COS 226 **Algorithms and Data Structures** **Fall 2024**

Final

This exam has 12 questions worth a total of 100 points. You have 180 minutes.

Instructions. This exam is preprocessed by computer. Write neatly, legibly, and darkly. Put all answers (and nothing else) inside the designated spaces. *Fill in* bubbles and checkboxes completely: ● and ■. To change an answer, erase it completely and redo.

Resources. The exam is closed book, except that you are allowed to use a one page reference sheet (8.5-by-11 paper, both sides, in your own handwriting). No electronic devices are permitted.

Honor Code. This exam is governed by Princeton's Honor Code. Discussing the contents of this exam before the solutions are posted is a violation of the Honor Code.

Please complete the following information now.

Name:

NetID:

Exam room: ◯ McCosh 50 ◯ McCosh 60 ◯ Other

Precept:

P01	P02	P03	P05	P06	P07	P08	P09	P10
◯	◯	◯	◯	◯	◯	◯	◯	◯

"I pledge my honor that I will not violate the Honor Code during this examination."

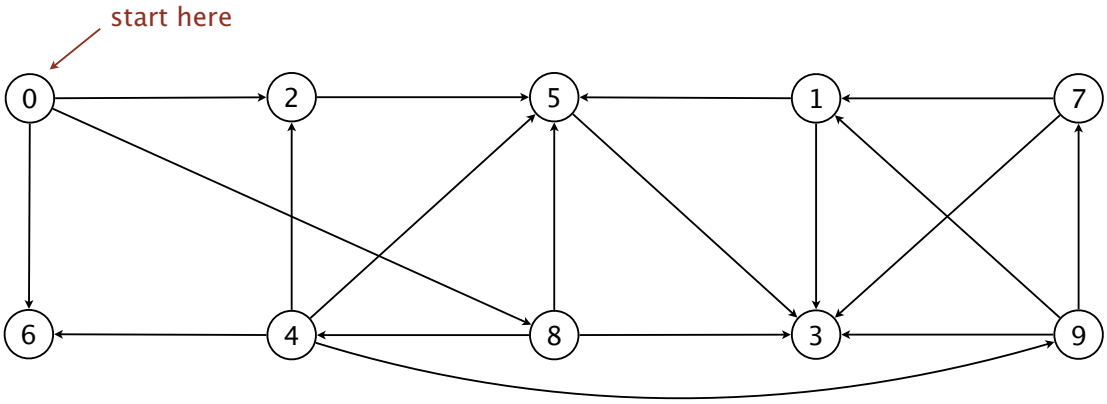
Signature

1. **Initialization. (1 point)**

   In the spaces provided on the front of the exam, write your name and NetID; fill in the bubble for your exam room and the precept in which you are officially registered; write and sign the Honor Code pledge.

2. **Graph search algorithms. (11 points)**

   Run *depth-first search* and *breadth-first search* on the following digraph, starting from vertex 0. Assume the adjacency lists are in sorted order: for example, when iterating over the edges leaving vertex 4, do so in the following order: 4→2, 4→5, 4→6 then 4→9.



   (a) List the 10 vertices in the order they are *removed from the queue* during the execution of BFS.

   0 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___

   (b) List the 10 vertices in *DFS preorder*.

   0 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___

(c) List the 10 vertices in *DFS postorder*.

