EECS 376 Midterm Exam, Winter 2022

Multiple Choice – 30 points

Each question has one answer unless otherwise indicated. Fully shade in the circle with a pencil or black ink. Each question is worth 3 points.

1. Consider the following code. Note that the variables x and y are real numbers.

```
Require: $x > y > 0$ are real numbers                ▷ Hint: This actually matters...
function Foo376($x, y$)
    if $x \leq 0$ then return 1;
    $z \leftarrow$ Foo376($x - \log y, y$)
    return ($z + 1$)
```

Which of the following is a valid potential function for the algorithm Foo376 (above)?

- ◯ $s = x + y$
- ◯ $s = e^y - x$
- ◯ $s = x$
- ◯ $s = x + 5y$
- [✓ **None of the above**]

Solution: If we choose some $y < 1$, then $\log y$ will always be negative, so that x is strictly increasing. Thus, the condition $x \leq 0$ will never become false. There is no valid potential function because Foo376 does not halt on all inputs.

2. In a new software company "Three-and-Out" a new "three-way" mergesort algorithm is used. The new algorithm splits the array into three equal parts instead of two equal parts, then sorts the three sub-arrays recursively and at the end merges the sorted sub-arrays. What is the correct recurrence relation for this new algorithm?

- ◯ $T(n) = 2T(n/3) + O(n)$
- ◯ $T(n) = 3T(n/2) + O(n \log n)$
- [✓ $T(n) = 3T(n/3) + O(n)$]
- ◯ $T(n) = 2T(n/2) + O(n)$
- ◯ $T(n) = 3T(n/2) + O(n)$
- ◯ None of the above

Solution: (C)

In each of the splitting, three new sub-arrays are generated and each of them have a length of $\frac{n}{3}$. The algorithm then runs the following two stages: recursion on three different sub-arrays, with a total time complexity of $3 \times T(n/3)$; merging all three sub-array into a larger array. In the second stage we can potentially use three pointers on the three sub-arrays and increment the smallest one, append it to the final array. Which shows that the second stage has a time complexity of $O(n)$.

Thus the time complexity is $T(n) = 3T(n/3) + O(n)$

3. Let $T(n) = 4T(n/3) + O(\sqrt{n})$. What is the tightest asymptotic upper bound for $T(n)$?
   - ○ $O(n)$
   - ○ $O(n^2)$
   - ○ $O(\log n)$
   - ○ $O(n \log n)$
   - [✓ $O(n^{\sqrt{2}})$]
   - ○ $O(2^n)$

Solution: Using the Master Theorem. $k = 4$, $b = 3$, $d = 0.5$. So $\frac{k}{b^d} > 1$. From the Master Theorem we get $O(n^{log_b k})$ or $O(n^{log_3 4})$. That's about $O(n^{1.26})$. Of the options, $n^{\sqrt{2}}$ is the tightest bound.

4. Suppose $S_1 \subseteq \mathbb{R}$ is an <u>uncountable</u> set and $S_2 \subseteq \mathbb{R}$ is a <u>countable</u> set. Which of the following will always true?

   i Their intersection $S_1 \cap S_2$ is countable.
   ii Their union $S_1 \cup S_2$ is countable.
   iii The complement of S_1 is sometimes, but not always, countable.

   ◯ i and ii
   ◯ only i
   [✓ **i and iii**]
   ◯ only iii
   ◯ ii and iii

Solution: Choice i is correct. Consider the intersection of the set of reals (uncountably infinite) and the set of integers. Their intersection is just the set of integers which are countably infinite.

Choice ii is incorrect. For contradiction assume this is true and the union is countably infinite. Then as the uncountably infinite language is a subset of this union it should also be countably infinite. This is clearly a contradiction.

Choice iii is correct. Consider the set where the Universe is the reals. The set of all the reals other than the integers has the integers as a complement and the integers are countable.

Now consider the set of of the non-positive reals. The complement of that, where the universe is the reals, are the positive reals, which are uncountable.

