Outline

**Turing Machine, Turing Degree**

- Polynomial and non polynomial problems
- Deterministic and non deterministic algorithms
- P, NP, NP hard and NP Complete
- NP Complete Proof - 3-SAT & Vertex Cover
- NP hard Problem - Hamiltonian Cycle
- Menagerie of complexity classes of Turing Degree
- Randomization - Sorting algorithm
- Approximation - TSP and Max Clique
Turing Machin
Turing machine

- A Turing machine is a hypothetical device that manipulates symbols on a strip of tape according to a table of rules.
- Despite its simplicity, a Turing machine can be adapted to simulate the logic of any computer algorithm, and is particularly useful in explaining the functions of a CPU inside a computer.
Turing Degree

- A measure of the level of algorithmic unsolvability of the decision problem of whether a given set of natural numbers contains any given number.
<table>
<thead>
<tr>
<th>Outline</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Turing Machine, Turing Degree</strong></td>
</tr>
<tr>
<td><strong>Polynomial and non polynomial problems</strong></td>
</tr>
<tr>
<td>Deterministic and non deterministic algorithms</td>
</tr>
<tr>
<td>P, NP, NP hard and NP Complete</td>
</tr>
<tr>
<td>NP Complete Proof - 3-SAT &amp; Vertex Cover</td>
</tr>
<tr>
<td>NP hard Problem - Hamiltonian Cycle</td>
</tr>
<tr>
<td>Menagerie of complexity classes of Turing Degree</td>
</tr>
<tr>
<td>Randomization - Sorting algorithm</td>
</tr>
<tr>
<td>Approximation - TSP and Max Clique</td>
</tr>
</tbody>
</table>
Polynomial Problems

- **Polynomial** is an expression consisting of variables (or indeterminates) and coefficients, that involves only the operations of addition, subtraction, multiplication, and non-negative integer exponents.

- An example of a polynomial of a single indeterminate (or variable), $x$, is $x^2 - 4x + 7$.

- An Example of Polynomial algorithm is $0(\log n)$
Non-Polynomial Problems

- The set or property of problems for which no polynomial-time algorithm is known.
- This includes problems for which the only known algorithms require a number of steps which increases exponentially with the size of the problem, and those for which no algorithm at all is known.
- Within these two there are problems which are "provably difficult" and "provably unsolvable".
Outline

- Turing Machine, Turing Degree
- Polynomial and non polynomial problems
- Deterministic and non deterministic algorithms
  - P, NP, NP hard and NP Complete
  - NP Complete Proof - 3-SAT & Vertex Cover
  - NP hard Problem - Hamiltonian Cycle
  - Menagerie of complexity classes of Turing Degree
  - Randomization - Sorting algorithm
  - Approximation - TSP and Max Clique
Deterministic Algorithm

- **Deterministic algorithm** is an algorithm which, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states.
Nondeterministic algorithm

- **Nondeterministic algorithm** is an algorithm that, even for the same input, can exhibit different behaviors on different runs, as opposed to a deterministic algorithm.

- An algorithm that solves a problem in **nondeterministic polynomial time** can run in polynomial time or exponential time depending on the choices it makes during execution.

- The nondeterministic algorithms are often used to find an approximation to a solution, when the exact solution would be too costly to obtain using a deterministic one.
## Outline

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Turing Machine , Turing Degree</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Polynomial and non polynomial problems</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Deterministic and non deterministic algorithms</strong></td>
<td></td>
</tr>
<tr>
<td><strong>P ,NP, NP hard and NP Complete</strong></td>
<td></td>
</tr>
<tr>
<td>NP Complete Proof- 3-SAT &amp; Vertex Cover</td>
<td></td>
</tr>
<tr>
<td>NP hard Problem- Hamiltonian Cycle</td>
<td></td>
</tr>
<tr>
<td>Menagerie of complexity classes of Turing Degree</td>
<td></td>
</tr>
<tr>
<td>Randomization- Sorting algorithm</td>
<td></td>
</tr>
<tr>
<td>Approximation-TSP and Max Clique</td>
<td></td>
</tr>
</tbody>
</table>
P Class

- The informal term *quickly*, used above, means the existence of an algorithm for the task that runs in **polynomial time**.
- The general class of questions for which some algorithm can provide an answer in polynomial time is called "class $\mathbf{P}$" or just "$\mathbf{P}$".
NP Class

