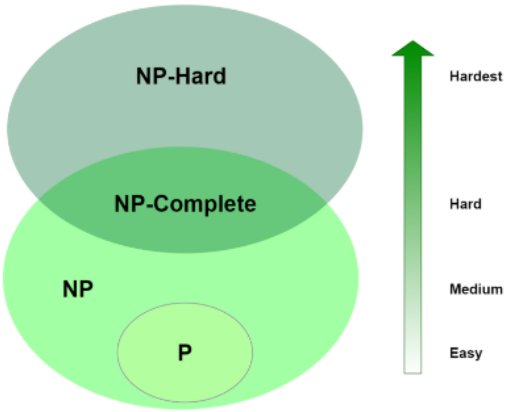
UNIT V NP-COMPLETE AND APPROXIMATION ALGORITHM

Tractable and intractable problems: Polynomial time algorithms – Venn diagram representation – NPalgorithms – NP-hardness and NP-completeness – Bin Packing problem – Problem reduction: TSP – 3-CNF problem. Approximation Algorithms: TSP – Randomized Algorithms: concept and application – primality testing – randomized quick sort – Finding kth smallest number

Introduction

In theoretical computer science, the classification and complexity of common problem definitions have two major sets; {P} which is "Polynomial" time and {NP} which "Non-deterministic Polynomial" time. There are also NP-Hard and NP-Complete sets, which we use to express more sophisticated problems.



Tractable Problem:

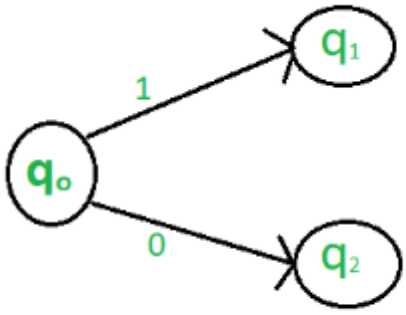
- A problem that is solvable by a polynomial-time algorithm. The upper bound is polynomial.
- Examples: Ordered searching -O(log n), Sorting - O(nlog n), String editing - O(mn).

Intractable Problem:

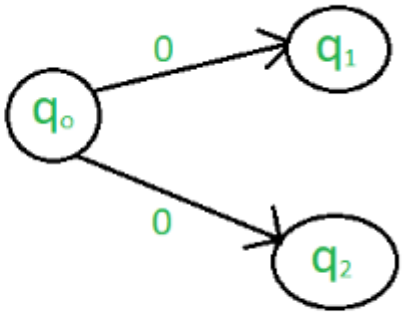
- A problem that cannot be solved by a polynomial-time algorithm. The lower bound is exponential.
- Examples: Travelling Salesman problem -$O(n^2 2^n)$, knapsack problem - $O(2^{n/2})$.
    - While considering 2n, where n is the number of input values.
        - When n = 10 – it takes 1024 operations
        - When n = 11 – it takes 2048 operations
- There is significant increase in time for small increase in the input, which makes the algorithm NP-type

In a deterministic algorithm, for a given particular input, the computer will always produce the same output going through the same states but in the case of the non-deterministic algorithm, for the same input, the compiler may produce different output

in different runs. In fact, non-deterministic algorithms can't solve the problem in polynomial time and can't determine what is the next step. The non-deterministic algorithms can show different behaviors for the same input on different execution and there is a degree of randomness to it.



Deterministic Algorithm



Non-Deterministic Algorithm

Unit V

Tractable Problems and Intractable Problems

The terms "tractable" and "intractable" are commonly used in the field of computer science and computational complexity theory to describe the solvability of problems.

1. Tractable Problems

- A problem is considered tractable if there exists an algorithm that can solve it efficiently.
- Tractable problems are generally those for which the solution can be found in polynomial time, meaning the time required to solve the problem grows at most polynomially with the size of the input.
- Examples of tractable problems include sorting a list of numbers, finding the shortest path in a graph, and multiplying two matrices.

2. Intractable Problems:

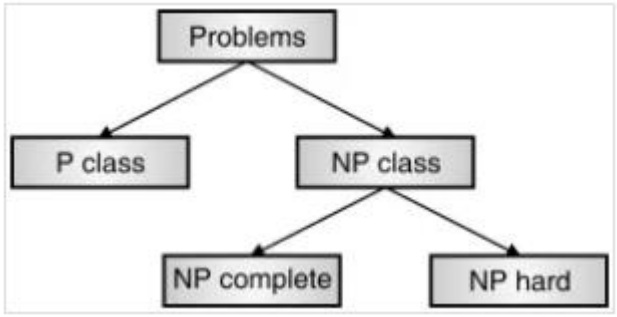
- Conversely, a problem is considered intractable if no known algorithm can solve it efficiently for all possible inputs.
- Intractable problems often have exponential time complexity, meaning the time required to solve the problem grows exponentially with the size of the input.
- The most famous example of an intractable problem is the traveling salesman problem (TSP), where one must find the shortest possible route that visits a given set of cities and returns to the starting city. While there are algorithms for solving TSP, they become impractical for a large number of cities.

One particularly significant concept related to intractable problems is NP-completeness. A problem is NP-complete if it is in the set of NP problems (problems for which a solution, once guessed, can be checked quickly) and is as hard as the hardest problems in NP. Many real-world problems that are of interest are NP-complete, and while no polynomial-time algorithm is known for solving them, they are not proven to be inherently unsolvable in polynomial time.

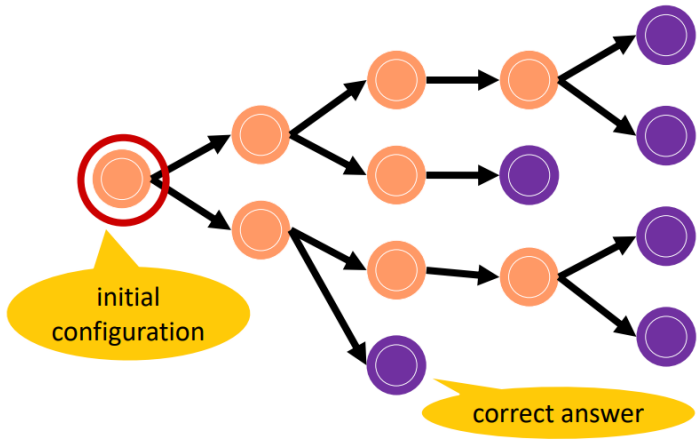
Understanding the tractability or intractability of a problem is crucial in various fields, especially when dealing with optimization, decision-making, and resource allocation. It helps guide the choice of algorithms and methodologies based on the practical constraints of solving a particular problem efficiently.

