A. Divide-and-Conquer Algorithm

1. **Quick Sort with Median of Three:**
    a. **best case: O(nlogn)**
    b. **worse case: O(n^2)**
    c. **average case: O(nlogn)**

The best case occurs when the pivot consistently divides the array into two nearly equal partitions, leading to Ω(n log n) time complexity. The worst case occurs when the pivot is consistently chosen poorly, leading to O(n^2) time complexity. The average case occurs when the choice of pivot is random or uses a median-of-three strategy, resulting in Θ(nlogn) time complexity.

Code:

```
#include <iostream> // Standard input-output stream

#include <vector> // Vector container

#include <algorithm> // For std::swap

// Function to partition the array

int partition(std::vector<int>& arr, int low, int high) {

 // Find median of three - O(1)

 int mid = (low + high) / 2; // Calculate mid index - O(1)

 if (arr[low] > arr[mid]) // O(1)

 std::swap(arr[low], arr[mid]); // O(1)

 if (arr[mid] > arr[high]) { // O(1)

 std::swap(arr[mid], arr[high]); // O(1)

 if (arr[low] > arr[mid]) // O(1)

 std::swap(arr[low], arr[mid]); // O(1)

 }

 int pivot = arr[mid]; // O(1)

 int i = low - 1; // O(1)

 int j = high + 1; // O(1)

 // Loop to rearrange elements around the pivot

 while (true) { // O(n)

 do { // Loop to find element greater than or equal to pivot from left side - O(n)

 i++; // O(1)

 } while (arr[i] < pivot); // O(n)



 do { // Loop to find element less than or equal to pivot from right side - O(n)

 j--; // O(1)

 } while (arr[j] > pivot); // O(n)

 // If pointers cross, partitioning is done

 if (i >= j) // O(1)

 return j; // O(1)

 // Swap elements at i and j

 std::swap(arr[i], arr[j]); // O(1)

 }

}

// Function to perform quicksort

void quick_sort(std::vector<int>& arr, int low, int high) {

 // Base case: If there is only one element or none

 if (low < high) {

 // Partition the array

 int pi = partition(arr, low, high); // O(n)

 // Recursively sort the two subarrays

 quick_sort(arr, low, pi); // T(n/2)

 quick_sort(arr, pi + 1, high); // T(n/2)

 }

}

// Main function

int main() {

 // Initialize an array

 std::vector<int> arr = {10, 7, 8, 9, 1, 5}; // O(n)

 int n = arr.size(); // O(1)

 // Call quick_sort function to sort the array

 quick_sort(arr, 0, n - 1); // O(n log n)

 // Print the sorted array

 std::cout << "Sorted array is: "; // O(1)

 for (int i = 0; i < n; i++) // O(n)

 std::cout << arr[i] << " "; // O(1)

 std::cout << std::endl; // O(1)

 return 0; // O(1)

}
```

2. **Closest Pair Problem:**

- Big O (O(n log n)): The algorithm sorts the input points by their x-coordinates and ycoordinates, each of which takes O(n log n) time. The divide and conquer approach for finding the closest pair recursively divides the points into smaller subsets until reaching base cases. Each recursive step takes O(n) time to process the points in the strip and O(1) time for other operations. Therefore, the overall time complexity of the algorithm is O(n log n).
- Omega (Ω(n log n)): In the worst case, the algorithm always needs to consider all the points, even if they are already sorted. This occurs when the closest pair spans across the middle of the points. Thus, the Omega (lower bound) is also O(n log n).
- Theta (Θ(n log n)): Since both the upper and lower bounds are O(n log n), the time complexity of the Closest Pair Problem algorithm can be expressed as Θ(n log n). Therefore, the time complexity of the algorithm is Θ(n log n), where "n" is the number of input points.

Code:

```
#include <iostream> // Including the header file for input and output operations. // O(1)

#include <vector> // Including the header file for vector container. // O(1)

#include <algorithm> // Including the header file for algorithms like sort. // O(1)

#include <cmath> // Including the header file for mathematical functions like sqrt. // O(1)

#include <limits> // Including the header file for numeric limits. // O(1)

struct Point { // Defining a structure to represent a point in 2D space. // O(1)

 double x, y; // Each point has x and y coordinates. // O(1)

};

double distance(const Point& p1, const Point& p2) {  // Function to calculate the Euclidean distance between two points. // O(1)

 return std::sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y)); // Time complexity: O(1) - constant time.

}

std::pair<double, std::pair<Point, Point>> closestPair(const std::vector<Point>& points) { // Function to find the closest pair of points among a vector of points. // O(n^2)

 std::vector<Point> sortedPoints = points; // Creating a copy of the input vector. // O(n)

 std::sort(sortedPoints.begin(), sortedPoints.end(), [](const Point& p1, const Point& p2)

{ // Sorting the points based on their x-coordinates. // O(n log n)

 return p1.x < p2.x; // Comparator function for sorting. // O(1)

 }); // Time complexity: O(n log n) - sorting algorithm (typically quicksort).

 double minDistance = std::numeric_limits<double>::max(); // Initializing the minimum distance with the maximum possible value. // O(1)

 std::pair<Point, Point> closestPair; // Initializing the pair of closest points. // O(1)

for (size_t i = 0; i < sortedPoints.size(); ++i)

{ // Iterating through the sorted points. // O(n)
```

```
 for (size_t j = i + 1; j < sortedPoints.size() && sortedPoints[j].x - sortedPoints[i].x < minDistance; ++j) { // Checking pairs within a certain distance. // O(n^2)

 double dist = distance(sortedPoints[i], sortedPoints[j]); // Calculating the distance between two points. // O(1)

 if (dist < minDistance) { // Checking if the distance is smaller than the current minimum distance. // O(1)

 minDistance = dist; // Updating the minimum distance. // O(1)

 closestPair = {sortedPoints[i], sortedPoints[j]}; // Updating the closest pair of points. // O(1)

 }

 }

 } // Time complexity: O(n^2) - nested loops iterating over all pairs of points.

 return {minDistance, closestPair}; // Returning the minimum distance and the closest pair of points. // O(1)

}

int main() { // Main function where the program execution starts. // O(1)

 std::vector<Point> points = {{2, 3}, {12, 30}, {4, 1}, {7, 16}, {5, 5}}; // Creating a vector of points. // O(n)

 auto result = closestPair(points); // Finding the closest pair of points. // O(n^2)

 double minDistance = result.first; // Extracting the minimum distance from the result. // O(1)

 std::pair<Point, Point> closestPair = result.second; // Extracting the closest pair of points from the result. // O(1)

 std::cout << "Minimum distance: " << minDistance << std::endl; // Printing the minimum distance. // O(1)

 std::cout << "Closest pair: (" << closestPair.first.x << ", " << closestPair.first.y << ")

and (" // Printing the closest pair of points. // O(1)

 << closestPair.second.x << ", " << closestPair.second.y << ")" << std::endl; // O(1)

 return 0; // Indicating successful completion of the program. // O(1)

}
```

3. Integer Exponentiation:

The loop in the function runs until the exponent (exp) becomes 0. In each iteration of the loop, the exponent is halved (exp >>= 1). So, the number of iterations required depends on the number of times the exponent can be halved before reaching 0.

Worst Case (Big O): In the worst-case scenario, the exponent exp is a power of 2 (e.g., 2, 4, 8, 16, ...). In this case, the loop runs until exp becomes 0, and each time the exponent is halved. Therefore, the worst-case time complexity is O(log n), where n is the value of the exponent.

Best Case (Omega): The best-case scenario occurs when the exponent exp is 0 or 1. In this case, the loop only runs once, regardless of the value of the base. Therefore, the best-case time complexity is Ω(1).

Average Case (Theta): The average-case time complexity is also O(log n), as it is similar to the worst case. In most practical scenarios, the exponent is not always a power of 2, so the average number of iterations tends to be logarithmic.

So, summarizing:

- **Big O: O(log n)**
- **Omega: Ω(1)**
- **Theta: Θ(log n)**

Code:

```
#include <iostream> // Including the header file for input and output operations. // O(1)

int ipow(int base, int exp) { // Function to calculate the power of a number. // O(log exp)

 if (exp == 0) // Base case: if exponent is 0, return 1 // O(1)

 return 1; // O(1)

 else if (exp % 2 == 0) { // If exponent is even // O(1)

 int half_pow = ipow(base, exp / 2); // Calculate half of the power // O(log exp)

 return half_pow * half_pow; // Return the square of half of the power // O(1)

 } else { // If exponent is odd // O(1)

 int half_pow = ipow(base, (exp - 1) / 2); // Calculate half of the power // O(log exp)

 return base * half_pow * half_pow; // Return base times the square of half of the power // O(1)

 }

}

int main() { // Main function where the program execution starts. // O(1)

 int base = 2; // Initializing base. // O(1)

 int exponent = 5; // Initializing exponent. // O(1)

 int result = ipow(base, exponent); // Calling the ipow function to calculate the result. // O(log exp)

 std::cout << "Result: " << result << std::endl; // Printing the result. // O(1)

 return 0; // Indicating successful completion of the program. // O(1)

}
```

Proof:

- **T(n) = T(n/2) + O(1)**
- **T(n)= T(logn)**

4. Majority Element:

The worst-case scenario, the time complexity of the algorithm is O(n log n), indicating that it has an upper bound on its time complexity. However, in the best-case scenario, the time complexity is Ω(n), indicating a lower bound. On average, the algorithm performs with a time complexity of Θ(n log n), reflecting its expected performance across various input scenarios.

Worst Case (Big O): In the worst-case scenario, the majority element is spread across both halves of the array in each recursion level.

Best Case (Omega): The best-case scenario occurs when the majority element is found in the first comparison of the entire array. In this case, the algorithm will terminate without needing to split the array further. Therefore, the best-case time complexity is Ω(n).

Average Case (Theta): The average-case time complexity is O(n log n), as it is similar to the worst case. In most practical scenarios, the majority element might be found earlier in some cases, but on average, the algorithm still has to traverse through a significant portion of the array in each recursion level.

