EECS 376 Midterm Exam, Winter 2023

Standard Languages

You may rely on the following definitions without repeating them.

- $L_{\text{ACC}} = \{(\langle M \rangle, x) : M$ is a Turing machine that accepts $x\}$
- $L_{\text{HALT}} = \{(\langle M \rangle, x) : M$ is a Turing machine that halts on $x\}$
- ~~$L_{\varepsilon\text{-HALT}} = \{\langle M \rangle : M$ is a Turing machine that halts on $\varepsilon\}$~~
- ~~$L_{\emptyset} = \{\langle M \rangle : M$ is a Turing machine for which $L(M) = \emptyset\}$~~
- ~~$L_{\text{EQ}} = \{(\langle M_1 \rangle, \langle M_2 \rangle) : M_1, M_2$ are Turing machines for which $L(M_1) = L(M_2)\}$~~

Multiple Choice – 36 points

For each question in this section, **select exactly ONE answer by completely filling its circle** with a pencil or black ink. Each question in this section is worth 4 points.

1. Consider the following algorithm:

```
function FUNC($A[1,\dots,n]$)
    if $n = 1$ then
        return $A[1]$
    $w \leftarrow$ Func($A[1,\dots,\lceil 3n/8 \rceil]$)
    $x \leftarrow$ Func($A[\lceil 3n/8 \rceil + 1,\dots,\lceil 6n/8 \rceil]$)
    $y \leftarrow$ Func($A[\lceil 5n/8 \rceil + 1,\dots,n]$)
    $z \leftarrow$ Helper($A[1,\dots,n]$)
    return max($w, x, y, z$)
```

Suppose that Helper takes $O(n)$ time on an array of n elements.

Choose the **tightest correct asymptotic bound** on the runtime of Func($A[1,\dots,n]$).

- ◯ $O(n)$
- ◯ $O(n^2)$
- [✓ $O(n^{\log_{8/3} 3})$]
- ◯ $O(n \log n)$

Solution: The recurrence for the runtime of this algorithm is

$$T(n) = 3T(3n/8) + O(n) = 3T\left(\frac{n}{8/3}\right) + O(n).$$

We have $k = 3, b = 8/3, d = 1$, so $k/b^d = 3/(8/3)^1 = 9/8 > 1$. From the Master Theorem, we see that $T(n) = O(n^{\log_b k}) = O(n^{\log_{8/3} 3})$.

2. On input n, algorithm M's worst-case runtime is $O(n^2)$, whereas algorithm N's is $\Theta(n^2 \log n)$. Choose the one statement that is **necessarily true**.

   ◯ On any input n, M's runtime is less than N's.

   ✓ **For any input n larger than some fixed threshold, M's runtime is less than 1% of N's.**

   ◯ For some (possibly small) input n, M's runtime is more than N's.

   ◯ For any input $n \geq 2^{10}$, M's runtime is at most 10% of N's.

Solution: Let $f(n), g(n)$ denote the runtimes of M, N (respectively) on input n. The key insight here is that $f(n) = o(g(n))$, because we have the upper bound $f(n) = O(n^2)$ and the lower bound $g(n) = \Omega(n^2 \log n)$. Recall that $f(n) = o(g(n))$ means that $\lim_{n\to\infty} f(n)/g(n) = 0$.

The first choice is incorrect because it could be the case that $f(n) \geq g(n)$ for "small" n below some threshold, even though $f(n) \ll g(n)$ for all "large" n.

The second choice is correct by the properties of limits: essentially, the ratio $f(n)/g(n)$ remains arbitrarily close to zero as n grows. Formally, for any positive constant $c > 0$ (and in particular, for $c = 1/100 = 1\%$), there is some fixed threshold n_0 for which $f(n) < c \cdot g(n)$ for all $n \geq n_0$.

The third choice is incorrect because it could be the case that $f(n) \leq g(n)$ for all n.

The fourth choice is incorrect because the "crossover" point n_0 at which M's runtime becomes less than N's might be much larger than 2^{10}. The asymptotic bounds guarantee that such a crossover point exists, but do not give any information about where it is.

