Multiple-choice (70 pt.)

1. **(8 pt.)** Which of the options correctly describe each of the following quantities?

   (a) **(2 pt.)** The function $f(n)$, where $f(n) = n\log^2(n)$. Note that the bounds do **not** necessarily need to be tight. **Circle all that apply.**

   (A) $O(n^2)$ (B) $\Theta(n^2)$ (C) $\Omega(n\log\log(n))$ (D) $O(n)$ (E) $O(\log^2(n))$.

   (b) **(2 pt.)** $T(n)$ given by $T(n) = T(\lfloor n/5 \rfloor) + T(\lfloor 7n/10 \rfloor) + \Theta(n)$ with $T(n) = 1$ for $n \leq 5$. Note that the bounds do **not** necessarily need to be tight. **Circle all that apply.**

   (A) $\Omega(n)$ (B) $O(n)$ (C) $O(n\log(n))$ (D) $\Omega(n^2)$ (E) $O(n^2)$.

   (c) **(2 pt.)** Consider the following divide-and-conquer algorithm.

   ```
   def mystery (int m, int n):
     if m <= 0 or n <= 0:
       return None
     if m == n:
       return m
     else if m > n:
       return mystery(m-n, n)
     else:
       return mystery(m, n-m)
   ```

   What does the `mystery` compute for positive integers m and n?
   Note that `mod` represents the modulo operator (i.e. `13 mod 5` is 3) and `gcd` represents the greatest common divisor (i.e. `gcd(24, 36)` is 12).

   (A) `ceil(m/n)` (B) `max(m-n,n-m)` (C) `m mod n` (D) `gcd(m,n)` (E) `m*n`.

   (d) **(2 pt.)** $T(n)$, which is the runtime of `mystery`. Your bound should be tight.

   (A) $O(m+n)$ (B) $\Theta(mn)$ (C) $O(m\log(n))$ (D) $O(n)$ (E) $O(\min\{m,n\})$.

2. **(4 pt.)** Suppose you want to alphabetize the following list of three letters words in ascending a-z alphabetical order with radix sort: `[dog, lot, caw, log, dot]`. Recall that radix sort calls bucket sort as a subroutine.

   (a) **(2 pt.)** After the first call to bucket sort, what is the order of the list?

   (A) `[dog, log, dot, lot, caw]`.
   (B) `[caw, dog, dot, log, lot]`.
   (C) `[caw, dog, dot, lot, log]`.
   (D) `[dog, log, lot, dot, caw]`.

   (b) **(2 pt.)** After the second call to bucket sort, what is the order of the list?

   (A) `[caw, dog, log, dot, lot]`.
   (B) `[caw, dog, dot, log, lot]`.
   (C) `[caw, dog, dot, lot, log]`.
   (D) `[caw, dog, log, lot, dot]`.

3. **(4 pt.)** Let H be a universal family of hash functions mapping a universe of keys U to one of n values $\{1, 2, \dots, \text{n}\}$. For some $x, y \in U$ such that $x \neq y$, the number of hash functions $h \in H$ for which $h(x) = h(y)$ is at most:

$$(A)\ |U|/n \qquad (B)\ n/|H| \qquad (C)\ n/|U| \qquad (D)\ 1/n \qquad (E)\ |H|/n.$$

4. **(4 pt.)** Consider a binary tree $T = (V, E)$ where $|v|$ denotes the number of vertices in the subtree rooted at vertex v. Let $n = |T.\text{root}|$.

