SORTING, SELECTION AND SEQUENCING

By: Ashok Basnet

Outline

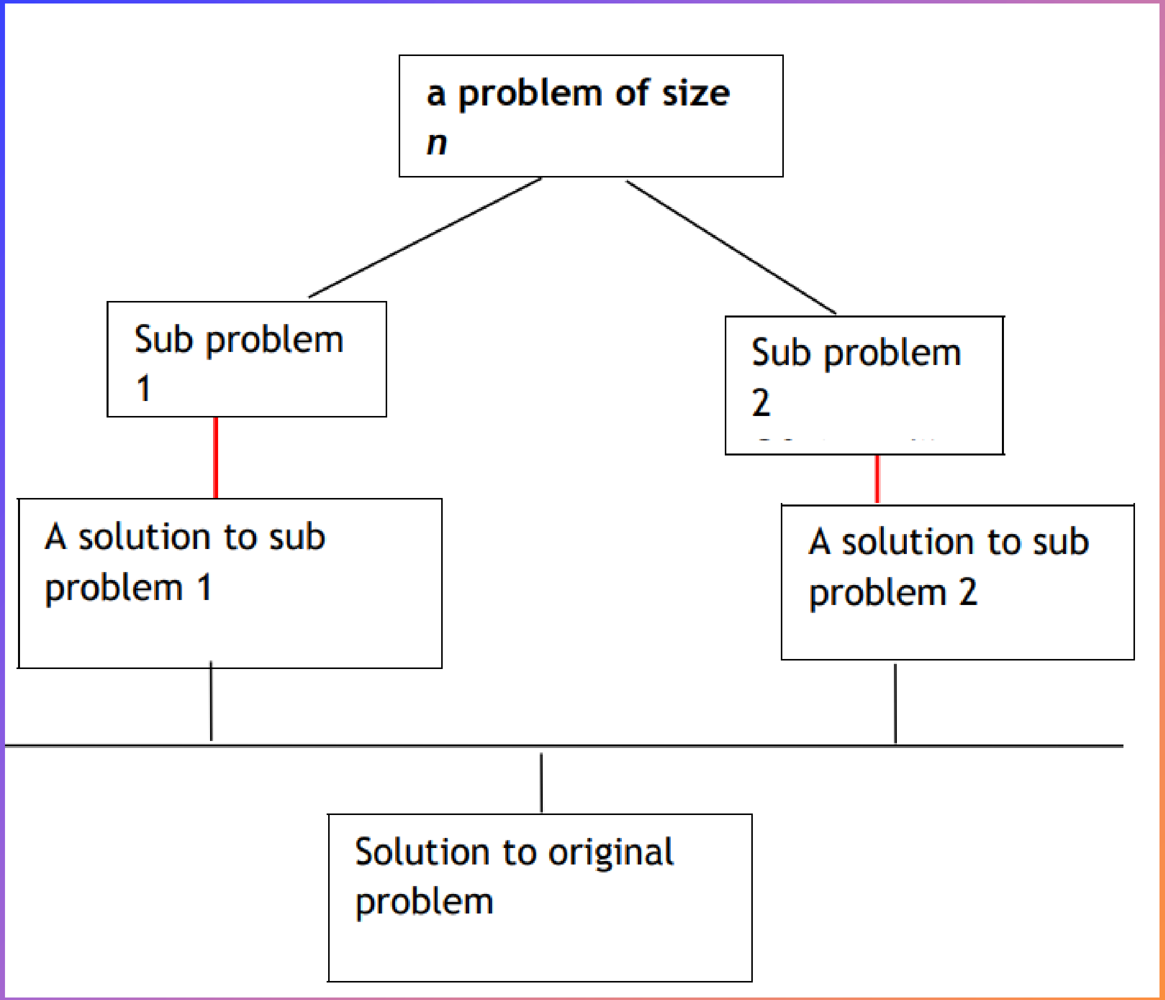
- ***General method of problem solving: Divide and Conquer Method***
  - Binary Search Technique
  - Finding the Minimum and Maximum
  - Merge Sort, Quick Sort and Selection Sort
  - Strassen's Matrix Multiplication
- ***General method of problem solving: The Greedy Method***
  - Knapsack Problem: Fractional Knapsack Problem
  - Tree Vertex Splitting
  - Job Sequencing with Deadlines
  - Minimum Cost Spanning Tree
  - Optimal Storage Allocation on Magnetic Tapes
  - Optimal Merge Patterns

Divide and Conquer Method

- ***Recursion:***
  - Divide and Conquer Method relies on recursion, where a problem is recursively broken down into smaller sub-problems.
- ***Breaking Down the Problem:***
  - The algorithm divides the problem into two or more sub-problems, often of the same or related type.
- ***Solving Sub-Problems:***
  - The sub-problems are solved independently, either directly if they are simple enough or by further breaking them down recursively.
- ***Combining Solutions:***
  - The solutions to the sub-problems are combined to produce a solution for the original problem.
- ***Efficiency:***
  - D&C often leads to efficient algorithms because it breaks down complex problems into simpler, manageable sub-problems.

Divide and Conquer Method

- ***Applications:*** D&C is widely used in solving various types of problems, including:
    - Sorting Algorithms: such as Quick Sort and Merge Sort.
    - Multiplying Large Numbers: for efficient multiplication.
    - Closest Pair of Points: finding the closest pair in a set of points.
    - Discrete Fourier Transform (FFT): a fast algorithm for computing the DFT.
- ***Example Algorithms:***
    - ***Quick Sort:*** Sorts an array by partitioning and recursively sorting sub-arrays.
    - ***Merge Sort:*** Divides the array into two halves, recursively sorts them, and then merges them.


a problem of size *n*
Sub problem 1
Sub problem 2
A solution to sub problem 1
A solution to sub problem 2
Solution to original problem

Divide and Conquer Method

- Divide the problem into a number of sub problems
- Conquer the sub problems by solving them recursively.
- If the sub problem sizes are small enough, solve the sub problems recursively, and then combine these solutions to create a solution to the original problem.

Divide and Conquer Method

```
DANDC (P)  {
        if SMALL (P) then return S (p);
        else
          {
              divide p into smaller instances p1, p2, …. Pk, k ≥ 1;
              apply DANDC to each of these sub problems;
              return (COMBINE (DANDC (p1) , DANDC (p2),…., DANDC (pk));
        }
}
```

- SMALL (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting.
- If this is so function 'S' is invoked otherwise, the problem 'p' into smaller sub problems. These sub problems p1, p2, . . . , pk are solved by recursive application of DANDC.

Binary Search (Recursive and Iterative)

- Binary search is a search algorithm used to find the position of a target value within a sorted array.
- It is a highly efficient algorithm with a time complexity of O(log n), where n is the number of elements in the array.
- The key idea behind binary search is to repeatedly divide the search interval in half.
- Steps:
  - **Initial Setup:**
    - Start with the entire sorted array.
  - **Define Search Interval:**
    - Define a search interval, initially the entire array.
  - **Compare with Midpoint:**
    - Compute the midpoint of the interval.
    - Compare the target value with the value at the midpoint.

Binary Search (Recursive and Iterative)

- Steps:
  - **Adjust Search Interval:**
    - If the target value is equal to the midpoint value, the search is successful, and the index is returned.
    - If the target value is less than the midpoint value, adjust the search interval to the lower half (discard the upper half).
    - If the target value is greater than the midpoint value, adjust the search interval to the upper half (discard the lower half).
  - **Repeat:**
    - Repeat steps 3-4 until the target value is found or the search interval becomes empty.
  - **Result:**
    - If the target value is found, return its index.
    - If the search interval becomes empty, the target value is not in the array, and -1 is typically returned.

Binary Search (Recursive)

```
function recursiveBinarySearch(arr, target, low, high):
  if low <= high:
    mid = (low + high) // 2
    if arr[mid] == target:
      return mid
    elif arr[mid] < target:
      return recursiveBinarySearch(arr, target, mid + 1, high)
    else:
      return recursiveBinarySearch(arr, target, low, mid - 1)
  else:
    return -1
```

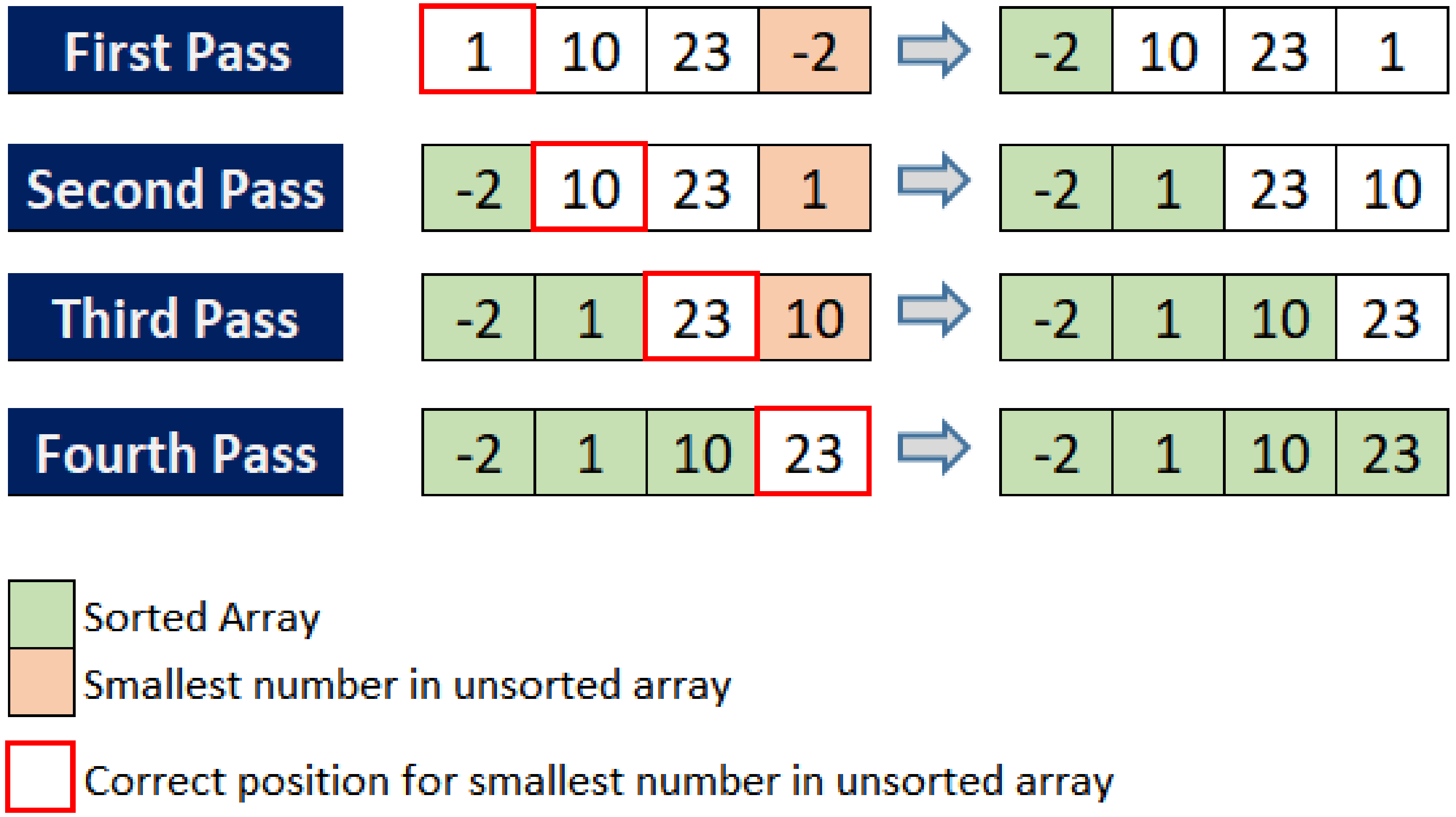
Binary Search (Iterative)

```
function iterativeBinarySearch(arr, target):
    low = 0
    high = length(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Selection sort

- Selection Sort is a simple comparison-based sorting algorithm that divides the input list into two parts: ***a sorted portion and an unsorted portion***.
- The algorithm repeatedly selects the smallest (or largest, depending on the sorting order) element from the unsorted portion and swaps it with the first unsorted element. This process continues until the entire list is sorted.
- ***Initialization:*** The entire list is considered unsorted initially.
- ***Selection of the Smallest Element:*** Iterate through the unsorted portion of the list to find the smallest element.
- ***Swap:*** Swap the smallest element found in step 2 with the first element in the unsorted portion.
- ***Expansion of the Sorted Portion:*** Expand the sorted portion to include the element that was just swapped.
- ***Repeat:*** Repeat steps 2-4 for the remaining unsorted portion of the list until the entire list is sorted.
- ***Result:*** The list is now sorted.

Selection sort



Selection sort - Iterative

```
function selectionSort(array, size):
    repeat (size - 1) times:
        set the first unsorted element as the minimum
        for each of the unsorted elements:
            if element < currentMinimum:
                set element as new minimum
        swap minimum with first unsorted position
    end repeat
end function
```

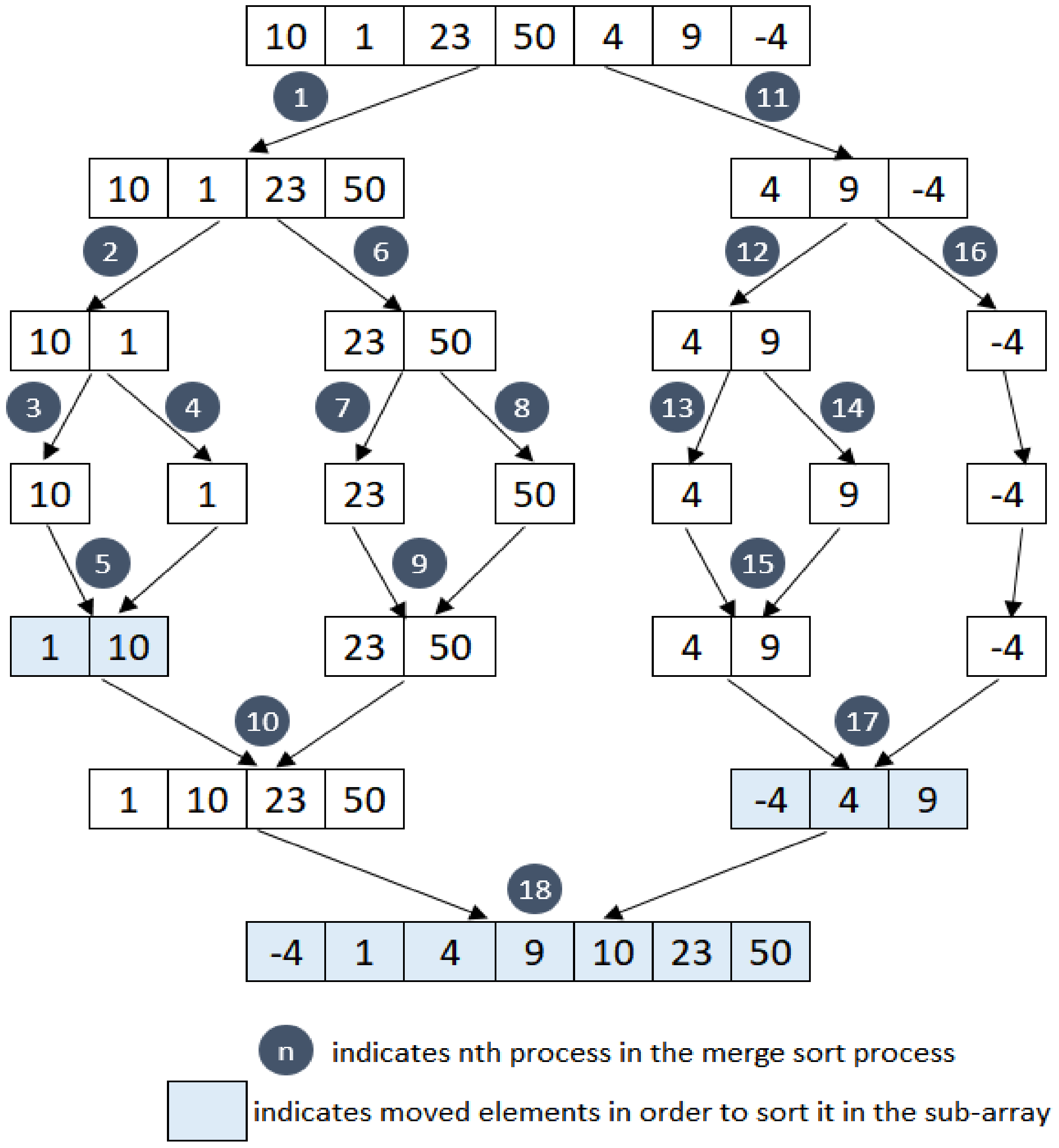
Selection sort - Recursive