3. Consider the following algorithm:

```
function PrintExamQuestion($n$)
    while $n > 0$ do
        Print "Why is a raven like a writing desk?"
        $n \leftarrow \lfloor 3n/4 \rfloor$
    Print "One is nevar backwards and one is for words"
```

   Choose the **tightest correct asymptotic bound** on the algorithm's runtime. (Assume that arithmetic operations take constant time.)

   ◯ $O(n^{4/3})$

   ◯ $O(n)$

   ✓ $O(\log n)$

   ◯ $O(1)$

Solution: Every iteration shrinks the value of n by a multiplicative factor of at least 4/3 (due to taking the floor), so there are at most $\log_{4/3} n = O(\log n)$ iterations. (Note also that the loop will eventually terminate, because $n > 0$ is equivalent to $n \geq 1$ since n is always a nonnegative integer.)

This can also be solved using the Master Theorem, if we view the loop as making a recursive call on the updated value of n. Then we have a runtime recurrence of $T(n) = T(3n/4) + O(1) = T(\frac{n}{4/3}) + O(1)$, so $k = 1, b = 4/3, d = 0$. Since $\log_b k = \log_{4/3} 1 = 0 = d$, the solution is $T(n) = O(\log n)$.

4. Choose the correct option: (All / some / none) of the following statements are **true**:

- Karatsuba's algorithm for multiplying n-bit integers is asymptotically faster than $\Theta(n^2)$-time naïve multiplication because it makes three recursive calls on integers of about $n/2$ bits, and its combine step takes $O(n)$ time.
- The MergeSort algorithm for sorting an n-element array is asymptotically faster than $\Theta(n^2)$-time naïve sorting because it makes two recursive calls on arrays of about $n/2$ elements, and its combine step takes $O(n)$ time.
- The divide-and-conquer algorithm for finding a closest pair of points in two dimensions, if modified to compute <u>all</u> pairwise distances between points in the "δ-strip," is not asymptotically faster than the $\Theta(n^2)$-time brute-force algorithm.

[✓ **All**]
◯ Some (but not all)
◯ None

Solution: Karatsuba's algorithm has a time complexity of the form $T(n) = 3T(n/2) + O(n) = O(n^{\log_2 3})$, which is asymptotically faster than $\Theta(n^2)$.

MergeSort has a time complexity of the form $T(n) = 2T(n/2) + O(n) = O(n \log n)$, which is asymptotically faster than $\Theta(n^2)$.

The divide-and-conquer algorithm for finding a closest pair of points in two dimensions, when modified as specified, has a time complexity of the form $T(n) = 2T(n/2) + O(n^2) = O(n^2)$, which is not asymptotically faster than the $\Theta(n^2)$-time brute-force algorithm. In particular, there exist inputs for which there are $\Theta(n)$ points in the "δ-strip". Hence, there are $\Theta(n^2)$ pairs of points in the strip. So the algorithm has a time complexity of the form $T(n) = 2T(n/2) + \Theta(n^2) = \Theta(n^2)$.

5. True or False: any DFA, given any input string (made up of characters from the DFA's alphabet), either accepts or rejects.

[✓ **True**]
◯ False
◯ Determining the answer is undecidable

Solution: A DFA processes its input string one character at a time, and after processing the final character, it either accepts or rejects according to the final state.

6. Choose the option that **makes the following statement true**: For (all / some / no) decidable languages L_1, L_2, L_3, the language

$$L = \{x \in \Sigma^* : x \text{ is in } \textbf{exactly 2 of } L_1, L_2, L_3\}$$

is decidable.

[✓ **All**]
◯ Some (but not all)
◯ No
◯ Determining the answer is undecidable

Solution: The languages L_1, L_2, L_3 are all decidable, so there are TMs M_1, M_2, M_3 that decide them (respectively). We can construct a decider M for L that works as follows: given an input x, run each of the three deciders M_i on x (in sequence). If exactly two of the three deciders accept, then accept; otherwise, reject. This M halts on any input, because all the deciders M_i do. And, by construction of M and by hypothesis on the M_i, it is clear that M accepts x if and only if $x \in L$, i.e., x is in exactly 2 of the languages L_i.