0

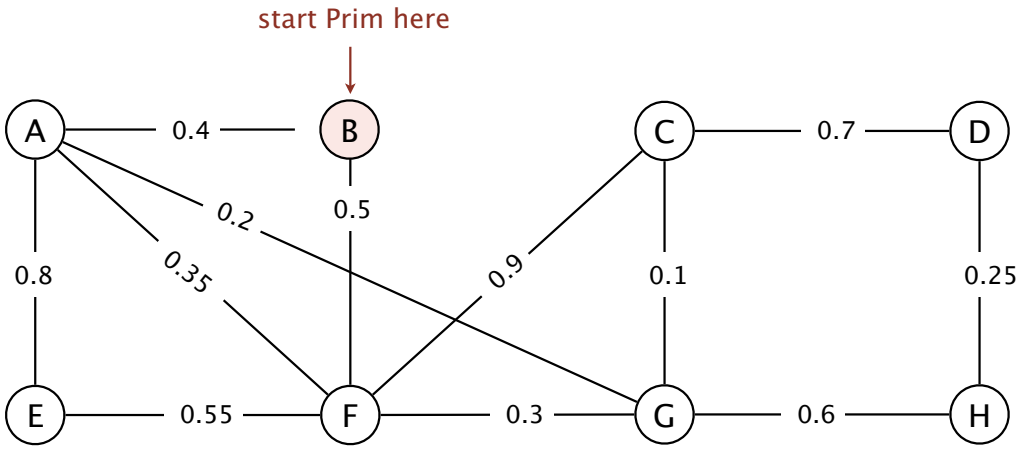
___ ___ ___ ___ ___ ___ ___ ___ ___ ___

(d) Is the *reverse* of the DFS postorder in part (c) a *topological order* for this digraph?

◯ *yes* ◯ *no*

3. **Minimum spanning trees and global mincut. (12 points)**

Consider the following edge-weighted graph.



(a) List the *weights* of the MST edges in the order that *Prim's algorithm* adds them to the MST. Start Prim's algorithm from vertex B.

(b) Run one execution of Karger's algorithm for finding a *global mincut* on the (unweighted version) of the graph from part (a). Assume that the uniformly random weights assigned to each edge in this execution of Karger's algorithm are the edge weights on that graph.

Which cut does this execution of Karger's algorithm output?

Mark all vertices that are on the same side of the cut as vertex B.

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
☐	■	☐	☐	☐	☐	☐	☐

(c) How many edges cross the cut output by this execution of Karger's algorithm?

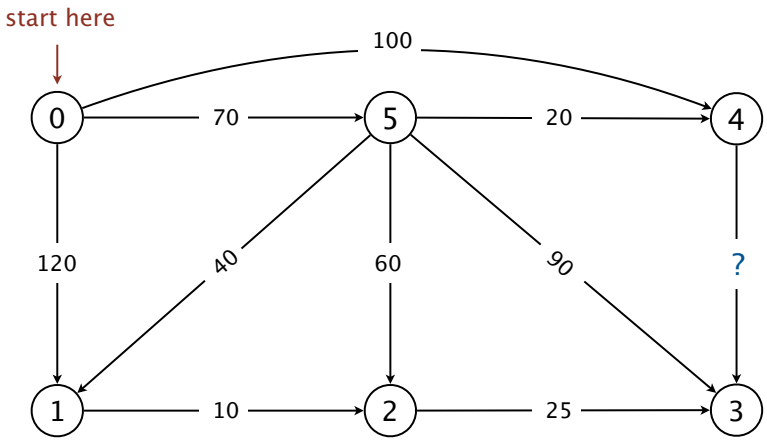
0	1	2	3	4	5	6
○	○	○	○	○	○	○

(d) Is the cut obtained by this execution of Karger's algorithm a *global mincut*?

yes	*no*
○	○

4. **Shortest paths. (15 points)**

Consider the following edge-weighted digraph G:



(a) During the execution of *Dijkstra's algorithm*, immediately after relaxing vertex 4, `distTo[3]` is 150. What is the weight of the edge $4 \to 3$?

○ 10 ○ 25 ○ 30 ○ 35 ○ 40 ○ 50 ○ 60 ○ 75 ○ 80

Use the weight of the edge $4 \to 3$ you found in this part of the question in all the other parts of the question.

(b) List the 6 vertices in the order they are *removed from the priority queue* during *Dijkstra's algorithm* with source vertex $s = 0$.