P and NP Problems

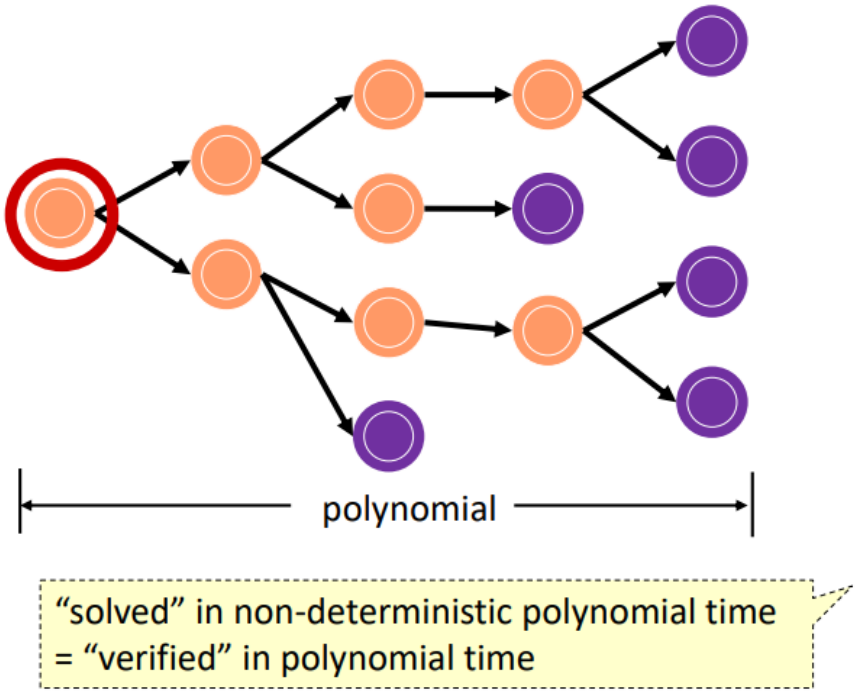
P and NP Problems are the class of problems. Problems are grouped based on their complexity.



Non- Deterministic Problem Solving



Non- Deterministic Polynomial



P-class Problems

- P problems are a set of problems that can be solved in polynomial time by deterministic algorithms.
- P is also known as PTIME or DTIME complexity class.
- P problems are a set of all decision problems which can be solved in polynomial time using the deterministic Turing machine.
- They are simple to solve, easy to verify and take computationally acceptable time for solving any instance of the problem. Such problems are also known as "tractable".
- In the worst case, searching an element from the list of size n takes n comparisons. The number of comparisons increases linearly with respect to the input size. So linear search is P problem.
- In practice, most of the problems are P problems. Searching an element in the array (O(n)), inserting an element at the end of a linked list (O(n)), sorting data using selection sort(O(n2)), finding the height of the tree (O(log2n)), sort data using merge sort(O(nlog2n)), matrix multiplication O(n3) are few of the examples of P problems.
- An algorithm with O(2n) complexity takes double the time if it is tested on a problem of size (n + 1). Such problems do not belong to class P.
- It excludes all the problems which cannot be solved in polynomial time. The knapsack problem using the brute force approach cannot be solved in polynomial time. Hence, it is not a P problem.
- There exist many important problems whose solution is not found in polynomial time so far, nor it has been proved that such a solution does not exist. TSP, Graph colouring, partition problem, knapsack etc. are examples of such classes.

Examples of P Problems:

- Insertion sort
- Merge sort
- Linear search
- Matrix multiplication
- Finding minimum and maximum elements from the array

NP-Class of Problems

- NP is a set of problems which can be solved in nondeterministic polynomial time. NP does not mean non-polynomial, it stands for Non-Deterministic Polynomial-time.
- The non-deterministic algorithm operates in two stages.
- Nondeterministic (guessing) stage: For input instance I, some solution string S is generated, which can be thought of as a candidate solution.
- Deterministic (verification) stage: I and S are given as input to the deterministic algorithm, which returns "Yes" if S is a solution for input instance I.
- The solution to NP problems cannot be obtained in polynomial time, but given the solution, it can be verified in polynomial time.
- NP includes all problems of P, i.e. $P \subseteq NP$.
- Knapsack problem (O(2n)), Travelling salesman problem (O(n!)), Tower of Hanoi (O(2n – 1)), Hamiltonian cycle (O(n!)) are examples of NP problems.
- NP Problems are further classified into NP-complete and NP-hard categories.

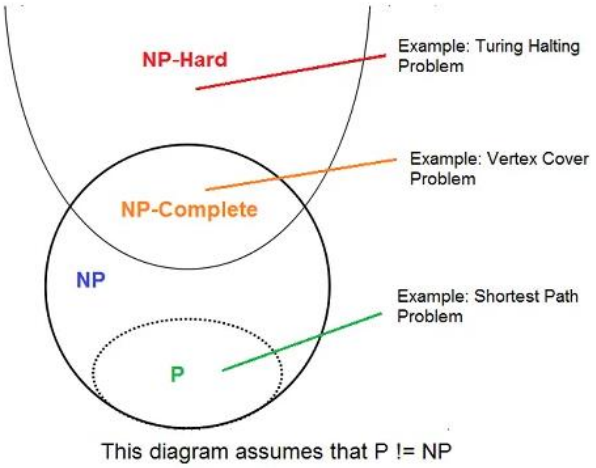
Examples of NP problems

- Knapsack problem (O(2n))
- Travelling salesman problem (O(n!))
- Tower of Hanoi (O(2n – 1))
- Hamiltonian cycle (O(n!))

Differentiate between P Problems and NP Problems	
P Problems	**NP Problems**
P problems are set of problems which can be solved in polynomial time by deterministic algorithms.	NP problems are problems which can be solved in nondeterministic polynomial time.
P Problems can be solved and verified in polynomial time	The solution to NP problems cannot be obtained in polynomial time, but if the solution is given, it can be verified in polynomial time.
P problems are a subset of NP problems	NP Problems are a superset of P problems
All P problems are deterministic in nature	All the NP problems are non-deterministic in nature
Example: Selection sort, Linear search	Example: TSP, Knapsack problem

APEC

NP-Complete Problems and NP-Hard Problems



NP-Complete and NP-Hard Problems are interesting classes of problems. These problems are addressed in different ways by computer scientists.

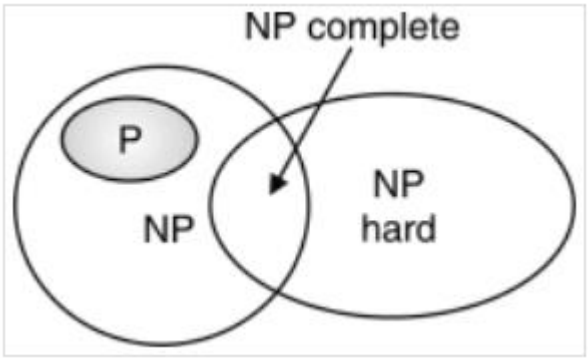
NP-Complete Problems

- Polynomial time reduction implies that one problem is at least as hard as another problem, within the polynomial time factor. If A ≤p B, implies A is not harder than B by some polynomial factor.
- Decision problem A is called NP-complete if it has the following two properties :
- It belongs to class NP.
- Every other problem B in NP can be transformed to A in polynomial time, i.e. For every B ∈ NP, B ≤p A.
- These two facts prove that NP-complete problems are the harder problems in class NP. They are often referred to as NPC.
- If any NP-complete problem belongs to class P, then
- P = NP. However, a solution to any NP-complete problem can be verified in polynomial time, but cannot be obtained in polynomial time.
- Theorem
- Let A be an NP-complete problem. For some decision problem B ∈ NP, if B ≤p A then B is also an NP-complete problem.
- NP-complete problems are often solved using randomization algorithms, heuristic approaches or approximation algorithms.
- Some of the well-known NP-complete problems are listed here :
    - Boolean satisfiability problem.
    - Knapsack problem.
    - Hamiltonian path problem.
    - Travelling salesman problem.

- Subset sum problem.
- Vertex covers the problem.
- Graph colouring problem.
- Clique problem.

