1 Short Questions

(a) For recurrences below give a tight asymptotic bound. You only need to give the bound. Please highlight the answer in a clear box or circle.

i. $T(n) = T(n/2) + T(n/3) + T(n/10) + 5n$ for $n > 10$ and $T(n) = 1$ for $1 \le n \le 10$.

Solution: Building out the recursion tree we find the work at level k is $5 \cdot (\frac{14}{15})^k n$. This results in a decreasing geometric series, which converges to $\boxed{\Theta(n)}$. ■

ii. $T(n) = T(n-1) + n^4$ for $n > 1$ and $T(1) = 1$.

Solution: Unrolling the recurrence we find that $T(n) = \sum_{i=1}^{n} i^4$. Obviously $T(n) \le \sum_{i=1}^{n} n^4 = n^5$. One can also see that $T(n) = \Omega(n^5)$: one way is to observe that $T(n)$ can be interpreted as the right-Riemann sum of the integral $\int_0^n x^4 \, dx$ over an interval where x^4 is increasing, i.e., $T(n) \ge \int_0^n x^4 \, dx = n^5/5 = \Omega(n^5)$; another way is to observe that $T(n) \ge \sum_{i=n/2+1}^{n} i^4 \ge \sum_{i=n/2+1}^{n} (n/2)^4 = (n/2)^5 = \Omega(n^5)$. In conclusion, $T(n) = \boxed{\Theta(n^5)}$. ■

(b) For each of the problems below choose the best running time/complexity from the list below, based on what you know of from the course so far. Please highlight the answer in a clear box or circle.

i. **P** Given an unweighted directed graph G check whether G has a cycle of length exactly 100.

Solution (Sketch): One can enumerate all sequences of exactly 100 vertices and check if said sequence corresponds to a directed cycle in $O(n^{100})$ ***time***. ■

ii. **L–P** Reduce an instance of L_u to an instance of L_{HALT}.

Solution (Sketch): Given $\langle M, w \rangle$, one possible reduction consists of encoding the following machine M':

```
M'():
  run M on input w
  if M accepted
      return TRUE
  else
      loop forever
```

The time it takes down to write down the encoding M' is dominated by time it takes to write down the encodings of M and w, i.e., ***linear time***. However since we never analyzed the runtime of these reductions in class we will take any answer between **L** and **P**. Because the reduction itself is not a *decision* problem the answer cannot be NP, D or U. ■

iii. **L** Find a longest path in a given DAG.

> **Solution (Sketch):** In class we saw a Dynamic Programming based algorithm that runs in $O(V+E)$ *time*. ■

iv. **L** Given graph $G = (V, E)$ and two disjoint node sets $S, T \subseteq V$ check if there is a path from some node in S to some node in T.

> **Solution (Sketch):** Add two vertices s^*, t^* with edges $s^* \to u$ for each $u \in S$ and $v \to t^*$ for each $v \in T$. Then there is a path from some node in S to some node in T if and only if there is a path from s^* to t^*, which we can determine by running Basic Search from s^* in $O(V+E)$ *time*. ■

v. **NP** Given G check if it is colorable with 100000000000000 colors.

> **Solution (Sketch):** One can reduce 3Color to this problem, so it is NP-hard. On the other hand it is in ***NP***, since one can generate a certificate in the form of a 100000000000000-coloring of the vertices and check in linear time if the coloring is proper. ■

vi. **Q** Compute the edit distance of two strings of the same length.

> **Solution (Sketch):** In class we saw a Dynamic Programming based algorithm that runs in time $O(n^2)$ where n is the common length of the strings. ■

Choose from the following choices:

- L There is an algorithm that runs in linear time
- Q There is an algorithm that runs in quadratic time
- C There is an algorithm that runs in cubic time
- P There is a polynomial-time algorithm
- NP The problem is in NP
- D The problem is decidable
- U The problem is undecidable

Rubric: 10 points. 1.25 points for each subpart, all or nothing (be careful about part (b)ii).

2 NP Hardness

Given a directed graph $G = (V, E)$, two distinct vertices s, t, and an integer k the LONGPATH problem is to decide whether G has an s-t-path of length at least k. Prove that LONGPATH is NP-Hard.