- For some questions, there is no known way to find an answer quickly, but if one is provided with information showing what the answer is, it is possible to verify the answer quickly.
- The class of questions for which an answer can be verified in polynomial time is called **NP**.
NP- Hard Class

- **NP-hard** (Non-deterministic Polynomial-time hard), is a class of problems that are, informally, "at least as hard as the hardest problems in **NP**".
- More precisely, a problem $H$ is NP-hard when every problem $L$ in NP can be reduced in polynomial time to $H$.
- As a consequence, finding a polynomial algorithm to solve any NP-hard problem would give polynomial algorithms for all the problems in NP, which is unlikely as many of them are considered hard.
NP Complete Class

- In computational complexity theory, a decision problem is NP-complete when it is both in NP and NP-hard.
- The set of NP-complete problems is often denoted by NP-C or NPC.
- The abbreviation NP refers to "nondeterministic polynomial time".
Relation between P, NP, NP hard and NP Complete
**Definition**  Let $L_1$ and $L_2$ be problems. $L_1$ reduces to $L_2$ (also written $L_1 \preceq L_2$) if and only if there is a way to solve $L_1$ by a deterministic polynomial time algorithm using a deterministic algorithm that solves $L_2$ in polynomial time.

This definition implies that if we have a polynomial time algorithm for $L_2$ then we can solve $L_1$ in polynomial time. One may readily verify that $\preceq$ is a transitive relation (i.e. if $L_1 \preceq L_2$ and $L_2 \preceq L_3$ then $L_1 \preceq L_3$).

Reducibility
Outline

- Turing Machine, Turing Degree
- Polynomial and non-polynomial problems
- Deterministic and non-deterministic algorithms
- P, NP, NP hard and NP Complete
- NP Complete Proof - 3-SAT & Vertex Cover
- NP hard Problem - Hamiltonian Cycle
- Menagerie of complexity classes of Turing Degree
- Randomization - Sorting algorithm
- Approximation - TSP and Max Clique
3-Satisfiability (3-SAT)

**Instance**: A collection of clause $C$ where each clause contains exactly 3 literals, boolean variable $v$.

**Question**: Is there a truth assignment to $v$ so that each clause is satisfied?

**Note**: This is a more restricted problem than normal SAT. If 3-SAT is NP-complete, it implies that SAT is NP-complete but *not* visa-versa, perhaps longer clauses are what makes SAT difficult?

1-SAT is trivial.

2-SAT is in P
Theorem: 3-SAT is NP-Complete

Proof:

1) 3-SAT is NP. Given an assignment, we can just check that each clause is covered.

2) 3-SAT is hard. To prove this, a reduction from SAT to 3-SAT must be provided. We will transform each clause independently based on its length.
Reducing SAT to 3-SAT

• Suppose a clause contains $k$ literals:

• if $k = 1$ (meaning $C_i = \{z_1\}$ ), we can add in two new variables $v_1$ and $v_2$, and transform this into 4 clauses:
  • $\{v_1, v_2, z_1\}$ $\{v_1, \neg v_2, z_1\}$ $\{\neg v_1, v_2, z_1\}$ $\{\neg v_1, \neg v_2, z_1\}$

• if $k = 2$ ( $C_i = \{z_1, z_2\}$ ), we can add in one variable $v_1$ and 2 new clauses: $\{v_1, z_1, z_2\}$ $\{\neg v_1, z_1, z_2\}$

• if $k = 3$ ( $C_i = \{z_1, z_2, z_3\}$ ), we move this clause as-is.
Continuing the Reduction…

if $k > 3$ ($C_i = \{z_1, z_2, \ldots, z_k\}$) we can add in $k - 3$ new variables ($v_1, \ldots, v_{k-3}$) and $k - 2$ clauses:

$$\{z_1, z_2, v_1\} \{\neg v_1, z_3, v_2\} \{\neg v_2, z_4, v_3\} \ldots \{\neg v_{k-3}, z_{k-1}, z_k\}$$

Thus, in the worst case, $n$ clauses will be turned into $n^2$ clauses. This cannot move us from polynomial to exponential time.