   Suppose vertices of T satisfy the invariant that for $v \in V$, $|v.\text{left}| = \lfloor |v.\text{right}|/2 \rfloor$ when $|v| \geq 4$ and $|v.\text{right}| = |v| - 1$ otherwise i.e. there are about twice as many vertices in the subtree rooted at v's right child than the subtree rooted at its left child; if there aren't enough vertices left, then the right child receives the remaining vertices. Which of the following bounds the number of edges in a path from the root of T to any leaf? **Circle all that apply.**

$$(A)\ \Omega(\log_3(n)) \quad (B)\ O(\log_{3/2}(n)) \quad (C)\ \Omega(\log^{3/2}(n)) \quad (D)\ O(\log^2(n)) \quad (E)\ \Omega(n\log(n))$$

5. **(4 pt.)** Let `monty` be some Monte Carlo algorithm which gives the correct solution for a problem with probability at p_1, independent of the input. How many independent executions of `monty` suffice to increase the probability of obtaining a correct solution to at least p_2? Assume $0 < p_1 < p_2 < 1$.

$$(A)\ \frac{\log(1-p_2)}{\log(1-p_1)} \qquad (B)\ \frac{\log(1-p_1)}{\log(1-p_2)} \qquad (C)\ \frac{\log(p_1)}{\log(p_2)} \qquad (D)\ \frac{\log(p_2)}{\log(p_1)} \qquad (E)\ \frac{1-p_1}{1-p_2}.$$

6. **(8 pt.)** Let U_k be the universe of all strings consisting of k numeric digits. (`0000`, `0123`, and `0101` are part of universe U_4 but `000a`, `012`, and `0!0!` are not.) Let u_i denote the ith digit of a string $u \in U_k$ where $0 \leq i < k$, so u_0 is `0` and u_1 is `2` for string `0246`.

   Let H_k be a family of k hash functions mapping universe U_k to values $\{0, 1, 2, \ldots, 9\}$ where $h_0 \in H_k$ hashes all strings according to their first digit. (For all strings u where $u_0 = 0$, $h_0(u) = 0$; for all strings u where $u_0 = 1$, $h_0(u) = 1$; for all strings u where $u_0 = 9$, $h_0(u) = 9$.) Likewise, $h_1 \in H_k$ hashes all strings according to their second digit. Generally, for all strings $u \in U_k$ where $u_i = x$, $h_i(u) = x$ for $0 \leq i < k$.

   (a) **(4 pt.)** Which of the following statements that are true? **Circle all that apply.**

   (A) H_1 is universal for the entire universe U_1.
   (B) H_2 is universal for the entire universe U_2.
   (C) H_3 is universal for the entire universe U_3.
   (D) H_4 is universal for the entire universe U_4.

   (b) **(4 pt.)** H_5 is not universal for the entire universe U_5, but there exist subsets of U_5 for which it is universal. Which of the following subsets of U_5 is H_5 universal? **Circle all that apply.**

   (A) {`00000`, `11111`, `22222`, `33333`, `44444`}.
   (B) {`00000`, `01234`}.
   (C) {`00000`, `11111`, `01234`}.
   (D) {`00000`, `11111`, `22222`, `33333`, `44444`, `01234`}.

7. **(4 pt.)** Which of the following correctly characterizes the advantage of partitioning around a random pivot selected from the sublist in `quicksort`, as opposed to partitioning around the first element of the sublist?

   (A) It reduces the expected runtime over all of the inputs.
   (B) It reduces the expected runtime for best-case inputs.
   (C) It reduces the expected runtime for worst-case inputs.
   (D) It reduces the worst-case runtime over all of the inputs.

8. **(8 pt.)** Dijkstra's algorithm solves the single-source shortes path problem in weighted graphs with non-negative edge-weights.

   (a) **(4 pt.)** Consider a specific graph $G_1 = (V, E)$ in which $w(e) = 1$ for all edges $e \in E$. Circle all of the following options that occur in the same order that Dijkstra's marks vertices as "done".

   Assume deterministic tie-breaking i.e. Suppose all vertices are labeled with unique identifiers and if multiple vertices share the same distance from s, then Dijkstra's dequeues them in lexicographic order. Likewise, BFS and DFS resolve ties between neighbors using this same lexicographic order.

   (A) BFS marks vertices as "visited".

   (B) DFS marks vertices as "in progress".

