• You can do hard things! Grades do matter, but not as much as you may think, but then life is uncertain anyway, so what.

• **Don't cheat.** The consequence for cheating is far greater than the reward. Just try your best and you'll be fine.

• **Please read the entire exam before writing anything.** There are 6 problems and most have multiple parts.

• This is a closed-book exam. At the end of the exam, you'll find a multi-page cheat sheet. *Do not tear out the cheat sheet!* No outside material is allowed on this exam.

• You should write your answers legibly and in the space given for the question. Overly verbose answers will be penalized.

• Scratch paper is available on the back of the exam. *Do not tear out the scratch paper!* It messes with the auto-scanner.

• **You have 75 minutes (1.25 hours) for the exam.** Manage your time well. *Do not spend too much time on questions you do not understand and focus on answering as much as you can!*

• Make sure you use the time well to think, be precise, and show as much work as possible.

Name: ________________________________

NetID: ________________________________

Date: ________________________________
Problem 1 [10 points]

For each of the following statements, answer if it is True or False. Use the table at the bottom to mark your choices.

i. Dijkstra’s algorithm works well on graphs with negative edge weights provided there is no negative length cycle.

ii. A problem can either be NP-Complete or NP-Hard but not both.

iii. If \( P = NP \) then every NP-Complete problem can be solved in polynomial time.

iv. Graph 2-Coloring can be decided in linear time.

v. The set of all programs is larger than the set of all languages.

vi. Every undecidable language is also unrecognizable.

vii. If language \( L \) is undecidable then either \( L \) or \( \overline{L} \) is unrecognizable.

viii. If using an Oracle for problem \( X \), one can obtain a decider for the \( \text{Halt}_{TM} \) then \( X \) is decidable.

ix. If a barber shaves everyone who doesn’t shave themselves then the barber shaves themselves.

x. If a graph is 3-colorable then it has 3 independent sets.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Your choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>i.</td>
<td>False</td>
</tr>
<tr>
<td>ii.</td>
<td>False</td>
</tr>
<tr>
<td>iii.</td>
<td>True</td>
</tr>
<tr>
<td>iv.</td>
<td>True</td>
</tr>
<tr>
<td>v.</td>
<td>False</td>
</tr>
<tr>
<td>vi.</td>
<td>False</td>
</tr>
<tr>
<td>vii.</td>
<td>True</td>
</tr>
<tr>
<td>viii.</td>
<td>False</td>
</tr>
<tr>
<td>ix.</td>
<td>False</td>
</tr>
<tr>
<td>x.</td>
<td>True</td>
</tr>
</tbody>
</table>
Problem 2 [10 points]

Given a directed graph $G = (V, E)$ with non-negative edge lengths $l(e), e \in E$ and a node $s \in V$, describe an algorithm to find the length of a shortest cycle containing the node $s$.

Solution: Refer to Lab 15, Problem 2.
Problem 3 [10 points]

Formally prove or disprove the following statement. *There is no program that always stops and solves the halting problem.*

**Solution:** **Answer:** The statement is True. Refer to Lecture 22 for the proof of halting is undecidable. Alternatively, we can prove halt is undecidable via the fact that self-halt is undecidable. We prove by 2 steps:

- First, we show that self-halt is undecidable: Assuming self-halt is decidable, there exists a TM, named “SH”, s.t. Accept(SH) = self-halt, and Diverge(SH) = ∅. Then if we create another TM, named “SH” and every transition to accept state in SH to be routed to a new hang state in SH. And similarly for every transition to reject state in SH to be accept in SH. Therefore Accept(SH) = Σ∗\self-halt and Reject(SH) = ∅. This means SH accepts <SH> if and only if SH does not halt on <SH>, which is a contradiction. So self-halt must be undecidable.

- Given self-halt is undecidable, we prove halt is undecidable. Now suppose halt is decidable, then there is a TM, H, that decides halt. Then we can create another machine, named SH, that decides H. Clearly, SH decides self-halt. Then given strings, we build the following transition/reduction function (assuming <M> ∈ self-halt):

```python
def SH(<M>):
    # reduction
    def M_(x):
        if x == <M>: return True
        return False
    # proxy
    if H(M_, <M_>):
        return True
    else:
        return False
```

As a result, we prove halt is undecidable (since self-halt is undecidable). Hence the statement is True (by the definition of undecidability).

**Rubrics:** 3 points for the proof of halt is undecidable, 3 points for the correct answer (True/False), 4 points for the reasoning that why undecidability leads to the statement: *There is no program that always stops and solves the halting problem.*
Problem 4 [20 points]

The 4-Set-Packing problem is defined as follows.

