Design and Analysis of Algorithms

A Course Material for
CSC 551

S. O. AKINOLA (PhD)

General Introduction and Course Objective

CSC 551 (Design and Analysis of Algorithms) is a three [3] credit unit course dealing with the fundamentals concepts of Algorithm designing and Analysis techniques. The study material provides adequate background information that is relevant for students to understand the concept of algorithms' design and analysis. The course is a continuation of CSC 236, which is on Introduction to Algorithms. .

Course Curriculum Contents

CSC 551 Design and Analysis of Algorithms

Asymptotic Notation. Recurrence Equations. Basic Algorithms and Analysis. Best, Worst and Average case analysis. Algorithm Design techniques: Divide and Conquer, Dynamic Programming and Greedy Algorithms, Randomized Algorithms. Amortized Analysis. Graph Algorithms. NP-completeness. Approximation algorithms. Laboratory Exercises

Table of Contents

Study Session 1: Fundamentals of Algorithms

Expected Duration: 1 week or 2 contact hours

Introduction
In this Session, learners will be refreshed with what they learned in CSC 236 (Introduction to Algorithms). Basic concepts, categories and types of algorithms as well as running time analysis are explained

Learning Outcomes
When you have studied this session, you should be able to explain:

1.1 Definition of Algorithm

An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions.

In addition, every algorithm must satisfy the following criteria:

- **Input**: there are zero or more quantities, which are externally supplied;
- **Output**: at least one quantity is produced

- **Definiteness**: each instruction must be clear and unambiguous

- **Finiteness**: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of step

- **Effectiveness**: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper.

1.2 Performance of Algorithms

The performance of an algorithm and / or program is the amount of computer memory and time needed to run the algorithm or program. Therefore, we have two measures of performance of algorithms:

- Time Complexity
- Space Complexity

(i) Time Complexity

The time needed by an algorithm expressed as a function of the size of a problem is called the time complexity of the algorithm. The time complexity of an algorithm is the amount of computer time it needs to run to completion. ***The limiting behaviour of the complexity as size increases is called the asymptotic time complexity***. It is the asymptotic complexity of an algorithm which ultimately determines the size of problems that can be solved by the algorithm.

(ii) Space Complexity

The space complexity of an algorithm and / or a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

(1) Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.
- The compiler used to compile the program into machine code.
- The target computer.

(2) Data space: Data space is the space needed to store all constant and variable values. Data space has two components:
- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and class instances.

(3) Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

1.3 Goals of Algorithm Design

The three basic design goals that one should strive for in an algorithm are:

(i) **Trying to save Time**: A program that runs faster is a better program, so saving time is an obvious goal.

(ii) **Trying to save Space**: A program that saves space over a competing program is considered desirable.

(iii) **Trying to save Face**: By preventing the program from locking up or generating reams of garbled (distorted or muddled) data.

1.4 Classification of Algorithms

Algorithms can be classified in various ways. They are:

1. Implementation Method
2. Design Method
3. Design Approaches
4. Other Classifications

The classification of algorithms is important for several reasons:

(i) **Organization:** Algorithms can be very complex and by classifying them, it becomes easier to organize, understand, and compare different algorithms.

(ii) **Problem Solving:** Different problems require different algorithms, and by having a classification, it can help identify the best algorithm for a particular problem.

(iii) **Performance Comparison:** By classifying algorithms, it is possible to compare their performance in terms of time and space complexity, making it easier to choose the best algorithm for a particular use case.

(iv) **Reusability:** By classifying algorithms, it becomes easier to re-use existing algorithms for similar problems, thereby reducing development time and improving efficiency.

(v) **Research:** Classifying algorithms is essential for research and development in computer science, as it helps to identify new algorithms and improve existing ones.

Overall, the classification of algorithms plays a crucial role in computer science and helps to improve the efficiency and effectiveness of solving problems.

1.4.1 Classification by Implementation Method

There are primarily three main categories into which an algorithm can be named in this type of classification. They are:

1. **Recursion or Iteration:** A recursive algorithm is an algorithm which calls itself again and again until a base condition is achieved whereas iterative algorithms use loops and/or data structures like stacks, queues to solve any problem. Every recursive solution can be implemented as an iterative solution and vice versa.

   Example: The Tower of Hanoi is implemented in a recursive fashion while Stock Span problem is implemented iteratively.

2. **Exact or Approximate:** Algorithms that are capable of finding an optimal solution for any problem are known as the exact algorithm. For all those problems, where it is not possible to find the most optimized solution, an approximation algorithm is used. Approximate algorithms are the type of algorithms that find the result as an average outcome of sub outcomes to a problem.

   Example: For NP-Hard Problems, approximation algorithms are used. Sorting algorithms are the exact algorithms.

3. **Serial or Parallel or Distributed Algorithms:** In serial algorithms, one instruction is executed at a time while parallel algorithms are those in which we divide the problem into subproblems and execute them on different processors. If parallel algorithms are distributed on different machines, then they are known as distributed algorithms.

1.4.2 Classification by Design Method

There are primarily seven main categories into which an algorithm can be named in this type of classification. They are:

1. **Greedy Method:** In the greedy method, at each step, a decision is made to choose the *local optimum*, without thinking about the future consequences.
   Example: Fractional Knapsack, Activity Selection.

2. **Divide and Conquer:** The Divide and Conquer strategy involves dividing the problem into sub-problem, recursively solving them, and then recombining them for the final answer.
   Example: Merge sort, Quicksort.

3. **Dynamic Programming:** The approach of Dynamic programming is similar to divide and conquer. The difference is that whenever we have recursive function calls with the same result, instead of calling them again we try to store the result in a data structure in the form of a table and retrieve the results from the table. Thus, the overall time complexity is reduced. “Dynamic” means we dynamically decide, whether to call a function or retrieve values from the table.

   Example: 0-1 Knapsack, subset-sum problem.

4. **Linear Programming:** In Linear Programming, there are inequalities in terms of inputs and maximizing or minimizing some linear functions of inputs.

   Example: Maximum flow of Directed Graph

5. **Reduction (Transform and Conquer):** In this method, we solve a difficult problem by transforming it into a known problem for which we have an optimal solution. Basically, the goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithms.

   Example: Selection algorithm for finding the median in a list involves first sorting the list and then finding out the middle element in the sorted list. These techniques are also called *transform and conquer.*

6. **Backtracking:** This technique is very useful in solving combinatorial problems that have *a single unique solution.* Where we have to find the correct combination of steps that lead to fulfillment of the task. Such problems have multiple stages and there are multiple options at each stage. This approach is based on exploring each available option at every stage one-by-one. While exploring an option if a point is reached that doesn't seem to lead to the solution, the program control backtracks one step, and starts exploring the next option. In this way, the program explores all possible course of actions and finds the route that leads to the solution.

   Example: N-queen problem, maize problem.

7. **Branch and Bound:** This technique is very useful in solving combinatorial optimization problem that have *multiple solutions* and we are interested in find the most optimum solution. In this approach, the entire solution space is represented in the form of a state space tree. As the program progresses each state combination is explored, and the previous solution is replaced by new one if it is not the optimal than the current solution.

   Example: Job sequencing, Travelling Salesman problem.

1.4.3 Classification by Design Approaches

There are two approaches for designing an algorithm. These approaches include:

(i) Top-Down Approach

In the top-down approach, a large problem is divided into small sub-problems, and keep repeating the process of decomposing problems until the complex problem is solved. It involves breaking down a complex problem into smaller, more manageable sub-problems and solving each sub-problem individually. Designing a system starting from the highest level of abstraction and moving towards the lower levels.

(ii) Bottom-Up Approach

The bottom-up approach is also known as the reverse of top-down approaches. In approach different, part of a complex program is solved using a programming language and then this is combined into a complete program. It involves building a system by starting with the

individual components and gradually integrating them to form a larger system. Solving sub-problems first and then using the solutions to build up to a solution of a larger problem.

Note: Both approaches have their own advantages and disadvantages and the choice between them often depends on the specific problem being solved.

Here are examples of the Top-Down and Bottom-Up approaches in code:

Top-Down Approach (in Python):

```
def solve_problem(problem):
    if problem == "simple":
        return "solved"
    elif problem == "complex":
        sub_problems = break_down_complex_problem(problem)
        sub_solutions = [solve_problem(sub) for sub in sub_problems]
        return combine_sub_solutions(sub_solutions)
```

Bottom-Up Approach (in Python):

```
def solve_sub_problems(sub_problems):
    return [solve_sub_problem(sub) for sub in sub_problems]

def combine_sub_solutions(sub_solutions):
    # implementation to combine sub-solutions to solve the larger problem

def solve_problem(problem):
    if problem == "simple":
        return "solved"
    elif problem == "complex":
        sub_problems = break_down_complex_problem(problem)
        sub_solutions = solve_sub_problems(sub_problems)
        return combine_sub_solutions(sub_solutions)
```

1.4.4 Other Classifications

Apart from classifying the algorithms into the above broad categories, the algorithm can be classified into other broad categories like:

1. **Randomized Algorithms:** Algorithms that make random choices for faster solutions are known as randomized algorithms.

   Example: Randomized Quicksort Algorithm.

2. **Classification by complexity:** Algorithms that are classified on the basis of time taken to get a solution to any problem for input size. This analysis is known as time complexity analysis.

   Example: Some algorithms take O(n), while some take exponential time.

3. **Classification by Research Area:** In Computer Science CS, each field has its own problems and needs efficient algorithms.

   Example: Sorting Algorithm, Searching Algorithm, Machine Learning etc.

4. **Branch and Bound Enumeration and Backtracking:** These are mostly used in Artificial Intelligence.

The following classification is by Complexity:

(1) **1 (Constant/Best case):** Each instruction of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is a constant, order of 1 [$\Theta(1)$].

(2) **Log n (Logarithmic/Divide ignore part):** When the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction. When n is a million, log n is a doubled. Whenever n doubles, log n increases by a constant, but log n does not double until n increases to n^2.

(3) **n (Linear/Examine each):** When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs.

(4) **n log n (Linear logarithmic/Divide use all parts)**: This running time arises for algorithms that solve a problem by breaking it up into smaller sub-problems, solving then independently, and then combining the solutions. When n doubles, the running time more than doubles.

(5) **n^2 (Quadratic/Nested loops):** When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases four-fold.

(6) **n^3 (Cubic/Nested loops):** Similarly, an algorithm that process triples of data items (perhaps in a triple–nested loop) has a cubic running time and is practical for use only on small problems. Whenever n doubles, the running time increases eight-fold.

(7) **2^n (Exponential/All subsets):** Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as "brute–force" solutions to problems. Whenever n doubles, the running time squares.

1.5 Types of Algorithms:

There are several types of algorithms available. Some important algorithms are:

1. Brute Force Algorithm: It is the simplest approach for a problem. A brute force algorithm is the first approach that comes to finding when we see a problem.

2. Recursive Algorithm: A recursive algorithm is based on recursion. In this case, a problem is broken into several sub-parts and called the same function again and again.

3. Backtracking Algorithm: The backtracking algorithm basically builds the solution by searching among all possible solutions. Using this algorithm, we keep on building the solution following criteria. Whenever a solution fails we trace back to the failure point and build on the next solution and continue this process till we find the solution or all possible solutions are looked after.

4. Searching Algorithm: Searching algorithms are the ones that are used for searching elements or groups of elements from a particular data structure. They can be of different types based on their approach or the data structure in which the element should be found.

5. Sorting Algorithm: Sorting is arranging a group of data in a particular manner according to the requirement. The algorithms which help in performing this function are called sorting algorithms. Generally sorting algorithms are used to sort groups of data in an increasing or decreasing manner.

6. Hashing Algorithm: Hashing algorithms work similarly to the searching algorithm. But they contain an index with a key ID. In hashing, a key is assigned to specific data.

7. Divide and Conquer Algorithm: This algorithm breaks a problem into sub-problems, solves a single sub-problem and merges the solutions together to get the final solution. It consists of the following three steps:

- Divide
- Solve
- Combine

8. Greedy Algorithm: In this type of algorithm the solution is built part by part. The solution of the next part is built based on the immediate benefit of the next part. The one solution giving the most benefit will be chosen as the solution for the next part.

9. Dynamic Programming Algorithm: This algorithm uses the concept of using the already found solution to avoid repetitive calculation of the same part of the problem. It divides the problem into smaller overlapping subproblems and solves them.

10. Randomized Algorithm: In the randomized algorithm we use a random number so it gives immediate benefit. The random number helps in deciding the expected outcome.

1.6 Complexity of Algorithms

The complexity of an algorithm M is the function f (n) which gives the running time and/or storage space requirement of the algorithm in terms of the size “n” of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size “n”. However, complexity usually refers to the running time of the algorithm.

The function f (n), gives the running time of an algorithm, depends not only on the size "n" of the input data but also on the data. The complexity function f(n) for certain cases are:

- **Best Case**: The minimum possible value of f(n) is called the best case.
- **Average Case**: The expected value of f(n).
- **Worst Case**: The maximum value of f(n) for any key possible input.

1.7 Rate of Growth

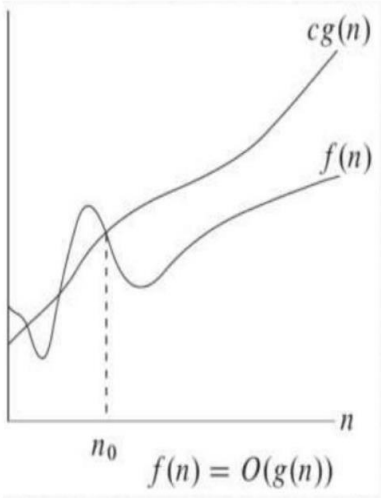
How the time complexity increases as data size increases is the main concern of Rate of Growth. The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

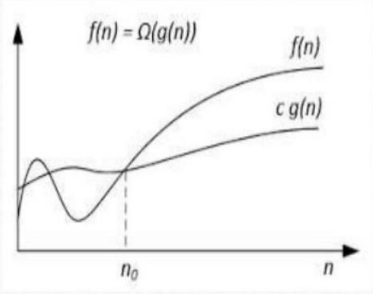
(i) Big–OH (O) (Upper Bound): The growth rate of f(n) is less than or equal (≤) that of g(n).

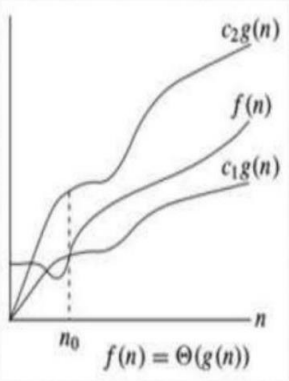
(ii) Big–THETA (Θ) (Same Order): The growth rate of f(n) equals (=) the growth rate of g(n).

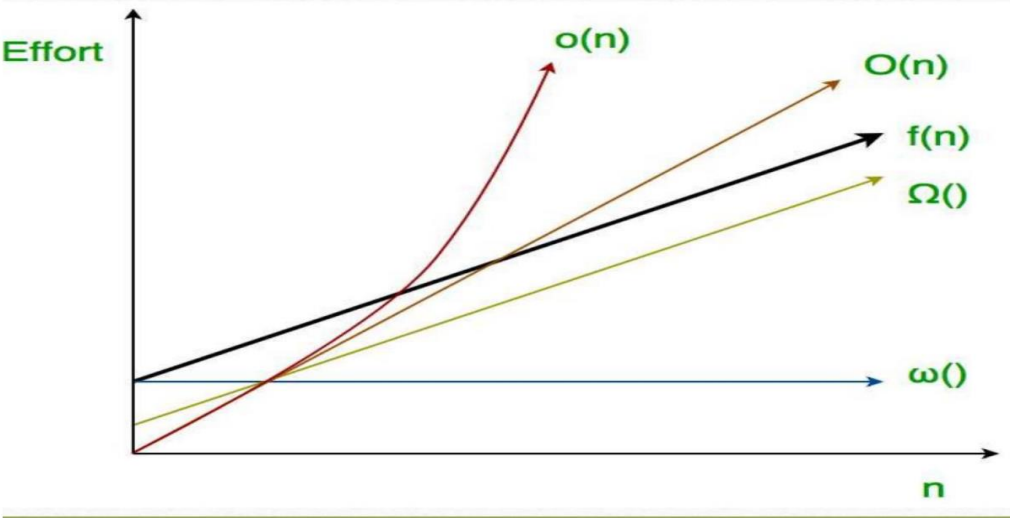
(iii) Little–OH (o):
$\Sigma_{n\to\infty}(n)/(n) = 0$
The growth rate of f(n) is less than that of g(n).

(iv) Little-OMEGA (ω): The growth rate of f(n) is greater than that of g(n).









From, the above chat, which of the functions performs best, and which performs worst?

Examples of the functions:

Examples of functions in $O(n^2)$:

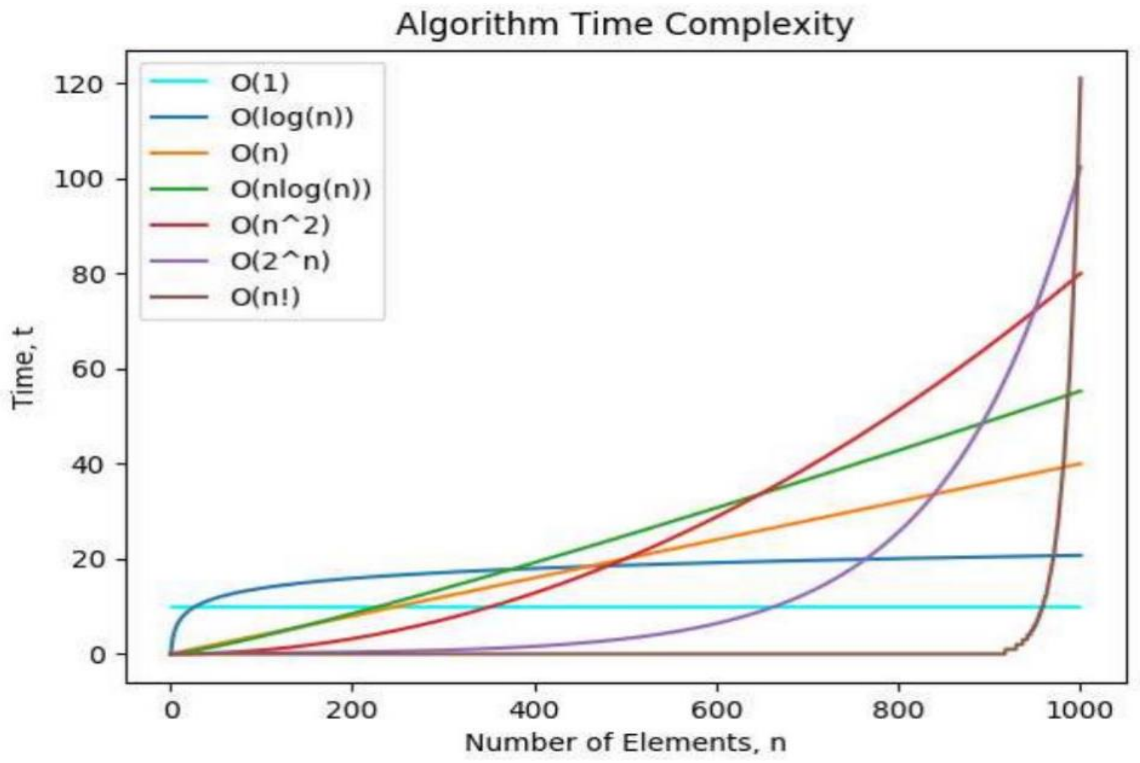
n^2
$n^2 + n$
$n^2 + 1000n$
$1000n^2 + 1000n$
Also,
n
$n/1000$
$n^{1.99999}$
$n^2/\lg\lg\lg n$

Examples of functions in $\Omega(n^2)$:

n^2
$n^2 + n$
$n^2 - n$
$1000n^2 + 1000n$
$1000n^2 - 1000n$
Also,
n^3
$n^{2.00001}$
$n^2 \lg\lg\lg n$
2^{2^n}

Sample Comparative Analysis:

n	log n	n*logn	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296
64	6	384	4096	2,62,144	????????
128	7	896	16,384	2,097,152	????????
256	8	2048	65,536	1,677,216	????????



1.8 Running Time of an Algorithm

The running time of a program depends on factors such as:

(i) The input to the program.
(ii) The quality of code generated by the compiler used to create the object program.

(iii) The nature and speed of the instructions on the machine used to execute the program, and

(iv) The time complexity of the algorithm underlying the program.

1.9 Asymptotic Analysis of Algorithms

This approach is based on the asymptotic complexity measure. This means that we don't try to count the exact number of steps of a program, but how that number grows with the size of the input to the program. That gives us a measure that will work for different operating systems, compilers and CPUs. The asymptotic complexity is written using big-O notation.

1.9.1 Rules for Using Big O

The most important property is that big-O gives an upper bound only. If an algorithm is $O(n^2)$, it doesn't have to take n^2 steps (or a constant multiple of n^2). But it can't take more than n^2. So, any algorithm that is O(n), is also an $O(n^2)$ algorithm. If this seems confusing, think of big-O as being like "<". Any number that is < n is also < n^2.

(i) Ignoring constant factors: O(c f(n)) = O(f(n)), where c is a constant; e.g., $O(20\ n^3) = O(n^3)$

(ii) Ignoring smaller terms: If a<b then O(a+b) = O(b), for example $O(n^2 + n) = O(n^2)$

(iii) Upper bound only: If a<b then an O(a) algorithm is also an O(b) algorithm.

(iv) n and log n are "bigger" than any constant, from an asymptotic view (that means for large enough n). So, if k is a constant, an O(n + k) algorithm is also O(n), by ignoring smaller terms. Similarly, an O(log n + k) algorithm is also O(log n).

(v) Another consequence of the last item is that an O(n log n + n) algorithm, which is O(n(log n + 1)), can be simplified to O(n log n)

1.9.2 Properties of Asymptotic Notations

1. **General Properties**: If f (n) is O (g (n)) then a * f (n) is also O (g(n)); where a is a constant. Similarly, this property satisfies both Θ and Ω notation.

2. **Transitive Properties:** If f (n) is O(g(n)) and g(n) is O(h(n)) then f(n) = O(h(n)). Similarly, this property satisfies both Θ and Ω notation. We can say:
   - If f(n) is Θ(g(n)) and g(n) is Θ(h(n)) then f(n) = Θ(h(n))
   - If f(n) is Ω (g(n)) and g(n) is Ω (h(n)) then f(n) = Ω (h(n))

3. **Reflexive Properties:** Reflexive properties are always easy to understand after transitive. If f(n) is given, then f(n) is O(f(n)). Since MAXIMUM VALUE OF f(n) will be f(n) ITSELF! Hence x = f(n) and y = O(f(n) tie themselves in reflexive relation

always. Example: $f(n) = n^2$; $O(n^2)$ i.e., $O(f(n))$. Similarly, this property satisfies both Θ and Ω notation. We can say that:

- If $f(n)$ is given, then $f(n)$ is $\Theta(f(n))$.
- If $f(n)$ is given, then $f(n)$ is $\Omega(f(n))$

4. **Symmetric Properties:** If $f(n)$ is $\Theta(g(n))$ then $g(n)$ is $\Theta(f(n))$. Example: $f(n) = n^2$ and $g(n) = n^2$, then $f(n) = \Theta(n^2)$ and $g(n) = \Theta(n^2)$. This property only satisfies for Θ notation.

5. **Transpose Symmetric Properties**: If $f(n)$ is $O(g(n))$ then $g(n)$ is $\Omega(f(n))$. Example: $f(n) = n$, $g(n) = n^2$, then n is $O(n^2)$ and n^2 is $\Omega(n)$. This property only satisfies O and Ω notations.

6. Some More Properties:
   - If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ then $f(n) = \Theta(g(n))$
   - If $f(n) = O(g(n))$ and $d(n)=O(e(n))$ then $f(n) + d(n) = O(\max(g(n), e(n)))$

   Example: $f(n) = n$ i.e., $O(n)$. $d(n) = n^2$ i.e., $O(n^2)$, then $f(n) + d(n) = n + n^2$ i.e., $O(n^2)$

   - If $f(n)=O(g(n))$ and $d(n)=O(e(n))$ then $f(n) * d(n) = O(g(n) * e(n))$

Example: $f(n) = n$ i.e., $O(n)$, $d(n) = n^2$ i.e., $O(n^2)$, then $f(n) * d(n) = n * n^2 = n^3$ i.e., $O(n^3)$

1.9.3 General rules for the analysis of programs

1. The running time of each assignment, read and write statement can usually be taken to be O(1).
2. The running time of a sequence of statements is determined by the sum rule.
3. The running time of an if–statement is the cost of conditionally executed statements, plus the time for evaluating the condition
4. The time to execute a loop is the sum, over all times around the loop, the time to execute the body and the time to evaluate the condition for termination.

1.10 Recurrence

Many algorithms are recursive in nature. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes. For example, in Merge Sort, to sort a given array, we divide it in two halves and recursively repeat the process for the two halves.

Time complexity of Merge Sort can be written as $T(n) = 2T(n/2) + c*n$.

There are mainly three ways for solving recurrences.

- Substitution Method
- Recurrence Tree Method

- Master Method
- Iteration Method

1.10.1 Substitution Method

One way to solve a divide-and-conquer recurrence equation is to use the iterative substitution method. In using this method, we assume that the problem size n is fairly large and we than substitute the general form of the recurrence for each occurrence of the function T on the right-hand side. For example, consider the recurrence $(n) = 2T\ \boldsymbol{n}\ \mathbf{2} + n$, We guess the solution as $(n) = (n \log \boldsymbol{n})$. Now we use induction to prove our guess. We need to prove that $(n) <= cn \log \boldsymbol{n}$. We can assume that it is true for values smaller than n.

$T\ (\boldsymbol{n} / \mathbf{2}) <= c\ \boldsymbol{n}\ /\mathbf{2} \log \boldsymbol{n} / \mathbf{2}$

$<= \mathbf{2}\ (c\ \boldsymbol{n}\ /\mathbf{2} \log \boldsymbol{n}/\ \mathbf{2}) + n$

$= cn \log \boldsymbol{n} - cn \log \mathbf{2} + n$

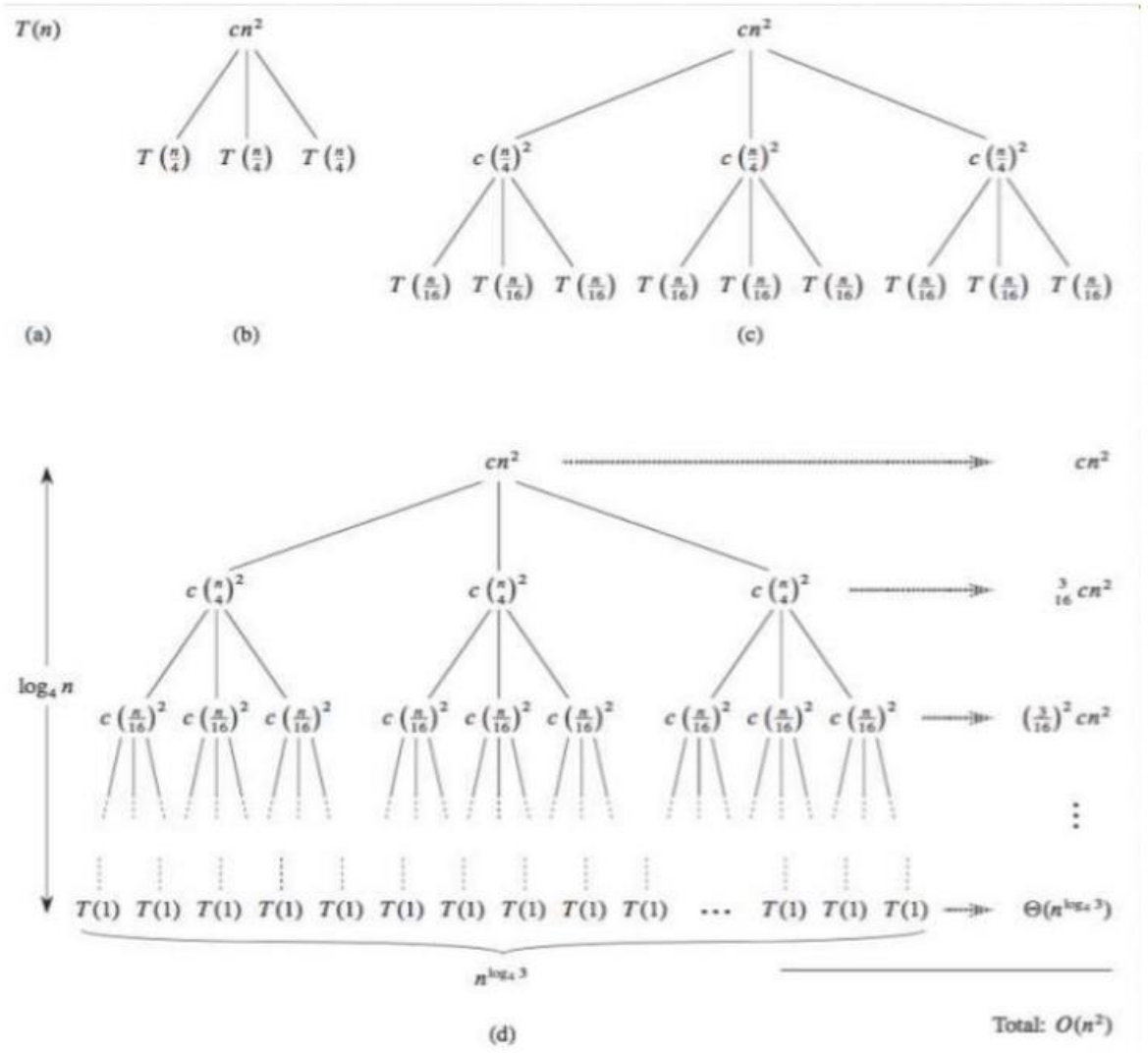
$= cn \log \boldsymbol{n} - cn + n$

$<= cn \log n$

1.10.2 Recurrence Tree Method

Another way of characterizing recurrence equations is to use the recursion tree method. Like the substitution method, this technique uses repeated substitution to solve a recurrence equation, but it differs from the iterative substitution method in that, rather than being an algebraic approach, it is a visual approach.