NP-Hard Problem

- Formally, a decision problem p is called NP-hard, if every problem in NP can be reduced to p in polynomial time.
- NP-hard is a superset of all problems. NPC is in NP-hard, but the converse may not be true.
- NP-hard problems are at least as hard as the hardest problems in NP.
- If we can solve any NP-hard problem in polynomial time, we would be able to solve all the problems in NP in polynomial time.
- NP-hard problems do not have to be in NP. Even they may not be a decision problem.
- The subset subproblem, the travelling salesman problem is NPC and also belongs to NP-hard. There are certain problems which belong to NP-hard but they are not NP-complete.
- A well-known example of the NP-hard problem is the Halting problem.
- The halting problem is stated as, "Given an algorithm and set of inputs, will it run forever ?" The answer to this question is Yes or No, so this is a decision problem.
- There does not exist any known algorithm which can decide the answer for any given input in polynomial time. So halting problem is an NP-hard problem.
- Different mathematicians have given different relationships considering the possibilities of P = NP and P ≠ NP.
- The relationship between P, NP, NP-Complete and NP-hard is described in below diagram assuming that P ≠ NP. This is a widely accepted relationship among these complexity classes.



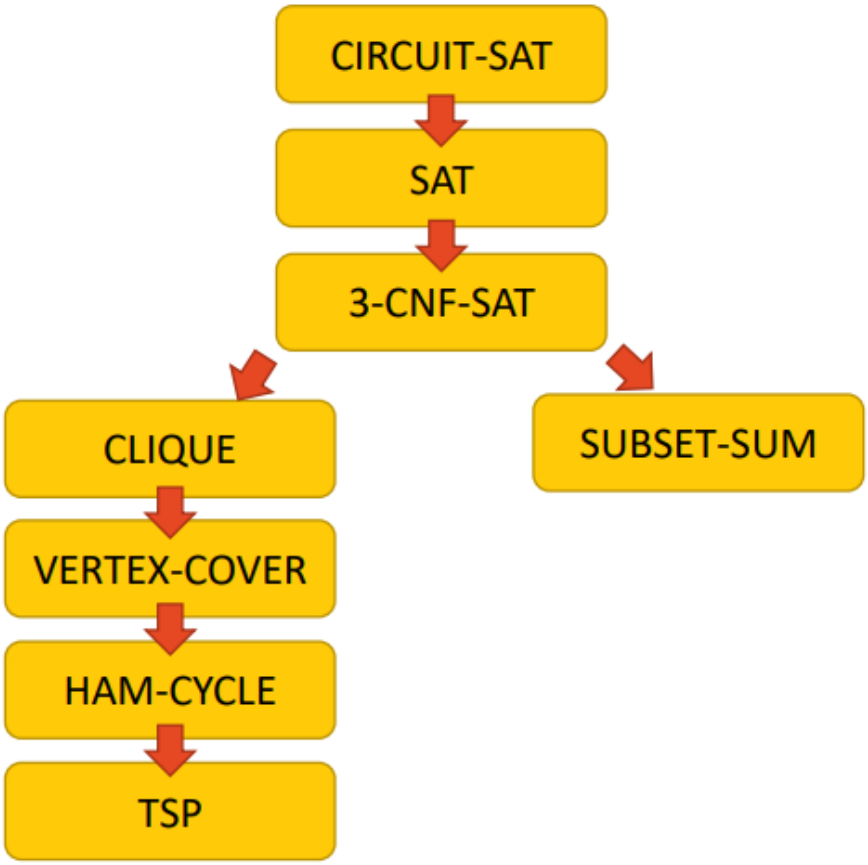
- Subset sum problem and travelling salesman problem are NPC and also belong to NP-hard. There are certain problems which belong to NP-hard but they are not

NP-complete. A well-known example of an NP-hard problem is the Halting problem.

- The halting problem is stated as, "Given the algorithm and set of inputs, will it run forever ?" The answer to this question is Yes or No, so this is a decision problem.
- There does not exist any known algorithm which can decide the answer for any given input in polynomial time. So halting problem is an NP-hard problem.
- Also, the Boolean satisfiability problem, which is in NPC, can be reduced to an instance of the halting problem in polynomial time by transforming it to the description of a Turing machine that tries all truth value assignments. The Turing machine halts when it finds such an assignment, otherwise, it goes into an infinite loop.

NP-Complete	**NP-Hard**
NP-complete problems are decision problems	The NP-hard problem might not be a decision problem.
NP-complete problems are harder problems	NP-hard problems are the hardest problem.
The NP-complete problems are in NP	NP-hard problems may not be in NP
Problem X is NP-complete if NP problem Y is reducible to X in polynomial time.	Problem X is NP-hard if NP-complete problem Y is reducible to X in polynomial time.
Ex. 3 SAT problem	Ex. Halting problem.

Roadmap of NP – completeness



Polynomial time algorithms

In computer science, the efficiency of algorithms is often analyzed in terms of their time complexity, which describes how the running time of an algorithm grows with the size of the input. An algorithm is said to have polynomial time complexity if its running time is polynomially related to the size of the input.

A polynomial is an expression of the form \(a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0\), where \(a_n, a_{n-1}, \ldots, a_1, a_0\) are constants, and \(n\) is a non-negative integer. The time complexity of an algorithm is expressed in big-O notation, and an algorithm is considered to have polynomial time complexity if its time complexity is \(O(n^k)\) for some constant \(k\), where \(n\) is the size of the input.

Examples of polynomial time algorithms include:

1. Linear Time (O(n)):
 - Algorithms that have a linear time complexity, such as iterating through a list or array once, have a running time proportional to the size of the input.

2. Quadratic Time (O(n^2)):
- Algorithms with quadratic time complexity have a running time proportional to the square of the size of the input. Nested loops are a common characteristic of quadratic time algorithms.

3. Cubic Time (O(n^3)):
- Algorithms with cubic time complexity have a running time proportional to the cube of the size of the input. These algorithms often involve three nested loops.

4. Polynomial Time (O(n^k)):
- In general, any algorithm with a time complexity of \(O(n^k)\), where \(k\) is a constant, is considered a polynomial time algorithm.

Many common and important algorithms, like quicksort, mergesort, and various graph algorithms, have polynomial time complexity and are widely used in practice.

Algorithm Discovery with Venn Diagrams

Algorithm discovery typically involves understanding the problem, identifying potential solutions, and refining those solutions through iterative processes. While Venn diagrams may not be the most conventional tool for algorithm discovery, they can still be employed in the initial stages to explore the relationships and intersections of different aspects of the problem. Here's a conceptual example of using Venn diagrams in the algorithm discovery process:

Example: Sorting Algorithm Discovery

1. Problem Understanding:
- Problem: Design an algorithm to sort a list of numbers in ascending order.

2. Initial Concepts:
- Venn Diagram Sets:
  - Set A: Comparison-based sorting algorithms.
  - Set B: Non-comparison-based sorting algorithms.

3. Initial Exploration:
- **Set A (Comparison-based Sorting):**
  - Examples: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, QuickSort.
- **Set B (Non-comparison-based Sorting):**
  - Examples: Counting Sort, Radix Sort.

- Use Venn diagram circles to represent these sets.

4. Iterative Refinement:
  - Refinement 1: Divide and Conquer vs. Non-Divide and Conquer:
    - Subsets within Set A:
      - Divide and Conquer Sorting Algorithms (e.g., MergeSort, QuickSort).
      - Non-Divide and Conquer Sorting Algorithms (e.g., Bubble Sort, Insertion Sort).

  - Refinement 2: In-place vs. Not In-place:
    - Subsets within Set A:
      - In-place Sorting Algorithms (e.g., Bubble Sort, Selection Sort).
      - Not In-place Sorting Algorithms (e.g., Merge Sort, QuickSort).