This makes option C the correct answer.

5. A bottom-up dynamic programming algorithm fills the entire table, where the table has dimensions $n \times m$. What can be said about the runtime of the algorithm?

   ◯ $O(nm)$
   ◯ $\Theta(nm)$
   ◯ $o(nm)$
   [✓ $\Omega(nm)$]
   ◯ No conclusions can be drawn from the information given.

Solution: The bottom-up dynamic programming approach involves filling each entry in the table. There are $n \times m$ entries in the table, and each gets filled. This takes at least nm steps, which gives us a lower-bound of $\Omega(nm)$ for the algorithm. Note that $O(nm)$ and $\Theta(nm)$ only hold if filling each entry takes constant time, and that is not always the case.

6. Let L_1, L_2 be languages such that $L_1 \leq_T L_2$. If L_2 is undecidable, L_1 is ________ undecidable.

- ○ always
- [✓ **sometimes**]
- ○ never

Solution:

L_2 being undecidable tells nothing on the decidability of L_1. To show this, we offer 2 examples.

If L_1 is decidable, then its decider can call a black-box decider for L_2 and simply ignore its results, making it true that $L_1 \leq_T L_2$.

If it is true that $L_1 \leq_T L_2$, L_1 being undecidable means that L_2 is undecidable as well. Thus, L_1 can also be undecidable, and this relation would still be possible.

Because we have shown both cases where L_1 is decidable or undecidable, the answer is sometimes.

7. Consider the following algorithm for the 0-1 knapsack problem. Here n is the number of items, K is the total weight the knapsack can carry, and $V[i], W[i]$ are respectively the value and weight of the ith item. The output is the maximum total value of items that can be carried in the knapsack.

```
function KNAPSACK($n, K, V, W$)
    if $n == 0$ or $K == 0$ then return 0
    if $W[n] > K$ then return KNAPSACK($n-1, K, V, W$)
    else return max{ KNAPSACK($n-1, K, V, W$),
                     KNAPSACK($n-1, K-W[n], V, W$) + $V[n]$ }
```

What is the tightest bound on the worst-case running time of KNAPSACK?

- ○ $O(n)$
- ○ $O(n^2)$
- ○ $O(nK)$
- [✓ $O(2^n)$]
- ○ $O(2^K)$

Solution: Notice that this function does not solve the knapsack problem using dynamic programming—it doesn't use a table to save answers to subproblems—so its time complexity is not necessarily $O(nK)$. Instead, in the worst case, the function will make two recursive calls while decreasing n by 1, so it will compute every possible combination of items in V to see which one has the greatest total value without exceeding a weight of

K. Since there are n items in V, there are 2^n possible combination of items in V, so the worst case time complexity of this function is $O(2^n)$.

By way of illustration, consider that when this function starts with n elements in V, it can make 2 calls, each with $n-1$ elements in its respective input vector V. This continues until $n = 0$. In the worst case, each call makes 2 more calls to the function, which will create 2^n calls. Each call (ignoring the work done by subroutine calls) takes constant $O(1)$ work, resulting in a time complexity of $O(2^n)$.

8. Which of the following languages (over ASCII) are decidable?

   i The set of all legal names of this term's EECS 376 IAs.

   ii The set of all prime numbers.

   iii The set of all valid natural deduction proofs.

   ◯ only i

   ◯ only i and ii

   ◯ only ii and iii

   ◯ none of i, ii, or iii

   [✓ **all of i, ii, and iii**]

Solution: All are true. i) is finite and so decidable. ii) is decidable by a program that given an integer n, checks that $n \geq 2$ and that none of the numbers in the range $[2, n-1]$ evenly divide n. iii) is decidable by a program that given a proof, checks that each statement in the proof follows from the axioms, the inference rules, and the previous statements.