0 ____ ____ ____ ____ ____ ____

(c) Consider running the *Bellman–Ford algorithm* on G, with source vertex $s = 0$. Assume that, in each pass, the edges are relaxed in sorted order:

$$0\to1,\ 0\to4,\ 0\to5,\ 1\to2,\ 2\to3,\ 4\to3,\ 5\to1,\ 5\to2,\ 5\to3,\ 5\to4$$

Immediately *after the first pass*, what are the values of `distTo[v]` for each vertex v? If `distTo[v]` is infinite, write '∞'.

Write the values in the corresponding boxes.

0					
`distTo[0]`	`distTo[1]`	`distTo[2]`	`distTo[3]`	`distTo[4]`	`distTo[5]`

(d) Does the value of `distTo[v]` change for any vertex v in the *second pass* of the *Bellman–Ford* algorithm?

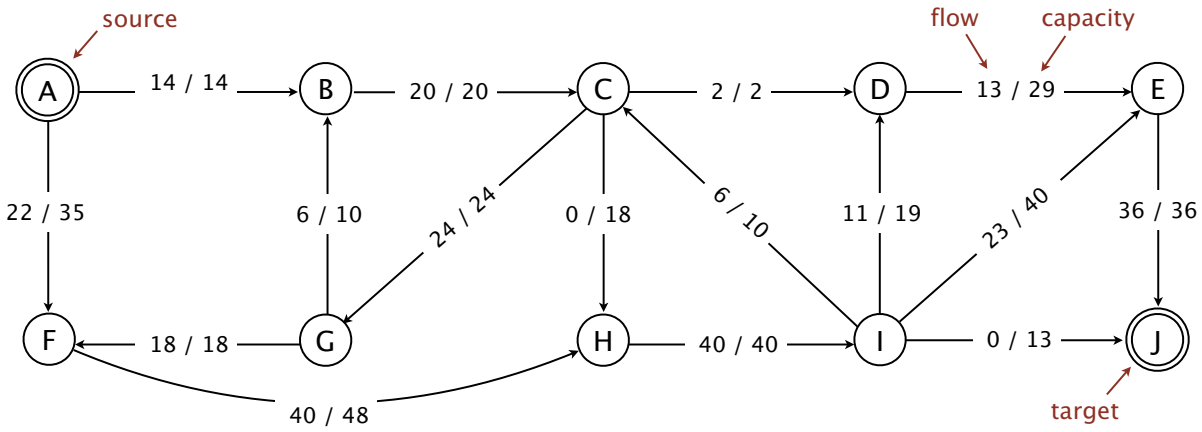
◯ *yes* ◯ *no*

(e) Does the value of `distTo[v]` change for any vertex v *in any pass* of the *Bellman–Ford* algorithm *after the second*?

◯ *yes* ◯ *no*

5. **Maxflows and mincuts. (15 points)**

Consider the following flow network and a *flow* f.



(a) What is the *value* of the flow f?

◯	◯	◯	◯	◯	◯	◯	◯	◯
14	22	23	30	32	36	42	44	48

(b) What is the *capacity* of the cut $\{A, B, C, D, E\}$?

◯	◯	◯	◯	◯	◯	◯	◯	◯
0	34	36	46	79	82	113	128	192

(c) Find an *augmenting path* with respect to f. Write the sequence of vertices in the path.

$A \rightarrow$

(d) What is the value of the *maximum flow*?

○	○	○	○	○	○	○	○	○
14	22	23	30	32	36	42	44	48

(e) Compute a *minimum A-J cut* in the graph. Which vertices are on the source side of your minimum cut? *Mark all that apply.*

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*
■	□	□	□	□	□	□	□	□

(f) If the *capacity* of edge $I \rightarrow C$ were 16 (instead of 10), what would the value of the *maximum flow* be?