Solution: We will reduce from the Directed Hamiltonian Path problem.

Given a directed graph G, we will create a graph H from G by adding two vertices s and t, with edges $s \to u$ and $u \to t$ for each $u \in V$.

We claim that G has a Hamiltonian Path if and only if H has an s-t path of length at least n + 1, where n is the number of vertices in G.

$\Longrightarrow$ Suppose G has a Hamiltonian Path, starting at some vertex x and ending at some other vertex y. This path has length $n-1$, since it must visit every vertex in G exactly once. Then the path $s \to x \rightsquigarrow y \to t$ is a path in H of length $n+1$.

$\Longleftarrow$ Suppose H has a path of length at least $n+1$. Since H has exactly $n+2$ vertices, H cannot have a path of length greater than $n+1$, so the path has length exactly $n+1$. Since s has no incoming edges and t has no outgoing edges, the path must be of the form $s \to x \rightsquigarrow y \to t$, where the subpath $x \rightsquigarrow y$ lies entirely within G and has length $n-1$. This can only happen if this subpath is in fact a Hamiltonian Path of G.

H can clearly be constructed in time polynomial (in fact linear) in the size of G, so this is a polynomial-time reduction. ■

Rubric: 10 points. Standard polynomial-time reduction rubric.

3 Undecidability

Let $L_{AE374} = \{\langle M \rangle \mid M \text{ is a TM that accepts exactly 374 strings}\}$. Prove that L_{AE374} is undecidable.

Solution: For the sake of contradiction, suppose there is an algorithm DECIDEAE374 that correctly decides the language L_{AE374}. Then we can solve HALT:

```
DECIDEHALT(⟨M⟩):
  Encode the following Turing machine M′:
      M′(x):
        run M on blank input
        if x is of the form 1^i where 1 ≤ i ≤ 374
            return TRUE
        else
            return FALSE
  if DECIDEAE374(⟨M′⟩)
      return TRUE
  else
      return FALSE
```

We prove this reduction correct as follows:

$\Longrightarrow$ Suppose M halts on blank input.

Then M' halts on and accepts exactly 374 distinct input strings, the strings in the language $\left\{1^i \;\middle|\; 1 \le i \le 374\right\}$.

So DECIDEAE374 accepts the encoding $\langle M' \rangle$.

So DECIDEHALT correctly accepts the encoding $\langle M \rangle$.

$\Longleftarrow$ Suppose M does not halt on blank input.

Then M' diverges (i.e., does not halt) on *every* input string x.

Therefore, M' does not halt and accept on more than 374 distinct input strings.

So DECIDEAE374 rejects the encoding $\langle M' \rangle$.

So DECIDEHALT correctly rejects the encoding $\langle M \rangle$.

In both cases, DECIDEHALT is correct, that is, there exists a decider for HALT. But that is impossible, because HALT is undecidable. We conclude that the algorithm DECIDEAE374 does not exist, i.e., L_{AE374} is undecidable.

The choice of M' only accepting $x \in \left\{1^i \;\middle|\; 1 \le i \le 374\right\}$ when M halts was arbitrary, one could have chosen any other set S of 374 distinct strings and had M' accept $x \in S$ instead. ■

Rubric: 10 points. Standard undecidability reduction rubric.

4 Longest Increasing Subsequence

Given a rooted tree T with vertex labels, each root-leaf path of T can be treated as a sequence. One can thus consider a *subsequence of a root-leaf path* of T. If the vertex labels are *integers*, then we can ask about the ***longest increasing subsequence of a root-leaf path*** (recall that a sequence $A[1..\ell]$ is *increasing* if $A[i] > A[i-1]$ for all $i \geq 2$).

Describe an algorithm that finds the length of a longest increasing subsequence of a root-leaf path in a rooted tree with integer vertex labels.

Solution (Two parameters): For vertex u we will use the notation $A[u]$ to refer to the integer label of u. Let r be the root of T. Add a new sentinel root vertex s with label $-\infty$; the only child of s is r.