If a problem could be solved in $O(n^k)$ time, squaring the number of inputs would make it take $O(n^{2k})$ time.
Generalizations about SAT

Since any SAT solution will satisfy the 3-SAT instance and a 3-SAT solution can set variables giving a SAT solution, the problems are equivalent. If there were $n$ clauses and $m$ distinct literals in the SAT instance, this transform takes $O(nm)$ time, so SAT == 3-SAT.
Given a graph $G = (N, E)$ and an integer $k$, does there exist a subset $S$ of at most $k$ vertices in $N$ such that each edge in $E$ is touched by at least one vertex in $S$?

No polynomial-time algorithm is known in NP (short and verifiable solution):

- If a graph is “$k$-coverable”, there exists $k$-subset $S \subseteq N$ such that
  - each edge is touched by at least one of its vertices
  - Length of $S$ encoding is polynomial in length of $G$ encoding
- There exists a polynomial-time algorithm that verifies whether $S$ is a valid $k$-cover
  - Verify that $|S| \leq k$
  - Verify that, for any $(u, v) \in E$, either $u \in S$ or $v \in S$
NP-completeness

Reduction of 3-Sat to Vertex Cover:

Technique: component design

- For each variable a gadget (that is, a sub-graph) representing its truth value
- For each clause a gadget representing the fact that one of its literals is true
- Edges connecting the two kinds of gadget

Gadget for variable \( u \):

\[
\begin{array}{c}
\bullet p_u \\
\hline
\bullet n_u
\end{array}
\]

- One vertex is sufficient and necessary to cover the edge

Gadget for clause \( c \):

\[
\begin{array}{c}
\bullet f_c \\
\hline
\bullet s_c \\
\bullet t_c
\end{array}
\]

- Two vertices are sufficient and necessary to cover the three edges

\( k = n + 2m \), where \( n \) is number of variables and \( m \) is number of clauses
Connections between variable and clause gadgets

- First (second, third) vertex of clause gadget connected to vertex corresponding to first (second, third) literal of clause
- Example: \((x_1 \lor x_2 \lor x_3) \land (\overline{x_1} \lor x_2 \lor x_3) \land (\overline{x_1} \lor x_2 \lor x_3)\)

Idea: if first (second, third) literal of clause is true (taken), then first (second, third) vertex of clause gadget has not to be taken in order to cover the edges between the gadgets
Proof of correctness

Show that Formula satisfiable $\Rightarrow$ Vertex cover exists:

- Include in $S$ all vertices corresponding to true literals
- For each clause, include in $S$ all vertices of its gadget but the one corresponding to its first true literal
- Example
  - $(x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor x_3) \land (x_1 \lor x_2 \lor \neg x_3)$
  - $x_1$ true, $x_2$ and $x_3$ false

Show that Vertex cover exists $\Rightarrow$ Formula satisfiable:

- Assign value true to variables whose $p$-vertices are in $S$
- Since $k = n + 2m$, for each clause at least one edge connecting its gadget to the variable gadgets is covered by a variable vertex
  - Clause is satisfied
Hamiltonian Cycle Problem

A Hamiltonian cycle in a graph is a cycle that visits each vertex exactly once

Problem Statement

Given  A directed graph $G = (V,E)$

To Find  If the graph contains a Hamiltonian cycle
Hamiltonian Cycle Problem

Hamiltonian Cycle Problem is NP-Complete

Hamiltonian Cycle Problem is in NP. Given a directed graph $G=(V,E)$, and a certificate containing an ordered list of vertices on a Hamiltonian Cycle. It can be verified in polynomial time that the list contains each vertex exactly once and that each consecutive pair in the ordering is joined by an edge.
Hamiltonian Cycle Problem

Hamiltonian Cycle Problem is NP Hard

3-SAT $\leq_p$ Hamiltonian Cycle

We begin with an arbitrary instance of 3-SAT having variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_k$

We model one by one, the $2^n$ different ways in which variables can assume assignments, and the constraints imposed by clauses.
Hamiltonian Cycle Problem

To correspond to the $2^n$ truth assignments, we describe a graph containing $2^n$ different Hamiltonian cycles.

The graph is constructed as follows:

Construct $n$ paths $P_1, \ldots, P_n$.