   (C) DFS marks vertices as "done".

   (D) None of the above.

   (b) **(2 pt.)** Consider a weighted directed graph $G = (V, E)$ with non-negative edge-weights i.e. $w_G(u, v) \geq 0$ for all edges $(u, v) \in E$. We construct a new graph $G' = (V, E')$ from G such that for all edges $(u, v) \in E$, we introduce two new vertices a' and b' where $(u, a'), (a', b'), (b', v) \in E'$ and

$$w_{G'}(u, a') = -1 \qquad w_{G'}(a', b') = w_G(u, v) \qquad w_{G'}(b', v) = 1$$

   Which of the following is the largest set of graphs G for which Dijkstra's algorithm works correctly for all corresponding G'?

   (A) All graphs with non-negative edge-weights.

   (B) A non-empty strict subset of all graphs with non-negative edge-weights.

   (C) No graphs with non-negative edge-weights.

   (c) **(2 pt.)** Same question as part (b) except

$$w_{G'}(u, a') = 1 \qquad w_{G'}(a', b') = w_G(u, v) \qquad w_{G'}(b', v) = 1$$

   (A) All graphs with non-negative edge-weights.

   (B) A non-empty strict subset of all graphs with non-negative edge-weights.

   (C) No graphs with non-negative edge-weights.

9. **(4 pt.)** In a parallel universe, our amazing CS 161 TAs decide they want to give homework hints to ONE student instead of answering Piazza questions or holding office hours. They plan to pick this student carefully, such that this student will tell their study group, and members of this study group will tell other study groups until the ENTIRE class hears about the hint.

   Suppose the TAs model this hint-flow network as a directed graph $G = (V, E)$ where V is the set of all CS 161 students and E contains a directed edge with weight 1 from student u to student v if and only if u conveys homework hints to v.

   From the perspective of the TAs, the students can be divided into equivalence classes, where `Alice = Bob` if and only if the set of students that would eventually hear a hint told to Alice is identical to the set of students that would eventually hear a hint told to Bob.

   How can the TAs compute the decomposition of students into these equivalence classes?

   (A) Compute the weakly connected components (WCC) of G with BFS and assign students in the same WCC to the same equivalence class.

   (B) Compute the strongly connected components (SCC) of G with Kosaraju's algorithm and assign students in the same SCC to the same equivalence class.

   (C) Compute the global minimum cut of G with Karger's algorithm and assign vertices in the same super-vertex to the same equivalence class.

10. **(4 pt.)** Consider the following two algorithms for computing the nth Fibonacci number:

```
def fibo (int n):
  if n < 2:
    return 1
  return fibo(n-1) + fibo(n-2)

cache = {}
def cached_fibo (int n):
  if n < 2:
    return 1
  else if n is in cache:
    return cache[n]
  else:
    result = cached_fibo(n-1) + cached_fibo(n-2)
    cache[n] = result
    return result
```

(a) **(2 pt.)** How many times is `fibo(2)` called on a single call to `fibo(6)`?

(A) 1 (B) 2 (C) 3 (D) 4 (E) 5

(b) **(2 pt.)** How many times is `cached_fibo(2)` called on a single call to `cached_fibo(6)`, assuming an initially empty cache?

(A) 1 (B) 2 (C) 3 (D) 4 (E) 5

11. **(4 pt.)** Given a connected, undirected graph $G = (V, E)$ in which each edge $e \in E$ has positive edge weight $w(e)$, suppose we want to compute a spanning tree $T = (V, E_T)$ such that the **product** of all its edge weights is maximized i.e. the spanning tree T maximizing $\prod_{e \in E_T} w(e)$.

Suppose we have a black box algorithm $\mathcal{A}$ that solves the MST problem. How can you use $\mathcal{A}$ to solve this new problem, using only $O(|E|)$ additional time (that is, not including the call to $\mathcal{A}$)?

