ITECH WORLD AKTU

SUBJECT NAME: DESIGN AND ANALYSIS OF ALGORITHM (DAA) SUBJECT CODE: BCS503

UNIT 1: INTRODUCTION

Syllabus

1. Introduction: Algorithms, Analyzing Algorithms, Complexity of Algorithms, Growth of Functions.
2. Performance Measurements.
3. Sorting and Order Statistics:
   - Shell Sort
   - Quick Sort
   - Merge Sort
   - Heap Sort
4. Comparison of Sorting Algorithms.
5. Sorting in Linear Time.

---

1 Introduction to Algorithms

1.1 Definition of an Algorithm

An **algorithm** is a clear and precise sequence of instructions designed to solve a specific problem or perform a computation. It provides a step-by-step method to achieve a desired result. .

1.2 Difference Between Algorithm and Pseudocode

Algorithm	Pseudocode

Algorithm to find the GCD of two numbers:
• Start with two numbers, say a and b.
• If $b = 0$, the GCD is a. Stop.
• If $b \neq 0$, assign $a = b$, $b = a$ mod b.
• Repeat the above steps until $b = 0$.
	Pseudocode for GCD:
	`function GCD(a, b)`
	`  while b  0`
	`  temp := b`
	`  b := a mod b`
	`  a := temp`
	`  return a`
	`end function`

1.3 Characteristics of an Algorithm

An effective algorithm must have the following characteristics:

1. **Finiteness:** The algorithm must terminate after a finite number of steps.
2. **Definiteness:** Each step must be precisely defined; the actions to be executed should be clear and unambiguous.
3. **Input:** The algorithm should have zero or more well-defined inputs.
4. **Output:** The algorithm should produce one or more outputs.
5. **Effectiveness:** Each step of the algorithm must be simple enough to be carried out exactly in a finite amount of time.

1.4 Difference Between Algorithm and Pseudocode

Algorithm	Pseudocode
A step-by-step procedure to solve a problem, expressed in plain language or mathematical form.	A representation of an algorithm using structured, human-readable code-like syntax.
Focuses on the logical sequence of steps to solve a problem.	Focuses on illustrating the algorithm using a syntax closer to a programming language.
Language independent; can be written in natural language or mathematical notation.	Language dependent; mimics the structure and syntax of programming languages.
More abstract and high-level.	More concrete and closer to actual code implementation.
No need for specific formatting rules.	Requires a consistent syntax, but not as strict as actual programming languages.

1.5 Analyzing Algorithms

Analyzing an algorithm involves understanding its **time complexity** and **space complexity**. This analysis helps determine how efficiently an algorithm performs, especially in terms of execution time and memory usage.

What is Analysis of Algorithms?

- **Definition:** The process of determining the computational complexity of algorithms, including both the time complexity (how the runtime of the algorithm scales with the size of input) and space complexity (how the memory requirement grows with input size).
- **Purpose:** To evaluate the efficiency of an algorithm to ensure optimal performance in terms of time and space.
- **Types:** Analyzing algorithms typically involves two main types of complexities:
  - **Time Complexity:** Measures the total time required by the algorithm to complete as a function of the input size.
  - **Space Complexity:** Measures the total amount of memory space required by the algorithm during its execution.

Example:

- Analyzing the time complexity of the Binary Search algorithm.

2 Complexity of Algorithms

2.1 Time Complexity

Time Complexity is the computational complexity that describes the amount of time it takes to run an algorithm as a function of the length of the input.

Cases of Time Complexity:

- **Best Case:** The minimum time required for the algorithm to complete, given the most favorable input. Example: In Binary Search, the best-case time complexity is $O(1)$ when the target element is the middle element.
- **Average Case:** The expected time required for the algorithm to complete, averaged over all possible inputs. Example: For Quick Sort, the average-case time complexity is $O(n \log n)$.
- **Worst Case:** The maximum time required for the algorithm to complete, given the least favorable input. Example: In Linear Search, the worst-case time complexity is $O(n)$ when the element is not present in the array.

Example:

- Time complexity of the Merge Sort algorithm is $O(n \log n)$.

2.2 Space Complexity

Space Complexity refers to the total amount of memory space required by an algorithm to complete its execution.

Cases of Space Complexity:

- **Auxiliary Space:** Extra space or temporary space used by an algorithm.
- **Total Space:** The total space used by the algorithm, including both the input and auxiliary space.

Example:

- Space complexity of the Quick Sort algorithm is $O(n)$.

3 Growth of Functions

Growth of Functions describes how the time or space requirements of an algorithm grow with the size of the input. The growth rate helps in understanding the efficiency of an algorithm.

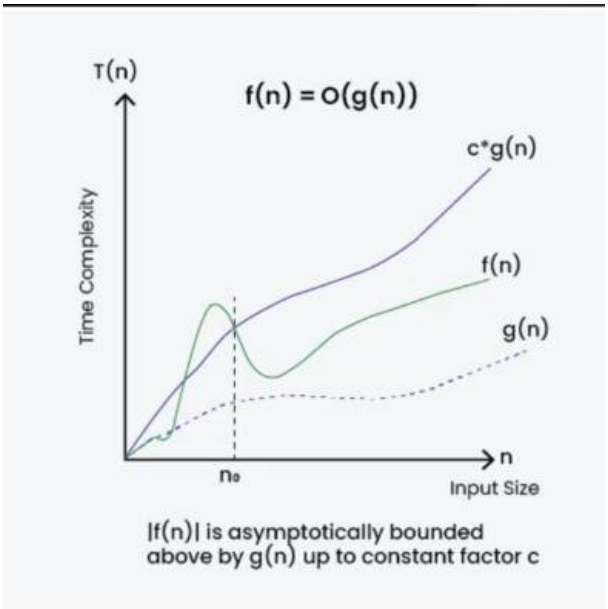
Examples:

- **Polynomial Growth:** $O(n^2)$ - Example: Bubble Sort algorithm.
- **Exponential Growth:** $O(2^n)$ - Example: Recursive Fibonacci algorithm.

3.1 Big-O Notation

Big-O notation, denoted as $O(f(n))$, describes the upper bound of an algorithm's time or space complexity. It gives the worst-case scenario of the growth rate of a function.

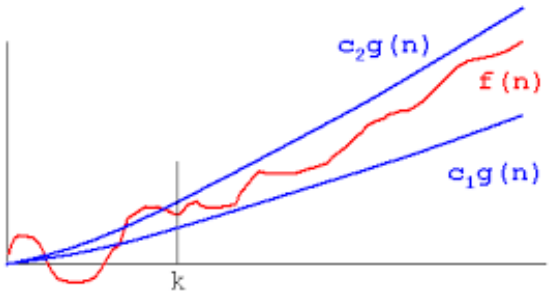
Graphical Representation:



3.2 Theta Notation

Theta notation, denoted as $\Theta(f(n))$, describes the tight bound of an algorithm's time or space complexity. It represents both the upper and lower bounds, capturing the exact growth rate.

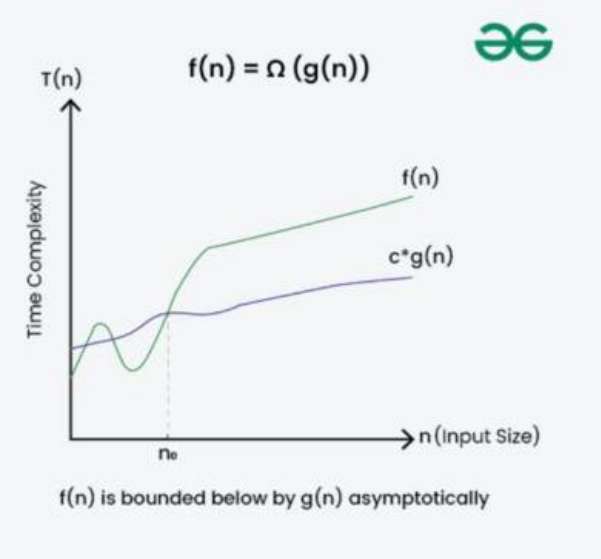
Graphical Representation:



3.3 Omega Notation

Omega notation, denoted as $\Omega(f(n))$, describes the lower bound of an algorithm's time or space complexity. It gives the best-case scenario of the growth rate of a function.

Graphical Representation:



3.4 Numerical Example

Problem: If $f(n) = 100 \cdot 2^n + n^5 + n$, show that $f(n) = O(2^n)$.

Solution:

- The term $100 \cdot 2^n$ dominates as $n \to \infty$.
- n^5 and n grow much slower compared to 2^n.
- Therefore, $f(n) = 100 \cdot 2^n + n^5 + n = O(2^n)$.

3.5 Recurrences

Recurrence relations are equations that express a sequence in terms of its preceding terms. In the context of Data Structures and Algorithms (DAA), a recurrence relation often represents the time complexity of a recursive algorithm. For example, the time complexity $T(n)$ of a recursive function can be expressed as:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

where:

- a is the number of subproblems in the recursion,
- b is the factor by which the subproblem size is reduced in each recursive call,
- $f(n)$ represents the cost of the work done outside of the recursive calls.

There are several methods to solve recurrence relations:

1. **Substitution Method:** Guess the form of the solution and use mathematical induction to prove it.
2. **Recursion Tree Method:** Visualize the recurrence as a tree where each node represents the cost of a recursive call and its children represent the costs of the subsequent subproblems.

3. **Master Theorem:** Provides a direct way to find the time complexity of recurrences of the form $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ by comparing $f(n)$ to $n^{\log_b a}$.

3.6 Master Theorem

The **Master Theorem** provides a solution for the time complexity of divide-and-conquer algorithms. It applies to recurrence relations of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where:

- $a \geq 1$ and $b > 1$ are constants.
- $f(n)$ is an asymptotically positive function.

Cases of the Master Theorem:

1. **Case 1:** If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. **Case 2:** If $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ for some $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$.
3. **Case 3:** If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$ and large n, then $T(n) = \Theta(f(n))$.

Example:

- Consider the recurrence $T(n) = 2T\left(\frac{n}{2}\right) + n$.
- Here, $a = 2$, $b = 2$, and $f(n) = n$.
- $\log_b a = \log_2 2 = 1$.
- $f(n) = n = \Theta(n^{\log_2 2}) = \Theta(n^1)$, so it matches Case 2.
- Therefore, $T(n) = \Theta(n \log n)$.

Question 1.6

Solve the recurrence relation:

$$T(n) = 7T\left(\frac{n}{2}\right) + n^2$$

Now, consider another algorithm with the recurrence:

$$T'(n) = aT'\left(\frac{n}{4}\right) + n^2$$

Find the largest integer a such that the algorithm T' runs faster than the first algorithm.

Solution:

$$T(n) = 7T\left(\frac{n}{2}\right) + n^2 \quad \text{and} \quad T'(n) = aT'\left(\frac{n}{4}\right) + n^2$$

By comparing, we have:

$$a = 7, \quad b = 2, \quad f(n) = n^2$$

Using Master's theorem:

$$n^{\log_b a} = n^{\log_2 7} = n^{2.81}$$

Case 1 of Master's theorem applies:

$$f(n) = O(n^{\log_b a - \epsilon}) = O(n^{2.81-\epsilon}) = O(n^2)$$

Thus,

$$T(n) = \Theta(n^{2.81})$$

Now for the second recurrence:

$$\frac{\log 7}{\log 2} = \frac{\log a}{\log 4} \quad \Rightarrow \quad \log a = \frac{\log 7}{\log 2} \times \log 4 = 1.6902$$

Taking antilog, we get:

$$a = 48.015$$

Thus, for $a = 49$, algorithm A' will have the same complexity as A. The largest integer a such that A' is faster than A is:

$$a = 48$$

Question 1.7

Solve the recurrence relation:

$$T(n) = 7T\left(\frac{n}{3}\right) + n^2$$

Now, consider another algorithm with the recurrence:

$$S(n) = aS\left(\frac{n}{9}\right) + n^2$$

Find the largest integer a such that the algorithm S runs faster than the first algorithm.

Solution:

$$T(n) = 7T\left(\frac{n}{3}\right) + n^2 \quad \text{and} \quad S(n) = aS\left(\frac{n}{9}\right) + n^2$$

Comparing the equations:

$$a = 7, \quad b = 3, \quad f(n) = n^2$$

Using Master's theorem:

$$n^{\log_b a} = n^{\log_3 7} = n^{1.771}$$

Case 3 of Master's theorem applies:

$$f(n) = \Omega(n^{\log_b a + \epsilon}) = \Omega(n^{1.771+\epsilon}) = \Omega(n^2)$$

Thus, the complexity is:

$$T(n) = \Theta(n^2)$$

For algorithm B, we get:

$$n^{\log_9 a} = n^{\log_9 81} = n^2$$

Thus, for $a = 81$, both algorithms have the same complexity.

If $a > 81$, algorithm B has a higher complexity than A:

$$S(n) = \Theta(n^2 \log n) > T(n)$$

Therefore, algorithm B can never be faster than A.

Question 1.8

Solve the recurrence relation:

$$T(n) = T(\sqrt{n}) + O(\log n)$$

Solution:

Let:

$$m = \log n \quad \text{and} \quad n = 2^m \quad \Rightarrow \quad n^{1/2} = 2^{m/2}$$

Then:

$$T(2^m) = T(2^{m/2}) + O(\log 2^m) \quad \text{Let} \quad x(m) = T(2^m)$$

Substituting into the equation:

$$x(m) = x\left(\frac{m}{2}\right) + O(m)$$

The solution is:

$$x(m) = \Theta(m \log m) \quad \Rightarrow \quad T(n) = \Theta(\log n \log \log n)$$

Recursion:

Recursion is a process where a function calls itself either directly or indirectly to solve a problem. In recursion, a problem is divided into smaller instances of the same problem, and solutions to these smaller instances are combined to solve the original problem. Recursion typically involves two main parts:

- **Base Case:** A condition under which the recursion stops.
- **Recursive Case:** The part where the function calls itself to break the problem into smaller instances.

Example of Recursion: Let's take the example of calculating the factorial of a number n, which is defined as:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

The recursive definition of factorial is:

$$\text{factorial}(n) = \begin{cases} 1, & \text{if } n = 0 \\ n \times \text{factorial}(n-1), & \text{if } n > 0 \end{cases}$$

In this example, the base case is factorial(0) = 1, and the recursive case is $n \times$ factorial$(n-1)$.

Recursion Tree:

A recursion tree is a tree representation of the recursive calls made by a recursive algorithm. Each node represents a function call, and its children represent the subsequent recursive calls. The depth of the tree represents the depth of recursion.

3.7 Sorting Algorithms

3.7.1 Shell Sort

Shell sort is an in-place comparison sort algorithm that extends the basic insertion sort algorithm by allowing exchanges of elements that are far apart. The main idea is to rearrange the elements so that elements that are far apart are sorted before doing a finer sort using insertion sort.

Shell sort improves the performance of insertion sort by breaking the original list into sublists based on a gap sequence and then sorting each sublist using insertion sort. This allows the algorithm to move elements more efficiently, especially when they are far apart from their correct positions.

Algorithm:

1. Start with a large gap between elements. A commonly used gap sequence is to divide the length of the list by 2 repeatedly until the gap is 1.
2. For each gap size, go through the list and compare elements that are that gap distance apart.
3. Use insertion sort to sort the sublists created by these gaps.
4. Continue reducing the gap until it becomes 1. When the gap is 1, the list is fully sorted by insertion sort.

Shell Sort Algorithm (Pseudocode):

```
shellSort(arr, n):
    gap = n // 2  # Initialize the gap size
    while gap > 0:
        for i = gap to n-1:
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j = j - gap
            arr[j] = temp
        gap = gap // 2
```

Example: Let's sort the array $[12, 34, 54, 2, 3]$ using Shell sort.

Step 1: Initial array

$$[12, 34, 54, 2, 3]$$

1. Start with gap $= 5//2 = 2$, meaning the array will be divided into sublists based on the gap 2.

- Compare elements at index 0 and 2: $[12, 54]$. No change since $12 < 54$.
- Compare elements at index 1 and 3: $[34, 2]$. Swap since $34 > 2$, resulting in:

$$[12, 2, 54, 34, 3]$$

- Compare elements at index 2 and 4: $[54, 3]$. Swap since $54 > 3$, resulting in:

$$[12, 2, 3, 34, 54]$$

Step 2: After first pass with gap 2

$$[12, 2, 3, 34, 54]$$

2. Reduce gap to 1: gap $= 2//2 = 1$. Now we perform insertion sort on the whole array:

- Compare index 0 and 1: $[12, 2]$. Swap since $12 > 2$, resulting in:

$$[2, 12, 3, 34, 54]$$

- Compare index 1 and 2: $[12, 3]$. Swap since $12 > 3$, resulting in:

$$[2, 3, 12, 34, 54]$$

- Compare index 2 and 3: $[12, 34]$. No change.
- Compare index 3 and 4: $[34, 54]$. No change.

Step 3: After final pass with gap 1

$$[2, 3, 12, 34, 54]$$

At this point, the array is sorted.
Key Insights:

- Shell sort is more efficient than insertion sort for large lists, especially when elements are far from their final positions.
- The efficiency depends on the choice of the gap sequence. A commonly used sequence is gap $= n//2$, reducing until gap equals 1.

3.7.2 Quick Sort

Quick Sort is a divide-and-conquer algorithm that sorts an array by partitioning it into two sub-arrays around a pivot element. The sub-arrays are then sorted recursively.
Algorithm:

1. **Choose a Pivot:** Select an element from the array to act as the pivot.
2. **Partition:** Rearrange the array such that elements less than the pivot come before it, and elements greater come after it.

3. **Recursively Apply:** Apply the same process to the sub-arrays formed by the partition.

