Design and Analysis of Algorithms

Unit 1

Asymptotic Analysis

Asymptotic analysis of an algorithm refers to defining the mathematical foundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as f(n) and may be for another operation it is computed as *g*(n2). This means the first operation running time will increase linearly with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases. Similarly, the running time of both operations will be nearly the same if **n** is significantly small.

Usually, the time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

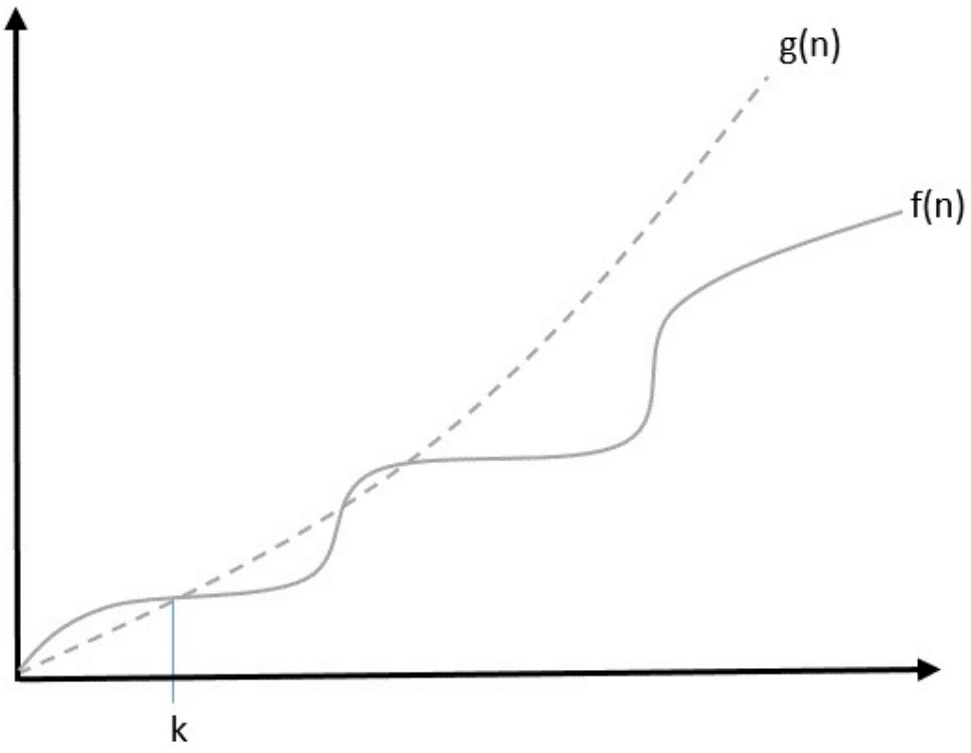
Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O − Big Oh Notation
- Ω − Big Omega Notation
- Θ − Theta Notation
- o − Little Oh Notation
- ω − Little Omega Notation

Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the **worst case time complexity** or the longest amount of time an algorithm can possibly take to complete.



For example, for a function ***f*(n)**

```
O(f(n)) = { g(n) : there exists c > 0 and n0 such that g(n) ≤ c.f(n) for all n > n0. }
```

Example

Let us consider a given function, ***f*(*n*) = 4.*n*^3+10.*n*^2+5.*n*+1.**

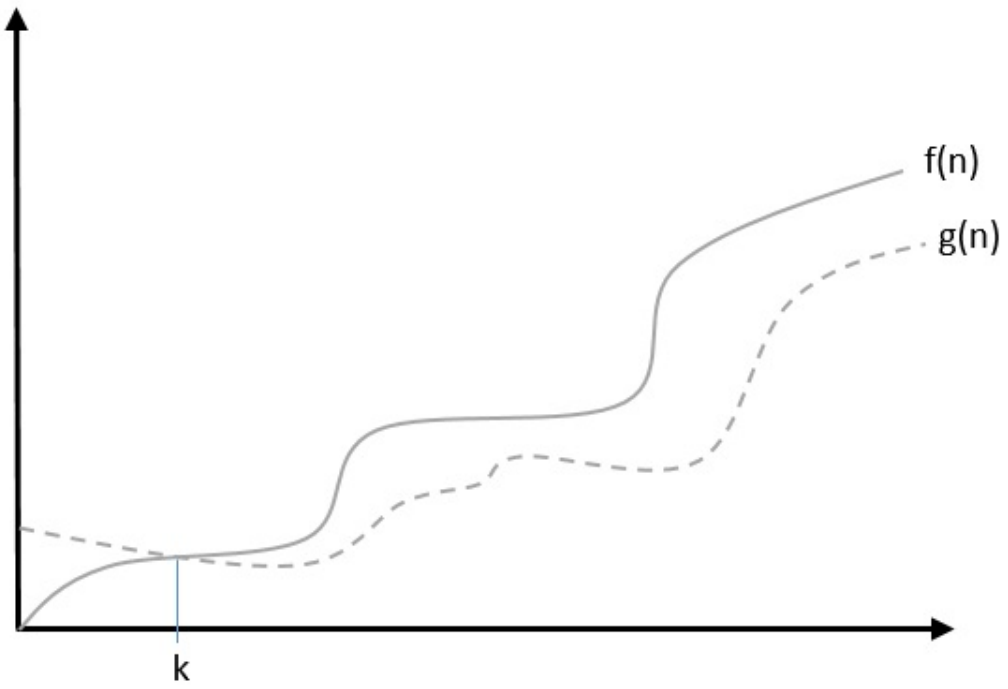
Considering ***g(n) = n^3***

***f*(*n*) ≥ 5.*g*(*n*)** for all the values of ***n* > 2**.

Hence, the complexity of ***f*(n)** can be represented as ***O* (*g* (*n*)) ,i.e. *O* (*n*^3)**.

Big Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the **best case time complexity** or the best amount of time an algorithm can possibly take to complete.



For example, for a function ***f*(*n*)**

```
Ω(f(n)) ≥ { g(n) : there exists c > 0 and n0 such that g(n) ≤ c.f(n) for all n > n0. }
```

Example

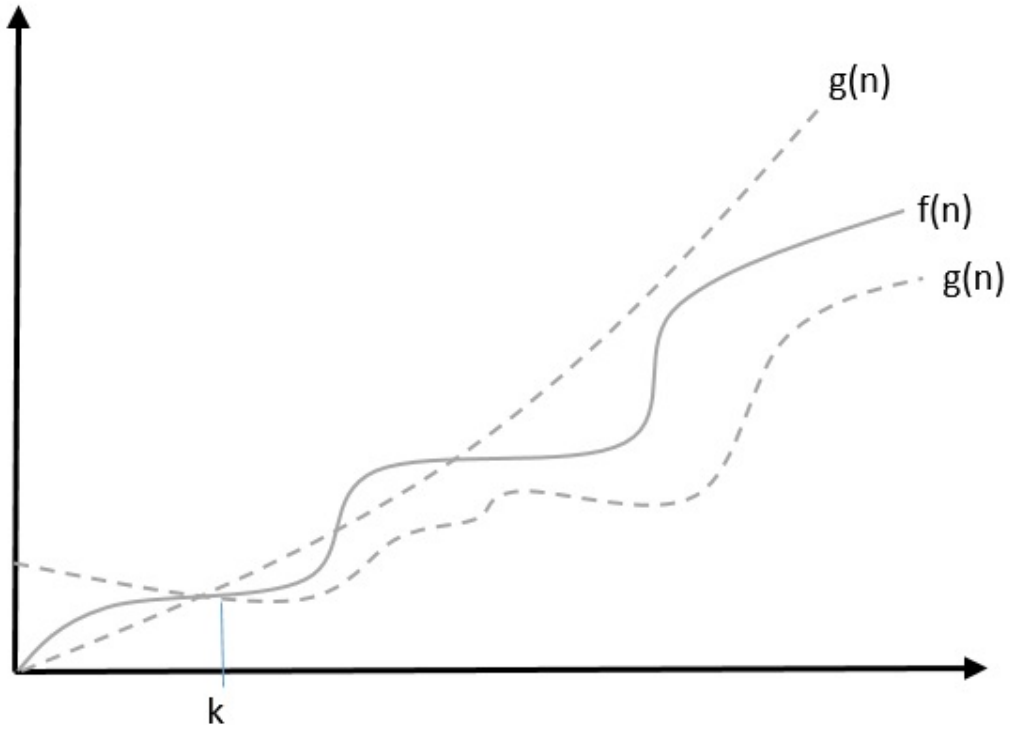
Let us consider a given function, ***f*(*n*) = 4.*n*^3+10.*n*^2+5.*n*+1**

Considering ***g*(*n*) = *n*^3** , ***f*(*n*) ≥ 4.*g*(*n*)** for all the values of ***n* > 0**.

Hence, the complexity of *f*(*n*) can be represented as ***Ω* (*g* (*n*)) ,i.e. *Ω* (*n*^3)**.

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. Some may confuse the theta notation as the average case time complexity; while big theta notation could be *almost* accurately used to describe the average case, other notations could be used as well. It is represented as follows −




```
θ(f(n)) = { g(n) if and only if g(n) = O(f(n)) and g(n) = Ω(f(n)) for all n > n0. }
```

Example

Let us consider a given function, ***f*(*n*) = 4.*n*^3+10.*n*^2+5.*n*+1**

Considering ***g*(*n*) = *n*^3** , **4.*g*(*n*)≤ *f*(*n*)≤ 5.*g*(*n*)** for all the values of ***n***.

Hence, the complexity of ***f*(*n*)** can be represented as ***θ* (*g* (*n*)) ,i.e. *θ* (*n*^3)**.

Little Oh (o) and Little Omega (ω) Notations

The Little Oh and Little Omega notations also represent the best and worst case complexities but they are not asymptotically tight in contrast to the Big Oh and Big Omega Notations. Therefore, the most commonly used notations to represent time complexities are Big Oh and Big Omega Notations only.

Common Asymptotic Notations

Following is a list of some common asymptotic notations −

constant	−	O(1)
logarithmic	−	O(log n)
linear	−	O(n)
n log n	−	O(n log n)
quadratic	−	O(n^2)
cubic	−	O(n^3)
polynomial	−	n^O(1)
exponential	−	2^O(n)

Recurrence Relation

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.