```
function recursiveSelectionSort(arr, start=0):
    n = length(arr)
    if start < n - 1:
        min_index = start
        for i from start + 1 to n - 1:
            if arr[i] < arr[min_index]:
                min_index = i
        swap(arr[start], arr[min_index])
        recursiveSelectionSort(arr, start + 1)
end function
```

Merge sort

- Merge Sort is a divide-and-conquer sorting algorithm that works by dividing the unsorted list into ***n sublists***, each containing one element (making them inherently sorted), and then ***repeatedly merging sublists*** to produce new sorted sublists until there is only one sublist remaining – the sorted list.
- <u>*Here's a step-by-step overview of the Merge Sort algorithm:*</u>
  - **Divide:** Divide the unsorted list into <u>***n sublists,***</u> each containing one element. This is the base case of the recursion.
  - **Conquer:** Recursively sort each sublist.
  - **Merge:** Merge the sorted sublists to <u>***produce new sorted sublists***</u> until there is only one sublist remaining – the sorted list.
- The key operation in Merge Sort is the <u>***merging step,***</u> where two sorted sublists are combined into a single sorted sublist. This process is repeated until the entire list is sorted.

EXAMPLE


10 1 23 50 4 9 -4
1
11
10 1 23 50
4 9 -4
2
6
12
16
10 1
23 50
4 9
-4
3
4
7
8
13
14
10
1
23
50
4
9
-4
5
9
15
1 10
23 50
4 9
-4
10
17
1 10 23 50
-4 4 9
18
-4 1 4 9 10 23 50
n
indicates nth process in the merge sort process
indicates moved elements in order to sort it in the sub-array

Merge sort

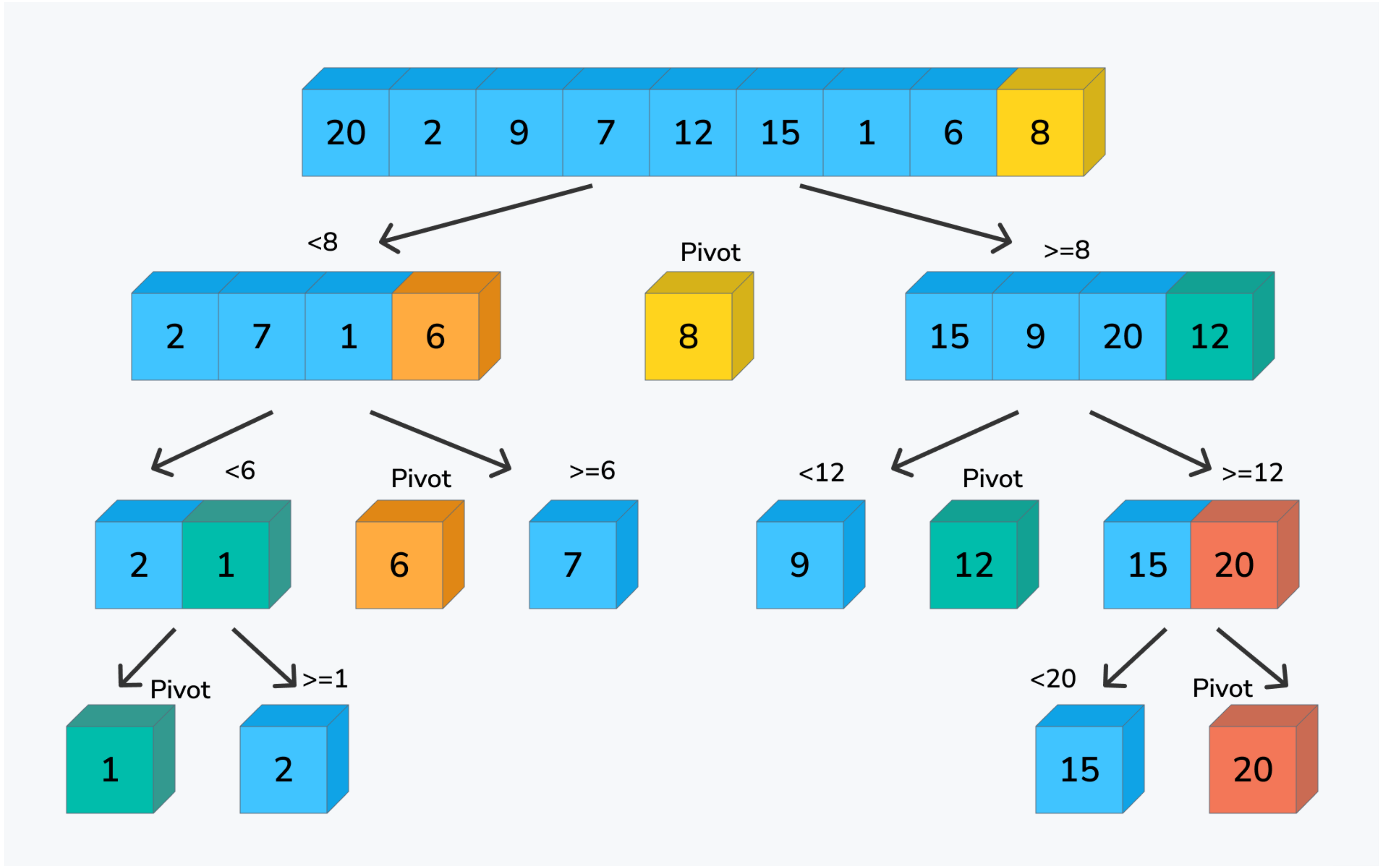
```
function merge_sort(arr):
    if length(arr) > 1:
        mid = length(arr) // 2
        left_half = arr[0:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        merge(arr, left_half, right_half)
```

```
function merge(arr, left, right):
    i = j = k = 0
    while i < length(left) and j < length(right):
        if left[i] < right[j]:
            arr[k] = left[i]
            i = i + 1
        else:
            arr[k] = right[j]
            j = j + 1
        k = k + 1
    while i < length(left):
        arr[k] = left[i]
        i = i + 1
        k = k + 1
    while j < length(right):
        arr[k] = right[j]
        j = j + 1
        k = k + 1
```

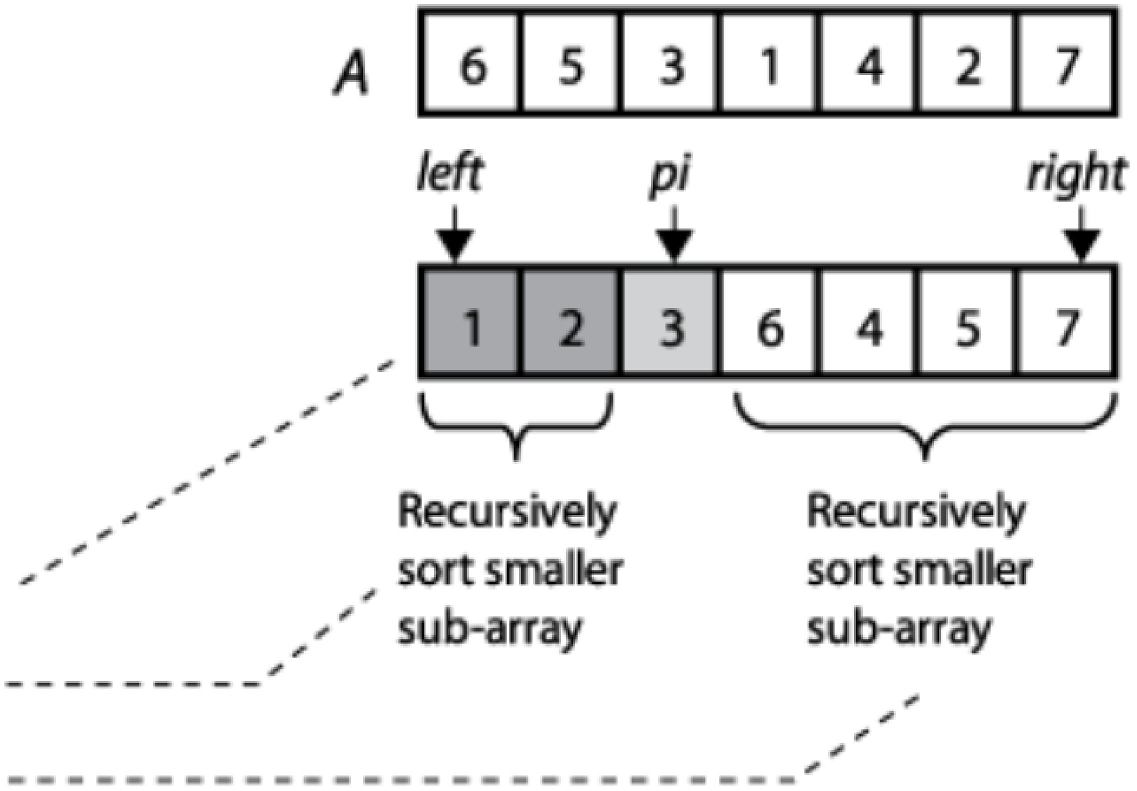
Quick sort

- Quick Sort is a highly efficient and widely used sorting algorithm that falls under the category of divide-and-conquer algorithms.
- The algorithm works by ***selecting a "pivot" element*** from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot.
- The sub-arrays are then recursively sorted.
- Here's a high-level overview of the Quick Sort algorithm:
  - **Partitioning**:
    - Choose a pivot element from the array.
    - Partition the other elements into two sub-arrays: elements less than the pivot and elements greater than the pivot.
  - **Recursion:**
    - Recursively apply the Quick Sort algorithm to the two sub-arrays.
  - **Combine:**
    - The sorted sub-arrays are combined to produce the final sorted array.

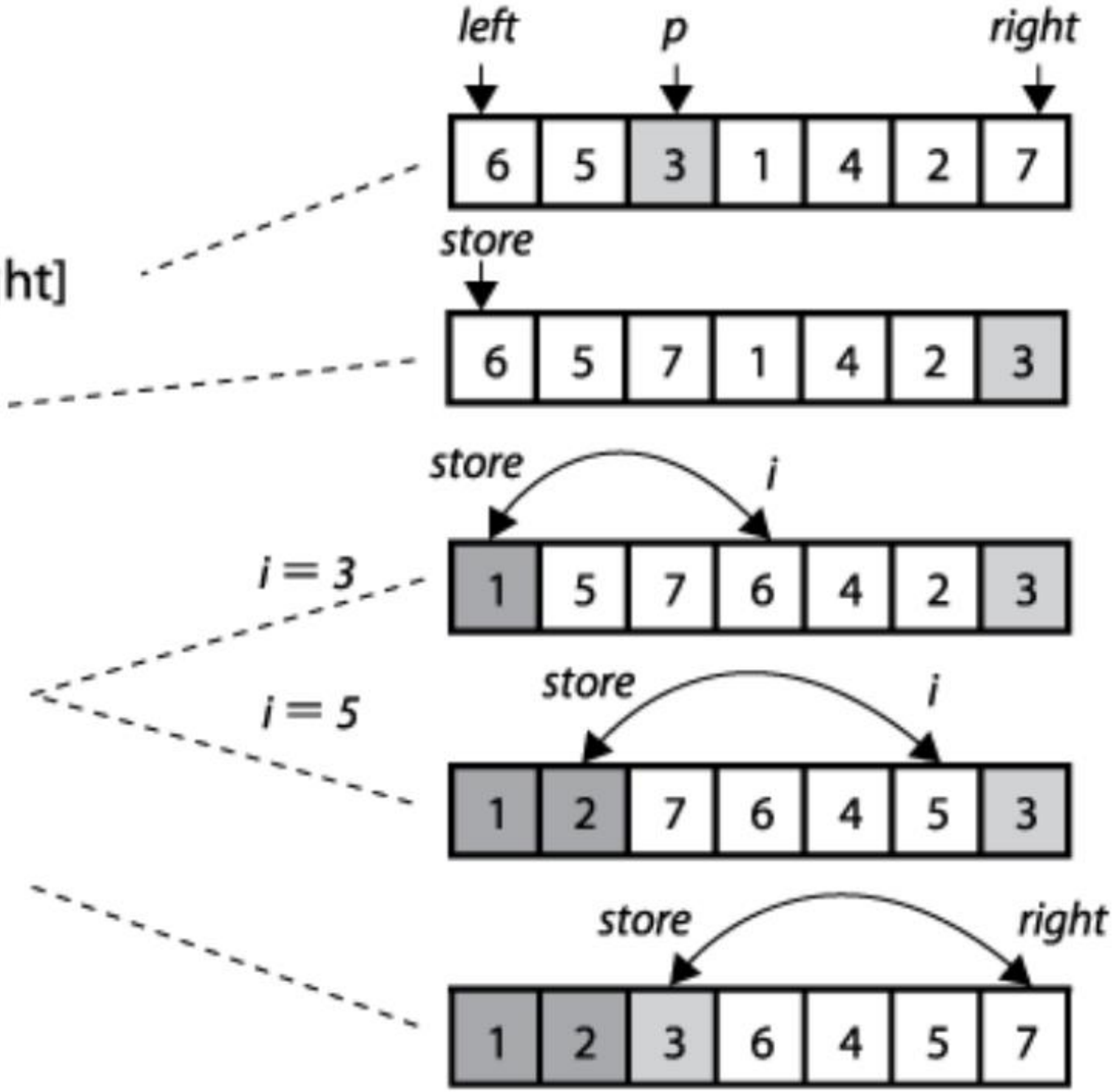
Quick sort


20 2 9 7 12 15 1 6 8
<8
Pivot
>=8
2 7 1 6
8
15 9 20 12
<6
Pivot
>=6
<12
Pivot
>=12
2 1
6
7
9
12
15 20
Pivot
>=1
<20
Pivot
1
2
15
20

```
sort (A)
1.    quickSort (A, 0, n − 1)
end

quickSort (A, left, right)
1.    if (left < right) then
2.        pi = partition (A, left, right)
3.        quickSort (A, left, pi − 1)
4.        quickSort (A, pi + 1, right)
end
```


A
6 5 3 1 4 2 7
left
pi
right
1 2 3 6 4 5 7
Recursively sort smaller sub-array
Recursively sort smaller sub-array

```
partition (A, left, right)
1.    p = select pivot in A[left, right]
2.    swap A[p] and A[right]
3.    store = left
4.    for i = left to right − 1 do
5.      if (A[i] ≤ A[right]) then
6.          swap A[i] and A[store]
7.          store++
8.    swap A[store] and A[right]
9.    return store
end
```



Selection Sort, Merge Sort, and Quick Sort

Selection Sort	Merge Sort	Quick Sort
• **Time Complexity:** O(n^2) in the worst and average cases. • **In-Place:** Yes. • **Stability:** No. • **Use Cases:** Small datasets, simple implementations, and situations where in-place sorting is crucial.	• **Time Complexity:** O(n log n) in all cases (worst, average, and best). • **In-Place:** No (requires additional space for merging). • **Stability:** Yes. • **Use Cases:** Large datasets, situations where stability is important, linked lists, and external sorting.	• **Time Complexity:** O(n log n) in the average case, O(n^2) in the worst case (can be mitigated with randomization or careful pivot selection). • **In-Place:** Yes (with the potential for log n recursion stack in the worst case). • **Stability:** No. • **Use Cases:** General-purpose sorting, large datasets, situations where average-case performance is crucial.

Selection Sort, Merge Sort, and Quick Sort

- **Time Complexity:**
  - Quick Sort has an average-case time complexity of **O(n log n)**, making it more efficient than Selection Sort for larger datasets. However, Quick Sort's worst-case time complexity is **O(n^2)**, which can be a concern in certain situations.
  - Merge Sort consistently performs at **O(n log n)** in all cases, making it a reliable choice when a stable sort is needed.
  - Selection Sort consistently has a time complexity of **O(n^2)**, making it less efficient for large datasets.
- **In-Place vs. Extra Space:**
  - Selection Sort and Quick Sort are in-place algorithms, meaning they don't require additional memory proportional to the size of the input.
  - Merge Sort, on the other hand, requires additional space for merging, proportional to the size of the input. This can be a drawback in terms of space complexity.

Selection Sort, Merge Sort, and Quick Sort

- **Stability:**
  - Quick Sort is not stable, meaning equal elements may not maintain their original order.
  - Selection Sort is not stable.
  - Merge Sort is stable.
- **Ease of Implementation:**
  - Selection Sort is straightforward to implement.
  - Quick Sort involves partitioning and recursion and can be more challenging to implement correctly.
  - Merge Sort is relatively easy to implement but involves additional memory.

Finding the Maximum and Minimum

- The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array.
- To find the maximum and minimum numbers in a given array ***numbers[ ]*** of size **n**, the following algorithm can be used.
    - **naive method**
    - **divide and conquer approach**.

Naïve Method

- Naïve method is a basic method to solve any problem. In this method, the maximum and minimum number can be found separately.
- ***Algorithm:***
- *Naive_Max_Min(numbers[ ])*

```
Initialize max_value and min_value to the first element of the array
For each element num in numbers[1:]:
    If num > max_value:
          update max_value
    If num < min_value:
          update min_value
Return (max_value, min_value)
```

- The number of comparison in Naive method is ***2n - 2***. The number of comparisons can be reduced using the divide and conquer approach.

Divide and Conquer Approach

- Divide the array into two halves.
- Recursively find the maximum and minimum values in each half.
- Combine the results by taking the maximum of two maxima and the minimum of two minima.

Divide and Conquer Approach