- Each $P_i$ consists of nodes $v_{i1}, \ldots, v_{ib}$, where $b = 2k$ (k being the number of clauses)
Hamiltonian Cycle Problem

• Draw edges from $v_{ij}$ to $v_{i,j+1}$
Hamiltonian Cycle Problem

• Draw edges from $v_{i,j+1}$ to $v_{i,j}$
Hamiltonian Cycle Problem

• For each $i = 1, 2, \ldots, n-1$, define edges from $v_{i1}$ to $v_{i+1,1}$ and to $v_{i+1,b}$.
• Also, define edges from $v_{ib}$ to $v_{i+1,1}$ and $v_{i+1,b}$
Add two extra nodes $s$ and $t$. Define edges from $s$ to $v_{11}$ and $v_{1b}$, from $v_{n1}$ and $v_{nb}$ to $t$, and from $t$ to $s$. 
Hamiltonian Cycle Problem

Observations:

- Any Hamiltonian cycle must use the edge \((t,s)\)
- Each \(P_i\) can either be traversed left to right or right to left. This gives rise to \(2^n\) Hamiltonian cycles.
- We have therefore modeled the \(n\) independent choices of how to set each variable; if \(P_i\) is traversed left to right, \(x_i=1\), else \(x_i=0\)
Hamiltonian Cycle Problem

We now add nodes to model the clauses
Consider the clause $C = (x_1' \lor x_2 \lor x_3')$

What is the interpretation of the clause?
The path $P_1$ should be traversed right to left, or $P_2$ should be traversed left to right, or $P_3$ right to left.
We add a node that does this
Hamiltonian Cycle Problem

In general

• We define a node $c_j$ for each clause $C_j$.
• In each path $P_i$, positions $2j-1$ and $2j$ are reserved for variables that participate in clause $C_j$
• If $C_j$ contains $x_i$, add edges $(v_{i,2j-1}, c_j)$ and $(c_j, v_{i,2j})$
• If $C_j$ contains $x_i'$, add edges $(v_{i,2j}, c_j)$ and $(c_j, v_{i,2j-1})$

Gadget constructed!
Hamiltonian Cycle Problem

Consider an instance of 3-SAT having 4 variables: $x_1, x_2, x_3, x_4$

3 clauses $C_1: (x_1 \lor x_2 \lor x_3')$

$C_2: (x_2' \lor x_3 \lor x_4)$

$C_3: (x_1' \lor x_2 \lor x_4')$
Hamiltonian Cycle Problem

We reduce the given instance as follows:
\[ n = 4 \quad k = 3 \quad b = 2 \times 3 = 6 \]

Construct 4 paths \( P_1, P_2, P_3, P_4 \)
- \( P_1 \) consists of nodes \( v_{1,1}, v_{1,2}, \ldots, v_{1,6} \)
- \( P_2 \) consists of nodes \( v_{2,1}, v_{2,2}, \ldots, v_{2,6} \)
- \( P_3 \) consists of nodes \( v_{3,1}, v_{3,2}, \ldots, v_{3,6} \)
- \( P_4 \) consists of nodes \( v_{4,1}, v_{4,2}, \ldots, v_{4,6} \)
Hamiltonian Cycle Problem

**Claim:** 3-SAT instance is satisfiable if and only if $G$ has a Hamiltonian cycle

**Proof:** Part I

**Given** A satisfying assignment for the 3-SAT instance

If $x_i = 1$, traverse $P_i$ left to right, else right to left. Since each clause $C_j$ is satisfied by the assignment, there has to be at least one path $P_i$ that moves in the right direction to be able to cover node $c_j$. This $P_i$ can be spliced into the tour there via edges incident on $v_{i,2j-1}$ and $v_{i,2j}$
Hamiltonian Cycle Problem

Let us try to verify this with our example

**Given** A satisfying assignment for 3-SAT, say $x_1 = 1 \quad x_2 = 0 \quad x_3 = 1 \quad x_4 = 0$

Let us check out a corresponding Hamiltonian cycle
Hamiltonian Cycle Problem

Part II

Given  A Hamiltonian cycle in G.

Observe that if the cycle enters a node $c_j$ on an edge from $v_{i,2j-1}$ it must depart on an edge to $v_{i,2j}$.

Why?
Hamiltonian Cycle Problem

Because otherwise, the tour will not be able to cover this node while still maintaining the Hamiltonian property.

Similarly, if the path enters from $v_{i,2j}$, it has to depart immediately to $v_{i,2j-1}$. 
Hamiltonian Cycle Problem

However, in some situations it may so happen that the path enters $c_j$ from the first (or last) node of $P_i$ and departs at the first (or last) node of $P_{i+1}$.