○	○	○	○	○	○	○	○	○
14	22	23	30	32	36	42	44	48

(g) If the *capacity* of the edge $H \rightarrow I$ were 34 (instead of 40), what would the value of the *maximum flow* be?

○	○	○	○	○	○	○	○	○
14	22	23	30	32	36	42	44	48

6. **Properties of graph algorithms (5 points).**

Recall that in this course we assume all graphs and digraphs are simple, meaning they contain no self-loops or parallel edges.

Identify each statement as either always true or sometimes/always false.

true	*false*	
◯	◯	Every undirected graph with V vertices and at least V edges has a cycle.
◯	◯	Suppose we run DFS on an undirected graph that has a path between u and v. If u is marked before v, then the recursive call `dfs(v)` returns before the recursive call `dfs(u)` does.
◯	◯	Let G_1 and G_2 be two flow networks (weighted digraphs) with the same vertex set $\mathcal{V}$. Their *union* is defined as a flow network with vertex set $\mathcal{V}$, where for any $u \neq v \in \mathcal{V}$, the capacity of the edge $u \to v$ is the sum of the capacities of the edges $u \to v$ in G_1 and G_2. (If $u \to v$ does not exist in one of the networks, the capacity of $u \to v$ in this network is 0.) The value of the maximum st-flow in the union network equals the sum of the values of the maximum st-flows in G_1 and G_2.
◯	◯	If f is a flow that contains a directed cycle with a positive flow value (i.e., the flow on every edge of the cycle is positive), then f is not a maximum flow.
◯	◯	A "star" digraph that consists of one vertex of out-degree n and n vertices with out-degree 0 has exactly n distinct *topological orderings*.

7. **Dynamic programming: Halloween. (5 points)**

It's Halloween, and your young cousins are dragging you trick-or-treating down the street. You know how many pieces of candy they'll collect at each house and want to minimize the total amount they receive. After visiting house i (starting from house 0, which is your house, where they collect 0 pieces), your cousins allow you to either go to house $i+1$ or skip it and go to house $i+2$ (but you can't get away with skipping two houses in a row).

Assume the number of pieces of candy collected at each house is represented by an integer array `candy[]` of length $n \geq 2$, where the amount given away at house i is `candy[i]` (with `candy[0]=0` representing your house). Determine the minimum total number of pieces of candy that can be collected without skipping two consecutive houses.

Example. If $n = 10$ and `candy[]` is the array below, the minimum obtainable number is 23. This is achieved by visiting houses 0, 2, 3, 5, 6 and 8 (skipping houses 1, 4, 7 and 9), thus collecting $3 + 2 + 5 + 1 + 12 = 23$ pieces of candy.

house	**0**	1	**2**	**3**	4	**5**	**6**	7	**8**	9
candy	**0**	5	**3**	**2**	8	**5**	**1**	10	**12**	13

We will solve this problem using *dynamic programming*. For each i with $0 \leq i < n$, define the following *subproblems*:

`opt[i]` = minimum number of pieces of candy collected from houses 0 to i, *given that house i is visited.*

Consider the following bottom-up implementation, with some parts omitted:

```
static int minCandy(int[] candy) {
     int n = candy.length;
     int[] opt = new int[n];

     opt[0] = candy[0];
     opt[1] = candy[1];

     for (int i = 2; i < n; i++)
         opt[i] = <blank 1>;

     return Math.min(<blank 2>, <blank 3>);
}
```

Fill in the ommited blanks with valid Java code that completes a correct implementation of `minCandy()`:

`<blank 1>` is

`<blank 2>` is

`<blank 3>` is

8. **Multiplicative weights (5 points).**

Consider the *experts problem* with $n \geq 2$ experts over a period of $T \geq 100$ days.