- Inputs: A collection of $m$ sets $S = \{S_1, S_2, \ldots, S_m\}$ such that $|S_i| = 4$ for all $i \in \{1, \ldots, m\}$ and an integer $k$.
- Output: True if there exists a disjoint subcollection $L \subseteq S$ of size $k$. False otherwise.

Note: Disjoint subcollection means no individual element belongs to two different sets in it.

The 3-Dimensional-Matching problem is defined as follows.

- Inputs: Three disjoint sets $X$, $Y$, and $Z$ of $n$ elements each, and a set of triplets $T \subseteq X \times Y \times Z$.
- Output: True if there exist disjoint triplets from $T$ whose union is $X \cup Y \cup Z$. False otherwise.

Given 3-Dimensional Matching is NP-Complete, show that 4-Set-Packing is NP-Complete.

Solution: 4-Set-Packing is in NP, as given any instance $L$, we can check in polynomial time: if the size of $L$ is $k$; and that no element appears in more than one set in $L$.

To prove 4-Set-Packing is NP-Hard, we reduce from 3-Dimensional-Matching, which is known to be NP-Complete, in two steps:

**Step 1: Construct the polynomial time reduction**

Given an instance of 3-Dimensional-Matching, with disjoint sets $X$, $Y$, and $Z$, and a set of triplets $T \subseteq X \times Y \times Z$, construct a 4-Set-Packing instance by creating a set $S_i = \{x, y, z, w_i\}$ for each triplet $(x, y, z) \in T$, where $w_i$ is a new element introduced to ensure each set has exactly four elements. The new elements $w_i$ are unique to each set $S_i$.

**Step 2: Prove the correctness of the reduction**

Forward Direction: If there is a solution to the 3-Dimensional-Matching instance, then we have a subset of triplets $M \subseteq T$ where $|M| = n$ and no two triplets in $M$ share any element. We can map $M$ to a collection $L$ of $k = n$ sets in the 4-Set-Packing instance, where each set in $L$ corresponds to a triplet in $M$ plus the unique element $w_i$. This collection $L$ will be a valid solution to the 4-Set-Packing problem.

Reverse Direction: Conversely, if there is a solution to the 4-Set-Packing instance under our reduction, then we have a collection of sets $L$ where no element is repeated across the sets. By removing the unique elements $w_i$ from each set in $L$, we obtain a collection of triplets that form a valid solution to the original 3-Dimensional-Matching instance.

Therefore, 4-Set-Packing is NP-Hard, and since it is also in NP, it is NP-Complete.

Note: A reduction that doesn’t allow different values of $n$ will be completely incorrect, and just stating for NP it is verifiable in polynomial time is worth no point.
Problem 5 [14 points]

a. A quasi-satisfying assignment (quasiSAT) for a 3CNF boolean formula $\phi$ is an assignment of truth values to the variables such that at most one clause in $\phi$ does not contain a True literal. Prove that it is NP-Complete to determine whether a given 3CNF boolean formula has a quasi-satisfying assignment or not.

**Solution:** quasiSAT is trivially NP, since given a quasi-satisfying assignment for an instance, the quasi-satisfiability of the instance can be checked in polynomial time by applying the assignment and evaluating the formula.

To prove NP-hardness of quasiSAT, let $\phi$ be an arbitrary 3CNF formula, and let $\phi'$ be a 3CNF formula constructed by introducing 3 new variables $x, y, z$ and combining $\phi$ with every 8 clauses that contains the literals of $x, y, z$ as the following:

$$\phi' = \phi \land (x \lor y \lor z) \land (x \lor y \lor \overline{z}) \land (x \lor \overline{y} \lor z) \land \ldots \land (\overline{x} \lor \overline{y} \lor \overline{z})$$

Then, $\phi$ is satisfiable if and only if $\phi'$ is quasi-satisfiable.

$\rightarrow$ Suppose $\phi$ is satisfiable. Then, there exists a satisfying assignment $A$ for $\phi$. Let $A'$ be an assignment constructed by combining $A$ with the assignment $x = T, y = T, z = T$. If we apply the assignment $A'$ on $\phi'$, all clauses except for $(\overline{x} \lor \overline{y} \lor \overline{z})$ would contain a True literal. Therefore, $\phi'$ is quasi-satisfiable if $\phi$ is satisfiable.

$\leftarrow$ Suppose $\phi'$ is quasi-satisfiable. Note that regardless of the assignment on the variables $x, y, z$, exactly one of the last 8 clauses would not contain a True literal. This implies that there exists an assignment $A'$ for $\phi'$ such that all clauses before the last 8 clauses contain a True literal. We can construct a satisfying assignment $A$ for $\phi$ by getting rid of the assignments for $x, y, z$ from $A'$. Therefore, $\phi$ is satisfiable if $\phi'$ is quasi-satisfiable.