For Example, the Worst Case Running Time T(n) of the MERGE SORT Procedures is described by the recurrence.

```
T (n) = θ (1) if n=1
2T(n/2) + θ (n) if n>1
```

There are four methods for solving Recurrence:

1. Substitution Method
2. Iteration Method

3. Recursion Tree Method
4. Master Method

How to analyse Complexity of Recurrence Relation:
https://www.geeksforgeeks.org/how-to-analyse-complexity-of-recurrence-relation/
https://www.javatpoint.com/daa-recurrence-relation

Graph Basics:

- A **graph** is a mathematical and data structure representation of a set of **vertices** (nodes) connected by **edges** (lines).
- Graphs are used to model various relationships, networks, and structures in different domains, such as social networks, transportation systems, computer networks, and more.

Types of Graphs:

1. **Undirected Graph**: In an undirected graph, edges have no direction. If there's an edge from vertex A to B, there's also an edge from B to A.
2. **Directed Graph (Digraph)**: In a directed graph, edges have direction. An edge from vertex A to B doesn't imply an edge from B to A.
3. **Weighted Graph**: A weighted graph assigns a weight or cost to each edge, representing some value associated with the connection between vertices.
4. **Cyclic Graph**: A graph that contains at least one cycle (a closed path that starts and ends at the same vertex).
5. **Acyclic Graph**: A graph that has no cycles. A special case is a **DAG (Directed Acyclic Graph)**, commonly used for topological sorting and dynamic programming.

Graph Terminology:

- **Vertex (Plural: Vertices)**: The fundamental building blocks of a graph.
- **Edge**: Represents a connection between two vertices.
- **Adjacent**: Two vertices are adjacent if there is an edge connecting them.
- **Path**: A path is a sequence of vertices where each adjacent pair is connected by an edge.

- **Cycle**: A path that starts and ends at the same vertex.
- **Degree**: The degree of a vertex is the number of edges connected to it.
- **Connected Graph**: In an undirected graph, there is a path between any pair of vertices.

Graph Representations:

- **Adjacency Matrix**: A 2D array where `matrix[i][j]` is `1` if there's an edge between vertex `i` and `j`, and `0` otherwise. Suitable for dense graphs.
- **Adjacency List**: A collection of lists, one for each vertex, where each list contains the vertices adjacent to that vertex. Suitable for sparse graphs.

Graph Algorithms:

- **Breadth-First Search (BFS)**: Used to explore all vertices reachable from a starting vertex in the shortest path order.
- **Depth-First Search (DFS)**: Used to explore as far as possible along each branch before backtracking.
- **Shortest Path Algorithms**: Such as Dijkstra's and Bellman-Ford, used to find the shortest path between vertices in weighted graphs.
- **Minimum Spanning Tree**: Algorithms like Kruskal's and Prim's are used to find the minimum spanning tree in a weighted graph.

Applications:

- **Social Networks**: Modeling relationships between people.
- **Routing Algorithms**: Finding the shortest path in computer networks.
- **Recommendation Systems**: Suggesting items based on user preferences.
- **Transportation Networks**: Optimizing traffic flow and routes.
- **Circuit Design**: Creating efficient electronic circuits.
- **Game Development**: Modeling game maps and character movements.

Graphs are a versatile data structure with a wide range of applications. Understanding their properties and algorithms is essential for solving various computational problems efficiently.

Sets and disjoint sets, union

Sets and disjoint sets (also known as disjoint-set data structures) are essential components in computer science and are often used for solving problems related to partitions and connectivity. Here, I'll provide an overview and code examples for sets and disjoint sets in C++.

Sets:

In C++, a set is a data structure that stores a collection of distinct elements. It is part of the Standard Template Library (STL) and is typically implemented as a Red-Black Tree or Hash Table, ensuring that elements are unique and efficiently accessible.

Here's an example of using the `std::set` in C++:

```
#include <iostream>
#include <set>

int main() {
    std::set<int> mySet;

    mySet.insert(5);
    mySet.insert(2);
    mySet.insert(8);

    // Check if an element is in the set
    if (mySet.find(5) != mySet.end()) {
        std::cout << "5 is in the set." << std::endl;
    }

    // Iterate through the set
    for (int num : mySet) {
        std::cout << num << " ";
    }

    return 0;
}
```

In this example, we create a `std::set` to store integers, insert elements, check if an element exists, and iterate through the set.

Disjoint Sets (Union-Find Data Structure):

Disjoint-set data structures (also known as the union-find data structure) are used to manage partitions of a finite set into disjoint sets. It's often used to solve problems

involving connectivity, like finding connected components in a graph.

Here's an example of a basic union-find implementation in C++:

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

class DisjointSet {
public:
    DisjointSet(int size) {
        parent.resize(size);
        rank.resize(size, 0);

        for (int i = 0; i < size; i++) {
            parent[i] = i;
        }
    }

    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    void unionSets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY) {
            if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }

private:
    std::vector<int> parent;
    std::vector<int> rank;
};

int main() {
    DisjointSet dsu(5);

    dsu.unionSets(0, 1);
```

```
    dsu.unionSets(2, 3);
    dsu.unionSets(0, 4);

    std::cout << "Are 1 and 4 in the same set? " << (dsu.find(1) == dsu.find(4)) << std::e
ndl;

    return 0;
}
```

In this code:

- We create a `DisjointSet` class to represent a disjoint-set data structure.
- `find` finds the representative (root) of a set to which an element belongs.
- `unionSets` combines two sets if they are not in the same set already.
- We demonstrate the connectivity check between elements by using the `find` method.

These are the fundamental concepts and code examples for sets and disjoint sets in C++. Depending on your specific problem, you can utilize these data structures to manage collections and solve connectivity-related problems efficiently.

Sorting algos

1. **Bubble Sort**:

- **Time Complexity**: O(n^2) in the worst and average cases, O(n) in the best case (when the array is already sorted).
- **Space Complexity**: O(1) as it sorts in-place.
- Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

2. **Selection Sort**:

- **Time Complexity**: O(n^2) in all cases.
- **Space Complexity**: O(1) as it sorts in-place.
- Selection Sort divides the input into a sorted and an unsorted region, repeatedly selects the minimum element from the unsorted region, and moves it to the sorted

region.

3. **Insertion Sort**:

- **Time Complexity**: O(n^2) in the worst and average cases, O(n) in the best case (when the array is nearly sorted).
- **Space Complexity**: O(1) as it sorts in-place.
- Insertion Sort builds the final sorted array one item at a time by repeatedly moving elements between the sorted and unsorted regions.

4. **Merge Sort**:

- **Time Complexity**: O(n log n) in all cases.
- **Space Complexity**: O(n) for additional space due to the merge step.
- Merge Sort uses the divide-and-conquer strategy to divide the array into smaller subarrays, sort them, and merge them back together.

5. **Quick Sort**:

- **Time Complexity**: O(n^2) in the worst case (rare), O(n log n) on average.
- **Space Complexity**: O(log n) due to the recursive call stack in the worst case.
- Quick Sort selects a 'pivot' element and partitions the array into two subarrays - those less than the pivot and those greater. It recursively sorts the subarrays.

6. **Heap Sort**:

- **Time Complexity**: O(n log n) in all cases.
- **Space Complexity**: O(1) as it sorts in-place.
- Heap Sort transforms the input array into a max-heap, repeatedly extracts the maximum element from the heap and places it in the sorted region.

7. **Counting Sort**:

- **Time Complexity**: O(n + k), where 'k' is the range of input values.
- **Space Complexity**: O(n + k) for additional storage.
- Counting Sort is a non-comparative sorting algorithm that counts the number of occurrences of each element and uses this information to sort the elements.

8. **Radix Sort**:

- **Time Complexity**: O(k * (n+b)), where 'k' is the number of digits and 'b' is the base.
- **Space Complexity**: O(n + k) for additional storage.
- Radix Sort sorts numbers by processing individual digits, from the least significant to the most significant.

The choice of sorting algorithm depends on the specific use case, the size of the data, and the desired time and space constraints. Some algorithms, like Quick Sort and Merge Sort, are more efficient for large datasets, while others, like Bubble Sort and Selection Sort, are simpler but less efficient. Counting Sort and Radix Sort are non-comparative algorithms suitable for specific scenarios.

1. Bubble Sort:

```
#include <iostream>

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                std::swap(arr[j], arr[j + 1]);
            }
        }
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    bubbleSort(arr, n);

    std::cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }
    return 0;
}
```

2. Selection Sort:

```
#include <iostream>

void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        std::swap(arr[i], arr[minIndex]);
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    selectionSort(arr, n);

    std::cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }
    return 0;
}
```

3. Insertion Sort:

```
#include <iostream>

void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
```

```
    std::cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }
    return 0;
}
```

4. Merge Sort:

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

void merge(std::vector<int>& arr, int left, int middle, int right) {
    int n1 = middle - left + 1;
    int n2 = right - middle;

    std::vector<int> L(n1);
    std::vector<int> R(n2);

    for (int i = 0; i < n1; i++) {
        L[i] = arr[left + i];
    }
    for (int i = 0; i < n2; i++) {
        R[i] = arr[middle + 1 + i];
    }

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
```

```
    }
}

void mergeSort(std::vector<int>& arr, int left, int right) {
    if (left < right) {
        int middle = left + (right - left) / 2;
        mergeSort(arr, left, middle);
        mergeSort(arr, middle + 1, right);
        merge(arr, left, middle, right);
    }
}

int main() {
    std::vector<int> arr = {64, 34, 25, 12, 22, 11, 90};
    int n = arr.size();

    mergeSort(arr, 0, n - 1);

    std::cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }
    return 0;
}
```

5. Quick Sort:

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

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

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return (i + 1);
}

void quickSort(std::vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
```

```
    }
}

int main() {
    std::vector<int> arr = {64, 34, 25, 12, 22, 11, 90};
    int n = arr.size();

    quickSort(arr, 0, n - 1);

    std::cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }
    return 0;
}
```

6. Heap Sort:

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

void heapify(std::vector<int>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    if (largest != i) {
        std::swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(std::vector<int>& arr) {
    int n = arr.size();

    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    for (int i = n - 1; i >= 0; i--) {
        std::swap(arr[0], arr[i]);
```

```
        heapify(arr, i, 0);
    }
}

int main() {
    std::vector<int> arr = {64, 34, 25, 12, 22, 11, 90};
    int n = arr.size();

    heapSort(arr);

    std::cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }
    return 0;
}
```

Searching algos

1. **Linear Search**:

- **Time Complexity**: O(n) in the worst case (element is at the end), O(1) in the best case (element is at the beginning).
- **Space Complexity**: O(1) as it doesn't require additional space.