9. ~~Fill in the blank to make the following statement true and as strong as possible:~~

$$L \text{ is recognizable } \underline{\hspace{2cm}} \; L \text{ is decidable.}$$

   [✓ **If** ($\Leftarrow$)]

   ◯ Only if ($\Rightarrow$)

   ◯ If and only if ($\Leftrightarrow$)

   ◯ None of the above make the statement true

Solution: A

If L is decidable, then both L and $\overline{L}$ are recognizable, so "if" is correct. We have seen that L_{ACC} is recognizable but undecidable, so "only if" is not correct.

UNIQNAME (print): ______________________________

10. Which language is decidable? Assume M is a Turing Machine.

   - ✓ $\{\langle M \rangle \mid M$ **is a TM and** $\exists M'$ **such that** $\epsilon \notin (L(M) \cap L(M'))\}$
   - ○ $\{\langle M \rangle \mid M$ is a TM such that $|L(M)|$ is odd$\}$
   - ○ $\{\langle M \rangle \mid M$ is a TM such that $0^{376k} \in L(M),$ for all $k \geq 1\}$
   - ○ $\{\langle M \rangle \mid M$ is a TM such that $x \notin L(M)$ for some $|x| = 376\}$
   - ○ $\{\langle M \rangle \mid M$ is a TM and $\exists M'$ such that $L(M) \cup L(M') = \{376\}\}$

Solution: The language $L_{inc} = \{\langle M \rangle |$ there exists a TM M' such that the empty string $\epsilon \notin L(M) \cap L(M')\}$ includes all languages (if $M' = \emptyset$ for example) and all machines have that property and so the language is trivially decidable.

UNIQNAME (print): ____________________

Shorter Answer – 30 points

1. (8) Briefly prove or disprove the following statement:
   If $L_1, L_2 \subseteq \{0,1\}^*$ are undecidable languages and $L_1 \neq L_2$, then their symmetric difference, $(L_1 \setminus L_2) \cup (L_2 \setminus L_1)$, is also an undecidable language.

   Solution: The statement is False. Consider $L_1 = L$ and $L_2 = \overline{L}$ where L is an undecidable language. Then their symmetric difference is $\{0,1\}^*$, which is decidable.

UNIQNAME (print): ____

2. (6) Alice and Bob are competing to write a new algorithm to sort an array of size n. After some complexity analysis, they find that Alice's algorithm has time complexity $\Theta(n^2)$, while Bob's algorithm has time complexity $\Theta(n \log n)$. Bob expects his algorithm to run faster, but real-world testing on some moderately large arrays shows that Alice's algorithm actually runs faster. Briefly (ideally in two sentences or less each) answer the following questions:

   (a) Explain why this could be the case.

   (b) Can you be sure there exist inputs for which Bob's algorithm will run faster than Alice's? Justify your answer.

Solution:

(a) Big-Θ allows us to disregard constant factors for performance in the long-run, but for smaller values of n, these constant factors might matter.

(b) As the input size grows at some point, Bob's algorithm will outperform Alice's.

UNIQNAME (print): ____________________

3. (6) A stay-put TM is an alternate model of TM. It is identical to the model of TM we learned in class (tape bounded on left, unbounded on right) except that the tape head is allowed to perform a “no-move.” Namely, on each transition, the tape head is allowed to move one spot left, one spot right, or stay in place. Briefly answer the following questions (at a high level; one sentence per part is fine).

   (a) Can a stay-put TM simulate a normal TM?

   Solution: Yes. A stay-put TM can simulate a normal TM by never using the “no-move.”