```
DivideConquer_Max_Min(numbers, low, high):
    If low == high:
        return (numbers[low], numbers[low])
    If high == low + 1:
        compare numbers[low] and numbers[high] to determine max and min
    Otherwise:
        mid = (low + high) / 2
        (max1, min1) = DivideConquer_Max_Min(numbers, low, mid)
        (max2, min2) = DivideConquer_Max_Min(numbers, mid + 1, high)
        Return (max(max1, max2), min(min1, min2))
```

STRASSEN'S MATRIX MULTIPLICATION

Strassen's Matrix Multiplication

- Indeed, Strassen's Matrix Multiplication is a ***divide-and-conquer-based algorithm*** that aims to reduce the time complexity of matrix multiplication compared to the traditional cubic time complexity (O(n^3)).
- The algorithm operates by dividing each input matrix into *four smaller matrices and recursively multiplying these smaller matrices.*
- It involves *seven recursive multiplications instead of the traditional eight.*
- The time complexity of Strassen's algorithm is approximately *O(n^log2(7)), which is about O(n^2.81).*
- This improvement over the naive cubic time complexity is achieved through clever mathematical manipulations.

Strassen's Matrix Multiplication

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

- For larger matrices this approach will continue until we recurse all the smaller sub matrices.
- Suppose we have two matrices, A and B, and we want to multiply them to form a new matrix, C.
- C=AB, where all A,B,C are square matrices. We will divide these larger matrices into smaller sub matrices n/2.

Strassen's Matrix Multiplication

- **Matrix Partitioning:**
  - Given two matrices A and B, each of size n x n, divide them into four equal-sized submatrices. Let A be divided into A11, A12, A21, and A22, and B into B11, B12, B21, and B22.
- **Recursive Multiplication:**
  - Recursively compute seven products using the smaller submatrices:
    - P1 = A11 * (B12 - B22)
    - P2 = (A11 + A12) * B22
    - P3 = (A21 + A22) * B11
    - P4 = A22 * (B21 - B11)
    - P5 = (A11 + A22) * (B11 + B22)
    - P6 = (A12 - A22) * (B21 + B22)
    - P7 = (A11 - A21) * (B11 + B12)

Strassen's Matrix Multiplication

- **Matrix Combinations:**
  - Compute the four quadrants of the result matrix using these products:
    - C11 = P5 + P4 - P2 + P6
    - C12 = P1 + P2
    - C21 = P3 + P4
    - C22 = P5 + P1 - P3 - P7
- **Merge:**
  - Combine the four quadrants to form the final result matrix C.

Strassen's Matrix Multiplication

$p1 = a(f - h)$

$p2 = (a + b)h$

$p3 = (c + d)e$

$p4 = d(g - e)$

$p5 = (a + d)(e + h)$

$p6 = (b - d)(g + h)$

$p7 = (a - c)(e + f)$

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}_{C}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Strassen's Matrix Multiplication

- Steps of Strassen’s matrix multiplication:
  - *Divide the matrices A and B into smaller submatrices of the size n/2xn/2.*
  - *Using the formula of scalar additions and subtractions compute smaller matrices of size n/2.*
  - *Recursively compute the seven matrix products $P_i = A_i B_i$ for i=1,2,…7.*
  - *Now compute the r,s,t,u submatrices by just adding the scalars obtained from above points.*
- In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of Strassen’s method is to reduce the number of recursive calls to 7.
- **Exercise**

$$A = \begin{bmatrix} 5 & 6 \\ -4 & 3 \end{bmatrix} ; \quad B = \begin{bmatrix} -7 & 6 \\ 5 & 9 \end{bmatrix}$$

RECURRENCE RELATION

Recurrence Relation

- A recurrence relation is a mathematical expression that ***defines a sequence recursively in terms of one or more of its preceding terms.*** In other words, it is ***a way of defining the terms of a sequence based on the values of earlier terms in the sequence.***
- A general form of a recurrence relation for a sequence {a_n} is often written as:

$$a_n = f(a_{n-1}, a_{n-2}, \ldots, a_{n-k})$$

- Here, a_n represents the nth term in the sequence, and f is a function that relates the current term to one or more previous terms (up to k previous terms).
- Recurrence relations are commonly used in various branches of *mathematics, computer science, and other scientific fields* to model and analyze processes that evolve over time in a step-by-step manner.
- For example, the Fibonacci sequence is defined by the recurrence relation:

$$F(n) = F(n-1) + F(n-2)$$

- with initial conditions **$F(0)=0$ and $F(1)=1$.** This recurrence relation defines each term in the Fibonacci sequence in terms of the two preceding terms.

Recurrence Relation

- Let's consider a simple example of a recurrence relation. Suppose we have a sequence defined by the recurrence relation:

$$a_n = 2a_{n-1} + 3$$

- with an initial condition $a_0=1$.
- Let's use this recurrence relation to find the first few terms of the sequence:

$$a_1 = 2a_0+3 = 2\times1+3=5$$

$$a_2 = 2a_1+3 = 2\times5+3=13$$

$$a_3 = 2a_2+3 = 2\times13+3=29$$

- This recurrence relation expresses each term a_n in terms of the previous term a_{n-1} and a constant term (3 in this case). Solving the recurrence relation means finding ***a closed-form expression for a_n in terms of n without referring to previous terms.***
- Depending on the specific recurrence relation, this process might involve techniques like substitution, characteristic equations, or other methods depending on the complexity of the relation.

Recurrence Relation

- **Linear Recurrence Relations:**
  - The example provided earlier ***($a_n = 2a_{n-1} + 3$)*** is a linear recurrence relation because each term is a linear combination of previous terms.
  - Linear recurrence relations are common and often have closed-form solutions.
- **Initial Conditions:**
  - To fully specify a sequence defined by a recurrence relation, you typically need initial conditions. In the example, ***$a_0 = 1$*** is an initial condition.
- **Degree of Recurrence:**
  - The degree of a recurrence relation is determined by the highest power of the term a_n in the relation. In the linear example, the degree is 1.
- **Homogeneous and Non-homogeneous Recurrence Relations:**
  - A homogeneous recurrence relation has no external terms (like constants) on the right-hand side of the equation. For example, ***$a_n = 2a_{n-1}$*** is ***homogeneous.***
  - A non-homogeneous recurrence relation includes external terms. The previous example ***$a_n = 2a_{n-1} + 3$*** is ***non-homogeneous.***

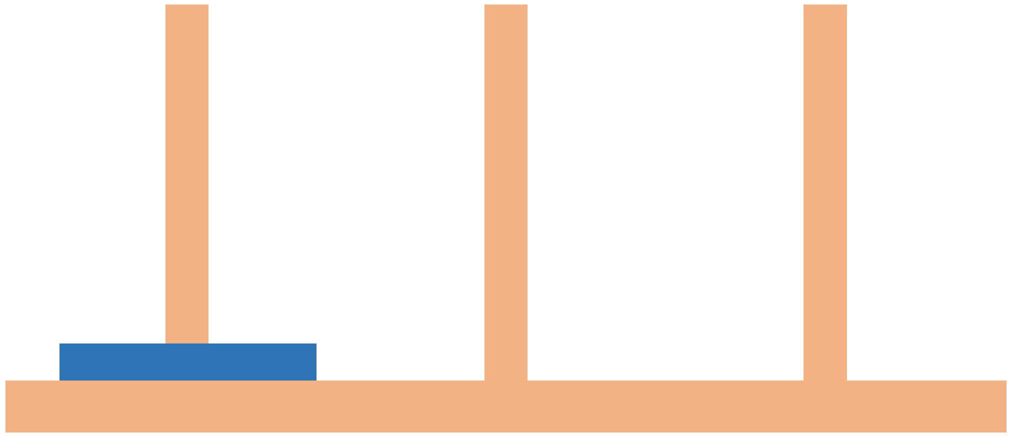
Setting up a Recurrence Relation

- Recurrence relations are used to reduce complicated problems to an iterative process based on simpler versions of the problem. An example problem in which this approach can be used is the Tower of Hanoi puzzle.
- It's not immediately clear that a recursion solution will work for this problem. However, there are a couple things about this problem that make it a good candidate to solve with a recurrence relation.
- **Identifying a candidate problem to solve with a recurrence relation:**
  - The problem can be reduced to simpler cases. The Tower of Hanoi puzzle can be simplified by removing some of the disks.
  - There is a numerical value, *n*, to identify each case. For the Tower of Hanoi puzzle, this numerical value is the number of disks.
  - The problem increases in complexity as the numerical identifier increases. As the number of disks in the Tower of Hanoi problem increases, it becomes more difficult to find a solution.
- The goal for this exercise will be to develop a recurrence relation for T

Setting up a Recurrence Relation

- Recurrence relations are used to reduce complicated problems to an iterative process based on simpler versions of the problem. ***An example problem in which this approach can be used is the Tower of Hanoi puzzle.***
- The Tower of Hanoi puzzle consists of three vertical pegs and several disks of various sizes. Each disk has a hole in its center for the pegs to go through.
- The rules of the puzzle are as follows:
    - The puzzle begins with all disks placed on one of the pegs. They are placed in order of largest to smallest, bottom to top.
    - The goal of the puzzle is to move all of the disks onto another peg.
    - Only one disk may be moved at a time, and disks are always placed onto pegs.
    - Disks may only be moved onto an empty peg or onto a larger disk.
- Let T_n be defined as the minimum number of moves needed to solve a puzzle with n disks.

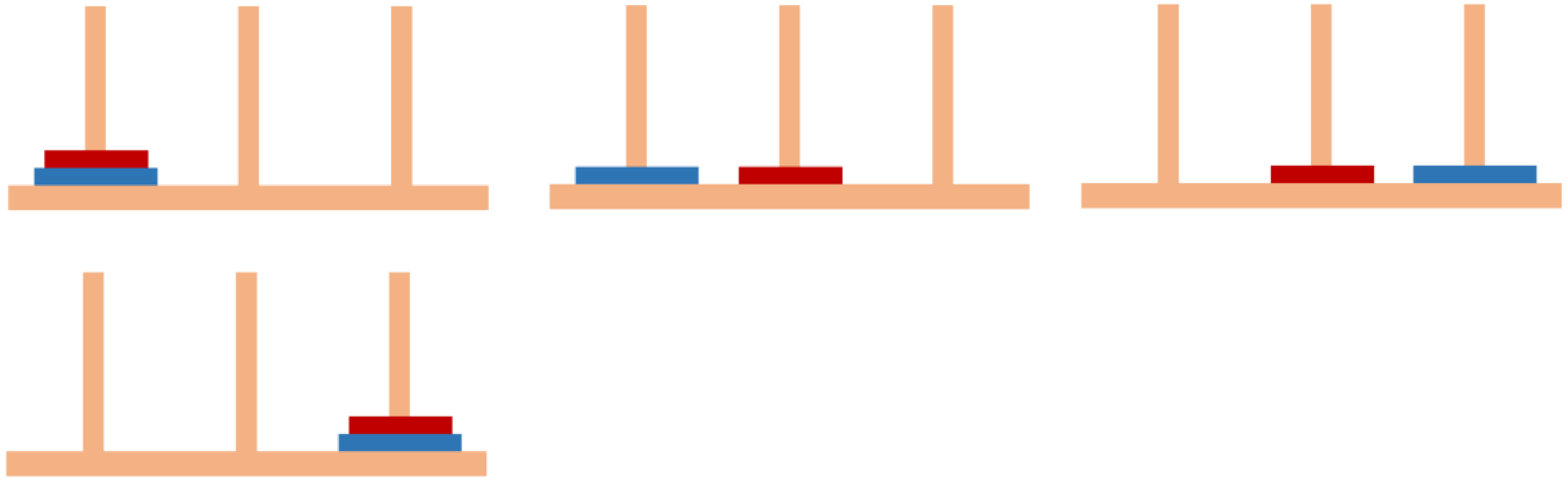
Recurrence Relation



- The most simple version of the Tower of Hanoi puzzle would contain only one disk.
- In terms of the recurrence relation, n=1.
  - T_0=1, because it would only take 1 move to move all the disks to another peg.

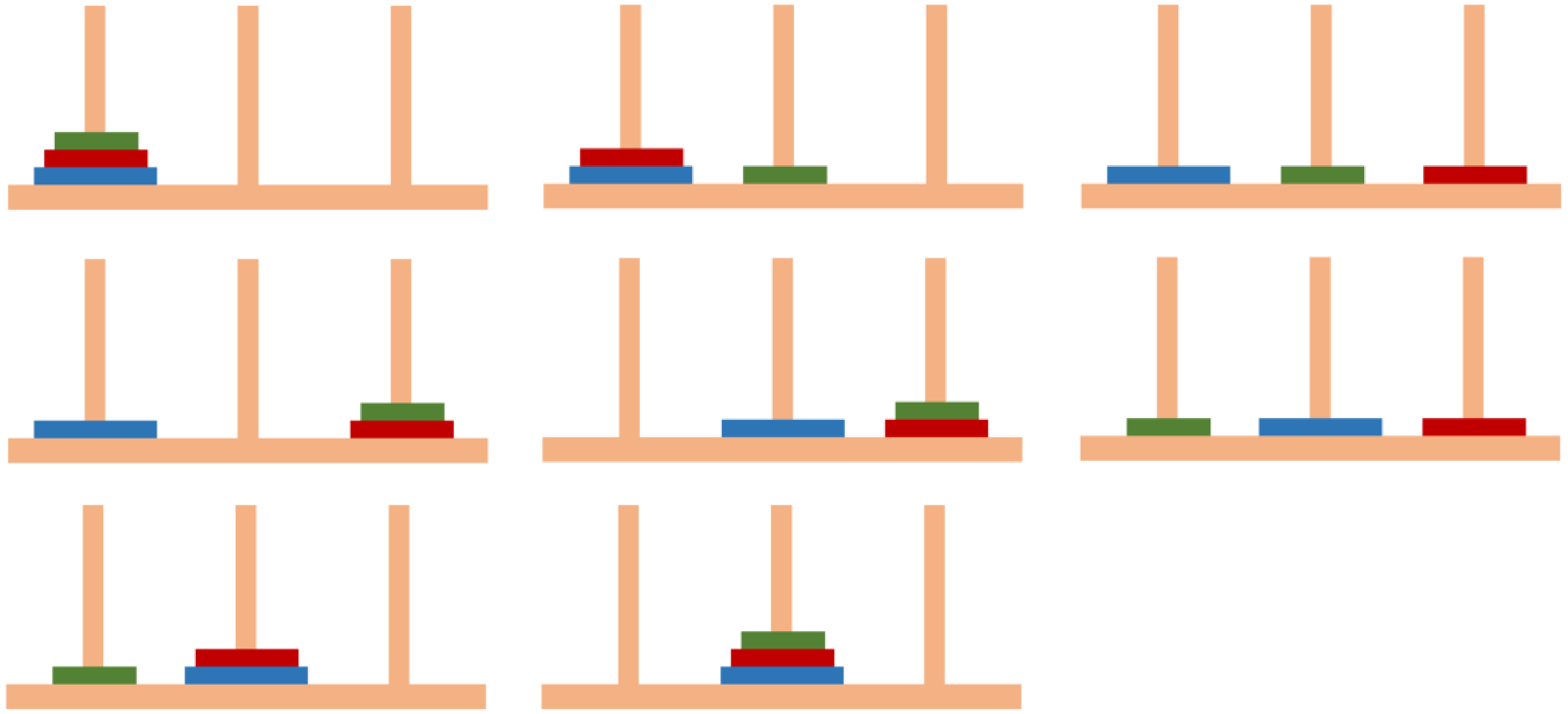
Recurrence Relation

- Below is the solution to a Tower of Hanoi puzzle with n=2.
- It can be seen from below that T_1 =3.



Recurrence Relation

- Below is a solution to a Tower of Hanoi puzzle with n=3.
- It can be seen from above that $T_2 = 7$.



Recurrence Relation

- In terms of *n*,
  - Do T_{n-1} moves to get the smaller disks off the largest disk.
  - Do 1 move to move the largest disk.
  - Do T_{n-1} moves to get the smaller disks back onto the largest disk.
- In total, the number of moves for *n* disks is
  - $T_n = 2\,T_{n-1} + 1$

Recurrence Relation

```
#include <stdio.h>
void Test(int n) {
    if (n > 0) {
        printf("%d ", n);  // Print the value
        Test(n - 1);       // Recursive call
    }
}
int main() {
    int n = 5;  // You can change this to the desired value of n
    Test(n);
    return 0;
}
```

Preparing Recurrence Relation

```
void Test(int n) { // n
    if (n > 0) {
        printf("%d ", n);  // 1
        Test(n - 1);       // n-1
    }
}
```

- ***So, $T(n) = T(n-1) + 1$***
- ***Hence, $T(n) = 1$ for $n=0$, and $T(n-1) + 1$ for $n>0$***

Solving Recurrence Relation

- **Definition of Recurrence Relation:** $T(n) = T(n-1)+1$
- **Initial Condition:** $T(n) = 1$ for $n = 0$
- **Substituting and Simplifying:**
    - $T(n) = T(n-1) + 1$
    - $T(n-1) = T(n-2) + 1$
- Substitute $T(n-1)$ into the first equation: $T(n) = T(n-2) + 2$
- Continue substituting $T(n-k)$ for k times: $T(n) = T(n-k) + k$
- **Assuming n-k=0:** $T(n) = T(0) + n = n + 1$

Preparing Recurrence Relation

```
void Test(int n) {
    if (n > 0) {
        for (int i = 1; i < n; i = i + 1) {
            printf("%d", n);
        }
        Test(n - 1);
    }
}
```

Solving Recurrence Relation

- ***So, T (n) = T (n-1) + n***
- ***Hence, T (n) = 1 for n=0, and T (n-1) + n for n>0***
- ***Solving above recurrence relation:***
- *T (n) = T (n-1) + n*
- *T (n-1) = T (n-2) + (n-1)*
- *Substitute T (n-1), Hence T (n) = T (n-2) + (n-1) + n*
- *T (n) = T (n-3) + (n-2) + (n-1) + n ….. continue for k times*
- *Assume n-k=0, then n=k hence T(n) = 1 + 2 + 3 + ……. + n = n(n+1) / 2*
- *Time complexity = O(n^2)*

Preparing Recurrence Relation