Pseudocode:

```
QuickSort(arr, low, high):
    if low < high:
        pivotIndex = Partition(arr, low, high)
        QuickSort(arr, low, pivotIndex - 1)
        QuickSort(arr, pivotIndex + 1, high)

Partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j = low to high - 1:
        if arr[j] < pivot:
            i = i + 1
            swap arr[i] with arr[j]
    swap arr[i + 1] with arr[high]
    return i + 1
```

Example: Consider the array:

$$[10, 7, 8, 9, 1, 5]$$

- Choose pivot: 5
- Partition around pivot 5:
$$[1, 5, 8, 9, 7, 10]$$
- Recursively apply Quick Sort to [1] and $[8, 9, 7, 10]$
- Continue until the entire array is sorted:
$$[1, 5, 7, 8, 9, 10]$$

Visualization:

Initial Array:
$$[10, 7, 8, 9, 1, 5]$$

After Partitioning:
$$[1, 5, 8, 9, 7, 10]$$

Final Sorted Array:
$$[1, 5, 7, 8, 9, 10]$$

Advantages of Quick Sort:

- **Efficient Average Case:** Quick Sort has an average-case time complexity of $O(n \log n)$.

- **In-Place Sorting:** It requires minimal additional space.

Disadvantages of Quick Sort:

- **Worst-Case Performance:** The worst-case time complexity is $O(n^2)$, typically occurring with poor pivot choices.
- **Not Stable:** Quick Sort is not a stable sort.

3.7.3 Merge Sort

Merge Sort is a stable, comparison-based divide-and-conquer algorithm that divides the array into smaller sub-arrays, sorts them, and then merges them back together.

Algorithm:

1. **Divide:** Recursively divide the array into two halves until each sub-array contains a single element.
2. **Merge:** Merge the sorted sub-arrays to produce sorted arrays until the entire array is merged.

Pseudocode:

```
MergeSort(arr, left, right):
    if left < right:
        mid = (left + right) // 2
        MergeSort(arr, left, mid)
        MergeSort(arr, mid + 1, right)
        Merge(arr, left, mid, right)

Merge(arr, left, mid, right):
    n1 = mid - left + 1
    n2 = right - mid
    L = arr[left:left + n1]
    R = arr[mid + 1:mid + 1 + n2]
    i = 0
    j = 0
    k = left
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i = i + 1
        else:
            arr[k] = R[j]
            j = j + 1
        k = k + 1
    while i < n1:
        arr[k] = L[i]
        i = i + 1
        k = k + 1
    while j < n2:
```

```
arr[k] = R[j]
j = j + 1
k = k + 1
```

Example: Consider the array:

$$[38, 27, 43, 3, 9, 82, 10]$$

- Divide the array into:

  $$[38, 27, 43, 3]$$

  and

  $$[9, 82, 10]$$

- Recursively divide these sub-arrays until single elements are obtained.
- Merge the single elements to produce sorted arrays:

  $$[27, 38, 43, 3]$$

  and

  $$[9, 10, 82]$$

- Continue merging until the entire array is sorted:

  $$[3, 9, 10, 27, 38, 43, 82]$$

Visualization:

Initial Array:

$$[38, 27, 43, 3, 9, 82, 10]$$

After Dividing and Merging:

$$[3, 9, 10, 27, 38, 43, 82]$$

Advantages of Merge Sort:

- **Stable Sort:** Merge Sort maintains the relative order of equal elements.
- **Predictable Performance:** It has a time complexity of $O(n \log n)$ in the worst, average, and best cases.

Disadvantages of Merge Sort:

- **Space Complexity:** It requires additional space for merging.
- **Slower for Small Lists:** It may be slower compared to algorithms like Quick Sort for smaller lists.

3.7.4 Heap Sort

Heap Sort is a comparison-based sorting algorithm that utilizes a binary heap data structure. It works by building a max heap and then repeatedly extracting the maximum element to build the sorted array.

Algorithm:

1. **Build a Max Heap:** Convert the input array into a max heap where the largest element is at the root.
2. **Extract Max:** Swap the root of the heap (maximum element) with the last element of the heap and then reduce the heap size by one. Heapify the root to maintain the max heap property.
3. **Repeat:** Continue the extraction and heapify process until the heap is empty.

Pseudocode:

```
HeapSort(arr):
    n = length(arr)
    BuildMaxHeap(arr)
    for i = n - 1 down to 1:
        swap arr[0] with arr[i]
        Heapify(arr, 0, i)

BuildMaxHeap(arr):
    n = length(arr)
    for i = n // 2 - 1 down to 0:
        Heapify(arr, i, n)

Heapify(arr, i, n):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        swap arr[i] with arr[largest]
        Heapify(arr, largest, n)
```

Advantages of Heap Sort:

- **In-Place Sorting:** Heap Sort does not require additional space beyond the input array.
- **Time Complexity:** It has a time complexity of $O(n \log n)$ for both average and worst cases.

Disadvantages of Heap Sort:

- **Not Stable:** Heap Sort is not a stable sort, meaning equal elements may not retain their original order.
- **Performance:** It can be slower compared to algorithms like Quick Sort due to the overhead of heap operations.

4 Comparison of Sorting Algorithms

Comparison Table:

Algorithm	Time Complexity (Best)	Time Complexity (Worst)	Space Complexity
Shell Sort	$O(n \log n)$	$O(n^2)$	$O(1)$
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(1)$

5 Sorting in Linear Time

5.1 Introduction to Linear Time Sorting

Linear time sorting algorithms such as Counting Sort, Radix Sort, and Bucket Sort are designed to sort data in linear time $O(n)$.

Example:

- Counting Sort: Efficient for small range of integers.

5.1.1 Bucket Sort

Bucket Sort is a distribution-based sorting algorithm that divides the input into several buckets and then sorts each bucket individually. It is particularly useful when the input is uniformly distributed over a range.

Algorithm:

1. **Create Buckets:** Create an empty bucket for each possible range.
2. **Distribute Elements:** Place each element into the appropriate bucket based on its value.
3. **Sort Buckets:** Sort each bucket individually using another sorting algorithm (e.g., Insertion Sort).
4. **Concatenate Buckets:** Combine the sorted buckets into a single sorted array.

Pseudocode:

```
BucketSort(arr):
    minValue = min(arr)
    maxValue = max(arr)
    bucketCount = number of buckets
    buckets = [[] for _ in range(bucketCount)]
```

```
    for num in arr:
        index = (num - minValue) // bucketWidth
        buckets[index].append(num)
    sortedArray = []
    for bucket in buckets:
        InsertionSort(bucket)
        sortedArray.extend(bucket)
    return sortedArray

InsertionSort(arr):
    for i from 1 to length(arr):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j = j - 1
        arr[j + 1] = key
```

Example: Consider the array:

$$[0.78, 0.17, 0.39, 0.26, 0.72]$$

- **Buckets Creation:** Create 5 buckets.
- **Distribute Elements:** Place elements into buckets.
- **Sort Buckets:** Sort each bucket using Insertion Sort.
- **Concatenate Buckets:** Combine the sorted buckets:

$$[0.17, 0.26, 0.39, 0.72, 0.78]$$

5.1.2 Stable Sort

Stable Sort maintains the relative order of equal elements. Examples include Insertion Sort and Merge Sort.

Algorithm for Stable Sort:

- **Insertion Sort:** Maintain the order of elements with equal keys by inserting each element into its correct position relative to previously sorted elements.

Pseudocode:

```
InsertionSort(arr):
    for i from 1 to length(arr):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j = j - 1
        arr[j + 1] = key
```

Example: Consider the array:

$$[4, 3, 2, 1]$$

- Sort the array:

$$[1, 2, 3, 4]$$

5.1.3 Radix Sort

Radix Sort is a non-comparative integer sorting algorithm that processes digits of numbers. It works by sorting numbers digit by digit, starting from the least significant digit to the most significant digit.

Algorithm:

1. **Determine Maximum Digits:** Find the maximum number of digits in the array.
2. **Sort by Digit:** Sort the array by each digit using a stable sort (e.g., Counting Sort).
3. **Repeat:** Continue until all digits are processed.

Pseudocode:

```
RadixSort(arr):
    maxValue = max(arr)
    exp = 1
    while maxValue // exp > 0:
        CountingSort(arr, exp)
        exp = exp * 10

CountingSort(arr, exp):
    n = length(arr)
    output = [0] * n
    count = [0] * 10
    for i in range(n):
        index = (arr[i] // exp) % 10
        count[index] += 1
    for i in range(1, 10):
        count[i] += count[i - 1]
    for i in range(n - 1, -1, -1):
        index = (arr[i] // exp) % 10
        output[count[index] - 1] = arr[i]
        count[index] -= 1
    for i in range(n):
        arr[i] = output[i]
```

Example: Consider the array:

$$[170, 45, 75, 90, 802, 24, 2, 66]$$

- Sort by least significant digit:

$$[170, 90, 802, 2, 24, 45, 75, 66]$$

- Sort by next digit:

$$[802, 24, 45, 66, 75, 90, 170, 2]$$

- Continue until all digits are processed.

Question: Among Merge Sort, Insertion Sort, and Quick Sort, which algorithm performs the best in the worst case? Apply the best algorithm to sort the list

E, X, A, M, P, L, E

in alphabetical order.

Answer: - Merge Sort has a worst-case time complexity of $O(n \log n)$. - Insertion Sort has a worst-case time complexity of $O(n^2)$. - Quick Sort has a worst-case time complexity of $O(n^2)$, though its average-case complexity is $O(n \log n)$.

In the worst case, Merge Sort performs the best among these algorithms.

Sorted List using Merge Sort:

Step-by-Step Solution:

1. **Initial List:**
   - Given List:

     E, X, A, M, P, L, E

2. **Divide the List:**
   - Divide the list into two halves:

     E, X, A and M, P, L, E

3. **Recursive Division:**
   - For the first half E, X, A:
     - Divide further into:

       E and X, A

     - For X, A:
       * Divide into:

         X and A
   - For the second half M, P, L, E:
     - Divide further into:

       M, P and L, E

     - For M, P:
       * Divide into:

         M and P

     - For L, E:
       * Divide into:

         L and E

4. **Merge the Sorted Sublists:**

- Merge E and X, A:
  - Merge X and A to get:

    A, X
  - Merge E and A, X to get:

    A, E, X
- Merge M and P to get:

  M, P
- Merge L and E to get:

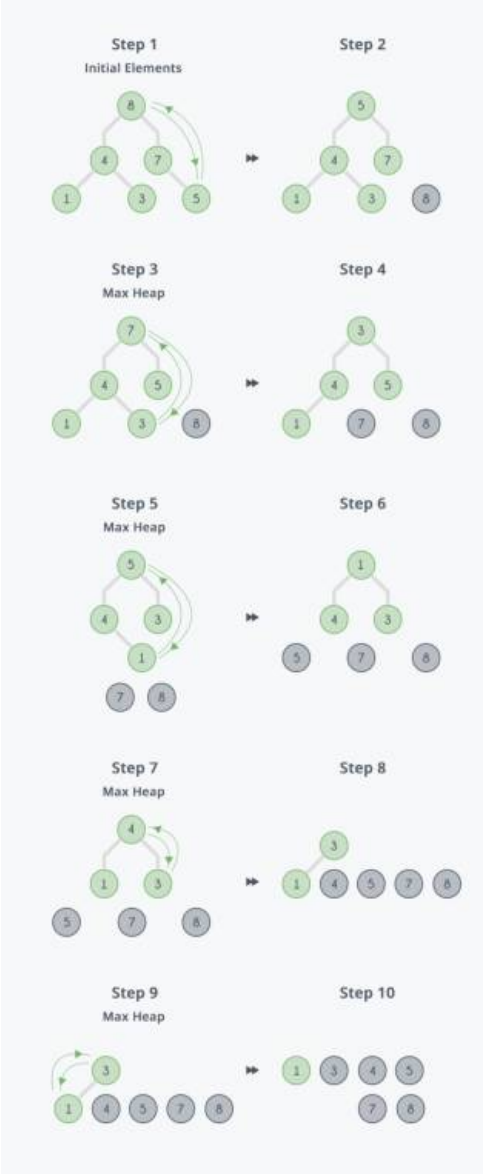
  E, L
- Merge M, P and E, L:
  - Merge M and E, L, P to get:

    E, L, M, P

5. **Merge Final Sublists:**
   - Merge A, E, X and E, L, M, P:
     - Final merge results in:

       A, E, E, L, M, P, X
6. **Sorted List:**
   - Sorted List:

     A, E, E, L, M, P, X


Step 1
Initial Elements
Step 2
Step 3
Max Heap
Step 4
Step 5
Max Heap
Step 6
Step 7
Max Heap
Step 8
Step 9
Max Heap
Step 10

ITECH WORLD AKTU

Subject Name: Design and Analysis of Algorithm (BCS503)

UNIT 2: Advanced Data Structures

Syllabus

- Red-Black Trees
- B-Trees
- Binomial Heaps
- Fibonacci Heaps
- Tries
- Skip List

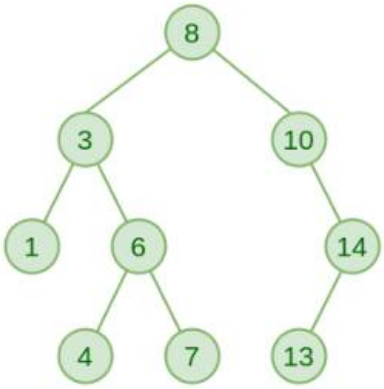
Binary Search Tree (BST)

A Binary Search Tree (BST) is a node-based binary tree data structure where each node contains the following components:

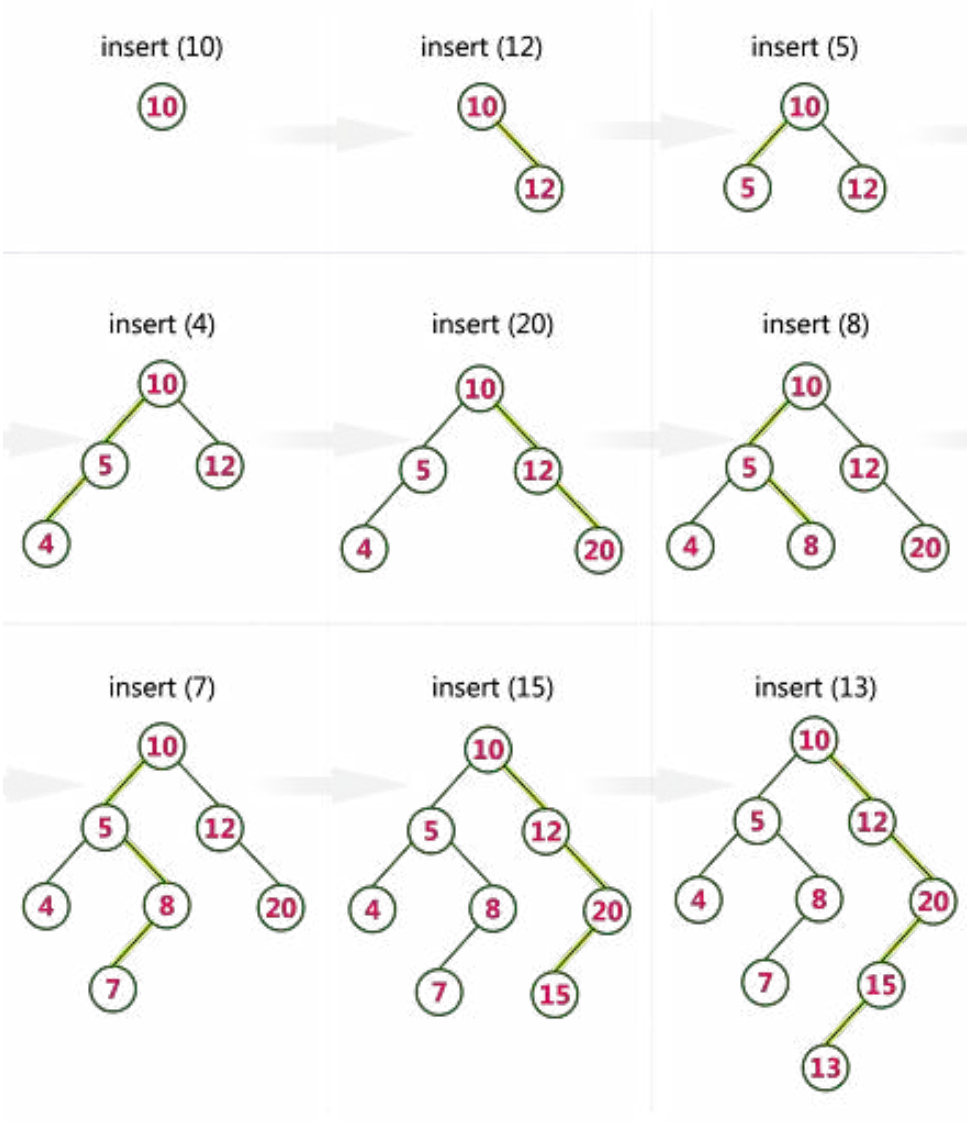
- **LEFT**: A pointer to the left child node, which contains only nodes with keys less than the current node's key.
- **KEY**: The value stored in the current node. This value determines the order within the tree.
- **PARENT**: A pointer to the parent node. The root node's parent pointer is `NULL`.
- **RIGHT**: A pointer to the right child node, which contains only nodes with keys greater than the current node's key.

Additionally, a BST must satisfy the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.



Binary Search Tree



Limitations of Binary Search Tree (BST)

Binary Search Trees (BSTs) have several limitations related to their complexity:

- **Worst-Case Time Complexity:** In the worst case, such as when the tree becomes unbalanced (e.g., inserting sorted data), the height of the BST can reach $O(n)$, resulting in search, insertion, and deletion operations having a time complexity of $O(n)$.
- **Space Complexity:** Each node requires extra memory for storing pointers to its children, which can lead to higher space complexity compared to array-based structures, especially in unbalanced trees.
- **Poor Performance with Sorted Data:** If input data is already sorted, the BST will degenerate into a linked list, causing all operations to degrade to $O(n)$ time complexity.
- **Balancing Overhead:** Self-balancing BSTs (like AVL or Red-Black Trees) require additional operations (rotations and recoloring), which add extra overhead to insertion and deletion operations.
- **Cache Inefficiency:** Due to pointer-based navigation, BSTs exhibit poor cache locality, leading to slower performance compared to structures like arrays.