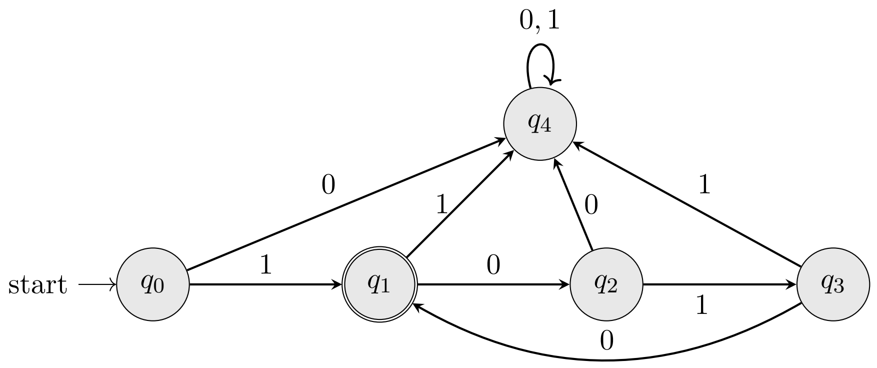
   (b) Can a normal TM simulate a stay-put TM? If your answer is “yes,” you only need to address how a normal TM would simulate writing a character and performing a “no-move.” If you answer “no,” you only need to provide a brief justification.

   Solution: Yes. Whenever a stay-put TM would write a character and perform a “no move,” a normal TM could perform that same write, then move one spot right, not modify the cell it just moved to (i.e. write the character that is already in the cell), and move one spot left.

   Note that moving left and then moving right is not a valid solution because this would not work if we were in the left-most cell of the tape.

UNIQNAME (print): ______________________________

4. (10) All DFAs in this problem are over the alphabet $\Sigma = \{0, 1\}$.

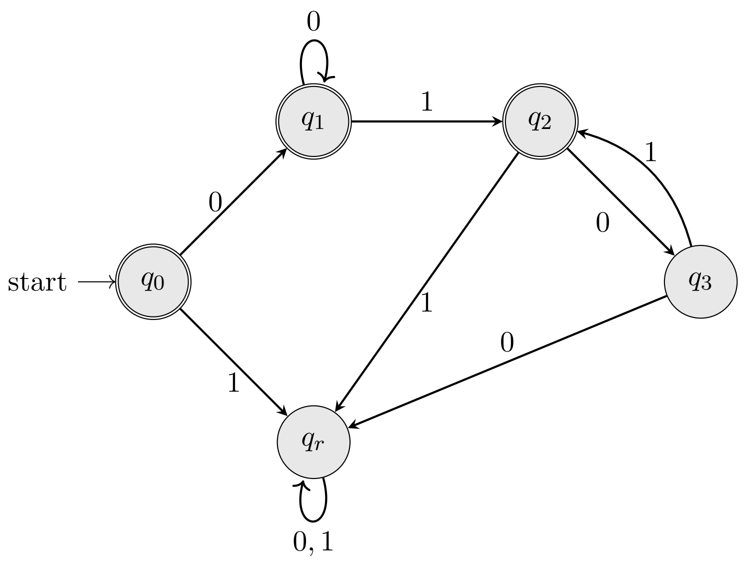


(a) What is the language of the above DFA?

Solution: Regular expression: $1(010)^*$

(b) Draw a DFA whose language is the set of all binary strings that begin with zero or more 0's followed by zero or more 01's (so $0^*(01)^*$). You should use 5 or fewer states.

Solution:



UNIQNAME (print): ____________________

Proofs and longer questions – 40 points

Answer two of the following three questions. **Clearly cross out the question you do not want graded.** If it isn't clear what problem you don't want graded, we will grade the first two. Each of the two questions are worth 20 points.

1. The **2-Arithmetic Subsequence Problem** is defined as follows: Given a sequence of n positive integers $a_1, \ldots, a_n$, find the length of the longest subsequence where each value in the subsequence is increasing by 2. For example, in the sequence $6, 1, 2, 3, 12, 4, 5, 6$, the length of the longest increasing-by-two subsequence is 3 and is achieved by the subsequence $2, 4, 6$ (also by $1, 3, 5$).

   (a) Find a recurrence relation that can be used to solve the 2ASP.

   (b) Write pseudocode which solves the problem using dynamic programming. It should have a run time that is $O(n^2)$.