So, summarizing:

- **Big O: O(n log n)**
- **Omega: Ω(n)**
- **Theta: Θ(n log n)**

Code:

```
#include <iostream> // Including the header file for input and output operations. // O(1)

#include <vector> // Including the header file for the vector container. // O(1)

// Function to count occurrences of 'num' in array 'arr' within range [low, high]

int countInRange(int arr[], int num, int low, int high) {

 int count = 0; // Initializing count to zero. // O(1)

 for (int i = low; i <= high; i++) { // Looping through the elements within the range [low, high]. // O(high - low + 1)

 if (arr[i] == num) // Checking if the element at index 'i' is equal to 'num'. // O(1)

 count++; // Incrementing count if the condition is true. // O(1)

 }

 return count; // Returning the count of occurrences of 'num'. // O(1)

}

// Function to find majority element in array 'arr' within range [low, high]
```

```
int findMajorityElementInRange(int arr[], int low, int high) {

 if (low == high) // Base case: when only one element in range // O(1)

 return arr[low]; // Returning the element when there's only one element in the range. // O(1)

 // Divide

 int mid = (low + high) / 2; // Calculating the midpoint of the range. // O(1)

 int leftMajority = findMajorityElementInRange(arr, low, mid); // Recursive call for the left half. // O(log n)

 int rightMajority = findMajorityElementInRange(arr, mid + 1, high); // Recursive call for the right half. // O(log n)

 // Conquer

 if (leftMajority == rightMajority) // If left and right parts have the same majority element // O(1)

 return leftMajority; // Returning the common majority element. // O(1)

 // Count occurrences of left and right majority elements in the whole range

 int leftCount = countInRange(arr, leftMajority, low, high); // Counting occurrences of the left majority element in the whole range. // O(n)

 int rightCount = countInRange(arr, rightMajority, low, high); // Counting occurrences of the right majority element in the whole range. // O(n)

 // Return the majority element with more occurrences

 return (leftCount > rightCount) ? leftMajority : rightMajority; // Returning the majority element with more occurrences. // O(1)

}

// Function to find majority element in array 'arr'

int findMajorityElement(int arr[], int size) {

 return findMajorityElementInRange(arr, 0, size - 1); // Calling the helper function to find the majority element within the whole range. // O(log n)

}

int main() { // Main function where the program execution starts. // O(1)

 int arr[] = {3, 3, 3, 3, 3, 3, 3, 3, 4, 2, 4, 4, 2, 4, 4}; // Initializing the array. // O(1)

 int size = sizeof(arr) / sizeof(arr[0]); // Calculating the size of the array. // O(1)

 int result = findMajorityElement(arr, size); // Calling the function to find the majority element. // O(log n)

 if (result != -1) // Checking if the majority element is found. // O(1)

 std::cout << "Majority element: " << result << std::endl; // Printing the majority

element. // O(1)

 else

 std::cout << "No majority element found." << std::endl; // Printing message if no majority element is found. // O(1)

 return 0; // Indicating successful completion of the program. // O(1)

}
```

5. Convex Hull :

The time complexity of this code is O(nlogn), where n is the number of points in the input set. This complexity arises due to the divide-and-conquer approach used to find the convex hull. The space complexity is O(n). Therefore, the code efficiently computes the convex hull while maintaining good performance characteristics, especially for large input sizes.

Best Case (Omega): The best-case scenario for the provided code is when the points are already sorted in increasing order of their x-coordinates. In this case, the sorting step will take O(nlogn) time, and the subsequent divide-and-conquer steps will also take O(nlogn) time. Therefore, the best-case time complexity is Ω(nlogn).

Worse Case (O): The worst-case scenario occurs when the points are arranged in such a way that the divide-and-conquer algorithm has to repeatedly split the set into two nearly equal halves. In this case, each level of recursion will take O(n) time, resulting in a total time complexity of O(nlogn).

Average Case (Θ): The average-case time complexity is also O(nlogn). because, on average, the algorithm will require Θ (nlogn) operations to compute the convex hull, assuming a random distribution of points.

Summarizing:

- **Big O: O(nlogn)**
- **Omega: Ω(nlogn)**
- **Theta: Θ (nlogn)**

Code:

```
// A divide and conquer program to find convex hull of a given set of points.

#include <bits/stdc++.h> // Including all standard C++ headers. Avoid using this in production code.

using namespace std;

// stores the centre of polygon (It is made global because it is used in compare function)

pair<int, int> mid;

// determines the quadrant of a point (used in compare())

int quad(pair<int, int> p)

{

  if (p.first >= 0 && p.second >= 0)

    return 1;

  if (p.first <= 0 && p.second >= 0)

    return 2;

  if (p.first <= 0 && p.second <= 0)

    return 3;

  return 4;

}

// Time complexity: O(1) - each condition check is constant time.

// Checks whether the line is crossing the polygon

int orientation(pair<int, int> a, pair<int, int> b,

          pair<int, int> c)

{

  int res = (b.second - a.second) * (c.first - b.first) -

        (c.second - b.second) * (b.first - a.first);

  if (res == 0)

    return 0;

  if (res > 0)

    return 1;

  return -1;

}

// Time complexity: O(1) - all operations are constant time. compare function for sorting

bool compare(pair<int, int> p1, pair<int, int> q1)

{

  pair<int, int> p = make_pair(p1.first - mid.first,

                p1.second - mid.second);

  pair<int, int> q = make_pair(q1.first - mid.first,

                q1.second - mid.second);

  int one = quad(p);

  int two = quad(q);

  if (one != two)

    return (one < two);

  return (p.second * q.first < q.second * p.first);

}

// Time complexity: O(1) - constant number of operations.

// Finds upper tangent of two polygons 'a' and 'b' represented as two vectors.
```

```
vector<pair<int, int>>
merger(vector<pair<int, int>> a,
                vector<pair<int, int>> b)
{
  // n1 -> number of points in polygon a
  // n2 -> number of points in polygon b
  int n1 = a.size(), n2 = b.size();
  int ia = 0, ib = 0;
  for (int i = 1; i < n1; i++)
    if (a[i].first > a[ia].first)
      ia = i;
  // ib -> leftmost point of b
  for (int i = 1; i < n2; i++)
    if (b[i].first < b[ib].first)
      ib = i;
  // finding the upper tangent
  int inda = ia, indb = ib;
  bool done = 0;
  while (!done)
  {
    done = 1;
    while (orientation(b[indb], a[inda],
a[(inda + 1) % n1]) >= 0)
      inda = (inda + 1) % n1;
    while (orientation(a[inda], b[indb], b[(n2
+ indb - 1) % n2]) <= 0)
    {
      indb = (n2 + indb - 1) % n2;
      done = 0;
    }
  }
  int uppera = inda, upperb = indb;
  inda = ia, indb = ib;
  done = 0;
  int g = 0;
  while (!done) // finding the lower tangent
  {
    done = 1;
    while (orientation(a[inda], b[indb],
b[(indb + 1) % n2]) >= 0)
      indb = (indb + 1) % n2;
    while (orientation(b[indb], a[inda], a[(n1
+ inda - 1) % n1]) <= 0)
    {
      inda = (n1 + inda - 1) % n1;
      done = 0;
    }
  }
  int lowera = inda, lowerb = indb;
  vector<pair<int, int>> ret;
  // ret contains the convex hull after
merging the two convex hulls with the
points sorted in anti-clockwise order
  int ind = uppera;
  ret.push_back(a[uppera]);
  while (ind != lowera)
  {
    ind = (ind + 1) % n1;
    ret.push_back(a[ind]);
  }
  ind = lowerb;
  ret.push_back(b[lowerb]);
  while (ind != upperb)
  {
    ind = (ind + 1) % n2;
    ret.push_back(b[ind]);
  }
  return ret;
}
// Time complexity: O(n1 + n2) - n1 and n2
are the sizes of vectors a and b respectively.
Brute force algorithm to find convex hull for
a set of less than 6 points
vector<pair<int, int>>
bruteHull(vector<pair<int, int>> a)
{
  // Take any pair of points from the set and
check whether it is the edge of the convex
hull or not. if all the remaining points are on
the same side of the line then the line is the
edge of convex hull otherwise not
  set<pair<int, int>> s;
  for (int i = 0; i < a.size(); i++)
  {
    for (int j = i + 1; j < a.size(); j++)
    {
      int x1 = a[i].first, x2 = a[j].first;
      int y1 = a[i].second, y2 = a[j].second;
      int a1 = y1 - y2;
      int b1 = x2 - x1;
      int c1 = x1 * y2 - y1 * x2;
      int pos = 0, neg = 0;
      for (int k = 0; k < a.size(); k++)
      {
        if (a1 * a[k].first + b1 * a[k].second +
c1 <= 0)
          neg++;
        if (a1 * a[k].first + b1 * a[k].second +
c1 >= 0)
          pos++;
      }
      if (pos == a.size() || neg == a.size())
      {
        s.insert(a[i]);
        s.insert(a[j]);
      }
    }
  }
  vector<pair<int, int>> ret;
  for (auto e : s)
    ret.push_back(e);
  // Sorting the points in the anti-clockwise
order
  mid = {0, 0};
  int n = ret.size();
  for (int i = 0; i < n; i++)
  {
    mid.first += ret[i].first;
    mid.second += ret[i].second;
    ret[i].first *= n;
    ret[i].second *= n;
  }
  sort(ret.begin(), ret.end(), compare);
  for (int i = 0; i < n; i++)
    ret[i] = make_pair(ret[i].first / n,
ret[i].second / n);
  return ret;
}
// Time complexity: O(n^4) - nested loops
iterating over all possible pairs of points.
// Returns the convex hull for the given set
of points
vector<pair<int, int>>
divide(vector<pair<int, int>> a)
{
  // If the number of points is less than 6
then the function uses the brute algorithm
to find the convex hull
  if (a.size() <= 5)
    return bruteHull(a);
  // left contains the left half points
```

```
    // right contains the right half points
    vector<pair<int, int>> left, right;
    for (int i = 0; i < a.size() / 2; i++)
        left.push_back(a[i]);
    for (int i = a.size() / 2; i < a.size(); i++)
        right.push_back(a[i]);
    // convex hull for the left and right sets
    vector<pair<int, int>> left_hull =
divide(left);
    vector<pair<int, int>> right_hull =
divide(right);
    // merging the convex hulls
    return merger(left_hull, right_hull);
}
// Time complexity: O(n log n) - dividing the
set into two halves recursively.
// Driver code
int main()
{
    vector<pair<int, int>> a;
    a.push_back(make_pair(0, 0));
    a.push_back(make_pair(1, -4));
    a.push_back(make_pair(-1, -5));
    a.push_back(make_pair(-5, -3));
    a.push_back(make_pair(-3, -1));
    a.push_back(make_pair(-1, -3));
    a.push_back(make_pair(-2, -2));
    a.push_back(make_pair(-1, -1));
    a.push_back(make_pair(-2, -1));
    a.push_back(make_pair(-1, 1));
    int n = a.size();
    // sorting the set of points according to
the x-coordinate
    sort(a.begin(), a.end());
    vector<pair<int, int>> ans = divide(a);
    cout << "convex hull:\n";
    for (auto e : ans)
        cout << e.first << " "
             << e.second << endl;
    return 0;
}
// Time complexity: O(n log n) - sorting the
input points.
```

B. Dynamic Programming

1. **Word Break:**

The algorithm's time complexity varies depending on the input and the specific conditions. Its worst-case time complexity, denoted as Big O, is O(n^3), suggesting that the algorithm's running time could grow cubically with the size of the input in the worst scenario.