Red-Black Trees

A Red-Black Tree is a type of self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black. The tree maintains its balance by following a set of rules during insertion and deletion operations.

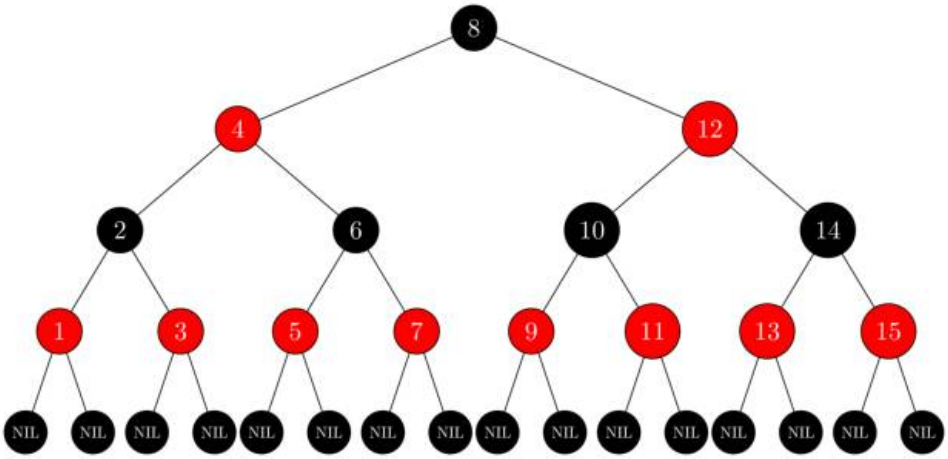
Properties of Red-Black Trees:

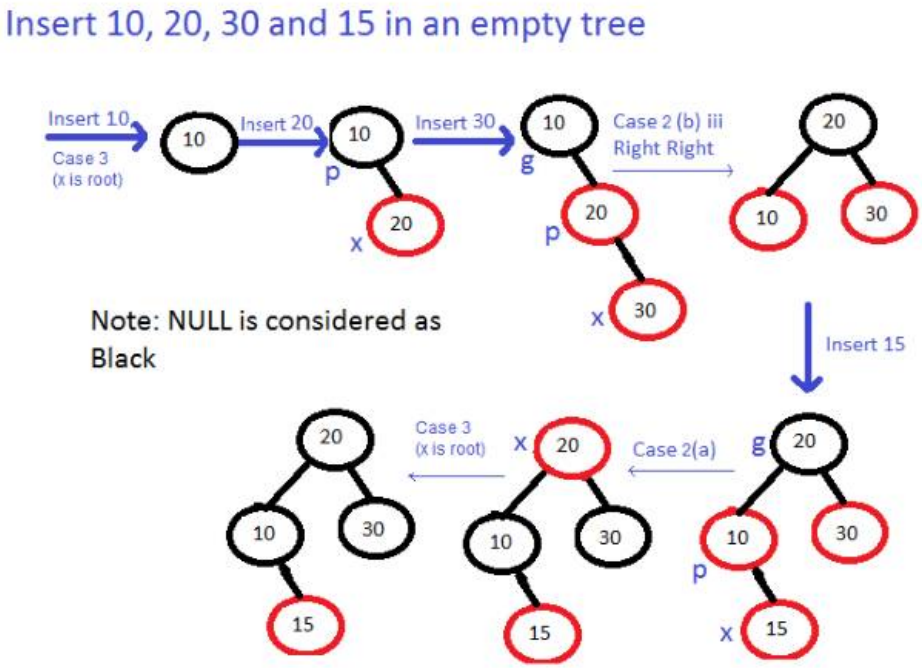
- Every node is either red or black.
- The root is always black.
- All leaves (NIL nodes) are black.
- If a node is red, then both its children are black.
- Every path from a given node to its descendant NIL nodes has the same number of black nodes.

Node Structure in Red-Black Trees:

Each node in a Red-Black Tree consists of the following components:

- **COLOUR**: Indicates the color of the node, either `red` or `black`.
- **KEY**: The value stored in the node, used to maintain the binary search tree property.
- **LEFT**: A pointer to the left child node.
- **PARENT**: A pointer to the parent node; the parent of the root node is `NIL`.
- **RIGHT**: A pointer to the right child node.





Finding the Height of a Red-Black Tree Using Black Height

In a Red-Black Tree, the **black height** of a node is defined as the number of black nodes on the path from that node to any leaf, not including the node itself. The black height is an important property that helps in maintaining the balance of the tree.

Definition of Black Height:

- The **black height** of a node x, denoted as $bh(x)$, is the number of black nodes from x to any leaf, including the leaf itself.
- The black height of a Red-Black Tree is the black height of its root node.

Calculating Black Height:

1. Start at the **root** node.
2. Initialize the black height, bh, to 0.
3. Traverse any path from the root to a leaf:
   - Whenever a black node is encountered, increment bh by 1.
   - Ignore the red nodes, as they do not contribute to the black height.
4. The value of bh when a leaf (NIL node) is reached is the black height of the tree.

Relation Between Black Height and Tree Height:

In a Red-Black Tree:

- The height of the tree, h, is at most $2 \times bh$, where bh is the black height of the tree. This is because at most every alternate node along a path from the root to a leaf can be red.
- Thus, if the black height of a Red-Black Tree is bh, the maximum height of the tree is $2 \times bh$.

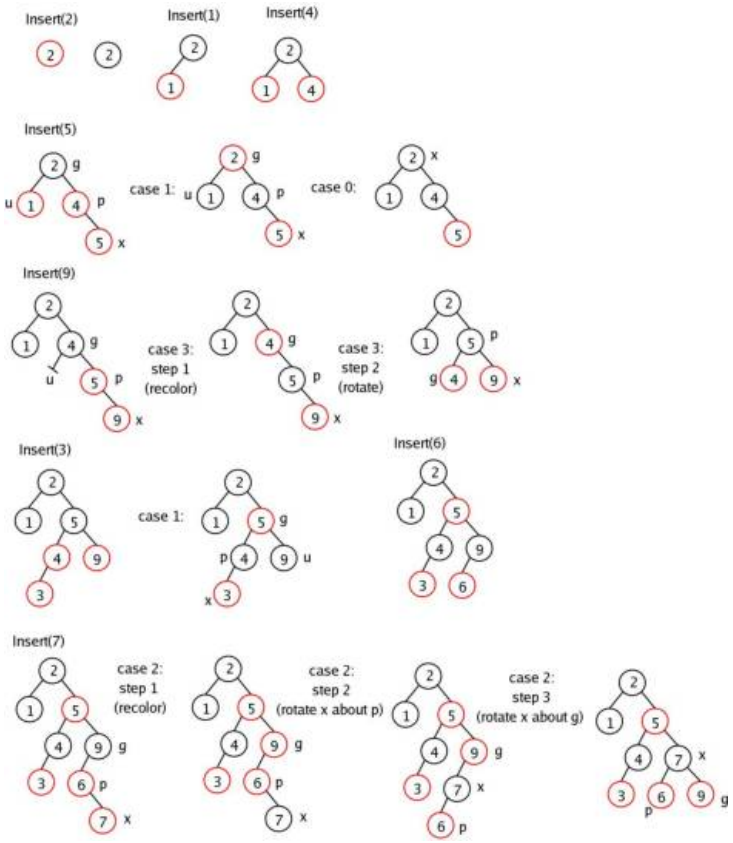
Time Complexity:

Calculating the black height of a Red-Black Tree requires traversing from the root to any leaf, resulting in a time complexity of $O(\log n)$, where n is the number of nodes in the tree.

Red-Black Tree Insert Pseudocode

```
INSERT(T, z)
 1. y = NIL
 2. x = T.root
 3. while x  NIL
 4.   y = x
 5.   if z.key < x.key
 6.     x = x.left
 7.   else
 8.     x = x.right
 9. z.p = y
 10. if y == NIL
 11.   T.root = z
 12. else if z.key < y.key
 13.   y.left = z
 14. else
 15.   y.right = z
 16. z.left = NIL
 17. z.right = NIL
```

```
18. z.color = RED
19. INSERT-FIXUP(T, z)
```



Insert Fixup Pseudocode

```
INSERT-FIXUP(T, z)
  while z.p.color == RED
    if z.p == z.p.p.left
      y = z.p.p.right
      if y.color == RED
        z.p.color = BLACK
        y.color = BLACK
        z.p.p.color = RED
        z = z.p.p
      else if z == z.p.right
        z = z.p
        LEFT-ROTATE(T, z)
      z.p.color = BLACK
      z.p.p.color = RED
      RIGHT-ROTATE(T, z.p.p)
    else
      (mirror image of the above)
  T.root.color = BLACK
```

Cases of RB Tree Insertion

- **Case 1:** Uncle is RED.
- **Case 2:** Uncle is BLACK and node is a right child.
- **Case 3:** Uncle is BLACK and node is a left child.

Advantages of Red-Black Tree over BST

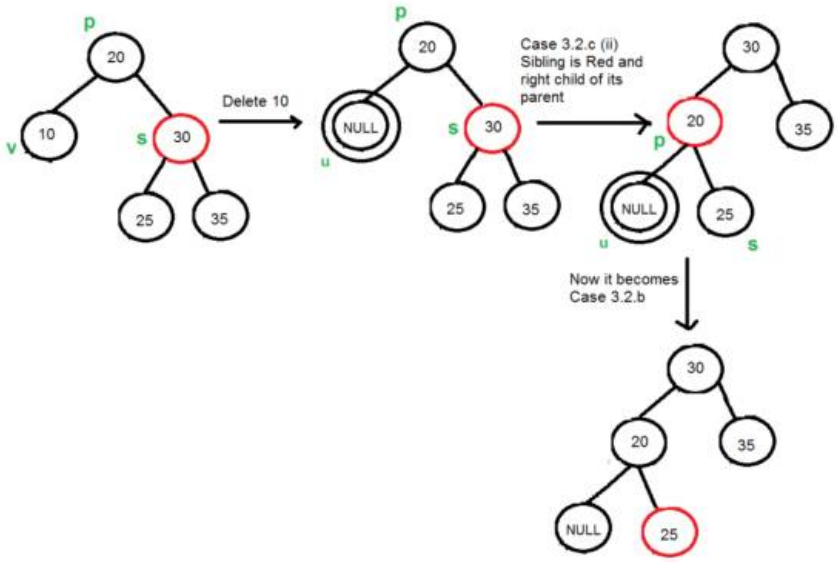
- Ensures balanced height.
- Provides better worst-case time complexity for insertion, deletion, and search operations.
- Helps maintain order in a dynamic data set.

Red-Black Tree Deletion

```
DELETE(T, z)
  1. if z.left == NIL or z.right == NIL
  2.   y = z
  3. else
  4.   y = TREE-SUCCESSOR(z)
  5. if y.left  NIL
  6.   x = y.left
  7. else
  8.   x = y.right
  9. x.p = y.p
  10. if y.p == NIL
  11.   T.root = x
  12. else if y == y.p.left
  13.   y.p.left = x
  14. else
  15.   y.p.right = x
  16. if y  z
  17.   z.key = y.key
  18. if y.color == BLACK
  19.   DELETE-FIXUP(T, x)
```

RB Tree Deletion Fixup

```
DELETE-FIXUP(T, x)
  while x  T.root and x.color == BLACK
    if x == x.p.left
      w = x.p.right
```




```
    if w.color == RED
      w.color = BLACK
      x.p.color = RED
      LEFT-ROTATE(T, x.p)
      w = x.p.right
    if w.left.color == BLACK and w.right.color == BLACK
      w.color = RED
      x = x.p
    else if w.right.color == BLACK
      w.left.color = BLACK
      w.color = RED
      RIGHT-ROTATE(T, w)
      w = x.p.right
    w.color = x.p.color
    x.p.color = BLACK
    w.right.color = BLACK
    LEFT-ROTATE(T, x.p)
    x = T.root
  else
    (mirror image of the above)
x.color = BLACK
```

Cases of RB Tree for Deletion

- **Case 1:** Sibling is RED.
- **Case 2:** Sibling is BLACK and both of sibling's children are BLACK.
- **Case 3:** Sibling is BLACK, sibling's left child is RED, and sibling's right child is BLACK.
- **Case 4:** Sibling is BLACK and sibling's right child is RED.

B-Trees

A B-Tree is a self-balancing search tree in which nodes can have more than two children. It is commonly used in databases and file systems to maintain sorted data and allow searches, sequential access, insertions, and deletions in logarithmic time.

Properties of B-Trees:

- All leaves are at the same level.
- A B-Tree of order m can have at most m children and at least $\lceil \frac{m}{2} \rceil$ children.
- Each node can contain at most $m - 1$ keys.
- Nodes are partially filled and data is sorted in increasing order.

Pseudocode for B-Tree Operations:

Searching in B-Tree: `B-Tree-Search(x, k)`

```
B-Tree-Search(x, k)
1. i = 1
2. while i  n[x] and k > keyi[x]
3.     i = i + 1
4. if i  n[x] and k == keyi[x]
5.     return (x, i)
6. if leaf[x]
7.     return NULL
8. else
9.     Disk-Read(ci[x])
10.    return B-Tree-Search(ci[x], k)
```

Insertion in B-Tree: `B-Tree-Insert(T, k)`

```
B-Tree-Insert(T, k)
1. r = root[T]
2. if n[r] == 2t - 1
3.     s = Allocate-Node()
4.     root[T] = s
5.     leaf[s] = FALSE
6.     n[s] = 0
7.     c1[s] = r
8.     B-Tree-Split-Child(s, 1, r)
9.     B-Tree-Insert-Nonfull(s, k)
10. else
11.    B-Tree-Insert-Nonfull(r, k)
```

Splitting a Child in B-Tree: `B-Tree-Split-Child(x, i, y)`

```
B-Tree-Split-Child(x, i, y)
1. z = Allocate-Node()
2. leaf[z] = leaf[y]
3. n[z] = t - 1
4. for j = 1 to t - 1
5.     keyj[z] = key(j + t)[y]
6. if not leaf[y]
7.     for j = 1 to t
8.         cj[z] = c(j + t)[y]
9. n[y] = t - 1
10. for j = n[x] + 1 downto i + 1
11.     c(j + 1)[x] = cj[x]
12. c(i + 1)[x] = z
13. for j = n[x] downto i
14.     key(j + 1)[x] = keyj[x]
15. keyi[x] = keyt[y]
16. n[x] = n[x] + 1
```

Insertion in Non-Full Node: `B-Tree-Insert-Nonfull(x, k)`

```
B-Tree-Insert-Nonfull(x, k)
1. i = n[x]
2. if leaf[x]
3.     while i  1 and k < keyi[x]
4.         key(i + 1)[x] = keyi[x]
5.         i = i - 1
6.     key(i + 1)[x] = k
7.     n[x] = n[x] + 1
8. else
9.     while i  1 and k < keyi[x]
10.        i = i - 1
11.    i = i + 1
12.    if n[ci[x]] == 2t - 1
13.        B-Tree-Split-Child(x, i, ci[x])
14.        if k > keyi[x]
15.            i = i + 1
16.    B-Tree-Insert-Nonfull(ci[x], k)
```

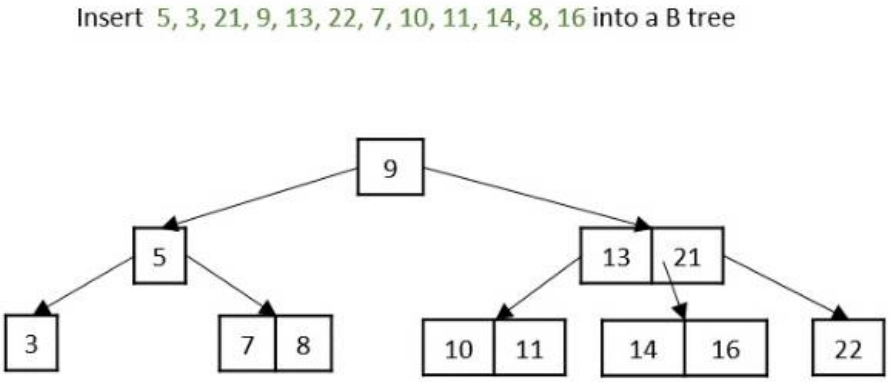
Deletion in B-Tree: `B-Tree-Delete(T, k)`

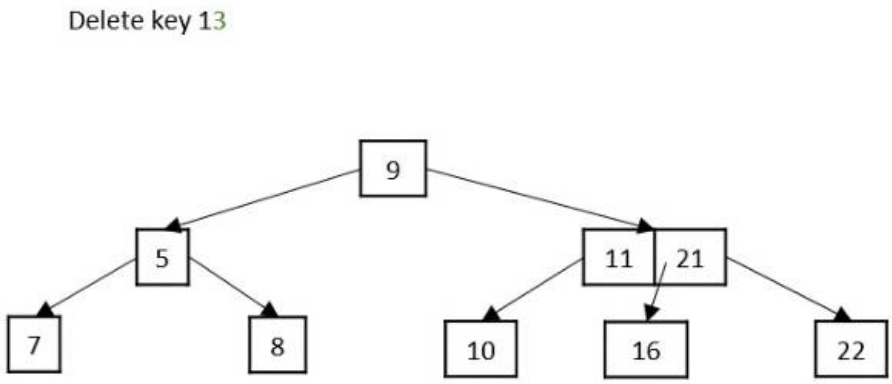
```
B-Tree-Delete(T, k)
1. Find the node x that contains the key k.
2. If x is a leaf, delete k from x.
3. If x is an internal node:
   a. If the predecessor y has at least t keys, replace k with the predecessor.
   b. If the successor z has at least t keys, replace k with the successor.
   c. If both y and z have t-1 keys, merge k, y, and z.
```

`4. Recursively delete the key from the appropriate node.`

Characteristics of B-Trees:

- B-Trees are height-balanced.
- Each node has a variable number of keys.
- Insertion and deletion are done in such a way that the tree remains balanced.
- Searching, insertion, and deletion have time complexity of $O(\log n)$.