Solution:

Let $L(i)$ be the length of the longest increasing-by-2 subsequence that ends at a_i. Then we may define the recurrence $L(i) = 1 + \max(\{L(j) : 1 \le j < i \text{ and } a_j = a_i - 2\} \cup \{0\})$. In other words, we compute the length of the longest such subsequence ending at each $a_j = a_i - 2$ (so that we can extend it with a_i). We may then compute the length of the longest such subsequence as $\max\{L(i) : 1 \le i \le n\}$.

The pseudocode is as follows:

```
(a) Initialize $L(i) = 0$ for each $1 \le i \le n$
(b) For $i = 1..n$: $L(i) = 1 + \max(\{L(j) : 1 \le j < i \text{ and } a_j = a_i - 2\} \cup \{0\})$
(c) Return $\max\{L(i) : 1 \le i \le n\}$
```

We observe that the max in (b) may be computed in $O(n)$ time. (Similarly for the max in (c).) Hence, the total running time of step (b) is $O(n^2)$. The total running time is therefore $O(n^2)$.

UNIQNAME (print): ________________________________

2. Let $L_{bob} = \{\langle M \rangle \mid \langle \text{“bob”} \rangle \notin L(M)\}$. Show that $L_{ACC} \leq_T L_{bob}$ or show that $L_{HALT} \leq_T L_{bob}$. (Do whichever of the two you would prefer.)

Solution: $L_{ACC} \leq_T L_{bob}$: Letting E be a ”black box” that decides L_{bob}, we construct a program D that uses E to decide L_{ACC}:

```
$D(\langle M \rangle, x)$ :
    construct a machine $M'$ = “on input $w$:
        ignore $w$, run $M$ on $x$
        If $M$ accepts $x$, accept. Otherwise, reject.”
    call $E(\langle M' \rangle)$
    if $E$ accepts, then reject
    if $E$ rejects, then accept
```

Analysis: First, we verify that D accepts all strings in the language L_{ACC}:

$$\begin{aligned}(\langle M \rangle, x) \in L_{ACC} &\implies M \text{ accepts } x\\ &\implies \text{for all } w \in \Sigma^*,\ M' \text{ accepts } w\\ &\implies L(M') = \Sigma^*,\ \text{which includes } \langle \text{“bob”} \rangle\\ &\implies E \text{ rejects } \langle M' \rangle\\ &\implies D \text{ accepts } (\langle M \rangle, x).\end{aligned}$$

Next, we verify that D rejects all strings not in the language L_{ACC}:

$$\begin{aligned}(\langle M \rangle, x) \notin L_{ACC} &\implies M \text{ either loops on or rejects } x\\ &\implies \text{for all } w \in \Sigma^*,\ M' \text{ either loops on or rejects } w\\ &\implies L(M') = \emptyset,\ \text{which doesn't include } \langle \text{“bob”} \rangle\\ &\implies E \text{ accepts } \langle M' \rangle\\ &\implies D \text{ rejects } (\langle M \rangle, x).\end{aligned}$$

Therefore, D is a decider for L_{ACC}, which implies that $L_{ACC} \leq_{\mathrm{T}} L_{bob}$.

$L_{HALT} \leq_T L_{bob}$: Letting E be a ”black box” that decides L_{bob}, we construct a program D that uses E to decide L_{HALT}:

```
$D(\langle M \rangle, x)$ :
    construct a machine $M'$ = “on input $w$:
        run $M$ on $x$, and then accept.”
    call $E(\langle M' \rangle)$
    if $E$ accepts, then reject
    if $E$ rejects, then accept
```

<u>Analysis:</u> First, we verify that D accepts all strings in the language L_{HALT}:

$$\begin{aligned}(\langle M\rangle, x) \in L_{HALT} &\implies M \text{ halts on } x \text{ in a finite number of steps}\\ &\implies \text{for all } w \in \Sigma^*,\ M' \text{ accepts } w\\ &\implies L(M') = \Sigma^*, \text{ which includes } \langle\text{“bob”}\rangle\\ &\implies E \text{ rejects } \langle M'\rangle\\ &\implies D \text{ accepts } (\langle M\rangle, x).\end{aligned}$$