Linear search, also known as sequential search, checks each element in the array or list one by one until it finds the target element.

Here's the C++ code for linear search:

```
#include <iostream>

int linearSearch(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target) {
            return i;  // Return the index of the target element
        }
    }
    return -1;  // Target element not found
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 22;
```

```
    int result = linearSearch(arr, n, target);

    if (result != -1) {
        std::cout << "Element " << target << " found at index " << result << std::endl;
    } else {
        std::cout << "Element not found" << std::endl;
    }

    return 0;
}
```

2. **Binary Search**:

- **Time Complexity**: O(log n) as it divides the search interval in half at each step.
- **Space Complexity**: O(1) as it doesn't require additional space.

Binary search is a fast search algorithm that works on sorted arrays. It compares the target element with the middle element of the array and narrows down the search space by half with each comparison.

Here's the C++ code for binary search:

```
#include <iostream>

int binarySearch(int arr[], int n, int target) {
    int left = 0;
    int right = n - 1;

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

        if (arr[middle] == target) {
            return middle;  // Return the index of the target element
        } else if (arr[middle] < target) {
            left = middle + 1;
        } else {
            right = middle - 1;
        }
    }
    return -1;  // Target element not found
}

int main() {
    int arr[] = {11, 12, 22, 25, 34, 64, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 22;
```

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

    if (result != -1) {
        std::cout << "Element " << target << " found at index " << result << std::endl;
    } else {
        std::cout << "Element not found" << std::endl;
    }

    return 0;
}
```

These are the code examples for linear search and binary search. Linear search is straightforward but less efficient, while binary search is highly efficient for sorted data. Both have their use cases depending on the specific problem and dataset.

Divide and Conquer

Divide and Conquer is a fundamental algorithmic technique used to solve problems by breaking them down into smaller subproblems, solving the subproblems, and then combining their solutions to obtain the solution to the original problem. This technique is widely used in various algorithms and has applications in various domains of computer science and mathematics. Here's an explanation of the Divide and Conquer technique and some C++ code examples to illustrate it:

Divide and Conquer Approach:

The Divide and Conquer technique involves the following steps:

1. **Divide**: Divide the problem into smaller subproblems that are similar to the original problem but of a smaller size. This step continues until the subproblems become simple enough to solve directly.
2. **Conquer**: Solve the subproblems. If the subproblems are small enough, you can solve them directly. Otherwise, recursively apply the Divide and Conquer approach to solve them.
3. **Combine**: Combine the solutions of the subproblems to obtain the solution to the original problem. This is typically done in a way that the solutions are efficiently merged to achieve the desired result.

C++ Code Examples Using Divide and Conquer:

1. **Merge Sort**:

Merge Sort is a classic example of a Divide and Conquer sorting algorithm. It divides the array into smaller subarrays, sorts them, and then merges them back together.

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

void merge(std::vector<int>& arr, int left, int middle, int right) {
    // Merge two sorted subarrays
}

void mergeSort(std::vector<int>& arr, int left, int right) {
    if (left < right) {
        int middle = left + (right - left) / 2;
        mergeSort(arr, left, middle);
        mergeSort(arr, middle + 1, right);
        merge(arr, left, middle, right);
    }
}

int main() {
    std::vector<int> arr = {64, 34, 25, 12, 22, 11, 90};
    int n = arr.size();

    mergeSort(arr, 0, n - 1);

    std::cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }
    return 0;
}
```

2. **Binary Search**:

   Binary Search is another example of a Divide and Conquer algorithm. It divides the search space into smaller segments and searches for a target element efficiently.

```
#include <iostream>

int binarySearch(int arr[], int left, int right, int target) {
    // Perform binary search
}

int main() {
    int arr[] = {11, 12, 22, 25, 34, 64, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    int target = 22;

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

    if (result != -1) {
        std::cout << "Element " << target << " found at index " << result << std::end
l;
    } else {
        std::cout << "Element not found" << std::endl;
    }
    return 0;
}
```

These examples demonstrate how Divide and Conquer is used to solve problems efficiently by dividing them into smaller, manageable subproblems and then combining the solutions. The technique is applicable to a wide range of problems in computer science and beyond.

General Method

The "General Method" of Divide and Conquer is a fundamental algorithmic technique used to solve a wide range of problems. It involves dividing a complex problem into smaller, more manageable subproblems, solving these subproblems independently, and then combining their solutions to obtain the solution to the original problem. This technique is applicable to a variety of computational problems and is a common approach in algorithm design. Here's a general overview of the Divide and Conquer method and an example of its application:

Divide and Conquer Approach:

The Divide and Conquer approach typically consists of the following steps:

1. **Divide**: Break the original problem into smaller, similar subproblems. The goal is to make the subproblems as simple as possible while maintaining the integrity of the original problem. This division process continues recursively until the subproblems become trivial to solve.
2. **Conquer**: Solve each of the subproblems independently. If the subproblems are simple enough, they can be solved directly. If not, the Divide and Conquer method is applied to solve them recursively.
3. **Combine**: Combine the solutions of the subproblems to obtain the solution to the original problem. The combination should be done in a way that ensures the overall

problem's solution is correct and efficient.

Example: Finding the Maximum Subarray Sum (Kadane's Algorithm)

Let's illustrate the Divide and Conquer approach with an example: finding the maximum subarray sum within an array. Kadane's algorithm is a classic example that uses this technique.

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

int maxCrossingSum(std::vector<int>& arr, int low, int mid, int high) {
    int leftMax = INT_MIN;
    int rightMax = INT_MIN;

    int sum = 0;
    for (int i = mid; i >= low; i--) {
        sum += arr[i];
        if (sum > leftMax) {
            leftMax = sum;
        }
    }

    sum = 0;
    for (int i = mid + 1; i <= high; i++) {
        sum += arr[i];
        if (sum > rightMax) {
            rightMax = sum;
        }
    }

    return leftMax + rightMax;
}

int maxSubarraySum(std::vector<int>& arr, int low, int high) {
    if (low == high) {
        return arr[low];
    }

    int mid = (low + high) / 2;

    int leftMax = maxSubarraySum(arr, low, mid);
    int rightMax = maxSubarraySum(arr, mid + 1, high);
    int crossMax = maxCrossingSum(arr, low, mid, high);

    return std::max(std::max(leftMax, rightMax), crossMax);
}

int main() {
```

```
    std::vector<int> arr = {1, -3, 2, 1, -1};
    int n = arr.size();

    int maxSum = maxSubarraySum(arr, 0, n - 1);

    std::cout << "Maximum subarray sum: " << maxSum << std::endl;

    return 0;
}
```

In this example, the Divide and Conquer technique is applied to find the maximum subarray sum within an array. The array is divided into smaller subarrays, and the maximum subarray sum is found for each subarray using recursion. Then, the results from the subproblems are combined to find the maximum subarray sum for the original array. This approach provides an efficient way to solve this problem with a time complexity of O(n log n), where 'n' is the size of the array.

The Divide and Conquer technique can be applied to various problems in computer science, ranging from sorting and searching to optimization and more complex algorithmic challenges. It's a versatile and powerful approach to problem-solving.

Quick Sort (Divide and Conquer):

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

using namespace std;

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;
}

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 main() {
    vector<int> arr = {64, 34, 25, 12, 22, 11, 90};
    int n = arr.size();

    quickSort(arr, 0, n - 1);

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

    return 0;
}
```

Selection Sort (Divide and Conquer):

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

using namespace std;

void selectionSort(vector<int>& arr) {
    int n = arr.size();

    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        swap(arr[i], arr[minIndex]);
    }
}

int main() {
    vector<int> arr = {64, 34, 25, 12, 22, 11, 90};

    selectionSort(arr);

    cout << "Sorted array: ";
    for (int num : arr) {
        cout << num << " ";
    }
```