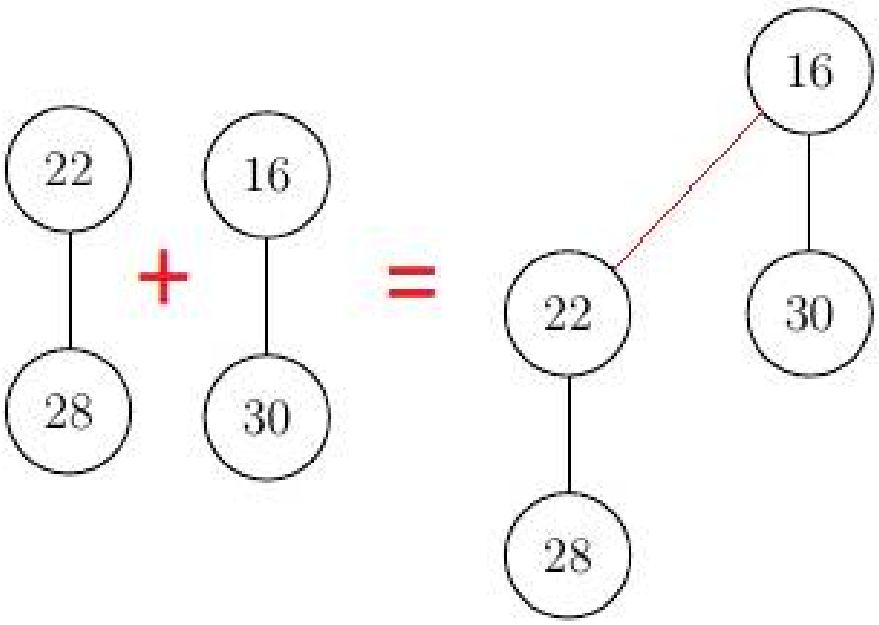


Binomial Heaps

Binomial Heaps are a type of heap data structure that supports efficient merging of two heaps. It is composed of a collection of binomial trees that satisfy the heap property.

Properties of Binomial Heaps:

- Each binomial heap is a collection of binomial trees.
- A binomial tree of order k has exactly 2^k nodes.
- The root has the smallest key and the heap is represented as a linked list of binomial trees.



1 Union of Binomial Heap

A binomial heap is a collection of binomial trees. The union of two binomial heaps involves merging two heaps into one, preserving the properties of binomial heaps.

1.1 Conditions for Union of Two Existing Binomial Heaps

The union of two binomial heaps H_1 and H_2 is performed by merging their binomial trees. The following conditions are important for the union:

1. Both heaps are merged into one by merging their respective binomial trees.
2. The resulting heap is adjusted to maintain the binomial heap properties.
3. If two trees of the same degree appear, they are merged into one tree by linking.

1.2 Algorithm for Union of Two Binomial Heaps

BINOMIAL-HEAP-UNION(H1, H2)

1. $H \leftarrow$ MERGE(H1, H2)
2. **If** $H = NULL$ **then return** NULL

3. Initialize pointers: x, $prev\_x$, and $next\_x$

4. $x \leftarrow$ head of H

5. **While** $next\_x \neq$ NULL:

6. **If** Degree(x) $\neq$ Degree($next\_x$) **or** Degree($next\_x$) == Degree($next\_next\_x$)

7. Move to the next tree

8. **Else if** Key(x) $\leq$ Key($next\_x$)

9. Link $next\_x$ as a child of x

10. $next\_x \leftarrow next\_next\_x$

11. **Else**

12. Link x as a child of $next\_x$

13. $x \leftarrow next\_x$

14. **Return** H

1.3 Time Complexity

The time complexity of binomial heap union is $O(\log n)$, where n is the total number of elements in the two heaps.

2 Binomial Heap Merge Algorithm

The merge algorithm combines two binomial heaps into one by merging their binomial trees of the same degree, similar to the binary addition process. BINOMIAL-HEAP-MERGE(H1, H2)

1. Create a new binomial heap H

2. Set $H.head$ to the root of the merged list of $H1$ and $H2$

3. **Return** H

2.1 Four Cases for Union

1. Both trees have different degrees: No merging is required.

2. Both trees have the same degree: The two trees are merged.

3. Three trees of the same degree appear consecutively: The middle tree is merged with one of its neighbors.

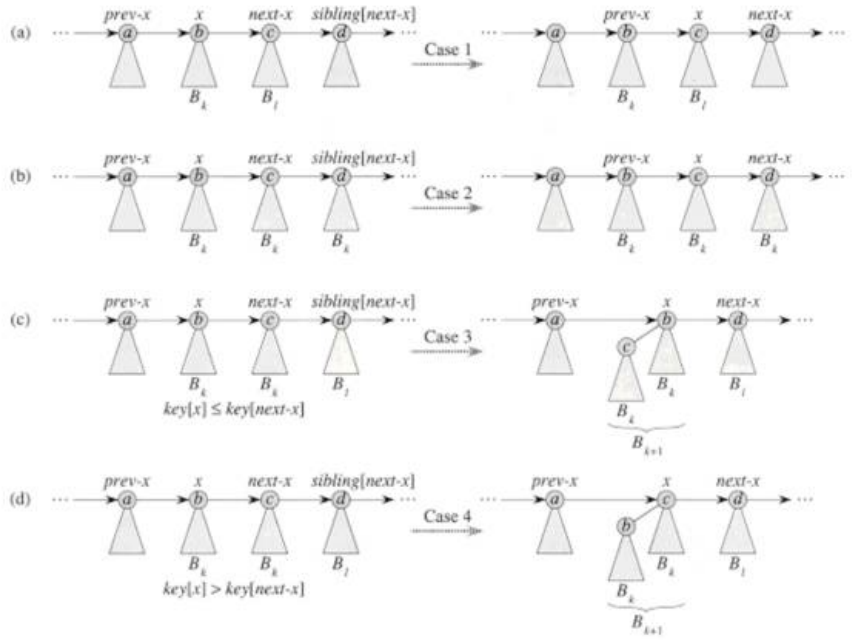
4. Two consecutive trees have the same degree: The tree with the smaller root is made the parent.

BINOMIAL-HEAP-EXTRACT-MIN(H)

1. Find the root x with the minimum key in the root list of H

2. Remove x from the root list of H

3. Create a new binomial heap H'



4. Make the children of x a separate binomial heap by reversing the order of the linked list of x's children

5. Union H and H'

6. **Return** x

3 Deleting a Given Element in a Binomial Heap

Deleting a specific element in a binomial heap is done by reducing its key to $-\infty$ and then extracting the minimum element. BINOMIAL-HEAP-DELETE(H, x)

1. Call BINOMIAL-HEAP-DECREASE-KEY(H, x, $-\infty$)

2. Call BINOMIAL-HEAP-EXTRACT-MIN(H)

4 BINOMIAL-HEAP-DECREASE-KEY Algorithm

The decrease-key operation reduces the key of a given node to a smaller value and then adjusts the heap to maintain the binomial heap property.

BINOMIAL-HEAP-DECREASE-KEY(H, x, k)

1. **If** $k > x.key$ **then** Error: New key is larger than current key

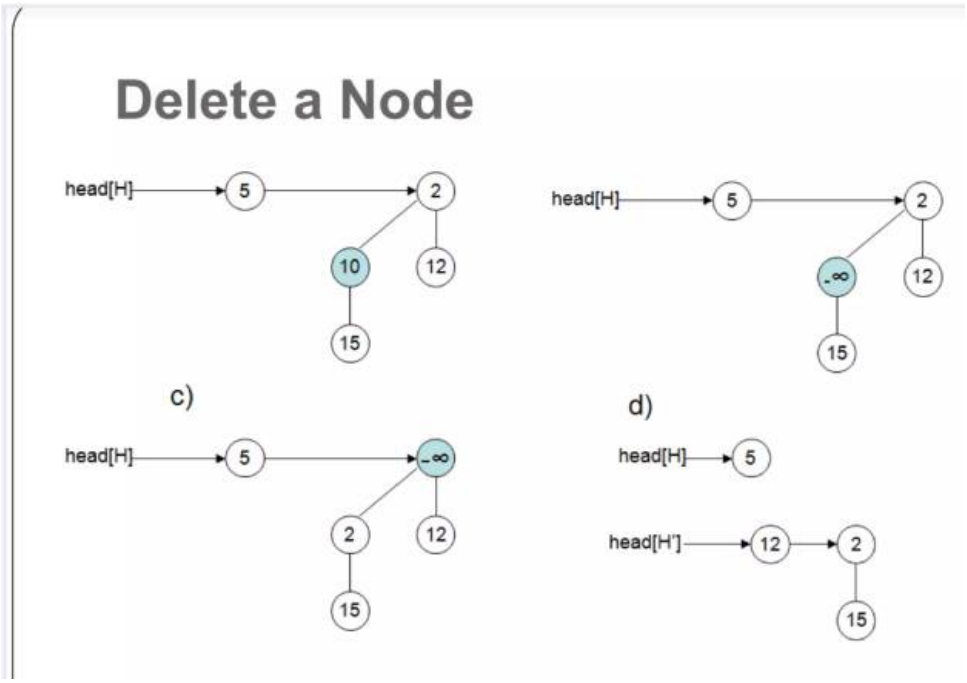
2. Set $x.key \leftarrow k$

3. Set $y \leftarrow x$, $z \leftarrow y.p$

4. **While** $z \neq$ NIL **and** $y.key < z.key$:

5. Exchange y and z

6. Set $y \leftarrow z$, $z \leftarrow y.p$



4.1 Time Complexity

The time complexity for BINOMIAL-HEAP-DECREASE-KEY is $O(\log n)$.

5 Fibonacci Heaps and Its Applications

5.1 Structure of Fibonacci Heap

- **Node:** A node in a Fibonacci heap contains a key, pointers to its parent, children, and a sibling. It also keeps track of the degree (number of children) and a mark indicating whether it has lost a child since it was made a child.
- **Heap:** A Fibonacci heap consists of a collection of heap-ordered trees. The trees are rooted, and the heap maintains a pointer to the minimum node in the heap.

5.2 Algorithm for Consolidate Operation

FIB-HEAP-CONSOLIDATE(H)

1. Let H be a Fibonacci heap
2. Initialize an empty array A of size $\lfloor \log_2(n) \rfloor + 1$
3. For each node w in the root list of H:
4. Set $x \leftarrow w$
5. Set $d \leftarrow x.degree$
6. **While** $A[d] \neq$ NIL:
7. **If** Key(x) > Key($A[d]$):
8. Set $temp \leftarrow x$

9. Set $x \leftarrow A[d]$
10. Set $A[d] \leftarrow temp$
11. Link $A[d]$ as a child of x
12. **Else**
13. Link x as a child of $A[d]$
14. **End if**
15. Increment d by 1
16. **End while**
17. Set $H.min$ to the minimum of the roots of H

5.3 Algorithm for Fib-Heap-Link

FIB-HEAP-LINK(H, y, x)

1. Remove y from the root list of H
2. Make y a child of x
3. Increase the degree of x by 1
4. Set the mark of y to false

5.4 Function for Uniting Two Fibonacci Heaps

FIB-HEAP-UNION(H1, H2)

1. Create a new Fibonacci heap H
2. Set $H.min$ to the minimum of $H1.min$ and $H2.min$
3. Combine the root lists of $H1$ and $H2$ into H
4. **Return** H

5.5 Algorithm for Make Heap

MAKE-HEAP

1. Create an empty Fibonacci heap H
2. **Return** H

5.6 Algorithm for Insert

FIB-HEAP-INSERT(H, x)

1. Insert x into the root list of H
2. **If** $H.min = NULL$ **then** set $H.min \leftarrow x$
3. **Else if** Key(x) < Key($H.min$) **then** set $H.min \leftarrow x$

5.7 Algorithm for Minimum

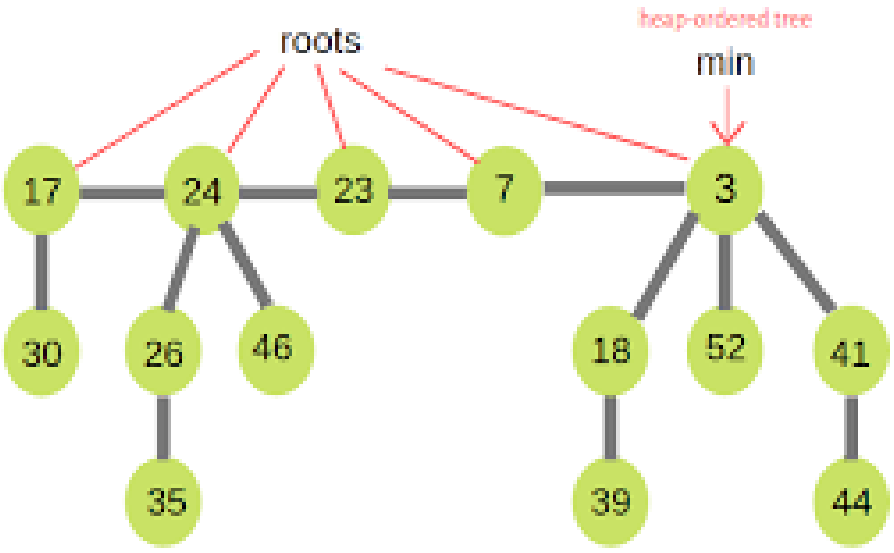
FIB-HEAP-MINIMUM(H)

1. **Return** $H.min$

5.8 Algorithm for Extract Min

FIB-HEAP-EXTRACT-MIN(H)

1. Let $z \leftarrow H.min$
2. **If** $z \neq$ NIL **then**
3. **For each** child x of z:
4. Remove x from the root list of H
5. Insert x into the root list of H
6. Remove z from the root list of H
7. **If** $H.min = z$ **then**
8. **If** H has no more nodes **then** set $H.min \leftarrow$ NIL
9. **Else** set $H.min \leftarrow$ minimum of the root list
10. Call FIB-HEAP-CONSOLIDATE(H)
11. **Return** z



Fibonacci Heaps: Structure

6 Trie and Skip List

6.1 Trie and Its Properties

Trie: A trie, also known as a prefix tree or digital tree, is a tree data structure used to store a dynamic set of strings, where the keys are usually strings. It provides a way to efficiently search, insert, and delete keys.

Properties:

- Each node represents a single character of the keys.
- The root represents an empty string.
- A path from the root to a node represents a prefix of some keys.
- Each node has a boolean flag indicating whether it marks the end of a key.
- The time complexity of search, insert, and delete operations is proportional to the length of the key.

6.2 Algorithm to Search and Insert a Key in Trie

TRIE-INSERT(TRIE, key)

1. Set *node* ← root of TRIE
2. **For each** character *c* in *key*:
3. **If** *c* is not a child of *node*:
4. Create a new node for *c*
5. **End if**
6. Set *node* ← child of *node* corresponding to *c*
7. Set *node.isEnd* ← true

TRIE-SEARCH(TRIE, key)

1. Set *node* ← root of TRIE
2. **For each** character *c* in *key*:
3. **If** *c* is not a child of *node*:
4. **Return** false
5. **Else**
6. Set *node* ← child of *node* corresponding to *c*
7. **Return** *node.isEnd*

6.3 Skip List and Its Properties

Skip List: A skip list is a data structure that allows fast search within an ordered sequence of elements. It uses multiple layers of linked lists to skip over elements, making search operations faster.

Properties:

- Skip lists have multiple levels of linked lists.
- Each element in a higher level list represents a shortcut to the lower level lists.
- The time complexity of search, insertion, and deletion operations is $O(\log n)$ on average.
- The space complexity is $O(n \log n)$.

6.4 Insertion, Searching, and Deletion Operations

INSERT(list, searchKey)

1. Set *node* ← head of list
2. **While** *node* has a next node:
3. **If** $node.next.key > searchKey$:
4. Create a new node with $searchKey$
5. Insert the new node between *node* and *node.next*
6. **Return**
7. **Else** set *node* ← node.next
8. **Insert** $searchKey$ at the end of the list

SEARCH(list, searchKey)

1. Set *node* ← head of list
2. **While** *node* has a next node:
3. **If** $node.next.key = searchKey$:
4. **Return** true
5. **Else if** $node.next.key > searchKey$:
6. **Return** false
7. Set *node* ← node.next
8. **Return** false

DELETE(list, searchKey)

1. Set *node* ← head of list
2. **While** *node* has a next node:
3. **If** $node.next.key = searchKey$:
4. Set *node.next* ← node.next.next
5. **Return**
6. Set *node* ← node.next
7. **Return** false

6.5 Divide and Conquer Approach to Compute x^n

POWER(x, n)

1. **If** $n = 0$ **then Return** 1
2. **If** n is even:
3. Set $y \leftarrow$ POWER(x, $n/2$)
4. **Return** $y \times y$
5. **Else**
6. **Return** $x\times$ POWER(x, $n - 1$)

ITECH WORLD AKTU

Design and Analysis of Algorithms (BCS503)

UNIT 3: Divide and Conquer & Greedy Methods

Syllabus

- Divide and Conquer with examples such as Sorting, Matrix Multiplication, Convex Hull, and Searching.
- Greedy Methods with examples such as Optimal Reliability Allocation, Knapsack, Minimum Spanning Trees (Prim's and Kruskal's algorithms), Single Source Shortest Paths (Dijkstra's and Bellman-Ford algorithms).

Divide and Conquer

The **Divide and Conquer** technique involves solving problems by breaking them into smaller sub-problems, solving each sub-problem independently, and then combining their results. It follows three key steps:

- **Divide**: The problem is divided into smaller sub-problems, typically into two or more parts.
- **Conquer**: The sub-problems are solved recursively. If the sub-problem is small enough, solve it directly.
- **Combine**: The solutions of the sub-problems are combined to get the final solution to the original problem.

Divide and Conquer Algorithm

Generic Divide and Conquer Algorithm

[H] Generic Divide and Conquer Algorithm [1] **DivideAndConquer**(problem):

- If the problem is small enough:
  - Solve the problem directly.
- Else:
  - Divide the problem into smaller sub-problems.
  - Recursively solve each sub-problem using **DivideAndConquer**.
  - Combine the solutions of the sub-problems to get the final solution.