$(n) = 3T\ n / 4 + c\boldsymbol{n}^2$



1.10.3 Master Method

The master theorem is a formula for solving recurrences of the form $(n) = (n/b) + (n)$, where $a \geq 1$ and $b > 1$ and f(n) is asymptotically positive. This recurrence describes an algorithm that divides a problem of size n into a subproblems, each of size $n /$, and solves them recursively.

The Master Theorem (MT)

The master method depends on the following theorem.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 $T(n)$ can bebounded asymptotically as follows.

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

To find which case the T(n) belongs we can find the log$b\ a$ and if the value is:

- Greater than c, it lies in case 1
- Equal to c, it lies in case 2
- Less than c, it lies in case 3

Where c being the power of n in f(n)

Examples:

(i) $T(n) = 5T(n / 4) + 5n$

$a = 5, b = 4, f(n) = 5n, c = 1, k = 0$

Find: $\log_b a = \log_4 5 = 1.16$

Because: $c < \log_b a$, It is case 1 of MT

$(n) = (n \log\ )$,

$(n) = (n^{1.16})$

(ii) $T(n) = 4T(n / 4) + 5n$

$a = 4, b = 4, f(n) = 5n, c = 1, k = 0$

Find: $\log_b a = \log_4 4 = 1$

Because: $c = \log_b a$ It is case 2 of MT

$T(n) = (n^{\log_{b\ a\ log K^{+1}}} n)$

$T(n) = \theta(n^{1\ log^{0+1}} n) = \theta(n\ log\ n)$

(iii) $T(n) = 4T(n / 5) + 5n$

$a = 4, b = 5, f(n) = 5n, c = 1, k = 0$

Find: $\log_b a = \log_5 4 = 0.86$

Because: $c > \log_b a$ It is case 3 of MT

$T(n) = \theta(f(n))$

$T(n) = \theta(n)$

1.10.4 Iteration Method

The Iteration Method, is also known as the Iterative Method, Backwards Substitution, Substitution Method, and Iterative Substitution. It is a technique or procedure in computational mathematics used to solve a recurrence relation that uses an initial guess to generate a sequence of improving approximate solutions for a class of problems, in which the n^{th} approximation is derived from the previous ones. A Closed-Form Solution is an equation that solves a given problem in terms of functions and mathematical operations from a given generally-accepted set.

Solve the following recurrence relation:

$$T(n) = \begin{cases} 2 & , n = 1 \\ 2.T\left(\frac{n}{2}\right) + 7 & , n > 1 \end{cases}$$

- $T(n) = 2T\left(\frac{n}{2}\right) + 7$
- Find what $T\left(\frac{n}{2}\right)$ is:
- $T\left(\frac{n}{2}\right) = 2T\left(\frac{\frac{n}{2}}{2}\right) + 7 = 2T\left(\frac{n}{4}\right) + 7$
- Now put $T\left(\frac{n}{2}\right)$ in initial equation
- $T(n) = 2\left(2T\left(\frac{n}{4}\right) + 7\right) + 7 = 4T\left(\frac{n}{4}\right) + 21$
- By generalizing the equation
- $T(n) = 2^i T\left(\frac{n}{2^i}\right) + 7(2^i - 1)$

- Recurrence stops when $n = 1$ so we can say that
- $\frac{n}{2^i} = 1$
- $2^i = n$
- $\log 2^i = \log n$
- $i \log 2 = \log n$
- $i = \log n$

- Now replace i in general equation
- $T(n) = 2^{\log n} T\left(\frac{n}{2^{\log n}}\right) + 7(2^{\log n} - 1)$
- $T(n) = nT\left(\frac{n}{n}\right) + 7(n - 1)$
- $T(n) = nT(1) + 7(n - 1)$
- $T(n) = 2n + 7(n - 1)$
- $T(n) = 9n - 7$
- $T(n) = O(n)$

$$2^{\log_2 n} = n$$

The proof:

Let's set the original statement equal to y.

$$y = 2^{\log_2 n}$$

Now, we can apply log base 2 to each side.

$$\log_2 y = \log_2 2^{\log_2 n}$$

Using the previously stated property of log,

$$\log_2 y = \log_2 n \log_2 2$$

Log base b of b will always equal 1.

$$\log_2 y = \log_2 n$$

Therefore,

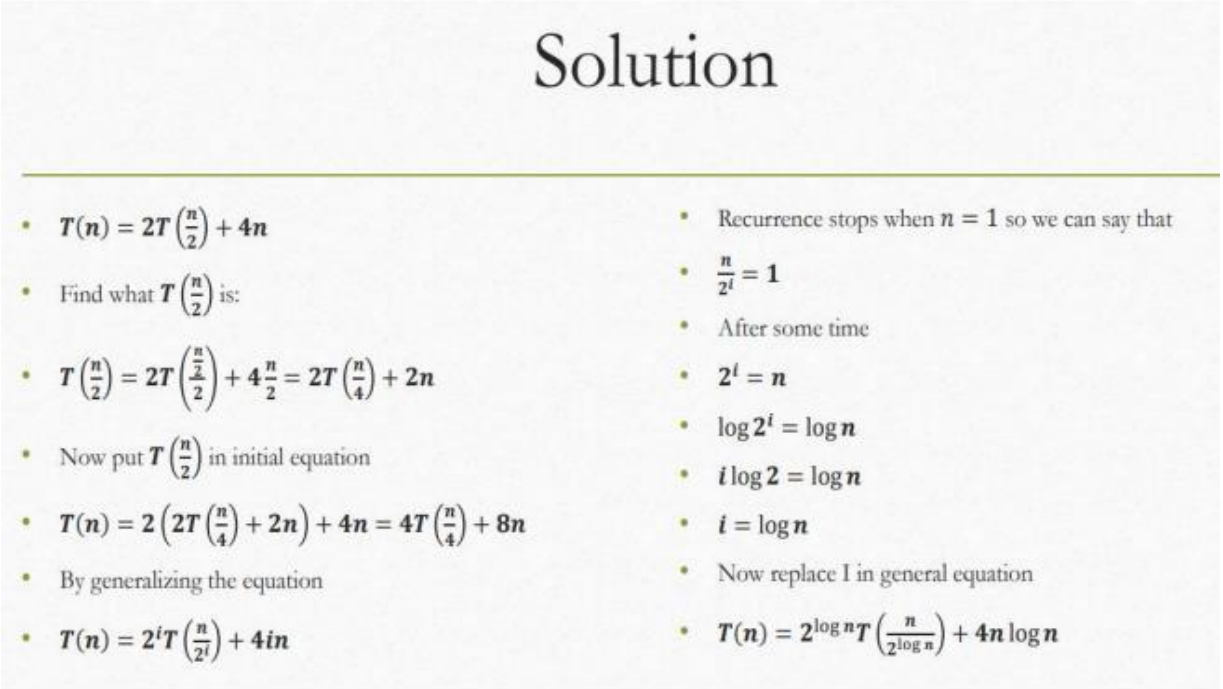
$$y = n$$

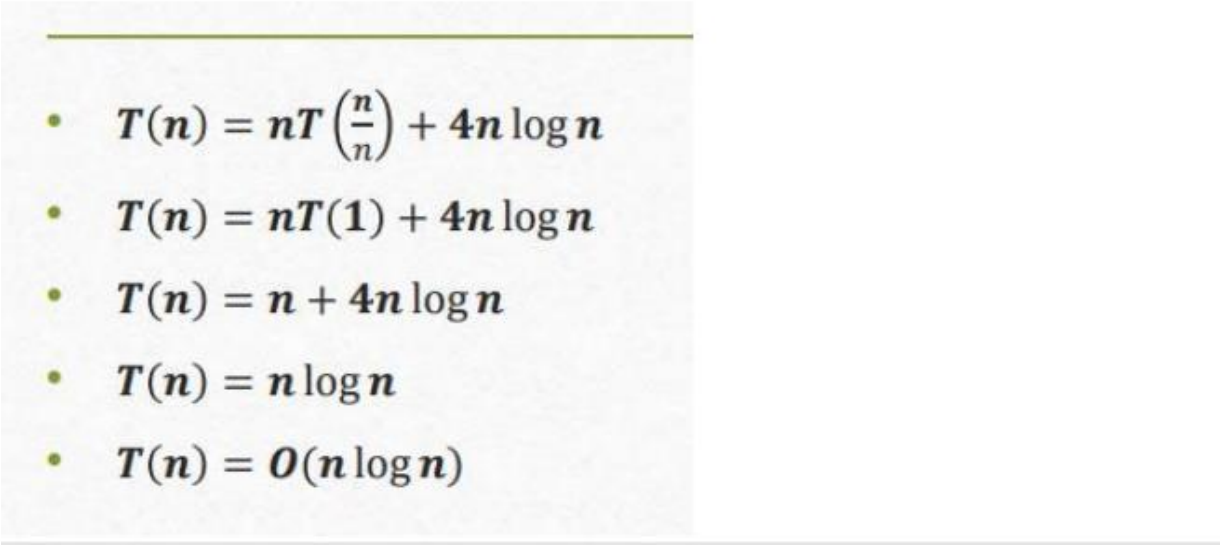
Example 2

$$T(n) = \begin{cases} 1 & , n = 1 \\ 2.T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 4.\ n & , n > 1 \end{cases}$$

Example 2

$$T(n) = \begin{cases} 1 & , n = 1 \\ 2.T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 4.\ n & , n > 1 \end{cases}$$





Study Session Summary

In this session, we examined algorithms. We began by explaining what an algorithm is. Thereafter, we discuss the classification of algorithms and algorithms' performance. Lastly, we examined the solution strategies for recurrences.

Self-Assessment Questions

1. Discuss the different classifications of Algorithms
2. Discuss the different methods for solving recurrences with examples

Study Session 2: Algorithm Design Techniques: Divide and Conquer

Expected Duration: 1 week or 2 contact hours

Introduction
In this Session, we will closely examine how Divide and Conquer Algorithms work, with examples.

Learning Outcomes
When you have studied this session, you should be able to explain:

2.1 Meaning of Divide and Conquer
2.2 Standard Algorithms that Follow Divide and Conquer Algorithm
2.3 Example of Divide and Conquer algorithm
2.4 How Merge Sort works
2.4 Complexity Analysis of Merge Sort:
2.5 Applications of Merge Sort
2.6 Advantages of Merge Sort
2.7 Drawbacks of Merge Sort:

2.1 Meaning of Divide and Conquer

Divide and Conquer is an algorithmic paradigm in which the problem is solved using the Divide, Conquer, and Combine strategy. A typical Divide and Conquer algorithm solves a problem using following three steps:

1. **Divide:** This involves dividing the problem into smaller sub-problems.
2. **Conquer:** Solve sub-problems by calling recursively until solved.
3. **Combine:** Combine the sub-problems to get the final solution of the whole problem.

2.2 Standard Algorithms that Follow Divide and Conquer Algorithm

The following are some standard algorithms that follow Divide and Conquer algorithm.

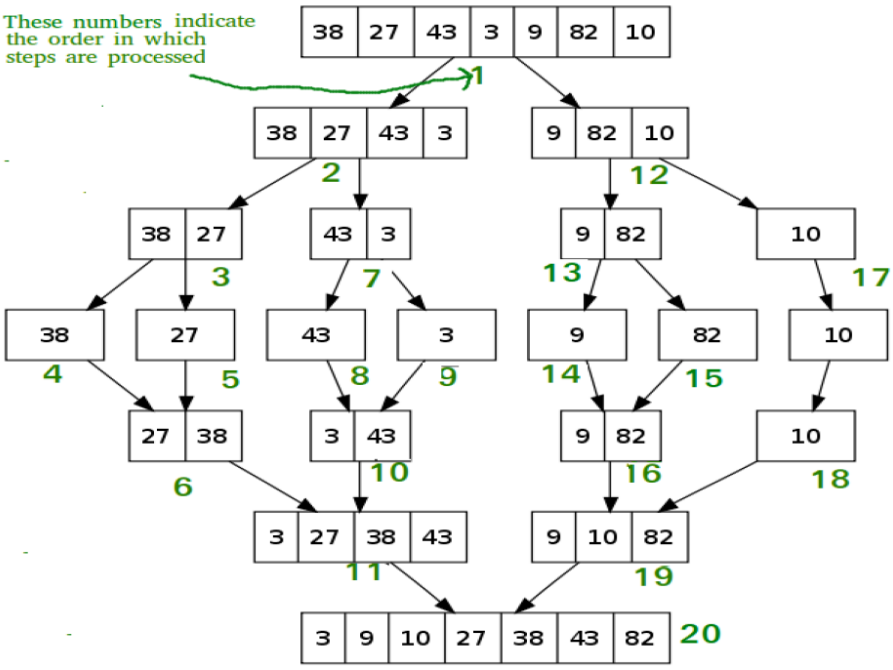
1. **Quicksort** is a sorting algorithm. The algorithm picks a pivot element and rearranges the array elements so that all elements smaller than the picked pivot element move to the left side of the pivot, and all greater elements move to the right side. Finally, the algorithm recursively sorts the subarrays on the left and right of the pivot element.
2. **Merge Sort** is also a sorting algorithm. The algorithm divides the array into two halves, recursively sorts them, and finally merges the two sorted halves.
3. **Closest Pair of Points** The problem is to find the closest pair of points in a set of points in the x-y plane. The problem can be solved in $O(n^2)$ time by calculating the distances of

every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in O(n log n) time.

4. **Strassen's Algorithm** is an efficient algorithm to multiply two matrices. A simple method to multiply two matrices needs 3 nested loops and is O(n^3). Strassen's algorithm multiplies two matrices in O(n^2.8974) time.
5. **Cooley–Tukey Fast Fourier Transform (FFT) algorithm** is the most common algorithm for FFT. It is a divide and conquer algorithm which works in O(N log N) time.
6. **Karatsuba algorithm for fast multiplication** does the multiplication of two *n*-digit numbers in at most [redacted] single-digit multiplications in general (and exactly [redacted] when *n* is a power of 2). It is, therefore, faster than the classical algorithm, which requires n^2 single-digit products. If $n = 2^{10} = 1024$, in particular, the exact counts are $3^{10} = 59,049$ and $(2^{10})^2 = 1,048,576$, respectively.

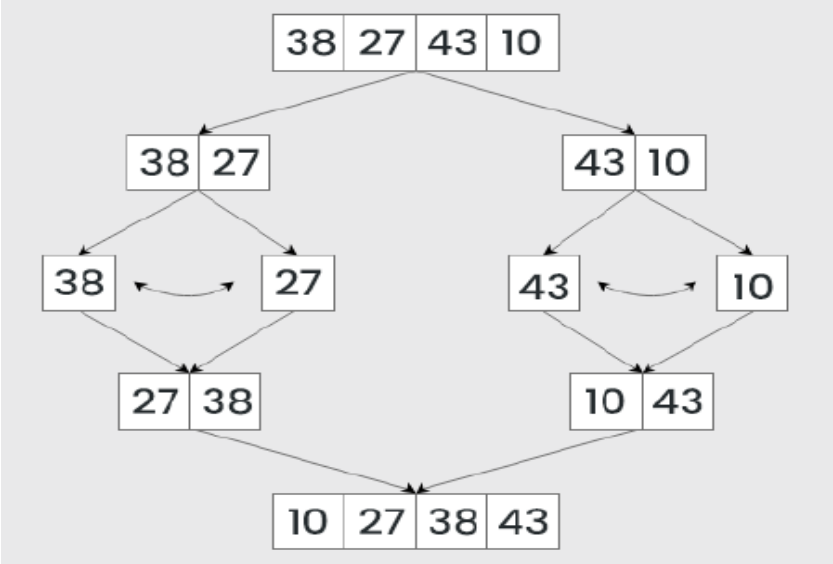
2.3 Example of Divide and Conquer algorithm

A classic example of Divide and Conquer is Merge Sort demonstrated below. In Merge Sort, we divide array into two halves, sort the two halves recursively, and then merge the sorted halves.



Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.



Merge Sort Algorithm

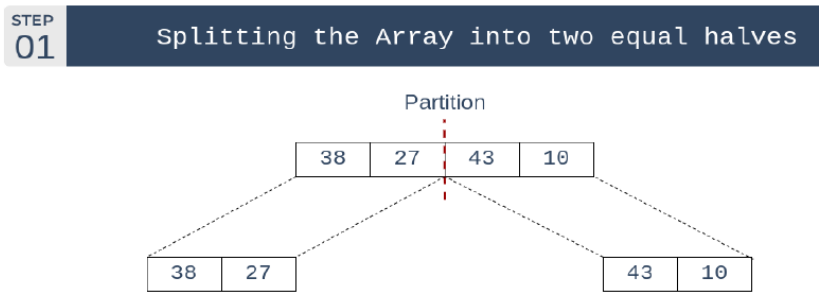
2.4 How does Merge Sort work?

Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

Consider the following illustration to understand the working of merge sort.

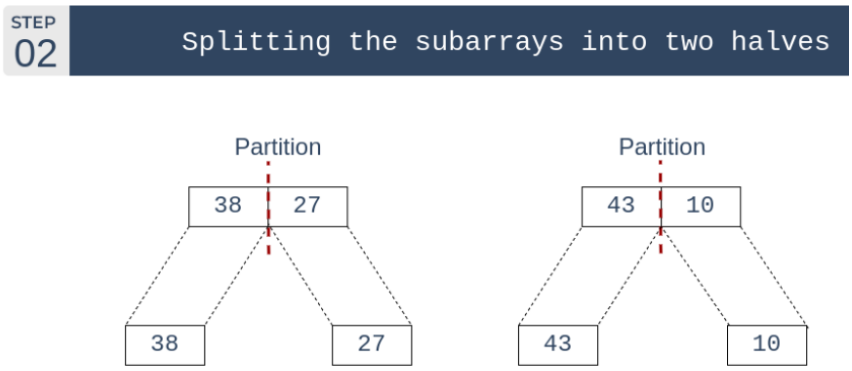
Illustration:
Lets consider an array **arr[] = {38, 27, 43, 10}**

- Initially divide the array into two equal halves:



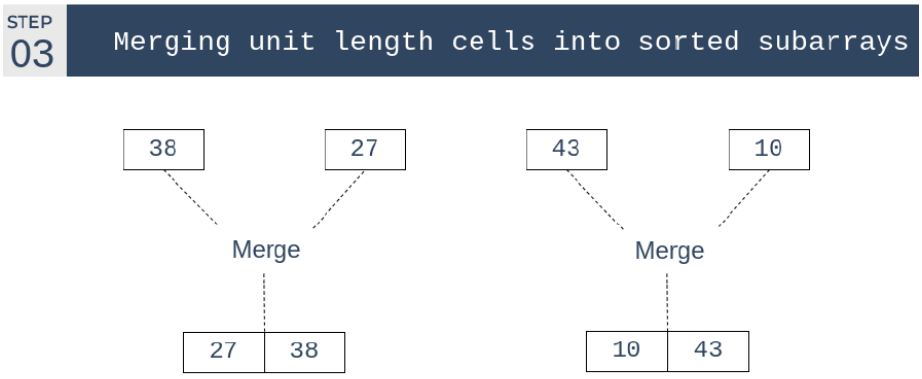
Merge Sort: Divide the array into two halves

- These subarrays are further divided into two halves. Now they become array of unit length that can no longer be divided and array of unit length are always sorted.



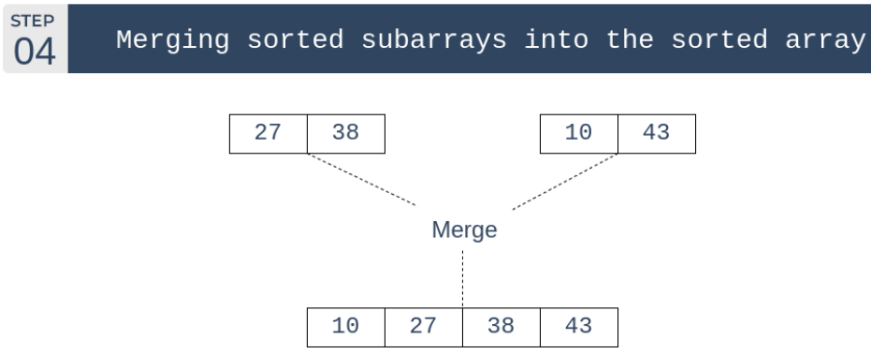
Merge Sort: Divide the subarrays into two halves (unit length subarrays here)

These sorted subarrays are merged together, and we get bigger sorted subarrays.



Merge Sort: Merge the unit length subarrys into sorted subarrays

This merging process is continued until the sorted array is built from the smaller subarrays.



Merge Sort: Merge the sorted subarrys to get the sorted array

The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}.

Below is the Code implementation of Merge Sort.

```
// C++ program for Merge Sort
#include <bits/stdc++.h>
using namespace std;

// Merges two subarrays of array[].
// First subarray is arr[begin..mid]
// Second subarray is arr[mid+1..end]
void merge(int array[ ], int const left, int const mid, int const right)
{
   int const subArrayOne = mid - left + 1;
   int const subArrayTwo = right - mid;

   // Create temp arrays
   auto *leftArray = new int[subArrayOne],
        *rightArray = new int[subArrayTwo];

   // Copy data to temp arrays leftArray[] and rightArray[]
   for (auto i = 0; i < subArrayOne; i++)
      leftArray[i] = array[left + i];
   for (auto j = 0; j < subArrayTwo; j++)
      rightArray[j] = array[mid + 1 + j];

   auto indexOfSubArrayOne = 0, indexOfSubArrayTwo = 0;
   int indexOfMergedArray = left;

   // Merge the temp arrays back into array[left..right]
   while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo < subArrayTwo) {
      if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo]) {
         array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
         indexOfSubArrayOne++;
      }
      else {
         array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
         indexOfSubArrayTwo++;
      }
      indexOfMergedArray++;
   }

   // Copy the remaining elements of
   // left[], if there are any
   while (indexOfSubArrayOne < subArrayOne) {
      array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
      indexOfSubArrayOne++;
```

```
        indexOfMergedArray++;
    }

    // Copy the remaining elements of
    // right[ ], if there are any
    while (indexOfSubArrayTwo < subArrayTwo) {
        array[indexOfMergedArray]  = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
        indexOfMergedArray++;
    }
    delete[ ] leftArray;
    delete[ ] rightArray;
}

// begin is for left index and end is right index
// of the sub-array of arr to be sorted
void mergeSort(int array[], int const begin, int const end) {
    if (begin >= end)
        return;

    int mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}

// UTILITY FUNCTIONS
// Function to print an array
void printArray(int A[], int size)
{
    for (int i = 0; i < size; i++)
        cout << A[i] << " ";
    cout << endl;
}

// Driver code
int main()
{
    int arr[ ] = { 12, 11, 13, 5, 6, 7 };
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    cout << "Given array is \n";
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);
```

```
    cout << "\nSorted array is \n";
    printArray(arr, arr_size);
    return 0;
}

// This code is contributed by Mayank Tyagi
// This code was revised by Joshua Estes
```

Output

Given array is

12 11 13 5 6 7

Sorted array is

5 6 7 11 12 13

2.4 Complexity Analysis of Merge Sort:

Time Complexity: O (n log(n)), Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.
$T(n) = 2T(n/2) + \theta(n)$

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of the Master Method and the solution of the recurrence is θ(nlog(n)). The time complexity of Merge Sort is θ(nlog(n)) in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Auxiliary Space: O (n), In merge sort all elements are copied into an auxiliary array. So N auxiliary space is required for merge sort.

2.5 Applications of Merge Sort

1. **Sorting large datasets:** Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of O(n log n).
2. **External sorting:** Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.
3. **Custom sorting:** Merge sort can be adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.
4. Inversion Count Problem

2.6 Advantages of Merge Sort

1. **Stability**: Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.

2. **Guaranteed worst-case performance:** Merge sort has a worst-case time complexity of O (n log n), which means it performs well even on large datasets.
3. **Parallelizable**: Merge sort is a naturally parallelizable algorithm, which means it can be easily parallelized to take advantage of multiple processors or threads.

2.7 Drawbacks of Merge Sort:

1. **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
2. **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
3. **Not always optimal for small datasets:** For small datasets, Merge sort has a higher time complexity than some other sorting algorithms, such as insertion sort. This can result in slower performance for very small datasets.

2.8 Closest Pair of Points Problem

Another example of Divide and Conquer Algorithm is Closest Pair of Points Problem. In this problem, a set of n points are given on the 2D plane. In this problem, we have to find the pair of points, whose distance is minimum.

To solve this problem, we have to divide points into two halves, after that smallest distance between two points is calculated in a recursive way. Using distances from the middle line, the points are separated into some strips. We will find the smallest distance from the strip array. At first two lists are created with data points, one list will hold points which are sorted on x values, another will hold data points, sorted on y values. The time complexity of this algorithm will be O(n log n).