```
    return 0;
}
```

Strassen's Matrix Multiplication

Strassen's Matrix Multiplication is a divide-and-conquer algorithm used to multiply two matrices. It's named after Volker Strassen, who introduced the algorithm in 1969. This algorithm is known for its efficient matrix multiplication using recursive partitioning and has a time complexity lower than the standard matrix multiplication algorithm. Let's dive into Strassen's Matrix Multiplication and analyze it in detail.

Algorithm Overview:

The standard matrix multiplication of two N×N matrices has a time complexity of O(N^3). Strassen's algorithm reduces the number of basic multiplications by using a divide-and-conquer approach. The key idea is to split the matrices into four smaller submatrices and perform several recursive multiplications.

Here are the steps involved in Strassen's Matrix Multiplication:

1. Divide both input matrices (A and B) into four equal-sized submatrices:
    - A = {{A11, A12}, {A21, A22}}
    - B = {{B11, B12}, {B21, B22}}
2. Calculate seven products (P1 to P7) using the submatrices A and B:
    - P1 = A11 * (B12 - B22)
    - P2 = (A11 + A12) * B22
    - P3 = (A21 + A22) * B11
    - P4 = A22 * (B21 - B11)
    - P5 = (A11 + A22) * (B11 + B22)
    - P6 = (A12 - A22) * (B21 + B22)
    - P7 = (A11 - A21) * (B11 + B12)
3. Calculate the resulting submatrices C11, C12, C21, and C22:
    - C11 = P5 + P4 - P2 + P6

- C12 = P1 + P2
- C21 = P3 + P4
- C22 = P5 + P1 - P3 - P7

4. Combine the resulting submatrices C11, C12, C21, and C22 to obtain the final result matrix C.

Time Complexity Analysis:

The time complexity of Strassen's Matrix Multiplication can be analyzed as follows:

- The recursive algorithm divides the matrices into smaller submatrices, and each submatrix multiplication requires seven multiplications and 18 additions or subtractions (constant operations).
- The recurrence relation for the time complexity can be expressed as:
  T(N) = 7T(N/2) + O(N^2)
- By applying the Master Theorem, Strassen's algorithm has a time complexity of O(N^log2(7)), which is approximately O(N^2.81).

Space Complexity Analysis:

The space complexity of Strassen's Matrix Multiplication depends on the auxiliary storage required for the submatrices. In the divide-and-conquer approach, extra space is needed for the submatrices, and this space usage can be analyzed as O(N^2).

C++ Code Example:

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

using namespace std;

// Function to add two matrices
vector<vector<int>> matrixAddition(const vector<vector<int>>& A, const vector<vector<int>>
& B) {
    int n = A.size();
    vector<vector<int>> result(n, vector<int>(n, 0));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            result[i][j] = A[i][j] + B[i][j];
        }
    }
```

```
    return result;
}

// Function to subtract two matrices
vector<vector<int>> matrixSubtraction(const vector<vector<int>>& A, const vector<vector<in
t>>& B) {
    int n = A.size();
    vector<vector<int>> result(n, vector<int>(n, 0));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            result[i][j] = A[i][j] - B[i][j];
        }
    }

    return result;
}

// Function to multiply two matrices using Strassen's algorithm
vector<vector<int>> strassenMatrixMultiply(const vector<vector<int>>& A, const vector<vect
or<int>>& B) {
    int n = A.size();

    // Base case: if the matrix size is 1x1
    if (n == 1) {
        vector<vector<int>> result(1, vector<int>(1, 0));
        result[0][0] = A[0][0] * B[0][0];
        return result;
    }

    // Divide matrices into four submatrices
    int halfSize = n / 2;
    vector<vector<int>> A11(halfSize, vector<int>(halfSize));
    vector<vector<int>> A12(halfSize, vector<int>(halfSize));
    vector<vector<int>> A21(halfSize, vector<int>(halfSize));
    vector<vector<int>> A22(halfSize, vector<int>(halfSize));

    vector<vector<int>> B11(halfSize, vector<int>(halfSize));
    vector<vector<int>> B12(halfSize, vector<int>(halfSize));
    vector<vector<int>> B21(halfSize, vector<int>(halfSize));
    vector<vector<int>> B22(halfSize, vector<int>(halfSize));

    for (int i = 0; i < halfSize; i++) {
        for (int j = 0; j < halfSize; j++) {
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j + halfSize];
            A21[i][j] = A[i + halfSize][j];
            A22[i][j] = A[i + halfSize][j + halfSize];

            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j + halfSize];
```

```
            B21[i][j] = B[i + halfSize][j];
            B22[i][j] = B[i + halfSize][j + halfSize];
        }
    }

    // Recursive calls for seven products
    vector<vector<int>> P1 = strassenMatrixMultiply(A11, matrixSubtraction(B12, B22));
    vector<vector<int>> P2 = strassenMatrixMultiply(matrixAddition(A11, A12), B22);
    vector<vector<int>> P3 = strassenMatrixMultiply(matrixAddition(A21, A22), B11);
    vector<vector<int>> P4 = strassenMatrixMultiply(A22, matrixSubtraction(B21, B11));
    vector<vector<int>> P5 = strassenMatrixMultiply(matrixAddition(A11, A22), matrixAdditi
on(B11, B22));
    vector<vector<int>> P6 = strassenMatrixMultiply(matrixSubtraction(A12, A22), matrixAdd
ition(B21, B22));
    vector<vector<int>> P7 = strassenMatrixMultiply(matrixSubtraction(A11, A21), matrixAdd
ition(B11, B12));

    // Calculate the resulting submatrices
    vector<vector<int>> C11 = matrixAddition(matrixSubtraction(matrixAddition(P5, P4), P
2), P6);
    vector<vector<int>> C12 = matrixAddition(P1, P2);
    vector<vector<int>> C21 = matrixAddition(P3, P4);
    vector<vector<int>> C22 = matrixSubtraction(matrixSubtraction(matrixAddition(P5, P1),
P3), P7);

    // Combine the resulting submatrices to obtain the final result
    vector<vector<int>> result(n, vector<int>(n, 0));

    for (int i = 0; i < halfSize; i++) {
        for (int j = 0; j < halfSize; j++) {
            result[i][j] = C11[i][j];
            result[i][j + halfSize] = C12[i][j];
            result[i + halfSize][j] = C21[i][j];
            result[i + halfSize][j + halfSize] = C22[i][j];
        }
    }

    return result;
}

int main

() {
    vector<vector<int>> A = {{1, 2}, {3, 4}};
    vector<vector<int>> B = {{5, 6}, {7, 8}};

    vector<vector<int>> C = strassenMatrixMultiply(A, B);

    cout << "Resultant Matrix:" << endl;
    for (int i = 0; i < C.size(); i++) {
        for (int j = 0; j < C[i].size(); j++) {
            cout << C[i][j] << " ";
```

```
        }
        cout << endl;
    }

    return 0;
}
```

This code implements Strassen's Matrix Multiplication and demonstrates the divide-and-conquer approach to multiplying matrices efficiently. The code splits the matrices, recursively performs multiplications, and combines the results to obtain the final matrix product. Keep in mind that the implementation may not be the most efficient for very large matrices, and practical implementations often include optimizations for cache usage and efficiency.

Unit 2

Greedy method

The Greedy Method is a problem-solving technique that involves making a series of choices at each step to arrive at a solution. In the Greedy Method, the choice made at each step is the one that appears to be the best option at that moment, without considering the consequences of that choice on future steps. The Greedy Method is suitable for solving optimization problems where the goal is to find the best solution from a set of feasible solutions.

The key components of the Greedy Method are:

1. **Greedy Choice Property**: At each step, make the choice that seems best at the moment. This choice should be locally optimal without considering the global consequences.
2. **Optimal Substructure**: The problem can be divided into smaller subproblems, and the choice of an optimal solution for the overall problem can be constructed from the optimal solutions of its subproblems.
3. **No Backtracking**: Once a choice is made, it cannot be reconsidered or changed.

While the Greedy Method can provide efficient solutions for many problems, it's important to note that it doesn't guarantee finding the globally optimal solution in all

cases. Some problems are not suited for the Greedy Method, as it can lead to suboptimal or incorrect results.

Here's a simple example of the Greedy Method in action:

Example: Coin Change Problem

Suppose you want to make change for a given amount of money using the fewest number of coins. You have coins of different denominations, and you want to find the optimal combination of coins to minimize the total number of coins used.

C++ Code Example for the Coin Change Problem using the Greedy Method:

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

using namespace std;

vector<int> coinChangeGreedy(vector<int>& coins, int amount) {
    vector<int> result;
    int n = coins.size();
    for (int i = n - 1; i >= 0; i--) {
        while (amount >= coins[i]) {
            amount -= coins[i];
            result.push_back(coins[i]);
        }
    }
    return result;
}

int main() {
    vector<int> coins = {1, 5, 10, 25};
    int amount = 63;

    vector<int> change = coinChangeGreedy(coins, amount);

    cout << "Coins used to make change for " << amount << ": ";
    for (int coin : change) {
        cout << coin << " ";
    }

    return 0;
}
```

The Greedy Method is used in this example to find the optimal combination of coins to make change for a given amount. It selects the largest denomination coins first and continues with smaller denominations until the desired amount is reached.

However, not all problems can be solved optimally using the Greedy Method. It's important to carefully analyze the problem and the specific choice made at each step to determine if the Greedy Method is appropriate. In cases where the Greedy Method doesn't provide an optimal solution, other algorithmic techniques may be needed.

Knapsack Problem

The Knapsack Problem is a classic optimization problem in computer science and mathematics. It comes in several variations, but the most common one is the 0/1 Knapsack Problem. In this problem, you are given a set of items, each with a weight and a value, and a knapsack with a maximum weight capacity. The goal is to select a combination of items to maximize their total value without exceeding the knapsack's weight limit.

The 0/1 Knapsack Problem can be solved using dynamic programming. The general idea is to build a table where each cell represents the maximum value that can be achieved with a specific weight capacity and a subset of items. Here's a high-level explanation of the algorithm:

1. Create a table (often a 2D array) with rows representing different items and columns representing different weight capacities (from 0 to the knapsack's maximum capacity).
2. Initialize the table with zeros.
3. Iterate over each item and weight capacity combination. For each cell, you have two choices:
    - Include the current item: Add its value to the maximum value achievable with the remaining weight capacity.
    - Exclude the current item: Use the maximum value achievable without this item.
4. Fill in the table by taking the maximum value from the above two choices.
5. The final cell in the table represents the maximum value achievable with the knapsack's weight capacity.
6. Trace back through the table to find the items that were selected to achieve the maximum value.

Here's a C++ code example for solving the 0/1 Knapsack Problem:

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

using namespace std;

int knapsack(int capacity, vector<int>& weights, vector<int>& values) {
    int n = weights.size();
    vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

    for (int i = 1; i <= n; i++) {
        for (int w = 1; w <= capacity; w++) {
            if (weights[i - 1] <= w) {
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i -
1]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    return dp[n][capacity];
}

int main() {
    int capacity = 10;
    vector<int> weights = {2, 3, 4, 5};
    vector<int> values = {3, 4, 5, 6};

    int maxValue = knapsack(capacity, weights, values);

    cout << "Maximum value: " << maxValue << endl;

    return 0;
}
```

This code finds the maximum value that can be obtained within a knapsack with a capacity of 10 units, given a set of items with weights and values. The result is printed as the maximum value that can be achieved.

Huffman Coding

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.

The variable-length codes assigned to input characters are **Prefix Codes**, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

See **this** for applications of Huffman Coding.

There are mainly two major parts in Huffman Coding

1. Build a Huffman Tree from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

Algorithm:

The method which is used to construct optimal prefix code is called **Huffman coding**.