  - Update the Venn diagram to reflect these refinements.

5. Final Exploration:
  - **Set C (Optimizations and Special Cases):**
    - Subsets within Set A:
      - Adaptive Sorting Algorithms (e.g., Insertion Sort).
      - Stable Sorting Algorithms (e.g., Merge Sort).

  - Update the Venn diagram to include Set C and its subsets.

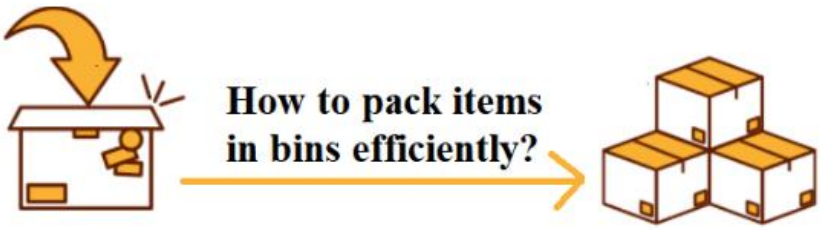
6. Algorithm Selection:
  - Consider the intersections of sets and subsets to guide algorithm selection based on specific requirements (e.g., stability, in-place requirements).

Venn diagrams can serve as a useful starting point for organizing and visualizing these complex relationships.

APEC

Bin Packing problem

Bin Packing problem involves assigning n items of different weights and bins each of capacity c to a bin such that number of total used bins is minimized. It may be assumed that all items have weights smaller than bin capacity.



Mathematical Formulation of Bin Packing

The Bin-Packing Problem (BPP) can also be described, using the terminology of knapsack problems, as follows. Given n items and n knapsacks (or bins), with

$$Wj = weight\ of\ item\ j,$$
$$cj = capacity\ of\ each\ bin$$

Assign each item to one bin so that the total weight of the items in each bin does not exceed c and the number of bins used is a minimum.

A possible mathematical formulation of the problem is:

$$\text{minimize} \quad z = \sum_{i=1}^{n} y_i$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_{ij} \le c y_i, \quad i \in N = \{1, \ldots, n\},$$

$$\sum_{i=1}^{n} x_{ij} = 1, \quad j \in N,$$

$$y_i = 0 \text{ or } 1, \quad i \in N,$$

$$x_{ij} = 0 \text{ or } 1, \quad i \in N, j \in N,$$

where

$$y_i = \begin{cases} 1 & \text{if bin } i \text{ is used;} \\ 0 & \text{otherwise,} \end{cases}$$

$$x_{ij} = \begin{cases} 1 & \text{if item } j \text{ is assigned to bin } i; \\ 0 & \text{otherwise.} \end{cases}$$

We will suppose, as is usual, that the weights Wj are positive integers. Hence, without loss of generality, we will also assume that c is a positive integer

$$Wj < c \text{ for } j \text{ belonging to } N.$$

Approximation algorithms allow for getting a solution close to the (optimal) solution of an optimization problem in polynomial time.

An algorithm is an α-approximation algorithm for an optimized problem if:

- The algorithm runs in polynomial time
- The algorithm always produces a solution that is within a factor of α of the optimal solution

For the Bin-Packing problem, let us consider bins of size 1



Assuming the sizes of the items be {0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6}.
The most optimal solution (z(I))for this instance I would be

Optimal packing:



Next Fit algorithm:

Let us consider the same example as used above and bins of size 1



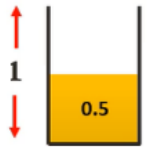
Assuming the sizes of the items be {0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6}.

The minimum number of bins required would be Ceil ((Total Weight) / (Bin Capacity))= Celi(3.7/1) = 4 bins.

The Next fit solution (NF(I))for this instance I would be-

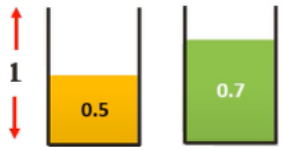
Considering 0.5 sized item first, we can place it in the first bin

0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6.



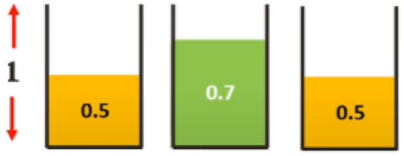
Moving on to the 0.7 sized item, we cannot place it in the first bin. Hence we place it in a new bin.

0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6.



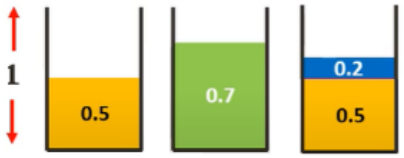
Moving on to the 0.5 sized item, we cannot place it in the current bin. Hence we place it in a new bin.

0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6.



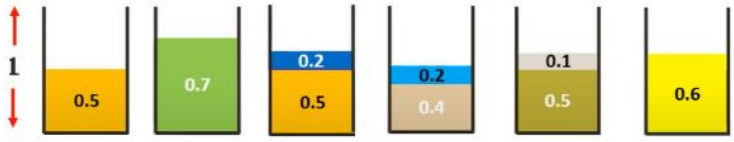
Moving on to the 0.2 sized item, we can place it in the current (third bin)

0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6.



Similarly, placing all the other items following the Next-Fit algorithm we get-

0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6.



Thus we need 6 bins as opposed to the 4 bins of the optimal solution. Thus we can see that this algorithm is not very efficient.

Analyzing the approximation ratio of Next-Fit algorithm

- Worst case time complexity: O(n).
- Average case time complexity: O(n)
- Best case time complexity: O(n).
- Space complexity: O(n).

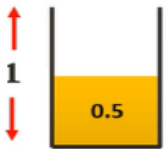
First Fit algorithm

A better algorithm, First-Fit (FF), considers the items according to increasing indices and assigns each item to the lowest indexed initialized bin into which it fits; only when the current item cannot fit into any initialized bin, is a new bin introduced

The First fit solution (FF(I))for this instance I would be-

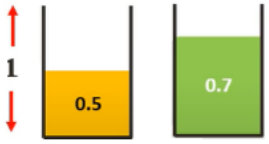
Considering 0.5 sized item first, we can place it in the first bin

0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6.



Moving on to the 0.7 sized item, we cannot place it in the first bin. Hence we place it in a new bin.

0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6.



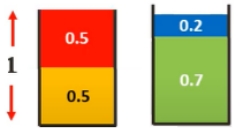
Moving on to the 0.5 sized item, we can place it in the first bin.

0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6.



Moving on to the 0.2 sized item, we can place it in the first bin, we check with the second bin and we can place it there.

0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6.



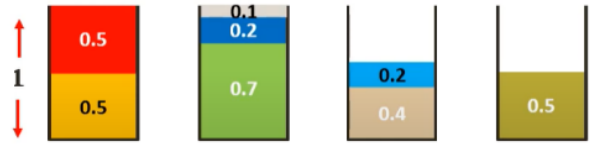
Moving on to the 0.4 sized item, we cannot place it in any existing bin. Hence we place it in a new bin.

0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6.



Similarly, placing all the other items following the First-Fit algorithm we get-

0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6.



Thus we need 5 bins as opposed to the 4 bins of the optimal solution but is much more efficient than Next-Fit algorithm.

Analyzing the approximation ratio of Next-Fit algorithm

If FF(I) is the First-fit implementation for I instance and z(I) is the most optimal solution, then:

$$FF(I) \leq \frac{17}{10} z(I) + 2$$

for all instances I of BPP, and that there exist instances I, with $z(I)$ arbitrarily large, for which

$$FF(I) > \frac{17}{10} z(I) - 8.$$

It can be seen that the First Fit never uses more than 1.7 * z(I) bins. So First-Fit is better than Next Fit in terms of upper bound on number of bins.