In either case the following holds true:

The nodes immediately before and after any $c_j$ in the cycle are joined by an edge in $G$, say $e$.

Let us consider the following Hamiltonian cycle given on our graph
Hamiltonian Cycle Problem

Obtain a Hamiltonian cycle on the subgraph $G - \{c_1, \ldots, c_k\}$ by removing $c_j$ and adding ‘e’ as shown below.
Hamiltonian Cycle Problem

We now use this new cycle on the subgraph to obtain the truth assignments for the 3-SAT instance.

If it traverses \( P_i \) left to right, set \( x_i = 1 \), else set \( x_i = 0 \).

We therefore get the following assignments:

\[ x_1 = 1 \quad x_2 = 0 \quad x_3 = 0 \quad x_4 = 1 \]
Can we claim that the assignment thus determined would satisfy all clauses.

YES!

Since the larger cycle visited each clause node $c_j$, at least one $P_i$ was traversed in the right direction relative to the node $c_j$. 
Menagerie of complexity classes of Turing Degree

- In computer science and mathematical logic the Turing degree (named after Alan Turing) or degree of unsolvability of a set of natural numbers measures the level of algorithmic unsolvability of the set. The concept of Turing degree is fundamental in computability theory, where sets of natural numbers are often regarded as decision problems. The Turing degree of a set tells how difficult it is to solve the decision problem associated with the set, that is, to determine whether an arbitrary number is in the given set.
Menagerie of complexity classes of Turing Degree

- The menagerie is a dynamic diagram of (downward closed) classes of Turing degrees.
- Between 2002 and 2003, Bjørn Kjos-Hanssen put together a remarkable diagram of classes of Turing degrees.
Menagerie of complexity classes of Turing Degree
Menagerie of complexity classes of Turing Degree

- Let’s start at the outermost class of all recognizable problems, also known as **decision problems**.

- In our study of the **halting problem**, we have seen that some problems are simply undecidable.

- So that’s why in this chart the class of decidable problems is contained strictly inside the class of all problems. This is something we know and has been proven.

- Now you see lots more complexity classes nested inside of the decidable problems, such as **EXPSPACE**, which is the set of all problems decidable by a **Turing Machine** in \(O(2^p(n))\) tape space where \(p(n)\) is a polynomial of \(n\) (which is the size of the input).

- Now the diagram shows this as being strictly larger than **EXPTIME**, which is the set of problems decidable in exponential time.
Menagerie of complexity classes of Turing Degree

- EXPSPACE is a strict superset of PSPACE. By now you should be able to guess what the definition of PSPACE looks like: it is the set of problems decidable by a Turing machine in $O(p(n))$ of space.
- Here is something else we know and have already encountered in Tech Tuesday. We saw that the class of regular problems, which can be solved using finite state machines, is strictly smaller than the class CFL, which are the context free languages.
- PS There are many more complexity classes than are shown in the diagram above. You can find a comprehensive listing in the Complexity Zoo.
Outline

- Turing Machine, Turing Degree
- Polynomial and non-polynomial problems
- Deterministic and non-deterministic algorithms
- P, NP, NP hard and NP Complete
- NP Complete Proof - 3-SAT & Vertex Cover
- NP hard Problem - Hamiltonian Cycle
- Menagerie of complexity classes of Turing Degree
- Randomization - Sorting algorithm
- Approximation - TSP and Max Clique
Introduction to Randomized Algorithms
Why Randomness? (Contd..)

Making good decisions could be expensive.

A randomized algorithm is faster.

Consider a sorting procedure. Picking an element in the middle makes the procedure very efficient, but it is expensive (i.e. linear time) to find such an element.

Consider a sorting procedure. Picking an element in the middle makes the procedure very efficient, but it is expensive (i.e. linear time) to find such an element.

5 9 13 11 8 6 7 10

5 6 7 8 9 13 11 10

Picking a random element will do.
To summarize, we use randomness because…

- Avoid worst-case behavior: randomness can (probabilistically) guarantee average case behavior
- Efficient approximate solutions to intractable problems
- In many practical problems, we need to deal with HUGE input, and don’t even have time to read it once. But can we still do something useful?
Randomized algorithms make random rather than deterministic decisions.

The main advantage is that no input can reliably produce worst-case results because the algorithm runs differently each time.