This algorithm builds a tree in bottom up manner. We can denote this tree by **T**

Let, |c| be number of leaves

|c| -1 are number of operations required to merge the nodes. Q be the priority queue which can be used while constructing binary heap.

```
Algorithm Huffman (c)
{
   n= |c|

   Q = c
   for i<-1 to n-1

   do
   {
```

```
    temp <- get node ()

    left (temp] Get_min (Q) right [temp] Get Min (Q)

    a = left [templ b = right [temp]

    F [temp]<- f[a] + [b]

    insert (Q, temp)

  }

return Get_min (0)
}
```

Steps to build Huffman Tree

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.Let us understand the algorithm with an example:

```
character   Frequency
    a            5
    b           9
    c           12
    d           13
    e           16
    f           45
```

Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.

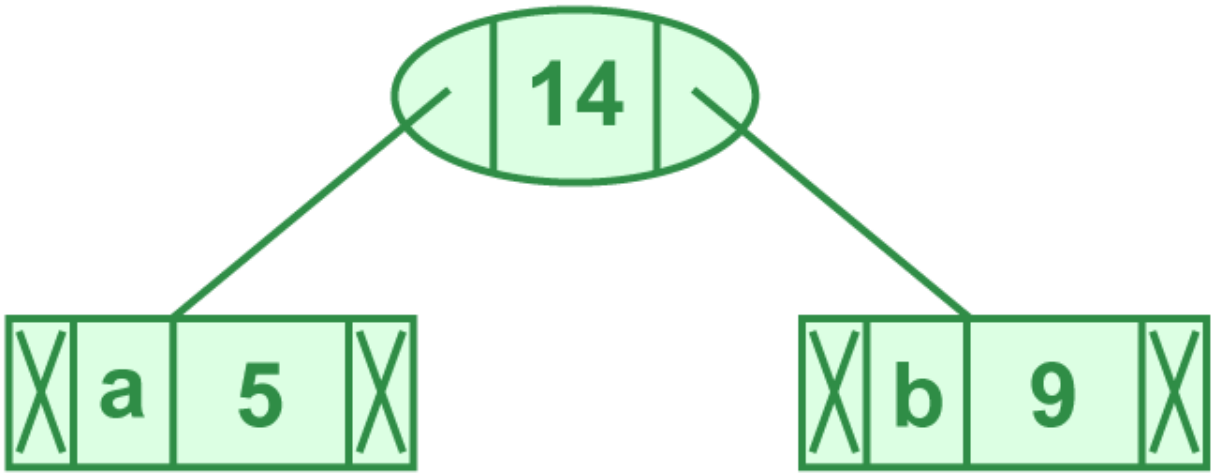


Illustration of step 2

Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

```
character           Frequency
       c               12
       d               13
 Internal Node         14
```

```
        e               16
        f                45
```

Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25

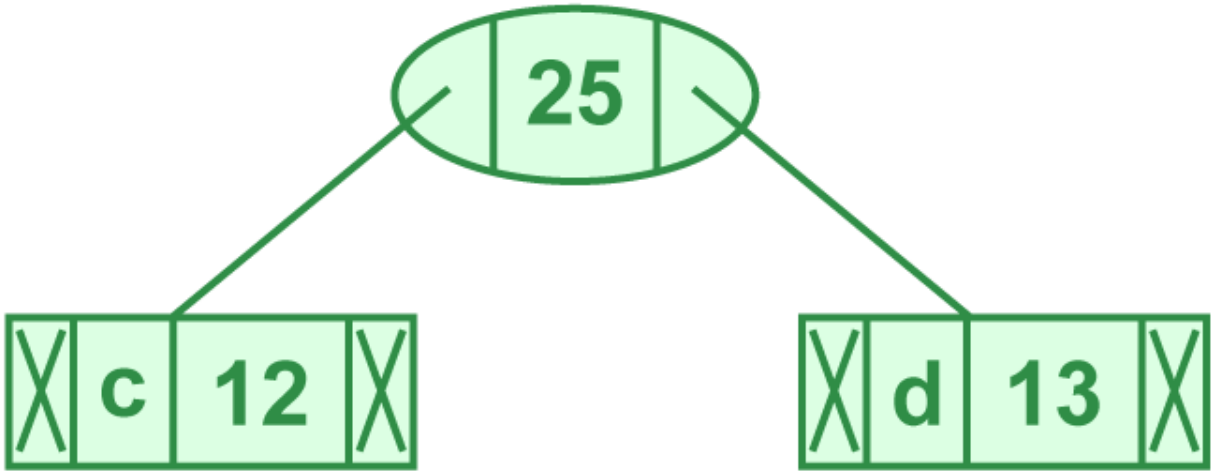


Illustration of step 3

Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes

```
character           Frequency
Internal Node          14
       e               16
Internal Node          25
       f               45
```

Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30

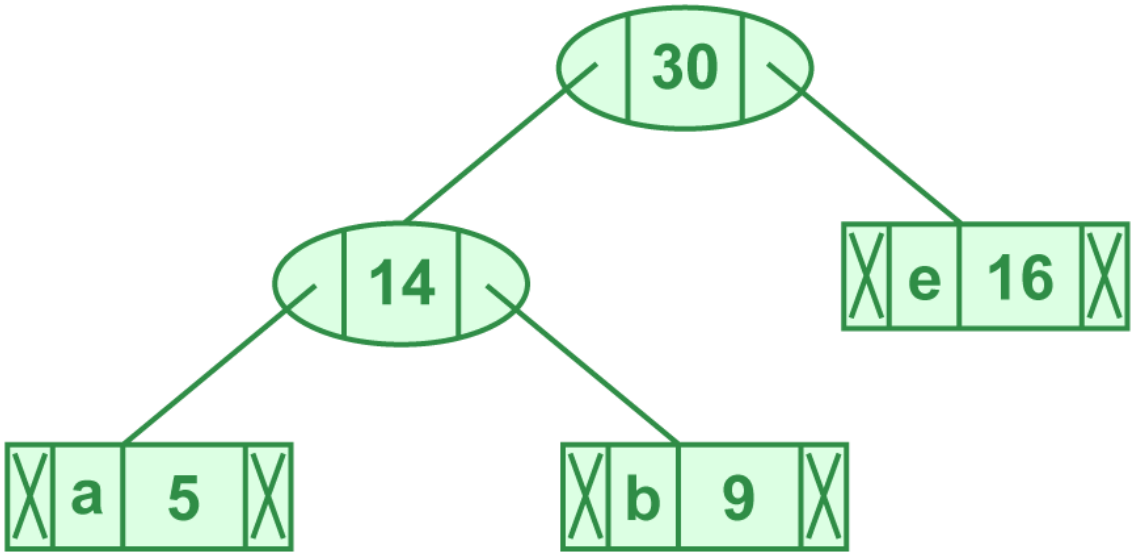


Illustration of step 4

Now min heap contains 3 nodes.

```
character           Frequency
Internal Node         25
Internal Node         30
      f               45
```

Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55

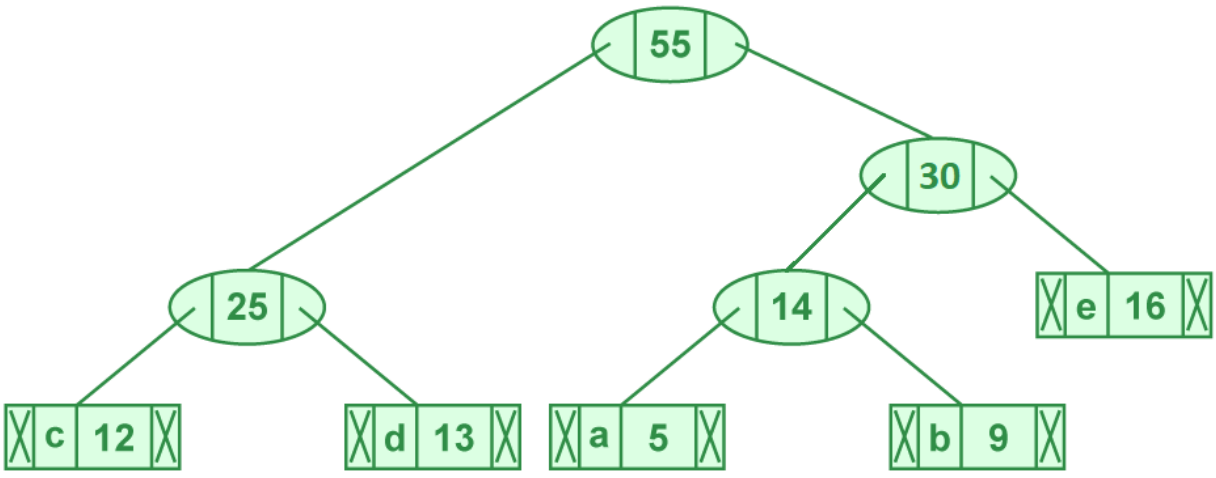


Illustration of step 5

Now min heap contains 2 nodes.

```
character     Frequency
       f         45
Internal Node    55
```