7. Choose the **only false statement** from the following.

[✓ **There exists a language L that is decidable by a program written in Python, but is not decidable by any Turing machine.**]
◯ Any language that is decidable by a DFA is also decidable by some Turing Machine.
◯ There exists a C++ program that recognizes $\emptyset$ (the empty set).
◯ If L_1 is undecidable and $L_1 \leq_T L_2$, then L_2 is undecidable.

Solution: The first statement is false because every legal Python operation can be implemented in (say) some assembly language, and a Turing machine can implement all the operations in the assembly language.

The second statement is true because every DFA has an equivalent TM that performs essentially the same computational steps as the DFA. More specifically, it has the DFA's state transitions, always moving its head to the right (and writing an arbitrary symbol), and it has 'blank'-transitions from every DFA accept state to the TM's accept state, and similarly from every DFA non-accept (reject) state to the TM's reject state. This ensures that the TM makes the same decision as the DFA would make, as soon as it reaches the end of the input string.

The third statement is true, as exhibited by the following program that simply rejects every input string:

```
        bool func(void *w)
        {
            return false; // equivalent to reject
        }
```

The fourth statement is true; it is a theorem from lecture. Recall that $L_1 \leq_T L_2$ means that there exists a Turing machine that, given access to a membership oracle for L_2, decides L_1. Supposing for contradiction that L_2 were decidable, then its decider TM could be used to implement the membership oracle, therefore an ordinary TM (without any oracle) would be able to decide L_1. But L_1 is undecidable by assumption, so we have reached a contradiction. We conclude that L_2 is undecidable.

8. Choose the **only decidable language** from the following.

   ○ $L_A = \{(\langle M\rangle, x) : M$ is a TM that accepts x after running for more than 376 steps$\}$

   [✓ $L_B = \{(\langle D\rangle, x) : D$ **is a DFA that accepts** $x\}$]

   ○ $L_C = \{(\langle M_1\rangle, \langle M_2\rangle) : M_1, M_2$ are TMs and $L(M_1) \subseteq L(M_2)\}$

   ○ $L_D = \{\langle M\rangle : M$ is a TM that accepts some input$\}$

Solution: The first language L_A can be proven undecidable via a straightforward reduction from L_{HALT}. The reduction construts a machine that does nothing for 376 steps, then runs $M(x)$ and outputs whatever M outputs. The analysis is an exercise.

The second language L_B can be decided by a TM that just simulates the input DFA D on x and outputs whatever D does. Crucially, because a DFA always 'halts' after processing all the characters of the input string (it cannot loop), the simulation will eventually halt, hence so will our TM.

The third language L_C was shown to be undecidable in the homework.

The fourth language L_D can be proven undecidable via a reduction from L_{ACC}. The reduction constructs a machine that ignores its input, runs $M(x)$, and output whatever M outputs. The analysis is an exercise.

9. Choose the **only true statement** from the following.

   ○ If language L is undecidable and Turing machine D is a decider, then there exists some $x \in L$ that D rejects.

   ○ If language L is undecidable, then even though no Turing machine decides L, there does exist a Turing machine M that accepts every $x \in L$ and does not accept every $x \notin L$.

   [✓ **If language L is undecidable and Turing machine D does not reject any $x \in L$ and does not accept any $x \notin L$, then D must loop on some input.**]

◯ If language L is undecidable and D is a Turing machine, then at least one of these must hold: (1) D rejects or loops on every $x \in L$; (2) D accepts or loops on every $x \notin L$.

Solution: The first statement is false. For example, D could accept all inputs (and thus not reject any input).

The second statement is false. If L is unrecognizable, then there is no such Turing machine. (The listed conditions on M are exactly the definition of a recognizer for L.)