Since quasiSAT is NP, and also there exists a polynomial time reduction from 3SAT to quasiSAT, we conclude that quasiSAT is NP-Complete. ■
b. Show that the Hamiltonian Cycle problem for undirected graphs is \textit{NP-Complete}. Note: You may use that Hamiltonian Cycle problem for directed graphs is \textit{NP-Complete}.

\textbf{Solution:} Hamiltonian Cycle for undirected graph is trivially \textit{NP}, since given a Hamiltonian cycle in an instance, we can simply iterate over the cycle and check if it indeed is a Hamiltonian cycle.

To prove \textit{NP-hardness}, let \( G = (V, E) \) be an arbitrary directed graph, and let \( G' = (V', E') \) be an undirected graph defined as the following:

\[
V' = \{ v_{in}, v_{mid}, v_{out} \mid v \in V \}
\]
\[
E' = \{ (u_{out}, v_{in}) \mid (u, v) \in E \} \cup \{ (v_{in}, v_{mid}), (v_{mid}, v_{out}) \mid v \in V \}
\]

Then, \( V \) has a directed Hamiltonian cycle if and only if \( V' \) has an undirected Hamiltonian cycle.

\[\rightarrow\] Suppose there exists a directed Hamiltonian cycle \( C = (c_1, c_2, ..., c_n) \) in \( G \). By construction, \( C' = (c_{1in}, c_{1mid}, c_{1out}, c_{2in}, ..., c_{nmid}, c_{nout}) \) is an undirected Hamiltonian cycle in \( G' \).

\[\leftarrow\] Suppose there exists an undirected Hamiltonian cycle \( C' \) in \( G' \). By construction, \( C' \) can be written in the following form:

\[
C' = (c_{1in}, c_{1mid}, c_{1out}, c_{2in}, ..., c_{nmid}, c_{nout})
\]

Since \( u_{out} \) is connected to \( v_{in} \) if and only if \((u, v) \in E \), \( C = (c_1, c_2, ..., c_n) \) forms a directed Hamiltonian cycle of \( G \).

We showed that Hamiltonian Cycle for undirected graphs is both \textit{NP} and \textit{NP-hard}. Therefore we conclude that Hamiltonian Cycle for undirected graphs is \textit{NP-complete}. \qed
Problem 6 [10 points]

Identify the errors in the following proofs.

a. Define the following problems.

- **DFA-Accepts**
  
  Inputs: A DFA $D$ and a string $w$. Output: True if $w \in L(D)$. False otherwise.

- **NFA-Accepts**
  
  Inputs: A NFA $N$ and a string $w$. Output: True if $w \in L(N)$. False otherwise.

Note the following.

- **DFA-Accepts** is in $P$ as there is a single execution path for $w$ on $D$.
- It’s highly unlikely that **NFA-Accepts** is in $P$. Intuitively, there are exponentially many ways to simulate $w$ on $N$ that makes **NFA-Accepts** $NP$-Hard.

Construct a solver for **NFA-Accepts** as follows.

Step 1. Convert the given NFA into an equivalent DFA.

Step 2. Now use the poly-time solver for **DFA-Accepts** to solve **NFA-Accepts**.

This implies **NFA-Accepts** which is $NP$-Hard has a poly-time solver implying $P = NP$. [Did we just solve the millennium problem!?!]

**Solution:** While a polynomial time reduction from a known $NP$-Hard problem to a problem in $P$ would imply that $P=NP$, the incremental subset construction algorithm for converting NFAs to DFAs is exponential in the number of states of the NFA.

■
b. Refer to the cheat sheet for the definition of the Independent Set decision problem. Consider the following decider for this problem.

```plaintext
DecideIndependentSet(G = (V, E), k):
For each S ⊆ V such that |S| = k:
  bool ← True
  For every pair of two vertices (u, v) from the set S:
    If there is an edge between u and v:
      bool ← False
  If bool == True:
    return True
  Else:
    return False
```

The runtime of the above algorithm is \( T(n) = O\left((n^k)k^2\right) \). This implies Independent Set which is NP-Hard has a poly-time solver implying \( P = NP \). [Did we just solve the millennium problem again!?

**Solution:** While a polynomial time solution to any NP-Hard problem would imply that \( P = NP \), the provided algorithm is not polynomial time in terms of all of the inputs. A polynomial of \( n \) and \( k \) takes the form \((n + k)^a\), which would never contain an \( n^k \) term.
Problem 7 [6 points]

Prove or disprove that the Halting problem is NP-Hard.