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100

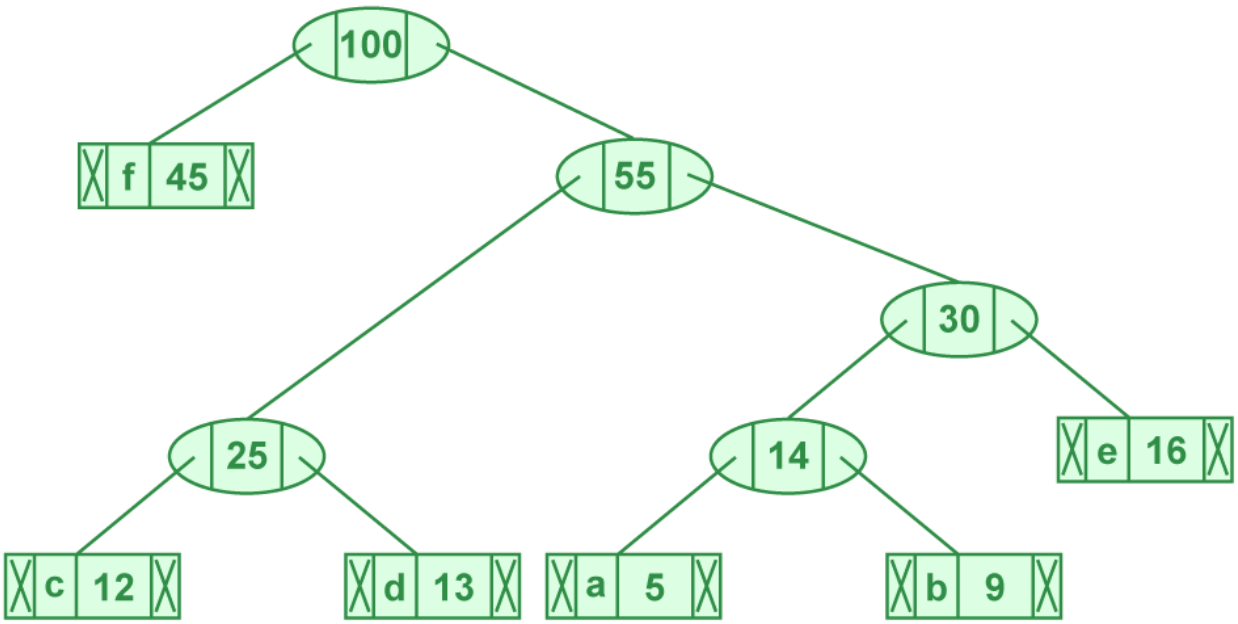


Illustration of step 6

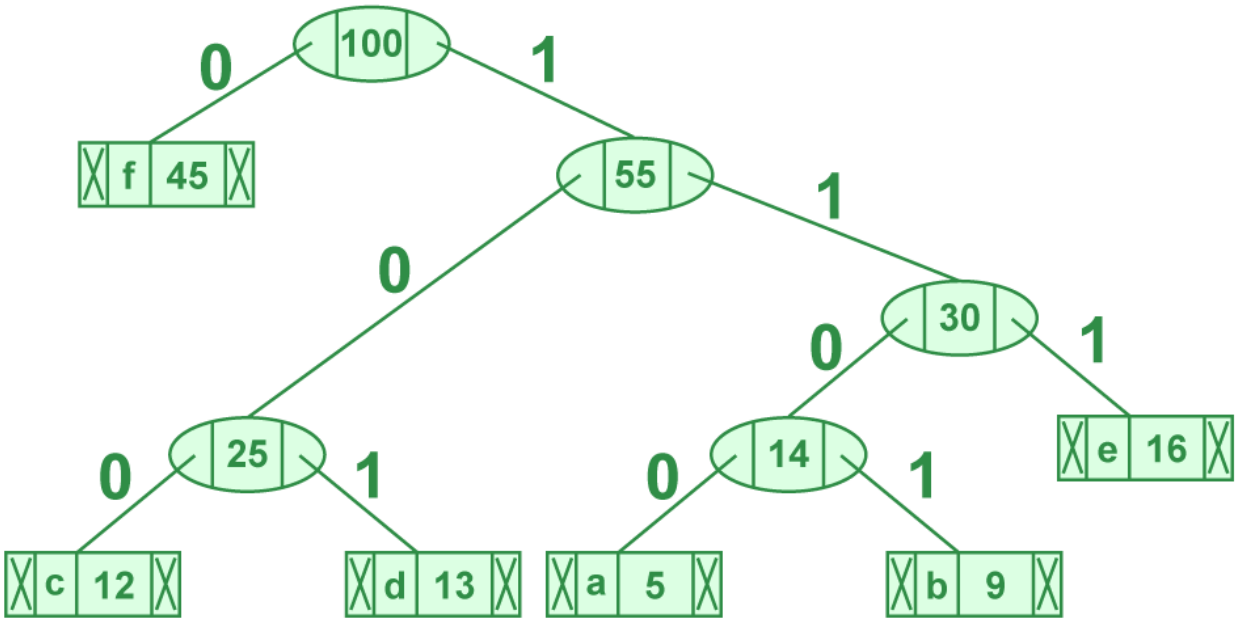
Now min heap contains only one node.

```
character      Frequency
Internal Node    100
```

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



Steps to print code from HuffmanTree

The codes are as follows:

```
character   code-word
    f          0
    c          100
    d          101
    a          1100
    b          1101
    e          111
```

Below is the implementation of above approach:

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

```
// Define the structure for a Huffman tree node
struct HuffmanNode {
    char data;
    int frequency;
    HuffmanNode* left;
    HuffmanNode* right;

    HuffmanNode(char data, int frequency) {
        this->data = data;
        this->frequency = frequency;
        left = nullptr;
        right = nullptr;
    }
};

// Comparison function for the priority queue
struct CompareHuffmanNode {
    bool operator()(HuffmanNode* a, HuffmanNode* b) {
        return a->frequency > b->frequency;
    }
};

// Function to build the Huffman tree and generate codes
void buildHuffmanTree(map<char, int> frequencies) {
    priority_queue<HuffmanNode*, vector<HuffmanNode*>, CompareHuffmanNode> pq;

    // Create a leaf node for each character and add them to the priority queue
    for (auto it = frequencies.begin(); it != frequencies.end(); it++) {
        HuffmanNode* node = new HuffmanNode(it->first, it->second);
        pq.push(node);
    }

    while (pq.size() > 1) {
        // Extract the two nodes with the lowest frequencies
        HuffmanNode* left = pq.top();
        pq.pop();
        HuffmanNode* right = pq.top();
        pq.pop();

        // Create a new internal node with the combined frequency
        int combinedFrequency = left->frequency + right->frequency;
        HuffmanNode* internalNode = new HuffmanNode('\\0', combinedFrequency);

        // Set the left and right children
        internalNode->left = left;
        internalNode->right = right;

        // Add the internal node back to the priority queue
        pq.push(internalNode);
    }
```

```
    // The remaining node in the priority queue is the root of the Huffman tree
    HuffmanNode* root = pq.top();

    // Generate Huffman codes for characters
    map<char, string> huffmanCodes;
    string code = "";
    generateHuffmanCodes(root, code, huffmanCodes);

    // Print the Huffman codes
    cout << "Huffman Codes:" << endl;
    for (auto it = huffmanCodes.begin(); it != huffmanCodes.end(); it++) {
        cout << it->first << ": " << it->second << endl;
    }
}

// Function to generate Huffman codes
void generateHuffmanCodes(HuffmanNode* root, string code, map<char, string>& huffmanCodes)
{
    if (!root)
        return;

    // Leaf node (character found)
    if (root->data != '\\0') {
        huffmanCodes[root->data] = code;
    }

    // Traverse left and append '0' to the code
    generateHuffmanCodes(root->left, code + "0", huffmanCodes);

    // Traverse right and append '1' to the code
    generateHuffmanCodes(root->right, code + "1", huffmanCodes);
}

int main() {
    map<char, int> frequencies = {
        {'a', 5},
        {'b', 9},
        {'c', 12},
        {'d', 13},
        {'e', 16},
        {'f', 45}
    };

    buildHuffmanTree(frequencies);

    return 0;
}
```

Output

```
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

Time complexity: O(nlogn) where n is the number of unique characters. If there are n nodes, extractMin() is called 2*(n – 1) times. extractMin() takes O(logn) time as it calls minHeapify(). So, the overall complexity is O(nlogn).

If the input array is sorted, there exists a linear time algorithm. We will soon be discussing this in our next post.

Space complexity :- O(N)

Applications of Huffman Coding:

1. They are used for transmitting fax and text.
2. They are used by conventional compression formats like PKZIP, GZIP, etc.
3. Multimedia codecs like JPEG, PNG, and MP3 use Huffman encoding(to be more precise the prefix codes).

It is useful in cases where there is a series of frequently occurring characters.

Job Sequencing Problem

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes a single unit of time, so the minimum possible deadline for any job is 1. Maximize the total profit if only one job can be scheduled at a time.

Examples:

Input: *Four Jobs with following deadlines and profits*

JobID Deadline Profit

a 4 20

b 1 10

c 1 40

d 1 30

Output: *Following is maximum profit sequence of jobs: c, a*

Input: *Five Jobs with following deadlines and profits*

JobID Deadline Profit

a 2 100

b 1 19

c 2 27

d 1 25

e 3 15

Output: *Following is maximum profit sequence of jobs: c, a, e*

Greedy approach for job sequencing problem:

> Greedily choose the jobs with maximum profit first, by sorting the jobs in decreasing order of their profit. This would help to maximize the total profit as choosing the job with maximum profit for every time slot will eventually maximize the total profit

Follow the given steps to solve the problem:

- Sort all jobs in decreasing order of profit.
- Iterate on jobs in decreasing order of profit.For each job , do the following :
  - Find a time slot i, such that slot is empty and i < deadline and i is greatest.Put the job in this slot and mark this slot filled.
  - If no such i exists, then ignore the job.

Below is the implementation of the above approach:

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

```
using namespace std;

// Structure to represent a job
struct Job {
    char id; // Job ID
    int deadline; // Deadline of job
    int profit; // Profit if job is finished before or on the deadline
};

// Comparator function for sorting jobs by profit in descending order
bool compareJobs(const Job& a, const Job& b) {
    return a.profit > b.profit;
}