```
void Test(int n) {
    if (n > 0) {
        for (int i = 1; i < n; i = i * 2) {
            printf("%d", n);
        }
        Test(n - 1);
    }
}
```

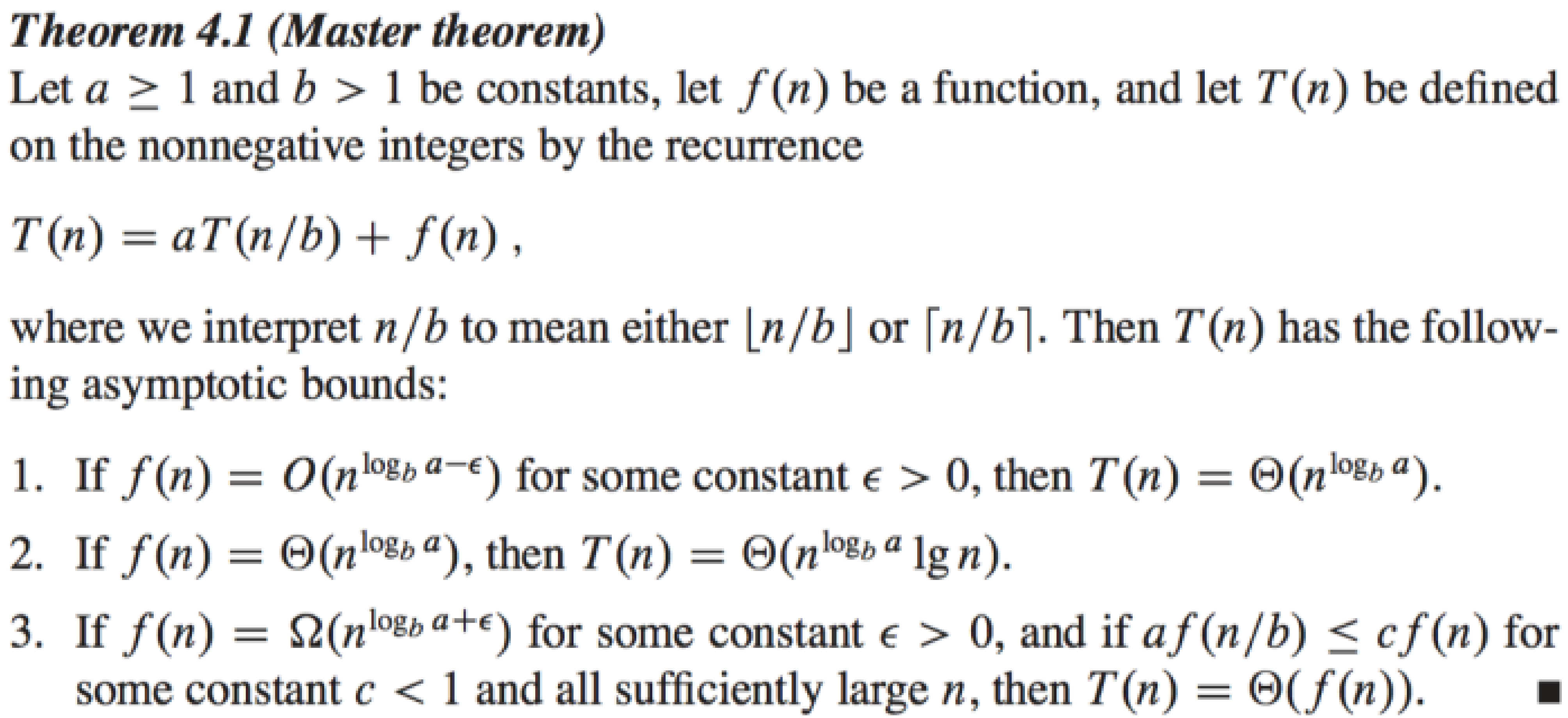
Solving Recurrence Relation

- ***So, T (n) = T (n-1) + log(n)***
- ***Hence, T (n) = 1 for n=0, and T (n-1) + log(n) for n>0***
- ***Solving above recurrence relation:***
- *T (n) = T (n-1) + log(n)*
- *T (n-1) = T (n-2) + log(n-1)*
- *Substitute T (n-1), Hence T (n) = T (n-2) + log(n-1) + log(n)*
- *T (n) = T (n-3) + log(n-2) + log(n-1) + log(n) ..... continue for k times*
- *T (n) = T (n-k) + log1 + log2 + ... + log(n-1) + log(n)*
- *Assume n-k=0, then n=k hence T (n) = T (0) + log(n!) = 1 + log(n!)*
- *Time complexity = O(nlogn)*

Recurrence Relation

- *Hence, from above relations:*
  - $T(n) = T(n-1) + 1 = O(n)$
  - $T(n) = T(n-1) + n = O(n^2)$
  - $T(n) = T(n-1) + logn = O(nlogn)$
  - $T(n) = T(n-1) + n^2 = O(n^3)$

Master Theorem



Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

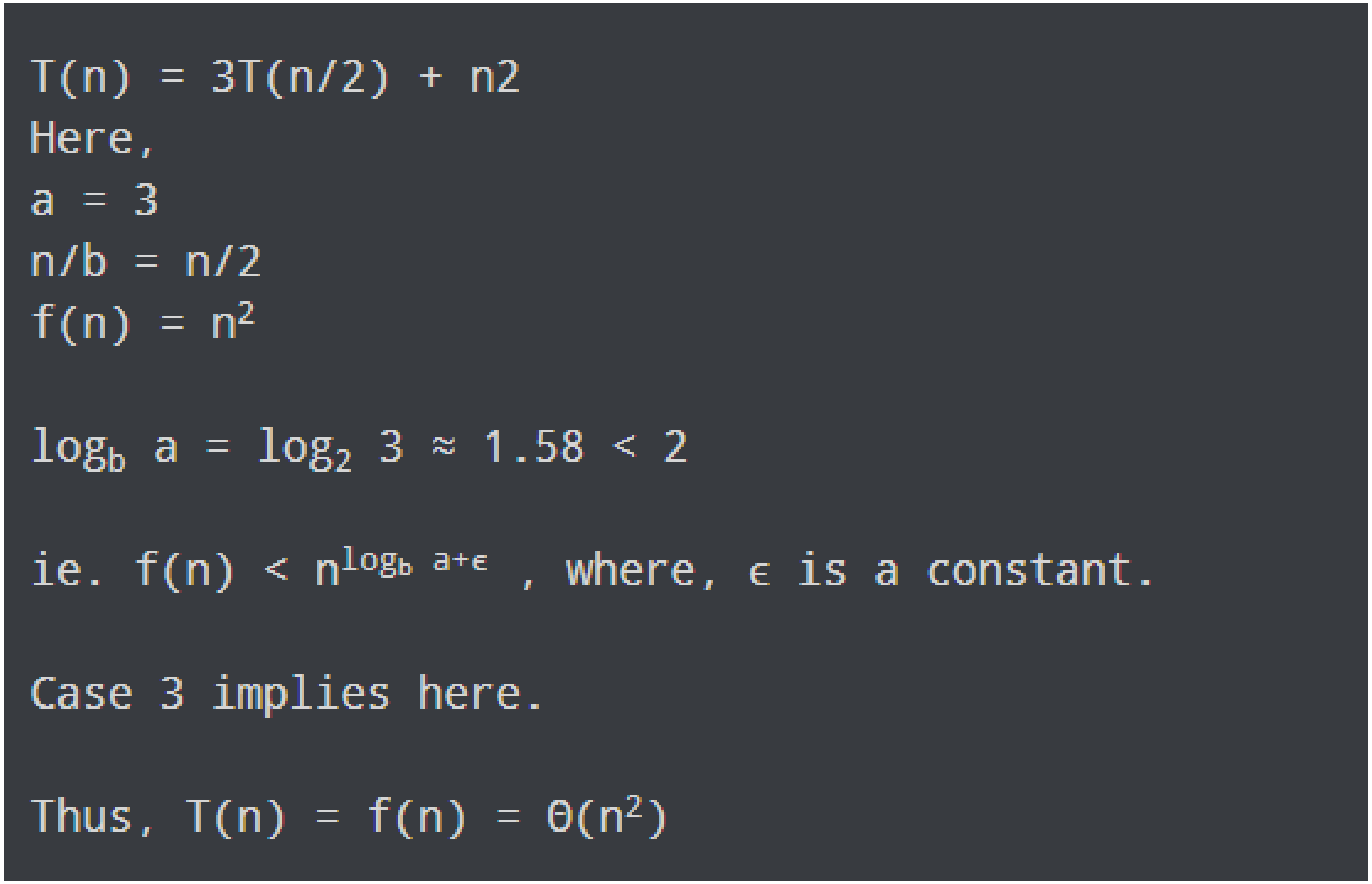
where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Master Method for recurrence relation

- In master method, the problem is divided into number of subproblems each of size n/b and need f(n) time to combine and or break the solution.
- $T(n) = aT(n/b) + f(n)$,
- where,
- n = size of input
- a = number of subproblems in the recursion
- n/b = size of each subproblem. All subproblems are assumed to have the same size.
- f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions
-
- Here, $a \geq 1$ and $b > 1$ are constants, and f(n) is an asymptotically positive function.

Example: Master Theorem



$T(n) = 3T(n/2) + n2$

Here,

$a = 3$

$n/b = n/2$

$f(n) = n^2$

$\log_b a = \log_2 3 \approx 1.58 < 2$

ie. $f(n) < n^{\log_b a+\epsilon}$, where, ϵ is a constant.

Case 3 implies here.

Thus, $T(n) = f(n) = \Theta(n^2)$

Example

Example-1: Binary Search – $T(n) = T(n/2) + O(1)$
$a = 1, b = 2, k = 0$ and $p = 0$
$b^k = 1$. So, $a = b^k$ and $p > -1$ [Case 2.(a)]
$T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
$T(n) = \theta(\log n)$

Example-2: Merge Sort – $T(n) = 2T(n/2) + O(n)$
$a = 2, b = 2, k = 1, p = 0$
$b^k = 2$. So, $a = b^k$ and $p > -1$ [Case 2.(a)]
$T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
$T(n) = \theta(n \log n)$

Example-3: $T(n) = 3T(n/2) + n^2$
$a = 3, b = 2, k = 2, p = 0$
$b^k = 4$. So, $a < b^k$ and $p = 0$ [Case 3.(a)]
$T(n) = \theta(n^k \log^p n)$
$T(n) = \theta(n^2)$

Example

Example-4: $T(n) = 3T(n/2) + \log^2 n$
$a = 3, b = 2, k = 0, p = 2$
$b^k = 1$. So, $a > b^k$ [Case 1]
$T(n) = \theta(n^{\log ba})$
$T(n) = \theta(n^{\log 23})$

Example-5: $T(n) = 2T(n/2) + n\log^2 n$
$a = 2, b = 2, k = 1, p = 2$
$b^k = 2$. So, $a = b^k$ [Case 2.(a)]
$T(n) = \theta(n^{\log ba}\log^{p+1} n)$
$T(n) = \theta(n^{\log 22}\log^3 n)$
$T(n) = \theta(n\log^3 n)$

Example-6: $T(n) = 2^n T(n/2) + n^n$
This recurrence can’t be solved using above method since function is not of form $T(n) = aT(n/b) + \theta(n^k \log^p n)$

THE GREEDY METHOD

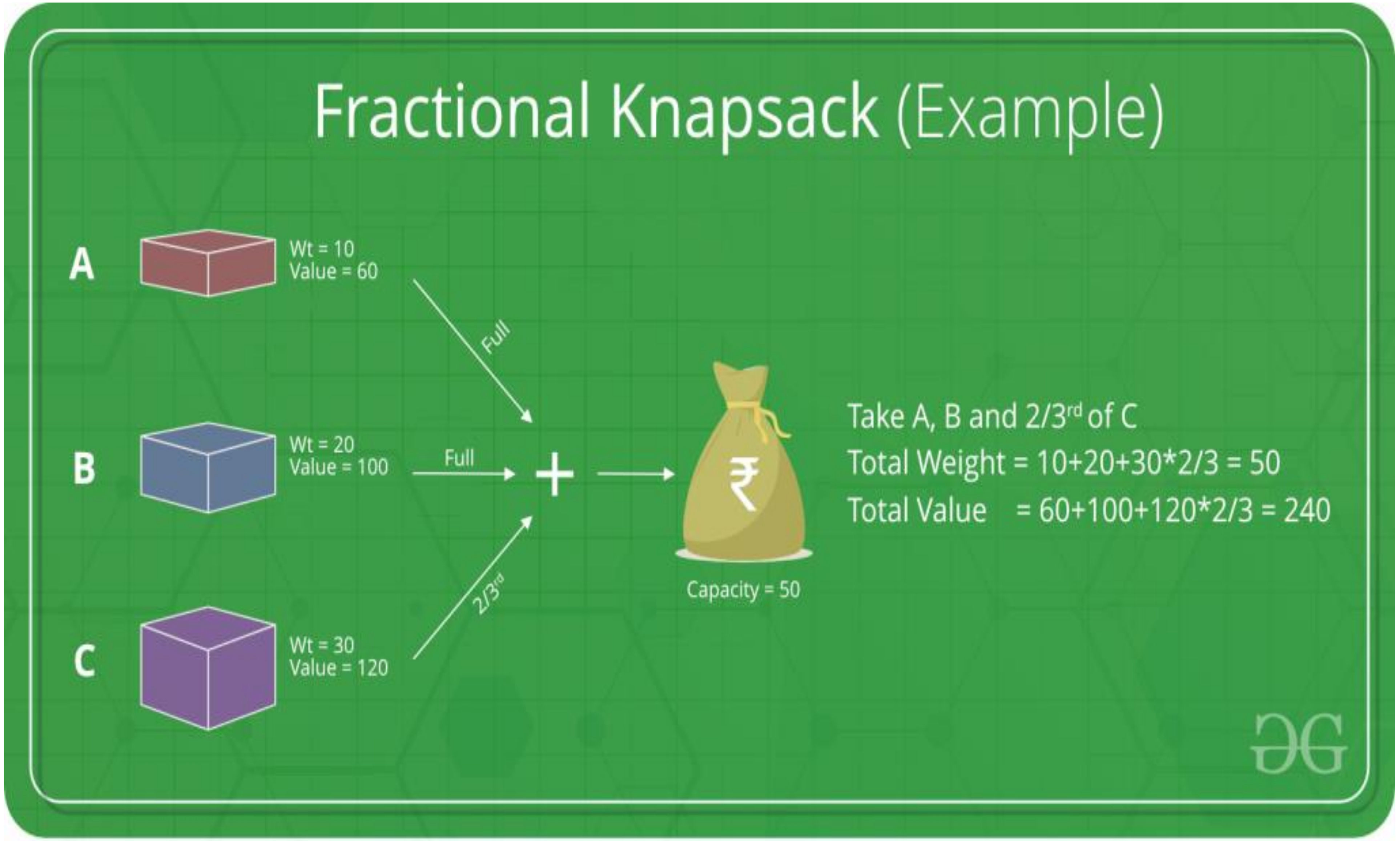
The Greedy Method

- The greedy method is a simple strategy of ***progressively building up a solution, one element at a time,*** by choosing the best possible element at each stage.
- At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order ***determined by some selection procedure.***
- If the inclusion of the next input, into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution.
- The selection procedure itself is based on some optimization measure. Several optimization measures are plausible for a given problem.
- Most of them, however, will result in algorithms that generate sub-optimal solutions. This version of greedy technique is called ***subset paradigm***.

The Greedy Method

- Some problems like ***Knapsack, Job sequencing with deadlines and minimum cost spanning trees*** are based on subset paradigm.
- For the problems that make decisions by considering the inputs in some order, each decision is made using an optimization criterion that can be computed using decisions already made.
- This version of greedy method is ordering paradigm. Some problems like optimal storage on tapes, optimal merge patterns and single source shortest path are based on ordering paradigm.

The Greedy Algorithm



The Greedy Algorithm

- Greedy is an algorithmic paradigm that builds up a solution piece by piece, always ***choosing the next piece that offers the most obvious and immediate benefit.***
- So, the problems where choosing locally optimal also leads to global solution are the best fit for Greedy.
- For example consider the ***Fractional Knapsack Problem.*** The local optimal strategy is to choose the item that has maximum value vs weight ratio.
- This strategy also leads to a globally optimal solution because we are allowed to take fractions of an item.

Fractional Knapsack Problem

- Given the ***weights and values*** of N items, in the form of {value, weight} put these items in a knapsack of capacity W to get the maximum total value in the knapsack.
- In Fractional Knapsack, *we can break items for maximizing the total value of the knapsack.*
- In the **0-1 Knapsack problem,** *we are not allowed to break items. We either take the whole item or don't take it.*
- **Input:** arr[] = {{60, 10}, {100, 20}, {120, 30}}, W = 50
- **Output:** 240
- Explanation: By taking items of weight 10 and 20 kg and 2/3 fraction of 30 kg.
- **Hence total price will be 60 + 100 + (2/3) (120) = 240**
- Input: arr[] = {{500, 30}}, W = 10
- Output: 166.667

Fractional Knapsack Problem

- The basic idea of the greedy approach is to calculate the ratio ***value/weight*** for each item and sort the item on the basis of this ratio.
- Then take the *item with the highest ratio and add them until we can't add the next item as a whole and at the end add the next item* as much as we can. Which will always be the optimal solution to this problem. It uses following steps:

Calculate the ratio(value/weight) for each item.

sort all the items in decreasing order of the ratio.

initialize result =0, current_capacity = given_capacity.

do the following for every item "i" in the sorted order:

If the weight of the current item is less than or equal to the remaining capacity then add the value of that item into the result

Else add the current item as much as we can and break out of the loop.

return result.

Knapsack Problem

- Say we have knapsack of capacity(m) = 20

Objects	Obj1	Obj2	Obj3
Profit	25	24	15
Weight	18	15	10

1. Greedy about profit : Obj1 have max profit. Now Remaining weight: 2kg. Now obj2 can only be included 2/15 of obj2. So overall profit = 25 + 24*2/15 = 28.2

2. Greedy about weight: Here we put least profitable object. Minimum weight object is obj3, we put it in the bag. Then 10/15 of the obj2 is added.

So ***overall profit:*** 15 + 24*2/3 = 31

Knapsack Problem

- Say we have knapsack of capacity(m) = 20

Objects	Obj1	Obj2	Obj3
Profit	25	24	15
Weight	18	15	10
p/w	1.3	1.6	1.5

- But we are not getting optimal solution from the previous method.
- So compute profit/weight
- So highest proportion is of object 2. We put obj2 in knapsack.
- Remaining capacity: 20-15 = 5kg
- Then we select 5/10th of obj3. So, remaining capacity: 0kg
- **Profit = 24 + 15 * ½ = 24+7.5 = 31.5**

Knapsack Problem Algorithm

- **Knapsack Problem: Algorithm**

```
for i=1 to n:
    calculate profit / weight  ratio
Sort objects in decreasing order of profit / weight  ratio
for i=1 to n:
    If $m > 0$ and $w_i <= m$
        $m = m - w_i$
        $p = p + p_i$
    else break,
if($m > 0$)
    $p = p + p_i (m / w_i)$
```