Example 1: Merge Sort

Merge Sort is a sorting algorithm that follows the Divide and Conquer paradigm. The array is divided into two halves, sorted independently, and then merged.

Algorithm:

- Divide the array into two halves.
- Recursively sort each half.
- Merge the two sorted halves to get the sorted array.

Example: For an array [38, 27, 43, 3, 9, 82, 10], the array is divided and merged in steps.

Matrix Multiplication

Matrix multiplication is a fundamental operation in many areas of computer science and mathematics. There are two main methods for matrix multiplication:

1. Conventional Matrix Multiplication (Naive Method)

The conventional method of multiplying two matrices A and B follows the standard $O(n^3)$ approach. If A and B are $n \times n$ matrices, the product matrix $C = AB$ is calculated as:

$$C[i][j] = \sum_{k=1}^{n} A[i][k] \cdot B[k][j]$$

The time complexity of this method is $O(n^3)$ since each element of the resulting matrix C is computed by multiplying n pairs of elements from A and B.

2. Divide and Conquer Approach to Matrix Multiplication

In the divide and conquer approach, matrices A and B are divided into smaller sub-matrices. This method recursively multiplies the sub-matrices and combines the results to obtain the final product matrix. The key idea is to reduce the matrix multiplication problem size by breaking down large matrices into smaller parts.

3. Strassen's Matrix Multiplication

Strassen's Algorithm is an optimized version of the divide and conquer method. It reduces the time complexity of matrix multiplication from $O(n^3)$ to approximately $O(n^{2.81})$.

Key Idea: Strassen's method reduces the number of recursive multiplications by cleverly reorganizing matrix products. Instead of 8 recursive multiplications (as in the naive divide-and-conquer method), Strassen's algorithm performs 7 multiplications and 18 additions/subtractions.

Steps of Strassen's Algorithm

Suppose we wish to compute the product $C = AB$, where A, B, and C are $n \times n$ matrices. The algorithm proceeds as follows:

1. Divide: - Split each $n \times n$ matrix A and B into four sub-matrices of size $\frac{n}{2} \times \frac{n}{2}$. Let:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

2. Conquer: - Perform seven recursive multiplications:

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M_2 = (A_{21} + A_{22})B_{11}$$

$$M_3 = A_{11}(B_{12} - B_{22})$$

$$M_4 = A_{22}(B_{21} - B_{11})$$

$$M_5 = (A_{11} + A_{12})B_{22}$$

$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

3. Combine: - Combine the seven products to get the final sub-matrices of C:

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

$$C_{12} = M_3 + M_5$$

$$C_{21} = M_2 + M_4$$

$$C_{22} = M_1 - M_2 + M_3 + M_6$$

Thus, the matrix C is:

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Recurrence Relation: The time complexity of Strassen's Algorithm can be expressed by the recurrence relation:

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

Here, $T(n)$ represents the time complexity for multiplying two $n \times n$ matrices. Solving this recurrence using the master theorem gives $T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$.

Advantages and Disadvantages

Advantages: - Reduced time complexity compared to the conventional method, especially for large matrices.

Disadvantages: - The algorithm involves more additions and subtractions, which increases constant factors. - Implementation is more complex, and the recursive approach can lead to overhead for small matrices.

Example:

Let's multiply two 2×2 matrices using Strassen's method.

Let A and B be:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

Using Strassen's method, we compute the seven products $M_1, M_2, \ldots, M_7$ and then combine them to get the resulting matrix C.

The resulting product matrix $C = AB$ is:

$$C = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

This example demonstrates the power of Strassen's method in reducing the number of multiplications and solving matrix multiplication more efficiently.

Example 3: Convex Hull Problem

The Convex Hull of a set of points is the smallest convex polygon that contains all the points. The problem can be solved using various algorithms, and one of the most efficient is the Graham Scan Algorithm.

Graham Scan Algorithm

The Graham Scan algorithm is an efficient method to compute the convex hull of a set of points in the plane. The main idea is to sort the points based on their polar angle with respect to a reference point and then process them to form the convex hull.

Steps of the Algorithm:

1. Find the point with the lowest y-coordinate (in case of a tie, the leftmost point). This point is the starting point P_0.
2. Sort the remaining points based on the polar angle they make with P_0. If two points have the same polar angle, keep the one that is closer to P_0.
3. Initialize the convex hull with the first three points from the sorted list.
4. Process each of the remaining points:
   (a) While the angle formed by the last two points in the hull and the current point makes a non-left turn (i.e., the turn is clockwise or collinear), remove the second-to-last point from the hull.
   (b) Add the current point to the hull.
5. After processing all points, the points remaining in the hull list form the convex hull.

Graham Scan Algorithm in Pseudocode:

[H] Graham Scan for Convex Hull [1] **GrahamScan**(points)

Find the point P_0 with the lowest y-coordinate.

Sort the points based on the polar angle with respect to P_0.

Initialize the convex hull with the first three points from the sorted list. each remaining point p_i the turn formed by the last two points of the hull and p_i is not left

Remove the second-to-last point from the hull.

Add p_i to the hull.

Return the points in the hull.

Time Complexity:

- Sorting the points based on the polar angle takes $O(n \log n)$.
- Processing each point and constructing the convex hull takes $O(n)$.

Thus, the overall time complexity of the Graham Scan algorithm is:

$$O(n \log n)$$

where n is the number of points.

Example 4: Binary Search

Binary Search is used to find an element in a sorted array. The array is divided into two halves, and the search is performed in the half where the element may exist.

Algorithm:

- Compare the middle element with the target value.
- If equal, return the position.
- If the target is smaller, search in the left half; otherwise, search in the right half.

article amsmath

Greedy Methods

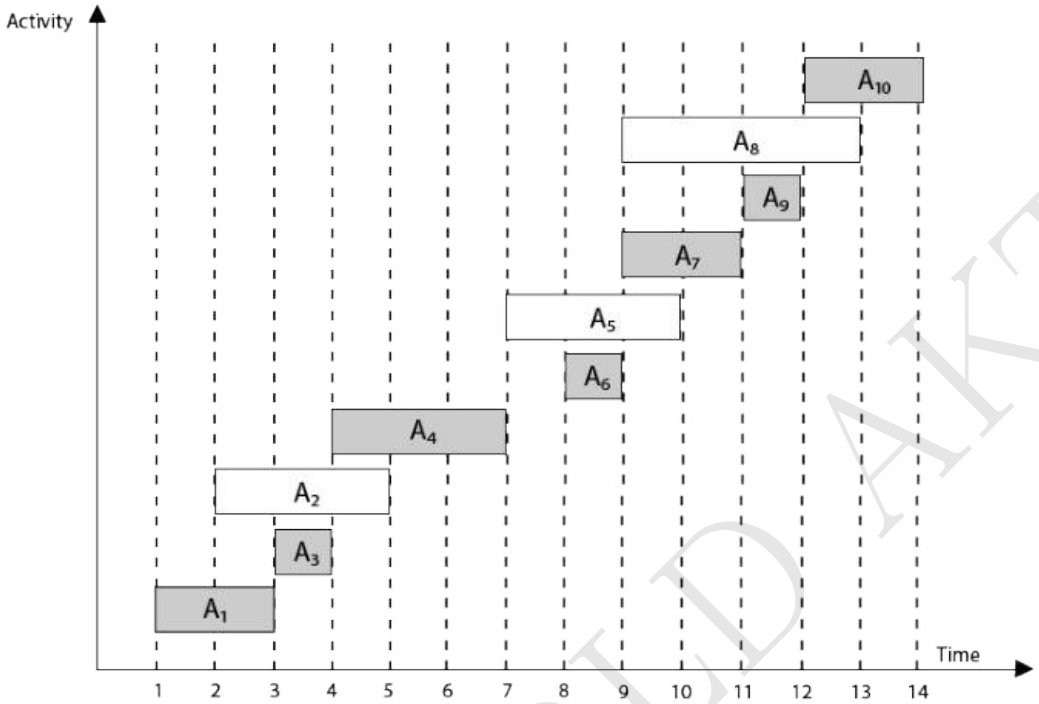
The Greedy method constructs a solution step by step by selecting the best possible option at each stage, without revisiting or considering the consequences of previous choices. It works under the assumption that by choosing a local optimum at every step, the overall solution will be globally optimal.

0.1 Key Characteristics of Greedy Algorithms

1. **Greedy Choice Property:** At every step, choose the best option available without worrying about the future implications. The choice must be feasible and should follow the rules of the problem.
2. **Optimal Substructure:** A problem has an optimal substructure if the optimal solution to the problem contains the optimal solution to its sub-problems.
3. **No Backtracking:** Unlike other methods such as dynamic programming, greedy algorithms do not reconsider the choices made previously.
4. **Efficiency:** Greedy algorithms are typically more efficient in terms of time complexity because they solve sub-problems once and only make one pass over the input data.

0.2 Activity Selection Problem

The Activity Selection Problem involves scheduling resources among several competing activities. The goal is to select the maximum number of activities that do not overlap in time.



0.2.1 Greedy Algorithm for Activity Selection

The Greedy Activity Selector algorithm selects activities based on their finish times. The idea is to always choose the next activity that finishes first and is compatible with the previously selected activities.

Pseudocode for Greedy Activity Selector (S, F):

```
GreedyActivitySelector(S, F):
    n = length(S)
    A = {1} // The first activity is always selected
    k = 1

    for m = 2 to n:
        if S[m] >= F[k]:
            A = A  {m}
            k = m

    return A
```

0.3 Example: Activity Selection Problem

Given the starting and finishing times of 11 activities:

- (2, 3)
- (8, 12)
- (12, 14)

- (3, 5)
- (0, 6)
- (1, 4)
- (6, 10)
- (5, 7)
- (3, 8)
- (5, 9)
- (8, 11)

0.3.1 Step 1: Sorting Activities

First, we sort the activities based on their finish times:

Activity	Start, Finish
1	$(2, 3)$
2	$(3, 5)$
3	$(5, 7)$
4	$(5, 9)$
5	$(6, 10)$
6	$(8, 11)$
7	$(8, 12)$
8	$(12, 14)$
9	$(0, 6)$
10	$(1, 4)$
11	$(3, 8)$

0.3.2 Step 2: Selecting Activities

Now we will select activities using the greedy approach: - Start with Activity 1 (2, 3) - The next compatible activity is Activity 2 (3, 5) - Continue this process.

The selected activities are:

- Activity 1: (2, 3)
- Activity 2: (3, 5)
- Activity 3: (5, 7)
- Activity 6: (8, 11)
- Activity 8: (12, 14)

0.3.3 Final Selected Activities

The selected activities are (2, 3), (3, 5), (5, 7), (8, 11), and (12, 14).

0.4 Pseudocode for Recursive and Iterative Approaches

Recursive Approach:

```
RecursiveActivitySelector(S, F, k, n):
    if k >= n:
        return []
    m = k + 1
    while m <= n and S[m] < F[k]:
        m = m + 1
    return [m] + RecursiveActivitySelector(S, F, m, n)
```

Iterative Approach:

```
IterativeActivitySelector(S, F):
    n = length(S)
    A = {1} // The first activity is always selected
    k = 1

    for m = 2 to n:
        if S[m] >= F[k]:
            A = A  {m}
            k = m

    return A
```

0.5 Optimization Problem

An optimization problem is a problem in which we seek to find the best solution from a set of feasible solutions. It involves maximizing or minimizing a particular objective function subject to constraints.

0.5.1 Using Greedy Method for Optimization Problems

The greedy method can be applied to solve optimization problems by:

- Breaking the problem into smaller sub-problems.
- Making the locally optimal choice at each step, hoping it leads to a globally optimal solution.
- Ensuring that the greedy choice property and optimal substructure hold for the specific problem.

Common examples of optimization problems solved using greedy algorithms include the Knapsack Problem, Minimum Spanning Tree, and Huffman Coding.

Example 2: Knapsack Problem

The Knapsack Problem involves selecting items with given weights and values to maximize the total value without exceeding the weight limit.

Greedy Approach

The greedy approach for the Knapsack Problem follows these steps:

- Sort items by their value-to-weight ratio.
- Pick items with the highest ratio until the weight limit is reached.

Branch and Bound Approach

The Branch and Bound method is another approach to solve the Knapsack Problem efficiently by exploring the solution space using an implicit tree structure:

- **Implicit Tree:** Each node represents a state of including or excluding an item.
- **Upper Bound of Node:** Calculate the maximum possible value that can be obtained from the current node to prune the tree.

Greedy Algorithm for Discrete Knapsack Problem

The greedy method can be effective for the fractional knapsack problem but not for the 0/1 knapsack problem.

0/1 Knapsack Problem

In the 0/1 Knapsack Problem, each item can either be included (1) or excluded (0) from the knapsack. The greedy method is not effective for solving the 0/1 Knapsack Problem because it may lead to suboptimal solutions.

Simple Knapsack Problem using Greedy Method

Simple Knapsack Problem using Greedy Method

Consider the following instance for the simple knapsack problem. Find the solution using the greedy method:

- $N = 8$
- $P = \{11, 21, 31, 33, 43, 53, 55, 65\}$
- $W = \{1, 11, 21, 23, 33, 43, 45, 55\}$
- $M = 110$

Solution

To solve the problem using the greedy method, we calculate the value-to-weight ratio for each item, sort them, and fill the knapsack until we reach the maximum weight M.

The total value obtained is 152.6.

Items in Knapsack

The items included in the knapsack are $I_1, I_2, I_3, I_4, I_5,$ and a fraction of I_6.

Final Answer: Maximum value = 152.6.

Item	Weight (W)	Value (P)	Remaining Load	Value Added
1	1	11	99	11
2	11	21	88	32
3	21	31	67	63
4	23	33	44	96
5	33	43	11	139
6	43	53	-32	152.6

Table 1: Knapsack Item Selection

0/1 Knapsack Problem using Dynamic Programming

Solve the following 0/1 knapsack problem using dynamic programming:

- $P = \{11, 21, 31, 33\}$
- $W = \{2, 11, 22, 15\}$
- $C = 40$
- $N = 4$

Solution

We will solve this problem using the dynamic programming approach by creating a table to keep track of the maximum value at each capacity.

Item	Weight (W)	Value (P)	Capacity (C)	Current Value	Set
0	0	0	0	0	{(0,0)}
1	2	11	2	11	{(11,2)}
2	11	21	11	21	{(21,11)}
3	22	31	22	31	{(31,22)}
4	15	33	15	33	{(33,15)}

Table 2: Dynamic Programming Table for 0/1 Knapsack

Starting with the initial set $S_0 = \{(0, 0)\}$, we add items according to their weight and value, resulting in the maximum capacity $C = 40$ being filled.

The answer is obtained by evaluating the maximum values for the given capacity and items, leading to the final output.

Final Answer: Maximum value = 4 (the total value of the selected items).

Conclusion

The greedy method provides an efficient way to solve the fractional knapsack problem, but it may not yield an optimal solution for the 0/1 knapsack problem. For 0/1 knapsack, dynamic programming or branch and bound methods are preferred.

Comparison of Kruskal's and Prim's Algorithms

Prim's Algorithm

Algorithm:

Point	Kruskal's Algorithm	Prim's Algorithm
1	Works on edges	Works on vertices
2	Greedily adds edges	Greedily adds vertices
3	Suitable for sparse graphs	Suitable for dense graphs
4	Requires sorting of edges	Uses a priority queue for edge selection
5	Can be used on disconnected graphs	Works only on connected graphs
6	Forms forests before the MST is complete	Grows a single tree
7	Easier to implement with disjoint set	Easier to visualize tree growth

Table 3: Comparison of Kruskal's and Prim's Algorithms

- Initialize the tree with an arbitrary vertex.
- Mark the vertex as included in the MST.
- While there are vertices not in the MST:
  - Select the edge with the minimum weight that connects a vertex in the MST to a vertex outside it.
  - Add the selected edge and vertex to the MST.

Example: Consider the following graph:

$$\text{Vertices: } A, B, C, D$$

$$\text{Edges: } (A, B, 1), (A, C, 4), (B, C, 2), (B, D, 5), (C, D, 3)$$

Starting from vertex A: 1. Add edge (A, B) (weight 1). 2. Add edge (B, C) (weight 2). 3. Add edge (C, D) (weight 3).

The minimum spanning tree consists of edges (A, B), (B, C), and (C, D) with total weight 6.

Kruskal's Algorithm

Algorithm:

- Sort all edges in non-decreasing order of their weights.
- Initialize the MST as an empty set.
- For each edge, in sorted order:
  - Check if adding the edge forms a cycle.
  - If it doesn't, add it to the MST.

Example: Using the same graph:

1. Sorted edges: $(A, B, 1), (B, C, 2), (C, D, 3), (B, D, 5), (A, C, 4)$. 2. Add edge (A, B). 3. Add edge (B, C). 4. Add edge (C, D).

The minimum spanning tree consists of edges (A, B), (B, C), and (C, D) with total weight 6.

Unique Minimum Spanning Tree

Theorem: If the weights on the edges of a connected undirected graph are distinct, then there exists a unique minimum spanning tree.

Example: Consider a graph with vertices A, B, C, D and edges: - $(A, B, 1)$ - $(A, C, 3)$ - $(B, C, 2)$ - $(B, D, 4)$

Since the weights are distinct, following either Kruskal's or Prim's algorithm will lead to the same unique MST: 1. Add (A, B) (weight 1). 2. Add (B, C) (weight 2). 3. Add (B, D) (weight 4).

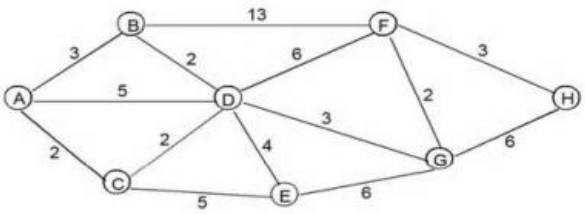
Prim's Minimum Spanning Tree Algorithm in Detail

- **Initialization:** Start with any vertex. - **Growth:** Always pick the least weight edge that expands the tree until all vertices are included. - **Termination:** When all vertices are included in the MST.