(A) Construct G' from G such that $w_{G'}(e) = \log(w_G(e))$ and run $\mathcal{A}$ on G'.

(B) Construct G' from G such that $w_{G'}(e) = -\log(w_G(e))$ and run $\mathcal{A}$ on G'.

(C) Construct G' from G such that $w_{G'}(e) = 1/w_G(e)$ and run $\mathcal{A}$ on G'.

(D) Run $\mathcal{A}$ on G.

12. **(4 pt.)** Over the last few years, you have kept a piggy bank of coins, waiting to cash them in for single dollar bills. You notice you have pennies, nickels, dimes, quarters, and half-dollar coins in the jar.

    As an over-achieving CS 161 student, you want to design a dynamic programming algorithm to count the number of ways W that your coins add to 100 cents.

    Assuming you have an unlimited number of each coin, what is the correct expression for the recursive case of the the recurrence $W(n, C)$, where C is a list containing the values of the coins (here, `[1, 5, 10, 25, 50]`) and n is the total cost of the coins that you're aiming to count?

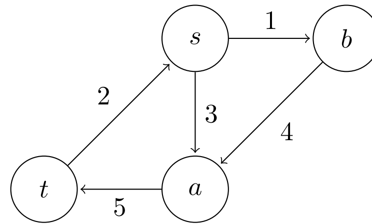
    (A) $W(n, C) = \begin{cases} \max\{W(n - C[0], C), W(n, C)\} & \text{if } n > 0 \text{ and } |C| > 0 \\ 1 & \text{if } n = 0 \\ 0 & \text{if } n < 0 \text{ or } |C| = 0 \end{cases}$

    (B) $W(n, C) = \begin{cases} \max\{W(n - C[0], C), W(n, C[1:])\} & \text{if } n > 0 \text{ and } |C| > 0 \\ 1 & \text{if } n = 0 \\ 0 & \text{if } n < 0 \text{ or } |C| = 0 \end{cases}$

    (C) $W(n, C) = \begin{cases} W(n - C[0], C) + W(n, C) & \text{if } n > 0 \text{ and } |C| > 0 \\ 1 & \text{if } n = 0 \\ 0 & \text{if } n < 0 \text{ or } |C| = 0 \end{cases}$

    (D) $W(n, C) = \begin{cases} W(n - C[0], C) + W(n, C[1:]) & \text{if } n > 0 \text{ and } |C| > 0 \\ 1 & \text{if } n = 0 \\ 0 & \text{if } n < 0 \text{ or } |C| = 0 \end{cases}$

13. **(4 pt.)** Consider the following graph.



    (a) **(2 pt.)** What is the value of the maximum $s - t$ flow?

    (A) 1 (B) 2 (C) 3 (D) 4 (E) 5

    (b) **(2 pt.)** What is the cost of the minimum $s - t$ cut?

    (A) 1 (B) 2 (C) 3 (D) 4 (E) 5

14. **(6 pt.)** Recall the following definitions:

- **P** is a complexity class that represents the set of all decision problems that can be solved in polynomial time.
- **NP** is a complexity class that represents the set of all decision problems for which the instances where the answer is "yes" have proofs that can be verified in polynomial time.
- **NP-complete** is a complexity class that represents the set of all problems in $x \in \mathbf{NP}$ for which it's possible to reduce any other problem $y \in \mathbf{NP}$ to x.
- **NP-hard** is a complexity class that represents the set of problems that are at least as hard as problems in **NP-complete**.

(a) **(4 pt.)** Circle all of the following problems that are in **NP**.

(A) Given a connected graph G, can its vertices be colored using two colors such that no edge connects two vertices of the same color?

(B) Given an undirected graph G and a cost function on the vertices, find a minimum cost vertex cover.

(C) Given a universe U of n elements, a collection of subsets $S = \{S_1, \ldots, S_k\}$ of U each of which has a cost, find a minimum cost subcollection of S that covers all elements of U.