Solution: Refer to Lecture 23 – Pre-Lecture Brain Teaser. The solution is in the scribbles.

Reduce SAT to Halting:
For an arbitrary SAT solver, modify it as follows:
If the input instance is satisfiable, return accept, otherwise let turning machine M not halt on input w.
Then the turning machine halts if and only if the SAT instance is satisfiable and the reduction takes polynomial time.
So Halting is NP-Hard.

Note that Halting is not NP-complete as verifying the input in polynomial time is impossible.

You cannot prove NP-hard from undecidable. Some problems are undecidable but not NP-hard (out of the scope of this course).
Problem 8 [20 points]

For definitions of $A_{TM}$, $Halt_{TM}$, $HaltB_{TM}$ refer to the cheat sheet.

a. Using undecidability of $A_{TM}$, show that $HaltB_{TM}$ is undecidable.

**Solution:** (Refer to Lecture 24. The solution is in the scribbles.) Remember that 

$$HaltB_{TM} = \{(M)|M \text{ is a TM and } M \text{ halts on blank input.}\}$$

Suppose there is an algorithm $\text{DecideHaltOnBlank}$ that correctly decides the language $\text{HaltOnBlank}$. Then, we can solve the $\text{AcceptOnInput}$ problem as follows:

```plaintext
\text{DecideAcceptOnInput}\((M, w)\):
\text{Encode the following Turing machine } M':
M'(x):
if \(x == \epsilon\):
if \(M(w)\):
return True
else:
 LOOP forever
else:
 LOOP forever
if \text{DecideHaltOnBlank}(\langle M' \rangle)
return True
else
return False
```

Alternatively, $M'$ can also be constructed as

```plaintext
M'(x):
if \(M(w)\)
return True
LOOP forever
```

We prove this reduction correct as follows:

$\implies$ Suppose $M$ accepts input $w$.
Then $M'$ accepts the input string $\epsilon$.
So $\text{DecideHaltOnBlank}$ accepts the encoding $\langle M' \rangle$.
So $\text{DecideAcceptOnInput}$ correctly accepts the encoding $\langle M, w \rangle$.

$\impliedby$ Suppose $M$ does not halt on input $w$.
Then $M'$ does not halt on any input string $x$.
So $\text{DecideHaltOnBlank}$ rejects the encoding $\langle M' \rangle$.
So $\text{DecideAcceptOnInput}$ correctly rejects the encoding $\langle M, w \rangle$.

In both cases, $\text{DecideAcceptOnInput}$ is correct. But that’s impossible because $\text{Halt}$ is undecidable. We conclude that the algorithm $\text{DecideHaltOnBlank}$ does not exist. ■
b. Using undecidability of \( \text{Halt}_{\text{TM}} \), show that the following language is undecidable.

\[ \text{Reg}_{\text{TM}} = \{ \langle M \rangle | M \text{ is a TM and } L(M) \text{ is regular.} \} \]

**Solution:** (Refer to Lab 22.)

Suppose there is an algorithm DECIDE\text{ACCEPT}\text{REGULAR} that correctly decides the language ACCEPT\text{REGULAR}. Then, we can solve the DECIDE\text{HALT}\text{ON}\text{INPUT} problem as follows:

\[
\begin{align*}
\text{DECIDE}\text{HALT}\text{ON}\text{INPUT}(\langle M, w \rangle): \ & \\
& \text{Encode the following Turing machine } M': \\
& \quad M'(x): \\
& \quad \text{if } x \text{ is of the form } 0^n1^n \text{ return True} \\
& \quad \text{else run } M \text{ on input } w \text{ return True} \\
& \quad \text{if } \text{DECIDE}\text{ACCEPT}\text{REGULAR}(\langle M' \rangle) \text{ return True} \\
& \quad \text{else return False}
\end{align*}
\]

We prove this reduction correct as follows:

\[\Rightarrow \] Suppose \( M \) halts on input \( w \).
Then \( M' \) accepts \textit{every} the input string, i.e., \( \Sigma^* \), which is regular.
So DECIDE\text{ACCEPT}\text{REGULAR} accepts the encoding \( \langle M' \rangle \).
So DECIDE\text{HALT}\text{ON}\text{INPUT} correctly accepts the encoding \( \langle M, w \rangle \).

\[\Leftarrow \] Suppose \( M \) does not halt on input \( w \).
Then \( M' \) does not halt on \textit{any} input string except \( 0^n1^n \), which is not regular.
So DECIDE\text{ACCEPT}\text{REGULAR} rejects the encoding \( \langle M' \rangle \).
So DECIDE\text{HALT}\text{ON}\text{INPUT} correctly rejects the encoding \( \langle M, w \rangle \).