The third statement is true. For contradiction/contrapositive, suppose such a D does not loop on any input. Then it would accept every $x \in L$ and reject every $x \notin L$, making it a decider for L. But L is undecidable, so this cannot be.

The fourth statement is false. Even though L is undecidable, we can "hard code" some fixed $x \in L$ and some $x' \notin L$ into a TM D for it to accept and reject, respectively. In other words, a TM can compute 'correct' answers (with respect to an undecidable L) for <u>some</u> inputs; but it cannot output correct answers on <u>all</u> inputs.

UNIQNAME (print): ____________________

Shorter Answer – 24 points

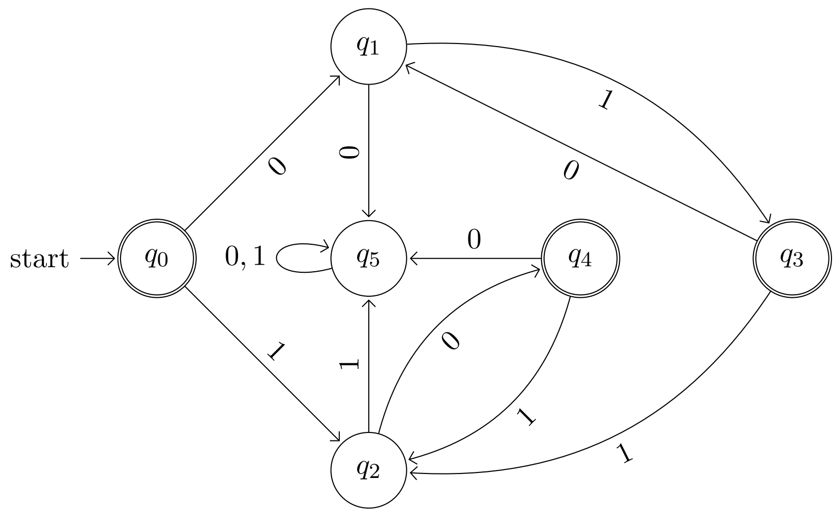
Each of the three questions in this section is worth 8 points.

1. Give the **tightest correct asymptotic** (big-O) bound, as a function of n, on the **worst-case number of additions** done by the following algorithm, along with **a value of** k **that induces the worst case**. Also **state whether this is polynomial in the input size** or not. No explanation or proof is needed.

```
function FUNK($n, k$)                ▷ $n$ is a positive integer, and $k \in \{1, 2, \ldots, n\}$
    $x = 0$
    for $i = 1, 2, \ldots, k$ do
        for $j = 1, 2, \ldots, n - k$ do
            $x = x + 1$
    return $x$
```

Solution: The number of additions is $k(n-k)$. (We could also include the additions done to increment i and j in the loops, but this increases the total number by at most a factor of 3, which does not affect the asymptotics.) The worst case arises (i.e., this expression attains its maximum) when $k = \lfloor n/2 \rfloor$, which gives a tight bound of $O(n^2)$ in terms of n. This is **not** polynomial—it is exponential—in the input size, because the input size is $\log n + \log k = \log(nk) = \Theta(\log n)$.

2. What language does the following DFA decide? Give your answer in "regex" form, or in precise English. No explanation or proof is needed.



UNIQNAME (print): ______________________________

Solution: The DFA decides the language $(01)^*(10)^*$.

There are 3 accept states, q_0, q_3, q_4. The only input string which can cause the DFA to stop in q_0 is ε.

The only input strings which would cause the DFA to stop in q_3 are those of the form $(01)^+$ following the edges between q_1 and q_3.

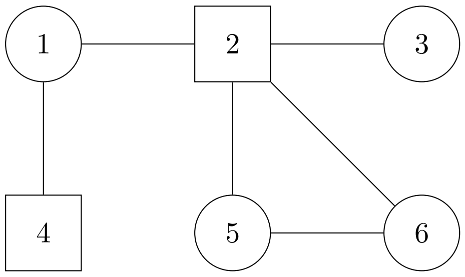
The only input strings which would cause the DFA to stop in q_4 are those of the form $(10)^+$ following the edges between q_0, q_2, q_4 and those of the form $(01)^+(10)^+$ following the edges between q_0, q_1, q_3, q_2, q_4.