On the other hand, its best-case time complexity, Omega, is Ω(n), implying that the running time could be linear at its best. However, when considering both the upper and lower bounds, the tight-bound time complexity,

Theta, is Θ(n^3), indicating that the algorithm's running time generally exhibits cubic growth with the input size and is bounded above and below by cubic functions.

Big O notation (O): The Big O notation represents the upper bound or worst-case scenario of an algorithm's time complexity. In this case, the worst-case time complexity of the algorithm is O(n^3).This notation indicates that the time taken by the algorithm will not exceed a certain multiple of n^3, where n is the size of the input.Even though there might be cases where the algorithm performs better than this upper bound, O(n^3) is a guarantee that the algorithm will not take more time than what can be expressed by a cubic function of the input size.

Omega (Ω): The best-case time complexity is Ω(n), which occurs when the input string cannot be broken down into words from the dictionary in any way. In such cases, the algorithm may exit early after checking the entire input string without needing to iterate over the matched indexes.

Theta (Θ): The average-case time complexity is also Θ(n^3) for the same reasons as the worst-case scenario.

In summary:

- **Big O: O(n^3)**
- **Omega: Ω(n)**
- **Theta: Θ(n^3)**

Code:

```
#include <bits/stdc++.h>
using namespace std;
/* A utility function to check whether a word
is present in dictionary or not. An array of
strings is used for dictionary. Using array 0f
strings for dictionary is definitely not a good
idea. We have used for simplicity of the
program */
int dictionaryContains(string word)
{
    string dictionary[] = {"mobile", "samsung",
"sam", "sung", "man",
                           "mango", "icecream", "and",
"go", "i",
                           "like", "ice", "cream", "love"};
    int size = sizeof(dictionary) /
sizeof(dictionary[0]);
    for (int i = 0; i < size; i++)
        if (dictionary[i].compare(word) == 0)
            return true;
    return false;
}
// Returns true if string can be segmented
into space separated words, otherwise
returns false
bool wordBreak(string s)
{
    int n = s.size();
    if (n == 0)
        return true;
    // Create the DP table to store results of
subproblems. The value dp[i] will be true if
str[0..i] can be segmented into dictionary
words, otherwise false.
    vector<bool> dp(n + 1, 0); // Initialize all
values as false. matched_index array
represents the indexes for which dp[i] is
true. Initially only -1 element is present in
this array.
    vector<int> matched_index;
    matched_index.push_back(-1);
    for (int i = 0; i < n; i++)
    {
        int msize = matched_index.size();
        // Flag value which tells that a substring
matches with given words or not.
        int f = 0;
        // Check all the substring from the
indexes matched earlier. If any of that
substring matches than make flag value = 1;
        for (int j = msize - 1; j >= 0; j--)
        {
            // sb is substring starting from
matched_index[j] + 1 and of length i -
matched_index[j]
            string sb = s.substr(matched_index[j]
+ 1,
                                 i - matched_index[j]);
            if (dictionaryContains(sb))
            {
                f = 1;
                break;
            }
        }
        // If substring matches than do dp[i] = 1
and push that index in matched_index array.
        if (f == 1)
```

```
    {
      dp[i] = 1;
      matched_index.push_back(i);
    }
  }
  return dp[n - 1];
}
// Driver code
int main()
{
  wordBreak("ilikesamsung") ? cout << "Yes\n"
                : cout << "No\n";
  wordBreak("iiiiiiii") ? cout << "Yes\n"
              : cout << "No\n";
  wordBreak("") ? cout << "Yes\n" : cout << "No\n";
  wordBreak("ilikelikeimangoiii") ? cout << "Yes\n"
                  : cout << "No\n";
  wordBreak("samsungandmango") ? cout << "Yes\n"
                : cout << "No\n";
  wordBreak("ilove") ? cout << "Yes\n"
            : cout << "No\n";
  return 0;
}
```

2. Subset Sum:

The time complexity of the algorithm is O(n * sum), where n is the number of elements in the set and sum is the target sum. This complexity arises from the nested loops used to fill the subset[][] array

Big O (O): The algorithm iterates over all elements of the set and for each element, it iterates over all possible sums from 0 to the target sum. Therefore, the time complexity is O(n * sum), where n is the number of elements in the set and sum is the target sum. In the worst case, it is O(n * sum).

Omega (Ω): The best-case time complexity occurs when the target sum is 0. In this case, the algorithm only needs to initialize the subset table, which takes constant time. Therefore, the best-case time complexity is Ω(1)

Theta (Θ): The time complexity of O(n * sum) is also the tightest bound, so the overall time complexity can be expressed as Θ(n * sum)

In summary:

- **O(n * sum)**
- **Ω(n * sum)**
- **Θ(n * sum)**

Code:

```
#include <bits/stdc++.h> // Importing all standard C++ headers. Not recommended in production code.
using namespace std;
// Returns true if there is a subset of set[] with sum equal to given sum
bool isSubsetSum(int set[], int n, int sum)
{
  // The value of subset[i][j] will be true if there is a subset of set[0..j-1] with sum equal to i
  bool subset[n + 1][sum + 1]; // Time complexity: O(n*sum) - Creating a 2D array of size
  (n + 1) x(sum + 1).
    // If sum is 0, then answer is true
    for (int i = 0; i <= n; i++)
      subset[i][0] = true; // Time complexity: O(n) - Initializing the first column of the
  subset table.
    // If sum is not 0 and set is empty, then answer is false
    for (int i = 1; i <= sum; i++)
      subset[0][i] = false; // Time complexity: O(sum) - Initializing the first row of the
  subset table.
    // Fill the subset table in bottom up manner
    for (int i = 1; i <= n; i++)
  {
    for (int j = 1; j <= sum; j++)
    {
      if (j < set[i - 1])
        subset[i][j] = subset[i - 1][j]; // Time complexity: O(1) - Assigning the value from
      the cell above.if (j >= set[i - 1])
        subset[i][j] // Time complexity: O(1) - Assigning the value from the cell above or
        diagonal.= subset[i - 1][j] || subset[i - 1][j - set[i - 1]];
    }
  }
  return subset[n][sum]; // Time complexity: O(1) - Returning the value from the bottomright cell of the subset table.
}
// Driver code
int main()
{
  int set[] = {3, 34, 4, 12, 5, 2};
  int sum = 38;
  int n = sizeof(set) / sizeof(set[0]);
  if (isSubsetSum(set, n, sum) == true)
    cout << "Found a subset with given sum";    // Time complexity: O(1) - Output
  operation.else cout << "No subset with given sum"; // Time complexity: O(1) - Output operation.
  return 0;
}
```

3. Optimal Binary Search Tree:

The Dynamic Programming approach to solve the Optimal Binary Search Tree Problem. It uses a bottom-up approach to fill in a 2D array cost[][] to store the optimal cost of forming a binary search tree with different combinations of keys. The solution considers all possible chains of keys and computes the optimal cost by considering each key as a root node in the subtree. The time complexity of this algorithm is O(n^3), where n is the number of keys.

Big O (O): This represents the upper bound or worst-case scenario of the algorithm. In the OBST algorithm, the worst-case time complexity is O(n^3). This is because the algorithm uses nested loops to fill in the cost matrix, where the outer loop iterates over the chain length (L), and the inner loops iterate over the rows (i) and columns (j) of the cost matrix. Since there are three nested loops, the time complexity becomes cubic with respect to the number of keys (n), resulting in O(n^3).

Omega (Ω): This represents the lower bound or best-case scenario of the algorithm. In the OBST algorithm, the best-case time complexity is Ω(n^2). The best case occurs when the algorithm can quickly determine that a certain subtree is optimal and doesn't need to explore all possible combinations. However, in the worst case, the algorithm needs to consider all possible subtrees, leading to a cubic time complexity.

Theta (Θ): This represents the average-case scenario of the algorithm. In the OBST algorithm, the average-case time complexity is Θ(n^3). This indicates that, on average, the algorithm performs similarly to the worst-case scenario. Although there may be instances where the algorithm finds optimal solutions more quickly, the overall behavior remains cubic with respect to the input size.

In summary:

- **O(n^3)**
- **Ω(n^2)**
- **Θ(n^3)**

Code:

```
#include <iostream>          // O(1) - Including a standard library header
```

```
#include <climits>              // O(1) - Including a standard library header

using namespace std;            // O(1) - Using directive for the standard namespace

int sum(int freq[], int i, int j) // O(j - i) - Function to calculate the sum of elements in the

   frequency array from index i to j

{

   int s = 0;                   // O(1) - Initializing the sum variable

   for (int k = i; k <= j; k++) // O(j - i) - Looping through the elements from index i to j

      s += freq[k];             // O(1) - Adding each element to the sum

   return s;                    // O(1) - Returning the sum

}

int optimalSearchTree(int keys[], int freq[], int n) // O(n^3) - Function to find the minimum

   cost of a binary search tree

{

   int cost[n][n];              // O(n^2) - Declaring a 2D array to store the cost of forming binary search

   trees for (int i = 0; i < n; i++) // O(n) - Initializing the cost matrix for single key

      cost[i][i] = freq[i];     // O(1)

   for (int L = 2; L <= n; L++)     // O(n^2) - Looping through different chain lengths

   {

      for (int i = 0; i <= n - L + 1; i++) // O(n^2) - Looping through row numbers

      {

         int j = i + L - 1;             // O(1) - Calculating the column number

         cost[i][j] = INT_MAX;          // O(1) - Initializing the cost to maximum value

         int off_set_sum = sum(freq, i, j);  // O(j - i) - Calculating the sum of frequencies from

         i to j for (int r = i; r <= j; r++) // O(n) - Looping through interval keys[i..j]

         {

            int c = ((r > i) ? cost[i][r - 1] : 0) + // O(1)

                  ((r < j) ? cost[r + 1][j] : 0) + // O(1)

                  off_set_sum;              // O(1)

            if (c < cost[i][j])             // O(1) - Updating the cost if necessary

               cost[i][j] = c;              // O(1)

         }

      }

   }

   return cost[0][n - 1]; // O(1) - Returning the minimum cost

}

int main()

{

   int keys[] = {10, 12, 20};          // O(1) - Array of keys

   int freq[] = {34, 8, 50};           // O(1) - Array of frequencies

   int n = sizeof(keys) / sizeof(keys[0]); // O(1) - Calculating the size of the keys array

   cout << "Keys: [";

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

   {

      cout << keys[i];

      if (i != n - 1)

         cout << ", ";

   }

   cout << "] Frequencies: [";

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

   {

      cout << freq[i];

      if (i != n - 1)

         cout << ", ";

   }

   cout << "]" << endl;

   cout << "Optimal BST: (root: " << keys[n / 2];

   if (n > 1)

   {

      cout << ", left: (";

      for (int i = 0; i < n / 2; i++)

      {

         cout << keys[i];

         if (i != n / 2 - 1)

            cout << ", ";

      }

      cout << "), right: (";

      for (int i = n / 2 + 1; i < n; i++)

      {

         cout << keys[i];

         if (i != n - 1)

            cout << ", ";

      }

      cout << ")";

   }

   cout << ")" << endl;

   cout << "Cost of Optimal BST is " << optimalSearchTree(keys, freq, n);

   return 0;

   // O(n) - Printing the keys and frequencies arrays

   // O(n) - Printing the root, left, and right subtrees

   // O(n^3) - Calling optimalSearchTree function

   // O(1) - Printing the minimum cost of the optimal BST

   // O(1) - Returning 0 to indicate successful completion

}
```

4. **Longest Common Subsequence:**

The Longest Common Subsequence (LCS) problem using a top-down dynamic programming approach. It uses memoization to store the results of subproblems in a 2D vector dp, which helps avoid redundant calculations by storing the length of LCS for substrings of X and Y. The function lcs recursively calculates the LCS length for the given strings X and Y, and returns the result. This approach is a classic example of dynamic programming, where the problem is broken down into smaller subproblems, and the solutions to these subproblems are stored and reused to compute the final solution efficiently.

Big O (O): The time complexity of this implementation is O(mxn) where m is the length of string X and n is the length of string Y. This is because the function lcs can potentially be called with m x n different combinations of m and n, and each call has constant time complexity due to memoization.

Omega (Ω): The best-case time complexity is also O(m×n). Even in the best-case scenario, where there are no common characters between the strings X and Y, the algorithm still needs to fill the entire memoization table dp to compute the LCS length.

Theta (Θ): The average-case time complexity is also O(m×n), as there is no significant difference between the best and worst cases.

So in summary:

- **O(mxn)**
- **Ω (m×n).**
- **Θ(m×n)**

Code:

```
#include <bits/stdc++.h>
// O(1) - Including a standard library header
```

```
using namespace std;
// O(1) - Using directive for the standard namespace

int lcs(char *X, char *Y, int m, int n, vector<vector<int>> &dp) // O(m*n) - Function to find the length of the Longest Common Subsequence(LCS)

{

   if (m == 0 || n == 0) // O(1) - Base case: if either of the strings is empty, LCS is 0

      return 0;

   if (X[m - 1] == Y[n - 1])                    // O(1) - If the last characters of both strings match

      return dp[m][n] = 1 + lcs(X, Y, m - 1, n - 1, dp); // O(1) + Recursive call

   if (dp[m][n] != -1)

   {              // O(1) - If the value is already computed

      return dp[m][n]; // O(1) - Return the stored value

   }

   return dp[m][n] = max(lcs(X, Y, m, n - 1, dp),  // O(1) + Recursive call

                  lcs(X, Y, m - 1, n, dp)); // O(1) + Recursive call

}

int main()

{

   char X[] = "AGGTAB";                        // O(1) - Initializing string X

   char Y[] = "GXTXAYB";                       // O(1) - Initializing string Y

   int m = strlen(X);                          // O(m) - Calculating the length of string X

   int n = strlen(Y);                          // O(n) - Calculating the length of string Y

   vector<vector<int>> dp(m + 1, vector<int>(n + 1, -1)); // O(m*n) - Initializing a 2D vector for memoization

cout << "Length of LCS is " << lcs(X, Y, m, n, dp); // O(m*n) - Calling the lcs function and printing the length of LCS return 0;   // O(1) - Returning 0 to indicate successful completion

}
```

5. Rod Cutting:

The rod cutting problem uses bottom-up dynamic programming to efficiently find the maximum obtainable value for cutting a rod of length n into different pieces, given the prices for each piece length.

Big O (O): The time complexity of the algorithm is O(n^2), where n is the length of the rod. This is because the algorithm involves nested loops, one iterating over the length of the rod (n), and the other iterating up to the current length (i). Since each iteration of the inner loop takes constant time, the overall time complexity is O(n^2).

Omega (Ω): The best-case time complexity is also Ω(n^2). Even if the input is such that the rod length is smaller, the algorithm still iterates through the nested loops, resulting in Ω(n^2).

Theta (Θ): Therefore, the tight bound or Theta notation of the algorithm is Θ(n^2), indicating that the time complexity is quadratic in terms of the length of the rod.

In summary:

- **O(n^2)**
- **Ω(n^2)**
- **Θ(n^2)**

Code:

```
#include <iostream>     // O(1) - Including a standard I/O library

#include <bits/stdc++.h> // O(1) - Including a standard library for utilities like INT_MIN

#include <math.h>       // O(1) - Including a standard math library


using namespace std; // O(1) - Using directive for the standard namespace


// A utility function to get the maximum of two integers

int max(int a, int b) { return (a > b) ? a : b; } // O(1) - Function to return the maximum of two integers

/* Returns the best obtainable price for a rod of length n and price[] as prices of different pieces */

int cutRod(int price[], int n) // O(n^2) - Function to find the maximum obtainable value for a rod of length n

{

   int val[n + 1]; // O(n) - Array to store the maximum obtainable value for rods of different lengths

   val[0] = 0;     // O(1) - Base case: If the length of the rod is 0, the maximum obtainable value is 0 int i, j;

   // Build the table val[] in bottom-up manner and return the last entry from the table


   for (i = 1; i <= n; i++) // O(n)

   {

      int max_val = INT_MIN;                   // O(1) - Initializing the maximum value to a very small integer for (j = 0; j < i; j++)
// O(n)

      max_val = max(max_val, price[j] + val[i - j - 1]); // O(1) - Updating the maximum value

      val[i] = max_val;                        // O(1) - Storing the maximum value for rod of length i

   }

   return val[n]; // O(1) - Returning the maximum obtainable value for the rod of length n

}


/* Driver program to test above functions */

int main()

{

   int arr[] = {1, 5, 8, 9, 10, 17, 17, 20}; // O(1) - Initializing an array with prices for different lengths of rod int size = sizeof(arr) / sizeof(arr[0]);

   // O(1) - Calculating the size of the array

   cout << "Maximum Obtainable Value is " << cutRod(arr, size); // O(n^2) - Calling the cutRod function and printing the result

   getchar();                                  // O(1) - Pausing the console

   return 0;                                   // O(1) - Indicating successful completion of the program

}
```

C. Greedy Algorithm

1. Huffman Coding:

The Huffman coding algorithm, which is indeed a greedy algorithm. It constructs an optimal prefix binary tree (Huffman tree) by repeatedly selecting and merging the two nodes with the lowest frequencies until only one node remains, which becomes the root of the Huffman tree. This process is inherently greedy because at each step, it chooses the locally optimal solution (merging the two lowest frequency nodes) with the aim of achieving the globally optimal solution (minimizing the total encoding length).

Big O (Upper Bound): In the worst case, where all characters have different frequencies, the algorithm will require creating and maintaining a priority queue (min heap) of all characters and their frequencies. The construction of this priority queue takes O(n) time, where n is the number of unique characters. Additionally, the algorithm involves repeatedly extracting the two minimum frequency nodes and merging them until only one node remains. This process requires n-1 iterations, each involving heapifying the min heap, which takes O(log n) time. Therefore, the overall time complexity is O(n log n), where n is the number of unique characters.

Omega (Lower Bound): The lower bound is determined by the same factors as the upper bound, as there are no best-case scenarios that would significantly improve the time complexity. Therefore, the Omega notation is also O(n log n).

Theta (Tight Bound): Since the upper and lower bounds are the same, the tight bound (Theta notation) is also O(n log n).

In Summary:

- **O(nlogn)**
- **Ω(nlogn)**
- **Θ(nlogn)**

Code:

```
#include <stdio.h>     // O(1) - Including a standard I/O library

#include <stdlib.h>    // O(1) - Including a standard library for utilities like malloc

#define MAX_TREE_HT 100 // O(1) - Defining a constant for maximum tree height

struct MinHeapNode

{

   char data;                  // O(1) - One of the input characters

   unsigned freq;              // O(1) - Frequency of the character

   struct MinHeapNode *left, *right; // O(1) - Left and right child of this node

};

struct MinHeap

{

   unsigned size;          // O(1) - Current size of min heap

   unsigned capacity;      // O(1) - Capacity of min heap

   struct MinHeapNode **array; // O(1) - Array of minheap node pointers

};

// A utility function to allocate a new min heap node with given character and frequency

struct MinHeapNode *newNode(char data, unsigned freq) // O(1)

{

   struct MinHeapNode *temp = (struct MinHeapNode *)malloc(sizeof(struct


MinHeapNode)); // O(1)

   temp->left = temp->right = NULL;
// O(1) - Initializing left and right child as NULL

   temp->data = data;
// O(1) - Assigning data to the node

   temp->freq = freq;
// O(1) - Assigning frequency to the node

   return temp;
// O(1) - Returning the newly created node

}

// A utility function to swap two min heap nodes

void swapMinHeapNode(struct MinHeapNode **a, struct MinHeapNode **b) // O(1)

{

   struct MinHeapNode *t = *a; // O(1)

   *a = *b;              // O(1)

   *b = t;               // O(1)

}

// The standard minHeapify function

void minHeapify(struct MinHeap *minHeap, int idx) // O(log n)

{

   int smallest = idx;
// O(1)

   int left = 2 * idx + 1;
// O(1)

   int right = 2 * idx + 2;
// O(1)

   if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq)  // O(1)

      smallest = left;
// O(1)

   if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq) // O(1)

      smallest = right;
// O(1)

   if (smallest != idx)

   {                                          // O(1)

      swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]); // O(1)

      minHeapify(minHeap, smallest);
// O(log n) - Recursive call

   }

}

// A utility function to check if size of heap is 1 or not

int isSizeOne(struct MinHeap *minHeap) // O(1)

{

   return (minHeap->size == 1); // O(1)

}

// A standard function to extract minimum value node from heap

struct MinHeapNode *extractMin(struct MinHeap *minHeap) // O(log n)

{

   struct MinHeapNode *temp = minHeap->array[0];       // O(1)

   minHeap->array[0] = minHeap->array[minHeap->size - 1]; // O(1)

   --minHeap->size;                          // O(1)

   minHeapify(minHeap, 0);                   // O(log n)

   return temp;                              // O(1)

}

// A utility function to insert a new node to Min Heap

void insertMinHeap(struct MinHeap *minHeap, struct MinHeapNode *minHeapNode) //

   O(log n)

{

   ++minHeap->size;        // O(1)

   int i = minHeap->size - 1; // O(1)

   while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq)

   {                                  // O(log n)

      minHeap->array[i] = minHeap->array[(i - 1) / 2]; // O(1)

      i = (i - 1) / 2;                // O(1)

   }

   minHeap->array[i] = minHeapNode; // O(1)

}

// A standard function to build min heap

void buildMinHeap(struct MinHeap *minHeap) // O(n)

{

   int n = minHeap->size - 1;       // O(1)

   int i;                           // O(1)

   for (i = (n - 1) / 2; i >= 0; --i) // O(n)

      minHeapify(minHeap, i);       // O(log n)

}

// A utility function to print an array of size n

void printArr(int arr[], int n) // O(n)

{

   int i;              // O(1)

   for (i = 0; i < n; ++i)  // O(n)

      printf("%d", arr[i]); // O(1)

   printf("\n");           // O(1)

}

// A utility function to check if this node is leaf

int isLeaf(struct MinHeapNode *root) // O(1)

{

   return !(root->left) && !(root->right); // O(1)
```

```
}

// Creates a min heap of capacity equal to size and inserts all character of data[] in min heap.

   // Initially, the size of the min heap is equal to capacity

   struct MinHeap *

   createAndBuildMinHeap(char data[], int freq[], int size) // O(n)

{

   struct MinHeap *minHeap = (struct MinHeap *)malloc(sizeof(struct MinHeap)); // O(1)

   minHeap->size = 0;
// O(1)

   minHeap->capacity = size;
// O(1)

   minHeap->array = (struct MinHeapNode **)malloc(minHeap->capacity * sizeof(struct


MinHeapNode *)); // O(1)

   for (int i = 0; i < size; ++i)
// O(n)

      minHeap->array[i] = newNode(data[i], freq[i]);                    // O(1)

   minHeap->size = size;
// O(1)

   buildMinHeap(minHeap);
// O(n)

   return minHeap;
// O(1)

}

// The main function that builds Huffman tree

struct MinHeapNode *buildHuffmanTree(char data[], int freq[], int size) // O(n log n)

{

   struct MinHeapNode *left, *right, *top;
// O(1)

   struct MinHeap *minHeap = createAndBuildMinHeap(data, freq, size); // O(n)

   while (!isSizeOne(minHeap))

   {                                    // O(n)

      left = extractMin(minHeap);               // O(log n)

      right = extractMin(minHeap);              // O(log n)

      top = newNode('$', left->freq + right->freq); // O(1)

      top->left = left;                    // O(1)

      top->right = right;                  // O(1)

      insertMinHeap(minHeap, top);
// O(log n)

   }

   return extractMin(minHeap); // O(log n)

}

// Prints huffman codes from the root of Huffman Tree. It uses arr[] to store codes

void printCodesUtil(struct MinHeapNode *root, int arr[], int top) // O(n)

{

   if (root->left)

   {                                    // O(1)

      arr[top] = 0;                     // O(1)

      printCodesUtil(root->left, arr, top + 1); // O(n)

   }

   if (root->right)

   {                                    // O(1)

      arr[top] = 1;                     // O(1)

      printCodesUtil(root->right, arr, top + 1);
// O(n)

   }

   if (isLeaf(root))

   {                        // O(1)

      printf("%c: ", root->data); // O(1)

      printArr(arr, top);       // O(n)

   }

}

// The main function that builds a Huffman Tree and prints codes by traversing the built Huffman Tree

void HuffmanCodes(char data[], int freq[], int size) // O(n log n)

{

   struct MinHeapNode *root = buildHuffmanTree(data, freq, size); // O(n log n)

   int arr[MAX_TREE_HT], top = 0;
// O(1)

   printCodesUtil(root, arr, top);
// O(n)

}



// Driver code

int main() // O(n log n)

{

   char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'}; // O(1)

   int freq[] = {5, 9, 12, 13, 16, 45};       // O(1)

   int size = sizeof(arr) / sizeof(arr[0]);    // O(1)

   HuffmanCodes(arr, freq, size);             // O(n log n)

   return 0;                          // O(1)

}
```

2. Activity Selection Problem:

The activity selection problem, which aims to find the maximum number of nonoverlapping activities that can be performed given a set of activities, each characterized by their start and finish times.

Big O (O-notation): It represents the upper bound or worst-case scenario of the algorithm's time complexity. It provides an estimate of how the algorithm's runtime grows as the input size increases. For the provided algorithm, the time complexity is O(n log n) because the sorting step dominates the runtime, and sorting typically requires O(n log n) time in the worst case. In simple terms, O(n log n) means that the algorithm's runtime grows proportionally to n multiplied by the logarithm of n.

Big Omega (Ω-notation): It signifies the lower bound or best-case scenario of the algorithm's time complexity. It denotes the minimum time required by the algorithm for any input size. In this case, the best-case scenario also occurs during sorting, which still requires O(n log n) time. Therefore, the Big Omega notation for the algorithm remains Ω(n log n). Essentially, Ω(n log n) indicates that the algorithm's runtime will not be faster than n log n, even in the best-case scenario.

Big Theta (Θ-notation): It represents both the upper and lower bounds of the algorithm's time complexity, providing a tight estimate of its behavior. Since the Big O and Big Omega notations for the algorithm are both Θ(n log n), it means that the algorithm's runtime is bounded both from above and below by n log n. In other words, the algorithm's time complexity is precisely proportional to n log n for all input sizes, considering both the best and worst-case scenarios.

In summary:

- **O(n log n)**
- **Ω(n log n)**
- **Θ(n log n)**

Code:

```
#include <iostream>  // O(1)

#include <algorithm> // O(1)

using namespace std; // O(1)

// Structure to represent an activity

struct Activity

{                // O(1)

   int start, finish; // O(1)

};
```

```
// Comparison function to sort the activities based on their finish time

bool activityCompare(Activity s1, Activity s2)

{                       // O(1)

   return (s1.finish < s2.finish); // O(1)

}

// Function to print the maximum number of activities that can be performed

void printMaxActivities(Activity arr[], int n)

{                                       // O(1)

   sort(arr, arr + n, activityCompare);
// O(n log n) - Sorting the activities based on finish

   time int i = 0;                      // O(1)

   cout << "The following activities are selected: " << endl; // O(1)

   // Select the first activity

   cout << "(" << arr[i].start << ", " << arr[i].finish << ") "; // O(1)

   // Iterate through the rest of the activities

   for (int j = 1; j < n; j++)

   { // O(n) - Iterating through the activities array

      // If this activity has a start time greater than or equal to the finish time of the previous activity, select it

      if (arr[j].start >= arr[i].finish)

      {                                 // O(1)

         cout << "(" << arr[j].start << ", " << arr[j].finish << ") "; // O(1)

         i = j;                         // O(1)

      }

   }

}

int main()

{                                       // O(1)

   Activity arr[] = {{5, 9}, {1, 2}, {3, 4}, {0, 6}, {5, 7}, {8, 9}}; // O(1)

   int n = sizeof(arr) / sizeof(arr[0]);
// O(1)

   printMaxActivities(arr, n);
// O(n log n) - Sorting the activities

   return 0;                            // O(1)

}
```

3. Dijkstra's Algorithm:

The graph is represented using an adjacency matrix, as in the provided code. If the graph is represented using other data structures like adjacency lists, the time complexity may vary.

Best Case (Omega): In the best-case scenario, Dijkstra's algorithm may terminate early if the destination vertex is encountered before visiting all vertices. In this case, the time complexity would be O(1), as it would only need to process the source vertex.

Worst Case (Big O): In the worst-case scenario, Dijkstra's algorithm iterates over all vertices and edges to find the shortest path from the source vertex to all other vertices. The main loop runs V times, and for each iteration, it needs to select the vertex with the minimum distance (which requires O(V) time) and update the distances of its adjacent vertices (which requires visiting all edges). Therefore, the worst-case time complexity is O(V^2).

Average Case (Theta): The average-case time complexity of Dijkstra's algorithm is also O(V^2) because it iterates over all vertices and edges similarly to the worst case.