Let $LIS(u, v)$ denote the length of a longest increasing subsequence of a root-leaf path of the *subtree rooted at* v, where every element is larger than $A[u]$.

We need to compute $\boldsymbol{LIS(s, r)}$.

The function obeys the recurrence

$$LIS(u,v) = \begin{cases} \max_{w \text{ child of } v} \{LIS(u,w)\} & \text{if } A[u] \geq A[v] \\ \max \left\{ \begin{array}{c} \max_{w \text{ child of } v} \{LIS(u,w)\} \\ 1 + \max_{w \text{ child of } v} \{LIS(v,w)\} \end{array} \right\} & \text{otherwise} \end{cases}$$

If v is a leaf then we are taking the max over an empty set, which we assume to be 0, so no explicit base case is necessary.

Assume that the vertices of T are indices between 1 and n, and set $s = 0$. We can memoize LIS into a two-dimensional array $LIS[0..n, 1..n]$. Each entry $LIS[u, v]$ depends only on entries $LIS[\cdot, w]$ where w is a child of v, so we can fill the *columns* in reverse topological order (i.e., via a post-order traversal) in the outer loop, and the entries of each column in arbitrary order in the inner loop.

```
LIS(T):
  A[0] ← −∞ ⟨⟨Add a sentinel⟩⟩

  v_0, ..., v_n ← a topological ordering of the vertices of T w/ sentinel
  for j ← n down to 1
      for i ← j − 1 down to 0
          LIS[v_i, v_j] ← 0
          for v_k child of v_j
              LIS[v_i, v_j] ← max{LIS[v_i, v_j], LIS[v_i, v_k]}
              if A[v_i] < A[v_j]
                  LIS[v_i, v_j] ← max{LIS[v_i, v_j], 1 + LIS[v_j, v_k]}
  return LIS[v_0, v_1] ⟨⟨v_0 and v_1 are s and r, respectively⟩⟩
```

Despite the triple for loop, each edge is considered only once per v_i in the inner loop, so the algorithm runs in $\boldsymbol{O(n^2)}$ ***time***. ■

Solution (One parameter): For vertex u we will use the notation $A[u]$ to refer to the integer label of u. We will add a new sentinel root vertex s with label $-\infty$, and make the (now former) root of T the only child of s.

Let $LIS(u)$ denote the length of a longest increasing subsequence of a root-leaf path that *starts at* u. We need to compute $\boldsymbol{LIS(s)-1}$; the -1 removes the sentinel $-\infty$ from the subsequence. The function obeys the recurrence

$$LIS(u) = 1 + \max\left\{LIS(v) \;\middle|\; v \text{ is a descendant of } u \text{ and } A[u] < A[v]\right\}$$

Each $LIS(u)$ depends $LIS(v)$ where v is a descendant of u, so we will memoize LIS in *reverse* topological order.

```
LIS(T):
  ⟨⟨Add a sentinel⟩⟩
  Add vertex s as the new root
  A[s] ← −∞

  v_0, ..., v_n ← a topological ordering of the vertices of T w/ sentinel
  for i ← n down to 1
      LIS[v_i] ← 1
      for v_j in a traversal of the subtree rooted at v_i
          if A[v_i] < A[v_j]
              LIS[v_i] ← max{LIS[v_i], 1 + LIS[v_j]}
  return LIS[v_0] − 1 ⟨⟨v_0 is the sentinel⟩⟩
```

Overall the algorithm takes $\boldsymbol{O(n^2)}$ ***time***. ■

Solution (Reduction to Longest Path in DAG): For vertex u we will use the notation $A[u]$ to refer to the integer label of u.

We will create a new DAG G from T as follows. We start with the vertices of T, and then include edge $u \to v$ if and only if v is a descendant of u and $A[u] < A[v]$. We then add a sentinel source vertex s with an edge $s \to u$ to each vertex u in G. *This is exactly the dependency DAG of the one-parameter LIS recurrence in the previous solution!*

Now an increasing subsequence of a path in T corresponds directly to a path in G, and so we can find the length of a longest increasing subequence of a path in T by simply finding the length of a longest path in G starting from s via Dynamic Programming.