In both cases, DECIDE\text{HALT}\text{ON}\text{INPUT} is correct. But that’s impossible because \text{HALT}\text{ON}\text{INPUT} is undecidable. We conclude that the algorithm DECIDE\text{ACCEPT}\text{REGULAR} does not exist. Therefore \text{REG}_{\text{TM}} \text{ must be undecidable.} \quad \blacksquare
This page is for additional scratch work!
1 Recursion

**Simple recursion**

- **Reduction:** solve one problem using the solution to another.
- **Recursion:** a special case of reduction - reduce problem to a smaller instance of itself (self-reduction).

**Definitions**

- Problem instance of size \( n \) is reduced to one or more instances of size \( n - 1 \) or less.
- For termination, problem instances of small size are solved by some other method as base cases

Arguably the most famous example of recursion. The goal is to move \( n \) disks one at a time from the first peg to the last peg.

```
Hanoi (n, src, dest, tmp):
  if (n > 0) then
    Move disk n from src to dest
    Hanoi (n-1, tmp, dest, src)
```

**Recurrences**

Suppose you have a recurrence of the form \( T(n) = rT(n/c) + f(n) \).

The master theorem gives a good asymptotic estimate of the recurrence. If the work at each level is:

- Decreasing: \( r f(n/c) = \kappa f(n) \) where \( \kappa < 1 \) \( T(n) = O(f(n)) \)
- Equal: \( r f(n/c) = f(n) \) \( T(n) = O(f(n) \log n) \)
- Increasing: \( r f(n/c) = O(f(n)) \) \( T(n) = O(n \log^r n) \)

Some useful identities:

- Sum of integers: \( \sum_{k=1}^{n} k = \frac{n(n+1)}{2} \)
- Geometric series closed-form formula: \( \sum_{k=0}^{n} a^k = \frac{1 - a^{n+1}}{1 - a} \) \( a \neq 1 \)
- Logarithmic identities: \( \log(ab) = \log a + \log b \) \( \log(a/b) = \log a - \log b \) \( a^{\log_b c} = b^{\log_b a} \quad (a, b, c > 1) \)

**Backtracking**

Backtracking is the algorithm paradigm involving guessing the solution to a single step in some multi-step process and recursing backwards if it doesn't lead to a solution. For instance, consider the longest increasing subsequence (LIS) problem. You can either check all possible subsequences:

```
algLISNaive(A[1..n]):
  maxmax = 0
  for each subsequence B of A do
    if B is increasing and |B| > maxmax then
      maxmax = |B|
  return maxmax
```

On the other hand, we don't need to generate every subsequence; we only need to generate the subsequences that are increasing:

```
LIS_smaller(A[1..n], x):
  if n = 0 then return 0
  max = LIS_smaller(A[1..n-1], x)
  if A[n] < x then
    max = max {max, 1 + LIS_smaller(A[1..(n-1)], A[n])}
  return max
```

**Divide and conquer**

Divide and conquer is an algorithm paradigm involving the decomposition of a problem into the same subproblem, solving them separately and combining their results to get a solution for the original problem.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Runtime</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mergesort</td>
<td>( O(n \log n) )</td>
<td>( O(n \log n) )</td>
</tr>
<tr>
<td>Quicksort</td>
<td>( O(n^2) )</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

We can divide and conquer multiplication like so:

\[
bc = 10^n b_Lc_L + 10^{n/2} (b_lc_R + b_rc_L) + b_rc_R.
\]

We can rewrite the equation as:

\[
b_c = b(x)x(c(x)) - (b_Lc_L + b_lc_R - b_lc_R) x + b_rc_R.
\]

Its running time is \( O(n^{\log_2 3}) = O(n^{1.585}) \).