```
Input: n: Number of items,  weights[]: Array of item weights, profits[]: Array of item profits, m:
    Knapsack capacity
1. Create an array ratio[] to store profit/weight ratios for each item.
2. For i = 1 to n:
   a. Calculate profit/weight ratio for each item: ratio[i] = profits[i] / weights[i].
3. Sort objects in decreasing order of profit/weight ratio.
4. Initialize variables:
   - totalProfit = 0, remainingCapacity = m
5. For i = 1 to n:
   a. If remainingCapacity > 0 and weights[i] <= remainingCapacity:
      - Include the entire item in the knapsack.
      - remainingCapacity  - = weights[i]
      - totalProfit += profits[i]
   b. Otherwise, break out of the loop.
6. If remainingCapacity > 0:
   - Include a fraction of the next item to fill the remaining capacity.
   - totalProfit += (remainingCapacity / weights[i]) * profits[i]
7. Output totalProfit as the maximum achievable profit.
```

Knapsack Problem

Object(O)	1	2	3	4	5	6	7
Profits(P)	10	5	15	7	6	18	3
Weights(w)	2	3	5	7	1	4	1

- There are n=7 items
- Capacity, m = 15 kg
- So, problem is container loading problem

Knapsack Problem

Object(O)	1	2	3	4	5	6	7
Profits(P)	10	5	15	7	6	18	3
Weights(w)	2	3	5	7	1	4	2

- There are n=7 items
- Capacity, m = 15 kg
- So, problem is container loading problem

P/w	5	1.6	3	1	6	4.5	3
X	Second: 1				First: 1		
	x1	x2	x3	x4	x5	x6	x7

- 15-1 = 14kg
- 14-2 = 12kg and so on

Knapsack Problem

Object(O)	1	2	3	4	5	6	7
Profits(P)	10	5	15	7	6	18	3
Weights(w)	2	3	5	7	1	4	2

P/w	5	1.6	3	1	6	4.5	3
X	Second: 1	2/3	1	0	First: 1	1	1
	x1	x2	x3	x4	x5	x6	x7

15-1 = 14kg
14-2 = 12kg
12-4 = 8kg
8-5 = 3kg
3-1 = 2kg
2-2 = 0

Knapsack Problem

Object(O)	1	2	3	4	5	6	7
Profits(P)	10	5	15	7	6	18	3
Weights(w)	2	3	5	7	1	4	2

P/w	5	1.3	3	1	6	4.5	3
X	Second: 1	2/3	1	0	First: 1	1	1
	x1	x2	x3	x4	x5	x6	x7

- Compute sum($x_i w_i$)
- Profit sum($x_i p_i$)
- Constraint Sum($x_i w_i$) <= m
- Objective Sum($x_i p_i$) needs to be maximum
- Hence constraint needs to be fulfilled having maximized objective function.
- **Code Implementation**

Job Sequencing with deadlines

- When we are given a set of ‘n’ jobs. Associated with each Job i, ***deadline*** $\boldsymbol{d_i} \boldsymbol{>= 0}$ ***and profit*** $\boldsymbol{P_i >= 0}$.
- For any job ‘i’ the profit p_i is earned iff the job is completed by its deadline.
- Only one machine is available for processing jobs. An optimal solution is the feasible solution with maximum profit.
- Sort the jobs in ‘j’ ordered by their deadlines. The array ***d [1 : n]*** is used to store the deadlines of the order of their p-values.
- The set of jobs j [1 : k] such that $j[r], 1 \leq r \leq k$ are the jobs in ‘j’ and $d(j[1]) \leq d(j[2]) \leq \dots \leq d(j[k])$.

Job Sequencing with deadlines

- There are ***n jobs*** to be processed on a machine.
- Each job ***i*** has a deadline $d_i \geq 0$ and profit $p_i \geq 0$.
- P_i is earned iff the job is completed by its deadline.
- The job is completed if it is processed on a machine for unit time.
- Only ***one machine is available*** for processing jobs.
- Only ***one job is processed at a time*** on the machine.
- A feasible solution is a subset of jobs J such that ***each job is completed by its deadline.***
- An optimal solution is a feasible ***solution with maximum profit value.***

Job Sequencing with deadlines

Let n = 4, (P_1, P_2, P_3, P_4,) = (100, 10, 15, 27) and (d_1 d_2 d_3 d_4) = (2, 1, 2, 1). The feasible solutions and their values are:

S. No	Feasible Solution	Procuring sequence	Value	Remarks
1	1,2	2,1	110	
2	1,3	1,3 or 3,1	115	
3	1,4	4,1	127	**OPTIMAL**
4	2,3	2,3	25	
5	3,4	4,3	42	
6	1	1	100	
7	2	2	10	
8	3	3	15	
9	4	4	27	

Using Greedy Method

- **Step-01:**
  - Sort all the given jobs in decreasing order of their profit.
- **Step-02:**
  - Check the value of maximum deadline.
  - Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.
- **Step-03:**
  - Pick up the jobs one by one.
  - Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.

Using Greedy Method

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

- Write the optimal schedule that gives maximum profit.
- Are all the jobs completed in the optimal schedule?
- What is the maximum earned profit?

Using Greedy Method

Jobs	J4	J1	J3	J2	J5	J6
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

- **Step-01:** Sort all the given jobs in decreasing order of their profit.
- **Step-02:** Value of maximum deadline: 5. So, draw a Gantt chart with maximum time on Gantt chart = 5.
- We take each job one by one in the order they appear in Step-01.
- We place the job on Gantt chart as far as possible from 0.
- We take job J4. Since its deadline is 2, so we place it in the first empty cell before deadline 2.
- We take job J1. Since its deadline is 5, so we place it in the first empty cell before deadline 5.
- We take job J3. Since its deadline is 3, so we place it in the first empty cell before deadline 3.
- We take job J2. Since its deadline is 3, so we place it in the first empty cell before deadline 3.
- Now, we take job J5. Since its deadline is 4, so we place it in the first empty cell before deadline 4.
- The only job left is job J6 whose deadline is 2. All the slots before deadline 2 are already occupied. Thus, job J6 can not be completed.

Using Greedy Method

Job Considered	Slot Assigned	Solution	Profit
---------	-------	---------	0
J1	[4, 5]	J1	200
J2	[0, 1], [4, 5]	J2, J1	380
J3	[0, 1], [2, 3], , [4, 5]	J2, J3, J1	570
J4	[0, 1], [1, 2], [2, 3], [4, 5]	J2, J4, J3, J1	870
J5	[0, 1], [1, 2], [2, 3], [3, 4], [4, 5]	J2, J4, J3, J5, J1	990
J6	-------	-------	----------

Using Greedy Method

- <u>**Summary:**</u>
  - The optimal schedule is: **J2 , J4 , J3 , J5 , J1**
  - This is the required order in which the jobs must be completed in order to obtain the maximum profit.
  - All the jobs are not completed in optimal schedule.
  - This is because job J6 could not be completed within its deadline.
  - **Maximum earned profit**
    - Sum of profit of all the jobs in optimal schedule
    - Profit of job J2 + Profit of job J4 + Profit of job J3 + Profit of job J5 + Profit of job J1
    - 180 + 300 + 190 + 120 + 200 = 990 units

Exercise

Jobs	J1	J2	J3	J4	J5
Deadlines	2	1	3	2	1
Profits	60	100	20	40	20

Jobs	J1	J2	J3	J4	J5
Profits	20	15	10	5	1
Deadlines	2	2	1	3	3

Optimal Merge Patterns

- Let's say we want merge two array:

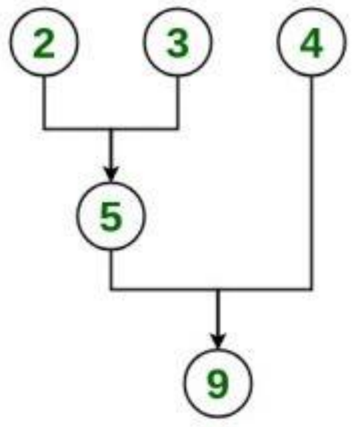
A	B
3	5
8	9
12	11
20	16

- Here we can compare elements from two array and place in the new array.
- Time taken to merge these items is the sum of number of elements in each list.
- Say we have 5 array with multiple elements to be merged, then we need optimal way to merge them.

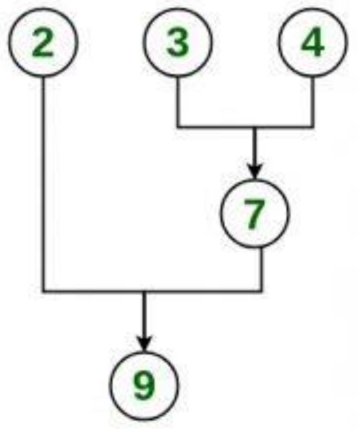
Optimal Merge Patterns

- Given n number of sorted files, the task is to find the minimum computations done to reach the Optimal Merge Pattern.
- When two or more sorted files are to be merged altogether to form a single file, the minimum computations are done to reach this file are known as **Optimal Merge Pattern**.
- If more than 2 files need to be merged then it can be done in pairs. For example, if need to merge 4 files A, B, C, D. ***First Merge A with B to get X1, merge X1 with C to get X2, merge X2 with D to get X3 as the output file.***
- If we have two files of sizes m and n, the total computation time will be m+n. Here, we use the greedy strategy by merging the two smallest size files among all the files present.
- Examples: Given 3 files with sizes 2, 3, 4 units. Find an optimal way to combine these files

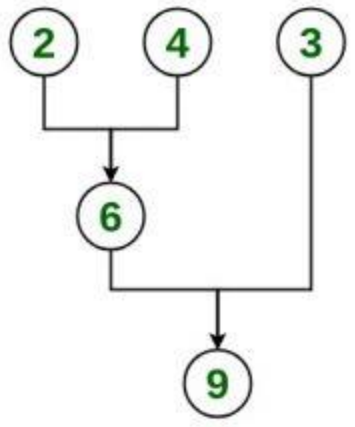
Optimal Merge Patterns



Cost = 5 + 9 = 14



Cost = 7 + 9 = 16



Cost = 6 + 9 = 15

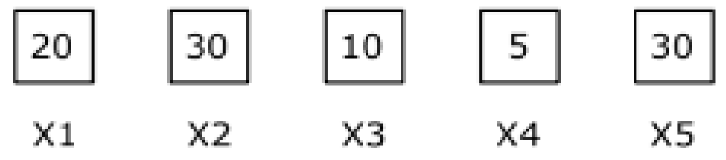
Optimal Merge Patterns

- Given ‘n’ sorted files, there are many ways to pair wise merge them into a single sorted file.
- As, different pairings require different amounts of computing time, we want to determine an optimal ***(i.e., one requiring the fewest comparisons)*** way to pair wise merge ‘n’ sorted files together.
- This type of merging is called as 2-way merge patterns.
- To merge an n-record file and an m-record file requires possibly n + m record moves, the obvious choice is, at each step merge the two smallest files together.
- The two-way merge patterns can be represented by binary merge trees.

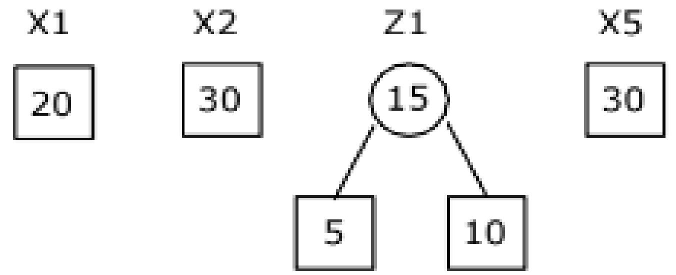
Optimal Merge Patterns

- **Solve:**
- Given five files *(X1, X2, X3, X4, X5)* with sizes *(20, 30, 10, 5, 30)*. Apply greedy rule to find optimal way of pair wise merging to give an optimal solution using binary merge tree representation.

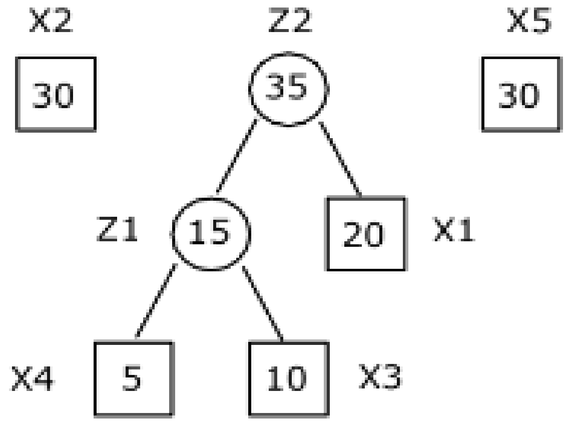
Solution:



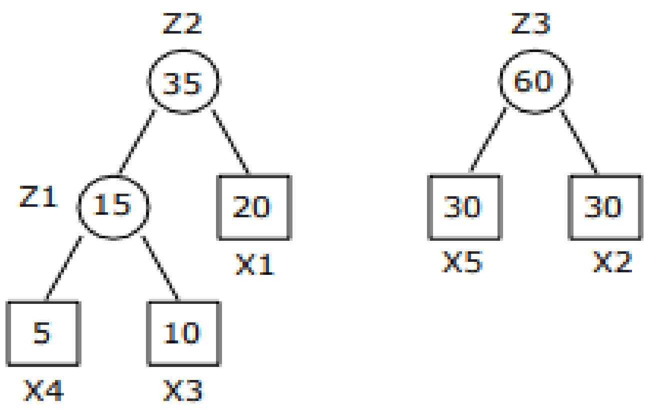
Merge X_4 and X_3 to get 15 record moves. Call this Z_1.



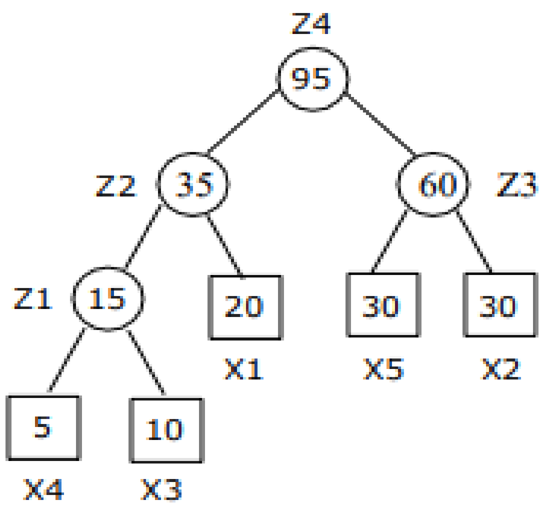
Merge Z_1 and X_1 to get 35 record moves. Call this Z_2.



Merge X_2 and X_5 to get 60 record moves. Call this Z_3.



Merge Z_2 and Z_3 to get 90 record moves. This is the answer. Call this Z_4.



Therefore the total number of record moves is 15 + 35 + 60 + 95 = 205. This is an optimal merge pattern for the given problem.

Optimal Merge Patterns

- **Solve:**
- Consider the sequence {3, 5, 9, 11, 16, 18, 20}. Find optimal merge patter for this data

Optimal Merge Patterns

```
Algorithm: TREE (n)
for i := 1 to n - 1 do
    declare new node
    node.leftchild := least (list) // function returns minimum
element from tree and delete it.
    node.rightchild := least (list)
    node.weight) := ((node.leftchild).weight) +
((node.rightchild).weight)
    insert (list, node);
return least (list);
```

Optimal Storage Allocation on Magnetic Tapes

- Given n programs ***$P_1, P_2, \ldots, P_n$*** of length ***$L_1, L_2, \ldots, L_n$*** respectively, store them on a tape of length L such that ***Mean Retrieval Time (MRT)*** is a minimum.
- The retrieval time of the jth program is a summation of the length of first j programs on tape.
- Let T_j be the time to retrieve program P_j. The retrieval time of P_j is computed as,

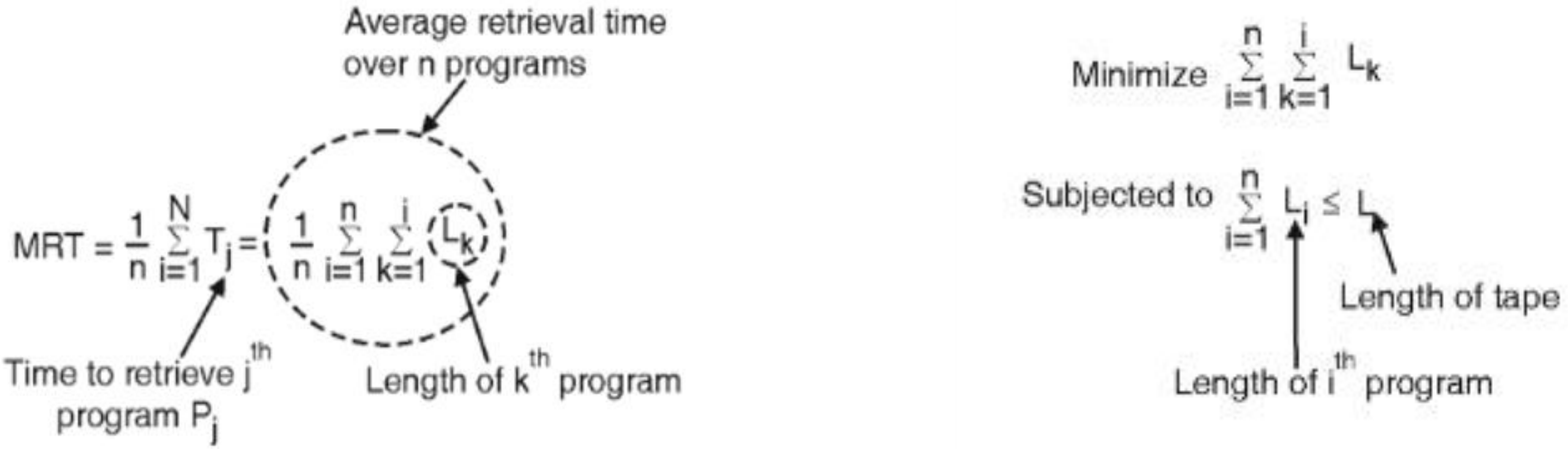
$$T_j = \sum_{k=1}^{j} L_k$$

Length of kth program

Optimal Storage Allocation on Magnetic Tapes

- Mean retrieval time of n programs is the average time required to retrieve any program.
- It is required to store programs in an order such that their Mean Retrieval Time is minimum. MRT is computed as,

Optimal storage on tape is minimization problem which,



$$\text{MRT} = \frac{1}{n}\sum_{i=1}^{N} T_j = \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{i} L_k$$

$$\text{Minimize} \sum_{i=1}^{n}\sum_{k=1}^{i} L_k$$

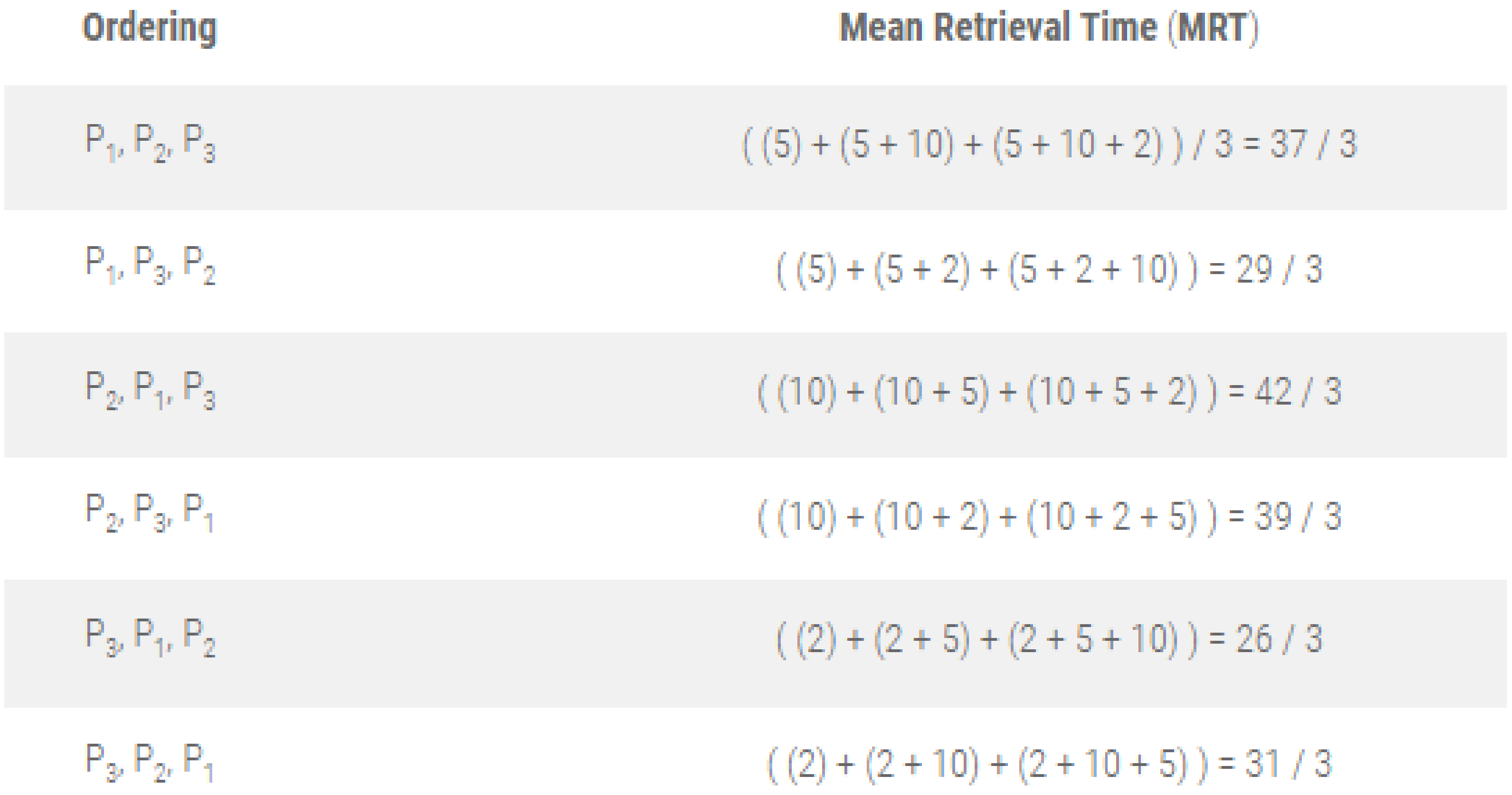
$$\text{Subjected to} \sum_{i=1}^{n} L_i \le L$$

Average retrieval time over n programs
Time to retrieve jth program P_j
Length of kth program
Length of tape
Length of ith program

Optimal Storage Allocation on Magnetic Tapes

- In this case, we have to find the permutation of the program order which minimizes the MRT after storing all programs on *single tape* only.
- There are many permutations of programs. Each gives a different MRT. Consider three programs ***(P1, P2, P3)*** with a length of ***(L1, L2, L3)*** *= (5, 10, 2).*
- Let's find the MRT for different permutations. 6 permutations are possible for 3 items.
- The ***Mean Retrieval Time*** for each permutation is listed in the following table.

Optimal Storage Allocation on Magnetic Tapes



Ordering	Mean Retrieval Time (MRT)
P_1, P_2, P_3	((5) + (5 + 10) + (5 + 10 + 2)) / 3 = 37 / 3
P_1, P_3, P_2	((5) + (5 + 2) + (5 + 2 + 10)) = 29 / 3
P_2, P_1, P_3	((10) + (10 + 5) + (10 + 5 + 2)) = 42 / 3
P_2, P_3, P_1	((10) + (10 + 2) + (10 + 2 + 5)) = 39 / 3
P_3, P_1, P_2	((2) + (2 + 5) + (2 + 5 + 10)) = 26 / 3
P_3, P_2, P_1	((2) + (2 + 10) + (2 + 10 + 5)) = 31 / 3

Optimal Storage Allocation on Magnetic Tapes

- It should be observed from the above table that the MRT is 26/3, which is achieved by storing the programs in **ascending order of their length**.
- Thus, greedy algorithm stores the programs on tape in non-decreasing order of their length, which ensures the minimum MRT.