G consists of $n+1$ vertices and $O(n^2)$ edges, and can be constructed via brute force in $O(n^2)$ time. Finding a longest path in G from s takes $O(V+E) = \boldsymbol{O(n^2)}$ ***time***. ■

Rubric: 10 points. Standard dynamic programming or graph reduction rubric, as appropriate. −2 for slower polynomial time algorithms.

5 Matching to SAT

Let $G = (V, E)$ be a given graph and let $X \subseteq V$ be a subset of the vertices. A set of edges $M \subseteq E$ is called a *matching* if no vertex is the end point of more than one edge in M. A matching M is said to saturate X if every vertex in X has an edge of M incident to it.

Given $G = (V, E)$ and $X \subseteq V$ we say that G has an X-saturating matching if there is a matching $M \subseteq E$ that saturates X. This is a decision problem. Describe a polynomial time reduction from this problem to SAT. A natural way to do this is to use a variable x_e for each edge $e \in E$, and then writing appropriate constraints over the variables.

- Formally, describe an efficient algorithm that takes as input G and X and outputs a SAT formula φ_G such that G has a matching that saturates X iff φ_G is satisfiable. You need to briefly justify the correctness of your reduction. *(You may want to do the next part first.)*

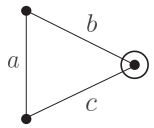
  Solution: We will create a variable x_e for each $e \in E$. A variable being 1 in a satisfying assignment to φ_G will imply that the corresponding edge is in the matching.

  - To encode the constraint that no vertex is the end point of more than one edge in the matching, we will define the following clauses for each vertex $u \in V$: $\bigwedge_{e,f \text{ incident to } u} (\neg x_e \vee \neg x_f)$. This ensures that two incident edges of u cannot simultaneously be in the matching.
  - To encode the constraint that each vertex in X has an incident edge in the matching, we will define the following clause for each vertex $u \in X$: $\left(\bigvee_{e \text{ incident to } u} x_e\right)$. This ensures that at least one edge incident to u appears in the matching.

  The boolean formula φ_G is obtained by taking the AND of the clauses described above.

  The algorithm considers all pairs of edges for each endpoint when enumerating the clauses of the first type, and at once for each endpoint when enumerating the clauses of the second type. Thus the running time is $\boldsymbol{O(V + E^2)}$, which is polynomial in the size of the input. ■

- For the graph shown below with edges a, b, c and a single node in X, what is the formula obtained from your reduction?



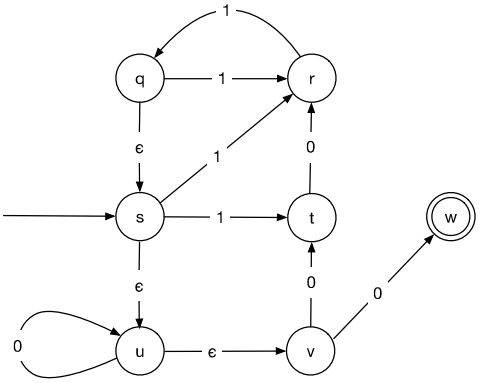
  Solution: $(\neg x_a \vee \neg x_b) \wedge (\neg x_a \vee \neg x_c) \wedge (\neg x_b \vee \neg x_c) \wedge (x_b \vee x_c)$. ■

Rubric: 10 points.
- 8 points for the first part:
  - + 3 points for each type of constraint: 2 points for the constraints, 1 point for justification.
  - + 1 point for putting φ_G together
  - + 1 point for time analysis ("polynomial-time" suffices).
- 2 points for the second part, all or nothing.

6 Shortest string accepted by a NFA

Recall that an NFA N is specified as $(Q, \delta, \Sigma, s, A)$ where Q is a finite set of states, Σ is a finite alphabet, $s \in Q$ is the start state, $A \subseteq Q$ is the set of accepting states and $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ is the transition function.

- In the NFA N shown below, what is the length of a shortest string accepted by N?