**Linear time selection**

The median of medians (MoM) algorithms give a element that is larger than \( \frac{1}{2} \)'s and smaller than \( \frac{1}{2} \)'s of the array elements. This is used in the linear time selection algorithm to find element of rank \( k \).

```
Median-of-medians(A, k):
  sublists = [A[p:p+5] for p in range(0, len(A), 5)]
  medians = [sorted(sublist[:len(sublist)//2]) for sublist in sublists]
  if len(sublist) <= 5
    return sorted(sublist)[len(sublist)//2]
  else
    pivot = Median-of-medians(medians, len(medians)//2)
    if pivot > pivot
      return Median-of-medians(low, i)
    else if pivot < pivot
      return Median-of-medians(low, i+1)
    else
      return pivot
```

**Linear time selection**

The median of medians (MoM) algorithms give a element that is larger than \( \frac{1}{2} \)'s and smaller than \( \frac{1}{2} \)'s of the array elements. This is used in the linear time selection algorithm to find element of rank \( k \).

```
Median-of-medians(A, k):
  sublists = [A[p:p+5] for p in range(0, len(A), 5)]
  medians = [sorted(sublist[:len(sublist)//2]) for sublist in sublists]
  if len(sublist) <= 5
    return sorted(sublist)[len(sublist)//2]
  else
    pivot = Median-of-medians(medians, len(medians)//2)
    if pivot > pivot
      return Median-of-medians(low, i)
    else if pivot < pivot
      return Median-of-medians(low, i+1)
    else
      return pivot
```

**Backtracking**

Backtracking is the algorithm paradigm involving guessing the solution to a single step in some multi-step process and recursing backwards if it doesn't lead to a solution. For instance, consider the longest increasing subsequence (LIS) problem. You can either check all possible subsequences:

```
algLISNaive(A[1..n]):
  maxmax = 0
  for each subsequence B of A do
    if B is increasing and |B| > maxmax then
      maxmax = |B|
  return maxmax
```

On the other hand, we don't need to generate every subsequence; we only need to generate the subsequences that are increasing:

```
LIS_smaller(A[1..n], x):
  if n = 0 then return 0
  max = LIS_smaller(A[1..n-1], x)
  if A[n] < x then
    max = max {max, 1 + LIS_smaller(A[1..(n-1)], A[n])}
  return max
```
### Dynamic programming

Dynamic programming (DP) is the algorithm paradigm involving the computation of a recursive backtracking algorithm iteratively to avoid the recomputation of any particular subproblem.

#### Longest increasing subsequence

The longest increasing subsequence problem asks for the length of a longest increasing subsequence in an unordered sequence, where the sequence is assumed to be given as an array. The recurrence can be written as:

\[
LIS(i,j) = \begin{cases} 
0 & \text{if } i = 0 \\
LIS(i-1,j) & \text{if } A[i] \geq A[j] \\
\max \{ LIS(i-1,j), 1+LIS(i-1,i) \} & \text{else}
\end{cases}
\]

#### Edit distance

The edit distance problem asks how many edits we need to make to a sequence for it to become another one. The recurrence is given as:

\[
Opt(i,j) = \begin{cases} 
\alpha_{x_i y_j} + Opt(i-1,j-1), & \text{if } A[i] \neq B[j] \\
\delta + Opt(i-1,j), & \text{if } i > 0 \text{ and } A[i] = B[j] \\
\delta + Opt(i,j-1), & \text{if } j > 0 \text{ and } A[i] = B[j]
\end{cases}
\]

**Base cases:**

- \(Opt(0,j) = \delta \cdot j\)
- \(Opt(i,0) = \delta \cdot i\)

#### Graph algorithms

### Graph basics

A graph is defined by a tuple \( G = (V,E) \) and we typically define \( n = |V| \) and \( m = |E| \). We define \((u,v)\) as the edge from \( u \) to \( v \). Graphs can be represented as adjacency lists, or adjacency matrices though the former is more commonly used.

- **path:** sequence of distinct vertices \( v_1, v_2, \ldots, v_k \) such that \( (v_i, v_{i+1}) \in E \) for \( 1 \leq i < k-1 \). The length of the path is \( k-1 \) (the number of edges in the path).
  
  **Note:** a single vertex \( u \) is a path of length 0.

- **cycle:** sequence of distinct vertices \( v_1, v_2, \ldots, v_k \) such that \( (v_i, v_{i+1}) \in E \) for \( 1 \leq i \leq k-1 \) and \( (v_k, v_1) \in E \). A single vertex is not a cycle according to this definition.

  **Caveat:** Sometimes people use the term cycle to also allow vertices to be repeated; we will use the term tour.

- A vertex \( u \) is connected to \( v \) if there is a path from \( u \) to \( v \).
- The **connected component** of \( u \), \( \text{con}(u) \), is the set of all vertices connected to \( u \).
- A vertex \( u \) can reach \( v \) if there is a path from \( u \) to \( v \). Alternatively \( v \) can be reached from \( u \). Let \( \text{rch}(u) \) be the set of all vertices reachable from \( u \).
Directed acyclic graphs

Directed acyclic graphs (dags) have an intrinsic ordering of the vertices that enables dynamic programming algorithms to be used on them. A topological ordering of a dag \( G = (V, E) \) is an ordering \( \prec \) on \( V \) such that if \( (u, v) \in E \) then \( u \prec v \).