Analyzing the approximation ratio of First-Fit algorithm

- Worst case time complexity: Θ(n*n)
- Average case time complexity: Θ(n*n)
- Best case time complexity (Can be achieved using Self-balancing Binary trees): Θ(nlogn)
- Space complexity: Θ(n)

Best Fit Algorithm

The next algorithm, Best-Fit (BF), is obtained from FF by assigning the current item to the feasible bin (if any) having the smallest residual capacity (breaking ties in favour of the lowest indexed bin).

Simply put the idea is to places the next item in the tightest spot. That is, put it in the bin so that the smallest empty space is left.

For the above example, Best Fit works similarly like First Fit algorithm only.

Analyzing the approximation ratio of First-Fit algorithm

- Worst case time complexity: Θ(n*n)
- Average case time complexity: Θ(n*n)
- Best case time complexity (Can be achieved using Self-balancing Binary trees): Θ(nlogn)
- Space complexity: Θ(n)

Applications of Bin-Packing Algorithms

- Loading of containers like trucks.
- Placing data on multiple disks.
- This is used extensively while transporting goods over ships
- Job scheduling.

Simple Python Program for Bin Packing Algorithm – Using First Fit Algorithm

```
def first_fit(items, bin_capacity):
    bins = [[]]  # Initialize the first bin
    for item in items:
        packed = False
        # Try to pack the item into existing bins
        for bin_index, bin_contents in enumerate(bins):
            if sum(bin_contents) + item <= bin_capacity:
                bins[bin_index].append(item)
                packed = True
                break
        # If the item couldn't be packed into existing bins, create a new bin
        if not packed:
            bins.append([item])
    return bins

def print_bins(bins):
    for i, bin_contents in enumerate(bins):
        print(f'Bin {i + 1}: {bin_contents}')

if __name__ == "__main__":
    # Example input: item sizes and bin capacity
    item_sizes = [3, 5, 2, 8, 7, 4, 1]
    bin_capacity = 10

    # Apply First Fit heuristic
    packed_bins = first_fit(item_sizes, bin_capacity)

    # Print the result
    print("Original item sizes:", item_sizes)
    print("Bin Capacity:", bin_capacity)
    print("\nResult:")
    print_bins(packed_bins)
```

Execution

```
Original item sizes: [3, 5, 2, 8, 7, 4, 1]
Bin Capacity: 10

Result:
Bin 1: [3, 5, 2]
Bin 2: [8, 1]
Bin 3: [7]
Bin 4: [4]
```

What is the difference between bin packing problem and knapsack problem?

One of the most well-known packing problems is bin-packing, in which there are multiple containers (called bins) of equal capacity. Unlike the multiple knapsack problem, the number of bins is not fixed. Instead, the goal is to find the smallest number of bins that will hold all the items

CNF (Conjunctive Normal Form):

Conjunctive Normal Form (CNF) is a standard way of representing logical formulas in propositional logic. In CNF, a formula is expressed as a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals. A literal is either a variable or its negation. The CNF form is useful in various areas, including formal verification, artificial intelligence, and automated theorem proving.

A logical formula in CNF looks like this:

$(x1 \vee \neg x2) \wedge (\neg x1 \vee x3 \vee x4) \wedge (\neg x2 \vee \neg x3)$

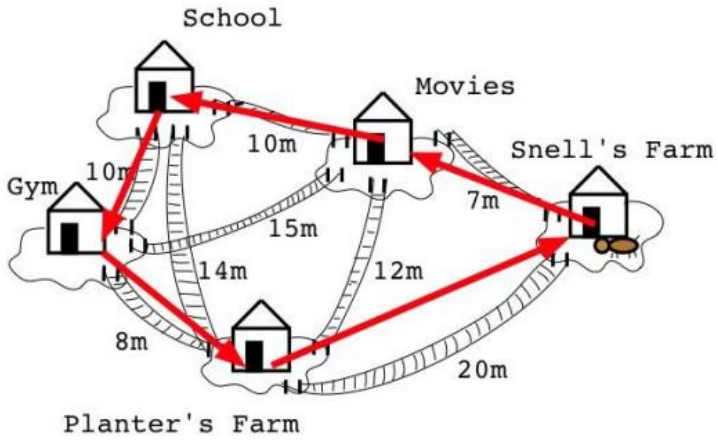
Here:

- x1, x2, x3, x4 are variables.
- $\vee$ represents logical OR.
- $\wedge$ represents logical AND.
- $\neg$ represents logical NOT.

CNF is important in the context of the Boolean Satisfiability Problem (SAT), as it provides a standardized way to represent logical formulas.

Traveling Salesman Problem (TSP)

- Optimization problem: Given a set of cities and their pairwise distances, find a tour of lowest cost that visits each city exactly once.
- Decision problem: is there a traveling salesman tour with cost at most *k*



Finally, define the *Traveling Salesman Problem (TSP)*: **given** a complete graph on n vertices, an assignment of positive integer weights to all edges, and an integer W, **decide** if there exists a Hamiltonian cycle in G of weight at most W, where the weight of the cycle is the sum of the weights of its edges.

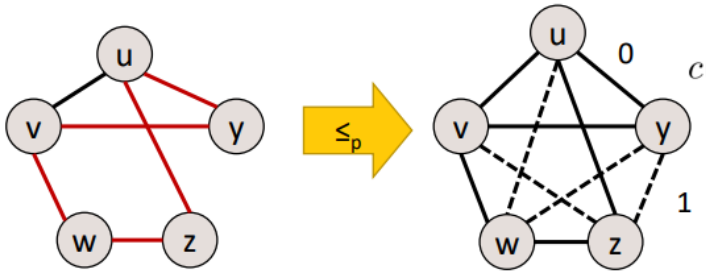
We'll show TSP is NP-complete, via the reduction $HamCycle \leq_p TSP$. The reduction maps a graph G on n nodes to a complete graph on n nodes, with the weights 1 for all edges of G, and weights $n+1$ for all non-edges of G. We also set $W = n$.

Observe that if G has a Hamiltonian cycle, then this cycle is a tour of weight n in our TSP instance. Conversely, if there is a tour of weight at most n, that tour can't use any non-edges of G (which would make its total weight at least $n+1$). Thus, the original graph G must have a Hamiltonian cycle.