```
Algorithm MRT_SINGLE_TAPE(L):
  // Description: Find storage order of n programs such that mean retrieval
  time is minimum
  // Input: L is an array of program lengths sorted in ascending order
  // Output: Minimum Mean Retrieval Time
  $T_j \leftarrow 0$   // Initialize total retrieval time
  for $i \leftarrow 1$ to n do
     for $j \leftarrow 1$ to i do
        $T_j \leftarrow T_j + L[j]$   // Accumulate retrieval time for each program
     end for
  end for
  $MRT \leftarrow T_j / n$   // Calculate mean retrieval time
  return MRT
```

Example: Given the program lengths L = {12, 34, 56, 73, 24, 11, 34, 56, 78, 91, 34, 91, 45}. Store them on three taps and minimize MRT

Solution:

Given data :

P_i	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}
L_i	12	34	56	73	24	11	34	56	78	91	34	91	45

First sort the programs in increasing order of their size.

Sorted data:

P_i	P_6	P_1	P_5	P_2	P_7	P_{11}	P_{13}	P_3	P_8	P_4	P_9	P_{10}	P_{12}
L_i	11	12	24	34	34	34	45	56	56	73	78	91	91

Now distribute the files among all three tapes.

Tape 1	P_6	P_2	P_{13}	P_4	P_{12}
Tape 2	P_1	P_7	P_3	P_9	
Tape 3	P_5	P_{11}	P_8	P_{10}	

$MRT_{Tape1} = ((11) + (11 + 34) + (11 + 34 + 45) + ((11 + 34 + 45 + 73)) + (11 + 34 + 45 + 73 + 91)) / 4$

$= 563 / 5$

$= 112.6$

$MRT_{Tape2} = ((12) + (12 + 34) + (12 + 34 + 56) + (12 + 34 + 56 + 78)) / 4$

$= 340 / 4$

$= 85$

$MRT_{Tape3} = ((24) + (24 + 34) + (24 + 34 + 56) + (24 + 34 + 56 + 91)) / 4$

$= 353 / 4$

$= 88.25$

$MRT = (MRT_{Tape3} + MRT_{Tape3} + MRT_{Tape3}) / 3$

$= (112.6 + 85 + 88.25) / 3$

$= 95.28$

Minimum Cost Spanning Tree

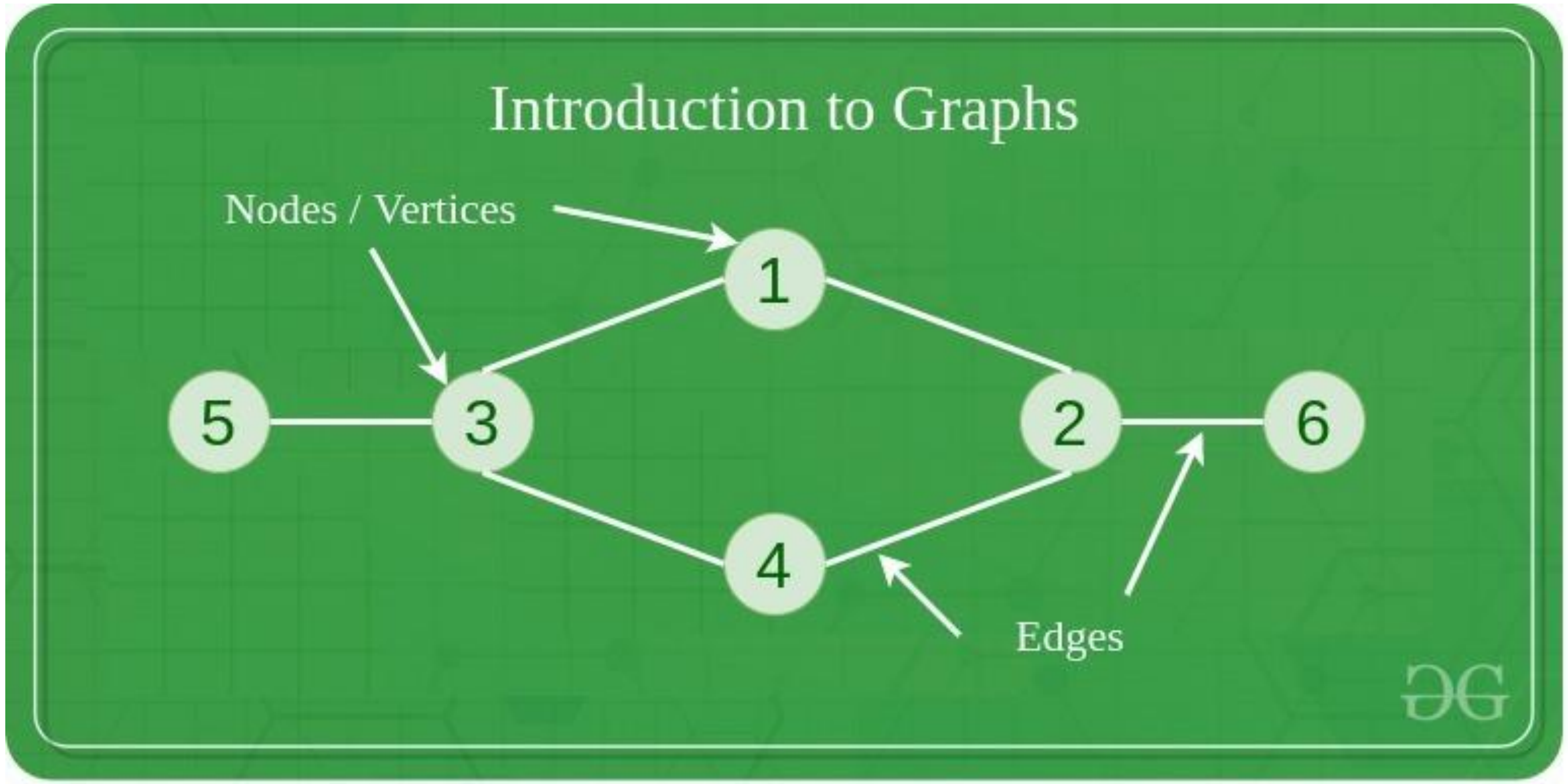
- A ***minimum spanning tree (MST)*** is a tree that spans all the vertices in a connected, undirected graph while minimizing the sum of the weights (costs) of the edges.
- In simpler terms, *it is a subset of the edges of the graph that forms a tree and connects all the vertices with the minimum possible total edge weight.*
- A single graph can have multiple spanning trees.
- A **Minimum Spanning Tree(MST)** or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree having a weight less than or equal to the weight of every other possible spanning tree.
- The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

Minimum Cost Spanning Tree

- Key characteristics of a minimum spanning tree:
    - **Spanning Tree:** It covers all the vertices of the graph without forming any cycles.
    - **Acyclic:** It does not contain any cycles since it is a tree.
    - **Connected:** It connects all the vertices of the original graph.
    - **Minimum Weight:** The sum of the edge weights in the spanning tree is minimized.

Minimum Cost Spanning Tree

- A Graph is a non-linear data structure consisting of vertices and edges. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(E, V).

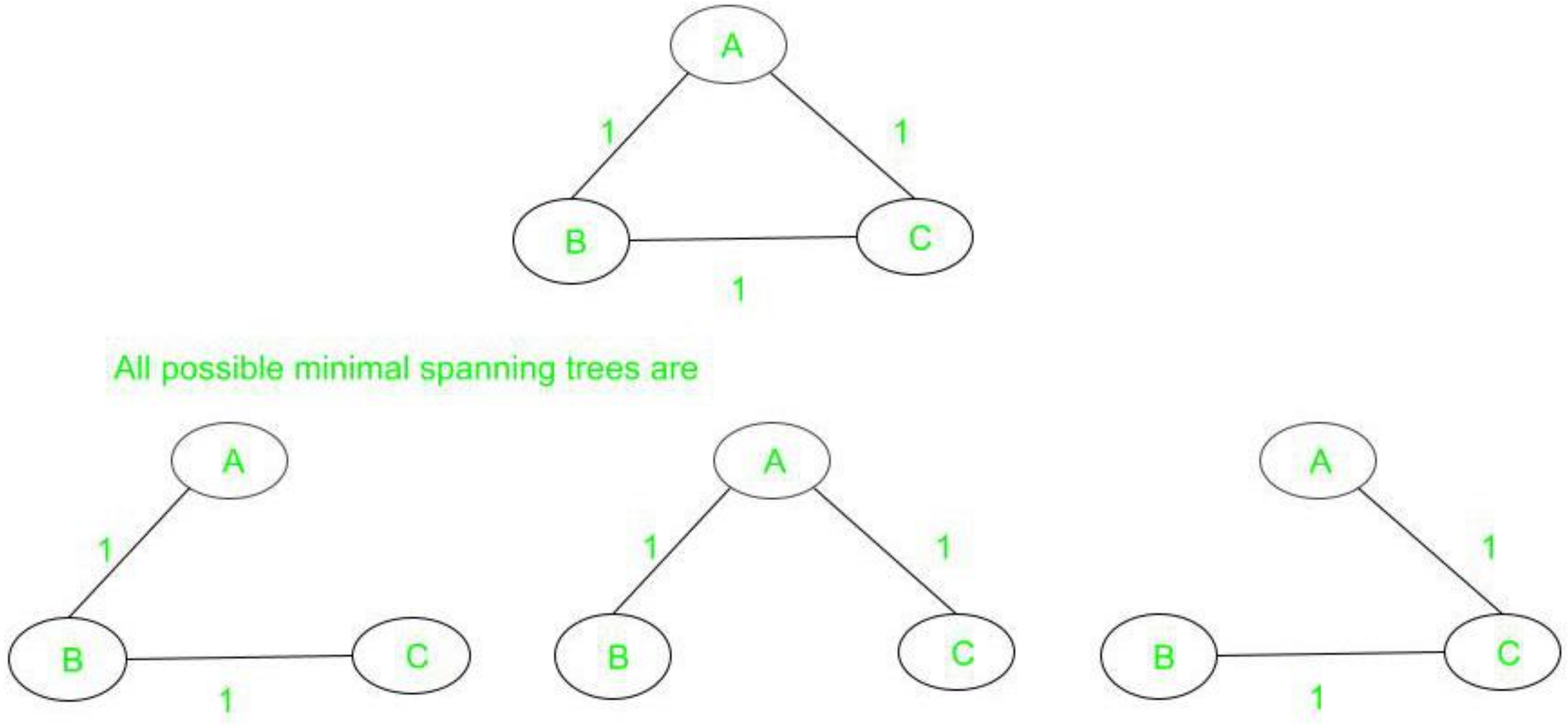


Minimum Cost Spanning Tree

- **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabeled.
- **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labeled/unlabeled.
- Graphs are used to solve many real-life problems. Graphs are used to represent ***networks***. The networks may include paths in a city or telephone network or circuit network.
- *If G(V, E) is a graph then every spanning tree of graph G consists of (V – 1) edges, where V is the number of vertices in the graph and E is the number of edges in the graph.*

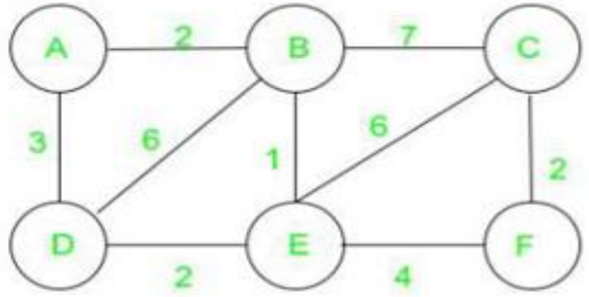
Minimum Cost Spanning Tree

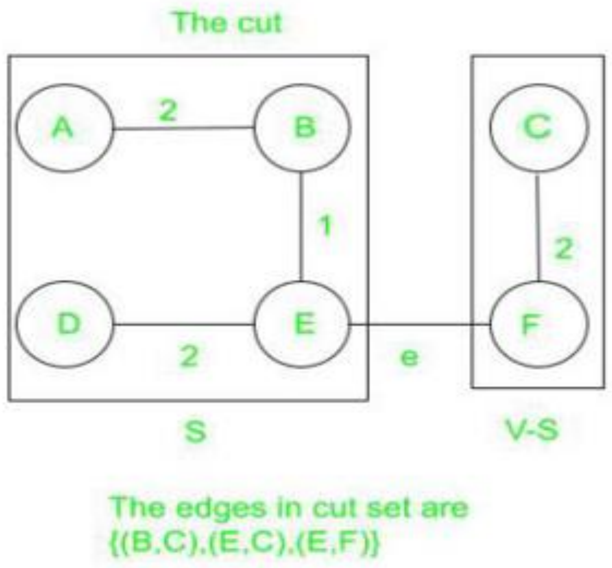
- Consider a complete graph of three vertices and all the edge weights are the same then there will be three spanning trees(which are also minimal) of the same path length are possible.

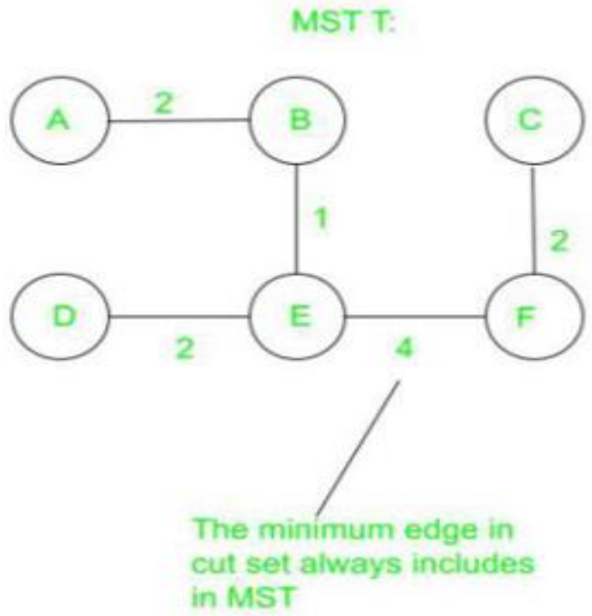


Minimum Cost Spanning Tree

- For any cut C of the graph, if the weight of an edge E in the cut-set of C is strictly smaller than the weights of all other edges of the cut-set of C, then this edge belongs to all the MSTs of the graph.

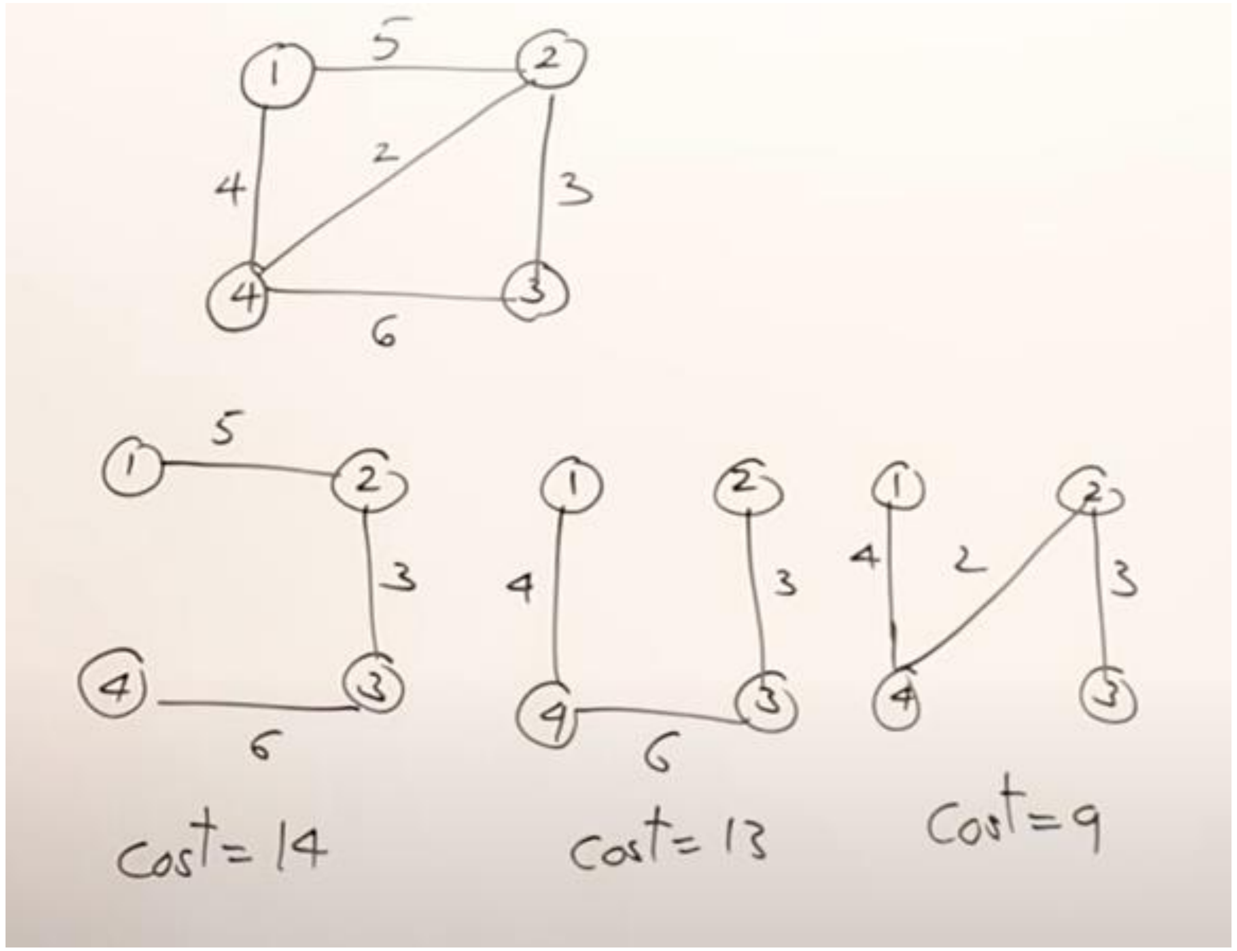






Minimum Cost Spanning Tree

- Consider Example with possible spanning tree



Minimum Cost Spanning Tree

- **Algorithms for finding Minimum Spanning Tree(MST):-**
  - Prim’s Algorithm
  - Kruskal’s Algorithm

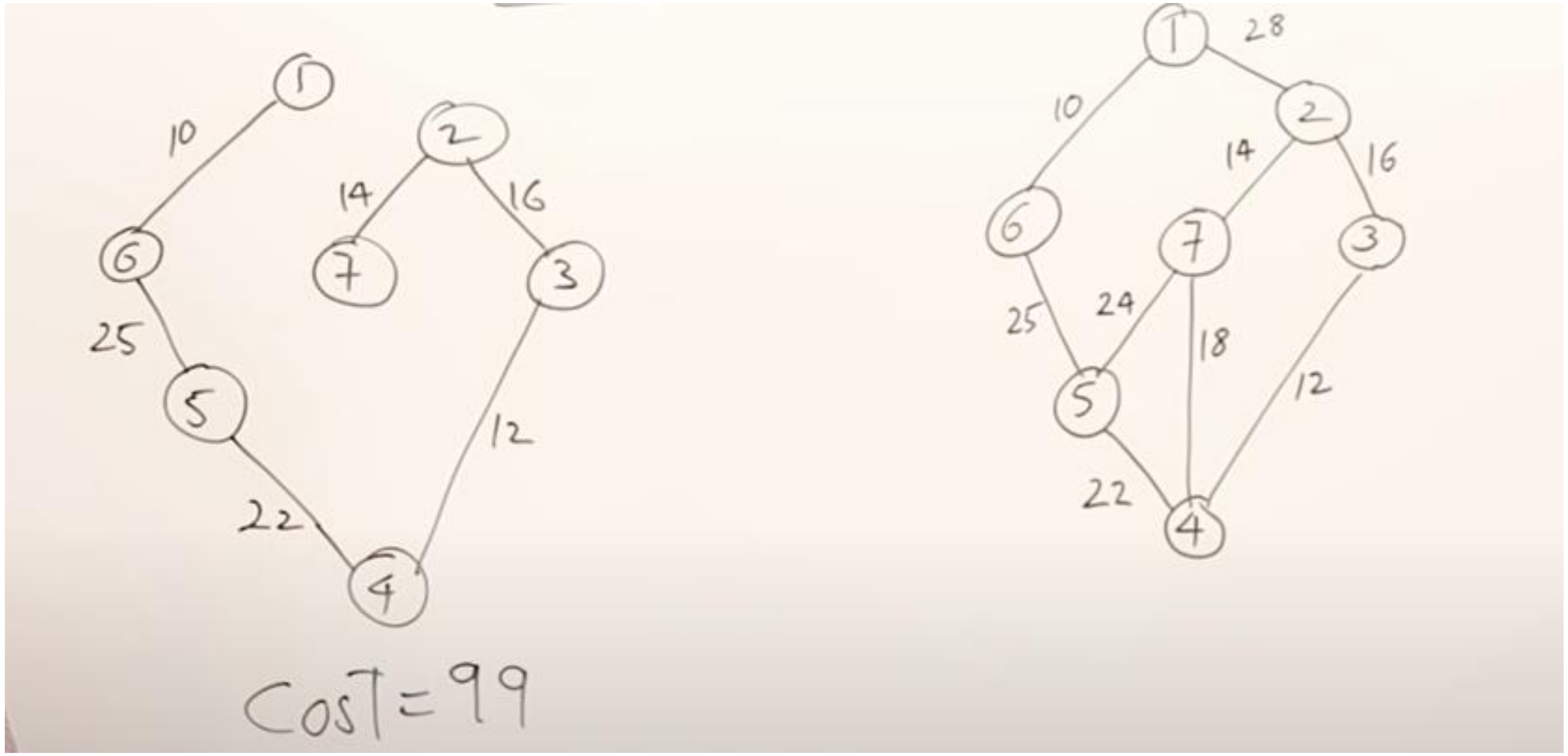
Prim's Algorithm

- It starts with an ***empty spanning tree.*** The idea is to maintain two sets of vertices.
- The first set contains the vertices ***already included*** in the MST, the other set contains the vertices ***not yet included.***
- At every step, it considers all the edges that *connect the two sets and picks the minimum weight edge from these edges.*
- After picking the edge, it moves the other endpoint of the edge to the set containing MST.
- A group of edges that connects two sets of vertices in a graph is called ***cut in graph theory.***
- So, at every step of Prim's algorithm, find a cut (of two sets, one contains the vertices already included in MST and the other contains the rest of the vertices), ***pick the minimum weight edge from the cut,*** and include this vertex to MST Set (the set that contains already included vertices).

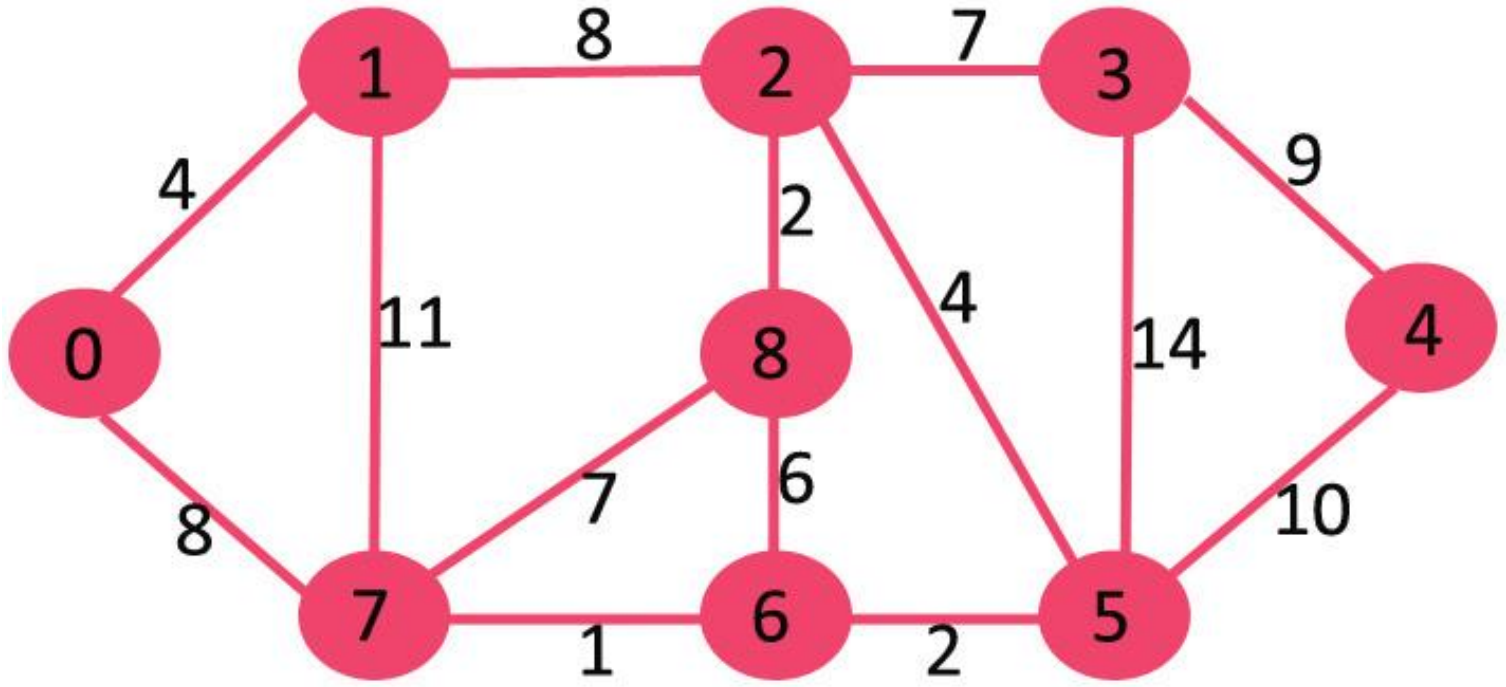
Prim’s Algorithm