// Function to find the maximum profit sequence of jobs
void findMaxProfitJobs(vector<Job>& jobs) {
    // Sort the jobs by profit in descending order
    sort(jobs.begin(), jobs.end(), compareJobs);

    int n = jobs.size();
    vector<char> result(n);
    vector<bool> slot(n, false);

    for (int i = 0; i < n; i++) {
        for (int j = min(n, jobs[i].deadline) - 1; j >= 0; j--) {
            if (!slot[j]) {
                result[j] = jobs[i].id;
                slot[j] = true;
                break;
            }
        }
    }

    cout << "Following is the maximum profit sequence of jobs: ";
    for (char job : result) {
        cout << job << " ";
    }
    cout << endl;
}

int main() {
    vector<Job> jobs = {
        { 'a', 4, 20 },
        { 'b', 1, 10 },
        { 'c', 1, 40 },
        { 'd', 1, 30 }
    };

    findMaxProfitJobs(jobs);

    vector<Job> moreJobs = {
        { 'a', 2, 100 },
```

```
        { 'b', 1, 19 },
        { 'c', 2, 27 },
        { 'd', 1, 25 },
        { 'e', 3, 15 }
    };

    findMaxProfitJobs(moreJobs);

    return 0;
}
```

Output

```
Following is maximum profit sequence of jobs
c a e
```

Time Complexity: O(N2)

Auxiliary Space: O(N)

Minimum Spanning Trees

A Minimum Spanning Tree (MST) is a tree that spans all the vertices of a connected, undirected graph while minimizing the sum of the edge weights. Prim's algorithm and Kruskal's algorithm are two well-known greedy algorithms for finding the Minimum Spanning Tree. I'll provide a C++ code example for each algorithm, along with a brief explanation.

Kruskal's Algorithm

Kruskal's algorithm starts with an empty set of edges and repeatedly adds the smallest edge that does not form a cycle with the edges already chosen. Here's the code for Kruskal's algorithm:

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

using namespace std;

// Structure to represent an edge
struct Edge {
    int src, dest, weight;
};
```

```
// Structure to represent a disjoint set
struct DisjointSet {
    int *parent, *rank;
    int n;

    DisjointSet(int n) {
        this->n = n;
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 0;
        }
    }

    int find(int u) {
        if (u != parent[u])
            parent[u] = find(parent[u]);
        return parent[u];
    }

    void unionSets(int u, int v) {
        u = find(u);
        v = find(v);
        if (u != v) {
            if (rank[u] < rank[v])
                swap(u, v);
            parent[v] = u;
            if (rank[u] == rank[v])
                rank[u]++;
        }
    }
};

// Function to find the MST using Kruskal's algorithm
void kruskalMST(vector<Edge> &edges, int V) {
    vector<Edge> result;
    sort(edges.begin(), edges.end(), [](Edge a, Edge b) {
        return a.weight < b.weight;
    });
    DisjointSet ds(V);

    for (Edge edge : edges) {
        int src = edge.src;
        int dest = edge.dest;
        int srcSet = ds.find(src);
        int destSet = ds.find(dest);
        if (srcSet != destSet) {
            result.push_back(edge);
            ds.unionSets(srcSet, destSet);
        }
```

```
    }

    cout << "Edges in the Minimum Spanning Tree:" << endl;
    for (Edge edge : result) {
        cout << edge.src << " - " << edge.dest << " : " << edge.weight << endl;
    }
}

int main() {
    int V = 4; // Number of vertices
    vector<Edge> edges = {
        {0, 1, 10},
        {0, 2, 6},
        {0, 3, 5},
        {1, 3, 15},
        {2, 3, 4}
    };

    kruskalMST(edges, V);

    return 0;
}
```

Kruskal's algorithm sorts the edges by weight, then iterates through them in ascending order, adding each edge to the MST if it doesn't create a cycle.

Prim's Algorithm

Prim's algorithm starts with an arbitrary vertex and repeatedly adds the minimum-weight edge connected to the set of vertices already in the MST. Here's the code for Prim's algorithm:

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

// Structure to represent an edge
struct Edge {
    int to, weight;
};

// Function to find the MST using Prim's algorithm
void primMST(vector<vector<Edge>> &graph, int V) {
    vector<bool> inMST(V, false);
    vector<int> parent(V, -1);
    vector<int> key(V, INT_MAX);
```

```
    key[0] = 0; // Start with the first vertex
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({0, 0}); // {key, vertex}

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

        inMST[u] = true;

        for (Edge &edge : graph[u]) {
            int v = edge.to;
            int weight = edge.weight;

            if (!inMST[v] && weight < key[v]) {
                key[v] = weight;
                parent[v] = u;
                pq.push({key[v], v});
            }
        }
    }

    cout << "Edges in the Minimum Spanning Tree:" << endl;
    for (int i = 1; i < V; i++) {
        cout << parent[i] << " - " << i << " : " << key[i] << endl;
    }
}

int main() {
    int V = 5; // Number of vertices
    vector<vector<Edge>> graph(V);

    // Add edges and weights to the graph
    graph[0].push_back({1, 2});
    graph[0].push_back({3, 6});
    graph[1].push_back({0, 2});
    graph[1].push_back({2, 3});
    graph[1].push_back({3, 8});
    graph[2].push_back({1, 3});
    graph[3].push_back({0, 6});
    graph[3].push_back({1, 8});
    graph[3].push_back({4, 9});
    graph[4].push_back({3, 9});

    primMST(graph, V);

    return 0;
}
```

Prim's algorithm starts with an initial vertex and iteratively selects the edge with the minimum weight to expand the MST. It uses a priority queue to keep track of the edges that connect the MST to the remaining vertices.

Single source path

Single-source shortest path problems involve finding the shortest path from a specified source vertex to all other vertices in a graph. The two most well-known algorithms for solving single-source shortest path problems are Dijkstra's algorithm and Bellman-Ford algorithm. I'll provide explanations and C++ code examples for both algorithms.

Dijkstra's Algorithm

Dijkstra's algorithm is used to find the shortest path from a source vertex to all other vertices in a weighted, directed or undirected graph. It works for non-negative edge weights and is known for its efficiency in such cases.

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

// Structure to represent an edge
struct Edge {
    int to, weight;
};

// Function to find the shortest paths from a source vertex using Dijkstra's algorithm
void dijkstra(vector<vector<Edge>> &graph, int V, int source) {
    vector<int> dist(V, INT_MAX); // Initialize distances to infinity
    dist[source] = 0; // Distance from source to itself is 0
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({0, source}); // {distance, vertex}

    while (!pq.empty()) {
        int u = pq.top().second;
        int udist = pq.top().first;
        pq.pop();

        if (udist < dist[u])
            continue;

        for (Edge &edge : graph[u]) {
            int v = edge.to;
```

```
            int weight = edge.weight;

            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }

    cout << "Shortest distances from source vertex " << source << " to all other vertice
s:" << endl;
    for (int i = 0; i < V; i++) {
        cout << "Vertex " << i << ": " << dist[i] << endl;
    }
}

int main() {
    int V = 5; // Number of vertices
    vector<vector<Edge>> graph(V);

    // Add edges and weights to the graph
    graph[0].push_back({1, 2});
    graph[0].push_back({3, 6});
    graph[1].push_back({0, 2});
    graph[1].push_back({2, 3});
    graph[1].push_back({3, 8});
    graph[2].push_back({1, 3});
    graph[3].push_back({0, 6});
    graph[3].push_back({1, 8});
    graph[3].push_back({4, 9});
    graph[4].push_back({3, 9});

    int source = 0; // Source vertex

    dijkstra(graph, V, source);

    return 0;
}
```

Dijkstra's algorithm uses a priority queue to select the vertex with the minimum distance from the source and updates distances to neighboring vertices.

Bellman-Ford Algorithm

Bellman-Ford algorithm is used to find the shortest path from a source vertex to all other vertices in a weighted, directed or undirected graph. Unlike Dijkstra's algorithm, Bellman-Ford can handle graphs with negative weight edges and detect negative weight cycles.

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

// Structure to represent an edge
struct Edge {
    int from, to, weight;
};

// Function to find the shortest paths from a source vertex using Bellman-Ford algorithm
void bellmanFord(vector<Edge> &edges, int V, int source) {
    vector<int> dist(V, INT_MAX); // Initialize distances to infinity
    dist[source] = 0; // Distance from source to itself is 0

    // Relax edges V-1 times to find the shortest paths
    for (int i = 0; i < V - 1; i++) {
        for (Edge &edge : edges) {
            int u = edge.from;
            int v = edge.to;
            int weight = edge.weight;

            if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
            }
        }
    }

    // Check for negative weight cycles
    for (Edge &edge : edges) {
        int u = edge.from;
        int v = edge.to;
        int weight = edge.weight;

        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            cout << "Graph contains a negative weight cycle!" << endl;
            return;
        }
    }

    cout << "Shortest distances from source vertex " << source << " to all other vertice
s:" << endl;
    for (int i = 0; i < V; i++) {
        cout << "Vertex " << i << ": " << dist[i] << endl;
    }
}

int main() {
    int V = 5; // Number of vertices
    vector<Edge> edges = {
```

```
            {0, 1, 2},
            {0, 3, 6},
            {1, 2, 3},
            {1, 3, 8},
            {1, 4, 4},
            {2, 4, 9},
            {3, 4, 9}
        };

        int source = 0; // Source vertex

        bellmanFord(edges, V, source);

        return 0;
    }
```

Bellman-Ford algorithm iteratively relaxes edges V-1 times to find the shortest paths, and it can also detect negative weight cycles in the graph.

Backtracking

Backtracking is a general algorithmic technique used for solving problems by incrementally building a solution. It is especially useful for solving problems where you need to find one or more solutions among a large number of possibilities. The key idea behind backtracking is to explore potential solutions and backtrack (undoing the last step) when a solution is not viable or when all possibilities have been explored.

Here is a general method to implement backtracking:

1. Define the Problem: Clearly define the problem you want to solve and understand the constraints and requirements.
2. Create a Recursive Function: Create a recursive function that explores potential solutions incrementally. This function will be the core of your backtracking algorithm.
3. Base Case: Define one or more base cases for the recursive function. Base cases are situations where a solution has been found or where backtracking should stop.
4. Explore Possibilities: In the recursive function, explore different possibilities or choices step by step. This may involve trying different options, making decisions, or selecting elements from a set.
5. Constraints and Pruning: Apply constraints and pruning techniques to reduce the search space. If a partial solution violates a constraint, stop exploring that path

(backtrack).

6. Check for a Solution: Check if the current partial solution is a valid solution to the problem. If it is, record the solution or take appropriate actions.
7. Recursion: Recursively call the function to explore the next step or choice in the solution space.
8. Backtrack: When there are no more choices to explore or a solution is not possible, backtrack by returning from the current recursive call and undoing the last step.
9. Explore All Possibilities: Continue the process of exploration, recursion, and backtracking until you find a valid solution or exhaust all possibilities.
10. Record Solutions: If you need to find multiple solutions, record each valid solution when it is found.
11. Optimize: Implement optimizations and strategies to improve the efficiency of the backtracking algorithm. These optimizations may include avoiding unnecessary work or pruning the search space early.
12. Handle Output: If you need to return or use the solutions, make sure to collect and process the results appropriately.

Here's a simple example of a backtracking problem: finding all permutations of a set of elements.

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

// Function to generate all permutations of a set of elements
void generatePermutations(vector<int>& nums, vector<int>& current, vector<vector<int>>& re
sult) {
    if (current.size() == nums.size()) {
        result.push_back(current);
        return;
    }

    for (int i = 0; i < nums.size(); i++) {
        if (find(current.begin(), current.end(), nums[i]) == current.end()) {
            current.push_back(nums[i]);
            generatePermutations(nums, current, result);
            current.pop_back();
        }
    }
```

```
}

int main() {
    vector<int> nums = {1, 2, 3};
    vector<vector<int>> result;
    vector<int> current;

    generatePermutations(nums, current, result);

    cout << "All permutations of {1, 2, 3} are:" << endl;
    for (vector<int> perm : result) {
        for (int num : perm) {
            cout << num << " ";
        }
        cout << endl;
    }

    return 0;
}
```

In this example, we use backtracking to generate all permutations of a set of elements. The backtracking algorithm explores all possible arrangements and records valid solutions.

