[i][j] = count(s, i + 1, j, dp) +
                count(s, i, j - 1, dp) + 1;
        else
            dp[i][j] = count(s, i + 1, j, dp) + count(s, i, j
            - 1, dp) -
                count(s, i + 1, j - 1, dp);
        return dp[i][j];
    }
};

int main() {
    Solution sol;
    string s = "aab";

    cout << sol.countPalindromicSubsequences(s)
    << endl;

    return 0;
}

```

Time Complexity: $O(n^2)$, each substring state is computed once and reused.

Space Complexity: $O(n^2 + n)$, space for the memoization table and recursion stack.

Optimal Approach – Tabulation

```

#include <bits/stdc++.h>

using namespace std;

class Solution {
public:
    int countPalindromicSubsequences(string s) {
        int n = s.size();
        vector<vector<int>> dp(n, vector<int>(n, 0));
        for (int i = 0; i < n; i++) {

```

```

        dp[i][i] = 1;
    }
    for (int len = 2; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            if (s[i] == s[j])
                dp[i][j] = dp[i + 1][j] + dp[i][j - 1] + 1;
            else
                dp[i][j] = dp[i + 1][j] + dp[i][j - 1] -
dp[i + 1][j - 1];
        }
    }
    return dp[0][n - 1];
}
};

int main() {
    Solution sol;
    string s = "aab";
    cout << sol.countPalindromicSubsequences(s)
<< endl;

    return 0; }

```

Time Complexity: $O(n^2)$, all substrings of all lengths are processed once.

Space Complexity: $O(n^2)$, 2D DP table is used to store the results of subproblems.