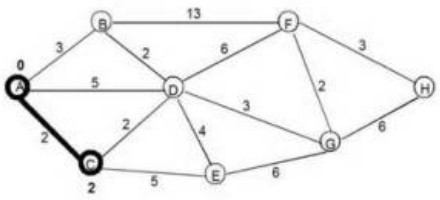
Greedy Single Source Shortest Path Algorithm (Dijkstra's Algorithm)

Algorithm:

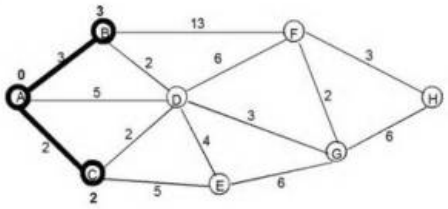
- Initialize distances from the source vertex to all others as infinity, except the source itself (0).
- Create a priority queue and insert the source vertex.
- While the queue is not empty:
  - Extract the vertex with the smallest distance.
  - For each neighbor, calculate the potential distance through the current vertex.
  - If the calculated distance is smaller, update it and add the neighbor to the queue.



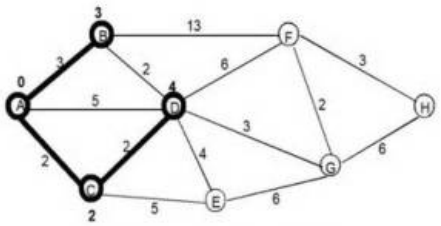
Find the shortest route from A to H



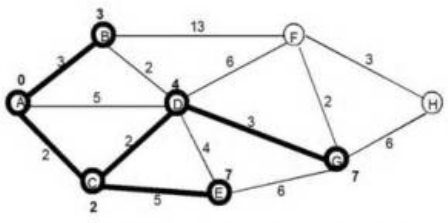
First closest node to A is C



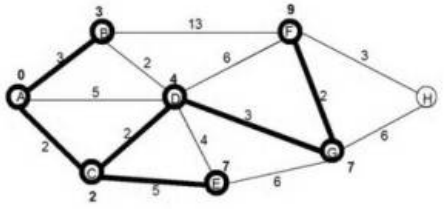
Second closest node to A is B



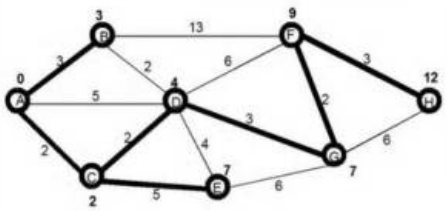
Third closest node to A is D



4th and 5th closest node to A are E and G



6th closest node to A is F



7th closest node to A is H finished

Step 1: Initialization

Node	Distance from A	Previous Node
A	0	-
B	∞	-
C	∞	-
D	∞	-
E	∞	-
F	∞	-
G	∞	-
H	∞	-

Table 4: Initial Distance Table

Step 2: Extract Minimum (A = 0)
The node closest to A is extracted first. Here, the closest node is C with a distance of 2.

- Extract C: Update the distances of its adjacent nodes.

Step 3: Relax Edges Leaving C

Node	Distance from A	Previous Node
A	0	-
B	∞	-
C	2	A
D	7	C
E	7	C
F	∞	-
G	∞	-
H	∞	-

Table 5: Distance Table After Extracting C

Step 4: Extract Minimum (B = 3)

- Extract B: Update the distances of its adjacent nodes.

Step 5: Relax Edges Leaving B

Node	Distance from A	Previous Node
A	0	-
B	3	A
C	2	A
D	7	C
E	7	C
F	9	B
G	∞	-
H	∞	-

Table 6: Distance Table After Extracting B

Step 6: Extract Minimum (D = 7)

- Extract D: Update the distances of its adjacent nodes.

Step 7: Relax Edges Leaving D

Node	Distance from A	Previous Node
A	0	-
B	3	A
C	2	A
D	7	C
E	7	C
F	9	B
G	10	D
H	∞	-

Table 7: Distance Table After Extracting D

Step 8: Continue Extracting Nodes (E, G, F, H)

Node	Distance from A	Previous Node
A	0	-
B	3	A
C	2	A
D	7	C
E	7	C
F	9	B
G	10	D
H	12	G

Table 8: Final Distance Table

Repeat the process of extracting the minimum node and updating the distances until all nodes have been visited. After completing the process, the shortest path from A to H will be found.

Final Distance Table:

Conclusion: The shortest path from A to H is:

$$A \to C \to D \to G \to H$$

with a total distance of 12.

Bellman-Ford Algorithm

Algorithm:

- Initialize the distance of the source to 0 and all others to infinity.
- For each edge, relax it $|V| - 1$ times:
  - For each edge (u, v, w):
  - If $distance[u] + w < distance[v]$, update $distance[v]$.
- Check for negative weight cycles by iterating through all edges again.

Example: Consider a graph with vertices A, B, C and edges: - $(A, B, 1)$ - $(B, C, -2)$ - $(C, A, -1)$

Starting from A: 1. Initialize distances: $A = 0$, $B = \infty$, $C = \infty$. 2. After one iteration: $B = 1$, $C = -1$. 3. After two iterations: $A = 0$, $B = 1$, $C = -1$ (no updates).

Final distances: $A = 0$, $B = 1$, $C = -1$.

article tikz amsmath booktabs graphicx

Bellman-Ford Algorithm Analysis ITECH WORLD AKTU

1 When Dijkstra and Bellman-Ford Fail to Find the Shortest Path

Dijkstra Algorithm Failure: Dijkstra's algorithm fails when there are negative weight edges in the graph. It assumes that once a node's shortest path is found, it doesn't need to be updated again. However, in the presence of negative weight edges, shorter paths may be found after visiting a node, leading to incorrect results.

Bellman-Ford Algorithm Failure: The Bellman-Ford algorithm can handle negative weight edges and will find the shortest path as long as there is no negative weight cycle.

However, it will fail to find a shortest path if the graph contains a negative weight cycle that is reachable from the source node.

2 Can Bellman-Ford Detect All Negative Weight Cycles?

Yes, Bellman-Ford can detect negative weight cycles. After completing the main relaxation process (updating distances), the algorithm performs one more iteration to check if any further relaxation is possible. If a distance is updated in this extra iteration, a negative weight cycle exists in the graph.

3 Applying Bellman-Ford Algorithm on the Given Graph

The graph provided can be analyzed using the Bellman-Ford algorithm. The algorithm iteratively updates the shortest distances from the source node to all other nodes. Here's how to apply the Bellman-Ford algorithm:

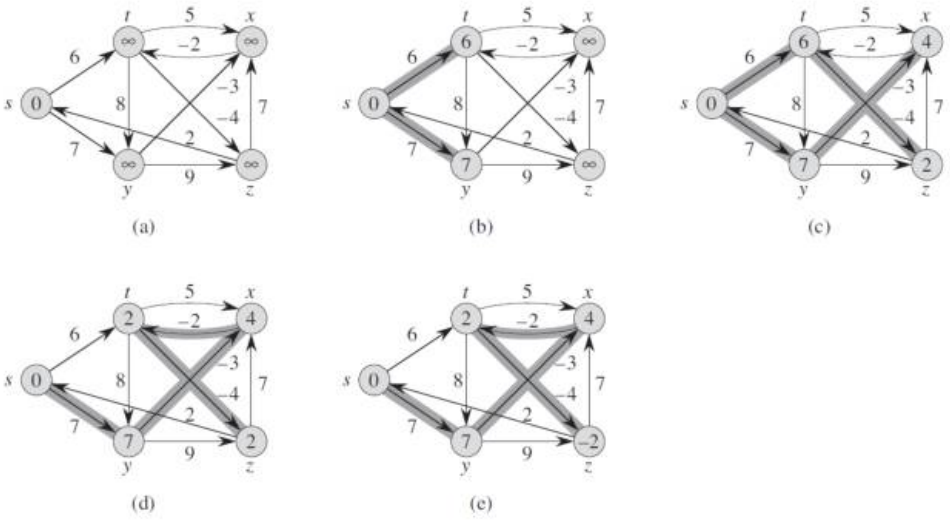


Figure 1: Graph for Bellman-Ford Application

The graph contains both positive and negative weights. We will perform multiple relaxations, updating the shortest path estimates.

4 Bellman-Ford Algorithm Table

The table below shows the shortest path distances at each iteration:

Iteration	Distance to s	Distance to t	Distance to x	Distance to y	Distance to z
0	0	∞	∞	∞	∞
1	0	6	∞	7	9
2	0	6	4	7	2
3	0	2	4	7	2
4	0	2	4	7	-2

Table 9: Distance Table for Bellman-Ford Algorithm

The algorithm terminates after the last iteration when no more updates occur.

5 Bellman-Ford Algorithm Code

Below is the Python code for applying the Bellman-Ford algorithm on this graph:

```
# Python code for Bellman-Ford Algorithm

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    def bellman_ford(self, src):
        dist = [float("Inf")] * self.V
        dist[src] = 0

        for _ in range(self.V - 1):
            for u, v, w in self.graph:
                if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w

        for u, v, w in self.graph:
            if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                print("Graph contains negative weight cycle")
                return

        print("Vertex Distance from Source")
        for i in range(self.V):
            print("{0}\t\t{1}".format(i, dist[i]))

g = Graph(5)
g.add_edge(0, 1, 6)
g.add_edge(0, 2, 7)
g.add_edge(1, 2, 8)
g.add_edge(1, 3, 5)
g.add_edge(1, 4, -4)
g.add_edge(2, 3, -3)
g.add_edge(2, 4, 9)
g.add_edge(3, 1, -2)
g.add_edge(4, 0, 2)
g.add_edge(4, 3, 7)

g.bellman_ford(0)
```

ITECH WORLD AKTU

Design and Analysis of Algorithm (DAA)
Subject Code: BCS503

UNIT 4:

Syllabus

- Dynamic Programming with Examples Such as Knapsack.
- All Pair Shortest Paths – Warshal's and Floyd's Algorithms.
- Resource Allocation Problem.
- Backtracking, Branch and Bound with Examples Such as:
  - Travelling Salesman Problem.
  - Graph Coloring.
  - n-Queen Problem.
  - Hamiltonian Cycles.
  - Sum of Subsets.

Dynamic Programming

Dynamic Programming (DP) is a technique for solving complex problems by breaking them down into simpler overlapping subproblems. It applies to problems exhibiting two main properties:

- **Overlapping subproblems:** The problem can be broken into smaller, repeating subproblems that can be solved independently.
- **Optimal substructure:** The solution to a problem can be constructed from the optimal solutions of its subproblems.

Differences Between DP and Other Approaches

1. DP solves each subproblem only once and saves the results.
2. Other approaches like divide and conquer might solve the same subproblem multiple times.

Example

The Fibonacci sequence is a classic example, where each number is the sum of the two preceding ones. DP avoids redundant calculations by storing already computed values.

Principles of Optimality and Approaches

- **Bottom-up:** Solving the smallest subproblems first and combining them to solve larger problems.
- **Top-down:** Starting with the main problem, then breaking it into subproblems recursively (usually with memoization).

Elements of Dynamic Programming

Optimal Substructure:

1. The solution to a problem can be constructed from the solutions to its subproblems.
2. Example: The shortest path in a graph can be obtained by combining the shortest paths from intermediate nodes.

Overlapping Subproblems:

1. Recursive solutions revisit the same subproblems.
2. Example: Fibonacci sequence calculations.
3. DP solves each subproblem once and stores the result.
4. Memoization or tabulation can be used to store these results.

Memoization:

1. Store the result of each subproblem.
2. Reuse the stored results when needed.
3. Avoids redundant calculations.
4. Top-down approach usually uses memoization.
5. Saves computation time in recursive solutions.

Algorithm for Longest Common Subsequence (LCS)

The LCS problem finds the longest subsequence common to two strings. Here's the algorithm:

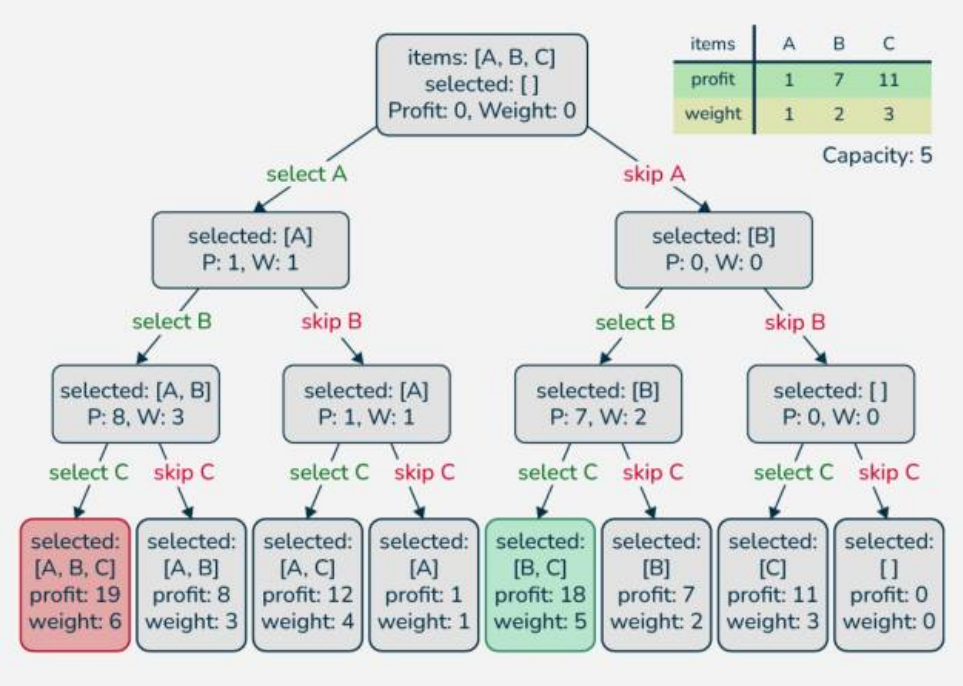
LCS-Length(x, y)

1. Let m be the length of string x and n the length of string y.
2. Create a table $dp[m+1][n+1]$ to store LCS lengths.
3. Initialize the first row and first column to 0.
4. For each $i = 1$ to m and $j = 1$ to n:
   - If $x[i] == y[j]$, set $dp[i][j] = dp[i-1][j-1] + 1$.
   - Else, set $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$.
5. Return $dp[m][n]$ as the length of the LCS.

Time Complexity: $O(mn)$, where m and n are the lengths of the two strings.

Dynamic 0/1 Knapsack Problem Algorithm

The 0/1 knapsack problem can be solved using DP by constructing a table where each entry represents the maximum value that can be obtained for a given weight limit. **Dy-**



namic 0/1-Knapsack(v, w, n, W)

1. Create a table $dp[n+1][W+1]$ to store the maximum value obtainable.
2. Initialize $dp[0][w] = 0$ for all w.
3. For each item $i = 1$ to n and weight $w = 0$ to W:

- If $w[i] \leq w$, set $dp[i][w] = \max(dp[i-1][w], dp[i-1][w-w[i]] + v[i])$.
- Else, set $dp[i][w] = dp[i-1][w]$.

4. Return $dp[n][W]$ as the maximum value.

Greedy vs Dynamic Programming

Here's a table comparing DP and Greedy:

Greedy Algorithm	**Dynamic Programming**
Makes a locally optimal choice at each step, aiming for a global optimum.	Breaks the problem into subproblems, solves them, and combines their solutions for a global optimum.
Faster as it does not consider all solutions.	Slower due to the need to store and compute multiple subproblems.
Works well when a greedy choice property holds.	Works well when overlapping subproblems and optimal substructure exist.
Example: Kruskal's algorithm for finding Minimum Spanning Tree.	Example: Finding the shortest path using the Bellman-Ford algorithm.
Does not guarantee an optimal solution in all cases.	Guarantees an optimal solution if the problem has an optimal substructure.
Uses less memory as it does not store results of previous steps.	Requires more memory to store the results of subproblems.
Suitable for simpler problems with a straightforward solution path.	Suitable for complex problems where decisions depend on the outcomes of previous steps.

Table 1: Comparison between Greedy Algorithm and Dynamic Programming

0/1 Knapsack Problem: Example

Solve the 0/1 knapsack problem with capacity = 10, profits = {1, 6, 18, 22, 28}, and weights = {1, 2, 5, 6, 7}.

Item	Profit	Weight	Profit/Weight
1	1	1	1
2	6	2	3
3	18	5	3.6
4	22	6	3.67
5	28	7	4

Maximum profit: 35, weight used: 10.

Comparison of Programming Paradigms

Here's a comparison of Divide and Conquer, Dynamic Programming, and Greedy approaches:

Aspect	**Divide and Conquer**	**Dynamic Programming**	**Greedy Approach**
Subproblem Overlap	No overlap	Solves overlapping subproblems	No overlap
Subproblem Independence	Subproblems are independent	Subproblems are dependent	Subproblems are independent
Optimal Solution	Not guaranteed	Guaranteed	May or may not be optimal
Memory Usage	Lower	Higher due to storage	Typically lower
Time Complexity	Often logarithmic	Often polynomial	Usually faster, linear
Applications	Merge Sort, Quick Sort	Knapsack, LCS, Matrix Chain Multiplication	Activity Selection, Huffman Coding
Approach Type	Recursive	Iterative or recursive	Iterative, decision-making

Table 2: Comparison between Divide and Conquer, Dynamic Programming, and Greedy Approach

Knapsack Problem

The Knapsack Problem is a classic example of dynamic programming, where the goal is to maximize the value that can be put into a knapsack of limited capacity.

Problem Statement: Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value.

Formula:

$$K(i, w) = \max(K(i-1, w), v_i + K(i-1, w-w_i)) \quad \text{if } w_i \leq w$$

where $K(i, w)$ is the maximum value that can be obtained with i items and capacity w, v_i is the value of the i^{th} item, and w_i is the weight.

Example: Consider $n = 3$ items with the following values and weights:

Item	Value	Weight
1	60	10
2	100	20
3	120	30

Knapsack capacity $W = 50$. The solution using dynamic programming will give us a maximum value of 220.