Solution: The string `0` is accepted via the path $s \xrightarrow{\varepsilon} u \xrightarrow{\varepsilon} v \xrightarrow{0} w$. However the string ε has no accepting path in N. So the answer is $\boxed{1}$. ■

- Describe an efficient algorithm that takes a description of an NFA $N = (Q, \Sigma, \delta, s, A)$ and outputs the length of a shortest string that is accepted by N. Express the running time of your algorithm in terms of n the number of states of N and $m = \sum_{p \in Q} \sum_{b \in \Sigma \cup \{\varepsilon\}} |\delta(p, b)|$ which is the natural representation size of the NFA's transition function. If N does not accept any strings then your algorithm should report that.

Solution: Create a directed graph G whose vertices are the states of N, where for states q, r we will have an edge $q \to r$ of length 0 if $r \in \delta(q, \varepsilon)$ and an edge $q \to r$ of length 1 if $r \notin \delta(q, \varepsilon)$ but $r \in \delta(q, a)$ for some $a \in \Sigma$. The length of a shortest string that is accepted by N is exactly the length of a shortest path in G from s to some $q \in A$ (if no string is accepted by N then the shortest path lengths to each $q \in A$ will be ∞).

G has n vertices and at most m edges, and can be computed via brute force enumeration of δ in $O(m+n)$ time, after which one can find the shortest path lengths from s to each $q \in A$ via Dijkstra's algorithm in $O(E + V \log V) = \boldsymbol{O(m + n \log n)}$ ***time***. Note that the edges in G have only two possible edge lengths, so actually one can reduce the running time of Dijkstra's algorithm to $O(V + E) = \boldsymbol{O(m + n)}$ ***time*** by implementing the priority queue correctly (you should think about how). ■

Rubric: 10 points.
- 2 points for the first part, all or nothing.
- 8 points for the second part: scaled graph reduction rubric.
  No penalty for not optimizing the priority queue or citing $O(E \log V)$ for Dijkstra's algorithm.

Rubric (Standard dynamic programming rubric): For problems worth 10 points:

- 6 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - + 1 point for a clear **English** description of the function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.) **Automatic zero if the English description is missing.**
  - + 1 point for stating how to call your function to get the final answer.
  - + 1 point for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 points for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error. **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- 4 points for details of the dynamic programming algorithm
  - + 1 point for describing the memoization data structure
  - + 2 points for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested loops, be sure to specify the nesting order.
  - + 1 point for time analysis
- Unless otherwise specified, it is *not* necessary to state a space bound.
- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem says otherwise.
- Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, ***but iterative pseudocode is not required for full credit***. If your solution includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. (But you still need to describe the underlying recursive function in English.)

Rubric (Standard rubric for graph reduction problems): For problems out of 10 points:

- + 1 for correct vertices, *including English explanation for each vertex*
- + 1 for correct edges
  - − ½ for forgetting "directed" if the graph is directed
- + 1 for stating the correct problem
  - − "Breadth-first search" is not a problem; it's an algorithm!
- + 1 for correctly applying the correct algorithm
  - − ½ for using a slower or more specific algorithm than necessary
- + 1 for time analysis in terms of the input parameters.
- + 5 for other details of the reduction
  - − If your graph is constructed by naive brute force, you do not need to describe the construction algorithm; in this case, points for vertices, edges, problem, algorithm, and running time are all doubled.
  - − Otherwise, apply the appropriate rubric, *including Deadly Sins*, to the construction algorithm. For example, for a solution that uses dynamic programming to build the graph quickly, apply the standard dynamic programming rubric.

Rubric (Standard undecidability reduction rubric): 10 points =

- \+ 2 points for the reduction itself
- \+ 4 points for the "if" proof of correctness
- \+ 4 points for the "only if" proof of correctness
- • A reduction in the wrong direction is worth 0/10.

Rubric (Standard polynomial-time reduction rubric): 10 points =

- \+ 3 points for the reduction itself
- \+ 3 points for the "if" proof of correctness
- \+ 3 points for the "only if" proof of correctness
- \+ 1 point for writing "polynomial time"
- • An incorrect polynomial-time reduction that still satisfies half of the correctness proof is worth at most 4/10.
- • A reduction in the wrong direction is worth 0/10.