```
Prim's Algorithm(Graph G):
    1. Initialize an empty priority queue Q.
    2. Add an arbitrary starting vertex to the priority queue.
    3. While Q is not empty:
        a. Extract the edge with the smallest weight from Q.
        b. If adding this edge doesn't form a cycle, add it to the minimum spanning tree.
        c. Add the vertex at the other end of the chosen edge to the priority queue if not already present.
    4. Return the minimum spanning tree.
```

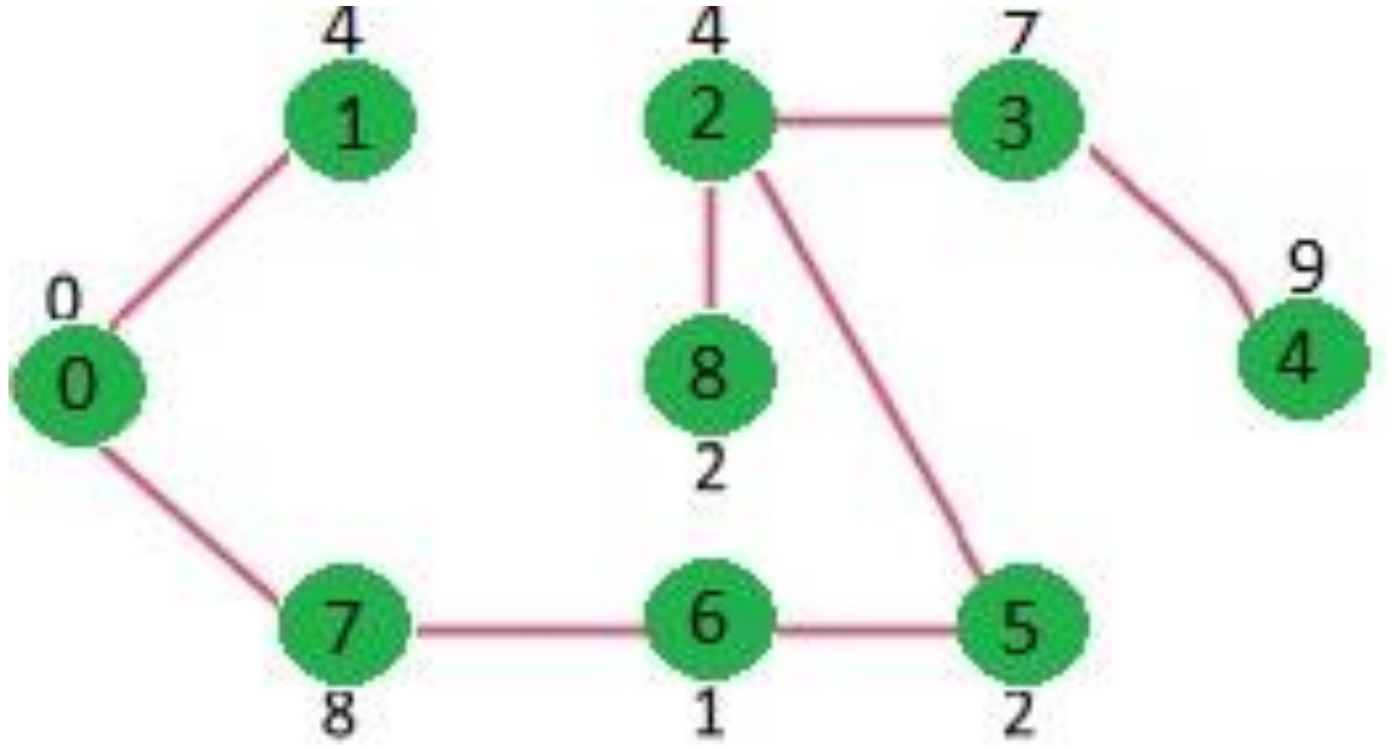
Prim’s Algorithm



Prim’s Algorithm (Start with minimum edge and then start adding connected smallest)



Prim’s Algorithm

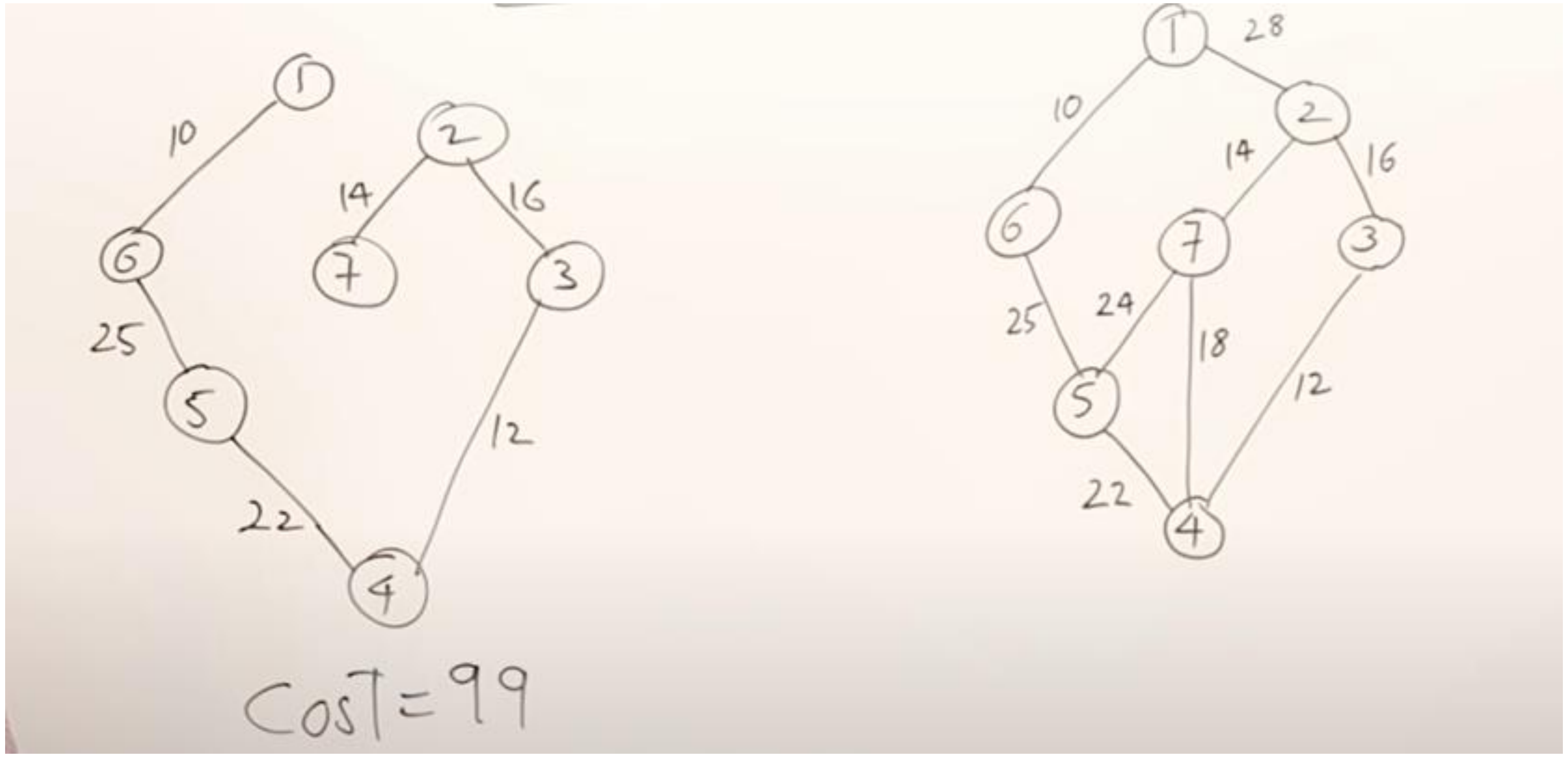


Kruskal’s Algorithm

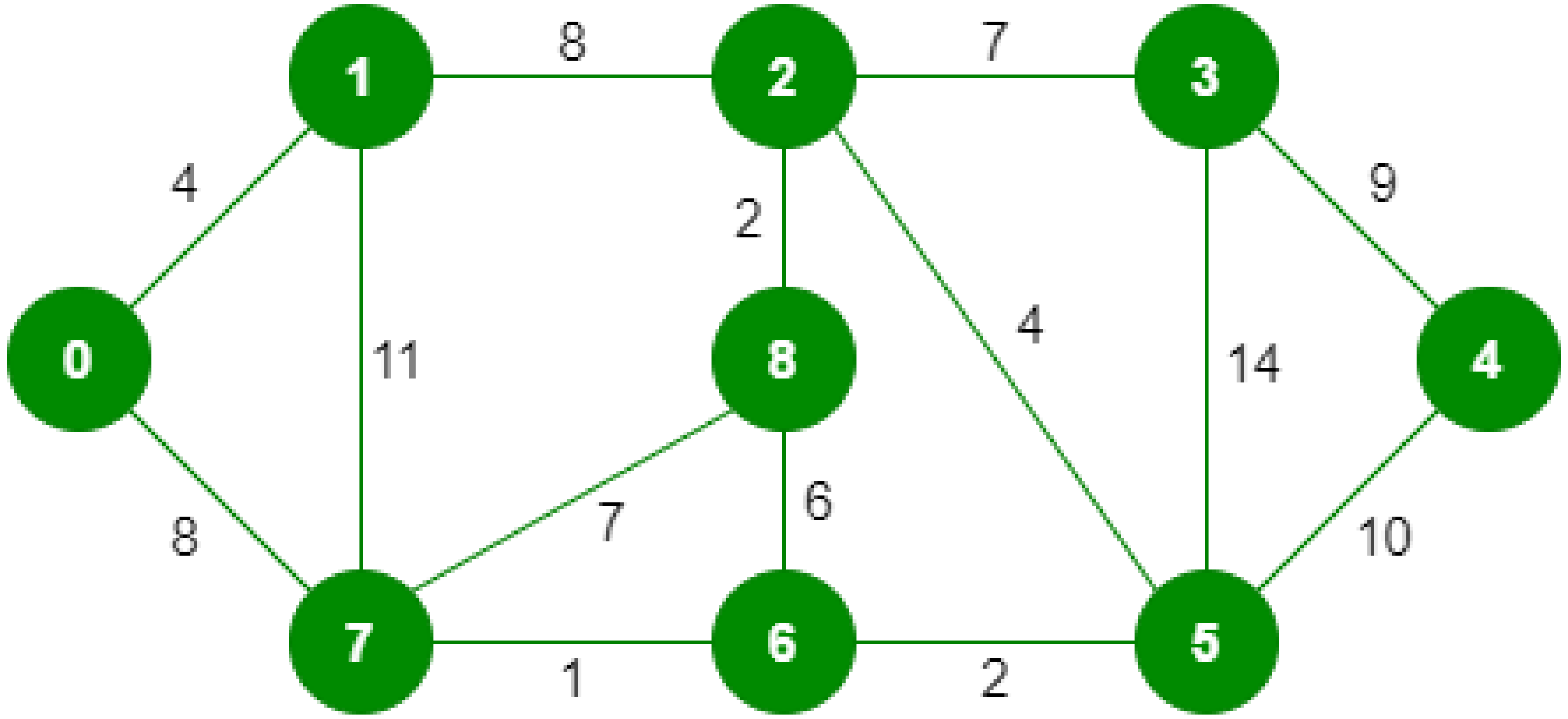
Kruskal's Algorithm(Graph G):