These algorithms are commonly used in situations where no exact and fast algorithm is known.

Behavior can vary even on a fixed input.
Deterministic Algorithms

**Goal:** Prove for all input instances the algorithm solves the problem correctly and the number of steps is bounded by a polynomial in the size of the input.
Randomized Algorithms

- In addition to input, algorithm takes a source of random numbers and makes random choices during execution;

- Behavior can vary even on a fixed input;
Few applications

- Minimum spanning trees
  A linear time randomized algorithm,
  but no known linear time deterministic algorithm.

- Primality testing
  A randomized polynomial time algorithm,
  but it takes thirty years to find a deterministic one.

- Volume estimation of a convex body
  A randomized polynomial time approximation algorithm,
  but no known deterministic polynomial time approximation algorithm.
Types of Randomized algorithms

- Las Vegas
- Monte Carlo
LAS VEGAS

- Always gives the true answer.
- Running time is random.
- Running time is variable whose expectation is bounded (say by a polynomial).
- E.g. Randomized QuickSort Algorithm
Las Vegas Randomized Algorithms

**Goal:** Prove that for all input instances the algorithm solves the problem correctly and the expected number of steps is bounded by a polynomial in the input size.

**Note:** The expectation is over the random choices made by the algorithm.

Las Vegas Example-QUICKSORT
MONTE CARLO

- It may produce incorrect answer!
- We are able to bound its probability.
- By running it many times on independent random variables, we can make the failure probability arbitrarily small at the expense of running time.
- E.g. Randomized Mincut Algorithm
Quick Sort

- **Select**: pick an arbitrary element $x$ in $S$ to be the pivot.

- **Partition**: rearrange elements so that elements with value less than $x$ go to List $L$ to the left of $x$ and elements with value greater than $x$ go to the List $R$ to the right of $x$.

- **Recursion**: recursively sort the lists $L$ and $R$. 
Worst Case Partitioning of Quick Sort

\[ \Theta(n^2) \]
Best Case Partitioning of Quick Sort

\[ \Theta(n \log n) \]
Average Case of Quick Sort

\[ \Theta(n \lg n) \]
Randomized Quick Sort

**Randomized-Quicksort**(A, p, r)
1. if \( p < r \)
2. then \( q \leftarrow \text{Randomized-Partition}(A, p, r) \)
3. Randomized-Quicksort(A, p, q-1)
4. Randomized-Quicksort(A, q+1, r)

**Randomized-Partition**(A, p, r)
1. \( i \leftarrow \text{Random}(p, r) \)
2. exchange \( A[r] \leftrightarrow A[i] \)
3. return Partition(A, p, r)
Randomized Quick Sort


The pivot element is equally likely to be any of input elements.

For any given input, the behavior of Randomized Quick Sort is determined not only by the input but also by the random choices of the pivot.

We add randomization to Quick Sort to obtain for any input the expected performance of the algorithm to be good.
A Randomized Approach

- To improve QuickSort, *randomly* select $m$.
- Since half of the elements will be good splitters, if we choose $m$ at random we will get a 50% chance that $m$ will be a good choice.
- This approach will make sure that no matter what input is received, the expected running time is small.
Randomized Quicksort-Analysis

- Worst case runtime: $O(m^2)$
- *Expected runtime*: $O(m \log m)$.
- Expected runtime is a good measure of the performance of randomized algorithms, often more informative than worst case runtimes.
PROS

- Making a random choice is fast.
- An adversary is powerless; randomized algorithms have no worst case inputs.
- Randomized algorithms are often simpler and faster than their deterministic counterparts.
CONS

• In the worst case, a randomized algorithm may be very slow.
• There is a finite probability of getting incorrect answer.
• However, the probability of getting a wrong answer can be made arbitrarily small by the repeated employment of randomness.
• Getting true random numbers is almost impossible.
Outline

- Turing Machine, Turing Degree
- Polynomial and non-polynomial problems
- Deterministic and non-deterministic algorithms
- P, NP, NP hard and NP Complete
- NP Complete Proof- 3-SAT & Vertex Cover
- NP hard Problem- Hamiltonian Cycle
- Menagerie of complexity classes of Turing Degree
- Randomization- Sorting algorithm
- Approximation- TSP and Max Clique
A Solution to NP –complete Problems