All-Pair Shortest Path Algorithms

These algorithms find the shortest paths between every pair of nodes in a graph, which is useful in various applications such as routing, network analysis, and optimization problems.

Warshall's Algorithm

Warshall's algorithm is used to compute the transitive closure of a directed graph. The transitive closure tells us whether there is a path between any two vertices in a graph.

Steps of Warshall's Algorithm: 1. Initialize the transitive closure matrix using the adjacency matrix of the graph. 2. For each intermediate vertex k, update the matrix to indicate whether a path exists between two vertices i and j via k. 3. If there is a direct edge from i to j, or a path through k, update the matrix entry to reflect the path. 4. Repeat this process for all vertices as intermediates to compute the final transitive closure.

Algorithm:

$$T(i, j) = T(i, j) \vee (T(i, k) \wedge T(k, j))$$

Where: - $T(i, j)$ represents the transitive closure matrix entry for vertices i and j. - $\wedge$ represents the logical AND operator, and $\vee$ represents the logical OR operator.

Example: Consider a graph with the following adjacency matrix:

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

After applying Warshall's algorithm, the transitive closure will indicate all reachable vertices, showing which nodes are reachable from each other.

Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is used to find the shortest paths between every pair of vertices in a weighted graph. Unlike Warshall's algorithm, which focuses on transitive closure, Floyd-Warshall computes the minimum distances between all pairs of vertices, even if there are negative weights (but no negative cycles).

Steps of Floyd-Warshall Algorithm: 1. Initialize a distance matrix using the edge weights of the graph. 2. For each intermediate vertex k, update the distance between each pair of vertices i and j. 3. If the path through k offers a shorter route, update the distance matrix entry to reflect the shorter path. 4. Repeat this process for all vertices as intermediates to compute the shortest paths between all pairs.

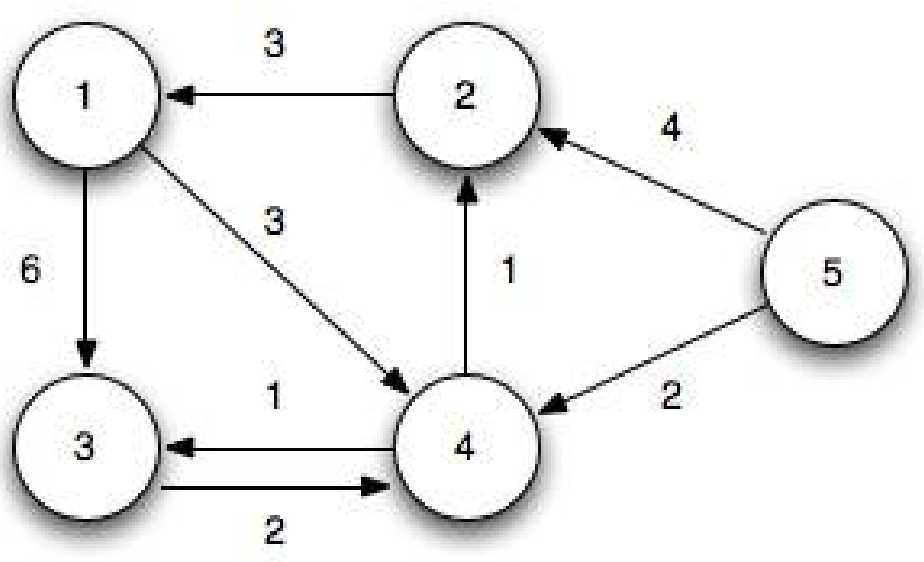
Algorithm:

$$d_{ij} = \min(d_{ij}, d_{ik} + d_{kj})$$

Where: - d_{ij} represents the shortest distance from vertex i to vertex j. - d_{ik} and d_{kj} represent the distances from i to k and k to j, respectively. - The algorithm considers all pairs of vertices and finds the shortest path for each pair using every vertex as an intermediate node.

Floyd-Warshall Algorithm Pseudocode

1. Initialize a 2D matrix D such that $D[i][j]$ is the distance between vertex i and vertex j. If no edge exists, set $D[i][j] = \infty$.
2. For each vertex k:
   (a) For each pair of vertices (i, j):
   (b) If $D[i][j] > D[i][k] + D[k][j]$, update $D[i][j] = D[i][k] + D[k][j]$.
3. Continue the process until all pairs of vertices are considered.
4. Output the distance matrix D, which now contains the shortest path between all pairs.



Floyd-Warshall Algorithm Example

Given the graph, we can represent it using an initial distance matrix. For each iteration, we update the matrix by considering an intermediate vertex.

Step 1: Initialize Distance Matrix

The initial distance matrix D_0 is as follows:

$$D_0 = \begin{pmatrix} 0 & 3 & 6 & \infty & \infty \\ \infty & 0 & 3 & 1 & 4 \\ \infty & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & 2 \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$

Iteration 1: Using Vertex 1 as Intermediate

In this step, we update the matrix using vertex 1 as an intermediate. No updates are made in this iteration.

$$D_1 = \begin{pmatrix} 0 & 3 & 6 & \infty & \infty \\ \infty & 0 & 3 & 1 & 4 \\ \infty & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & 2 \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$

Iteration 2: Using Vertex 2 as Intermediate

Using vertex 2 as an intermediate, we get the following updates: - $d_{15} = \min(\infty, 3+4) = 7$

The updated matrix is:

$$D_2 = \begin{pmatrix} 0 & 3 & 6 & 4 & 7 \\ \infty & 0 & 3 & 1 & 4 \\ \infty & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & 2 \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$

Iteration 3: Using Vertex 3 as Intermediate

Using vertex 3 as an intermediate, there are no further updates, and the matrix remains the same:

$$D_3 = \begin{pmatrix} 0 & 3 & 6 & 4 & 7 \\ \infty & 0 & 3 & 1 & 4 \\ \infty & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & 2 \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$

Iteration 4: Using Vertex 4 as Intermediate

Using vertex 4 as an intermediate, we get the following updates: - $d_{15} = \min(7, 4+2) = 6$

The updated matrix is:

$$D_4 = \begin{pmatrix} 0 & 3 & 6 & 4 & 6 \\ \infty & 0 & 3 & 1 & 3 \\ \infty & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & 2 \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$

Iteration 5: Using Vertex 5 as Intermediate

In this iteration, there are no further updates, so the matrix remains unchanged:

$$D_5 = \begin{pmatrix} 0 & 3 & 6 & 4 & 6 \\ \infty & 0 & 3 & 1 & 3 \\ \infty & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & 2 \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$

Final Result

The final distance matrix represents the shortest paths between all pairs of vertices.

$$\begin{pmatrix} 0 & 3 & 6 & 4 & 6 \\ \infty & 0 & 3 & 1 & 3 \\ \infty & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & 2 \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$

Point	Comparison
1	Warshall's algorithm is used for computing the transitive closure of a graph, whereas Floyd-Warshall is used for computing the shortest paths between all pairs of vertices.
2	Warshall's algorithm only determines if a path exists, while Floyd-Warshall finds the actual shortest path.
3	Floyd-Warshall works with weighted graphs, while Warshall's algorithm works on unweighted graphs.
4	Both algorithms use a dynamic programming approach, updating the solution incrementally.
5	Both have a time complexity of $O(n^3)$, where n is the number of vertices.
6	Warshall's algorithm uses logical operators, whereas Floyd-Warshall uses arithmetic operations.

Table 3: Comparison Between Warshall's and Floyd-Warshall Algorithms

Backtracking

Backtracking is a general algorithmic technique that incrementally builds candidates to the solutions of a problem and abandons each partial candidate (backtracks) as soon as it determines that the candidate cannot possibly be a valid solution.

It is often used in problems involving combinatorial search, where the goal is to find all or some solutions, and discarding non-promising paths as early as possible leads to efficient exploration.

General Iterative Algorithm for Backtracking

1. Start by placing the first candidate in the solution space.
2. Check if this candidate leads to a solution. If not, discard the candidate and backtrack to the previous step.
3. If it leads to a valid solution, move forward with the next step.
4. Repeat the process until all candidates are checked or the solution is found.
5. Backtrack whenever a candidate doesn't work, and try the next possible solution.
6. Continue until the solution is complete or all possible options are exhausted.

Travelling Salesman Problem (TSP)

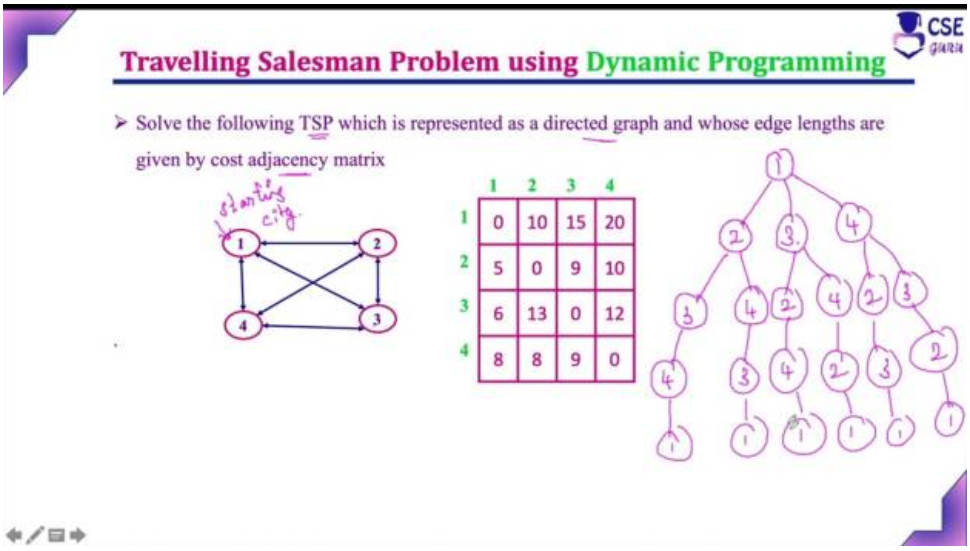
The Travelling Salesman Problem (TSP) involves finding the shortest possible route that visits each city exactly once and returns to the origin city. This problem is NP-hard, and exact solutions for large instances are difficult to compute, but branch and bound or dynamic programming techniques can be used.

A brute-force approach would generate all possible routes and compute the total distance for each, but using branch and bound allows us to discard unpromising routes based on lower bounds on the minimum distance.

Approach to Solve TSP

To solve TSP using Branch and Bound:

1. Start by selecting an initial city as the root of the tree.
2. Generate all possible routes from that city.
3. Compute a bound on the minimum possible route length (using heuristics or lower-bound calculation).
4. Eliminate any routes where the bound exceeds the known minimum distance.
5. Continue exploring promising routes by branching further, repeating the bound-checking process.
6. If a route completes with a lower total distance than the known minimum, update the minimum.
7. Repeat until all cities are explored or the minimum route is found.



Working Rule of TSP (Branch and Bound)

1. **Step 1: Initialization** Choose an initial starting city and calculate the bounds on the possible routes from that city.
2. **Step 2: Explore Branches** Generate all branches (routes) from the initial city to other cities.
3. **Step 3: Bound Calculation** For each branch, calculate the bound for the shortest possible route by using lower bounds or heuristic methods.
4. **Step 4: Pruning** Eliminate branches that exceed the current known minimum route length.
5. **Step 5: Backtrack and Explore** If necessary, backtrack to the previous step and explore other branches. Repeat the bound calculation and pruning.
6. **Step 6: Update Minimum** If a valid route with a shorter total distance is found, update the minimum. Repeat the process until no further branches are left to explore.
7. **Step 7: Output Solution** Once all possible routes have been explored, the minimum recorded route is the optimal solution.

Graph Colouring

Graph colouring is the process of assigning colours to the vertices of a graph such that no two adjacent vertices share the same colour. This problem is a type of combinatorial optimization, and the goal is to minimize the number of colours used, which is known as the chromatic number of the graph.

- It is widely used in map colouring, scheduling problems, and register allocation in compilers.
- The problem can be solved using backtracking, greedy algorithms, or other optimization techniques.

- A graph is k-colourable if it can be coloured using k colours.
- The chromatic polynomial of a graph counts the number of ways to colour the graph using at most k colours.

N-Queen Problem

The N-Queen Problem is a classic combinatorial problem where the goal is to place n queens on an $n \times n$ chessboard such that no two queens threaten each other. This means no two queens can be placed in the same row, column, or diagonal.

Pseudocode:

```
PlaceQueen(board, row, n):
    if row >= n:
        print board
        return
    for col = 0 to n-1:
        if Safe(board, row, col, n):
            board[row][col] = 1
            PlaceQueen(board, row + 1, n)
            board[row][col] = 0
```

Algorithm:

1. Start from the first row and place a queen in the first valid column.
2. Move to the next row and repeat the process.
3. If no valid position is found in a row, backtrack to the previous row and move the queen.
4. Continue this process until a solution is found or all possibilities are exhausted.

Sum of Subset Problem

The Sum of Subset Problem is a combinatorial problem where we are given a set of integers and a target sum. The goal is to determine if there exists a subset of the given set whose sum equals the target sum.

Example: Given the set $S = \{3, 34, 4, 12, 5, 2\}$ and a target sum of 9, we need to find whether there is a subset that sums up to 9.

Solution:

- We can apply backtracking or dynamic programming to solve this problem.
- For example, the subset $\{4, 5\}$ sums to 9, so the answer is yes.

Backtracking Algorithm:

1. Start with an empty subset and explore all possible subsets by either including or excluding each element.
2. If a subset sums to the target, output the subset.
3. If the sum exceeds the target, backtrack.

Dynamic Programming Approach:

- Define a boolean DP table $dp[i][j]$ where i is the index of the set and j is the sum.
- The value of $dp[i][j]$ will be true if there exists a subset with sum j using the first i elements.
- The final solution is the value of $dp[n][target]$.

Hamiltonian Circuit Problem

A Hamiltonian circuit in a graph is a cycle that visits each vertex exactly once and returns to the starting vertex. This problem is NP-complete, making it computationally difficult for large graphs.

Key Points:

1. **Definition:** A Hamiltonian circuit is a cycle that visits every vertex exactly once and returns to the starting vertex.
2. **Graph Type:** The problem can be applied to undirected or directed graphs.
3. **Existence:** Not all graphs have a Hamiltonian circuit.
4. **Solution Approach:** It can be solved using backtracking or dynamic programming.
5. **Applications:** Used in routing problems, such as in network design and travelling salesman problem.
6. **Euler vs Hamiltonian:** Unlike an Euler circuit, a Hamiltonian circuit focuses on visiting vertices rather than edges.
7. **Algorithm Complexity:** The problem is NP-complete, meaning no known polynomial-time algorithm exists for large graphs.

ITECH WORLD AKTU

Design and Analysis of Algorithms (BCS503)

Unit 5: Selected Topics

Syllabus

- Algebraic Computation
- Fast Fourier Transform (FFT)
- String Matching
- Theory of NP-Completeness
- Approximation Algorithms
- Randomized Algorithms

1. Algebraic Computation

Algebraic computation involves solving polynomial equations, symbolic computation, and manipulating algebraic structures. It is crucial in algorithm design, especially for problems in computational geometry, cryptography, and more.

FFT

Fast Fourier Transform (FFT) is an efficient algorithm for computing the Discrete Fourier Transform (DFT) and its inverse. It reduces the time complexity from $O(n^2)$ to $O(n \log n)$ by recursively breaking down a DFT of any composite size n into many smaller DFTs.

Recursive FFT Procedure

The DFT of a sequence $x = [x_0, x_1, \ldots, x_{n-1}]$ is defined as:

$$X(k) = \sum_{j=0}^{n-1} x_j \cdot e^{-2\pi ijk/n} \quad \text{for } k = 0, 1, \ldots, n-1.$$

To compute this efficiently, FFT splits the DFT into smaller sub-problems using the divide-and-conquer approach. Specifically, the DFT of an n-point sequence is divided into two $n/2$-point DFTs:

$$X(k) = \sum_{j=0}^{n/2-1} \left(x_{2j} \cdot e^{-2\pi ijk/n/2} + x_{2j+1} \cdot e^{-2\pi i(j+n/2)k/n} \right).$$

Here, x_{2j} represents the even-indexed terms, and x_{2j+1} represents the odd-indexed terms. This results in:

$$X(k) = E(k) + e^{-2\pi ik/n} O(k),$$
$$X(k + n/2) = E(k) - e^{-2\pi ik/n} O(k),$$

where $E(k)$ is the DFT of the even-indexed terms, and $O(k)$ is the DFT of the odd-indexed terms. This recursive approach continues until n becomes 1.

Recursive FFT Algorithm

The recursive algorithm for computing the FFT is as follows:

```
Algorithm FFT(x)
    Input: Array x of n complex numbers, where n is a power of 2.
    Output: Array X of n complex numbers, the DFT of x.

    if n = 1 then
        return x
    else
        X_even = FFT(x[0], x[2], ..., x[n-2])
        X_odd = FFT(x[1], x[3], ..., x[n-1])

        for k = 0 to n/2 - 1 do
```

```
        t = X_odd[k] * exp(-2 * pi * i * k / n)
        X[k] = X_even[k] + t
        X[k + n/2] = X_even[k] - t
    end for

    return X
```

This algorithm recursively breaks down the problem until the input size becomes 1, then combines the results of smaller DFTs to produce the final output.

Applications of FFT

FFT has numerous applications in different fields, including:

- **Signal Processing:** FFT is widely used to transform time-domain signals into their frequency-domain representations, enabling analysis and filtering of signals.
- **Image Processing:** In image processing, FFT is used to perform operations like image filtering, compression, and convolution.
- **Audio Compression:** FFT is essential in audio compression algorithms, such as MP3 encoding, where it helps to convert sound waves into frequency components.
- **Solving Partial Differential Equations:** FFT is utilized in numerical methods for solving partial differential equations (PDEs) like the heat equation and wave equation, where it accelerates convolution operations.

3. String Matching

String matching involves finding all occurrences of a pattern P within a given text T. It is crucial in various applications like searching within documents, DNA sequence analysis, and pattern recognition.