8Queen

The 8 Queens Problem is a classic problem in computer science and combinatorial optimization. The goal is to place eight queens on an 8x8 chessboard in such a way that no two queens threaten each other. This means that no two queens can share the same row, column, or diagonal. The problem can be solved using a backtracking algorithm.

Here's a C++ implementation of the 8 Queens Problem, followed by an analysis of its time and space complexity:

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

const int N = 8; // The size of the chessboard

// Function to print the chessboard
void printBoard(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            cout << board[i][j] << " ";
        }
```

```
        cout << endl;
    }
}

// Function to check if it's safe to place a queen at board[row][col]
bool isSafe(int board[N][N], int row, int col) {
    // Check the left side of the row
    for (int i = 0; i < col; i++) {
        if (board[row][i])
            return false;
    }

    // Check upper diagonal on the left
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j])
            return false;
    }

    // Check lower diagonal on the left
    for (int i = row, j = col; i < N && j >= 0; i++, j--) {
        if (board[i][j])
            return false;
    }

    return true;
}

// Recursive function to solve the 8 Queens Problem
bool solveNQueens(int board[N][N], int col) {
    if (col >= N) {
        // All queens are placed, the problem is solved
        printBoard(board);
        return true;
    }

    bool res = false;
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1; // Place the queen
            res = solveNQueens(board, col + 1) || res; // Recur to the next column
            board[i][col] = 0; // Backtrack
        }
    }
    return res;
}

int main() {
    int board[N][N] = {0}; // Initialize the chessboard
    if (!solveNQueens(board, 0)) {
        cout << "No solution exists." << endl;
    }
```

```
    return 0;
}
```

Time Complexity Analysis:

- The backtracking algorithm explores possibilities for each column, and for each column, it tries all rows (N possibilities). So, the time complexity is O(N^N), which is exponential.
- In practice, the time complexity is much better than the worst-case due to early pruning and heuristics.

Space Complexity Analysis:

- The space complexity is O(N^2) because it requires an NxN chessboard to represent the solution.

Keep in mind that while the time complexity is exponential, this problem can be efficiently solved for small values of N using backtracking algorithms, and there are also more efficient algorithms for solving larger instances of the N-Queens problem.

Graph colouring

Graph coloring is a combinatorial optimization problem that involves assigning colors to the vertices of a graph such that no two adjacent vertices share the same color. The objective is to minimize the number of colors used. This problem has various applications, including scheduling, register allocation in compilers, and more. It can be solved using different algorithms, including backtracking and greedy algorithms.

Graph Coloring Algorithm

Here's a C++ implementation of the graph coloring problem using a backtracking algorithm:

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

const int V = 4; // Number of vertices

// Function to check if it's safe to color vertex v with color c
bool isSafe(int v, vector<vector<int>>& graph, vector<int>& color, int c) {
```

```
    for (int u = 0; u < V; u++) {
        if (graph[v][u] && color[u] == c) {
            return false;
        }
    }
    return true;
}

// Recursive function to solve the graph coloring problem
bool graphColoringUtil(vector<vector<int>>& graph, int m, vector<int>& color, int v) {
    if (v == V) {
        return true; // All vertices are colored
    }

    for (int c = 1; c <= m; c++) {
        if (isSafe(v, graph, color, c)) {
            color[v] = c;
            if (graphColoringUtil(graph, m, color, v + 1)) {
                return true;
            }
            color[v] = 0; // Backtrack
        }
    }
    return false;
}

// Function to solve the graph coloring problem
void graphColoring(vector<vector<int>>& graph, int m) {
    vector<int> color(V, 0);

    if (graphColoringUtil(graph, m, color, 0)) {
        cout << "Solution exists. The coloring is:" << endl;
        for (int i = 0; i < V; i++) {
            cout << "Vertex " << i << " is colored with color " << color[i] << endl;
        }
    } else {
        cout << "No solution exists." << endl;
    }
}

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

    int m = 3; // Number of colors

    graphColoring(graph, m);
```

```
    return 0;
}
```

Time Complexity Analysis:

- The time complexity of the graph coloring problem using the backtracking algorithm is exponential. It is O(m^V), where V is the number of vertices and m is the number of colors. In the worst case, it explores all possible combinations of colors for each vertex.

Space Complexity Analysis:

- The space complexity is O(V), where V is the number of vertices. It is required to store the color of each vertex.

Keep in mind that graph coloring is an NP-hard problem, and finding an optimal solution can be computationally expensive, especially for large graphs. There are heuristic and approximation algorithms that can provide good solutions in a reasonable amount of time, but they may not always find the optimal solution.

Hamiltonian Cycle

A Hamiltonian cycle is a cycle in a graph that visits every vertex exactly once and returns to the starting vertex. Determining whether a Hamiltonian cycle exists in a given graph is an NP-complete problem. There is no known polynomial-time algorithm for solving the Hamiltonian cycle problem for arbitrary graphs. However, there are algorithms and heuristics that can be used to find Hamiltonian cycles in specific types of graphs. Let's discuss this problem along with its time and space complexity.

Hamiltonian Cycle Problem

The Hamiltonian cycle problem is typically solved using backtracking or dynamic programming techniques. Here's a high-level overview of the backtracking approach:

1. Start at an arbitrary vertex as the initial vertex.
2. At each step, try to extend the current path by selecting an unvisited neighbor of the current vertex. If no unvisited neighbor is available, backtrack to the previous vertex.
3. Continue this process until all vertices have been visited and the path returns to the initial vertex, forming a cycle.

The backtracking algorithm explores all possible paths in the graph to find a Hamiltonian cycle. If a valid Hamiltonian cycle is found, the algorithm stops; otherwise, it backtracks and explores other paths.

Time Complexity

The time complexity of finding a Hamiltonian cycle using a backtracking algorithm depends on the number of vertices in the graph and the specific structure of the graph. In the worst case, where all paths need to be explored, the time complexity is exponential, O(N!), where N is the number of vertices. This is because there are N! possible permutations of vertices to consider.

However, the practical performance of the backtracking algorithm can vary significantly based on the graph's structure. In some cases, it may find a Hamiltonian cycle relatively quickly, while in other cases, it may take a very long time.

Space Complexity

The space complexity of the backtracking algorithm is primarily determined by the space required for storing the current path and whether additional data structures, such as a boolean array to track visited vertices, are used. The space complexity is typically O(N), where N is the number of vertices.

Keep in mind that finding Hamiltonian cycles is a computationally challenging problem, and in many cases, heuristic algorithms or approximation algorithms are used to find near-optimal solutions for large graphs in a reasonable amount of time. These heuristics aim to find a cycle that is close to being Hamiltonian but may not guarantee an exact Hamiltonian cycle.

Finding a Hamiltonian cycle in a graph is a complex problem, and the code can get quite lengthy. I'll provide a simplified example in C++ using a backtracking approach. This example assumes an undirected graph represented as an adjacency matrix.

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

const int V = 5; // Number of vertices in the graph

// Function to check if vertex v can be added to the Hamiltonian cycle
bool isSafe(int v, vector<int> path, vector<vector<int>> graph, int pos) {
    if (!graph[path[pos - 1]][v]) {
```

```
        return false; // There's no edge between the previous vertex and v
    }

    for (int i = 0; i < pos; i++) {
        if (path[i] == v) {
            return false; // Vertex v is already in the path
        }
    }

    return true;
}

// Recursive function to find a Hamiltonian cycle
bool hamiltonianCycleUtil(vector<vector<int>> graph, vector<int>& path, int pos) {
    if (pos == V) {
        // All vertices are included in the path, check if it's a Hamiltonian cycle
        if (graph[path[pos - 1]][path[0]]) {
            return true;
        }
        return false;
    }

    for (int v = 1; v < V; v++) {
        if (isSafe(v, path, graph, pos)) {
            path[pos] = v;
            if (hamiltonianCycleUtil(graph, path, pos + 1)) {
                return true;
            }
            path[pos] = -1; // Backtrack
        }
    }

    return false;
}

// Function to find a Hamiltonian cycle in the graph
void findHamiltonianCycle(vector<vector<int>> graph) {
    vector<int> path(V, -1);
    path[0] = 0; // Start with the first vertex as the initial vertex

    if (hamiltonianCycleUtil(graph, path, 1)) {
        cout << "A Hamiltonian cycle exists. Path:" << endl;
        for (int v : path) {
            cout << v << " ";
        }
        cout << path[0] << endl;
    } else {
        cout << "No Hamiltonian cycle exists." << endl;
    }
}

int main() {
```

```
    vector<vector<int>> graph = {
        {0, 1, 0, 1, 0},
        {1, 0, 1, 1, 1},
        {0, 1, 0, 0, 1},
        {1, 1, 0, 0, 1},
        {0, 1, 1, 1, 0}
    };

    findHamiltonianCycle(graph);

    return 0;
}
```

This code attempts to find a Hamiltonian cycle in a simple graph with five vertices. It uses a backtracking approach to explore different paths through the graph. Please note that this code is a simplified example and may not work for more complex graphs. Solving the Hamiltonian cycle problem efficiently for arbitrary graphs is a challenging task.

updated version here: https://yashnote.notion.site/Design-and-Analysis-of-Algorithms-c81977a159da4e12bc8971aa44fee7d7?pvs=4