(D) Given a list of elements A, is it sorted?

(E) Given a list of elements A, is 5 its kth-order statistic?

(b) **(2 pt.)** Which of the following algorithmic design paradigms did we employ to solve **NP-hard** problems vertex cover and set cover?

(A) Divide-and-conquer.

(B) Randomized.

(C) Greedy.

(D) Dynamic programming.

Short-answers (30 pt.)

---

1. **(6 pt.)** What are the high-level similarities and differences between the divide-and-conquer, dynamic programming, and greedy algorithmic paradigms?

   [**We are expecting: A similarity shared by all three algorithmic paradigms and a difference between all three algorithmic paradigms.**]

2. **(6 pt.)** Why does the $\Omega(n \log(n))$-time lower bound not contradict the existence of the $O(n)$-time bucket sort?

   [**We are expecting: A 1-2 sentence answer to the question.**]

3. **(6 pt.)** A friend suggests that you can use Dijkstra's algorithm on graphs with negative edge weights (provided there are no negative-weight cycles) by adding a constant value to all weights in the graph to make them non-negative. Why doesn't this work?

   [**We are expecting: A counterexample graph in which your friend's approach will still produce an incorrect path and 1-2 explanation of why it fails.**]

4. **(6 pt.)** Breadth-first search, Dijkstra's algorithm, the Bellman-Ford algorithm, and the Floyd-Warshall algorithm can all be used to find shortest paths in a graph. **Draw a line from each question to the *best* answer to that question.**

Question	Answer
When might you prefer breadth-first search to Dijkstra's algorithm?	When the graph has negative edge weights.
When might you prefer Floyd-Warshall to Bellman-Ford?	When the graph is unweighted.
When might you prefer Bellman-Ford to Dijkstra's algorithm?	When you want to find the shortest paths between all pairs of vertices.
	When you want to find the shortest paths from a specific vertex s to any other vertex t.

5. **(6 pt.)** Consider a directed, weighted, graph $G = \{V, E\}$, and two special vertices, $s, t \in V$. Let f be an s-t flow for the graph (i.e. the amount of flow that f sends on each edge is at most the weight of the corresponding edge, and flow is conserved at every vertex other than s and t), and let the sets S and $T = V \setminus S$ partition the set of vertices, with $s \in S$ and $t \in T$. What can you say about the relationship between the *value* of the flow f, and the *value* of the cut S, T (where the value of the cut is defined to be the sum of the weights of the edges that go from a node in S to a node in T)? What can you say about the relationship between the value of the flow that *maximizes* the sum of the flows leaving s, and the value of the cut $S_{\min}, T_{\min}$ that *minimizes* the value of the cut?

[We are expecting: Answers to both questions.]

Problems (50 pt.)

1. **(Zig-Zags) (20 pt.)**

   A sequence of numbers is called a zig-zag sequence if the differences between successive numbers strictly alternate between positive and negative. In other words, in order for a sequence $[x_1, x_2, \ldots, x_n]$ to be an zig-zag sequence, then its elements must satisfy one of the following relationships: $x_1 < x_2 > x_3 < x_4 \ldots$ or $x_1 > x_2 < x_3 > x_4 \ldots$.

   The function `find_zig_zag_length(A)` takes as input a list of positive integers A and returns the length of the longest zig-zag subsequence.

   The length of the longest zig-zag subsequence of `[1, 10, 15, 5, 30, 25, 20]` is 5, given by the sublist `[1, 10, 5, 30, 25]`.

   (a) **(8 pt.)** Give the recursive formulation for `find_zig_zag_length`. If you can't specify the recursive formulation, we'll award partial credit to English descriptions of the optimal substructure. (Hint: Your table should be A.length $\times$ 2.)

   [**We are expecting: A recursive formulation or, for partial credit, an English description of the optimal substructure.**]

(b) **(8 pt.)** Design a **dynamic programming algorithm** that computes the **length** of the longest zig-zag subsequence.