Next, we verify that D rejects all strings not in the language L_{HALT}:

$$\begin{aligned}(\langle M\rangle, x) \notin L_{HALT} &\implies M \text{ loops on } x\\ &\implies \text{for all } w \in \Sigma^*,\ M' \text{ loops on } w\\ &\implies L(M') = \emptyset, \text{ which doesn't include } \langle\text{“bob”}\rangle\\ &\implies E \text{ accepts } \langle M'\rangle\\ &\implies D \text{ rejects } (\langle M\rangle, x).\end{aligned}$$

Therefore, D is a decider for L_{HALT}, which implies that $L_{HALT} \leq_{\mathrm{T}} L_{bob}$.

UNIQNAME (print): ______________________________

3. Consider the following two-player game on a 12×12 grid. At the beginning of the game, there is at most one stone occupying each square (so each square either has 1 or 0 stones). On their turn, each player must pick a stone and remove it. Subsequent to removing the stone, the player can place or remove stones as desired, on any of the squares to the **right** of that stone (on the same row). No square may ever have more than one stone. The game ends when there are no more stones left on the board.

   Are there any initial board states **and** sets of moves that can make the game continue indefinitely, or will the game always end **regardless** of the initial position and the moves of the players? If so, describe them. If not, provide a proof that demonstrates this.

   Hint: Try simulating the game on a 4×4 grid.

Solution: Solution 1 (Potential Method): At any given game state, we can assign a number to each row on the board by considering the row to be a 12-bit number where empty squares represent 0-bits and occupied squares represent 1-bits (most significant bit on the left, least on right). Let our potential function be the sum of these 12 numbers (one per row). Note that our potential function can never be less than zero, yet each move will strictly decrease the number associated with a single row, thus strictly decreasing our potential function. The game always ends in at most $12(2^{12}-1)$ moves, so it must always terminate.

Solution 2 (Induction): We'll just prove the game always terminates on a 1×12 grid, as the different rows don't affect each other. To do this, we prove that the game always terminates on a $1 \times k$ grid whenever $k \geq 1$ by induction on k:

- $k = 1$: If there are no stones, the game ends. If there's one stone, we take it and the game ends.
- $k \geq 2$: If the leftmost cell has no stone, then we cannot change it (nothing to its left). That is, we'd effectively be playing on a $1 \times (k-1)$ grid and terminate by inductive hypothesis. Similarly, if the leftmost cell has a stone, then we must eventually take it (else we'd never change the cell and terminate by inductive hypothesis), and if we do take it, the leftmost cell loses its stone and the game terminates as before.

  (In the above paragraph, "terminates" means "eventually terminates," not "immediately terminates.")

Solution 3 (Contradiction/Infinite Dependency Chain): Suppose it was somehow possible for the game to go forever. We'd have to visit some board state S more than once. To leave S, we remove a stone from some location ℓ_0. To get the board back into state S, we must repopulate ℓ_0. So in some future move, we must remove a stone from some location ℓ_1^* left of ℓ_0. There are 2 possibilities for the stone we remove at ℓ_1^*:

- That stone was there when we were in state S.

- That stone was not there when we were in state S. Then state S must have contained a stone at some location ℓ_1 left of ℓ_1^* which we eventually removed to populate ℓ_1^*. This is because the only way to populate ℓ_1^* would be to remove a stone to its left.

Either way, if we remove a stone at location ℓ_0, where ℓ_0 contained a stone in state S, then we must remove a stone from some location ℓ_1 left of ℓ_0, where ℓ_1 contained a stone in state S. We can repeat the above logic arbitrarily many times, contradicting the fact that the grid has finitely many stones.

UNIQNAME (print): ______________________________

This page intentionally left blank.

If you have answers on this page be certain you write "answer continues on page 12" under the actual question.