TSP ∈ NP-Complete

TSP = {<G, c, k>: G = (V,E) is a complete graph, c is a cost function for edges, G has a traveling-salesman tour with cost at most k}

- Is TSP ∈ NP-Complete? HAM-CYCLE $\leq_p$ TSP
- Construct a reduction *f* transforming every HAM-CYCLE instance to a TSP instance (polynomial-time reduction)
- **G contains a Hamiltonian cycle $h = \langle v_1, v_2, \ldots, v_n, v_1 \rangle \Leftrightarrow \langle v_1, v_2, \ldots, v_n, v_1 \rangle$ is a traveling-salesman tour with cost 0**

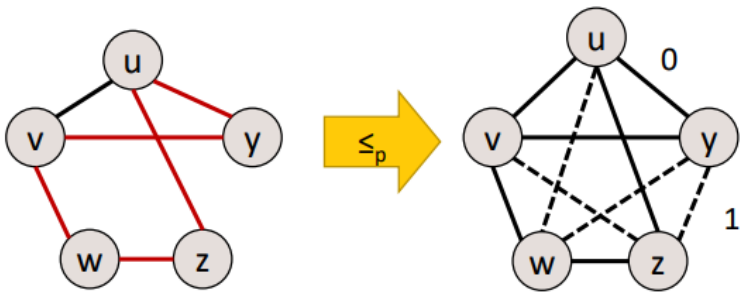


$$c(i,j) = \begin{cases} 0 & \text{if } (i,j) \in E \\ 1 & \text{if } (i,j) \notin E \end{cases}$$

41

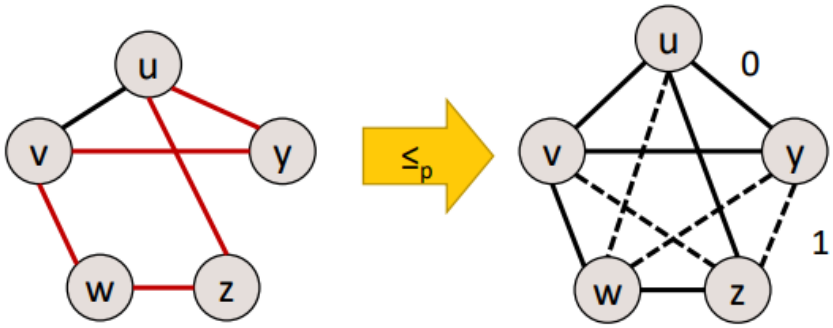
HAM-CYCLE $\leq_p$ TSP

- Correctness proof: ***x* ∈ HAM-CYCLE → *f*(*x*) ∈ TSP**
- If Hamiltonian cycle is $h = \langle v_1, v_2, \ldots, v_n, v_1 \rangle$
- → *h* is also a tour in the transformed TSP instance
- → The distance of the tour *h* is *0* since there are *n* consecutive edges in *E*, and so has distance 0 in *f*(*x*)
- → *f*(*x*) ∈ TSP (*f*(*x*) has a TSP tour with cost ≤ 0)



HAM-CYCLE $\leq_p$ TSP

- Correctness proof: ***f*(*x*) ∈ TSP → *x* ∈ HAM-CYCLE**
- **After reduction**, if a TSP tour with cost ≤ 0 as $<v_1, v_2, \ldots, v_n, v_1>$
- → The tour contains only edges in *E*
- → Thus, $<v_1, v_2, \ldots, v_n, v_1>$ is a Hamiltonian cycle



Approximation Algorithms for *NP* -Hard Problems

In this section, we discuss a different approach to handling difficult problems of combinatorial optimization, such as the traveling salesman problem and the knapsack problem. As we pointed out in Section 11.3, the decision versions of these problems are *NP*-complete. Their optimization versions fall in the class of ***NP-hard problems***—problems that are at least as hard as *NP*-complete problems.[2] Hence, there are no known polynomial-time algorithms for these problems, and there are serious theoretical reasons to believe that such algorithms do not exist. What then are our options for handling such problems, many of which are of significant practical importance?

If an instance of the problem in question is very small, we might be able to solve it by an exhaustive-search algorithm (Section 3.4). Some such problems can be solved by the dynamic programming technique. But even when this approach works in principle, its practicality is limited by dependence on the instance parameters being relatively small. The discovery of the branch-and-bound technique has proved to be an important breakthrough, because this technique makes it possible to solve many large instances of difficult optimization problems in an acceptable amount of time. However, such good performance cannot usually be guaranteed.

There is a radically different way of dealing with difficult optimization problems: solve them approximately by a fast algorithm. This approach is particularly appealing for applications where a good but not necessarily optimal solution will suffice. Besides, in real-life applications, we often have to operate with inaccurate data to begin with. Under such circumstances, going for an approximate solution can be a particularly sensible choice.

Although approximation algorithms run a gamut in level of sophistication, most of them are based on some problem-specific heuristic. A heuristic is a common-sense rule drawn from experience rather than from a mathematically proved assertion. For example, going to the nearest unvisited city in the traveling salesman problem is a good illustration of this notion. We discuss an algorithm based on this heuristic later in this section.

Of course, if we use an algorithm whose output is just an approximation of the actual optimal solution, we would like to know how accurate this approximation is. We can quantify the accuracy of an approximate solution sa to a problem of minimizing some function f by the size of the relative error of this approximation,

$$re(s_a) = \frac{f(s_a) - f(s^*)}{f(s^*)},$$

where s^* is an exact solution to the problem. Alternatively, since $re(s_a) = f(s_a)/f(s^*) - 1$, we can simply use the ***accuracy ratio***

$$r(s_a) = \frac{f(s_a)}{f(s^*)}$$

as a measure of accuracy of s_a. Note that for the sake of scale uniformity, the accuracy ratio of approximate solutions to maximization problems is usually computed as

$$r(s_a) = \frac{f(s^*)}{f(s_a)}$$

to make this ratio greater than or equal to 1, as it is for minimization problems. Obviously, the closer r(sa) is to 1, the better the approximate solution is.

For most instances, however, we cannot compute the accuracy ratio, because we typically do not know f (s∗), the true optimal value of the objective function. Therefore, our hope should lie in obtaining a good upper bound on the values of r(sa). This leads to the following definitions.

DEFINITION A polynomial-time approximation algorithm is said to be a c-approximation algorithm, where c ≥ 1, if the accuracy ratio of the approximation it produces does not exceed c for any instance of the problem in question:

$$r(s_a) \leq c.$$

The best (i.e., the smallest) value of c for which inequality (12.3) holds for all instances of the problem is called the performance ratio of the algorithm and denoted RA.

The performance ratio serves as the principal metric indicating the quality of the approximation algorithm. We would like to have approximation algorithms with RA as close to 1 as possible. Unfortunately, as we shall see, some approxima-tion algorithms have infinitely large performance ratios (RA = ∞). This does not necessarily rule out using such algorithms, but it does call for a cautious treatment of their outputs.

There are two important facts about difficult combinatorial optimization problems worth keeping in mind. First, although the difficulty level of solving most such problems exactly is the same to within a polynomial-time transforma-tion of one problem to another, this equivalence does not translate into the realm of approximation algorithms. Finding good approximate solutions is much easier for some of these problems than for others. Second, some of the problems have special classes of instances that are both particularly important for real-life appli-cations and easier to solve than their general counterparts. The traveling salesman problem is a prime example of this situation.

The Travelling Salesperson Problem

The Travelling Salesperson Problem (TSP) is defined as follows:

- INPUT: A distance function $d : [n] \times [n] \to \mathbb{R}^{\geq 0}$ between n vertices of a complete graph.
- OUTPUT: Find a tour $v_0, v_1, \ldots, v_n$ of the vertices s.t. $v_0 = v_n$ and each vertex appears exactly once in the list $\{v_0, v_1, \ldots, v_{n-1}\}$.
- OBJECTIVE: Minimize the total length $\sum_{i=0}^{n-1} d(v_i, v_{i+1})$ of the tour.

The Metric-TSP problem is the special case of this problem where the input distance function is a metric: i.e it is a symmetric function satisfying the triangle inequality:

$$\forall x, y, z \in [n] \quad d(x, z) \leq d(x, y) + d(y, z)$$

It turns out that the general TSP (without assuming the metric property) cannot have an efficient α-approximation algorithm for any α that is polynomial-time computable unless P = NP. This follows from a simple reduction from the well-known NP-complete problem called the Hamiltonian Cycle problem: the input here is a directed graph and we need to check if the graph contains a Hamiltonian cycle, a cycle in which each vertex of the graph appears exactly once.