[We are expecting: Pseudocode.]

(c) **(4 pt.)** Describe an approach for retrieving the list of indices describing the longest zig-zag subsequence, given a dynamic programming algorithm that produces the length of the longest zig-zag subsequence. (We discussed two approaches for accomplishing this task; either one will do here.)

[We are expecting: One to two sentences with your approach.]

2. **(Prioritizing) (30 pt.)**

Suppose you have a to-do list of n tasks with their deadlines that must be performed individually (you can't multi-task).

(a) **(6 pt.)** Assume all tasks require one unit of time to complete and share the same importance.

i	Task	Deadline
0	Watch Game of Thrones	3
1	Mow the lawn	1
2	Adopt Koko the Koala	2.5
3	Feed baby breakfast	2

In this (hopefully) theoretical to-do list, you could accomplish tasks 0, 1, and 3 by their respective deadlines by first doing task 1, then doing task 3, then doing task 0. Unfortunately, you cannot always complete all n tasks, as in this case.

Design an $O(n \log(n))$-time **greedy algorithm** for `prioritize`, which takes as input a list of deadlines $\{d_0, d_1, \ldots, d_{n-1}\}$ and returns the list of tasks that maximizes the number completed before their deadlines.

[**We are expecting: Pseudocode.**]

(b) **(8 pt.)** Prove that your algorithm is correct—that is, `prioritize` returns a feasible (all returned tasks can be performed individually by their deadlines) and optimal (there cannot be a longer feasible list of tasks) solution.

(i) **(2 pt.) Inductive hypothesis**

(ii) **(1 pt.) Base case**

(iii) **(4 pt.) Inductive step**

(iv) **(1 pt.) Conclusion**

(c) **(6 pt.)** All tasks still require one unit of time to complete, but now assume tasks have different values.

i	Task	Deadline	Value
0	Watch Game of Thrones	3	1
1	Mow the lawn	1	1
2	Adopt Koko the Koala	2.5	1.5
3	Feed baby breakfast	2	1

In this to-do list, it is now optimal to complete tasks 0, 1, and 2 (we get to adopt Koko!). We can complete task 1 by its deadline at $t = 1 \leq 1 = d_1$, then complete task 2 by its deadline at $t = 2 \leq 2.5 = d_2$, then complete task 0 by its deadline at $t = 3 \leq 3 = d_0$.

Design an $O(n^2)$-time **greedy algorithm** for `weighted_prioritize`, which takes as input a list of deadlines $\{d_0, d_1, \ldots, d_{n-1}\}$ and values $\{v_0, v_1, \ldots, v_{n-1}\}$ and returns the list of tasks that maximize the value of the tasks completed before their deadlines.

[**We are expecting: Pseudocode.**]

(d) **(10 pt.)** Now tasks might require different units of time to complete.

i	Task	Deadline	Value	Time Required
1	Watch Game of Thrones	3	1	2
2	Mow the lawn	1	1	1
3	Adopt Koko the Koala	2.5	1.5	1
4	Feed baby breakfast	2	1	0.5

In this to-do list, it is optimal to complete tasks 2, 3, and 4. We can complete task 2 by its deadline at $t = 1 \leq 1 = d_2$, then complete task 4 before its deadline at $t = 1.5 \leq 2 = d_4$, then complete task 3 by its deadline at $t = 2.5 \leq 2.5 = d_3$.

The function `timed_prioritize` and returns the list of tasks that maximize the value of the tasks completed before their deadlines. The function `timed_prioritize_value` takes the same inputs but returns the maximum value of tasks completed before their deadlines.

Design a **dynamic programming algorithm** that takes as input a list of deadlines, values, and times $\{t_1, t_2, \ldots, t_n\}$ and computes the **value** of the optimal list of tasks that maximizes the value of the tasks completed before their deadlines.

[**We are expecting: An English description or pseudocode of your algorithm.**]