Input: A set of different points are given. (2, 3), (12, 30), (40, 50), (5, 1), (12, 10), (3, 4)
Output: Find the minimum distance from each pair of given points.
Here the minimum distance is: 1.41421 unit

The Algorithm:

```
findMinDist(pointsList, n)
Input: Given point list and number of points in the list.
Output − Finds minimum distance from two points.
Begin
  min := ∞
  for all items i in the pointsList, do
    for j := i+1 to n-1, do
      if distance between pointList[i] and pointList[j] < min, then
        min = distance of pointList[i] and pointList[j]
   done
  done
  return min
```

```
End

stripClose(strips, size, dist)
Input − Different points in the strip, number of points, distance from the midline.
Output − Closest distance from two points in a strip.
Begin
  for all items i in the strip, do
    for j := i+1 to size-1 and (y difference of ithand jth points) <min, do
      if distance between strip[i] and strip [j] < min, then
        min = distance of strip [i] and strip [j]
    done
  done
  return min
End

findClosest(xSorted, ySorted, n)
Input − Points sorted on x values, and points sorted on y values, number of points.
Output − Find minimum distance from the total set of points.
Begin
  if n <= 3, then
    call findMinDist(xSorted, n)
    return the result
  mid := n/2
  midpoint := xSorted[mid]
  define two sub lists of points to separate points along vertical line.
  the sub lists are, ySortedLeft and ySortedRight

  leftDist := findClosest(xSorted, ySortedLeft, mid)        //find left distance
  rightDist := findClosest(xSorted, ySortedRight, n - mid)   //find right distance

  dist := minimum of leftDist and rightDist

  make strip of points
  j := 0
  for i := 0 to n-1, do
    if difference of ySorted[i].x and midPoint.x<dist, then
      strip[j] := ySorted[i]
      j := j+1
  done

  close := stripClose(strip, j, dist)
  return minimum of close and dist
End
```

Example C - Code Implementation

```
#include <iostream>
```

```
#include<cmath>
#include<algorithm>
using namespace std;

struct point {
  int x, y;
};

intcmpX(point p1, point p2) {    //to sort according to x value
  return (p1.x < p2.x);
}

intcmpY(point p1, point p2) {    //to sort according to y value
  return (p1.y < p2.y);
}

float dist(point p1, point p2) {    //find distance between p1 and p2
  return sqrt((p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y));
}

float findMinDist(point pts[], int n) {    //find minimum distance between two points in a set
  float min = 9999;
  for (int i = 0; i < n; ++i)
    for (int j = i+1; j < n; ++j)
      if (dist(pts[i], pts[j]) < min)
        min = dist(pts[i], pts[j]);
  return min;
}

float min(float a, float b) {
  return (a < b)? a : b;
}

float stripClose(point strip[], int size, float d) {    //find closest distance of two points in a strip
  float min = d;
  for (int i = 0; i < size; ++i)
    for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
      if (dist(strip[i],strip[j]) < min)
        min = dist(strip[i], strip[j]);
  return min;
}

float findClosest(point xSorted[], point ySorted[], int n){
  if (n <= 3)
    return findMinDist(xSorted, n);
  int mid = n/2;
```

```
  point midPoint = xSorted[mid];
  point ySortedLeft[mid+1];     // y sorted points in the left side
  point ySortedRight[n-mid-1];  // y sorted points in the right side
  intleftIndex = 0, rightIndex = 0;

  for (int i = 0; i < n; i++) {       //separate y sorted points to left and right
    if (ySorted[i].x <= midPoint.x)
      ySortedLeft[leftIndex++] = ySorted[i];
    else
      ySortedRight[rightIndex++] = ySorted[i];
  }

  float leftDist = findClosest(xSorted, ySortedLeft, mid);
  float rightDist = findClosest(ySorted + mid, ySortedRight, n-mid);
  float dist = min(leftDist, rightDist);

  point strip[n];      //hold points closer to the vertical line
  int j = 0;

  for (int i = 0; i < n; i++)
    if (abs(ySorted[i].x - midPoint.x) <dist) {
      strip[j] = ySorted[i];
      j++;
    }
  return min(dist, stripClose(strip, j, dist));    //find minimum using dist and closest pair in strip
}

float closestPair(point pts[], int n) {    //find distance of closest pair in a set of points
  point xSorted[n];
  point ySorted[n];

  for (int i = 0; i < n; i++) {
    xSorted[i] = pts[i];
    ySorted[i] = pts[i];
  }

  sort(xSorted, xSorted+n, cmpX);
  sort(ySorted, ySorted+n, cmpY);
  return findClosest(xSorted, ySorted, n);
}

int main() {
  point P[] ={{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};
  int n = 6;
  cout<< "The minimum distance is " <<closestPair(P, n);
```

}
Output
The minimum distance is 1.41421

Study Session Summary

In this Session, you were introduced to the technique of Divide and Conquer as a means of designing Algorithms. Merge Sort algorithm and Closest Pair of Points Problem were used as an example. Now answer the following self-assessment questions.

Self-Assessment Questions

1. Explain the working principle of Divide and Conquer Algorithms
2. Explain how Merge Sort works. What are the strengths and weaknesses of Merge Sort

Study Session 3: Algorithm Design Techniques: Dynamic Programming

Expected Duration: 1 week or 2 contact hours

Introduction
In this Session, we will closely examine how dynamic programming works, with examples.

Learning Outcomes
When you have studied this session, you should be able to explain:

3.1 Meaning of Dynamic Programming

Dynamic programming is defined as a computer programming technique where an algorithmic problem is first broken down into sub-problems, the results are saved, and then the sub-problems are optimized to find the overall solution - which usually has to do with finding the maximum and minimum range of the algorithmic query.

Richard Bellman was the one who came up with the idea for dynamic programming in the 1950s. It is a method of **mathematical optimization as well as a methodology for computer programming**. It applies to issues one can break down into either overlapping subproblems or optimum substructures.

When a more extensive set of equations is broken down into smaller groups of equations, overlapping subproblems are referred to as equations that reuse portions of the smaller equations several times to arrive at a solution.

On the other hand, optimum substructures locate the best solution to an issue, then build the solution that provides the best results overall. This is how they solve problems. When a vast issue is split down into its constituent parts, a computer will apply a mathematical algorithm to determine which elements have the most desirable solution. Then, it takes the solutions to the

more minor problems and utilizes them to get the optimal solution to the initial, more involved issue.

This technique solves problems by breaking them into smaller, overlapping subproblems. The results are then stored in a table to be reused so the same problem will not have to be computed again.

For example, when using the dynamic programming technique to figure out all possible results from a set of numbers, the first time the results are calculated, they are saved and put into the equation later instead of being calculated again. So, when dealing with long, complicated equations and processes, it saves time and makes solutions faster by doing less work.

The dynamic programming algorithm tries to find the shortest way to a solution when solving a problem. It does this by going from the top down or the bottom up. The top-down method solves equations by breaking them into smaller ones and reusing the answers when needed. The bottom-up approach solves equations by breaking them up into smaller ones, then tries to solve the equation with the smallest mathematical value, and then works its way up to the equation with the biggest value.

Using dynamic programming to solve problems is more effective than just trying things until they work. But it only helps with problems that one can break up into smaller equations that will be used again at some point.

3.2 Characteristics of Dynamic Programming Algorithm

(i) In general, dynamic programming (DP) is one of the most powerful techniques for solving a certain class of problems.
(ii) There is an elegant way to formulate the approach and a very simple thinking process, and the coding part is very easy.
(iii) Essentially, it is a simple idea, after solving a problem with a given input, save the result as a reference for future use, so you won't have to re-solve it. Briefly 'Remember your Past' :).
(iv) It is a big hint for DP if the given problem can be broken up into smaller sub-problems, and these smaller subproblems can be divided into still smaller ones, and in this process, you see some overlapping subproblems.
(v) Additionally, the optimal solutions to the subproblems contribute to the optimal solution of the given problem (referred to as the Optimal Substructure Property).
(vi) The solutions to the subproblems are stored in a table or array (memoization) or in a bottom-up manner (tabulation) to avoid redundant computation.
(vii) The solution to the problem can be constructed from the solutions to the subproblems.
(viii) Dynamic programming can be implemented using a recursive algorithm, where the solutions to subproblems are found recursively, or using an iterative algorithm, where the solutions are found by working through the subproblems in a specific order.

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

For example, if we write simple recursive solution for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.

(a)

```
int fib(int n) {
   if (n <= 1)
       return n;
  return fib(n – 1) + fib(n – 2);          Recursion: Exponential
}
```

(b)

```
f[0] = 0;
f[1] = 1;
for (i = 2; i <= n; i++) {
     f[i] = f[i – 1] + f[i  - 2];          Dynamic Programming: Linear
}
return f[n];
```

The development of a dynamic programming algorithm can be subdivided into the following steps:

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution in a bottom-up fashion
4. Construct an optimal solution from computed information

3.3 Recursion vs. dynamic programming

In Computer Science, recursion is a crucial concept in which the solution to a problem depends on solutions to its smaller subproblems.

Meanwhile, dynamic programming is an optimization technique for recursive solutions. It is the preferred technique for solving recursive functions that make repeated calls to the same inputs. A function is known as recursive if it calls itself during execution. This process can repeat itself several times before the solution is computed and can repeat forever if it lacks a base case to enable it to fulfill its computation and stop the execution.

However, not all problems that use recursion can be solved by dynamic programming. Unless solutions to the subproblems overlap, a recursion solution can only be arrived at using a divide-and-conquer method.

For example, problems like merge, sort, and quick sort are not considered dynamic programming problems. This is because they involve putting together the best answers to subproblems that don't overlap.

3.4 Drawbacks of recursion

Recursion uses memory space less efficiently. Repeated function calls create entries for all the variables and constants in the function stack. As the values are kept there until the function returns, there is always a limited amount of stack space in the system, thus making less efficient use of memory space. Additionally, a stack overflow error occurs if the recursive function requires more memory than is available in the stack.

Recursion is also relatively slow in comparison to iteration, which uses loops. When a function is called, there is an overhead of allocating space for the function and all its data in the function stack in recursion. This causes a slight delay in recursive functions.

3.5 Where Should Dynamic Programming be Used?

Dynamic programming is used when one can break a problem into more minor issues that they can break down even further, into even more minor problems. Additionally, these subproblems have overlapped. That is, they require previously calculated values to be recomputed. With dynamic programming, the computed values are stored, thus reducing the need for repeated calculations and saving time and providing faster solutions.

3.6 How Does Dynamic Programming Work?

Dynamic programming works by breaking down complex problems into simpler subproblems. Then, finding optimal solutions to these subproblems. Memorization is a method that saves the outcomes of these processes so that the corresponding answers do not need to be computed when they are later needed. Saving solutions save time on the computation of subproblems that have already been encountered.

Dynamic programming can be achieved using two approaches:

1. Top-down approach

In computer science, problems are resolved by recursively formulating solutions, employing the answers to the problems' subproblems. If the answers to the subproblems overlap, they may be memoized or kept in a table for later use. The top-down approach follows the strategy of memorization. The memoization process is equivalent to adding the recursion and caching steps. The difference between recursion and caching is that recursion requires calling the function directly, whereas caching requires preserving the intermediate results.

The top-down strategy has many benefits, including the following:

- The top-down approach is easy to understand and implement. In this approach, problems are broken down into smaller parts, which help users identify what needs to be done. With each step, more significant, more complex problems become smaller, less complicated, and, therefore, easier to solve. Some parts may even be reusable for the same problem.
- It allows for subproblems to be solved upon request. The top-down approach will enable problems to be broken down into smaller parts and their solutions stored for reuse. Users can then query solutions for each part.
- It is also easier to debug. Segmenting problems into small parts allows users to follow the solution quickly and determine where an error might have occurred.

Disadvantages of the top-down approach include:

- The top-down approach uses the recursion technique, which occupies more memory in the call stack. This leads to reduced overall performance. Additionally, when the recursion is too deep, a stack overflow occurs.

2. Bottom-up approach

In the bottom-up method, once a solution to a problem is written in terms of its subproblems in a way that loops back on itself, users can rewrite the problem by solving the smaller subproblems first and then using their solutions to solve the larger subproblems.

Unlike the top-down approach, the bottom-up approach removes the recursion. Thus, there is neither stack overflow nor overhead from the recursive functions. It also allows for saving memory space. Removing recursion decreases the time complexity of recursion due to recalculating the same values.

The advantages of the bottom-up approach include the following:

- It makes decisions about small reusable subproblems and then decides how they will be put together to create a large problem.
- It removes recursion, thus promoting the efficient use of memory space. Additionally, this also leads to a reduction in timing complexity.

3.7 Signs of Dynamic Programming Suitability

Dynamic programming solves complex problems by breaking them up into smaller ones using recursion and storing the answers so they don’t have to be worked out again. It isn’t practical when there aren’t any problems that overlap because it doesn’t make sense to store solutions to the issues that won’t be needed again.

Two main signs are that one can solve a problem with dynamic programming: subproblems that overlap and the best possible substructure.

3.7.1 Overlapping subproblems

When the answers to the same subproblem are needed more than once to solve the main problem, we say that the subproblems overlap. In overlapping issues, solutions are put into a table so developers can use them repeatedly instead of recalculating them. The recursive program for the Fibonacci numbers has several subproblems that overlap, but a binary search doesn't have any subproblems that overlap.

A binary search is solved using the divide and conquer technique. Every time, the subproblems have a unique array to find the value. Thus, binary search lacks the overlapping property.

For example, when finding the nth Fibonacci number, the problem F(n) is broken down into finding F(n-1) and F.(n-2). You can break down F(n-1) even further into a subproblem that has to do with F. (n-2).In this scenario, F(n-2) is reused, and thus, the Fibonacci sequence can be said to exhibit overlapping properties.

3.7.2 Optimal substructure

The optimal substructure property of a problem says that you can find the best answer to the problem by taking the best solutions to its subproblems and putting them together. Most of the time, recursion explains how these optimal substructures work.

This property is not exclusive to dynamic programming alone, as several problems consist of optimal substructures. However, most of them lack overlapping issues. So, they can't be called problems with dynamic programming.

You can use it to find the shortest route between two points. For example, if a node p is on the shortest path from a source node t to a destination node w, then the shortest path from t to w is the sum of the shortest paths from t to p and from p to w.

Examples of problems with optimal substructures include the longest increasing subsequence, longest palindromic substring, and longest common subsequence problem. Examples of problems without optimal substructures include the most extended path problem and the addition-chain exponentiation.

3.8 Understanding the Longest Common Subsequence Concept in Dynamic Programming

In dynamic programming, the phrase "largest common subsequence" (LCS) refers to the subsequence that is shared by all of the supplied sequences and is the one that is the longest. It is different from the challenge of finding the longest common substring in that the components of the LCS do not need to occupy consecutive locations within the original sequences to be considered part of that problem.

The LCS is characterized by an optimal substructure and overlapping subproblem properties. This indicates that the issue may be split into many less complex sub-issues and worked on

individually until a solution is found. The solutions to higher-level subproblems are often reused in lower-level subproblems, thus, overlapping subproblems.

Therefore, when solving an LCS problem, it is more efficient to use a dynamic algorithm than a recursive algorithm. Dynamic programming stores the results of each function call so that it can be used in future calls, thus minimizing the need for redundant calls.

For instance, consider the sequences (MNOP) and (MONMP). They have five length-2 common subsequences (MN), (MO), (MP), (NP), and (OP); two length-3 common subsequences (MNP) and (MOP); MNP and no longer frequent subsequences (MOP). Consequently, (MNP) and (MOP) are the largest shared subsequences. LCS can be applied in bioinformatics to the process of genome sequencing.

3.9 Dynamic Programming Algorithms

When dynamic programming algorithms are executed, they solve a problem by segmenting it into smaller parts until a solution arrives. They perform these tasks by finding the shortest path. Some of the primary dynamic programming algorithms in use are:

1. Greedy algorithms

An example of dynamic programming algorithms, greedy algorithms are also optimization tools. The method solves a challenge by searching for optimum solutions to the subproblems and combining the findings of these subproblems to get the most optimal answer.

Conversely, when greedy algorithms solve a problem, they look for a locally optimum solution to find a global optimum. They make a guess that looks optimum at the time but does not guarantee a globally optimum solution. This could end up becoming costly down the road.

2. Floyd-Warshall algorithm

The Floyd-Warshall method uses a technique of dynamic programming to locate the shortest pathways. It determines the shortest route across all pairings of vertices in a graph with weights. Both directed and undirected weighted graphs can use it.

This program compares each pair of vertices' potential routes through the graph. It gradually optimizes an estimate of the shortest route between two vertices to determine the shortest distance between two vertices in a chart. With simple modifications to it, one can reconstruct the paths.

This method for dynamic programming contains two subtypes:

- **Behavior with negative cycles:** Users can use the Floyd-Warshall algorithm to find negative cycles. You can do this by inspecting the diagonal path matrix for a negative number that would indicate the graph contains one negative cycle. In a negative cycle, the sum of the edges is a negative value; thus, there cannot be a

shortest path between any pair of vertices. Exponentially huge numbers are generated if a negative cycle occurs during algorithm execution.

- **Time complexity:** The Floyd-Warshall algorithm has three loops, each with constant complexity. As a result, the Floyd-Warshall complexity has a time complexity of O(n3). Wherein n represents the number of network nodes.

3. Bellman Ford algorithm

The Bellman-Ford Algorithm determines the shortest route from a particular source vertex to every other weighted digraph vertices. The Bellman-Ford algorithm can handle graphs where some of the edge weights are negative numbers and produce a correct answer, unlike Dijkstra's algorithm, which does not confirm whether it makes the correct answer. However, it is much slower than Dijkstra's algorithm.

The Bellman-Ford algorithm works by relaxation; that is, it gives approximate distances that better ones continuously replace until a solution is reached. The approximate distances are usually overestimated compared to the distance between the vertices. The replacement values reflect the minimum old value and the length of a newly found path.

This algorithm terminates upon finding a negative cycle and thus can be applied to cycle-canceling techniques in network flow analysis.

3.10 Examples of Dynamic Programming

Here are a few examples of how one may use dynamic programming:

1. Identifying the number of ways to cover a distance

Some recursive functions are invoked three times in the recursion technique, indicating the overlapping subproblem characteristic required to calculate issues that use the dynamic programming methodology.

Using the top-down technique, just store the value in a HashMap while retaining the recursive structure, then return the value store without calculating each time the function is invoked. Utilize an extra space of dimension n when employing the bottom-up method and compute the values of states beginning with 1, 2,…, n, i.e., compute the values of I i+1 and i+2 and then apply them to determine the value of i+3.

2. Identifying the optimal strategy of a game

To identify the optimal strategy of a game or gamified experience, let's consider the "coins in a line" game. The memoization technique is used to compute the maximum value of coins taken by player A for coins numbered h to k, assuming player B plays optimally (Mh, k). To find out each player's strategy, assign values to the coin they pick and the value of the opponent's coin.

After computation, the optimal design for the game is determined by observing the Mh, k value for both players if player A chooses coin h or k.

3. Counting the number of possible outcomes of a particular die roll

With an integer M, the aim is to determine the number of approaches to obtain the sum M by tossing dice repeatedly. The partial recursion tree, where M = 8, provides overlapping subproblems when using the recursion method. By using dynamic programming, one can optimize the recursive method. One can use an array to store values after computation for reuse. In this way, the algorithm takes significantly less time to run with time complex: O(t * n * m), with t being the number of faces, n being the number of dice, and m being the given sum.

Session Summary

Dynamic programming is among the more advanced skills one must learn as a programmer or DevOps engineer, mainly if you specialize in Python. It is a relatively simple way to solve complex algorithmic problems and a skill you can apply to virtually any language or use case. For example, the viral game, Wordle, follows dynamic programming principles, and users can train an algorithm to resolve it by finding the most optimal combination of alphabets. In other words, the skill has versatile applications and must be part of every DevOps learning kit.

Self-Assessment Questions

1. Explain Dynamic Programming as a technique for designing algorithms

2. Explain the conditions that may warrant the use of Dynamic Programming

3. Briefly explain any two examples of Dynamic Programming problems

Study Session 4: Algorithm Design Techniques: Greedy Algorithms

Expected Duration: 1 week or 2 contact hours

Introduction

Greedy algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.

Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

In this Session, we will closely examine how Greedy Algorithm works, with examples.

Learning Outcomes

When you have studied this session, you should be able to explain:

4.1 Meaning of Greedy Algorithm
4.2 Advantages of Greedy Approach
4.3 Drawback of Greedy Approach
4.4 The Greedy Algorithm
4.5 Different Types of Greedy Algorithm

4.1 Meaning of Greedy Algorithm

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result. The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.

This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result. However, we can determine if the algorithm can be used with any problem if the problem has the following properties:

1. Greedy Choice Property

If an optimal solution to the problem can be found by choosing the best choice at each step without reconsidering the previous steps once chosen, the problem can be solved using a greedy approach. This property is called greedy choice property.

2. Optimal Substructure

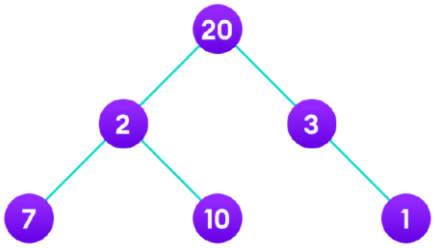
If the optimal overall solution to the problem corresponds to the optimal solution to its subproblems, then the problem can be solved using a greedy approach. This property is called optimal substructure.

4.2 Advantages of Greedy Approach

- The algorithm is **easier to describe**.
- This algorithm can **perform better** than other algorithms (but, not in all cases).

4.3 Drawback of Greedy Approach

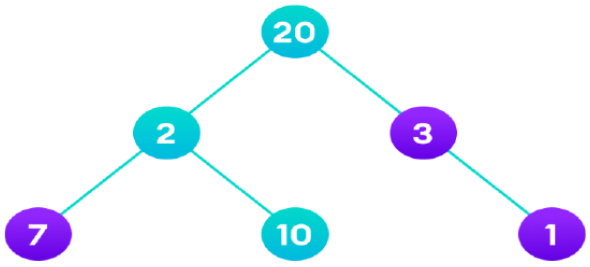
As mentioned earlier, the greedy algorithm doesn't always produce the optimal solution. This is the major disadvantage of the algorithm. For example, suppose we want to find the longest path in the graph below from root to leaf. Let's use the greedy algorithm here.



Apply greedy approach to this tree to find the longest route

Greedy Approach

1. Let's start with the root node **20**. The weight of the right child is **3** and the weight of the left child is **2**.
2. Our problem is to find the largest path. And, the optimal solution at the moment is **3**. So, the greedy algorithm will choose **3**.
3. Finally the weight of an only child of **3** is **1**. This gives us our final result 20 + 3 + 1 = 24. However, it is not the optimal solution. There is another path that carries more weight (20 + 2 + 10 = 32) as shown in the image below.



Longest path

Therefore, greedy algorithms do not always give an optimal/feasible solution.

4.4 The Greedy Algorithm

1. To begin with, the solution set (containing answers) is empty.
2. At each step, an item is added to the solution set until a solution is reached.
3. If the solution set is feasible, the current item is kept.
4. Else, the item is rejected and never considered again.

Let's now use this algorithm to solve some problems.

Example 1 - Greedy Approach
Problem: You have to make a change of an amount using the smallest possible number of coins.
Amount: ₦18

Assume the available coins are:
₦5 coin
₦2 coin
₦1 coin

But there is no limit to the number of each coin you can use.

Solution:
1. Create an empty solution-set = { }. Available coins are {5, 2, 1}.
2. We are supposed to find the sum = 18. Let's start with sum = 0.
3. Always select the coin with the largest value (i.e. 5) until the sum > 18. (When we select the largest value at each step, we hope to reach the destination faster. This concept is called **greedy choice property**.)
4. In the first iteration, solution-set = {5} and sum = 5.
5. In the second iteration, solution-set = {5, 5} and sum = 10.
6. In the third iteration, solution-set = {5, 5, 5} and sum = 15.
7. In the fourth iteration, solution-set = {5, 5, 5, 2} and sum = 17. (We cannot select 5 here because if we do so, sum = 20 which is greater than 18. So, we select the 2nd largest item which is 2.)
8. Similarly, in the fifth iteration, select 1. Now sum = 18 and solution-set = {5, 5, 5, 2, 1}.

Example 2: Activity Selection Problem

Suppose n different activities are given with their starting times and ending times. Select the maximum number of activities to solve by a single person. We will use the greedy approach to find the next activity whose finish time is minimum among rest activities, and the start time is more than or equal with the finish time of the last selected activity.

- The complexity of this problem is O(n log n) when the list is not sorted.
- When the sorted list is provided the complexity will be O(n).

Input and Output
Input:
A list of different activities with starting and ending times:
{(5,9), (1,2), (3,4), (0,6), (5,7), (8,9)}

Each pair of numbers is an activity indicating its start and finish times.

Output:
Selected Activities are:
Activity: 0 , Start: 1 End: 2
Activity: 1 , Start: 3 End: 4
Activity: 3 , Start: 5 End: 7
Activity: 5 , Start: 8 End: 9

The Algorithm:

maxActivity(act, size)
Input**:** A list of activity, and the number of elements in the list.
Output − The order of activities how they have been chosen.

```
Begin
  initially sort the given activity List
  set i := 1
  display the ith activity //in this case it is the first activity

  for j := 1 to n-1 do
    if start time of act[j] >= end of act[i] then
      display the jth activity
      i := j
  done
End
```

Example C- Language Code:

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

struct Activitiy {
  int start, end;
};

bool comp(Activitiy act1, Activitiy act2) {
  return (act1.end < act2.end);
}
```

```
void maxActivity(Activitiy act[], int n) {
  sort(act, act+n, comp); //sort activities using compare function

  cout << "Selected Activities are: " << endl;
  int i = 0;// first activity as 0 is selected
  cout << "Activity: " << i << " , Start: " <<act[i].start << " End:
    " << act[i].end <<endl;

  for (int j = 1; j < n; j++) { //for all other activities
    if (act[j].start >= act[i].end) { //when start time is >= end
      time, print the activity
      cout << "Activity: " << j << " , Start: " <<act[j].start << " End: " << act[j].end <<endl;
      i = j;
    }
  }
}

int main() {
  Activitiy actArr[] = {{5,9},{1,2},{3,4},{0,6},{5,7},{8,9}};
  int n = 6;
  maxActivity(actArr,n);
  return 0;
}
```

Output:
Selected Activities are:
Activity: 0 , Start: 1 End: 2
Activity: 1 , Start: 3 End: 4
Activity: 3 , Start: 5 End: 7
Activity: 5 , Start: 8 End: 9

4.5 Different Types of Greedy Algorithm

1. Selection Sort
2. Knapsack Problem
3. Minimum Spanning Tree
4. Single-Source Shortest Path Problem
5. Job Scheduling Problem
6. Prim's Minimal Spanning Tree Algorithm
7. Kruskal's Minimal Spanning Tree Algorithm
8. Dijkstra's Minimal Spanning Tree Algorithm
9. Huffman Coding
10. Ford-Fulkerson Algorithm

Session Summary

It has been explained in this Session that Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

Self-Assessment Questions

1. Explain Greedy Algorithm as a technique for designing algorithms
2. Take any two of all the different types of Greedy Algorithm mentioned in Section 4.5 of this Unit, and explain with pseudocodes their working principles.

Study Session 5: Algorithm Design Techniques: Randomized Algorithms

Expected Duration: 1 week or 2 contact hours

Introduction

Randomized Algorithms are different from deterministic algorithms; deterministic algorithms follow a definite procedure to get the same output every time an input is passed. In this Session, randomized algorithms as techniques of designing algorithms are considered.

Learning Outcomes

When you have studied this session, you should be able to explain:

5.1 Meaning of Randomized Algorithms
5.2 Classification of Randomized Algorithms
5.3 Need for Randomized Algorithms
  5.3.1 The Game Theory and Randomized Algorithms
  5.3.2 Zero-sum game

5.1 Meaning of Randomized Algorithms

Randomized algorithm is a different design approach taken by the standard algorithms where few random bits are added to a part of their logic. They are different from deterministic algorithms; deterministic algorithms follow a definite procedure to get the same output every time an input is passed where randomized algorithms produce a different output every time they're executed. It is important to note that it is not the input that is randomized, but the logic of the standard algorithm.

Unlike deterministic algorithms, randomized algorithms consider randomized bits of the logic along with the input that in turn contribute towards obtaining the output. However, the probability of randomized algorithms providing incorrect output cannot be ruled out either. Hence, the process called **amplification** is performed to reduce the likelihood of these erroneous outputs. Amplification is also an algorithm that is applied to execute some parts of the randomized algorithms multiple times to increase the probability of correctness. However, too much amplification can also exceed the time constraints making the algorithm ineffective.

5.2 Classification of Randomized Algorithms

Randomized algorithms are classified based on whether they have time constraints as the random variable or deterministic values. They are designed in their two common forms − Las Vegas and Monte Carlo.

(i) Las Vegas − The Las Vegas method of randomized algorithms never gives incorrect outputs, making the time constraint as the random variable. For example, in string matching algorithms, las vegas algorithms start from the beginning once they encounter

an error. This increases the probability of correctness. Eg., Randomized Quick Sort Algorithm.

(ii) **Monte Carlo** − The Monte Carlo method of randomized algorithms focuses on finishing the execution within the given time constraint. Therefore, the running time of this method is deterministic. For example, in string matching, if monte carlo encounters an error, it restarts the algorithm from the same point. Thus, saving time. Eg., Karger's Minimum Cut Algorithm

5.3 Need for Randomized Algorithms

This approach is usually adopted to reduce the time complexity and space complexity. But there might be some ambiguity about how adding randomness will decrease the runtime and memory stored, instead of increasing; we will understand that using the game theory.

5.3.1 The Game Theory and Randomized Algorithms

The basic idea of game theory actually provides with few models that help understand how decision-makers in a game interact with each other. These game theoretical models use assumptions to figure out the decision-making structure of the players in a game. The popular assumptions made by these theoretical models are that the players are both rational and take into account what the opponent player would decide to do in a particular situation of a game. We will apply this theory on randomized algorithms.

5.3.2 Zero-sum game

The zero-sum game is a mathematical representation of the game theory. It has two players where the result is a gain for one player while it is an equivalent loss to the other player. So, the net improvement is the sum of both players' status which sums up to zero.

Randomized algorithms are based on the zero-sum game of designing an algorithm that gives lowest time complexity for all inputs. There are two players in the game; one designs the algorithm and the opponent provides with inputs for the algorithm. The player two needs to give the input in such a way that it will yield the worst time complexity for them to win the game. Whereas, the player one needs to design an algorithm that takes minimum time to execute any input given.

For example, consider the quick sort algorithm where the main algorithm starts from selecting the pivot element. But, if the player in zero-sum game chooses the sorted list as an input, the standard algorithm provides the worst case time complexity. Therefore, randomizing the pivot selection would execute the algorithm faster than the worst time complexity. However, even if the algorithm chose the first element as pivot randomly and obtains the worst time complexity, executing it another time with the same input will solve the problem since it chooses another pivot this time.

On the other hand, for algorithms like merge sort the time complexity does not depend on the input; even if the algorithm is randomized the time complexity will always remain the same. Hence, **randomization is only applied on algorithms whose complexity depends on the input**.

Example: Randomized Quick Sort

Quicksort is a popular sorting algorithm that chooses a pivot element and sorts the input list around that pivot element. Randomized quick sort is designed to decrease the chances of the algorithm being executed in the worst case time complexity of $O(n^2)$. The worst case time complexity of quick sort arises when the input given is an already sorted list, leading to n(n – 1) comparisons. There are two ways to randomize the quicksort −

- **Randomly shuffling the inputs**: Randomization is done on the input list so that the sorted input is jumbled again which reduces the time complexity. However, this is not usually performed in the randomized quick sort.
- **Randomly choosing the pivot element**: Making the pivot element a random variable is commonly used method in the randomized quick sort. Here, even if the input is sorted, the pivot is chosen randomly so the worst case time complexity is avoided.

Example 1: The Randomized Quick Sort Algorithm

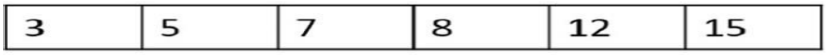
The algorithm exactly follows the standard algorithm except it randomizes the pivot selection.

Pseudocode:

```
partition-left(arr[], low, high)
  pivot = arr[high]
  i = low // place for swapping
  for j := low to high – 1 do
    if arr[j] <= pivot then
      swap arr[i] with arr[j]
      i = i + 1
  swap arr[i] with arr[high]
  return i

partition-right(arr[], low, high)
  r = Random Number from low to high
  Swap arr[r] and arr[high]
  return partition-left(arr, low, high)

quicksort(arr[], low, high)
  if low < high
    p = partition-right(arr, low, high)
    quicksort(arr, low , p-1)
    quicksort(arr, p+1, high)
```

Let us look at an example to understand how randomized quicksort works in avoiding the worst case time complexity. Since, we are designing randomized algorithms to decrease the occurrence of worst cases in time complexity let's take a sorted list as an input for this example.

The sorted input list is 3, 5, 7, 8, 12, 15. We need to apply the quick sort algorithm to sort the list.



Step 1:
Considering the worst case possible, if the random pivot chosen is also the highest index number, it compares all the other numbers and another pivot is selected.

Since 15 is greater than all the other numbers in the list, it won't be swapped, and another pivot is chosen.

Step 2:
This time, if the random pivot function chooses 7 as the pivot number –

Now the pivot divides the list into half so standard quick sort is carried out usually. However, the time complexity is decreased than the worst case.

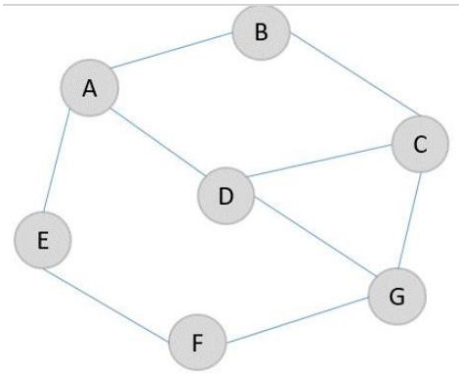
It is to be noted that the worst case time complexity of the quick sort will always remain $O(n^2)$ but with randomizations we are decreasing the occurrences of that worst case.

Example 2: Karger's Minimum Cut

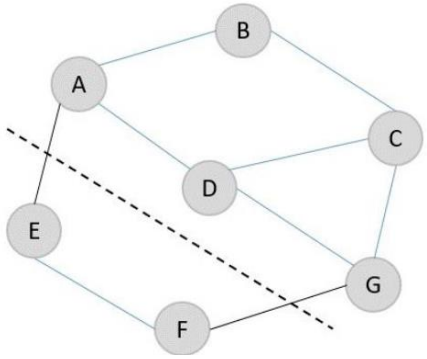
Considering the real-world applications like image segmentation where objects that are focused by the camera need to be removed from the image. Here, each pixel is considered as a node and the capacity between these pixels is reduced. The algorithm that is followed is the minimum cut algorithm.

Minimum Cut is the removal of minimum number of edges in a graph (directed or undirected) such that the graph is divided into multiple separate graphs or disjoint set of vertices.

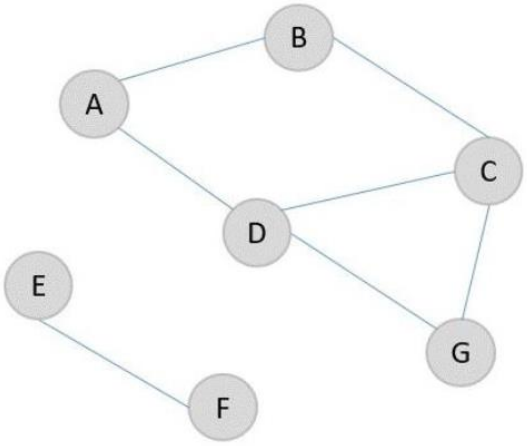
Let us look at an example for a clearer understanding of disjoint sets achieved



Edges {A, E} and {F, G} are the only ones loosely bonded to be removed easily from the graph. Hence, the minimum cut for the graph would be 2.



The resultant graphs after removing the edges A → E and F → G are {A, B, C, D, G} and {E, F}.



Karger's Minimum Cut algorithm is a randomized algorithm to find the minimum cut of a graph. It uses the monte carlo approach so it is expected to run within a time constraint and have a minimal error in achieving output. However, if the algorithm is executed multiple times the probability of the error is reduced. The graph used in karger's minimum cut algorithm is undirected graph with no weights.

The Karger's Minimum Cut Algorithm

The Karger's algorithm merges any two nodes in the graph into one node which is known as a supernode. The edge between the two nodes is contracted and the other edges connecting other adjacent vertices can be attached to the supernode.

The Algorithm

Step 1 − Choose any random edge [u, v] from the graph G to be contracted.

Step 2 − Merge the vertices to form a supernode and connect the edges of the other adjacent nodes of the vertices to the supernode formed. Remove the self-nodes, if any.

Step 3 − Repeat the process until there's only two nodes left in the contracted graph.

Step 4 − The edges connecting these two nodes are the minimum cut edges.

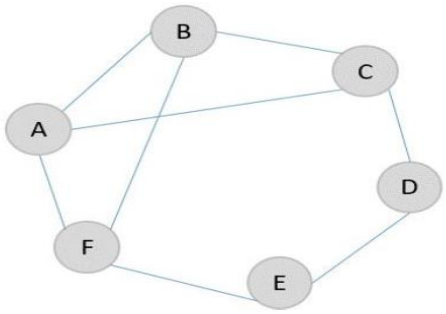
The algorithm does not always the give the optimal output so the process is repeated multiple times to decrease the probability of error.

Pseudocode:

```
Kargers_MinCut(edge, V, E):
  v = V
  while(v > 2):
    i=Random integer in the range [0, E-1]
    s1=find(edge[i].u)
    s2=find(edge[i].v)
    if(s1 != s2):
      v = v-1
      union(u, v)
  mincut=0
  for(i in the range 0 to E-1):
    s1=find(edge[i].u)
    s2=find(edge[i].v)
    if(s1 != s2):
      mincut = mincut + 1
  return mincut
```

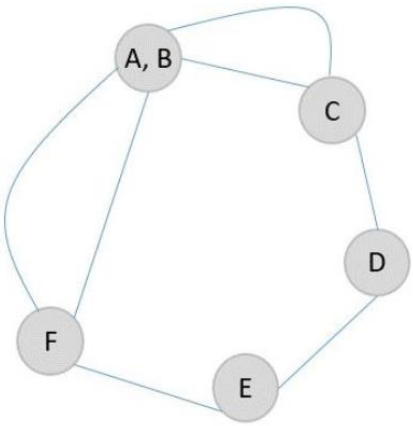
Example

Applying the algorithm on an undirected unweighted graph G {V, E} where V and E are sets of vertices and edges present in the graph, let us find the minimum cut −



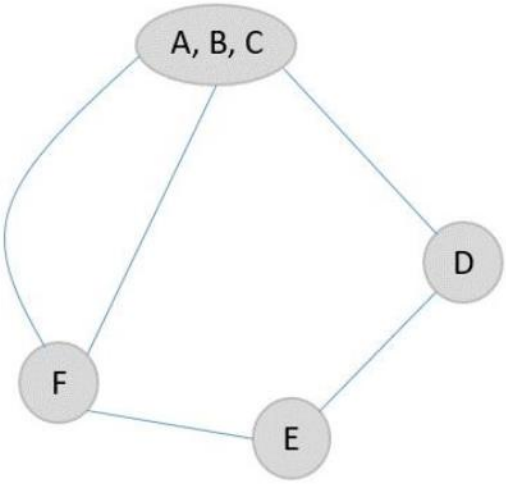
Step 1

Choose any edge, say A → B, and contract the edge by merging the two vertices into one supernode. Connect the adjacent vertex edges to the supernode. Remove the self loops, if any.



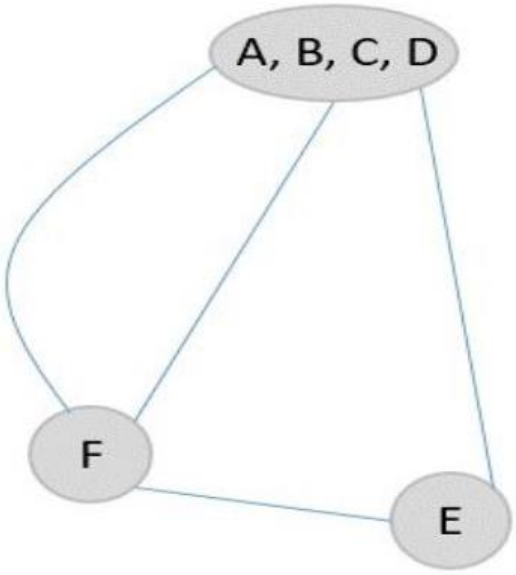
Step 2

Contract another edge (A, B) → C, so the supernode will become (A, B, C) and the adjacent edges are connected to the newly formed bigger supernode.



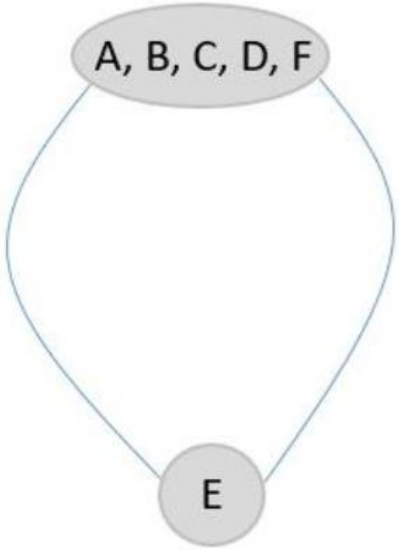
Step 3

The node D only has one edge connected to the supernode and one adjacent edge so it will be easier to contract and connect the adjacent edge to the new supernode formed.



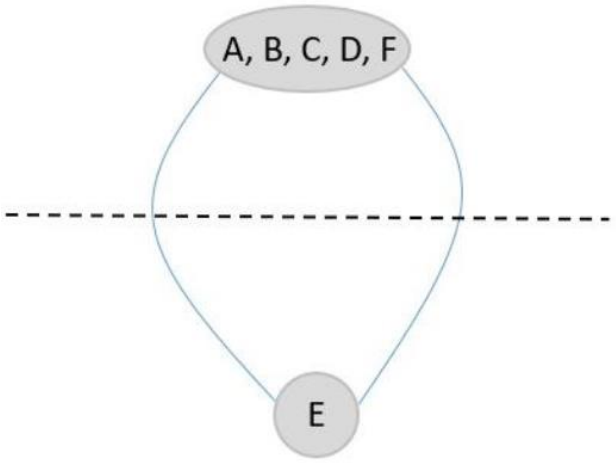
Step 4

Among F and E vertices, F is more strongly bonded to the supernode, so the edges connecting F and (A, B, C, D) are contracted.



Step 5

Since there are only two nodes present in the graph, the number of edges are the final minimum cut of the graph. In this case, the minimum cut of given graph is **2**.



The minimum cut of the original graph is 2 (E → D and E → F).

Session Summary

In this Session, you have been introduced to Randomized Algorithms which are different from Deterministic Algorithms. Two examples are used to illustrate the practical working principle of Randomized Algorithm design. Now answer the following questions.

Self-Assessment Questions

1. Explain Randomized Algorithm as a technique for designing algorithms
2. Describe why Quicksort Algorithm is a form of Randomized Algorithm

Study Session 6: Algorithm Design Techniques: Memoization

Expected Duration: 1 week or 2 contact hours

Introduction
Memoization is used to speed up computer programs by eliminating the repetitive computation of results, and by avoiding repeated calls to functions that process the same input. In this Session, memoization technique shall be explained.

Learning Outcomes
When you have studied this session, you should be able to explain:

6.1 What is memoization?

The term "**Memoization**" comes from the Latin word "**memorandum**" (**to remember**), which is commonly shortened to "memo" in American English, and which means "to transform the results of a function into something to remember."

In computing, memoization is used to speed up computer programs by eliminating the repetitive computation of results, and by avoiding repeated calls to functions that process the same input.

6.2 Why is Memoization used?

Memoization is a specific form of *caching* that is used in *dynamic programming*. The purpose of caching is to improve the performance of our programs and keep data accessible that can be used later. It basically stores the previously calculated result of the subproblem and uses the stored result for the same subproblem. This removes the extra effort to calculate again and again for the same problem. And we already know that if the same problem occurs again and again, then that problem is recursive in nature.

6.3 Where to use Memoization?

We can use the memoization technique where the **use of the previously-calculated results** comes into the picture. This kind of problem is mostly used in the context of recursion, especially with problems that involve overlapping_subproblems. Let's take an example where the same subproblem repeats again and again.

Example to show where to use memoization: Let us try to find the factorial of a number.

Below is a **recursive method** for finding the factorial of a number:

```
int factorial(unsigned int n)
{
  if (n == 0)
     return 1;
  return n * factorial(n – 1);
}
```

What happens if we use this recursive method?

If you write the complete code for the above snippet, you will notice that there will be 2 methods in the code:

1. factorial(n)
2. main()

Now if we have multiple queries to find the factorial, such as finding factorial of 2, 3, 9, and 5, then we will need to call the factorial() method 4 times:

factorial(2)
factorial(3)
factorial(9)
factorial(5)

Recursive method to find Factorial

So it is safe to say that for finding factorial of numbers K numbers, the time complexity needed will be **O(N*K)**

- **O(N)** to find the factorial of a particular number, and
- **O(K)** to call the factorial() method K different times.

How Memoization can help with such problems?

If we notice in the above problem, while calculation factorial of 9:

- We are calculating the factorial of 2
- We are also calculating the factorial of 3,
- and We are calculating the factorial of 5 as well

Therefore if we store the result of each individual factorial at the first time of calculation, we can easily return the factorial of any required number in just O(1) time. This process is known as Memoization.

6.4 Solution using Memoization (How does memoization work?):

If we find the factorial of 9 first and store the results of individual sub-problems, we can easily print the factorial of each input in O(1).

Therefore the time complexity to find factorial numbers using memoization will be **O(N)**

- **O(N)** to find the factorial of the largest input
- **O(1)** to print the factorial of each input.

6.5 Types of Memoization

The Implementation of memoization depends upon the changing parameters that are responsible for solving the problem. Most of the Dynamic Programming problems are solved in two ways:

1. **Tabulation:** Bottom Up
2. **Memoization:** Top Down

One of the easier approaches to solve most of the problems in DP is to write the recursive code at first and then write the Bottom-up Tabulation Method or Top-down Memoization of the recursive function. The steps to write the DP solution of Top-down approach to any problem is to:

1. Write the recursive code
2. Memoize the return value and use it to reduce recursive calls.

There are various dimensions of caching that are used in memoization technique, Below are some of them:

- 1D Memoization: The recursive function that has only one argument whose value was not constant after every function call.
- 2D Memoization: The recursive function that has only two arguments whose value was not constant after every function call.
- 3D Memoization: The recursive function that has only three arguments whose values were not constant after every function call.

6.5.1 1-D Memoization

The first step will be to write the recursive code. In the program below, a program related to recursion where only one parameter changes its value has been shown. Since only one parameter is non-constant, this method is known as 1-D memoization. E.g., the Fibonacci series problem to find the N-th term in the Fibonacci series.

```
// C++ program to find the Nth term
// of Fibonacci series
#include <bits/stdc++.h>
using namespace std;

// Fibonacci Series using Recursion
int fib(int n)
{
```

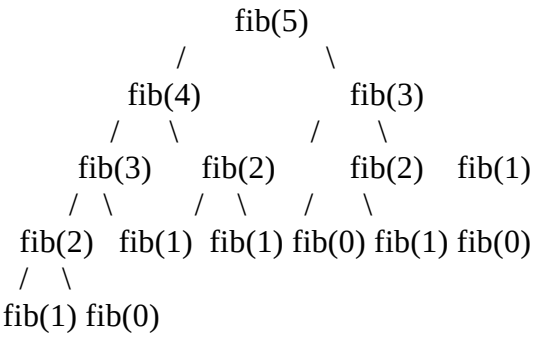
```
    // Base case
    if (n <= 1)
        return n;

    // recursive calls
    return fib(n - 1) + fib(n - 2);
}

// Driver Code
int main()
{
    int n = 6;
    printf("%d", fib(n));
    return 0;
}
```

Output:
8

Time Complexity: O(2^n)

As at every stage we need to take 2 function calls and the height of the tree will be of the order of n.

Auxiliary Space: O(n)
The extra space is used due to recursion call stack.
A common observation is that this implementation does a lot of **repeated work** (see the following recursion tree). So this will consume a lot of time for finding the N-th Fibonacci number if done.



In the above tree fib(3), fib(2), fib(1), fib(0) all are called more than once.

In the program below, the steps to write a **Top-Down approach program** have been explained. Some modifications in the recursive program will reduce the complexity of the

program and give the desired result. If fib(x) has not occurred previously, then we store the value of **fib(x) in an array term at index x and return term[x]**.
By memoizing the return value of fib(x) at index x of an array, reduce the number of recursive calls at the next step when fib(x) has already been called. So without doing further recursive calls to compute the value of fib(x), **return term[x] when fib(x)** has already been computed previously to avoid a lot of repeated work as shown in the tree.

Given below is the memoized recursive code to find the N-th term.

```
// CPP program to find the Nth term
// of Fibonacci series
#include <bits/stdc++.h>
using namespace std;
int term[1000];
// Fibonacci Series using memoized Recursion
int fib(int n)
{
   // base case
   if (n <= 1)
      return n;

   // if fib(n) has already been computed
   // we do not do further recursive calls
   // and hence reduce the number of repeated
   // work
   if (term[n] != 0)
      return term[n];

   else {

      // store the computed value of fib(n)
      // in an array term at index n to
      // so that it does not needs to be
      // precomputed again
      term[n] = fib(n - 1) + fib(n - 2);

      return term[n];
   }
}

// Driver Code
int main()
{
   int n = 6;
   printf("%d", fib(n));
```

```
    return 0;
}
```

Output:
8

Time Complexity: O(n)
As we have to calculate values for all function calls just once and there will be n values of arguments so the complexity will reduce to O(n).

Auxiliary Space: O(n)
The extra space is used due to recursion call stack.

If the recursive code has been written once, then memoization is just modifying the recursive program and storing the return values to avoid repetitive calls of functions that have been computed previously.

6.5.2 2-D Memoization

In the above program, the recursive function had only **one argument whose value was not constant** after every function call. Below, an implementation where the recursive program has two non-constant arguments has been shown.

For example, Program to solve the standard Dynamic Problem LCS_problem when two strings are given. The general recursive solution of the problem is to generate all subsequences of both given sequences and find the longest matching subsequence. The total possible combinations will be 2^n. Hence, the recursive solution will take **$O(2^n)$**.

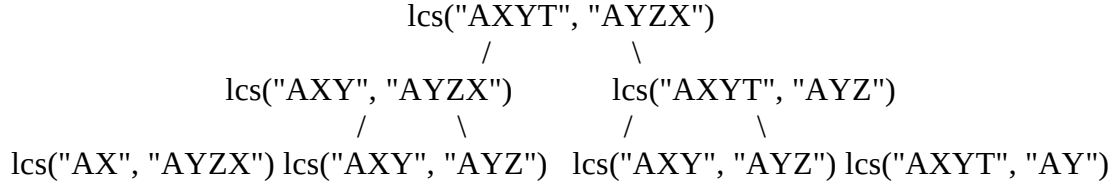
Given below is the recursive solution to the LCS problem:

```
// A Naive recursive implementation of LCS problem
#include <bits/stdc++.h>

int max(int a, int b);

// Returns length of LCS for X[0..m-1], Y[0..n-1]
int lcs(char* X, char* Y, int m, int n)
{
   if (m == 0 || n == 0)
      return 0;
   if (X[m - 1] == Y[n - 1])
      return 1 + lcs(X, Y, m - 1, n - 1);
   else
      return max(lcs(X, Y, m, n - 1),
```

```
            lcs(X, Y, m - 1, n));
}

// Utility function to get max of 2 integers
int max(int a, int b)
{
   return (a > b) ? a : b;
}

// Driver Code
int main()
{
   char X[] = "AGGTAB";
   char Y[] = "GXTXAYB";

   int m = strlen(X);
   int n = strlen(Y);

   printf("Length of LCS is %dn", lcs(X, Y, m, n));

   return 0;
}
```

Output:
Length of LCS is 4

Considering the above implementation, the following is a partial recursion tree for input strings “AXYT” and “AYZX”



In the above partial recursion tree, **lcs(“AXY”, “AYZ”) is being solved twice**. On drawing the complete recursion tree, it has been observed that there are many subproblems that are solved again and again. So this problem has Overlapping Substructure property and re-computation of same subproblems can be avoided by either using Memoization or Tabulation.

A common point of observation to use memoization in the recursive code will be the **two non-constant arguments M and N** in every function call. The function has 4 arguments, but 2 arguments are constant which does not affect the Memoization. The repetitive calls occur for N and M which have been called previously. So use a 2-D array to store the computed **lcs(m, n) value at arr[m-1][n-1]** as the string index starts from 0.

Whenever the function with the same argument m and n are called again, we do not perform any further recursive call and return arr[m-1][n-1] as the previous computation of the lcs(m, n) has already been stored in arr[m-1][n-1], hence reducing the recursive calls that happen more than once.

Below is the implementation of the Memoization approach of the recursive code.

```
// C++ program to memoize
// recursive implementation of LCS problem
#include <bits/stdc++.h>
int arr[1000][1000];
int max(int a, int b);

// Returns length of LCS for X[0..m-1], Y[0..n-1] */
// memoization applied in recursive solution
int lcs(char* X, char* Y, int m, int n)
{
   // base case
   if (m == 0 || n == 0)
      return 0;

   // if the same state has already been
   // computed
   if (arr[m - 1][n - 1] != -1)
      return arr[m - 1][n - 1];

   // if equal, then we store the value of the
   // function call
   if (X[m - 1] == Y[n - 1]) {

      // store it in arr to avoid further repetitive
      // work in future function calls
      arr[m - 1][n - 1] = 1 + lcs(X, Y, m - 1, n - 1);
      return arr[m - 1][n - 1];
   }
   else {
      // store it in arr to avoid further repetitive
      // work in future function calls
      arr[m - 1][n - 1] = max(lcs(X, Y, m, n - 1),
                              lcs(X, Y, m - 1, n));
      return arr[m - 1][n - 1];
   }
}

// Utility function to get max of 2 integers
```

```
int max(int a, int b)
{
   return (a > b) ? a : b;
}

// Driver Code
int main()
{
   memset(arr, -1, sizeof(arr));
   char X[] = "AGGTAB";
   char Y[] = "GXTXAYB";

   int m = strlen(X);
   int n = strlen(Y);

   printf("Length of LCS is %d", lcs(X, Y, m, n));

   return 0;
}
```

Output:
Length of LCS is 4

6.5.3 3-D Memoization

In the above program, the recursive function had only **two** arguments whose values were not constant after every function call. Below, an implementation where the recursive program has **three non-constant arguments** is done.

For example, Program to solve the standard Dynamic Problem LCS problem for three strings. The general recursive solution of the problem is to generate all subsequences of both given sequences and find the longest matching subsequence. The total possible combinations will be 3^n. Hence, a recursive solution will take **$O(3^n)$**.

Given below is the recursive solution to the LCS problem:

```
// A recursive implementation of LCS problem
// of three strings
#include <bits/stdc++.h>
int max(int a, int b);

// Returns length of LCS for X[0..m-1], Y[0..n-1]
int lcs(char* X, char* Y, char* Z, int m, int n, int o)
{
```

```
    // base case
    if (m == 0 || n == 0 || o == 0)
        return 0;

    // if equal, then check for next combination
    if (X[m - 1] == Y[n - 1] and Y[n - 1] == Z[o - 1]) {

        // recursive call
        return 1 + lcs(X, Y, Z, m - 1, n - 1, o - 1);
    }
    else {

        // return the maximum of the three other
        // possible states in recursion
        return max(lcs(X, Y, Z, m, n - 1, o),
                   max(lcs(X, Y, Z, m - 1, n, o),
                       lcs(X, Y, Z, m, n, o - 1)));
    }
}

// Utility function to get max of 2 integers
int max(int a, int b)
{
    return (a > b) ? a : b;
}

// Driver Code
int main()
{

    char X[] = "geeks";
    char Y[] = "geeksfor";
    char Z[] = "geeksforge";
    int m = strlen(X);
    int n = strlen(Y);
    int o = strlen(Z);
    printf("Length of LCS is %d", lcs(X, Y, Z, m, n, o));

    return 0;
}
```

Output:

Length of LCS is 5

On drawing the recursion tree completely, it has been noticed that there are many overlapping sub-problems which are been calculated multiple times. Since the function parameter has three

non-constant parameters, hence a 3-D array will be used to memorize the value that was returned when **lcs(x, y, z, m, n, o)** for any value of m, n, and o was called so that if **lcs(x, y, z, m, n, o)** is again called for the same value of m, n, and o then the function will return the already stored value as it has been computed previously in the recursive call.

arr[m][n][o] stores the value returned by the **lcs(x, y, z, m, n, o)** function call. The only modification that needs to be done in the recursive program is to store the return value of (m, n, o) state of the recursive function. The rest remains the same in the above recursive program.

Below is the implementation of the Memoization approach of the recursive code:

```
// A memoize recursive implementation of LCS problem
#include <bits/stdc++.h>
int arr[100][100][100];
int max(int a, int b);

// Returns length of LCS for X[0..m-1], Y[0..n-1] */
// memoization applied in recursive solution
int lcs(char* X, char* Y, char* Z, int m, int n, int o)
{
   // base case
   if (m == 0 || n == 0 || o == 0)
      return 0;

   // if the same state has already been
   // computed
   if (arr[m - 1][n - 1][o - 1] != -1)
      return arr[m - 1][n - 1][o - 1];

   // if equal, then we store the value of the
   // function call
   if (X[m - 1] == Y[n - 1] and Y[n - 1] == Z[o - 1]) {

      // store it in arr to avoid further repetitive work
      // in future function calls
      arr[m - 1][n - 1][o - 1] = 1 + lcs(X, Y, Z, m - 1,
                                         n - 1, o - 1);
      return arr[m - 1][n - 1][o - 1];
   }
   else {

      // store it in arr to avoid further repetitive work
      // in future function calls
      arr[m - 1][n - 1][o - 1] =
```

```
                        max(lcs(X, Y, Z, m, n - 1, o),
                          max(lcs(X, Y, Z, m - 1, n, o),
                            lcs(X, Y, Z, m, n, o - 1)));
      return arr[m - 1][n - 1][o - 1];
   }
}

// Utility function to get max of 2 integers
int max(int a, int b)
{
   return (a > b) ? a : b;
}