```
1. Sort all edges in non-decreasing order of their weights.
2. Initialize an empty minimum spanning tree.
3. Initialize a data structure to keep track of connected components.
4. For each edge in the sorted order:
    a. If adding the edge does not create a cycle, add it to the minimum
spanning tree and merge the connected components.
5. Return the minimum spanning tree.
```

Kruskal's Algorithm



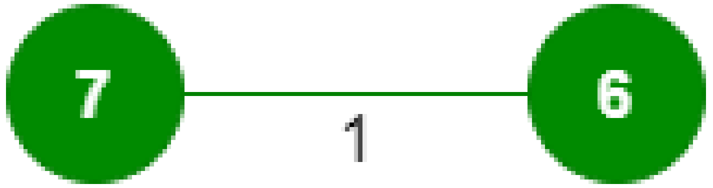
Kruskal’s Algorithm

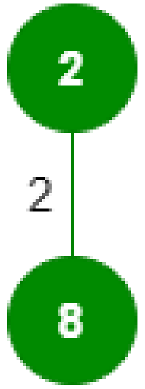


```
After sorting:
Weight   Src    Dest
1         7      6
2         8      2
2         6      5
4         0      1
4         2      5
6         8      6
7         2      3
7         7      8
8         0      7
8         1      2
9         3      4
10        5      4
11        1      7
14        3      5
```

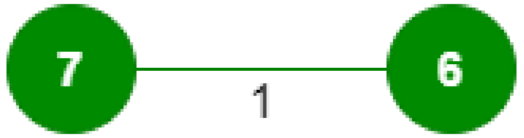
Kruskal’s Algorithm

- *Now pick all edges one by one from the sorted list of edges*
  Step 1: *Pick edge 7-6: No cycle is formed, include it.*



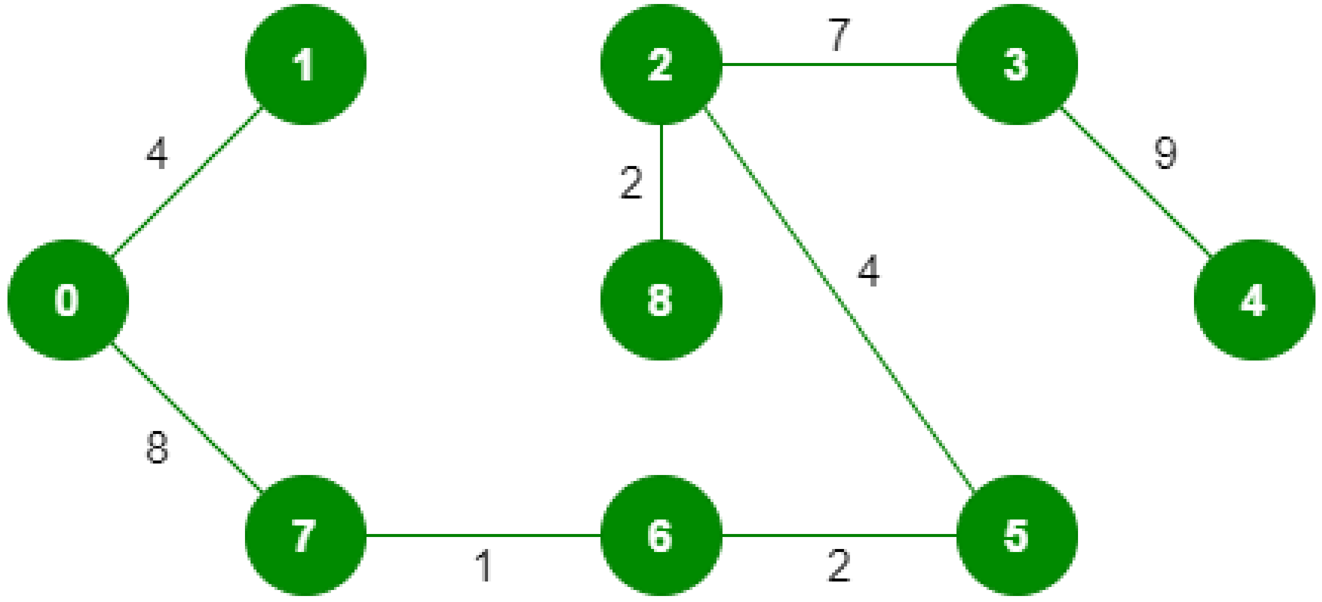


- ***Step 2:*** *Pick edge 8-2: No cycle is formed, include it.*
- ***Step 3:*** *Pick edge 6-5: No cycle is formed, include it.*
- ***Step 4:*** *Pick edge 0-1: No cycle is formed, include it.*
- ***Step 5:*** *Pick edge 2-5: No cycle is formed, include it.*
- ***Step 6:*** *Pick edge 8-6: Since including this edge results in the cycle, discard*
- ***Step 7:*** *Pick edge 2-3: No cycle is formed, include it.*



Kruskal’s Algorithm

- ***Step 8:*** *Pick edge 7-8: Since including this edge results in the cycle, discard.*
- ***Step 9:*** *Pick edge 0-7: No cycle is formed, include it.*
- ***Step 10:*** *Pick edge 1-2: Since including this edge results in the cycle, discard it.*
- ***Step 11:*** *Pick edge 3-4: No cycle is formed, include it.*
- ***Note:*** *Since the number of edges included in the MST equals to (V – 1), so the algorithm stops here.*



Greedy Strategy	Prim's algorithm is Vertex-centric	Kruskal's algorithm is Edge-centric
Data Structures	Priority queue (or min-heap)	Disjoint set data structure (union-find)
Edge/Vertex Selection	Selects edge with minimum weight connecting a vertex in the current tree to a vertex outside the tree	Selects edge with minimum weight from the sorted list of edges without a predetermined starting point
Complexity	O(E log V), where E is the number of edges, V is the number of vertices	O(E log E), where E is the number of edges
Starting Point	Starts with a single vertex and grows the tree	Starts with an empty tree and adds edges
Applications	Suitable for dense graphs or graphs with dense clusters	Suitable for sparse graphs or graphs with edges of similar weights
Termination	Terminates when all vertices are included in the minimum spanning tree	Terminates when all vertices are included in the minimum spanning tree

Tree Vertex Splitting

- Directed and weighted binary tree
- Consider a network of power line transmission
- The transmission of power from one node to the other results in some loss, such as drop in voltage
- Each edge is labeled with the loss that occurs (edge weight)
- Network may not be able to tolerate losses beyond a certain level
- You can place boosters in the nodes to account for the losses
- Definition 1 Given a network and a loss tolerance level, the tree vertex splitting problem is to determine the optimal placement of boosters.
- You can place boosters only in the vertices and nowhere else

Tree Vertex Splitting

- The tree vertex splitting problem involves optimizing the placement of boosters in a directed and weighted binary tree, representing a power transmission network.
- The objective is to minimize losses, which are associated with the edges (transmission lines) in the tree.
- **Network Description:**
  - The network is represented as a directed and weighted binary tree.
  - Nodes in the tree correspond to locations in the power transmission network.
  - Edges in the tree are labeled with weights that represent the losses occurring during the transmission between nodes.
- **Loss Tolerance Level:**
  - There is a specified loss tolerance level that the network can withstand.
  - The total loss along a path from the source to any node should not exceed this tolerance level.

Tree Vertex Splitting

- **Objective:**
  - The goal is to determine the optimal placement of boosters in the vertices (nodes) of the tree.
  - Boosters are placed to compensate for losses and ensure that the total loss along any path does not exceed the given tolerance level.
- **Constraints:**
  - Boosters can only be placed in the vertices of the tree, and not along the edges.
- **Optimization:**
  - The optimization problem involves finding the best locations for boosters to minimize losses in the network.

Tree Vertex Splitting

- Let T = (V, E, w) be a weighted directed tree
    - V is the set of vertices
    - E is the set of edges
    - w is the weight function for the edges
    - w_{ij} is the weight of the edge hi, ji ∈ E
    - We say that $w_{ij} = \infty$ if <i, j> !∈ E
    - A vertex with in-degree zero is called a source vertex
    - A vertex with out-degree zero is called a sink vertex
    - For any path P ∈ T, its delay d(P) is defined to be the sum of the weights ***(w_{ij})*** of that path, or

$$d(P) = \sum_{\langle i,j\rangle \in P} w_{ij}$$

Tree Vertex Splitting

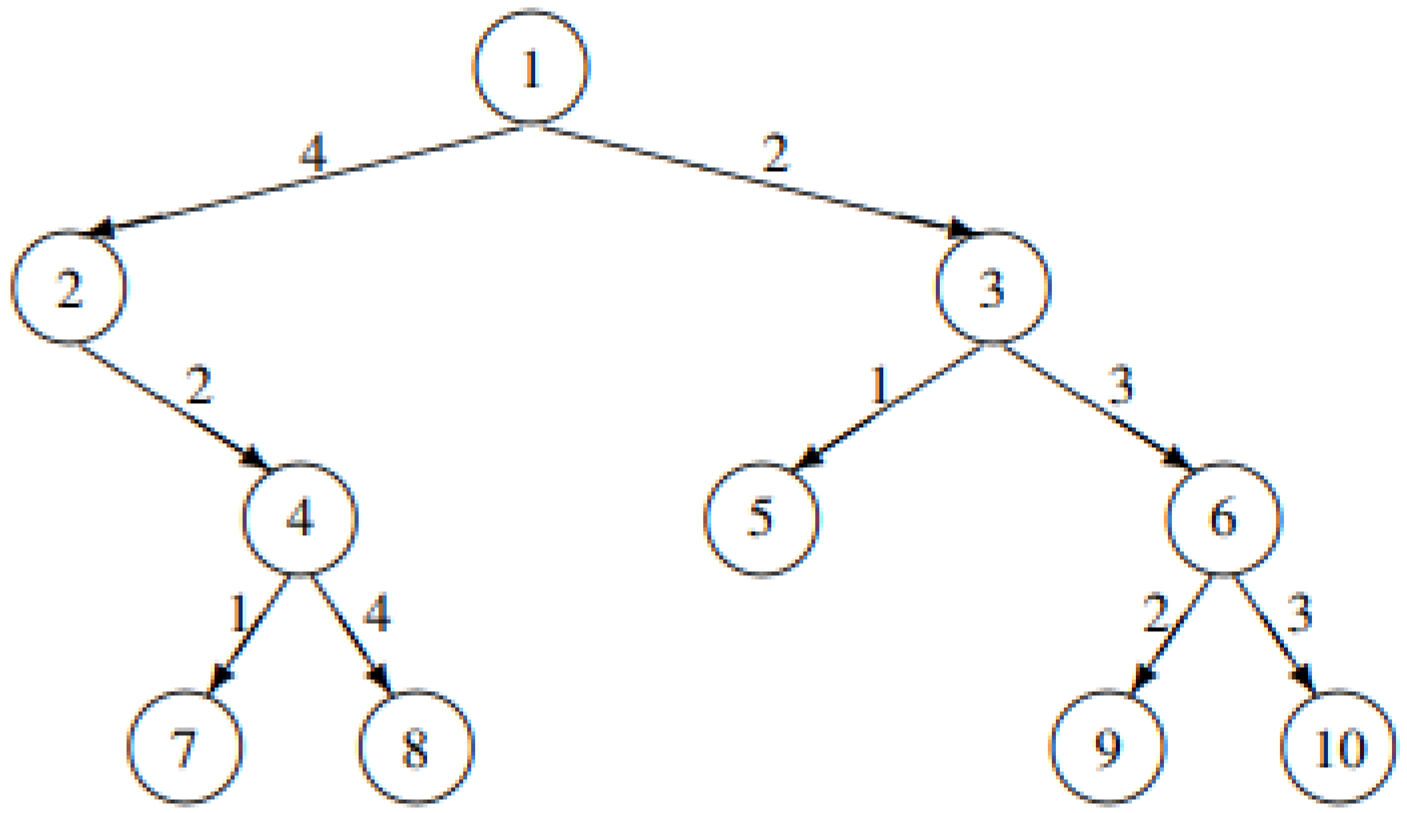
- Delay of the tree T, d(T) is the maximum of all path delays.
- Split node is the booster station
- Tree vertex splitting problem is to identify a set $X \subseteq V$ of minimum cardinality (minimum number of booster stations) for which $d(T/X) \leq \delta$ for some specified tolerance limit δ
- TVSP has a solution only if the maximum edge weight is $\leq \delta$

Greedy solution for TVSP

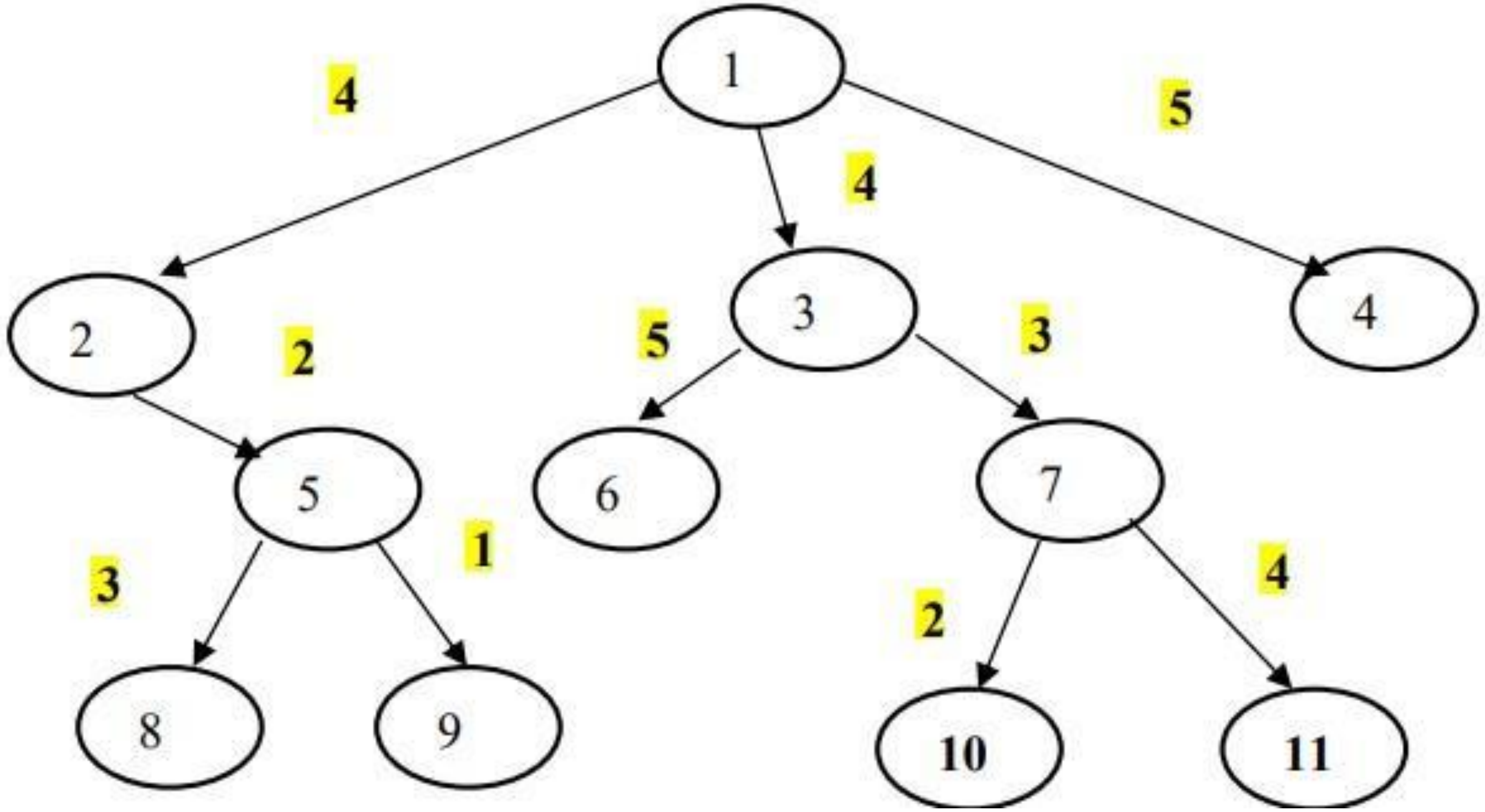
- We want to minimize the number of booster stations (X)
- For each node u ∈ V , compute the maximum delay d(u) from u to any other node in its subtree
- If u has a parent v such that ***d(u) + w(v, u) > δ***, split u and set d(u) to zero
- Computation proceeds from leaves to root
- Delay for each leaf node is zero
- The delay for each node v is computed from the delay for the set of its children C(v)

$$d(v) = \max_{u \in C(v)} \{d(u) + w(v, u)\}$$

Solve tree for δ = 5



For the given network and a loss tolerance level δ=10, determine the optimal placement of boosters in the network using the greedy algorithm for tree vertex splitting .



Thank You