So the DFA decides the language consisting of strings matching the regular expression $(\varepsilon|(01)^+|(10)^+|(01)^+(10)^+)$. This can be simplified to the language consisting of strings matching the regular expression $(01)^*(10)^*$.

3. A dominating set S in a graph G is a set of vertices for which every vertex of G either is in S, or is adjacent to some vertex in S.

   We are interested in a smallest dominating set of a given graph, i.e., one that has the fewest possible vertices. (There may be more than one smallest dominating set.)

   For example, the following graph has a smallest dominating set $S^* = \{2, 4\}$: every vertex other than 2 and 4 is adjacent to 2 or 4 (or both), and there is no dominating set consisting of a single vertex.



   Consider the following greedy algorithm for finding a dominating set in a graph.

```
1: function GreedyDS($G$)
2:     $S \leftarrow \emptyset$
3:     while $G$ has at least one vertex do
4:         Select any vertex $v$ in $G$ that has largest degree (i.e., the most neighbors)
5:         Add $v$ to $S$
6:         Remove $v$ and all its neighbors, including all incident edges, from $G$
7:     return $S$
```

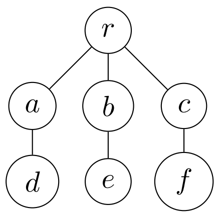
   Give a small graph G on which the algorithm **might not return a smallest dominating set**. Specifically, **give a sequence of vertices that the algorithm might choose** to make up its final output set, and **give an optimal dominating set of G that is smaller** than this output set.

UNIQNAME (print): ______________________________

You may use the box on the next page to continue your answer. If you do, clearly write that your "answer continues to the next page."

Solution:

There are many different graphs on which this algorithm does not (necessarily) give a smallest dominating set. Here is one (which does not even rely on the algorithm 'breaking ties' in a certain way when choosing vertices v):



A smallest dominating set is $\{a, b, c\}$. By inspection, no single vertex forms a dominating set of the graph. No set of two vertices works either, because at least one vertex from each pair (a, d), (b, e), and (c, f) must be taken to form a dominating set.

The given algorithm will first add r to S, because it is the unique vertex with largest degree (3). After this, the only vertices left in the graph are d, e, f, and the algorithm takes all three of them because no edges remain. So the algorithm outputs $\{r, d, e, f\}$ as its dominating set, which is not optimal.

UNIQNAME (print): ______________________________

Proofs and Longer Answers – 40 points

Each of the two questions in this section is worth 20 points.

1. Let $A[1, \dots, n]$ be an array of $n \geq 1$ positive real numbers. A decreasing subsequence of A is a sequence of array elements $A[i_1] > A[i_2] > \cdots > A[i_m]$, where $i_1 < i_2 < \cdots < i_m$ are some (not necessarily contiguous) array indices.

   We are interested in the maximum sum obtainable by decreasing subsequences of A, i.e.,

   $$\max\{A[i_1] + \cdots + A[i_m] : A[i_1] > A[i_2] > \cdots > A[i_m] \text{ is a decreasing subsequence of } A\}.$$

   For example, for $A = [4, 2, 2, 3, 5, 1]$, both $S_1 = [4, 3, 1]$ and $S_2 = [5, 1]$ are decreasing subsequences. The sum over S_1 is $8 = 4 + 3 + 1$, whereas the sum over S_2 is $6 = 5 + 1$. It can be verified that S_1 has the maximum sum overall, i.e., no decreasing subsequence of A has a sum greater than 8. (Note that the subsequence $[4, 2, 2, 1]$ has a sum of 9, but it is not decreasing, because $2 \not> 2$.)