In summary:

- **Best Case (Omega): O(1)**
- **Worst Case (Big O): O(V^2)**
- **Average Case (Theta): O(V^2)**

Code:

```
#include <iostream>  // O(1)

using namespace std; // O(1)

#include <limits.h>  // O(1)


// Number of vertices in the graph

#define V 9 // O(1)

// A utility function to find the vertex with minimum distance value, from the set of vertices not yet included in shortest path tree

int minDistance(int dist[], bool sptSet[]) // O(V)

{

   // Initialize min value

   int min = INT_MAX, min_index;            // O(1)

   for (int v = 0; v < V; v++)              // O(V)

      if (sptSet[v] == false && dist[v] <= min) // O(1)

         min = dist[v], min_index = v;      // O(1)

   return min_index;                        // O(1)

}

// A utility function to print the constructed distance array

void printSolution(int dist[]) // O(V)

{

   cout << "Vertex \t Distance from Source" << endl; // O(1)

   for (int i = 0; i < V; i++)              // O(V)

      cout << i << " \t\t\t\t" << dist[i] << endl; // O(1)

}

// Function that implements Dijkstra's single source shortest path algorithm for a graph represented using adjacency matrix representation

void dijkstra(int graph[V][V], int src) // O(V^2)

{

   int dist[V];   // The output array. dist[i] will hold the shortest distance from src to i

   bool sptSet[V]; // sptSet[i] will be true if vertex i is

   // included in shortest path tree or shortest distance from src to i is finalized

   // Initialize all distances as INFINITE and stpSet[] as false

   for (int i = 0; i < V; i++)          // O(V)

      dist[i] = INT_MAX, sptSet[i] = false; // O(1)

   // Distance of source vertex from itself is always 0

   dist[src] = 0; // O(1)

   // Find shortest path for all vertices

   for (int count = 0; count < V - 1; count++)

   { // O(V)

      // Pick the minimum distance vertex from the set of vertices not yet processed. u is always equal to src in the first iteration.

      int u = minDistance(dist, sptSet); // O(V)

      // Mark the picked vertex as processed

      sptSet[u] = true; // O(1)

      // Update dist value of the adjacent vertices of the picked vertex.

      for (int v = 0; v < V; v++) // O(V)

      {

         // Update dist[v] only if is not in sptSet, there is an edge from u to v, and total weight of path from src to v through u is smaller than current value of dist[v]

         if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v]) // O(1)

            dist[v] = dist[u] + graph[u][v]; // O(1)

      }

   }

   // print the constructed distance array

   printSolution(dist); // O(V)

}

// driver's code
```

```
int main() // O(1)
{
  /* Let us create the example graph discussed above */
  int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                     {4, 0, 8, 0, 0, 0, 0, 11, 0},
                     {0, 8, 0, 7, 0, 4, 0, 0, 2},
                     {0, 0, 7, 0, 9, 14, 0, 0, 0},
                     {0, 0, 0, 9, 0, 10, 0, 0, 0},
                     {0, 0, 4, 14, 10, 0, 2, 0, 0},
                     {0, 0, 0, 0, 0, 2, 0, 1, 6},
                     {8, 11, 0, 0, 0, 0, 1, 0, 7},
                     {0, 0, 2, 0, 0, 0, 6, 7, 0}};
  // Function call
  dijkstra(graph, 0); // O(V^2)
  return 0;         // O(1)
}
```

4. Fractional Knapsack Problem:

Best Case (Omega): O(N log N) - This occurs when the items are already sorted in nondecreasing order of their profit-to-weight ratio, allowing the algorithm to skip the sorting step and proceed directly to selecting items greedily, but most cases O(N log N).

Worse Case (O): O(N log N) - This scenario arises when the items are sorted in nonincreasing order of their profit-to-weight ratio. The dominant factor in determining the time complexity is the sorting step, which has a worst-case time complexity of O(N log N) using comparison-based sorting algorithms.

Average Case(Theta): O(N log N) - In the average case, assuming a random distribution of input data, the time complexity closely resembles the worst-case scenario due to the sorting step's dominance.

In Summary:

- **Ω(N log N)**
- **O(N log N)**
- **Θ(N log N)**

Code:

```
#include <bits/stdc++.h> // O(1)
using namespace std;    // O(1)

// Structure for an item which stores weight and corresponding value of Item
struct Item
{                 // O(1)
  int profit, weight; // O(1)
  // Constructor
  Item(int profit, int weight) // O(1)
  {
    this->profit = profit; // O(1)
    this->weight = weight; // O(1)
  }
};

// Comparison function to sort Item according to profit/weight ratio
static bool cmp(struct Item a, struct Item b) // O(1)
{
  double r1 = (double)a.profit / (double)a.weight; // O(1)
  double r2 = (double)b.profit / (double)b.weight; // O(1)
  return r1 > r2;                  // O(1)
}

// Main greedy function to solve problem
double fractionalKnapsack(int W, struct Item arr[], int N) // O(N log N)
{
  // Sorting Item on basis of ratio
  sort(arr, arr + N, cmp); // O(N log N)
  double finalvalue = 0.0; // O(1)
  // Looping through all items
  for (int i = 0; i < N; i++)
  { // O(N)

    // If adding Item won't overflow, add it completely
    if (arr[i].weight <= W)
    {                  // O(1)
      W -= arr[i].weight;      // O(1)
      finalvalue += arr[i].profit; // O(1)
    }
    // If we can't add current Item, add fractional part of it
    else
    {
      finalvalue += arr[i].profit * ((double)W / (double)arr[i].weight); // O(1)
      break;                                  // O(1)
    }
  }
  // Returning final value
  return finalvalue; // O(1)
}

// Driver code
int main() // O(1)
{
  int W = 50;                  // O(1)
  Item arr[] = {{60, 10}, {100, 20}, {120, 30}}; // O(1)
  int N = sizeof(arr) / sizeof(arr[0]);    // O(1)
  // Function call
  cout << fractionalKnapsack(W, arr, N); // O(N log N)
  return 0;                  // O(1)
}
```

5. Job Scheduling with Deadlines:

The greedy aspect of the algorithm lies in the selection of jobs. At each step, it selects the job with the highest profit that can be accommodated within its deadline. This approach ensures that the most profitable jobs are scheduled first, potentially leading to an optimal or near-optimal solution.

Big O (O): The algorithm involves sorting the jobs based on their profits, which takes O(nlogn) time, where n is the number of jobs. Then, for each job, it iterates through the time slots to find the latest available slot, which takes O(n2) time in the worst case scenario. Therefore, the overall time complexity is dominated by the sorting step, making it O(nlogn).

Omega (Ω): The best-case time complexity occurs when the number of distinct deadlines is much smaller than the total number of jobs. In this case, the time complexity can be closer to linear, but it still depends on the sorting step, which requires at least O(nlogn) comparisons. Therefore, the lower bound or Omega (ΩΩ) of the algorithm remains O(nlogn).

Theta (Θ): Since the upper bound (Big O) and lower bound (Omega) of the algorithm are the same O(nlogn), the time complexity of the algorithm can be expressed as Θ(nlogn). This means that the algorithm's time complexity grows at the same rate as nlogn, both in the best and worst-case scenarios.

In Summary:

- **Ω(N log N)**
- **O(N log N)**
- **Θ(N log N)**

Code:

```
#include <stdbool.h> // O(1)
#include <stdio.h>   // O(1)
#include <stdlib.h>  // O(1)

// A structure to represent a job
typedef struct Job
```

```
{           // O(1)
   char id;   // Job Id
   int dead;  // Deadline of job
   int profit; // Profit if job is over before or on deadline
} Job;

// This function is used for sorting all jobs according to profit
int compare(const void *a, const void *b) // O(1)
{
   Job *temp1 = (Job *)a;              // O(1)
   Job *temp2 = (Job *)b;              // O(1)
   return (temp2->profit - temp1->profit); // O(1)
}

// Find minimum between two numbers.
int min(int num1, int num2) // O(1)
{
   return (num1 > num2) ? num2 : num1; // O(1)
}

// Returns maximum profit from jobs
void printJobScheduling(Job arr[], int n) // O(n^2)
{
   // Sort all jobs according to decreasing order of profit
   qsort(arr, n, sizeof(Job), compare); // O(n*log(n))
   int result[n];                  // To store result (Sequence of jobs)
   bool slot[n];                   // To keep track of free time slots
   // Initialize all slots to be free
   for (int i = 0; i < n; i++) // O(n)
      slot[i] = false;      // O(1)
   // Iterate through all given jobs
   for (int i = 0; i < n; i++)
   { // O(n)

      // Find a free slot for this job (Note that we start from the last possible slot)
      for (int j = min(n, arr[i].dead) - 1; j >= 0; j--)
      { // O(n)

         // Free slot found
         if (slot[j] == false)
         {              // O(1)
            result[j] = i;  // Add this job to result
            slot[j] = true; // Make this slot occupied
            break;       // O(1)
         }
      }
   }
   // Print the result
   for (int i = 0; i < n; i++)           // O(n)
      if (slot[i])                    // O(1)
         printf("%c ", arr[result[i]].id); // O(1)
}

// Driver's code
int main() // O(1)
{
   Job arr[] = {{'a', 2, 100},
                {'b', 1, 19},
                {'c', 2, 27},
                {'d', 1, 25},
                {'e', 3, 15}};
   int n = sizeof(arr) / sizeof(arr[0]); // O(1)
   printf(
      "Following is maximum profit sequence of jobs \n");
   // Function call
   printJobScheduling(arr, n); // O(n^2)
   return 0;              // O(1)
}
```

D. Network Flow and Matching

1. Ford-Fulkerson Algorithm:

The time complexity of the Ford-Fulkerson algorithm is O(V·E), where V is the number of vertices and E is the number of edges in the graph.

Big O (O): In the worst-case scenario, the BFS function traverses through all the vertices and edges in the graph, which takes O(V+E) time, where V is the number of vertices and E is the number of edges. The main loop in the fordFulkerson function may iterate multiple times, but in the worst case, it runs O(V·E) times because each iteration of the loop may reduce the residual capacity of at least one edge.

Therefore, the overall time complexity of the Ford-Fulkerson algorithm is O(V^3* E), where V is the number of vertices and E is the number of edges.

Omega (Ω): In the best-case scenario, the BFS function may find a path from the source to the sink in a single iteration, which would take O(V+E) time. However, this scenario is rare, and in most cases, the algorithm would require multiple iterations of the main loop. Therefore, the best-case time complexity is still O(V^3* E).

Theta (Θ): Since the upper bound (Big O) and lower bound (Omega) of the algorithm are the same O(V·E)), the time complexity of the Ford-Fulkerson algorithm can be expressed as Θ(V^3* E). This means that the time complexity of the algorithm grows at the same rate as V·E in both the best and worst-case scenarios.