// Driver Code
int main()
{
   memset(arr, -1, sizeof(arr));
   char X[] = "geeks";
   char Y[] = "geeksfor";
   char Z[] = "geeksforgeeks";
   int m = strlen(X);
   int n = strlen(Y);
   int o = strlen(Z);
   printf("Length of LCS is %d", lcs(X, Y, Z, m, n, o));

   return 0;
}
```

Output:
Length of LCS is 5

Note: The array used to Memoize is initialized to some value (say -1) before the function call to mark if the function with the same parameters has been previously called or not.

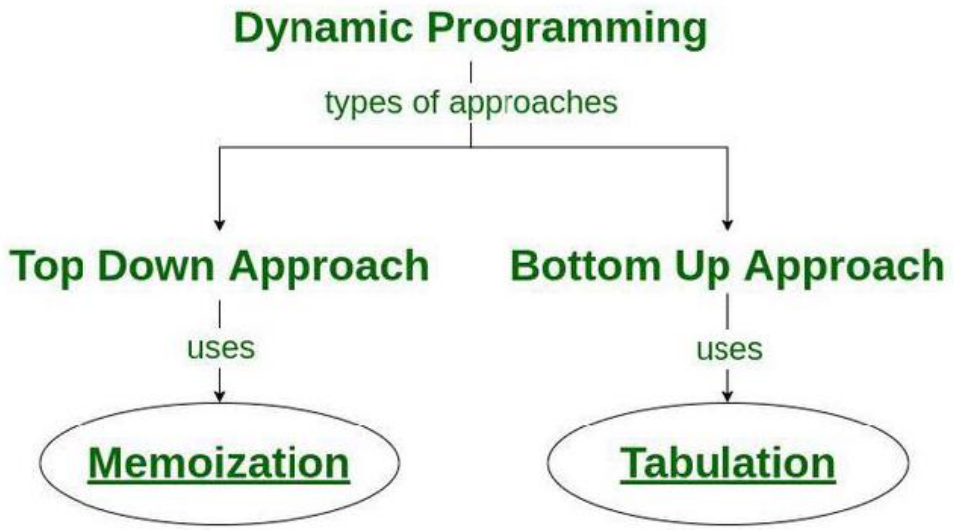
6.6 How Memoization Technique is Used in Dynamic Programming?

Dynamic programming helps to efficiently solve problems that have overlapping subproblems and optimal substructure properties. The idea behind dynamic programming is to break the problem into smaller sub-problems and save the result for future use, thus eliminating the need to compute the result repeatedly.

There are two approaches to formulate a dynamic programming solution:

1. **Top-Down Approach:** *This approach follows the* ***memoization*** *technique*. It consists of **recursion** and **caching**. In computation, recursion represents the process of calling functions repeatedly, whereas cache refers to the process of storing intermediate results.

2. **Bottom-Up Approach:** *This approach uses the* ***tabulation*** *technique* to implement the dynamic programming solution. It addresses the same problems as before, but without recursion. In this approach, iteration replaces recursion. Hence, there is no stack overflow error or overhead of recursive procedures.



How Memoization technique is used in Dynamic Programming

6.7 How Memoization is different from Tabulation?

Tabulation vs Memoization

	Tabulation	**Memoization**
State	State Transition relation is difficult to think	State Transition relation is easy to think
Code	Code get complicated when lot of conditions are required	Code is easy and less complicated
Speed	Fast, as we directly access previous states from the table	Slow due to lot of recursive calls and return statements
Subproblem solving	If all subproblems must be solved at least once, a bottom-up dynamic programming algorithm usually outperforms a top-down memorized algorithm by a constant factor	If some subproblems in the subproblem space need not be solved at all, the memorized solution has the advantage of solving only those subproblems that are definitely required
Table Entries	In Tabulated version, starting from the first entry, all entries are filled one by one	Unlike the tabulated version, all entries of the look-up table are not necessarily filled in Memoized version. The Table is filled on demand

6.8 Frequently asked questions (FAQs) about Memoization

1: Is memoization better than DP?
Memoization is the top-down approach to solving a problem with dynamic programming. It's called memoization because we will create a memo for the values returned from solving each problem.

2: Is memoization the same as caching?
Memoization is actually a specific type of caching. The term caching can generally refer to any storing technique (like HTTP caching) for future use, but memoizing refers more specifically to caching function that returns the value.

3: Why memoization is top-down?
The top-Down approach breaks the large problem into multiple subproblems. if the subproblem is solved already then reuse the answer. Otherwise, Solve the subproblem and store the result in some memory.

4: Does memoization use recursion?
Memoization follows top-down approach to solving the problem. It consists of recursion and caching. In computation, recursion represents the process of calling functions repeatedly, whereas cache refers to the process of storing intermediate results.

5: Should I use tabulation or memoization?
For problems requiring all subproblems to be solved, tabulation typically outperforms memoization by a constant factor. This is because the tabulation has no overhead of recursion which reduces the time for resolving the recursion call stack from the stack memory. Whenever a subproblem needs to be solved for the original problem to be solved, memoization is preferable since a subproblem is solved lazily, i.e. only the computations that are required are carried out.

6: Where is memoization used?
Memoization is an optimization technique used to speed up computer programs by caching the results of expensive function calls and returning them when the same inputs are encountered again.

7: Why is it called memoization?
The term "memoization" comes from the Latin word "memorandum" ("to remember"), which is commonly shortened to "memo" in American English, and which means "to transform the results of a function into something to remember.".

8: How does memoization reduce time complexity?
Solving the same problem again and again takes time and increases the run-time complexity of the overall program. This problem can be resolved by maintaining some cache or memory where we will store the already calculated result of the problem for some particular input. So that if we don't want to recalculate the same problem, we can simply use the result that is stored in the memory and reduce the time complexity.

9: What is the difference between memoization and caching?
Memoization is actually a specific type of caching that involves caching the return value of a function based on input. Caching is a more general term. For example, HTTP caching is caching but it is not memoization.

10: Why tabulation is faster than memoization?
Tabulation is usually faster than memoization, because it is iterative and solving subproblems requires no overhead of recursive calls. Memoization is a technique used in computer science to speed up the execution of recursive or computationally expensive functions by caching the results of function calls and returning the cached results when the same inputs occur again.

The basic idea of memoization is to store the output of a function for a given set of inputs and return the cached result if the function is called again with the same inputs. This technique can save computation time, especially for functions that are called frequently or have a high time complexity.

6.9 Step-by-Step Guide to Implementing Memoization

Here's a step-by-step guide to implementing memoization:

- Identify the function that you want to optimize using memoization. This function should have repeated and expensive computations for the same input.
- Create a dictionary object to store the cached results of the function. The keys of the dictionary should be the input parameters to the function, and the values should be the computed results.
- At the beginning of the function, check if the input parameters are already present in the cache dictionary. If they are, return the cached result. If the input parameters are not in the cache dictionary, compute the result and store it in the cache dictionary with the input parameters as the key.
- Return the computed result.

Here's a Python code example of implementing memoization using a dictionary:

- Python3

```
def fibonacci(n, cache={}):
    if n in cache:
        return cache[n]
    if n == 0:
        result = 0
    elif n == 1:
        result = 1
```

```
    else:
        result = fibonacci(n-1) + fibonacci(n-2)
    cache[n] = result
    return result
```

In the above code, we define a function called fibonacci that calculates the nth number in the Fibonacci sequence. We use a dictionary object called cache to store the results of the function calls. If the input parameter n is already present in the cache dictionary, we return the cached result. Otherwise, we compute the result using recursive calls to the fibonacci function and store it in the cache dictionary before returning it.

Memoization can be used to optimize the performance of many functions that have repeated and expensive computations. By caching the results of function calls, we can avoid unnecessary computations and speed up the execution of the function.

6.10 Memoization using decorators in Python

Recursion is a programming technique where a function calls itself repeatedly till a termination condition is met. Some of the examples where recursion is used are calculation of fibonacci series, factorial, etc. But the issue with them is that in the recursion tree, there can be chances that the sub-problem that is already solved is being solved again, which adds to overhead.
Memoization is a technique of recording the intermediate results so that it can be used to avoid repeated calculations and speed up the programs. It can be used to optimize the programs that use recursion. In Python, memoization can be done with the help of function decorators.
Let us take the example of calculating the factorial of a number. The simple program below uses recursion to solve the problem:

```
# Simple recursive program to find factorial using Python Language
def facto(num):
    if num == 1:
        return 1
    else:
        return num * facto(num-1)


print(facto(5))
print(facto(5)) # again performing same calculation
```

Output

```
120
120
```

The above program can be optimized by memoization using decorators.

```
# Factorial program with memoization using
# decorators.

# A decorator function for function 'f' passed
# as parameter
memory = {}
def memoize_factorial(f):

    # This inner function has access to memory
    # and 'f'
    def inner(num):
        if num not in memory:
            memory[num] = f(num)
            print('result saved in memory')
        else:
            print('returning result from saved memory')
        return memory[num]

    return inner

@memoize_factorial
def facto(num):
    if num == 1:
        return 1
    else:
        return num * facto(num-1)

print(facto(5))
print(facto(5)) # directly coming from saved memory
```

Output

```
result saved in memory
result saved in memory
result saved in memory
result saved in memory
result saved in memory
120
returning result from saved memory
120
```

Explanation:

1. A function called *memoize_factoria*l has been defined. Its main purpose is to store the intermediate results in the variable called memory.

2. The second function called *facto* is the function to calculate the factorial. It has been

annotated by a decorator (the function memoize_factorial). The *facto* has access to the *memory* variable as a result of the concept of closures. The annotation is equivalent to writing,
facto = memoize_factorial(facto)

3. When facto(5) is called, the recursive operations take place in addition to the storage of intermediate results. Every time a calculation needs to be done, it is checked if the result is available in *memory*. If yes, then it is used, else, the value is calculated and is stored in *memory*.

4. We can verify the fact that memoization actually works, please see the output of this program.

Session Summary

Memoization is a programming concept and can be applied to any programming language. Its absolute goal is to optimize the program. Usually, this problem is seen when programs perform heavy computations. This technique cache all the previous result that is computed so that it will not have to recalculate for the same problem.

Self-Assessment Questions

1. Explain the meaning of Memoization
2. Discuss the three types of Memoization

Study Session 7: Introduction to Amortized Analysis

Expected Duration: 1 week or 2 contact hours

Introduction

In computer science, **amortized analysis** is a method for analyzing a given algorithm's complexity, or how much of a resource, especially time or memory, it takes to execute. The motivation for amortized analysis is that looking at the worst-case run time can be too pessimistic. Instead, amortized analysis averages the running times of operations in a sequence over that sequence. In this Session, amortized analysis is discussed.

Learning Outcomes

When you have studied this session, you should be able to explain:

7.1 Meaning of Amortized Analysis
7.2 Methods for Performing Amortized Analysis
7.3 Amortized analysis of the push operation for a dynamic array
7.4 Advantages of Amortized Analysis
7.5 Disadvantages of amortized analysis

7.1 Meaning of Amortized Analysis

Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst-case average time that is lower than the worst-case time of a particularly expensive operation. The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets, and Splay Trees.

Amortized analysis is a technique used in computer science to analyze the average-case time complexity of algorithms that perform a sequence of operations, where some operations may be more expensive than others. The idea is to spread the cost of these expensive operations over multiple operations, so that the average cost of each operation is constant or less.

For a given operation of an algorithm, certain situations (e.g., input parametrizations or data structure contents) may imply a significant cost in resources, whereas other situations may not be as costly. The amortized analysis considers both the costly and less costly operations together over the whole sequence of operations. This may include accounting for different types of input, length of the input, and other factors that affect its performance.

For example, consider the dynamic array data structure that can grow or shrink dynamically as elements are added or removed. The cost of growing the array is proportional to the size of the array, which can be expensive. However, if we amortize the cost of growing the array over several insertions, the average cost of each insertion becomes constant or less.

Amortized analysis provides a useful way to analyze algorithms that perform a sequence of operations where some operations are more expensive than others, as it provides a guaranteed upper bound on the average time complexity of each operation, rather than the worst-case time complexity.

Amortized analysis is a method used in computer science to analyze the average performance of an algorithm over multiple operations. Instead of analyzing the worst-case time complexity of an algorithm, which gives an upper bound on the running time of a single operation, amortized analysis provides an average-case analysis of the algorithm by considering the cost of several operations performed over time.

The key idea behind amortized analysis is to spread the cost of an expensive operation over several operations. For example, consider a dynamic array data structure that is resized when it runs out of space. The cost of resizing the array is expensive, but it can be amortized over several insertions into the array, so that the average time complexity of an insertion operation is constant.

Amortized analysis is useful for designing efficient algorithms for data structures such as dynamic arrays, priority queues, and disjoint-set data structures. It provides a guarantee that the average-case time complexity of an operation is constant, even if some operations may be expensive.

Summarily,

In computer science and algorithms, amortized analysis is a technique used to estimate the average time complexity of an algorithm over a sequence of operations, rather than the worst-case complexity of individual operations. It allows us to make more accurate predictions about the overall efficiency of an algorithm, especially in cases where some operations may take longer than others.

1. The basic idea behind amortized analysis is to distribute the cost of expensive operations over a sequence of less expensive operations. For example, suppose we have an algorithm that occasionally performs a very expensive operation that takes O(n) time, but most of the time performs an operation that takes only O(1) time. In worst-case analysis, we would say that the algorithm takes O(n) time, but in amortized analysis, we can distribute the cost of the expensive operation over a sequence of n operations, resulting in an average cost of O(1) per operation.
2. There are several techniques for performing amortized analysis, including aggregate analysis, accounting method, and potential method. In aggregate analysis, we compute the total cost of a sequence of operations and divide it by the number of operations to get the average cost per operation. In the accounting method, we assign credits to each operation and use them to pay for expensive operations. In the potential method, we assign a potential value to the data structure being operated on and use it to measure the amount of work done by each operation.

7.2 Methods for Performing Amortized Analysis

Amortized analysis requires knowledge of which series of operations are possible. This is most commonly the case with data structures, which have state that persists between operations. The basic idea is that a worst-case operation can alter the state in such a way that the worst case cannot occur again for a long time, thus "amortizing" its cost.

There are generally three methods for performing amortized analysis: the aggregate method, the accounting method, and the potential method. All of these give correct answers; the choice of which to use depends on which is most convenient for a particular situation.

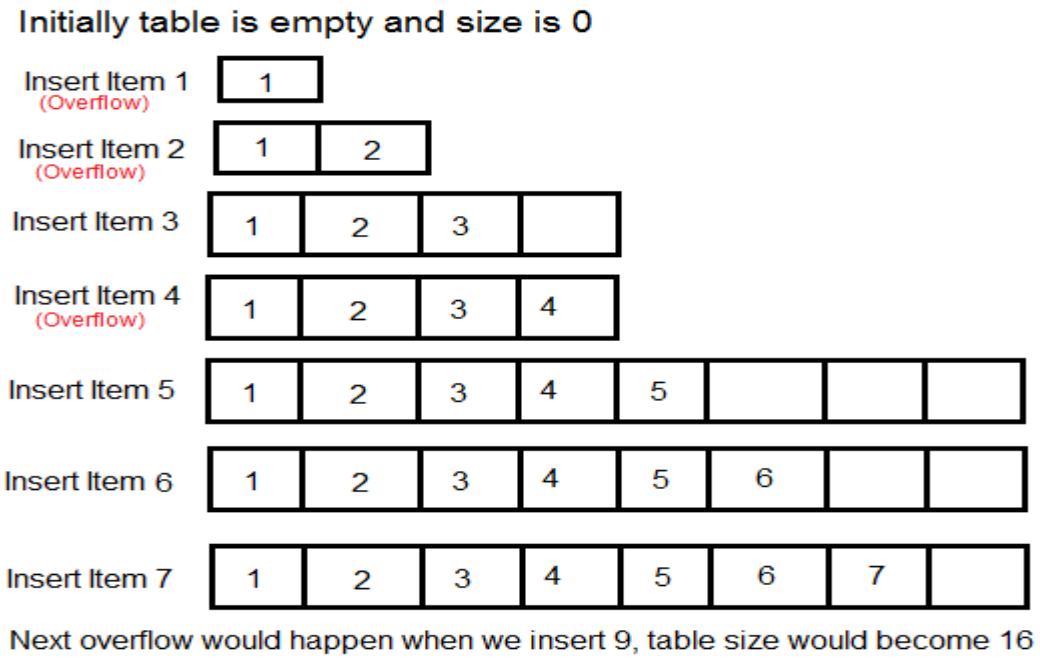
1. **Aggregate analysis** determines the upper bound $T(n)$ on the total cost of a sequence of n operations, then calculates the amortized cost to be $T(n) / n$.
2. **The accounting method** is a form of aggregate analysis which assigns to each operation an *amortized cost* which may differ from its actual cost. Early operations have an amortized cost higher than their actual cost, which accumulates a saved "credit" that pays for later operations having an amortized cost lower than their actual cost. Because the credit begins at zero, the

actual cost of a sequence of operations equals the amortized cost minus the accumulated credit. Because the credit is required to be non-negative, the amortized cost is an upper bound on the actual cost. Usually, many short-running operations accumulate such credit in small increments, while rare long-running operations decrease it drastically.

3. **The potential method** is a form of the accounting method where the saved credit is computed as a function (the "potential") of the state of the data structure. The amortized cost is the immediate cost plus the change in potential.[3]

Example 1: **Dynamic array**

Let us consider an example of simple hash table insertions. How do we decide on table size? There is a trade-off between space and time, if we make hash-table size big, search time becomes low, but the space required becomes high.



The solution to this trade-off problem is to use Dynamic Table (or Arrays). The idea is to increase the size of the table whenever it becomes full. Following are the steps to follow when the table becomes full.

1) Allocate memory for larger table size, typically twice the old table.
2) Copy the contents of the old table to a new table.
3) Free the old table.

If the table has space available, we simply insert a new item in the available space.

What is the time complexity of n insertions using the above scheme?

If we use simple analysis, the worst-case cost of insertion is O(n). Therefore, the worst-case cost of n inserts is n * O(n) which is $O(n^2)$. This analysis gives an upper bound, but not a tight upper bound for n insertions as all insertions don't take $\Theta(n)$ time.

Item No.	1	2	3	4	5	6	7	8	9	10	.....
Table Size	1	2	4	4	8	8	8	8	16	16	.....
Cost	1	2	3	1	5	1	1	1	9	1	.....

$$\text{Amortized Cost} = \frac{(1 + 2 + 3 + 5 + 1 + 1 + 9 + 1...)}{n}$$

We can simplify the above series by breaking terms 2, 3, 5, 9.. into two as (1+1), (1+2), (1+4), (1+8)

$$\text{Amortized Cost} = \frac{[\overbrace{(1 + 1 + 1 + 1...)}^{n \text{ terms}} + \overbrace{(1 + 2 + 4 + ...)}^{\lfloor Log_2(n-1) \rfloor + 1 \text{ terms}}]}{n}$$

$$<= \frac{[n + 2n]}{n}$$

$$<= 3$$

$$\text{Amortized Cost} = O(1)$$

So using Amortized Analysis, we could prove that the Dynamic Table scheme has O(1) insertion time which is a great result used in hashing. Also, the concept of the dynamic table is used in vectors in C++ and ArrayList in Java.

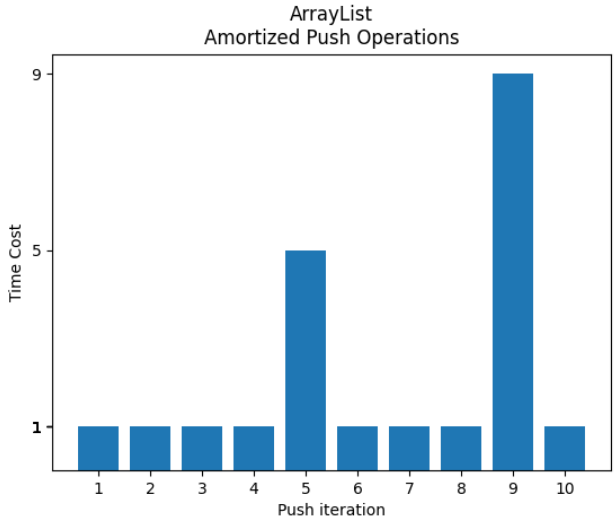
Following are a few important notes.

1) Amortized cost of a sequence of operations can be seen as expenses of a salaried person. The average monthly expense of the person is less than or equal to the salary, but the person can spend more money in a particular month by buying a car or something. In other months, he or she saves money for the expensive month.

2) The above Amortized Analysis was done for Dynamic Array example is called ***Aggregate Method***. There are two more powerful ways to do Amortized analysis called ***Accounting Method*** and ***Potential Method***.

3) The amortized analysis doesn't involve probability. There is also another different notion of average-case running time where algorithms use randomization to make them faster and the expected running time is faster than the worst-case running time. These algorithms are analyzed using Randomized Analysis. Examples of these algorithms are Randomized Quick Sort, Quick Select and Hashing. We will soon be covering Randomized analysis in a different post.

7.3 Amortized analysis of the push operation for a dynamic array



Consider a dynamic array that grows in size as more elements are added to it, such as ArrayList in Java or std::vector in C++. If we started out with a dynamic array of size 4, we could push 4 elements onto it, and each operation would take constant time. Yet pushing a fifth element onto that array would take longer as the array would have to create a new array of double the current size (8), copy the old elements onto the new array, and then add the new element. The next three push operations would similarly take constant time, and then the subsequent addition would require another slow doubling of the array size.

In general if we consider an arbitrary number of pushes $n + 1$ to an array of size n, we notice that push operations take constant time except for the last one which takes $\Theta(n)$ time to perform the size doubling operation. Since there were $n + 1$ operations total we can take the average of this and find that pushing elements onto the dynamic array takes: $[n\Theta(1) + \Theta(n)] / (n + 1) = \Theta(1)$, constant time.

Example 2: Amortized analysis of insertion in Red-Black Tree

Let us discuss the Amortized Analysis of Red-Black Tree operations (Insertion) using the Potential Method. To perform the amortized analysis of the Red-Black Tree Insertion operation, we use the Potential (or Physicist's) method. For the potential method, we define a potential function ϕ that maps a data structure to a non-negative real value. An operation can result in a change of this potential. Let us define the potential function ϕ in the following manner:

(1)

$$g(n) = \begin{cases} 0, & \text{if n is red.} \\ 1, & \text{if n is black with no red children.} \\ 0, & \text{if n is black with one red child.} \\ 2, & \text{if n is black and has two red children.} \end{cases}$$

where n is a node of Red-Black Tree

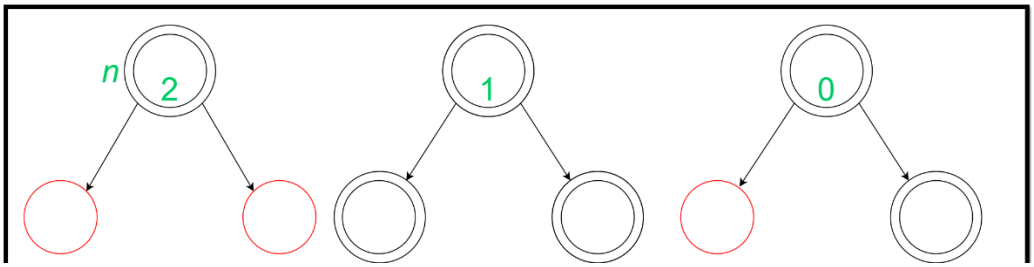
Potential function $\phi = \Sigma g(n)$, over all nodes of the red black tree.

Further, we define the amortized time of an operation as:

Amortized time= $c + \Delta\phi(h)$

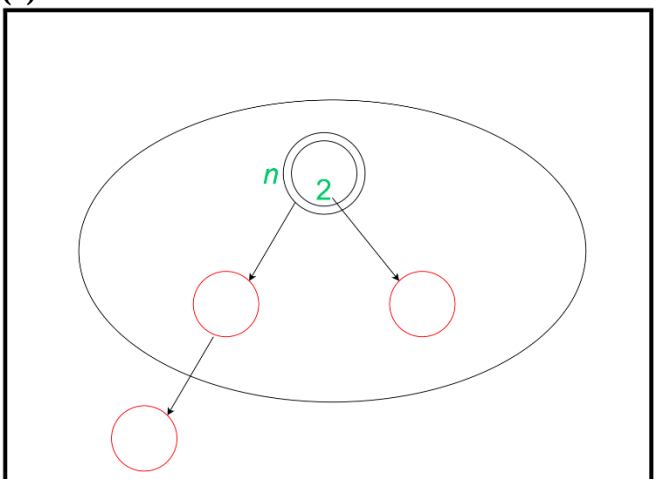
$\Delta\phi(h) = \phi(h') - \phi(h)$

where h and h' are the states of the Red-Black Tree before and after the operation respectively. c is the actual cost of the operation. The change in potential should be positive for low-cost operations and negative for high-cost operations. A new node is inserted on a leaf of a red-black tree. We have the leaves of a red-black tree of any one of the following types:

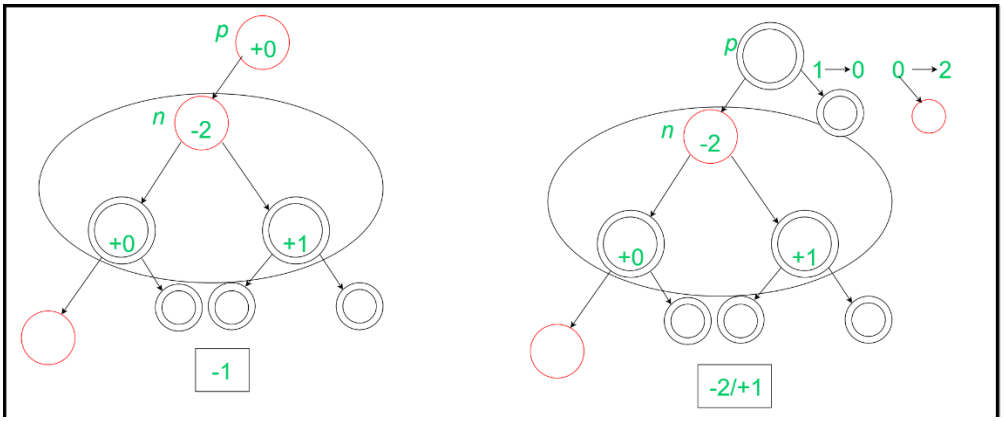


The insertions and their amortized analysis can be represented as:

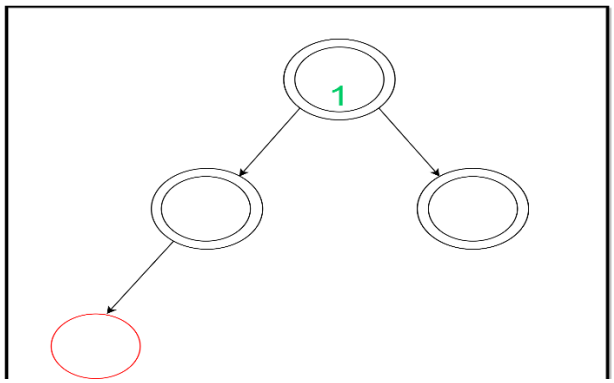
(1)



This insertion is performed by first recoloring the parent and the other sibling (red). Then the grandparent and uncle of that leaf node are considered for further recoloring which leads to the **amortized cost** to be **-1**(when the grandparent of the leaf node is red), **-2** (when uncle of the leaf is black and the grandparent is black) or **+1** (when uncle of the leaf is red and grandparent is black). The insertion can be shown as:

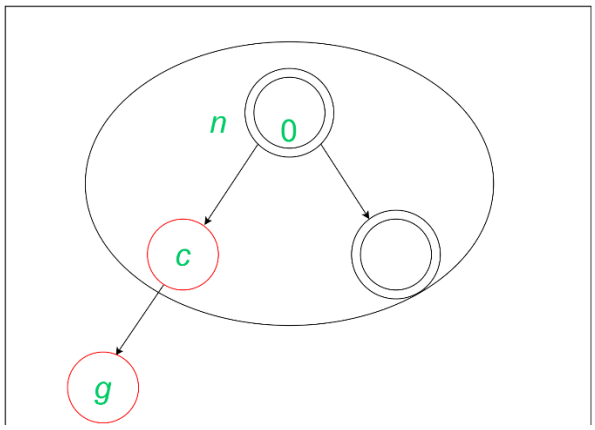


(2)



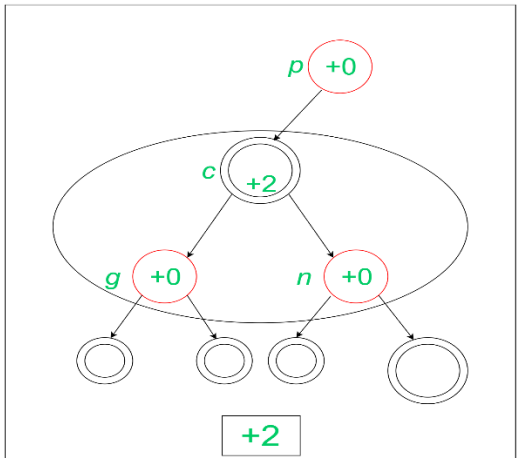
In this insertion, the node is inserted without any changes as the black depth of the leaves remains the same. This is the case when the leaf may have a **black sibling** or does **not have any sibling** (since we consider the color of the color of null node to be black). So, the **amortized cost** of this insertion is **0**.

(3)



In this insertion, we cannot recolor the leaf node, its parent, and the sibling such that the black depth stays the same as before. So, we need to perform a Left- Left rotation.

After rotation, there are no changes when the grandparent of g(the inserted node) is black. Also, for the case of the Red Grandparent of g(the inserted node), we do not have any changes. So, the insertion is completed with **amortized cost = +2**. The insertion has been depicted next:



After calculating these particular amortized costs at the leaf site of a red-black tree we can discuss the nature of the **total amortized cost** of insertion in a red-black tree. Since this may happen that two red nodes may have a parent-child relationship till the root of the red-black tree.

So in the extreme (or corner) case, we reduce the number of black nodes with two red children by 1, and we at most increase the number of black nodes with no red children by 1, leaving a net loss of at most 1 to the potential function. Since one unit of potential pays for each operation therefore

$\Delta\phi(h) \leq n$

where n is the total number of nodes. Thus, the **total amortized cost** of insertion in the Red-Black Tree is **O(n)**.

Queue Example
The following is a Python3 implementation of a Queue, a FIFO data structure:

```
# Define a class named Queue
class Queue:
  # Define the constructor method that initializes two empty lists
  def __init__(self):
    self.input = [ ] # This list will store the elements that are enqueued
    self.output = [ ] # This list will store the elements that are dequeued

  # Define a method named enqueue that takes an element as a parameter
  def enqueue(self, element):
    self.input.append(element) # Append the element to the input list

  # Define a method named dequeue that returns the first element that was enqueued
  def dequeue(self):
    if not self.output: # If the output list is empty
      while self.input: # While the input list is not empty
        self.output.append(self.input.pop()) # Pop the last element from the input list and append it to the output list

    return self.output.pop() # Pop and return the last element from the output list
```

The enqueue operation just pushes an element onto the input array; this operation does not depend on the lengths of either input or output and therefore runs in constant time. However the dequeue operation is

more complicated. If the output array already has some elements in it, then dequeue runs in constant time; otherwise, dequeue takes O(n) time to add all the elements onto the output array from the input array, where *n* is the current length of the input array. After copying *n* elements from input, we can perform *n* dequeue operations, each taking constant time, before the output array is empty again. Thus, we can perform a sequence of *n* dequeue operations in only O(n) time, which implies that the amortized time of each dequeue operation is O(1).

Alternatively, we can charge the cost of copying any item from the input array to the output array to the earlier enqueue operation for that item. This charging scheme doubles the amortized time for enqueue but reduces the amortized time for dequeue to O(1).

Amortized analysis is a powerful tool for analyzing the performance of algorithms over a sequence of operations. It allows us to make more accurate predictions about the average case complexity of an algorithm and can help us identify cases where an algorithm may perform poorly in practice, even if its worst-case complexity is low.

7.4 Advantages of Amortized Analysis

1. **More accurate predictions:** Amortized analysis provides a more accurate prediction of the average-case complexity of an algorithm over a sequence of operations, rather than just the worst-case complexity of individual operations.
2. **Provides insight into algorithm behavior**: By analyzing the amortized cost of an algorithm, we can gain insight into how it behaves over a longer period of time and how it handles different types of inputs.
3. **Helps in algorithm design**: Amortized analysis can be used as a tool for designing algorithms that are efficient over a sequence of operations.
4. **Useful in dynamic data structures**: Amortized analysis is particularly useful in dynamic data structures like heaps, stacks, and queues, where the cost of an operation may depend on the current state of the data structure.

7.5 Disadvantages of amortized analysis

1. **Complexity**: Amortized analysis can be complex, especially when multiple operations are involved, making it difficult to implement and understand.
2. **Limited applicability**: Amortized analysis may not be suitable for all types of algorithms, especially those with highly unpredictable behavior or those that depend on external factors like network latency or I/O operations.
3. **Lack of precision**: Although amortized analysis provides a more accurate prediction of average-case complexity than worst-case analysis, it may not always provide a precise estimate of the actual performance of an algorithm, especially in cases where there is high variance in the cost of operations.

Summary

We have seen in this Lecture that the motivation for amortized analysis is that looking at the worst-case run time can be too pessimistic. Instead, amortized analysis averages the running times of operations in a sequence over that sequence. Amortized analysis is a useful tool that complements other techniques such as worst-case and average-case analysis

Study Session 8: Graph Data Structure and Algorithm

Expected Duration: 1 week or 2 contact hours

Introduction

Graphs have become a powerful means of modelling and capturing data in real-world scenarios such as social media networks, web pages and links, and locations and routes in GPS. If you have a set of objects that are related to each other, then you can represent them using a graph.

Learning Outcomes

When you have studied this session, you should be able to explain:

8.1 Graph Data Structure

A graph data structure is a collection of nodes that have data and are connected to other nodes. 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. Graphs are also used in social networks like linkedIn, Facebook. On Facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, Note...anything that has data is a node. In Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender, locale etc. Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page, etc., a new edge is created for that relationship.

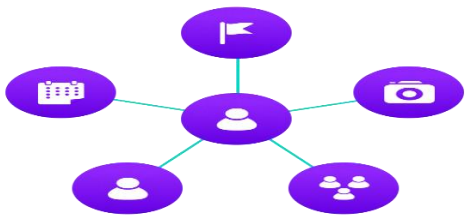


Fig. 8.1: Example of graph data structure

All of Facebook is then a collection of these nodes and edges. This is because Facebook uses a graph data structure to store its data.
More precisely, a graph is a data structure (V, E) that consists of

- A collection of vertices V
- A collection of edges E, represented as ordered pairs of vertices (u,v)

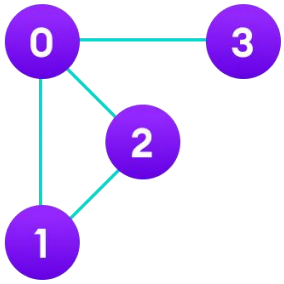


Fig. 8.2 Vertices and edges

In the graph,
V = {0, 1, 2, 3}
E = {(0,1), (0,2), (0,3), (1,2)}
G = {V, E}

8.2 Graph Terminologies

(i) A **graph** consists of a finite set of **vertices** or nodes and a set of **edges** connecting these 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 unlabelled. 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/unlabelled.

(ii) **Adjacency**: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them. Two vertices are said to be **adjacent** if they are connected to each other by the same edge.

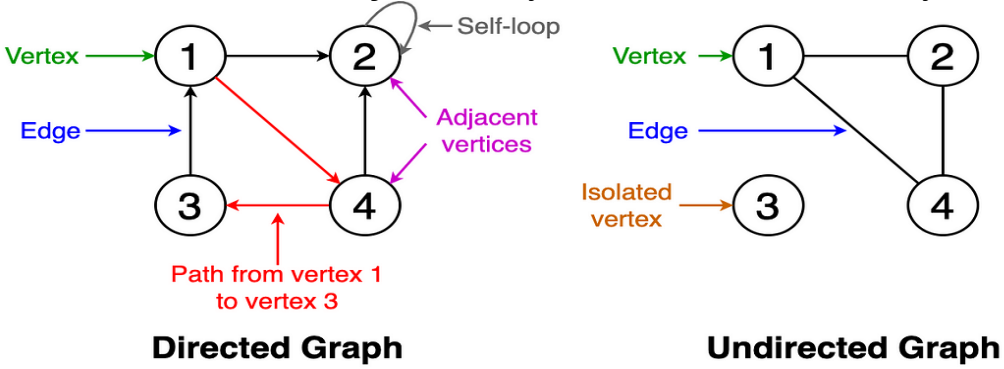


Figure 8.3: Graph Examples.

(iii) **Order:** The number of vertices in the graph
(iv) **Size:** The number of edges in the graph
(v) **Vertex degree:** The number of edges that are incident to a vertex
(vi) **Isolated vertex:** A vertex that is not connected to any other vertices in the graph
(vii) **Self-loop**: An edge from a vertex to itself
(viii) **Directed graph:** A graph where all the edges have a direction in form of arrows indicating what is the start vertex and what is the end vertex. A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.
(ix) **Undirected graph:** A graph with edges that have no direction
(x) **Weighted graph:** Edges of the graph has weights
(xi) **Unweighted graph:** Edges of the graph has no weights
(xii) **Path**: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.

8.3 Representations of Graph

There are the two most common ways to represent a graph:
1. Adjacency Matrix
2. Adjacency List

8.3.1 Adjacency Matrix

An adjacency matrix is a way of representing a graph as a matrix of booleans (0's and 1's). Let's assume there are **n** vertices in a graph So, create a 2D matrix **adjMat[n][n]** having dimension n x n.

- *If there is an edge from vertex **i** to **j**, mark **adjMat[i][j]** as **1**.*
- *If there is no edge from vertex **i** to **j**, mark **adjMat[i][j]** as **0**.*

Representation of Undirected Graph to Adjacency Matrix:

Figure 8.4 shows an undirected graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** to both cases (**adjMat[destination]** and **adjMat[destination])** because we can go either way.

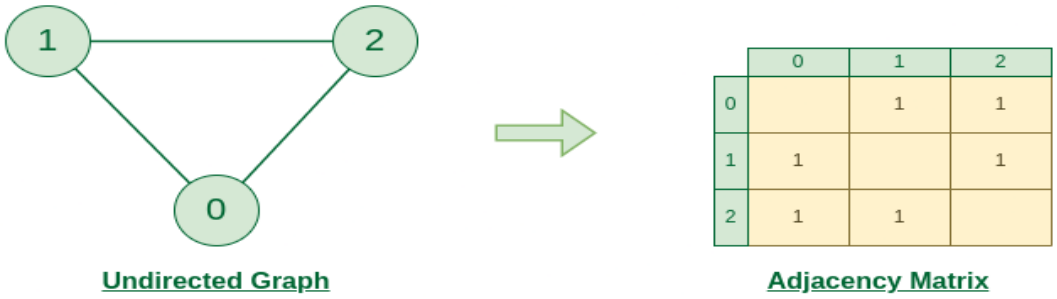


Fig. 8.4: Representation of Undirected Graph to Adjacency Matrix

Representation of Directed Graph to Adjacency Matrix:

Figure 8.5 shows a directed graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** for that particular **adjMat[destination]**.

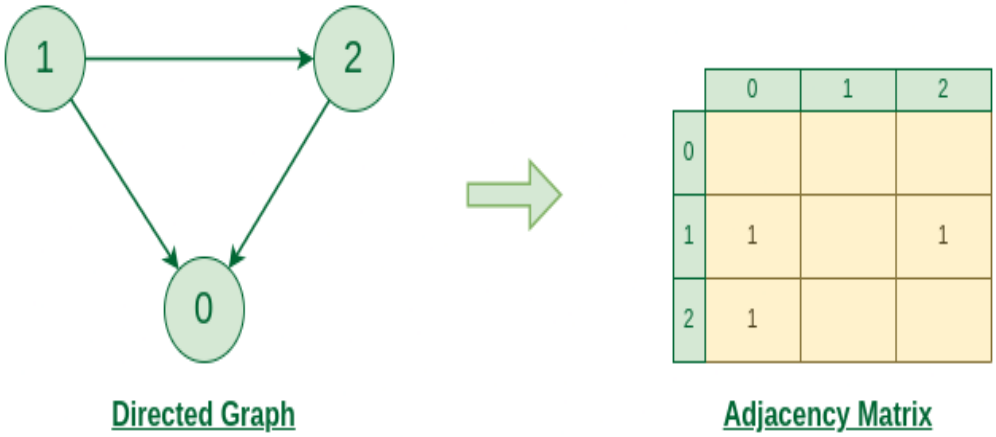


Fig. 8.5: Representation of Directed Graph to Adjacency matrix

8.3.2 Adjacency List

An array of Lists is used to store edges between two vertices. The size of array is equal to the number of **vertices (i.e. n)**. Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex **i**.

Let's assume there are **n** vertices in the graph, so, create an **array of list** of size **n** as **adjList[n].**

- *adjList[0] will have all the nodes which are connected (neighbour) to vertex **0**.*
- *adjList[1] will have all the nodes which are connected (neighbour) to vertex **1** and so on.*

Representation of Undirected Graph to Adjacency list:

Consider Fig. 8.6 an undirected graph having 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has two neighbours (i.e, 1 and 2). So, insert vertex 1 and 2 at indices 0 of array. Similarly, For vertex 1, it has two neighbour (i.e, 2 and 1) So, insert vertices 2 and 1 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.

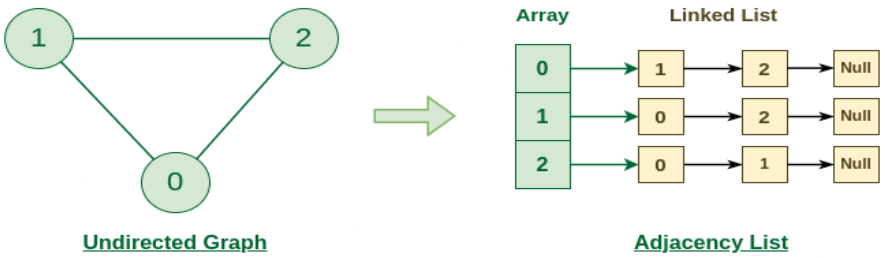


Fig. 8.6: Representation of Undirected Graph to Adjacency list

Representation of Directed Graph to Adjacency list:

Consider Fig. 8.7 is a directed graph having 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has no neighbours. For vertex 1, it has two neighbour (i.e, 0 and 2), so, insert vertices 0 and 2 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.

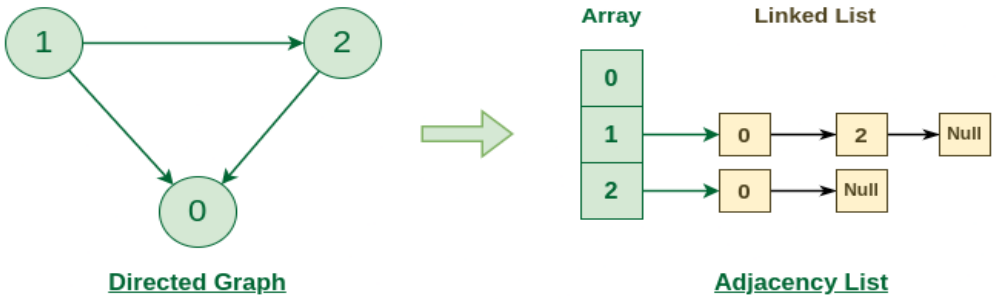


Fig. 8.7: Representation of Directed Graph to Adjacency list

8.4 Types of Graphs

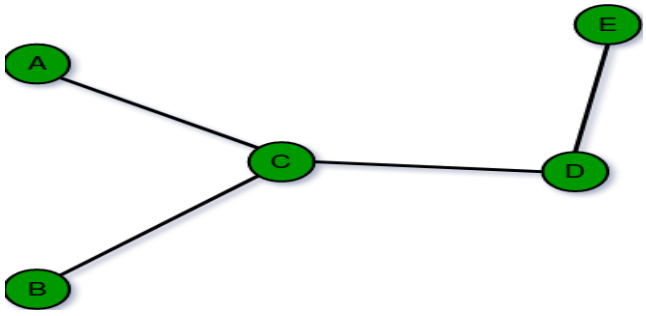
1. **Undirected Graphs**: A graph in which edges have no direction, i.e., the edges do not have arrows indicating the direction of traversal. Example: A social network graph where friendships are not directional.
2. **Directed Graphs**: A graph in which edges have a direction, i.e., the edges have arrows indicating the direction of traversal. Example: A web page graph where links between pages are directional.
3. **Weighted Graphs:** A graph in which edges have weights or costs associated with them. Example: A road network graph where the weights can represent the distance between two cities.
4. **Unweighted Graph**s: A graph in which edges have no weights or costs associated with them. Example: A social network graph where the edges represent friendships.
5. **Complete Graphs:** A graph in which each vertex is connected to every other vertex. Example: A tournament graph where every player plays against every other player.

6. **Bipartite Graphs:** A graph in which the vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to a vertex in the other set. Example: A job applicant graph where the vertices can be divided into job applicants and job openings.
7. **Trees**: A connected graph with no cycles. Example: A family tree where each person is connected to their parents.
8. **Cycles**: A graph with at least one cycle. Example: A bike-sharing graph where the cycles represent the routes that the bikes take.
9. **Sparse Graphs:** A graph with relatively few edges compared to the number of vertices. Example: A chemical reaction graph where each vertex represents a chemical compound and each edge represents a reaction between two compounds.
10. **Dense Graph**s: A graph with many edges compared to the number of vertices. Example: A social network graph where each vertex represents a person and each edge represents a friendship.

Other Types…

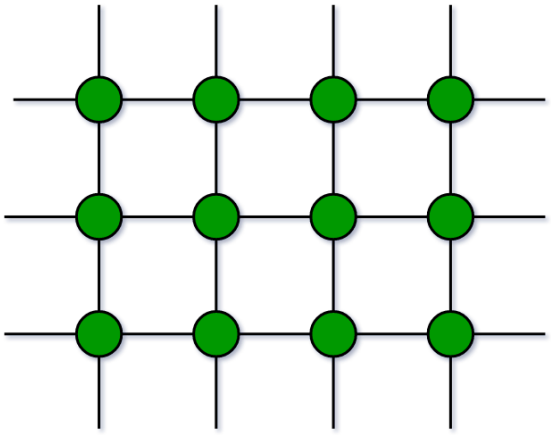
1. Finite Graphs

A graph is said to be finite if it has a finite number of vertices and a finite number of edges. A finite graph is a graph with a finite number of vertices and edges. In other words, both the number of vertices and the number of edges in a finite graph are limited and can be counted. Finite graphs are often used to model real-world situations, where there is a limited number of objects and relationships between them.

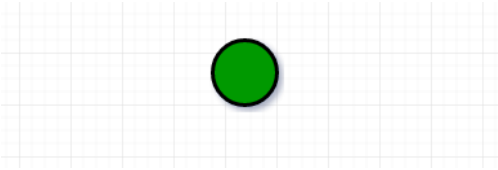


2. Infinite Graph:

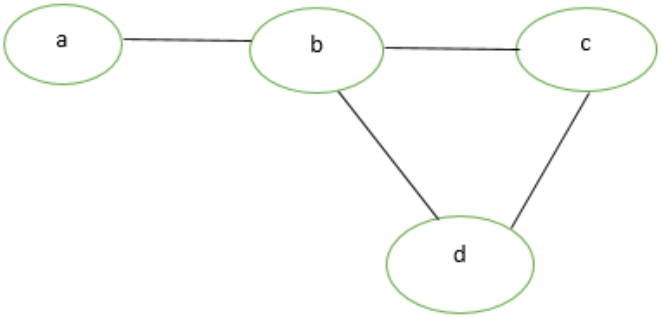
A graph is said to be infinite if it has an infinite number of vertices as well as an infinite number of edges.

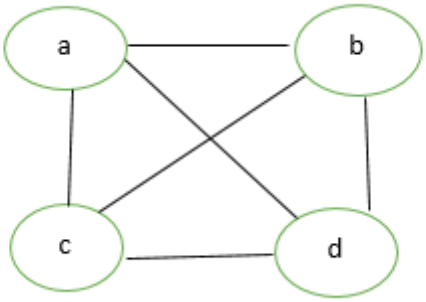


3. Trivial Graph:
A graph is said to be trivial if a finite graph contains only one vertex and no edge. A trivial graph is a graph with only one vertex and no edges. It is also known as a singleton graph or a single vertex graph. A trivial graph is the simplest type of graph and is often used as a starting point for building more complex graphs. In graph theory, trivial graphs are considered to be a degenerate case and are not typically studied in detail



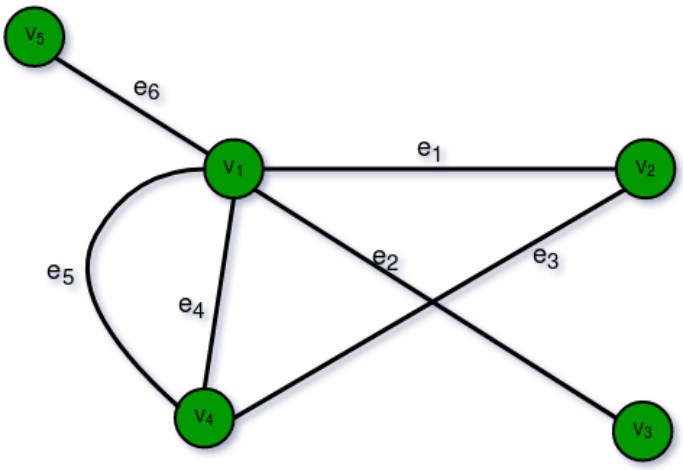
4. Simple Graph:
A simple graph is a graph that does not contain more than one edge between the pair of vertices. A simple railway track connecting different cities is an example of a simple graph.





5. Multi Graph:
Any graph which contains some parallel edges but doesn't contain any self-loop is called a multigraph. For example a Road Map.

- **Parallel Edges:** If two vertices are connected with more than one edge then such edges are called parallel edges that are many routes but one destination.
- **Loop:** An edge of a graph that starts from a vertex and ends at the same vertex is called a loop or a self-loop.

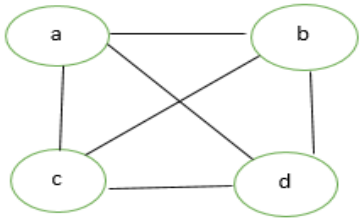


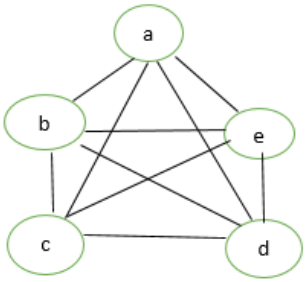
6. Null Graph:

A graph of order n and size zero is a graph where there are only isolated vertices with no edges connecting any pair of vertices. A null graph is a graph with no edges. In other words, it is a graph with only vertices and no connections between them. A null graph can also be referred to as an edgeless graph, an isolated graph, or a discrete graph

7. Complete Graph:

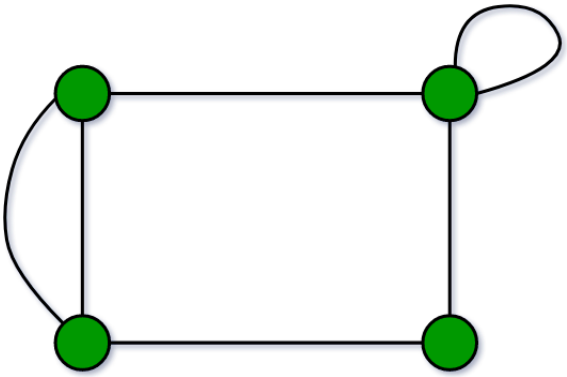
A simple graph with n vertices is called a complete graph if the degree of each vertex is n-1, that is, one vertex is attached with n-1 edges or the rest of the vertices in the graph. A complete graph is also called Full Graph.





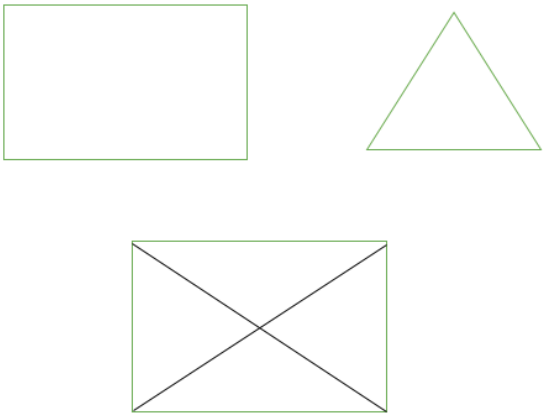
8. Pseudo Graph:

A graph G with a self-loop and some multiple edges is called a pseudo graph. A pseudograph is a type of graph that allows for the existence of loops (edges that connect a vertex to itself) and multiple edges (more than one edge connecting two vertices). In contrast, a simple graph is a graph that does not allow for loops or multiple edges.



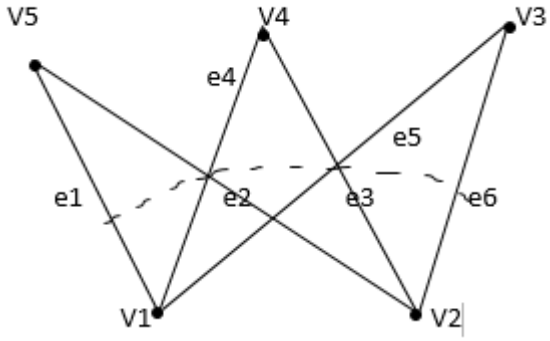
9. Regular Graph:

A simple graph is said to be regular if all vertices of graph G are of equal degree. All complete graphs are regular but vice versa is not possible. A regular graph is a type of undirected graph where every vertex has the same number of edges or neighbors. In other words, if a graph is regular, then every vertex has the same degree.

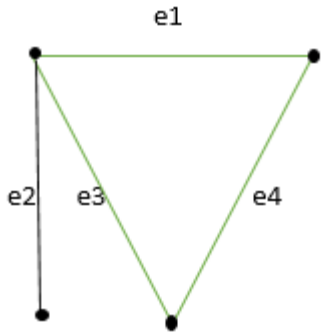


10. Bipartite Graph:

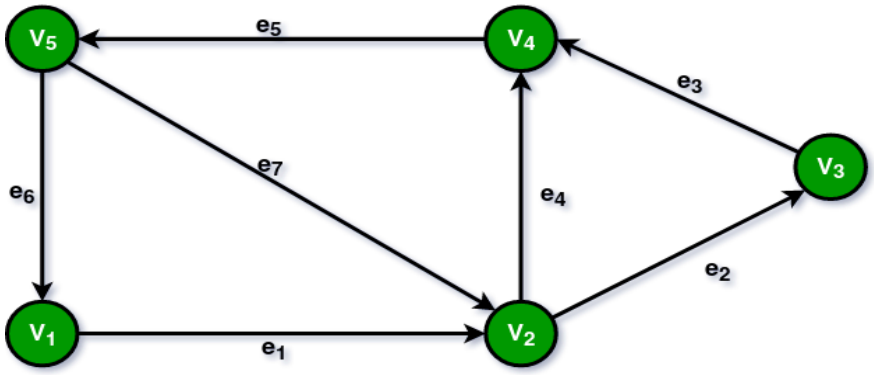
A graph G = (V, E) is said to be a bipartite graph if its vertex set V(G) can be partitioned into two non-empty disjoint subsets. V1(G) and V2(G) in such a way that each edge e of E(G) has one end in V1(G) and another end in V2(G). The partition V1 U V2 = V is called Bipartite of G. Here in the figure: V1(G)={V5, V4, V3} and V2(G)={V1, V2}



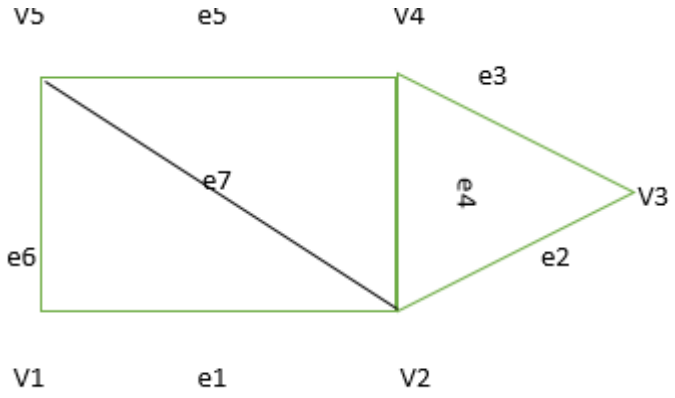
11. Labeled Graph:
If the vertices and edges of a graph are labeled with name, date, or weight then it is called a labeled graph. It is also called Weighted Graph.



12. Digraph Graph:
A graph G = (V, E) with a mapping f such that every edge maps onto some ordered pair of vertices (Vi, Vj) are called a Digraph. It is also called *Directed Graph*. The ordered pair (Vi, Vj) means an edge between Vi and Vj with an arrow directed from Vi to Vj. Here in the figure: e1 = (V1, V2) e2 = (V2, V3) e4 = (V2, V4)

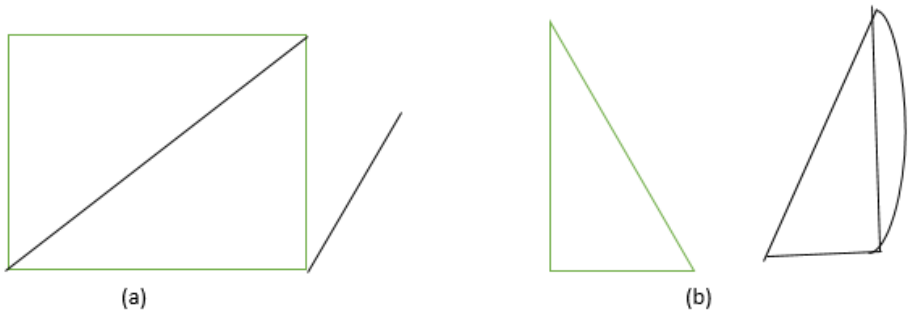


13. Subgraph:
A graph G1 = (V1, E1) is called a subgraph of a graph G(V, E) if V1(G) is a subset of V(G) and E1(G) is a subset of E(G) such that each edge of G1 has same end vertices as in G.



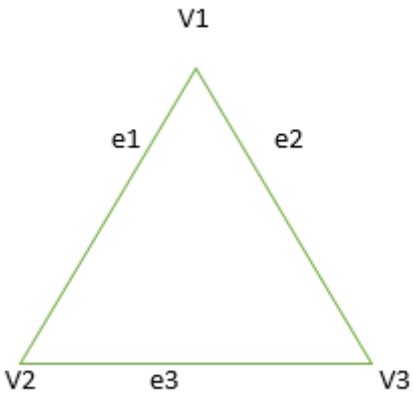
14. Connected or Disconnected Graph:

Graph G is said to be connected if any pair of vertices (Vi, Vj) of a graph G is reachable from one another. Or a graph is said to be connected if there exists at least one path between each and every pair of vertices in graph G, otherwise, it is disconnected. A null graph with n vertices is a disconnected graph consisting of n components. Each component consists of one vertex and no edge.



15. Cyclic Graph:

A graph G consisting of n vertices and n> = 3 that is V1, V2, V3- – – – Vn and edges (V1, V2), (V2, V3), (V3, V4)- – – – (Vn, V1) are called cyclic graph.



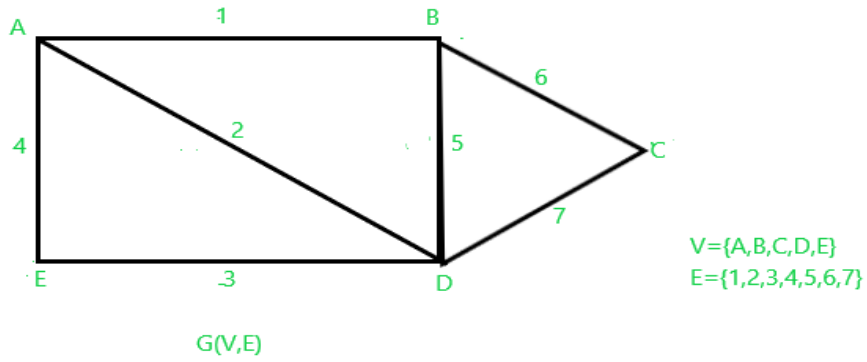
16. Types of Subgraphs:

- **Vertex disjoint subgraph:** Any two graph G1 = (V1, E1) and G2 = (V2, E2) are said to be vertex disjoint of a graph G = (V, E) if V1(G1) intersection V2(G2) = null. In the figure, there is no common vertex between G1 and G2.
- **Edge disjoint subgraph:** A subgraph is said to be edge-disjoint if E1(G1) intersection E2(G2) = null. In the figure, there is no common edge between G1 and G2.

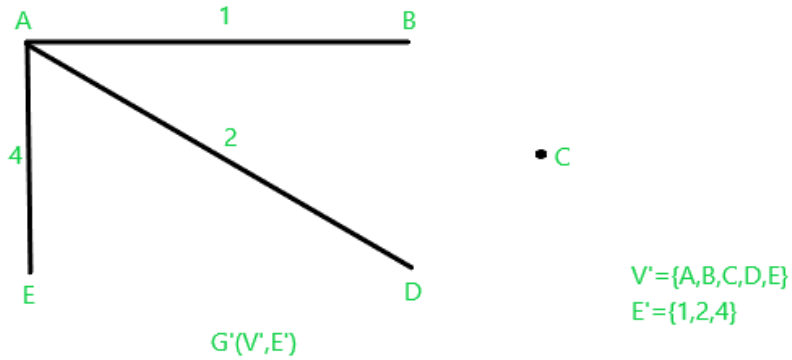
Note: Edge disjoint subgraph may have vertices in common but a vertex disjoint graph cannot have a common edge, so the vertex disjoint subgraph will always be an edge-disjoint subgraph.

17. Spanning Subgraph

Consider the graph G(V, E) as shown below. A spanning subgraph is a subgraph that contains all the vertices of the original graph G, that is, G'(V, E') is spanning if V'=V and E' is a subset of E.



So one of the spanning subgraph can be as shown below G'(V', E'). It has all the vertices of the original graph G and some of the edges of G.



This is just one of the many spanning subgraphs of graph G. We can create various other spanning subgraphs by different combinations of edges. Note that if we consider a graph G'(V',E') where V'=V and E'=E, then graph G' is a spanning subgraph of graph G(V,E).

8.5 Breadth First Search or BFS for a Graph

The **Breadth First Search (BFS)** algorithm is used to search a graph data structure for a node that meets a set of criteria. It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level.

Relation between BFS for Graph and Tree traversal:
Breadth-First Traversal (or Search) for a graph is similar to the Breadth-First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

- Visited and
- Not visited.

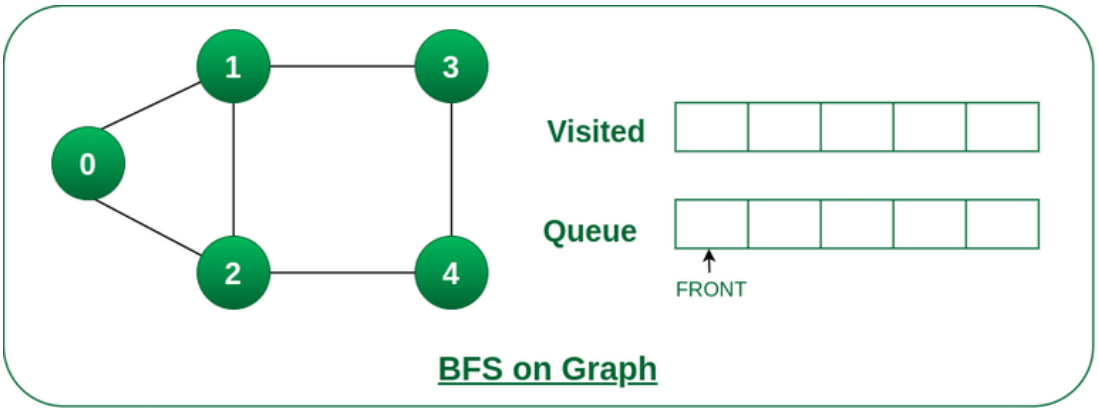
A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a **queue data structure** for traversal.

8.5.1 How does BFS work?

Starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited. To do this a queue is used. All the adjacent unvisited nodes of the current level are pushed into the queue and the nodes of the current level are marked visited and popped from the queue.

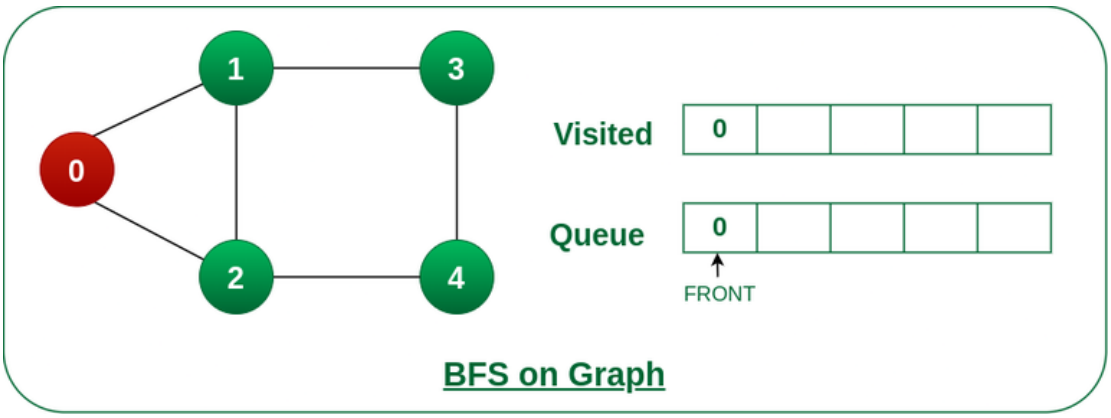
Illustration:
Let us understand the working of the algorithm with the help of the following example.

Step1: *Initially queue and visited arrays are empty.*



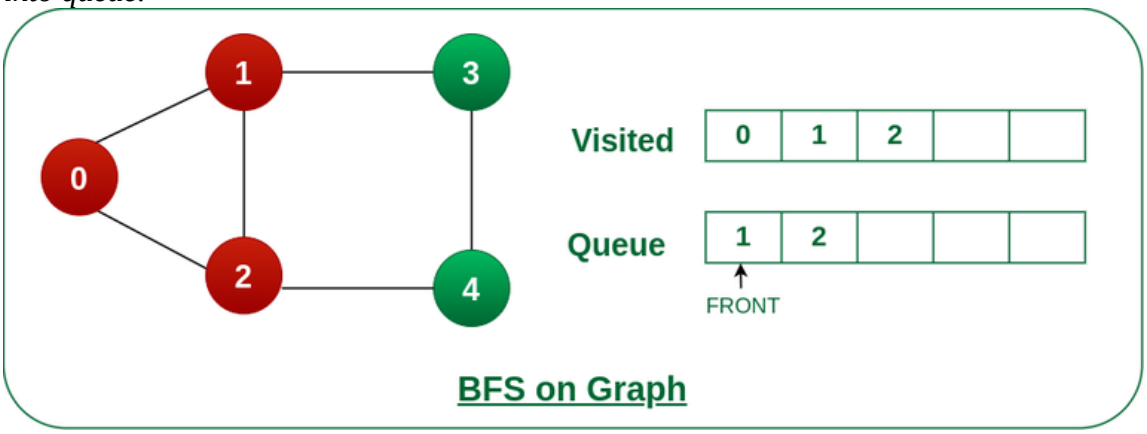
Queue and visited arrays are empty initially.

Step2: *Push node 0 into queue and mark it visited.*



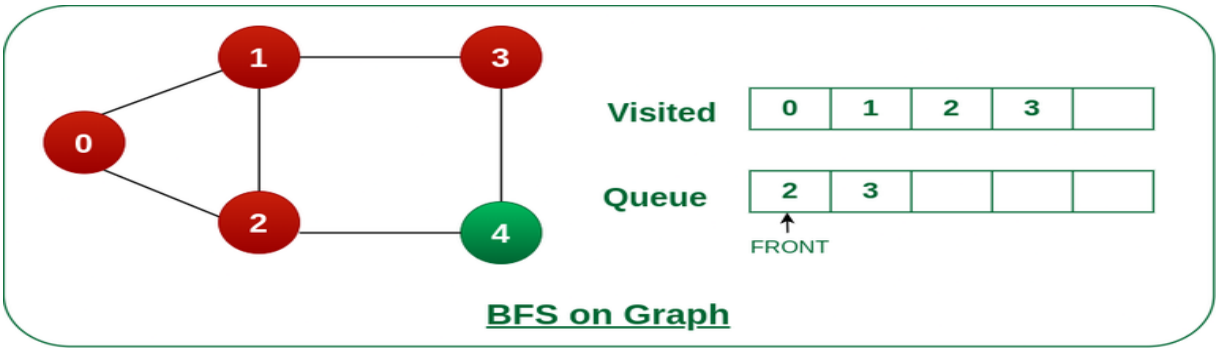
Push node 0 into queue and mark it visited.

Step 3: *Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.*



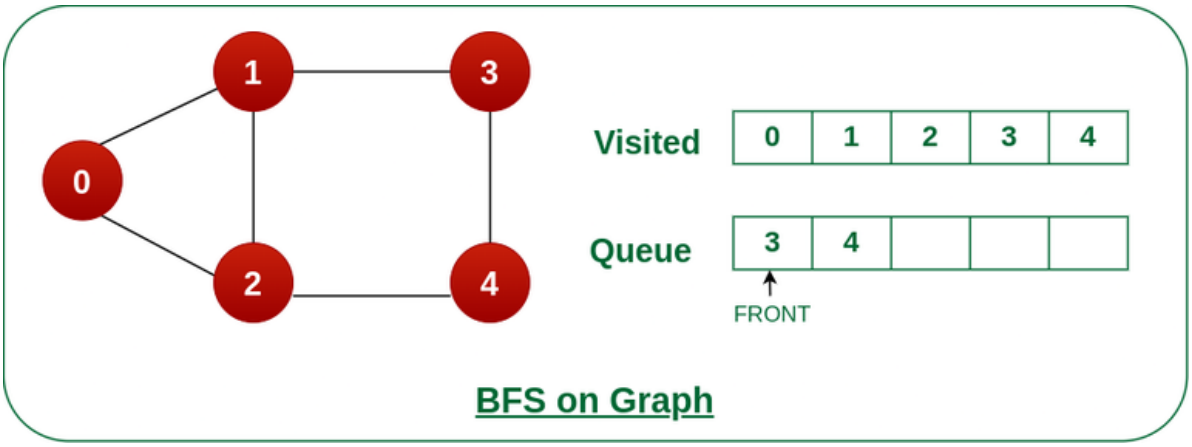
Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.

Step 4: *Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.*



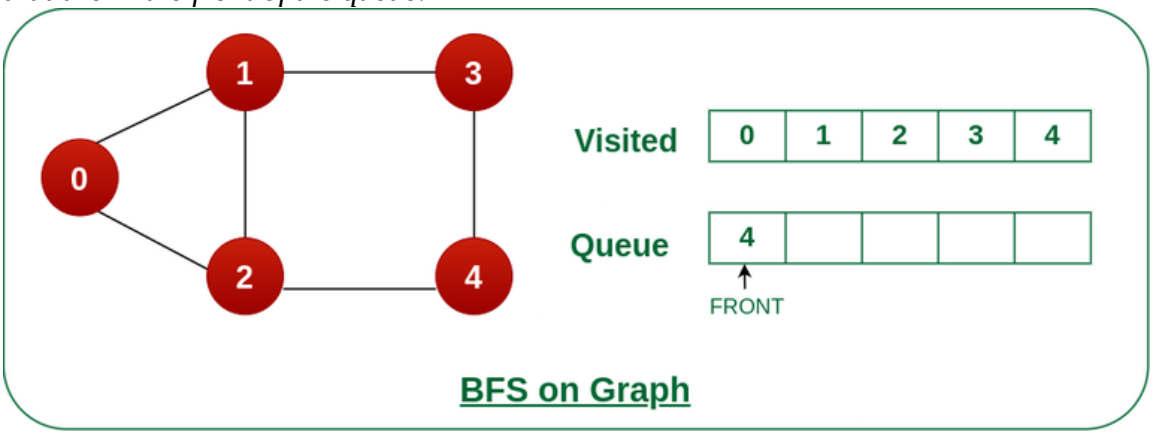
Remove node 1 from the front of queue and visited the unvisited neighbours and push

Step 5: *Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.*



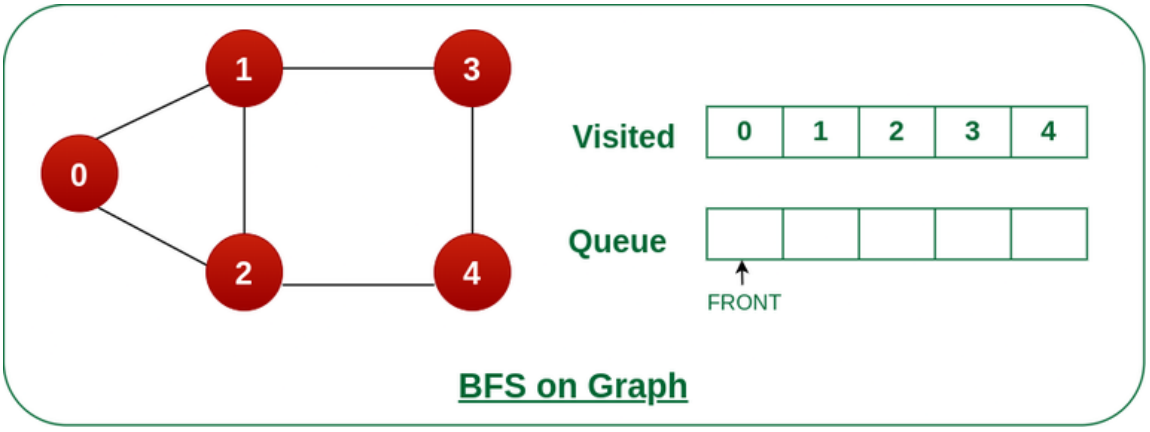
Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.

Step 6: *Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue. As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.*



Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

Steps 7: *Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue. As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.*



Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.
Now, Queue becomes empty, So, terminate these process of iteration.

Implementation of BFS for Graph using Adjacency List:

```
// C++ code to print BFS traversal from a given
// source vertex

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

// This class represents a directed graph using
// adjacency list representation
class Graph {

    // No. of vertices
    int V;

    // Pointer to an array containing adjacency lists
    vector<list<int> > adj;

public:
    // Constructor
    Graph(int V);

    // Function to add an edge to graph
    void addEdge(int v, int w);

    // Prints BFS traversal from a given source s
    void BFS(int s);
};
```

```
Graph::Graph(int V)
{
   this->V = V;
   adj.resize(V);
}

void Graph::addEdge(int v, int w)
{
   // Add w to v's list.
   adj[v].push_back(w);
}

void Graph::BFS(int s)
{
   // Mark all the vertices as not visited
   vector<bool> visited;
   visited.resize(V, false);

   // Create a queue for BFS
   list<int> queue;

   // Mark the current node as visited and enqueue it
   visited[s] = true;
   queue.push_back(s);

   while (!queue.empty()) {

      // Dequeue a vertex from queue and print it
      s = queue.front();
      cout << s << " ";
      queue.pop_front();

      // Get all adjacent vertices of the dequeued
      // vertex s.
      // If an adjacent has not been visited,
      // then mark it visited and enqueue it
      for (auto adjacent : adj[s]) {
         if (!visited[adjacent]) {
            visited[adjacent] = true;
            queue.push_back(adjacent);
         }
      }
   }
}
```

```
// Driver code
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Breadth First Traversal "
         << "(starting from vertex 2) \n";
    g.BFS(2);

    return 0;
}
```

Output

Following is Breadth First Traversal (starting from vertex 2)

2 0 3 1

Time Complexity: O(V+E), where V is the number of nodes and E is the number of edges.

Auxiliary Space: O(V)

8.6 Probable Problems related to BFS:

Some of the problems that you can be asked to solve in BFS are:

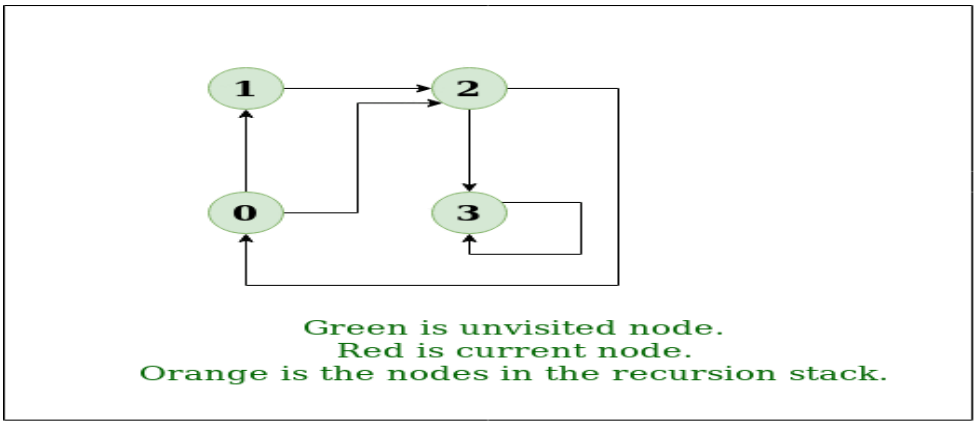
S.no	Problems
1.	Find the level of a given node in an Undirected Graph
2.	Minimize maximum adjacent difference in a path from top-left to bottom-right
3.	Minimum jump to the same value or adjacent to reach the end of an Array
4.	Maximum coin in minimum time by skipping K obstacles along the path in Matrix

S.no	Problems
5.	Check if all nodes of the Undirected Graph can be visited from the given Node
6.	Minimum time to visit all nodes of a given Graph at least once
7.	Minimize moves to the next greater element to reach the end of the Array
8.	Shortest path by removing K walls
9.	Minimum time required to infect all the nodes of the Binary tree
10.	Check if destination of given Matrix is reachable with required values of cells

8.7 Depth First Search or DFS for a Graph

Depth First Traversal (or DFS) for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal.

Example:
Input: *n = 4, e = 6*
0 -> 1, 0 -> 2, 1 -> 2, 2 -> 0, 2 -> 3, 3 -> 3
Output: *DFS from vertex 1 : 1 2 0 3*
Explanation:
DFS Diagram:

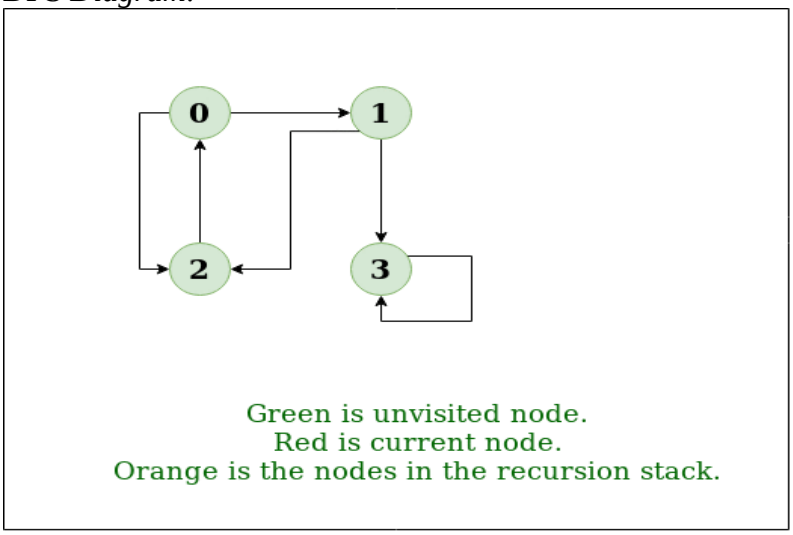


Example 1
Check: *https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/*

Input: *n = 4, e = 6*
2 -> 0, 0 -> 2, 1 -> 2, 0 -> 1, 3 -> 3, 1 -> 3

Output: *DFS from vertex 2 : 2 0 1 3*

Explanation:
DFS Diagram:

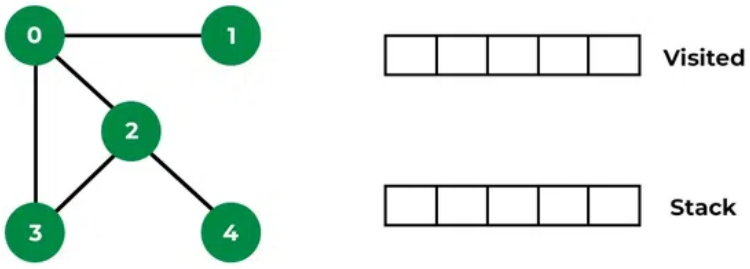


Example 2

8.7.1 How does DFS work?

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. Let us understand the working of **Depth First Search** with the help of the following illustration:

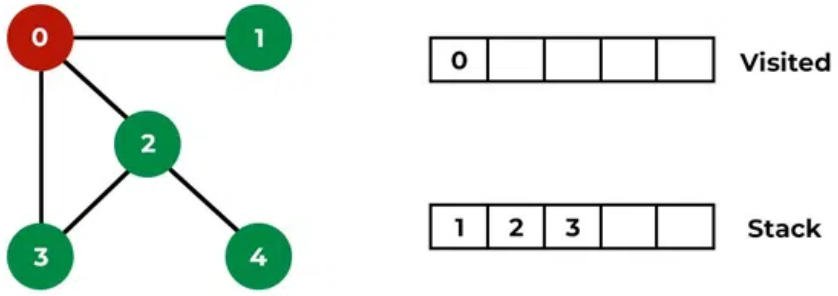
Step1: *Initially stack and visited arrays are empty.*



DFS on Graph

Stack and visited arrays are empty initially.

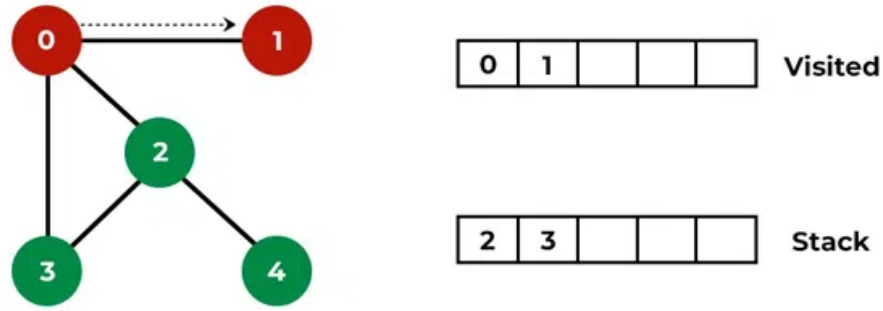
Step 2: *Visit 0 and put its adjacent nodes which are not visited yet into the stack.*



DFS on Graph

Visit node 0 and put its adjacent nodes (1, 2, 3) into the stack

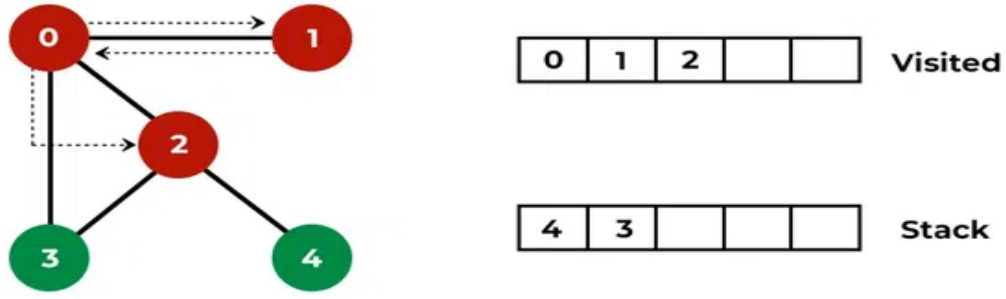
Step 3: *Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.*



DFS on Graph

Visit node 1

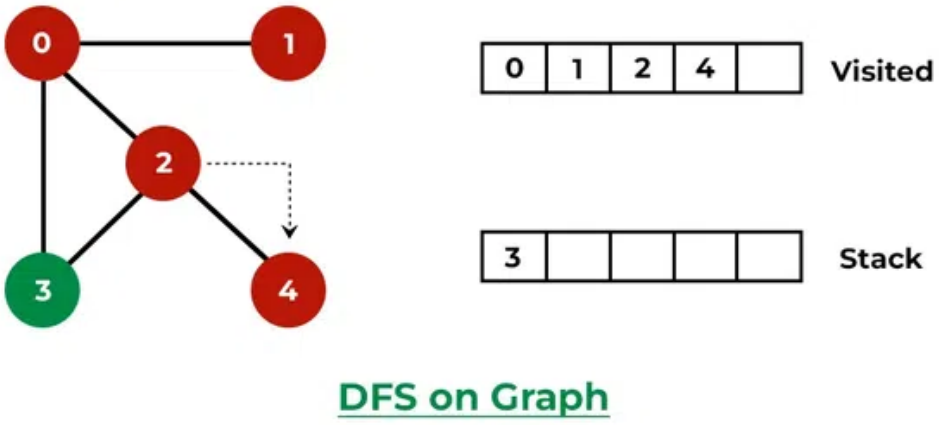
Step 4: *Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited (i.e, 3, 4) in the stack.*



DFS on Graph

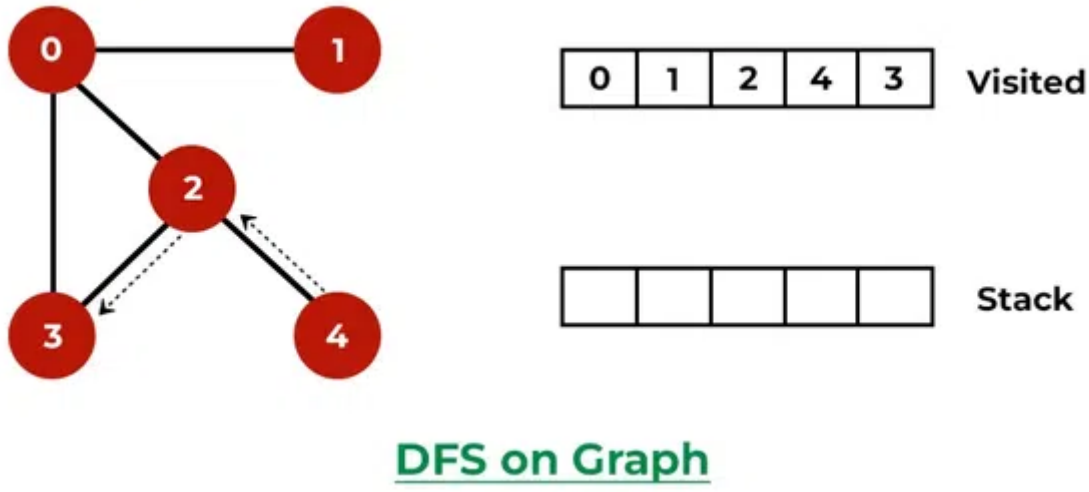
Visit node 2 and put its unvisited adjacent nodes (3, 4) into the stack

Step 5: *Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.*



Visit node 4

Step 6: *Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.*



Visit node 3

Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.

Below is the implementation of the above approach:

```
// C++ program to print DFS traversal from
// a given vertex in a  given graph
#include <bits/stdc++.h>
using namespace std;
 // Graph class represents a directed graph
// using adjacency list representation
class Graph {
public:
    map<int, bool> visited;
    map<int, list<int> > adj;

    // Function to add an edge to graph
    void addEdge(int v, int w);

    // DFS traversal of the vertices
    // reachable from v
    void DFS(int v);
};

void Graph::addEdge(int v, int w)
{
    // Add w to v's list.
    adj[v].push_back(w);
}

void Graph::DFS(int v)
{
    // Mark the current node as visited and
    // print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent
    // to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFS(*i);
}
 // Driver code
int main()
{
    // Create a graph given in the above diagram
```

```
    Graph g;
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal"
         " (starting from vertex 2) \n";

    // Function call
    g.DFS(2);

    return 0;
}
 // improved by Vishnudev C
```

Output

Following is Depth First Traversal (starting from vertex 2)

2 0 1 3

Time complexity: O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Auxiliary Space: O(V + E), since an extra visited array of size V is required, And stack size for iterative call to DFS function.

8.8 Advantages of Graphs

1. Graphs can be used to model and analyze complex systems and relationships.
2. They are useful for visualizing and understanding data.
3. Graph algorithms are widely used in computer science and other fields, such as social network analysis, logistics, and transportation.
4. Graphs can be used to represent a wide range of data types, including social networks, road networks, and the internet.

8.9 Disadvantages of Graphs

1. Large graphs can be difficult to visualize and analyze.
2. Graph algorithms can be computationally expensive, especially for large graphs.
3. The interpretation of graph results can be subjective and may require domain-specific knowledge.
4. Graphs can be susceptible to noise and outliers, which can impact the accuracy of analysis results.

Study Session 9: Spanning Trees

Expected Duration: 1 week or 2 contact hours

Introduction

A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. We discuss spanning trees in this Session.

Learning Outcomes

When you have studied this session, you should be able to explain:

9.1 Spanning tree
9.2 Minimum Spanning Tree
9.3 Kruskal's Algorithm
  9.3.1 How Kruskal's algorithm works
  9.3.2 Kruskal Algorithm Pseudocode
  9.3.3 Kruskal's vs Prim's Algorithm
  9.3.4 Kruskal's Algorithm Complexity
  9.3.5 Kruskal's Algorithm Applications
9.4 Spanning Tree Applications
9.5 Minimum Spanning tree Applications

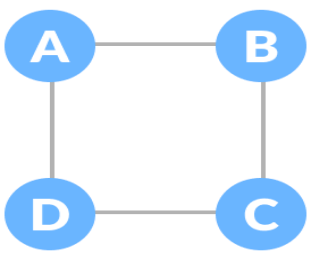
9.1 Spanning tree

A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree. The edges may or may not have weights assigned to them.

The total number of spanning trees with n vertices that can be created from a complete graph is equal to $n^{(n-2)}$. If we have n = 4, the maximum number of possible spanning trees is equal to 4^{4-2} = 16. Thus, 16 spanning trees can be formed from a complete graph with 4 vertices.

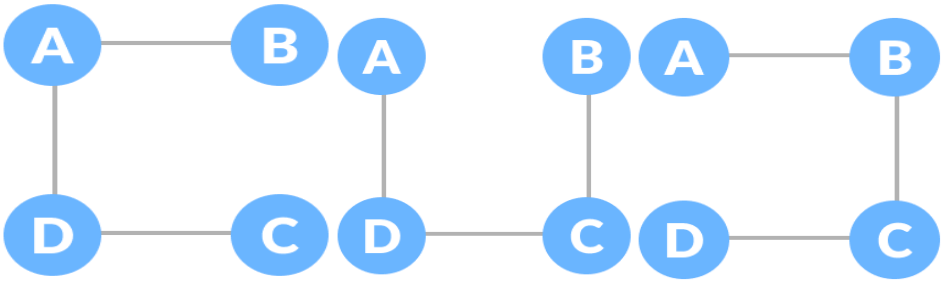
Example of a Spanning Tree

Let's understand the spanning tree with examples below:
Let the original graph be:



Normal graph

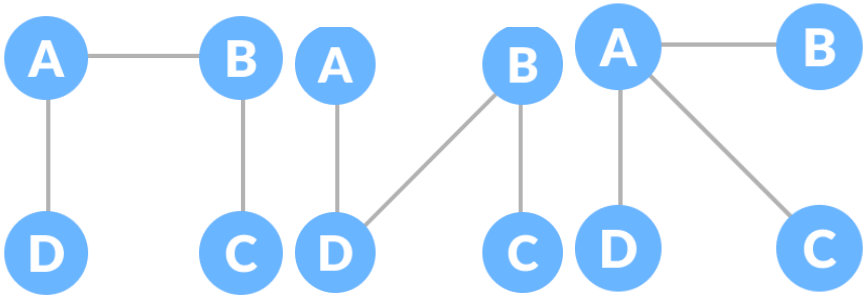
Some of the possible spanning trees that can be created from the above graph are:



(a)

(b)

(c)



(d)

(e)

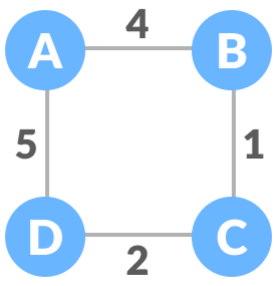
(f)

9.2 Minimum Spanning Tree

A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.

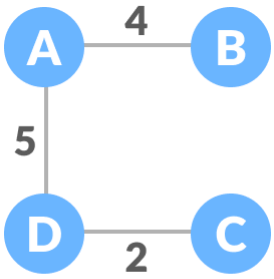
Example of a Spanning Tree
Let's understand the above definition with the help of the example below.

The initial graph is:

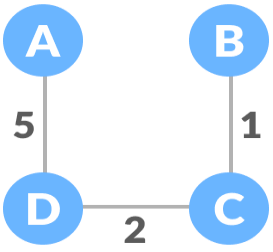


Weighted graph

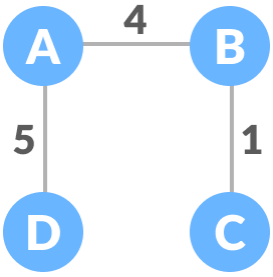
The possible spanning trees from the above graph are:



sum = 11

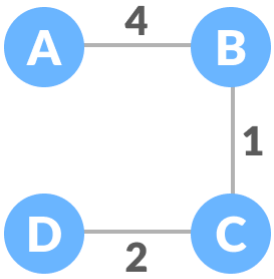


sum = 8



sum = 10

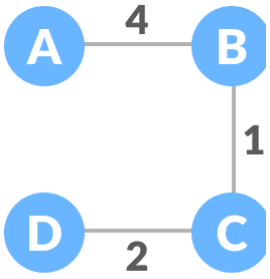
Minimum spanning tree – 1 Minimum spanning tree – 2 Minimum spanning tree – 3



sum = 7

Minimum spanning tree - 4

The minimum spanning tree from the above spanning trees is:



sum = 7

The final minimum spanning tree

The minimum spanning tree from a graph is found using the following algorithms:

1. Prim's Algorithm
2. Kruskal's Algorithm

9.3 Kruskal's Algorithm

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

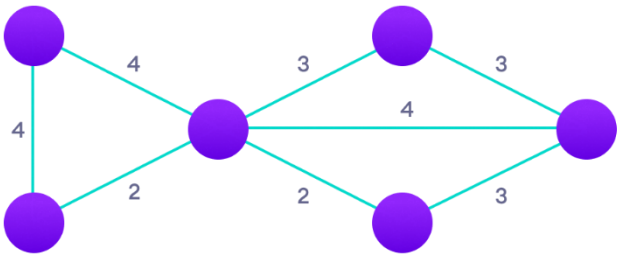
- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

9.3.1 How Kruskal's algorithm works

It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.
We start from the edges with the lowest weight and keep adding edges until we reach our goal.
The steps for implementing Kruskal's algorithm are as follows:

1. Sort all the edges from low weight to high
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edges until we reach all vertices.

Example of Kruskal's algorithm

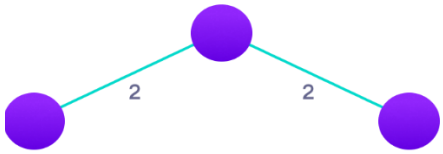


Step: 1

Start with a weighted graph

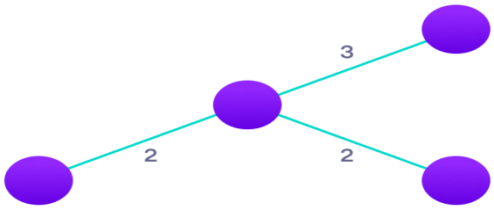
Step: 2

Choose the edge with the least weight, if there are more than 1, choose anyone.



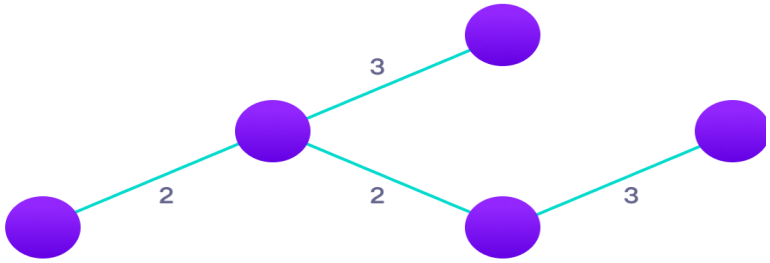
Step: 3

Choose the next shortest edge and add it



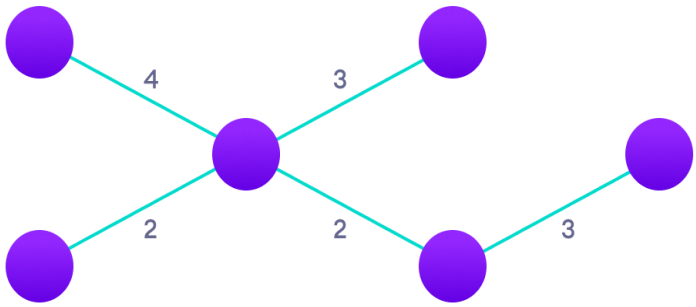
Step: 4

Choose the next shortest edge that doesn't create a cycle and add it



Step: 5

Choose the next shortest edge that doesn't create a cycle and add it



Step: 6

Repeat until you have a spanning tree

9.3.2 Kruskal Algorithm Pseudocode

Any minimum spanning tree algorithm revolves around checking if adding an edge creates a loop or not. The most common way to find this out is an algorithm called Union FInd. The Union-Find algorithm divides the vertices into clusters and allows us to check if two vertices belong to the same cluster or not and hence decide whether adding an edge creates a cycle.

```
KRUSKAL(G):
A = ∅
For each vertex v ∈ G.V:
   MAKE-SET(v)
For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
   if FIND-SET(u) ≠ FIND-SET(v):
   A = A ∪ {(u, v)}
   UNION(u, v)
return A
```

 # Kruskal's algorithm in Python

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

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

   # Search function

   def find(self, parent, i):
      if parent[i] == i:
         return i
      return self.find(parent, parent[i])

   def apply_union(self, parent, rank, x, y):
      xroot = self.find(parent, x)
      yroot = self.find(parent, y)
      if rank[xroot] < rank[yroot]:
         parent[xroot] = yroot
      elif rank[xroot] > rank[yroot]:
         parent[yroot] = xroot
      else:
         parent[yroot] = xroot
         rank[xroot] += 1
```

```
    #  Applying Kruskal algorithm
    def kruskal_algo(self):
        result = [ ]
        i, e = 0, 0
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = [ ]
        rank = [ ]
        for node in range(self.V):
            parent.append(node)
            rank.append(0)
        while e < self.V - 1:
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.apply_union(parent, rank, x, y)
        for u, v, weight in result:
            print("%d - %d: %d" % (u, v, weight))

g = Graph(6)
g.add_edge(0, 1, 4)
g.add_edge(0, 2, 4)
g.add_edge(1, 2, 2)
g.add_edge(1, 0, 4)
g.add_edge(2, 0, 4)
g.add_edge(2, 1, 2)
g.add_edge(2, 3, 3)
g.add_edge(2, 5, 2)
g.add_edge(2, 4, 4)
g.add_edge(3, 2, 3)
g.add_edge(3, 4, 3)
g.add_edge(4, 2, 4)
g.add_edge(4, 3, 3)
g.add_edge(5, 2, 2)
g.add_edge(5, 4, 3)
g.kruskal_algo( )
```

9.3.3 Kruskal's vs Prim's Algorithm

Prim's algorithm is another popular minimum spanning tree algorithm that uses a different logic to find the MST of a graph. Instead of starting from an edge, Prim's algorithm starts from a vertex and keeps adding lowest-weight edges which aren't in the tree, until all vertices have been covered.

9.3.4 Kruskal's Algorithm Complexity

The time complexity Of Kruskal's Algorithm is: O(E log E).

9.3.5 Kruskal's Algorithm Applications

(i) In order to layout electrical wiring
(ii) In computer network (LAN connection)

9.4 Spanning Tree Applications

(i) Computer Network Routing Protocol
(ii) Cluster Analysis
(iii)Civil Network Planning

9.5 Minimum Spanning tree Applications

(i) To find paths in the map
(ii) To design networks like telecommunication networks, water supply networks, and electrical grids.

Study Session 10: NP – Complete Problems

Expected Duration: 1 week or 2 contact hours

Introduction

NP-complete problems are tractable problems. NP means Nondeterministic-Polynomial. We briefly discuss NP – Problems in this Session.

Learning Outcomes

When you have studied this session, you should be able to explain:

10.1 Meaning of NP-Complete Problem
10.2 Characteristics of NP-Complete Problems
10.3 Formal definition
10.4 NP-complete problems
10.5 Solving NP-complete problems
10.6 Completeness under different types of reduction
10.7 Common misconceptions About NP-complete Problems

10.1 Meaning of NP-Complete Problem

NP-complete problem is any of a class of computational problems for which no efficient solution algorithm has been found. Many significant computer-science problems belong to this class, e.g., the traveling salesman problem, satisfiability problems, and graph-covering problems.

So-called easy, or tractable, problems can be solved by computer algorithms that run in polynomial time; i.e., for a problem of size *n*, the time or number of steps needed to find the solution is a polynomial function of *n, such as O(n) or O(n^2)*. Algorithms for solving hard, or intractable, problems, on the other hand, require times that are exponential functions of the problem size *n*. Polynomial-time algorithms are considered to be efficient, while exponential-time algorithms are considered inefficient, because the execution times of the latter grow much more rapidly as the problem size increases.

A problem is called NP (Nondeterministic Polynomial) if its solution can be guessed and verified in polynomial time; nondeterministic means that no particular rule is followed to make the guess. ***If a problem is NP and all other NP problems are polynomial-time reducible to it, the problem is NP-complete***. Thus, finding an efficient algorithm for any NP-complete problem implies that an efficient algorithm can be found for all such problems, since any problem belonging to this class can be recast into any other member of the class. It is not known whether any polynomial-time algorithms will ever be found for NP-complete problems, and determining whether these problems are tractable or intractable remains one of the most important questions in theoretical Computer Science. When an NP-complete problem must be solved, one approach is to use a polynomial algorithm to approximate the solution; the answer thus obtained will not necessarily be optimal but will be reasonably close.

The name "NP-complete" is short for "nondeterministic polynomial-time complete". In this name, "nondeterministic" refers to nondeterministic Turing machines, a way of mathematically formalizing the idea of a brute-force search algorithm. Polynomial time refers to an amount of time that is considered "quick" for a deterministic algorithm to check a single solution, or for a nondeterministic Turing machine to perform the whole search. "Complete" refers to the property of being able to simulate everything in the same complexity class.

More precisely, each input to the problem should be associated with a set of solutions of polynomial length, whose validity can be tested quickly (in polynomial time), such that the output for any input is "yes" if the solution set is non-empty and "no" if it is empty. The complexity class of problems of this form is called NP, an abbreviation for "nondeterministic polynomial time". A problem is said to be NP-hard if everything in NP can be transformed in polynomial time into it even though it may not be in NP. Conversely, a problem is NP-complete if it is both in NP and NP-hard. The NP-complete problems represent the hardest problems in NP. If some NP-complete problem has a polynomial time algorithm, all problems in NP do. The set of NP-complete problems is often denoted by **NP-C** or **NPC**.

Although a solution to an NP-complete problem can be *verified* "quickly", there is no known way to *find* a solution quickly. That is, the time required to solve the problem using any currently known algorithm increases rapidly as the size of the problem grows. As a consequence, determining whether it is possible to solve these problems quickly, called the P versus NP problem, is one of the fundamental unsolved problems in computer science today.

While a method for computing the solutions to NP-complete problems quickly remains undiscovered, computer scientists and programmers still frequently encounter NP-complete problems. NP-complete problems are often addressed by using **heuristic methods** and **approximation algorithms**.

10.2 Characteristics of NP-Complete Problems

In computational complexity theory, a problem is **NP-complete** when:

1. It is a decision problem, meaning that for any input to the problem, the output is either "yes" or "no".
2. When the answer is "yes", this can be demonstrated through the existence of a short (polynomial length) *solution*.
3. The correctness of each solution can be verified quickly (namely, in polynomial time) and a brute-force search algorithm can find a solution by trying all possible solutions.
4. The problem can be used to simulate every other problem for which we can verify quickly that a solution is correct. In this sense, NP-complete problems are the hardest of the problems to which solutions can be verified quickly. If we could find solutions of some NP-complete problem quickly, we could quickly find the solutions of every other problem to which a given solution can be easily verified.

10.3 Formal definition

A decision problem *C* is NP-complete if:

1. *C* is in NP, and
2. Every problem in NP is reducible to *C* in polynomial time.

C can be shown to be in NP by demonstrating that a candidate solution to *C* can be verified in polynomial time.

Note that a problem satisfying condition 2 is said to be NP-hard, whether or not it satisfies condition 1.

A consequence of this definition is that if we had a polynomial time algorithm (on a UTM, or any other Turing-equivalent abstract machine) for *C*, we could solve all problems in NP in polynomial time.

10.4 NP-complete problems

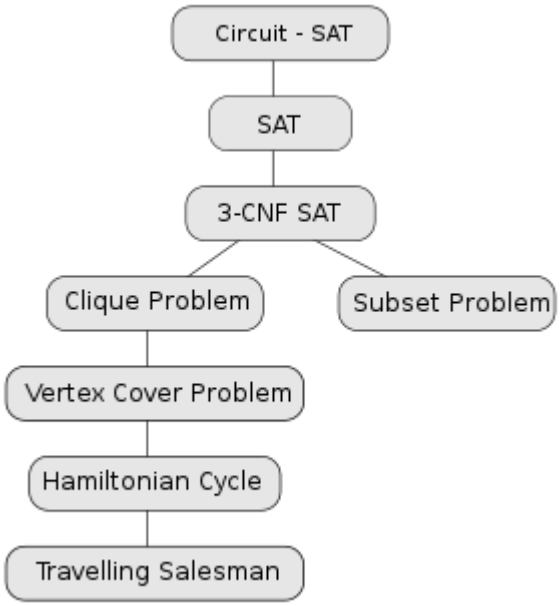


Figure 10.1: Some NP-complete problems, indicating the reductions typically used to prove their NP-completeness

The easiest way to prove that some new problem is NP-complete is first to prove that it is in NP, and then to reduce some known NP-complete problem to it. Therefore, it is useful to know a variety of NP-complete problems. The list below contains some well-known problems that are NP-complete when expressed as decision problems.

- Boolean satisfiability problem (SAT)
- Knapsack problem
- Hamiltonian path problem
- Travelling salesman problem (decision version)

- Subgraph isomorphism problem
- Subset sum problem
- Clique problem
- Vertex cover problem
- Independent set problem
- Dominating set problem
- Graph coloring problem

10.5 Solving NP-complete Problems

At present, all known algorithms for NP-complete problems require time that is superpolynomial in the input size, in fact exponential in $O(n^k)$ for some $k > 0$ and it is unknown whether there are any faster algorithms.The following techniques can be applied to solve computational problems in general, and they often give rise to substantially faster algorithms:

- **Approximation**: Instead of searching for an optimal solution, search for a solution that is at most a factor from an optimal one.
- **Randomization**: Use randomness to get a faster average running time, and allow the algorithm to fail with some small probability. Note: The Monte Carlo method is not an example of an efficient algorithm in this specific sense, although evolutionary approaches like Genetic algorithms may be.
- **Restriction**: By restricting the structure of the input (e.g., to planar graphs), faster algorithms are usually possible.
- **Parameterization**: Often there are fast algorithms if certain parameters of the input are fixed.
- **Heuristic**: An algorithm that works "reasonably well" in many cases, but for which there is no proof that it is both always fast and always produces a good result. Metaheuristic approaches are often used.

One example of a heuristic algorithm is a suboptimal O(n log n) greedy coloring algorithm used for graph coloring during the register allocation phase of some compilers, a technique called graph-coloring global register allocation. Each vertex is a variable, edges are drawn between variables which are being used at the same time, and colors indicate the register assigned to each variable. Because most RISC machines have a fairly large number of general-purpose registers, even a heuristic approach is effective for this application.

10.6 Completeness under Different Types of Reduction

In the definition of NP-complete given above, the term *reduction* was used in the technical meaning of a polynomial-time many-one reduction.Another type of reduction is polynomial-time Turing reduction. A problem X is polynomial-time Turing-reducible to a problem Y if, given a subroutine that solves Y in polynomial time, one could write a program that calls this subroutine and solves X in polynomial time. This contrasts with many-one reducibility, which has the restriction that the program can only call the subroutine once, and the return value of the subroutine must be the return value of the program.

If one defines the analogue to NP-complete with Turing reductions instead of many-one reductions, the resulting set of problems won't be smaller than NP-complete; it is an open question whether it will be any larger.

Another type of reduction that is also often used to define NP-completeness is the logarithmic-space many-one reduction which is a many-one reduction that can be computed with only a logarithmic amount of space. Since every computation that can be done in logarithmic space can also be done in polynomial time it follows that if there is a logarithmic-space many-one reduction then there is also a polynomial-time many-one reduction. This type of reduction is more refined than the more usual polynomial-time many-one reductions and it allows us to distinguish more classes such as P-complete. Whether under these types of reductions the definition of NP-complete changes is still an open problem. All currently known NP-complete problems are NP-complete under log space reductions.

10.7 Common Misconceptions about NP-complete Problems

The following misconceptions are frequent.

1. *"NP-complete problems are the most difficult known problems."* Since NP-complete problems are in NP, their running time is at most exponential. However, some problems have been proven to require more time, for example Presburger arithmetic. Of some problems, it has even been proven that they can never be solved at all, for example the halting problem.

2. *"NP-complete problems are difficult because there are so many different solutions."* On the one hand, there are many problems that have a solution space just as large, but can be solved in polynomial time (for example minimum spanning tree). On the other hand, there are NP-problems with at most one solution that are NP-hard under randomized polynomial-time reduction (see Valiant–Vazirani theorem).

3. *"Solving NP-complete problems requires exponential time."* First, this would imply $P \neq NP$, which is still an unsolved question. Further, some NP-complete problems actually have algorithms running in superpolynomial, but subexponential time such as $O(2^{\sqrt{n}}n)$. For example, the independent set and dominating set problems for planar graphs are NP-complete, but can be solved in subexponential time using the planar separator theorem.

4. *"Each instance of an NP-complete problem is difficult."* Often some instances, or even most instances, may be easy to solve within polynomial time. However, unless P=NP, any polynomial-time algorithm must asymptotically be wrong on more than polynomially many of the exponentially many inputs of a certain size.

5. *"If P=NP, all cryptographic ciphers can be broken."* A polynomial-time problem can be very difficult to solve in practice if the polynomial's degree or constants are large enough. In addition, information-theoretic security provides cryptographic methods that cannot be broken even with unlimited computing power.

References

Encyclopaedia Britannica (2023). NP-complete problem, https://www.britannica.com/topic/computational-complexity

Geeks for Geeks (2023). Introduction to Graphs Data Structure https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/?ref=lbp

Geeks for Geeks, Amortized Analysis, https://www.geeksforgeeks.org/introduction-to-amortized-analysis/

Geeksforgeeks, Divide and Conquer, https://www.geeksforgeeks.org/divide-and-conquer/

Geeksforgeeks, Memoizationhttps://www.geeksforgeeks.org/memoization-1d-2d-and-3d/Accessed October, 2023.

Geeksforgeeks, Merge Sort Algorithm, https://www.geeksforgeeks.org/merge-sort/

Geeksforgeeks.org (2023). What is Memoization? https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/ Accessed October, 2023.

Grossman, Dan. https://courses.cs.washington.edu/courses/cse332/10sp/lectures/lecture21.pdf "CSE332: Data Abstractions" (PDF). *cs.washington.edu*. Retrieved 14 March 2015.

Hemaspaandra, L. A.; Williams, R. (2012). "SIGACT News Complexity Theory Column 76". *ACM SIGACT News*. **43** (4): 70. doi:10.1145/2421119.2421135. S2CID 13367514.

Kozen, Dexter (Spring 2011). "CS 3110 Lecture 20: Amortized Analysis". Cornell University. Retrieved 14 March 2015.

Programiz, Greedy Algorithm,https://www.programiz.com/dsa/greedy-algorithm#google_vignette

Programiz, Kruskal Algorithm, https://www.programiz.com/dsa/kruskal-algorithm

Programiz, Spanning Trees, https://www.programiz.com/dsa/spanning-tree-and-minimum-spanning-tree#google_vignette

Samuel Sam (2023). Introduction to Divide & Conquer Algorithm, https://www.tutorialspoint.com/introduction-to-divide-and-conquer-algorithms

Samuel Sam (2023). Introduction to Greedy Algorithm https://www.tutorialspoint.com/introduction-to-greedy-algorithms

Syed Zaid Irshad Power Point Slides on Design& Analysis of Algorithms

Tutorilspoint, Design and Analysis of Algorithms, https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_randomized_quick_sort.htm

vanLeeuwen J. (1998). Handbook of Theoretical Computer Science. Elsevier. p. 84. ISBN 978-0-262-72014-4.

vanLeeuwen J. (1998). Handbook of Theoretical Computer Science. Elsevier. p. 80. ISBN 978-0-262-72014-4

Vijini Mallawaarachchi (2020). 10 Graph Algorithms Visually Explained, https://towardsdatascience.com/10-graph-algorithms-visually-explained-e57faa1336f3

Wikipedia (2023): NP-completeness**,** https://en.wikipedia.org/wiki/NP-#:~:text=NP%2Dcomplete%20problems%20are%20in,a%20non%2Ddeterministic%20Turing%20machine.

Wikipedia, Amortized Analysis: https://en.wikipedia.org/wiki/Amortized_analysis