Identify each statement as either always true or sometimes/always false.

true	*false*	
◯	◯	If there is an expert that always predicts correctly, the *multiplicative weights* algorithm makes no fewer mistakes than the *elimination* algorithm.
◯	◯	If there is an expert that always predicts correctly, then once the *multiplicative weights* algorithm does not make a mistake for $2.41 \log_2 n$ consecutive days, it will not make any mistakes in subsequent days.
◯	◯	In the *multiplicative weights* algorithm, if we halve the weight of an expert only on days when both the expert *and the algorithm* make a mistake, the worst-case number of mistakes remains at most $2.41(M + \log_2 n)$.
◯	◯	Suppose T consists of $T/7$ consecutive weeks and that, for each day of the week, there exists an expert who predicts correctly (e.g., one expert for Mondays, another for Thursdays, etc.). Then there exists a prediction algorithm that makes $O(\log n)$ mistakes.
◯	◯	In the *simplified AdaBoost* algorithm, suppose that $T = 100$ and that in every call to the iterate method, the trained decision stump correctly labels points in the training set whose collective weight is at least 90% of the total weight. If the final boosted model's predict method is applied to each point in the training set, it will correctly label all such points.

9. **Intractability (5 points).**

Identify each statement as either always true or sometimes/always false.

true	*false*	
◯	◯	If problems X and Y are in **P**, then X poly-time reduces to Y and Y poly-time reduces to X.
◯	◯	If problem Y is in **P** and 3-SAT poly-time reduces to Y, then **P = NP**.
◯	◯	Both of the following are polynomial-time algorithms: (1) an algorithm that receives a binary array as input and prints the value of every entry; (2) an algorithm that receives a positive integer x as input and prints all integers between 1 and x.
◯	◯	Recall that there is a randomized approximation algorithm for 3-SAT that returns an assignment satisfying a $\frac{7}{8}$ fraction of the equations with probability 0.99. Running this algorithm 100 times and returning the assignment that satisfies the largest number of equations yields an assignment that satisfies over 0.99 of the equations with probability at least 0.99.
◯	◯	Consider the following decision problem: given a 3-SAT instance, determine whether a satisfying assignment with the fewest variables assigned `true` has exactly $\frac{n}{2}$ variables assigned `true`. The witness and verification algorithm below prove that the problem above is in **NP**: *Witness:* A boolean assignment claimed to be a satisfying assignment with the fewest variables assigned `true`. *Verification:* Check that the witness assignment satisfies all equations and has exactly $\frac{n}{2}$ variables assigned `true`. Additionally, for every subset S of variables where $\lvert S\rvert < \frac{n}{2}$, check that the assignment with variables in S assigned `true` and all others assigned `false` is unsatisfiable.

10. **Randomness: majority element (5 points).**

    An element x is the *majority element* of the array `arr` if x appears in more than half of the entries in `arr`. For example, 8 is the majority element of the following integer array, as it appears in 9 out of 16 entries: `[ 8 7 8 8 8 0 8 8 4 4 4 4 8 8 4 8]`.

    The randomized methods `majA` and `majB` below are designed to find the majority element in an array that is *guaranteed to have a majority element*. Both methods utilize a deterministic helper function, `isMaj(arr,x)`, which checks in $\Theta(n)$ time whether a given candidate element x is the majority element of the array `arr`.

```
static Object majA(Object[] arr) {
     Object x;
     Object maj = null;

     for (int t = 0; t < 10; t++) {
          x = arr[StdRandom.uniformInt(arr.length)];

          if (isMaj(arr,x))
               maj = x;
     }

     return maj;
}

static Object majB(Object[] arr) {
     Object x;

     while (true) {
          x = arr[StdRandom.uniformInt(arr.length)];

          if (isMaj(arr,x))
               return x;
     }
}

static boolean isMaj(Object[] arr, Object x) {
     int count = 0;

     for (int i = 0; i < arr.length; i++)
          if (x.equals(arr[i]))
               count++;

     return (count > arr.length / 2);
}
```

Fill in all checkboxes that apply.