In Summary:

- **Ω(V^3* E)**
- **O(V^3 * E)**
- **Θ(V^3 * E)**

Code:

```
#include <iostream> // O(1)
#include <limits.h> // O(1)
#include <queue>    // O(1)
#include <string.h> // O(1)

using namespace std;

// Number of vertices in given graph
#define V 6 // O(1)

/* Returns true if there is a path from source 's' to sink 't' in residual graph. Also fills parent[] to store the path */
bool bfs(int rGraph[V][V], int s, int t, int parent[]) // O(V^2)
{
   // Create a visited array and mark all vertices as not visited
   bool visited[V];                // O(V)
   memset(visited, 0, sizeof(visited)); // O(V)
   // Create a queue, enqueue source vertex and mark source vertex as visited
   queue<int> q;     // O(1)
   q.push(s);        // O(1)
   visited[s] = true; // O(1)
   parent[s] = -1;   // O(1)

   // Standard BFS Loop
   while (!q.empty())
   {              // O(V)
```

```
    int u = q.front(); // O(1)
    q.pop();        // O(1)
    for (int v = 0; v < V; v++)
    { // O(V)
      if (visited[v] == false && rGraph[u][v] > 0)
      { // O(1)
        // If we find a connection to the sink node, then there is no point in BFS anymore We just have to set its parent and can return true
        if (v == t)
        {            // O(1)
          parent[v] = u; // O(1)
          return true;  // O(1)
        }
        q.push(v);      // O(1)
        parent[v] = u;    // O(1)
        visited[v] = true; // O(1)
      }
    }
  }
  // We didn't reach sink in BFS starting from source, so return false
  return false; // O(1)
}

// Returns the maximum flow from s to t in the given graph
int fordFulkerson(int graph[V][V], int s, int t) // O(V^3 * E)
{
  int u, v;
  // Create a residual graph and fill the residual graph with given capacities in the original graph as residual capacities in residual graph
  int rGraph[V]
      [V]; // Residual graph where rGraph[i][j]
  // indicates residual capacity of edge from i to j (if there is an edge. If rGraph[i][j] is 0, then there is not)
  for (u = 0; u < V; u++)         // O(V^2)
    for (v = 0; v < V; v++)       // O(V^2)
      rGraph[u][v] = graph[u][v]; // O(1)
  int parent[V];                  // This array is filled by BFS and to
  // store path
  int max_flow = 0; // There is no flow initially
  // Augment the flow while there is path from source to sink
  while (bfs(rGraph, s, t, parent))
  { // O(V^2 * E)
    // Find minimum residual capacity of the edges along the path filled by BFS. Or we can say find the maximum flow through the path found.
    int path_flow = INT_MAX; // O(1)
    for (v = t; v != s; v = parent[v])
    {                               // O(V)
      u = parent[v];                // O(1)
      path_flow = min(path_flow, rGraph[u][v]); // O(1)
    }
    // update residual capacities of the edges and
    // reverse edges along the path
    for (v = t; v != s; v = parent[v])
    {                   // O(V)
      u = parent[v];    // O(1)
      rGraph[u][v] -= path_flow; // O(1)
      rGraph[v][u] += path_flow; // O(1)
    }
    // Add path flow to overall flow
    max_flow += path_flow; // O(1)
  }
  // Return the overall flow
  return max_flow; // O(1)
}
// Driver program to test above functions
int main() // O(1)
{
  // Let us create a graph shown in the above example
  int graph[V][V] = {{0, 16, 13, 0, 0, 0}, {0, 0, 10, 12, 0, 0}, {0, 4, 0, 0, 14, 0}, {0, 0, 9, 0, 0, 20}, {0, 0, 0, 7, 0, 4}, {0, 0, 0, 0, 0, 0}};
  cout << "The maximum possible flow is "
    << fordFulkerson(graph, 0, 5); // O(V^3 * E)
  return 0;                  // O(1)
}
```

2. Minimum Cost Flow Algorithm:

the time complexity of the provided code for the successive shortest path algorithm is O(V2·E2).

Big O (O): The main loop of the algorithm iterates until no augmenting path from the source to the sink can be found. In the worst case, this loop may iterate O(V·E) times, where V is the number of vertices and E is the number of edges in the graph. Inside the loop, the search function performs a modified version of the Bellman-Ford algorithm, which takes O(V·E) time in the worst case. Therefore, the overall time complexity of the algorithm is O(V2·E2).

Omega (Ω): In the best-case scenario, the algorithm finds the maximum flow with minimum cost in a single iteration of the main loop. This typically occurs when the graph has a simple structure and the flow can be easily routed from the source to the sink without much exploration. Therefore, the best-case time complexity is O(V·E).

Theta (Θ): Since the upper bound (Big O) and lower bound (Omega) of the algorithm are the same O(V2·E2)), the time complexity of the successive shortest path algorithm can be expressed as Θ(V2·E2). This means that the time complexity of the algorithm grows at the same rate as V2·E2 in both the best and worst-case scenarios.

In Summary:

- **Ω(V* E)**
- **O(V^2 * E)**
- **Θ(V^2 * E)**

Code:

```
#include <iostream> // O(1)
#include <limits.h> // O(1)
#include <queue>    // O(1)
#include <string.h> // O(1)
using namespace std;
// Number of vertices in given graph

#define V 6 // O(1)
/* Returns true if there is a path from source 's' to sink 't' in residual graph. Also fills parent[] to store the path */
bool bfs(int rGraph[V][V], int s, int t, int parent[]) // O(V^2)
{
  // Create a visited array and mark all vertices as not visited
  bool visited[V];                // O(V)
  memset(visited, 0, sizeof(visited)); // O(V)
  // Create a queue, enqueue source vertex and mark source vertex as visited
  queue<int> q;     // O(1)
  q.push(s);        // O(1)
  visited[s] = true; // O(1)
  parent[s] = -1;   // O(1)
  // Standard BFS Loop
```

```
  while (!q.empty())
  {                 // O(V)
    int u = q.front(); // O(1)
    q.pop();        // O(1)
    for (int v = 0; v < V; v++)
    { // O(V)
      if (visited[v] == false && rGraph[u][v] > 0)
      { // O(1)
        // If we find a connection to the sink node, then there is no point in BFS anymore We just have to set its parent and can return true
        if (v == t)
        {             // O(1)
          parent[v] = u; // O(1)
          return true;  // O(1)
        }
        q.push(v);      // O(1)
        parent[v] = u;    // O(1)
        visited[v] = true; // O(1)
      }
    }
  }
  // We didn't reach sink in BFS starting from source, so return false
  return false; // O(1)
}
// Returns the maximum flow from s to t in the given graph
int fordFulkerson(int graph[V][V], int s, int t) // O(V^3 * E)
{
  int u, v;
  // Create a residual graph and fill the residual graph with given capacities in the original graph as residual capacities in residual graph
  int rGraph[V]
      [V]; // Residual graph where rGraph[i][j]
  // indicates residual capacity of edge from i to j (if there is an edge. If rGraph[i][j] is 0, then there is not)
  for (u = 0; u < V; u++)          // O(V^2)
    for (v = 0; v < V; v++)       // O(V^2)
      rGraph[u][v] = graph[u][v]; // O(1)
  int parent[V];                  // This array is filled by BFS and to
  // store path
  int max_flow = 0; // There is no flow initially
  // Augment the flow while there is path from source to sink
  while (bfs(rGraph, s, t, parent))
  { // O(V^2 * E)
    // Find minimum residual capacity of the edges along the path filled by BFS. Or we can say find the maximum flow through the path found.
    int path_flow = INT_MAX; // O(1)
    for (v = t; v != s; v = parent[v])
    {                              // O(V)
      u = parent[v];               // O(1)
      path_flow = min(path_flow, rGraph[u][v]); // O(1)
    }
    // update residual capacities of the edges and reverse edges along the path
    for (v = t; v != s; v = parent[v])
    {                    // O(V)
      u = parent[v];     // O(1)
      rGraph[u][v] -= path_flow; // O(1)
      rGraph[v][u] += path_flow; // O(1)
    }
    // Add path flow to overall flow
    max_flow += path_flow; // O(1)
  }
  // Return the overall flow
  return max_flow; // O(1)
}

// Driver program to test above functions
int main() // O(1)
{
  // Let us create a graph shown in the above example
  int graph[V][V] = {{0, 16, 13, 0, 0, 0}, {0, 0, 10, 12, 0, 0}, {0, 4, 0, 0, 14, 0}, {0, 0, 9, 0, 0, 20}, {0, 0, 0, 7, 0, 4}, {0, 0, 0, 0, 0, 0}};
  cout << "The maximum possible flow is "
     << fordFulkerson(graph, 0, 5); // O(V^3 * E)
  return 0;                  // O(1)
}
```

3. Circulation Network:

the time complexity of the provided code for finding the minimum flow in a negative weight cycle using Bellman-Ford's algorithm is O(V·E).

Big O (O): The main part of the algorithm involves running Bellman-Ford's algorithm, which has a time complexity of O(V·E), where V is the number of vertices and E is the number of edges in the graph. In the worst case, the algorithm iterates over all edges in the graph V times.

Omega (Ω): The best-case time complexity occurs when no negative weight cycles are found in the graph, and the algorithm completes after a single iteration of Bellman-Ford's algorithm. Therefore, the best-case time complexity is also O(V·E).

Theta (Θ): Since the upper bound (Big O) and lower bound (Omega) of the algorithm are the same O(V·E)), the time complexity of the provided code can be expressed as Θ(V·E). This means that the time complexity of the algorithm grows at the same rate as V·E in both the best and worst-case scenarios.

In Summary:

- **Ω(V* E)**
- **O(V * E)**
- **Θ(V * E)**

Code:

```
#include <iostream>     // O(1)
#include <vector>       // O(1)
#include <limits>       // O(1)
#include <unordered_map> // O(1)
#include <algorithm>    // O(1)

using namespace std;
// Define Edge class
class Edge
{
public:
  int from, to, flow, cost;
  Edge *reverse;
  Edge(int from, int to, int flow, int cost) : from(from), to(to), flow(flow), cost(cost) {}
  // Function to reverse the edge
  void setReverse(Edge *rev)
  {
    reverse = rev;
  }
};
// Define Graph class
class Graph
{
public:
  int nodesCnt;
  vector<Edge *> edges;
```

```
  Graph(int n) : nodesCnt(n) {}

  // Function to add an edge to the graph

  void addEdge(int from, int to, int flow, int
cost)

  {

    Edge *e1 = new Edge(from, to, flow,
cost);

    Edge *e2 = new Edge(to, from, 0, -cost);

    e1->setReverse(e2);

    e2->setReverse(e1);

    edges.push_back(e1);

    edges.push_back(e2);

  }

  // Function to iterate over edges in the
graph

  vector<Edge *>::iterator begin()

  {

    return edges.begin();

  }

  vector<Edge *>::iterator end()

  {

    return edges.end();

  }

};

class BellmanFordCycle

{

public:

  int searchForNegWtCycle(Graph
&graphWithWeightCosts, int totalNodes, int
src,

                vector<Edge *>
&nodesInCycle)

  {

    vector<int> weightArr =
initWeightArray(totalNodes, src);

    vector<Edge *>
predecessorEdgeArr(graphWithWeightCost
s.nodesCnt, nullptr);


printCostGraph(graphWithWeightCosts); //
O(E)

    int flowCost = 0;

    for (int k = 0; k < totalNodes; k++)

    { // O(V)

      for (Edge *e : graphWithWeightCosts)

      { // O(E)

        if (weightArr[e->from] ==
numeric_limits<int>::max())

        { // O(1)

          continue;

        }

        flowCost = e->cost; // O(1)

        if ((flowCost + weightArr[e->from]) <
weightArr[e->to])

        {                                // O(1)

          weightArr[e->to] = flowCost +
weightArr[e->from]; // O(1)

          predecessorEdgeArr[e->to] = e;
// O(1)

        }

      }

    }

    Edge *foundCycle = nullptr;

    for (Edge *e : graphWithWeightCosts)

    { // O(E)

      if (weightArr[e->from] ==
numeric_limits<int>::max())

      { // O(1)

        continue;

      }

      flowCost = e->cost; // O(1)

      if ((flowCost + weightArr[e->from]) <
weightArr[e->to])

      {           // O(1)

        foundCycle = e; // O(1)

        break;

      }

    }

    if (foundCycle == nullptr)

    {         // O(1)

      return -1; // O(1)

    }

    cout << "Found Cycle ";     // O(1)

    Edge *startEdge = foundCycle; // O(1)

    Edge *predEdge = nullptr;    // O(1)

    while (true)

    {                                // O(E)

      predEdge =
predecessorEdgeArr[startEdge->from]; //
O(1)

      if (predEdge != nullptr && predEdge ==
startEdge)

      { // O(1)

        break;

      }

      nodesInCycle.push_back(predEdge);
// O(1)

      startEdge = predEdge;          // O(1)

    }

    int idx = 0; // O(1)

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

    { // O(E)

      if (nodesInCycle[i] == predEdge)

      {        // O(1)

        idx = i; // O(1)

        break;

      }

    }


nodesInCycle.erase(nodesInCycle.begin(),
nodesInCycle.begin() + idx); // O(E)

    int minimumInCycle =
numeric_limits<int>::max();                //
O(1)

    for (Edge *e : nodesInCycle)

    {                                // O(E)

      minimumInCycle =
min(minimumInCycle, e->flow); // O(1)

    }

    return minimumInCycle; // O(1)

  }

  vector<int> initWeightArray(int
totalNodes, int src)

  {                                    // O(V)

    vector<int> wtArr(totalNodes,
numeric_limits<int>::max()); // O(V)

    wtArr[src] = 0;                    //
O(1)

    return wtArr;                      //
O(1)

  }

  void printCostGraph(Graph &graph)

  {                    // O(E)

    cout << "Cost Matrix" << endl; // O(1)

    for (Edge *e : graph.edges)

    {                                          //
O(E)

      cout << e->from << " -> " << e->to << "
: " << e->flow * e->cost << endl; // O(1)

    }

  }

};


int main()
```

```
{
    int totalNodes = 5;                 // O(1)
    Graph graphWithWeightCosts(totalNodes); // O(1)
    // Example usage: Add edges to the graph
    graphWithWeightCosts.addEdge(0, 1, 1, 2);  // O(1)
    graphWithWeightCosts.addEdge(1, 2, 1, -1); // O(1)
    graphWithWeightCosts.addEdge(2, 3, 1, 4);  // O(1)
    graphWithWeightCosts.addEdge(3, 0, 1, -3); // O(1)
    int src = 0;                        // O(1)
    vector<Edge *> nodesInCycle;            // O(1)
    BellmanFordCycle bellmanFordCycle;      // O(1)
    int minFlow = bellmanFordCycle.searchForNegWtCycle(graphWithWeightCosts,
                                totalNodes, src, nodesInCycle); // O(V * E)
    cout << "Minimum Flow in Negative Weight Cycle: " << minFlow << endl;          // O(1)
    return 0;                           // O(1)
}
```

4. Topological Sort:

the time complexity of the provided code for performing topological sorting using DFS is O(V+E).

Big O (O): The time complexity of this code is O(V+E), where V is the number of vertices and E is the number of edges in the graph. Constructing the adjacency list takes O(E) time since each edge is processed once to add it to the adjacency list. The topological sort algorithm itself consists of a DFS traversal, which visits each vertex and each edge once. Therefore, the time complexity of the DFS traversal is O(V+E). Printing the sorted vertices from the stack also takes O(V) time, as there are V vertices in the graph. Therefore, the overall time complexity is O(V+E).

Omega (Ω): The best-case scenario occurs when the graph is already in topological order, meaning there are no backward edges. In this case, the time complexity is the same as the worst-case, which is O(V+E).

Theta (Θ): Since the upper bound (Big O) and lower bound (Omega) of the algorithm are the same O(V+E)), the time complexity of the provided code can be expressed as Θ(V+E).

In summary:

- **Ω(V+ E)**
- **O(V + E)**
- **Θ(V + E)**

Code:

```
#include <bits/stdc++.h>
using namespace std;
// Function to perform DFS and topological sorting
void topologicalSortUtil(int v, vector<vector<int>> &adj,
                         vector<bool> &visited,
                         stack<int> &Stack)
{
    // Mark the current node as visited
    visited[v] = true; // O(1)
    // Recur for all adjacent vertices
    for (int i : adj[v])
    {                                   // O(E) where E is the number of edges
        if (!visited[i])                // O(1)
            topologicalSortUtil(i, adj, visited, Stack); // O(E)
    }
    // Push current vertex to stack which stores the result
    Stack.push(v); // O(1)
}
// Function to perform Topological Sort
void topologicalSort(vector<vector<int>> &adj, int V)
{
    stack<int> Stack;               // O(1)
    vector<bool> visited(V, false); // O(V)
    // Call the recursive helper function to store Topological Sort starting from all vertices one by one
    for (int i = 0; i < V; i++)
    {                                   // O(V)
        if (!visited[i])                // O(1)
            topologicalSortUtil(i, adj, visited, Stack); // O(V + E)
    }
    // Print contents of stack
    while (!Stack.empty())
    {                           // O(V)
        cout << Stack.top() << " "; // O(1)
        Stack.pop();                // O(1)
    }
}
int main()
{
    // Number of nodes
    int V = 4; // O(1)
    // Edges
    vector<vector<int>> edges = {{0, 1}, {1, 2}, {3, 1}, {3, 2}}; // O(1)
    // Graph represented as an adjacency list
    vector<vector<int>> adj(V); // O(1)
    for (auto i : edges)
    {                           // O(E) where E is the number of edges
        adj[i[0]].push_back(i[1]); // O(1)
    }
    cout << "Topological sorting of the graph: ";
    topologicalSort(adj, V); // O(V + E)
    return 0;                // O(1)
}
```

5. Bipartite Matching for Task Assignment (Bipartite Matching):

while O(V×E) represents the worst-case time complexity, the actual runtime performance of the algorithm may vary based on the characteristics of the input graph and the efficiency of the algorithm in finding matchings.

Best Case: In the best case scenario, the algorithm finds a matching for each applicant in the graph quickly without needing to traverse all potential edges. This could happen when the graph is small or when the structure of the graph allows for easy matching. In this case, the time complexity would be less than O(V×E), possibly approaching O(V) if the number of edges per vertex is limited.

Worst Case: The worst-case scenario occurs when the algorithm needs to traverse all edges in the graph to find the matching for each applicant. This typically happens when the graph is dense, meaning there are many potential job assignments for each applicant. In this case, the time complexity would be O(V×E), as the algorithm needs to visit every vertex and potentially traverse every edge in the graph.

Average Case: The average case time complexity depends on the distribution of edges in the graph and the efficiency of the algorithm in finding matchings. If the graph is randomly generated or follows a specific distribution, the average case time complexity may still be close to O(V×E) if the graph tends to be dense. However, if the graph tends to be sparse or has certain patterns that make matching easier, the average case time complexity may be lower than O(V×E).

In summary:

- **Ω(V* E)**
- **O(V * E)**
- **Θ(V * E)**

Code:

```
#include <iostream>

#include <string.h>



using namespace std;

// M is number of applicants and N is
number of jobs



#define M 6

#define N 6

// A DFS based recursive function that
returns true if a matching for vertex u is
possible

bool bpm(bool bpGraph[M][N], int u,

      bool seen[], int matchR[])

{

  // Try every job one by one

  for (int v = 0; v < N; v++) // O(N)

  {

    // If applicant u is interested in job v and
v is not visited

    if (bpGraph[u][v] && !seen[v])

    {

      // Mark v as visited

      seen[v] = true; // O(1)

      // If job 'v' is not assigned to an
applicant OR previously assigned applicant
for job v (which is matchR[v]) has an
alternate job available. Since v is marked as
visited in the above line, matchR[v] in the
following recursive call will not get job 'v'
again

      if (matchR[v] < 0 || bpm(bpGraph,
matchR[v],

                  seen, matchR)) // O(E)

      {

        matchR[v] = u; // O(1)

        return true;  // O(1)

      }

    }

  }

  return false; // O(1)

}



// Returns maximum number of matching
from M to N

int maxBPM(bool bpGraph[M][N])

{

  // An array to keep track of the applicants
assigned to jobs. The value of matchR[i] is
the applicant number assigned to job i, the
value -1 indicates nobody is assigned.

  int matchR[N]; // O(N)

  // Initially all jobs are available

  memset(matchR, -1, sizeof(matchR)); //
O(N)

  // Count of jobs assigned to applicants

  int result = 0;         // O(1)

  for (int u = 0; u < M; u++) // O(V)

  {

    // Mark all jobs as not seen for next
applicant.

    bool seen[N];             // O(N)

    memset(seen, 0, sizeof(seen)); // O(N)

    // Find if the applicant 'u' can get a job

    if (bpm(bpGraph, u, seen, matchR)) //
O(V * E)

      result++;               // O(1)

  }

  return result; // O(1)

}



// Driver Code

int main()

{

  // Let us create a bpGraph shown in the
above example

  bool bpGraph[M][N] = {{0, 1, 1, 0, 0, 0},

                {1, 0, 0, 1, 0, 0},

                {0, 0, 1, 0, 0, 0},

                {0, 0, 1, 1, 0, 0},

                {0, 0, 0, 0, 0, 0},

                {0, 0, 0, 0, 0, 1}};

  cout << "Maximum number of applicants
that can get job is "

     << maxBPM(bpGraph); // O(V * E)

  return 0;          // O(1)

}
```