   (a) Let $H(i)$ denote the maximum sum obtainable by a decreasing subsequence of A that ends at index i, i.e.,

   $$H(i) = \max\{A[i_1] + \cdots + A[i_m] : A[i_1] > \cdots > A[i_m] \text{ is a decreasing subsequence of } A \text{ with } i_m = i\}.$$

   Give a correct recurrence relation for $H(i)$, including base case(s), that is suitable for an efficient dynamic-programming algorithm. Briefly justify your answer.

   Hint: an optimal decreasing subsequence ending at index i is either a single element, or has an immediate predecessor (a second-to-last element). What are the possible indices for this predecessor? What is true about the part of the subsequence that ends at this predecessor?

   Solution: The base case is $H(1) = A[1]$, because $A[1]$ is the only decreasing subsequence of A that ends at index 1. The recurrence is

   $$H(i) = A[i] + \max_{\substack{j : 1 \leq j < i, \\ A[j] > A[i]}} H(j),$$

   where we define the max to be zero if there are no j that satisfy the constraints.

   A rigorous proof of correctness is as follows. (In grading, we are only looking for the key observations.) Let S be some optimal decreasing subsequence ending at index i. We consider the index j of the second-to-last term, when there is one. By definition, we must have $j < i$ and $A[j] > A[i]$.

   Define S' to be the decreasing subsequence obtained by removing the final element $A[i]$ from S; so, S' ends at index j. We claim that S' must be an optimal decreasing subsequence that ends at index j. For if it were not, then there would be some better one S'' (i.e., having a strictly larger sum than S'). Because both S' and S'' are decreasing subsequences that end with $A[j] > A[i]$, we could substitute S'' for S'

in our original subsequence S. This would yield a decreasing subsequence ending at $A[i]$ with a strictly larger sum than S's, contradicting S's assumed optimality.

Note that $H(i)$ is (by definition) the sum of S's elements. Either S is the single element $A[i]$, or else (as shown above) removing its last element yields an optimal decreasing subsequence that ends at some index $j < i$ for which $A[j] > A[i]$. The optimal sum $H(i)$ must be the largest one that can be obtained via these options. The recurrence written above directly corresponds to this fact.

(Note that the single-element subsequence $A[i]$ is necessarily worse than any multiple-element subsequence ending with $A[i]$, because the elements of A are positive integers. So, when a multiple-element subsequence ending at $A[i]$ is possible, i.e., some valid j exists, the recurrence does not need to compare against the single-element option. Conversely, when only the single-element case is possible, it is accurately captured by defining the max expression to zero.)

(b) Using your answer to part (a), **describe a dynamic-programming algorithm** that outputs the maximum sum over all decreasing subsequences of an input array A. Also **give the tightest correct asymptotic (big-O) running time** for your algorithm, as a function of n. Assume that addition, comparisons, and similar basic operations take constant time.

Solution: There are n distinct subproblems $H(i)$, for $1 \leq i \leq n$. So, we iterate bottom-up from $i = 1$ to n, filling in an n-entry array that computes and stores the values of $H(i)$ by using the above recurrence relation and the previously computed values in the array. Finally, we output the maximum over all values in this array, reflecting the fact that an optimal decreasing subsequence in A must end at some index $1 \leq i \leq n$.

We have n distinct subproblems. Furthermore, we do $O(n)$ work to solve each subproblem $H(i)$, since we examine all previously solved subproblems $H(j)$ and maximize over the ones that satisfy the conditions $j < i$ and $A[j] > A[i]$. So, solving all the subproblems takes $O(n \cdot n) = O(n^2)$ work. Finally, we have $O(n)$ work at the end to find the maximum over all $H(i)$. As a result, the overall runtime is $O(n^2) + O(n) = O(n^2)$.

2. ~~Define the language~~

$$L = \{(\langle M \rangle, x) : M \text{ is a TM and } x \text{ is a \underline{shortest} string that } M \text{ accepts}\}.$$

~~Prove that L is undecidable by **showing one of** $L_{\text{ACC}} \leq_T L$ or $L_{\text{HALT}} \leq_T L$.~~