### Kahn’s algorithm

**Pseudocode:**

```plaintext
Kahn(G(V, E), u):
start = empty list
do for each \( v \) in V:
    if \( \text{in}(v) = 0 \):
        add \( v \) to start
for each \( v \) in start:
    for each \( w \) in Adj(v):
        if \( \text{in}(w) - 1 \):
            add \( w \) to start
return start
```

**Running time:** \( O(n + m) \)

- A dag may have multiple topological sorts.
- A topological sort can be computed by DFS, in particular by listing the vertices in decreasing post-visit order.

### DFS and BFS

**Pseudocode:**

**Explore(G, u):**

- for \( i = 1 \) to \( n \):
  - \( \text{Visited}[i] = \text{False} \)
  - Add \( i \) to \( \text{ToExplore} \) and to \( S \)
- for each \( v \) in Adj(x) do:
  - if \( \text{Visited}[v] = \text{False} \):
    - Make tree \( T \) with root as \( u \)
    - \( \text{Visited}[v] = \text{True} \)
    - Add \( v \) to \( \text{ToExplore} \) and to \( T \)(with \( x \) as parent)
- while \( \text{ToExplore} \) is not empty do:
  - Remove node \( x \) from \( \text{B} \)
  - for each \( (x, y) \) in Adj(x) do:
    - if \( \text{Visited}[y] = \text{False} \):
      - \( \text{Visited}[y] = \text{True} \)
      - Add \( y \) to \( \text{B} \)

**Pre and post numbering aids in analyzing the graph structure.** By looking at the numbering we can tell if an edge \( (u, v) \) is:

- Forward edge: \( \text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u) \)
- Backward edge: \( \text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v) \)
- Cross edge: \( \text{pre}(u) < \text{post}(u) < \text{pre}(v) < \text{post}(v) \)

### Strongly connected components

- Given \( G \), \( u \) is strongly connected to \( v \) if \( u \in \text{rch}(v) \) and \( v \in \text{rch}(u) \).
- A maximal group of \( G \), vertices that are all strongly connected to one another is called a strong component.

**Pseudocode:**

**Metagraph(G(V, E)):**

Compute \( \text{rev}(G) \) by brute force ordering \( \prec \) reverse postordering of \( V \) in \( \text{rev}(G) \) by \( \text{DFS}(\text{rev}(G), x) \) for any vertex \( x \)
Mark all nodes as unvisited
for each \( u \) in ordering do:
  if \( u \) is not visited and \( u \in V \) then:
    \( S_u = \) nodes reachable by \( u \) by \( \text{DFS}(G, u) \)
  Output \( S_u \) as a strong connected component
  \( G(V, E) \leftarrow G - S_u \)

**Running time:** \( \Theta(n^3) \)

### Shortest paths

#### Dijkstra’s algorithm:
Find minimum distance from vertex \( s \) to all other vertices in graphs without negative weights.

**Pseudocode:**

```plaintext
Dijkstra(G(V, E), s):
\( d(s) \leftarrow 0 \)
\( X := \{ s \} \)
for each \( v \in V \) do:
    if \( d(v) = \infty \):
        \( d(v) \leftarrow d(s) + \ell(s, v) \)
for each \( u \) in Adj(v) do:
    if \( d(u) > d(v) + \ell(u, v) \):
        \( d(u) \leftarrow d(v) + \ell(u, v) \)
        \( X := X \cup \{ u \} \)
return \( d \)
```

**Running time:** \( O(m + n \log n) \) (if using a Fibonacci heap as the priority queue)

#### Bellman-Ford algorithm:
Find minimum distance from vertex \( s \) to all other vertices in graphs without negative cycles. It is a DP algorithm with the following recurrence:

\[
\begin{align*}
d(v, k) &= \begin{cases}
0 & \text{if } v = s \text{ and } k = 0 \\
\infty & \text{if } v \neq s \text{ and } k = 0 \\
\min_{u \in V} \{d(u, k - 1) + \ell(u, v)\} & \text{else}
\end{cases}
\end{align*}
\]

**Base cases:** \( d(s, 0) = 0 \) and \( d(v, 0) = \infty \) for all \( v \neq s \).

**Pseudocode:**

```plaintext
Bellman-Ford(G(V, E), s):
\( d(s) \leftarrow 0 \)
for each \( v \) in V do:
    \( d(v) \leftarrow \infty \)
for \( k = 1 \) to \( n - 1 \) do:
    for each \( v \) in V do:
        \( d(v) \leftarrow \min\{d(v), d(u) + \ell(u, v)\} \)
return \( d \)
```