Claim 1:. For any α(n) that can be computed in time poly(n), the TSP is not α(n) approximable unless P = NP.

Proof. We prove the claim by showing that if the TSP does have an efficient α(n)-approximation algorithm, then we can use such an algorithm to design an algorithm that solves the Hamiltonian cycle problem. Given an input graph G = ([n], E) to the Hamiltonian cycle problem, we construct an instance of the TSP as follows: a distance function is defined on [n] with

$$d(i, j) = \begin{cases} 1, & \text{if } (i, j) \in E, \\ \alpha(n) \cdot n + 1 & \text{otherwise.} \end{cases}$$

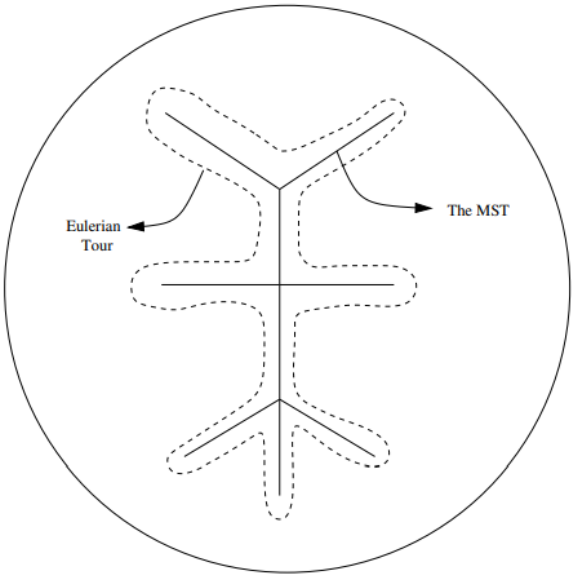
If G has a Hamiltonian cycle, then the optimum of the instance of the TSP problem is at most n; otherwise, the optimum is at least α(n)·n+ 1. If the TSP were α(n) approximable, we would be able to efficiently distinguish between these two cases and hence, we would have a polynomial-time algorithm for the Hamiltonian cycle problem which would imply that P = NP.

Note that the instance of TSP constructed in the above reduction is not an instance of Metric-TSP. Hence, we can still hope for an efficient algorithm to approximate the optimum to this problem. Indeed, we now show that this problem has a 3/2-approximation algorithm.

We start by describing a simpler 2-approximation algorithm for the problem. Let G be the weighted graph corresponding to a given instance of the TSP (the weight of the edge (i, j) is d(i, j)). As in the case of vertex cover, we would first like an efficiently computable lower bound to the optimum. This lower bound, in the case of the Metric TSP, is the weight of the Minimum Spanning Tree (MST) of G. To see that this is indeed a lower bound, note that removing any edge from any TSP tour of the vertices gives us a spanning tree of G, the weight of which is of course at least the weight of the MST of G.

Now, we use the MST of G to compute a short tour of all the vertices of G as follows:

- Find an MST T of G.
- Consider the subgraph H of G obtained by adding to T edges (i, j) for each i, j such that (j, i) ∈ T.
- Since the above graph is Eulerian, we can efficiently find an Eulerian tour of H. Output this tour.



Finding an Eulerian Tour from the MST

Note that the weight of the tour output by the above algorithm is twice the weight of T and hence at most twice the weight of the optimum TSP tour. To obtain a TSP tour of at most the same weight, we use the following useful fact about instances of Metric TSP.

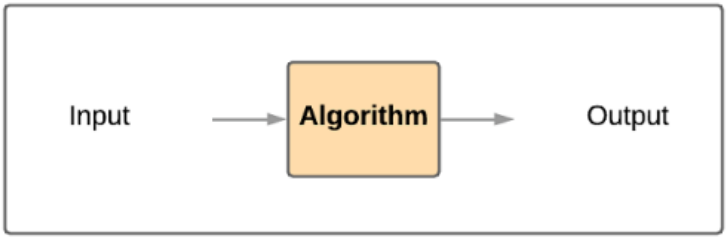
Lemma 1.. Let (G, d) be an instance of Metric TSP. Given any tour v0, v1, . . . , vm of all the vertices of G, one can find efficiently a TSP tour u0, u1, . . . , un of at most the same weight as the given tour.
Proof. The proof is simple. We define the TSP tour u0, u1, . . . , un to be a subsequence of v0, v1, . . . , vm as follows. Let u0 = v0. Say ui has been defined and is equal to vj . Then, we define ui+1 to be vk for the least k such that k > j and vj has not yet been visited in the path u0, u1, . . . , ui; if no such k exists, then we set un = vm. It is easy to prove, by the triangle inequality, that each such "short-cut" we take can only be of shorter length than the walk along the vi 's. Hence, the length of the TSP tour obtained is at most the length of the input tour.

$$\mathrm{OPT}(G,d) \leq A(G,d) \leq 2\mathrm{OPT}(G,d)$$

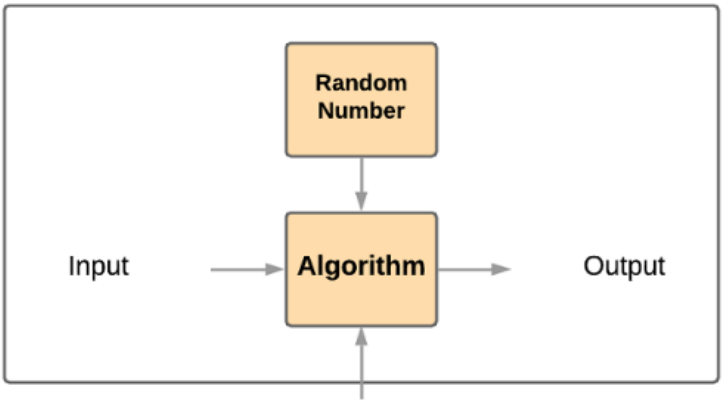
Randomized algorithm
Randomized algorithm is a different design approach taken by the standard algorithms where few random bits are added to a part of their logic. They are different from deterministic algorithms;

- Deterministic algorithms follow a definite procedure to get the same output every time an input is passed
- 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.



Deterministic
Algorithm Flowchart



Randomized Algorithm
Flowchart

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.

- 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.

- 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

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.

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.

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.

The Solovay-Strassen Primality Test
A primality tester is an algorithm that checks whether a given integer is prime or not. This is a fairly common problem, especially in cyber security and cryptography, and many other applications that rely on the special properties of prime numbers.

This test was developed by Robert M. Solovay and Volker Strassen, and is a probabilistic test to determine if a number is composite or probably prime.

Before we dive in to the algorithm, a bit of mathematics and modular arithmetic terms need to be discussed:

Key Concepts
Legendre symbol (a/p):
It is a function defined on a pair of integers a and p such that-

(a/p) = 0; if a≡0(mod p)
(a/p) = 1; if k exists such that k^2 = a mod p exists
(a/p) = -1; otherwise

It follows Euler's criterion (see references), a fact we shall make use of later.
Jacobi symbol (a/n):
It is a generalization of Legendre's symbol. For any integer a and any positive odd integer n, the Jacobi symbol(a/n)is defined as the product of the Legendre symbols corresponding to the prime factors of n:
(a/n) = (a/p1)k1.(a/p2)k2.(a/p3)k3.....(a/pn)kn

Where n = p1k1.p2k2.....pnkn

$$(a/n) = (a/p_1)^{k_1}.(a/p_2)^{k_2}.(a/p_3)^{k_3}.....(a/p_n)^{k_n}$$

$$\text{Where } n = p_1^{k_1}.p_2^{k_2}.....p_n^{k_n}$$

Certain properties of the Jacobi symbol will become useful to us when it comes to finding out if a number is prime or not. For odd primes, (a/n) = (a/p).