String Matching Problem

The string matching problem can be defined as follows:

1. Given a text T of length n and a pattern P of length m, the goal is to find all occurrences of P in T.
2. It should return the starting indices of all matches of P in T.
3. The complexity of a string matching problem depends on the length of the pattern and the text.
4. Common applications include text editors, search engines, and bioinformatics.
5. Efficient string matching algorithms help in speeding up the search process, especially for large texts.
6. Some algorithms pre-process the pattern, while others directly scan through the text.
7. Choosing the right algorithm depends on the nature of the text and pattern size.

Definitions

(a) **String:** A string is a sequence of characters, denoted as S, which can include letters, numbers, or symbols. Example: $S =$ "abcde".

(b) **Substring:** A substring of a string is any contiguous sequence of characters within the original string. Example: Substrings of $S =$ "abcde" include "abc", "bcd", and "de".

(c) **Proper Substring:** A proper substring is a substring that is not equal to the entire string. Example: For $S =$ "abcde", "abc" is a proper substring, but "abcde" is not.

(d) **Length of String:** The length of a string S is the total number of characters in it. For $S =$ "abcde", $|S| = 5$.

(e) **Prefix:** A prefix of a string is a substring that starts from the beginning of the string. Example: For $S =$ "abcde", "a" and "abc" are prefixes.

(f) **Suffix:** A suffix of a string is a substring that ends at the end of the string. Example: For $S =$ "abcde", "de" and "cde" are suffixes.

(g) **Overlapping Patterns:** String matching can involve overlapping patterns, where multiple occurrences of P overlap within T.

Types of String Matching

There are several algorithms for solving the string matching problem, including:

(a) **Naive String Matching Algorithm:** Compares the pattern with every possible position in the text.

(b) **Rabin-Karp Algorithm:** Uses hashing to find the pattern efficiently.

(c) **Knuth-Morris-Pratt (KMP) Algorithm:** Pre-processes the pattern to create a prefix function, reducing unnecessary comparisons.

Naive String Matching Algorithm

The naive approach checks for a match of P at each possible position in T.

```
Algorithm NaiveStringMatcher(T, P)
    Input: Text T of length n, Pattern P of length m
    Output: All starting indices where P occurs in T

    for s = 0 to n - m do
        if T[s:s+m] == P then
            print "Pattern occurs at index", s
```

Rabin-Karp Algorithm

This algorithm computes hash values for the pattern and substrings of the text, allowing for faster matching. However, it is susceptible to hash collisions.

Knuth-Morris-Pratt (KMP) Algorithm

The KMP algorithm pre-processes the pattern P to create a prefix function π, which helps in skipping characters while searching through the text.

Input: $T =$ "abcxabcdabcdabcy", $P =$ "abcdabcy"

The KMP algorithm matches the pattern efficiently, resulting in an optimal search time.

Computing the Prefix Function π for $P =$ "abacab" using KMP

The prefix function π stores the length of the longest proper prefix which is also a suffix for each prefix of the pattern. Here is how to compute it for $P =$ "abacab":

```
Pattern: P = "abacab"
Indices: 0  1  2  3  4  5
         a  b  a  c  a  b
Prefix Function : [0, 0, 1, 0, 1, 2]
```

Step-by-Step Calculation of π

(a) $\pi[0] = 0$ because "a" has no proper prefix.

(b) $\pi[1] = 0$ because "b" does not match with "a".

(c) $\pi[2] = 1$ because "a" matches with the prefix "a".

(d) $\pi[3] = 0$ because "c" does not match with any prefix.

(e) $\pi[4] = 1$ because "a" matches with the prefix "a".

(f) $\pi[5] = 2$ because "ab" matches with the prefix "ab".

Thus, the prefix function π for $P =$ "abacab" is $[0, 0, 1, 0, 1, 2]$, which helps in determining the next positions to continue matching.

4. Theory of NP-Completeness

NP-completeness is a class of problems for which no known polynomial-time solution exists, but a solution can be verified in polynomial time. Key concepts include NP, P, and the famous problem of $P = NP$.

Problem Classes: P, NP, and NP-Complete

- **Class P:** Problems that can be solved in polynomial time by a deterministic Turing machine. Example: Sorting algorithms like Merge Sort.
- **Class NP:** Problems for which a solution can be verified in polynomial time by a deterministic Turing machine. Example: Sudoku puzzle verification.
- **Class NP-Complete:** Problems that are both in NP and as hard as any problem in NP. If one NP-complete problem is solved in polynomial time, then every problem in NP can be solved in polynomial time. Example: Traveling Salesman Problem (TSP).

Relationship Between Classes

$$\text{P} \subseteq \text{NP} \subseteq \text{NP-Hard}$$

If $P = NP$, then all NP problems have polynomial time solutions.

Example: Traveling Salesman Problem (TSP)

Given a list of cities and distances between them, the TSP asks for the shortest possible route that visits each city exactly once and returns to the origin city. It is an NP-complete problem.

What is NP-Completeness?

- NP-Completeness refers to a category of decision problems for which any problem can be reduced to another NP problem in polynomial time.
- It is the intersection of NP and NP-Hard problem classes.
- If any NP-Complete problem has a polynomial-time solution, all NP problems would also be solvable in polynomial time.
- The concept was introduced by Stephen Cook and Richard Karp through the Cook-Levin theorem.

NP-Hard vs NP-Complete and Polynomial-Time Problems

NP-Hard: A problem is NP-Hard if every NP problem can be reduced to it in polynomial time, but it does not necessarily have to be in NP.

NP-Complete: A problem is NP-Complete if it is both in NP and NP-Hard.

Polynomial-Time Problems: These are problems that can be solved in $O(n^k)$ time, where k is a constant and n is the size of the input.

Procedure to Solve NP Problems

(a) Convert the problem into a decision problem.

(b) Use reduction techniques to transform the NP problem into a known NP-Complete problem.

(c) Apply approximation or heuristic methods to find near-optimal solutions.

(d) Validate the solution using verification algorithms.

Comparison of NP-Complete and NP-Hard

Criterion	**NP-Complete**	**NP-Hard**
Definition	Problems in NP with polynomial-time verifiable solutions.	Problems to which every NP problem can be reduced, but may not be in NP.
Verification	Solutions can be verified in polynomial time.	Solutions may not be verifiable in polynomial time.
Solution	If solved in polynomial time, all NP problems are solvable in polynomial time.	A solution does not imply a solution for all NP problems.
Examples	Traveling Salesman Problem, 3-SAT.	Halting Problem, TSP (optimization version).
Membership in NP	Must belong to NP.	May or may not belong to NP.
Reduction	Can be reduced from every problem in NP.	Every NP problem can be reduced to it, but it may not have a known polynomial-time solution.
Complexity Class	Intersection of NP and NP-Hard.	Superset of NP-Complete problems.

Table 1: Comparison of NP-Complete and NP-Hard

Minimum Vertex Cover Problem

The Minimum Vertex Cover problem is to find the smallest set of vertices such that every edge in the graph is incident to at least one vertex in this set. It is an NP-Complete problem and is used to demonstrate the hardness of other problems through reductions.

Types of NP-Complete Problems

- **Hamiltonian Cycle Problem:** Find a cycle that visits each vertex exactly once in a given graph.

- **Clique Problem:** Given a graph, find a complete subgraph of a given size.

Proof: Hamiltonian Circuit is NP-Complete

To prove that the Hamiltonian Circuit is NP-Complete:

(a) Show that the problem belongs to NP, as a given solution can be verified in polynomial time.

(b) Reduce an existing NP-Complete problem (such as 3-SAT) to the Hamiltonian Circuit problem.

(c) The reduction shows that solving the Hamiltonian Circuit also solves the 3-SAT problem, proving its NP-Completeness.

Proof: Clique Problem is NP-Complete

To prove that the Clique Problem is NP-Complete:

(a) Show that the problem is in NP by verifying a potential solution in polynomial time.

(b) Reduce the Independent Set problem (which is known to be NP-Complete) to the Clique problem.

(c) The reduction proves that solving the Clique problem also solves the Independent Set problem.

Complexity Classes and Examples

- **Randomized Polynomial Time (RP):** Problems for which a randomized algorithm can find a solution with a certain probability of correctness within polynomial time.
- **Zero Probabilistic Polynomial Time (ZPP):** Problems that have a randomized algorithm with zero error probability.
- **Probabilistic Polynomial Time (PP):** Problems where the probability of correctness is more than half.
- **Bounded Error Probabilistic Polynomial Time (BPP):** Problems that can be solved by a randomized algorithm with bounded error probability in polynomial time.

Proof: TSP is NP-Complete

Part 1: TSP in NP

- A solution to the TSP can be verified in polynomial time by checking if the total path length is below a given threshold.
- This verification process proves that TSP is in NP.

Part 2: TSP is NP-Hard

- To prove TSP is NP-Hard, reduce the Hamiltonian Cycle problem to the TSP.
- Construct a complete graph from the given graph, with a distance of 1 between connected vertices and 2 for others.
- The reduction shows that a solution to TSP would also solve the Hamiltonian Cycle problem.

Proof: 3-Coloring Problem is NP-Complete

(a) **Verification:** A given 3-coloring can be checked in polynomial time by verifying adjacent vertices.

(b) **Reduction:** Use a known NP-Complete problem, like 3-SAT, and reduce it to the 3-Coloring problem.

(c) The reduction establishes that solving the 3-Coloring problem is equivalent to solving 3-SAT.

(d) This demonstrates the NP-Completeness of the 3-Coloring problem.

5. Approximation Algorithms

Approximation algorithms provide near-optimal solutions for NP-hard problems within a guaranteed factor of the optimal solution. These algorithms are particularly useful when finding the exact solution is computationally infeasible.

Key Features of Approximation Algorithms

- **Definition:** Approximation algorithms are used for NP-hard problems, offering solutions that are not necessarily optimal but close to the best possible solution.
- **Performance Guarantee:** The performance of these algorithms is measured using an approximation ratio, which indicates how close the obtained solution is to the optimal solution.
- **Polynomial Time:** Most approximation algorithms run in polynomial time, making them practical for large inputs.
- **Trade-off:** They trade accuracy for computational efficiency.
- **Use Cases:** Commonly used in problems like TSP, Vertex Cover, and Knapsack.
- **Approach:** These algorithms often rely on heuristics or greedy strategies to achieve near-optimal solutions.
- **Guaranteed Bound:** The approximation ratio guarantees that the solution will be within a specific factor of the optimal solution.

Difference Between Deterministic and Approximation Algorithms

Criterion	Deterministic Algorithm	Approximation Algorithm
Definition	Produces an exact solution for a problem in a deterministic manner.	Produces a solution that is close to the optimal with a known approximation ratio.
Optimality	Always finds the optimal solution if it exists.	Does not necessarily find the optimal solution but guarantees proximity to it.
Time Complexity	May have high time complexity for NP-hard problems.	Offers polynomial time complexity for NP-hard problems.
Use Cases	Useful when exact solutions are necessary and feasible.	Suitable when the problem size makes exact solutions computationally expensive.
Example	Dijkstra's algorithm for shortest paths.	2-approximation algorithm for TSP.
Error Bound	No error in the solution.	Has a known approximation error, often expressed as a ratio.
Guarantee	Guarantees exact correctness.	Guarantees closeness to the optimal solution within a specific factor.

Table 2: Comparison of Deterministic and Approximation Algorithms

Example: TSP as a 2-Approximation Algorithm

The Traveling Salesman Problem (TSP) asks for the shortest route that visits each city exactly once and returns to the starting point. Finding the exact solution is NP-hard, but a 2-approximation algorithm can provide a solution that is at most twice the optimal path length.

- **Step 1: Construct a Minimum Spanning Tree (MST)**
  - Use algorithms like Prim's or Kruskal's to find the MST of the given graph.
  - The MST provides a lower bound on the cost of the optimal TSP tour.
- **Step 2: Perform Preorder Traversal**
  - Traverse the MST using a preorder walk to form a tour.
  - This step ensures that each edge is traversed twice.
- **Step 3: Shorten the Tour**
  - Skip repeated nodes in the preorder traversal to form a Hamiltonian circuit.
  - This new tour is the approximate solution.
- **Approximation Ratio:**

$$\text{Approximation Ratio: } \frac{\text{Cost of Approximate Solution}}{\text{Cost of Optimal Solution}} \leq 2$$

- **Example:** If the MST of a set of cities costs 10, the approximate TSP solution using this method would cost at most 20.

Vertex Cover Problem

The Vertex Cover Problem aims to find the smallest set of vertices such that each edge in the graph is incident to at least one vertex in this set.

Approximation Algorithm for Vertex Cover

(a) **Input:** A graph $G = (V, E)$.

(b) **Step 1:** Initialize the vertex cover set $C = \emptyset$.

(c) **Step 2:** While there are edges in E:

- Pick an edge $(u, v) \in E$.
- Add both u and v to C.
- Remove all edges incident to u or v from E.

(d) **Output:** The set C is a vertex cover with at most twice the size of the optimal solution.

Analysis of the Algorithm

- **Time Complexity:** $O(|E|)$, where $|E|$ is the number of edges in the graph.
- **Approximation Ratio:** $\leq 2 \times \text{OPT}$, where OPT is the size of the minimum vertex cover.
- **Performance Guarantee:** The solution is at most twice as large as the optimal vertex cover.

Proof: Vertex Cover Problem is 2-Approximate

- Each time an edge is selected, both of its vertices are added to the cover.
- In the worst case, the algorithm selects twice the minimum number of vertices needed.
- Since each edge must be covered, the solution is at most twice the size of the optimal solution.

What is p(n)-Approximation Algorithm?

A **p(n)-approximation algorithm** for an optimization problem guarantees that the solution will be within a factor of $p(n)$ times the optimal solution, where $p(n)$ is a polynomial function of the input size n. For example, a 2-approximation algorithm means the solution is at most twice the optimal solution.

Approximation Algorithm for TSP

(a) **Construct a Minimum Spanning Tree (MST).**

(b) **Perform a Preorder Traversal of the MST.**

(c) **Create a Tour by Shortening the Traversal:** Skip repeated vertices to form a Hamiltonian circuit.

(d) **Guarantee:** The length of the tour is at most twice the length of the optimal TSP solution.

$$\text{Approximation Ratio: } \leq 2 \times \text{Optimal Solution}$$

Example: If the MST of a set of cities costs 15, the TSP approximation would have a total cost of at most 30.

6. Randomized Algorithms

Randomized algorithms use random numbers at some point in their logic to make decisions. These algorithms are often used when deterministic approaches are inefficient or overly complex. They can provide a simpler or faster solution to certain computational problems.

Key Features of Randomized Algorithms

- **Definition:** A randomized algorithm makes use of random numbers to influence its behavior. The outcome may vary between executions, even for the same input.
- **Types of Randomized Algorithms:** Randomized algorithms can be categorized into two main types:
  - **Las Vegas Algorithms:** Always produce a correct result, but the running time varies.
  - **Monte Carlo Algorithms:** Running time is fixed, but there is a probability of error in the output.
- **Performance:** Randomized algorithms are often evaluated based on their expected running time or probability of error.
- **Use Cases:** Commonly applied in fields like cryptography, randomized search algorithms, and approximation algorithms.
- **Simplicity:** In many cases, they provide a simpler implementation than deterministic alternatives.
- **Handling Adversarial Input:** Randomization helps avoid worst-case scenarios that are crafted specifically to degrade deterministic algorithms.

Example: Randomized QuickSort

Randomized QuickSort is a variant of the classic QuickSort algorithm, where the pivot element is chosen randomly. This approach reduces the likelihood of encountering the worst-case time complexity of $O(n^2)$.

$$\text{Average Time Complexity: } O(n \log n)$$

- **Random Pivot Selection:** Selecting a pivot randomly helps to ensure balanced partitions.
- **Impact on Performance:** Reduces the probability of worst-case performance that arises with sorted or nearly sorted data.
- **Practical Efficiency:** This makes Randomized QuickSort perform efficiently in practice.

Categories of Randomized Algorithms

- **Las Vegas Algorithms:**
  - **Definition:** Las Vegas algorithms always produce a correct solution, but their running time may vary depending on the random choices made during execution.
  - **Example:** Randomized QuickSort is considered a Las Vegas algorithm because it always sorts the array correctly, but its runtime can vary.
  - **Use Case:** Particularly useful when correctness is critical, and time variation is acceptable.
- **Monte Carlo Algorithms:**
  - **Definition:** Monte Carlo algorithms have a fixed running time but allow a small probability of producing an incorrect solution.
  - **Example:** A randomized primality test can quickly determine if a number is prime, with a small probability of error.
  - **Use Case:** Useful in scenarios where a quick answer is needed and a small error margin is acceptable.

Merits of Randomized Algorithms

- **Simplicity:** Often simpler to implement than deterministic algorithms, making the code easier to understand.
- **Efficiency:** Can be faster than deterministic counterparts for certain problem instances, especially in cases where adversarial inputs exist.
- **Handling Complexity:** Capable of solving problems where deterministic solutions are either unknown or highly complex.
- **Reduced Probability of Worst-Case Scenarios:** By randomizing decisions, the chance of encountering worst-case input is minimized.
- **Versatility:** Applicable across various domains, including computational geometry, machine learning, and optimization problems.

- **Parallelizability:** Many randomized algorithms can be adapted for parallel execution, leading to further performance improvements.
- **Applications in Cryptography:** Essential in cryptographic algorithms for secure key generation and encryption schemes.

Applications of Randomized Algorithms

- **Cryptography:** Used in secure key generation, encryption, and digital signatures where random values are essential for security.
- **Machine Learning:** Randomized techniques like stochastic gradient descent (SGD) help optimize models efficiently.
- **Data Structures:** Randomized algorithms are used in data structures like randomized search trees and skip lists.
- **Optimization Problems:** Helpful in scenarios like approximation algorithms for NP-hard problems, where exact solutions are impractical.
- **Graph Algorithms:** Algorithms like randomized min-cut are used for finding cuts in graphs with high probability of correctness.
- **Network Algorithms:** Randomized routing and load balancing in distributed systems help in efficient data management.