- There are many important NP-Complete problems
  - There is no fast solution!
- But we want the answer …
  - If the input is small use backtrack.
  - Isolate the problem into P-problems!
  - Find the **Near-Optimal** solution in polynomial time.
Accuracy

- NP problems are often optimization problems
- It’s hard to find the EXACT answer
- Maybe we just want to know our answer is close to the exact answer?
Approximation Algorithms

- Can be created for optimization problems
- The exact answer for an instance is OPT
- The approximate answer will never be far from OPT

- We CANNOT approximate decision problems
Performance ratios

- We are going to find a Near-Optimal solution for a given problem.
- We assume two hypothesis:
  - Each potential solution has a positive cost.
  - The problem may be either a maximization or a minimization problem on the cost.
Performance ratios …

- If for any input of size \( n \), the cost \( C \) of the solution produced by the algorithm is within a factor of \( \rho(n) \) of the cost \( C^* \) of an optimal solution:

\[
\text{Max } \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n)
\]

- We call this algorithm as an \( \rho(n) \)-approximation algorithm.
Performance ratios …

- In Maximization problems:
  \[ \frac{C^*}{\rho(n)} \leq C \leq C^* \]

- In Minimization Problems:
  \[ C^* \leq C \leq \rho(n)C^* \]
  - \( \rho(n) \) is never less than 1.
  - A 1-approximation algorithm is the optimal solution.
  - The goal is to find a polynomial-time approximation algorithm with small constant approximation ratios.
Some examples:

- Traveling salesman problem.
- Max Clique/Vertex Cover
VERTEX-COVER

- Given a graph, G, return the smallest set of vertices such that all edges have an end point in the set
The vertex-cover problem

- A vertex-cover of an undirected graph G is a subset of its vertices such that it includes at least one end of each edge.
- The problem is to find minimum size of vertex-cover of the given graph.
- This problem is an NP-Complete problem.
The vertex-cover problem

- Finding the optimal solution is hard (it’s NP!) but finding a near-optimal solution is easy.
- There is an 2-approximation algorithm:
  - It returns a vertex-cover not more than twice of the size optimal solution.
The vertex-cover problem

APPROX-VERTEX-COVER(G)

1. $C \leftarrow \emptyset$
2. $E' \leftarrow E[G]$
3. while $E' \neq \emptyset$
   4. do let $(u, v)$ be an arbitrary edge of $E'$
   5. $C \leftarrow C \cup \{u, v\}$
   6. remove every edge in $E'$ incident on $u$ or $v$
4. return $C$
The vertex-cover problem

Near Optimal size=6
Optimal Size=3
The vertex-cover problem

- This is a polynomial-time 2-approximation algorithm. (Why?)
- Because:
  - \text{APPROX-VERTEX-COVER} is \(O(V+E)\)
  - \(|C^*| \geq |A|\)
  - \(|C| = 2|A|\)
  - \(|C| \leq 2|C^*|\)
Traveling salesman problem

- Given an undirected complete weighted graph $G$ we are to find a minimum cost Hamiltonian cycle.
- Satisfying triangle inequality or not this problem is NP-Complete.
- The problem is called *Euclidean TSP*. 
Traveling salesman problem

- Near Optimal solution
  - Faster
  - Easier to implement.
Euclidian Traveling Salesman Problem

APPROX-TSP-TOUR(G, W)
1 select a vertex \( r \in V[G] \) to be root.
2 compute a \textbf{MST} for \( G \) from root \( r \) using Prim Alg.
3 \( L= \)list of vertices in preorder walk of that \textbf{MST}.
4 \textbf{return} the Hamiltonian cycle \( H \) in the order \( L \).
Euclidian Traveling Salesman Problem

root

MST

Pre-Order walk

Hamiltonian Cycle
Traveling salesman problem

- This is polynomial-time 2-approximation algorithm. (Why?)
- Because:
  - APPROX-TSP-TOUR is $O(V^2)$
  - $C(MST) \leq C(H^*)$
  - $C(W) = 2C(MST)$
  - $C(W) \leq 2C(H^*)$
  - $C_H \leq C(W)$
  - $C(H) \leq 2C(H^*)$

\[ W: \text{Preorder walk} \]
\[ H^*: \text{optimal soln} \]
\[ W: \text{Preorder walk} \]
\[ H: \text{approx soln & triangle inequality} \]
Thank You......