The Jacobi symbol for a number can be computed in $O((\log n)^2)$ time. For more properties, see references below.

Algorithm for Solovay-Strassen Primality Test
The test works like this-

i. We first select a number n to test for its primality and a random number a which lies in the range of (2, n-1) and then compute the Jacobi symbol (a/n).
ii. If n is a prime number then the Jacobi will be equal to the Legendre and it will satisfy Euler's criterion.
iii. If it does not satisfy the given condition then n is composite and the program will stop.
iv. Like other probabilistic primality tests its accuracy is also directly proportional to the number of iterations. To get more accurate results, we shall run the test multiple times.

This random number a we took is called an Euler witness for n - it is a witness for the compositeness of n. Using fast algorithms for modular exponentiation, the running time of this algorithm is $O(k \cdot \log^3 n)$, where k is the number of different values of a we test.

Here is the pseudo-code implementation of the Solovay-Strassen Primality Test:

```
# check if integer p is prime or not, run for k iterations
boolean SolovayStrassen(double p, int k):
    if(p<2):
        return false
    if(p!=2 and p%2 == 0):
        return false

    for(int i = 0; i<k; i++):
        double a = rand() % (p-1) + 1
        # Jacobi() calculates the Jacobi symbol (a/n)
        # modExp() performs modular exponentiation
        double jacobian = (p + Jacobi(a, p)) % p
        double mod = modExp(a, (p-1)/2, p)

        if(!jacobian || mod != jacobian)
            return false
    return true
```

Randomized QuickSort Algorithm

Randomized QuickSort is a variation of the traditional QuickSort algorithm that introduces randomization to improve its average-case performance.

QuickSort Overview:

QuickSort is a popular sorting algorithm known for its efficiency and simplicity. It follows the divide-and-conquer paradigm, breaking the sorting problem into smaller subproblems, sorting them, and combining the results. The basic steps of the QuickSort algorithm are:

1. **Partitioning:**
   - Select a pivot element from the array.
   - Partition the array into two subarrays: elements less than the pivot and elements greater than the pivot.
   - The pivot is now in its final sorted position.
2. **Recursion:**
   - Recursively apply the QuickSort algorithm to the subarrays on either side of the pivot.
3. **Combining:**
   - As the recursion unwinds, the sorted subarrays are combined to produce the final sorted array.

Randomized QuickSort:

Randomized QuickSort introduces an element of randomness during the pivot selection step to mitigate the worst-case scenarios that can occur with deterministic pivot selection strategies.

Steps of Randomized QuickSort:

1. **Base Case:**
   - If the array has one element or is empty, it is already sorted, and no further action is needed.
2. **Random Pivot Selection:**
   - Randomly select a pivot element from the array. This randomization helps avoid worst-case scenarios when the array is nearly sorted or already sorted.
3. **Partitioning:**
   - Partition the array into three parts:
     - Elements less than the pivot.
     - Elements equal to the pivot.
     - Elements greater than the pivot.
   - The pivot is now in its final sorted position.
4. **Recursion:**
   - Recursively apply the Randomized QuickSort algorithm to the subarrays on either side of the pivot.
5. **Combining:**
   - As the recursion unwinds, the sorted subarrays are combined to produce the final sorted array.

```
RandomizedQuickSort(arr):
    if length(arr) <= 1:
        return arr
    else:
        randomly select pivot from arr
        less = elements in arr less than pivot
        equal = elements in arr equal to pivot
        greater = elements in arr greater than pivot
        return RandomizedQuickSort(less) + equal + RandomizedQuickSort(greater)
```

AREC

Python Code to implement Randomized QuickSort

```
import random

def randomized_quick_sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = random.choice(arr)  # Randomly select a pivot element
        less = [x for x in arr if x < pivot]
        equal = [x for x in arr if x == pivot]
        greater = [x for x in arr if x > pivot]
        return randomized_quick_sort(less) + equal + randomized_quick_sort(greater)

# Example usage:
arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
sorted_arr = randomized_quick_sort(arr)
print("Original array:", arr)
print("Sorted array:", sorted_arr)
```

Analysis:

- **Average-Case Complexity:**
  - Randomized QuickSort has an average-case time complexity of O(nlog⁡n)O(nlogn), similar to deterministic QuickSort.
  - The randomization reduces the likelihood of encountering worst-case scenarios, providing a more consistent performance.
- **Worst-Case Scenario:**
  - The worst-case time complexity is still O(n2)O(n2) in the presence of a bad random pivot selection strategy. However, the probability of encountering the worst case is significantly reduced with a good randomization strategy.
- **Space Complexity:**
  - Randomized QuickSort is an in-place sorting algorithm, meaning it sorts the elements within the input array without requiring additional memory.

Finding the kth smallest element

Finding the kth smallest element in an array is a common problem in computer science and algorithms. There are several approaches to solve this problem, each with its own trade-offs in terms of time and space complexity. Here, I'll describe two common methods: the QuickSelect algorithm and the use of a min-heap.

1. QuickSelect Algorithm:

QuickSelect is an algorithm that is based on the same underlying principle as QuickSort. It efficiently finds the kth smallest element in an unordered list.

Steps:

1. **Select a Pivot:**
   - Randomly select a pivot element from the array.
2. **Partition:**
   - Partition the array into two parts: elements less than the pivot and elements greater than the pivot.
3. **Recursive Calls:**
   - Depending on the position of the pivot after partitioning:
     - If the pivot is at index k, return it as the kth smallest element.
     - If the pivot is to the left of index k, recursively apply QuickSelect to the right subarray.
     - If the pivot is to the right of index k, recursively apply QuickSelect to the left subarray.
4. **Repeat:**
   - Repeat the process until the kth smallest element is found.

Python Program to implement QuickSelect Algorithm

```
import random

def quickselect(arr, k):
    if len(arr) == 1:
        return arr[0]

    pivot = random.choice(arr)
    left = [x for x in arr if x < pivot]
    right = [x for x in arr if x > pivot]
    equal = [x for x in arr if x == pivot]

    if k < len(left):
        return quickselect(left, k)
    elif k < len(left) + len(equal):
        return equal[0]
    else:
        return quickselect(right, k - len(left) - len(equal))
```

2. Min-Heap (Priority Queue):

Another approach is to use a min-heap (or a priority queue) to efficiently find the kth smallest element.

Steps:

1. **Build a Min-Heap:**
   - Create a min-heap and insert the first k elements from the array.
2. **Iterate Through the Rest:**
   - For each remaining element in the array:
     - If the element is smaller than the root of the min-heap, replace the root with the new element.

- Re-heapify to maintain the min-heap property.

3. **Result:**
   - After processing all elements, the root of the min-heap is the kth smallest element.

```
import heapq

def kth_smallest_min_heap(arr, k):
    heap = arr[:k]
    heapq.heapify(heap)

    for num in arr[k:]:
        if num < heap[0]:
            heapq.heapreplace(heap, num)

    return heap[0]
```

Comparison:

- **QuickSelect:**
  - Expected time complexity: O(n)on average.
  - Worst-case time complexity: $O(n^2)$
  - In-place, but not stable.
- **Min-Heap:**
  - Expected time complexity: O(n+klogn).
  - Extra space needed for the heap.

APEC