**Running time:** \( O(nm) \)

#### Floyd-Warshall algorithm:
Find minimum distance from every vertex to every vertex in a graph without negative cycles. It is a DP algorithm with the following recurrence:

\[
\begin{align*}
d(i, j, k) &= \begin{cases}
0 & \text{if } i = j \\
\infty & \text{if } (i, j) \notin E \text{ and } k = 0 \\
\min\{d(i, j, k - 1) + d(k, j, k - 1)\} & \text{else}
\end{cases}
\end{align*}
\]

Then \( d(i, j, n - 1) \) will give the shortest-path distance from \( i \) to \( j \).

**Pseudocode:**

```plaintext
Floyd-Warshall(G(V, E), s):
for each \( i \) in V do:
    \( d(i, i) \leftarrow 0 \)
    \( d(i, j) \leftarrow \ell(i, j) \) (** \( \ell(i, j) \leftarrow \infty \) if \( (i, j) \notin E \), 0 if \( i = j \)**)
for \( k = 0 \) to \( n - 1 \) do:
    for each \( i \) in V do:
        for each \( j \) in V do:
            \( d(i, j) \leftarrow \min\{d(i, j), d(i, k) + d(k, j)\} \)
return \( d \)
```

**Running time:** \( O(n^3) \)
**Turing Machines**

A Turing machine is the simplest model of computation. It consists of:
- A tape, which is infinite and can support both writing and reading.
- A read/write head that can move along the tape.
- A state register, which holds the current state of the machine.
- A transition function, δ, that maps the current state and symbol under the head to the next state, symbol to write, and movement of the head.

Transition Function: \( \delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{ \leftarrow, \rightarrow, \square \} \)

- \( q \): current state.
- \( c \): character under head.
- \( p \): new state.
- \( \leftarrow \): move left.
- \( \rightarrow \): move right.
- \( \square \): don't move.
- \( c/d \): character to write under head.
- \( L \): move left.
- \( R \): move right.
- \( \bot \): stop.

**Complexity Classes**

- **NP**
- **P**
- **EXP**
- **PSPACE**
- **NPC**
- **Undecidable**

**Sample NP-complete problems**

- **CIRCUITSAT**: Given a boolean circuit, is there a satisfying assignment?
- **3SAT**: Given a boolean formula in conjunctive normal form, with exactly three distinct literals per clause, does the formula have a satisfying assignment?
- **INDEPENDENTSET**: Given an undirected graph \( G \) and integer \( k \), is there a subset of vertices of size \( k \) that have no edges among them?
- **CLIQUE**: Given an undirected graph \( G \) and integer \( k \), is there a complete subgraph of \( G \) with exactly \( k \) vertices?
- **kPARTITION**: Given a set \( X \) of \( kn \) positive integers and an integer \( k \), can \( X \) be partitioned into \( n \) \( k \)-element subsets, all with the same sum?
- **3COLOR**: Given an undirected graph \( G \), can its vertices be colored with three colors, and that every edge touches vertices with two different colors?
- **HAMILTONIANPATH**: Given a graph \( G \) either directed or undirected, is there a path in \( G \) that visits every vertex exactly once?
- **HAMILTONIANCYCLE**: Given a graph \( G \) (either directed or undirected), is there a cycle in \( G \) that visits every vertex exactly once?
- **LONGESTPATH**: Given a graph \( G \) (either directed or undirected, possibly with weighted edges) and an integer \( k \), does \( G \) have a path of length \( k \)?

**Reductions**

- A general methodology to prove impossibility results.
  - Start with some known hard problem \( X \).
  - Reduce \( X \) to your favorite problem \( Y \).

If \( Y \) can be solved then so can \( X \Rightarrow Y \). But we know \( X \) is hard so \( Y \) has to be hard too. On the other hand if we know \( Y \) is easy, then \( X \) has to be easy too.

**Sample undecidable problems**

- **ACCEPTONINPUT**: \( \{ \langle M, w \rangle \mid M \text{ is a TM that accepts } w \} \)
- **HALTONINPUT**: \( \{ \langle M, w \rangle \mid M \text{ is a TM that halts on input } w \} \)
- **HALTONBLANK**: \( \{ \langle M \rangle \mid M \text{ is a TM and } M \text{ halts on blank input} \} \)
- **EMPTINESS**: \( \{ \langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset \} \)
- **EQUALITY**: \( \{ \langle M_A, M_B \rangle \mid M_A \text{ and } M_B \text{ are TMs and } L(M_A) = L(M_B) \} \)