☐ `majA` is a *randomized Las Vegas* algorithm and `majB` is a *randomized Monte Carlo* algorithm.

☐ The error probability of `majA` is at most $\frac{1}{2^{10}}$.

☐ The expected number of array accesses made by `majB` is $\Theta(n)$.

☐ The greater the frequency of the majority element in `arr`, the lower the (exact) expected number of array accesses made by `majB`.

☐ The greater the frequency of the majority element in `arr`, the lower the (exact) expected number of array accesses made by `majA`.

11. **Design: majority element. (12 points)**

    This question focuses on *deterministic* algorithms for finding the *majority element*, as defined in Question 10.

    (a) Design an algorithm that outputs the majority element if it exists, and returns `null` otherwise.

    Full credit: The algorithm should run in $O(n \log n)$ time in the worst case, where n is the length of the array (you may assume that n is a power of 2). The algorithm can only use the `equals()` instance method of array elements, which can be assumed to run in $\Theta(1)$ time. That is, implement the method `static Object maj(Object[] arr)`.

    Hint for full credit: *One option* is to use a *divide-and-conquer* approach: split the array into two subarrays, each of half the size. How can the majority elements of the two subarrays help determine the majority element of the entire array?

    Partial credit (at least half): The algorithm should run in $O(n \log n)$ time in the worst case, but may assume that the array elements are Comparables and use the `compareTo()` method. That is, implement the method `static Comparable maj(Comparable[] arr)`.

Choose <u>one</u> option to attempt:

- ◯ Full-credit solution (`Object[]` array).
- ◯ Partial credit solution (`Comparable[]` array).

In the space provided, give a concise English description of your algorithm for solving the problem. You may use any of the algorithms that we have considered in this course (e.g., lectures, precepts, textbook, assignments) as subroutines. If you modify such an algorithm, be sure to describe the modification. Feel free to use code or pseudocode to improve clarity.

(b) As in the previous part of this question, design an algorithm that outputs the *majority element* if it exists, and returns `null` otherwise. However, the new algorithm should run in $O(n)$ time on average *under the uniform hashing assumption.*

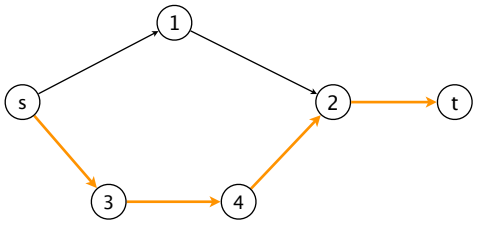
Advice: Recall that hash tables implement a symbol table data type, which does not support duplicate keys. Consider choices of keys and associated values that are correct under this constraint.

Give a concise English description of your algorithm. Feel free to use code or pseudocode to improve clarity.

12. **Design: reachability with even-length paths. (9 points)**

Given a digraph G, a start vertex s, and a target vertex t, we are interested in directed paths of *even length* (i.e., consisting of an even number of edges) from s to t.

Example. In the following digraph, the path $s \to 3 \to 4 \to 2 \to t$ (in orange) is an even-length path from s to t. Note that there is also an odd-length path from s to t: $s \to 1 \to 2 \to t$.



(a) *Is the following claim correct?*
There exists a digraph G with an even-length path from s to t, but none of the even-length paths from s to t in G are *simple*. Recall that a simple path is a path that does not visit any vertex more than once.

◯ *yes* ◯ *no*

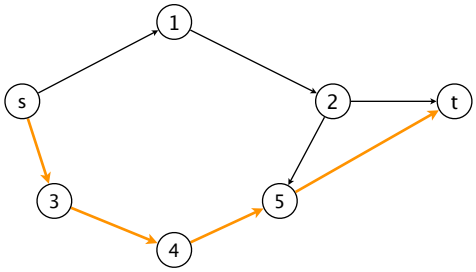
(b) Design an algorithm that, given a DAG G, a start vertex s, and a target vertex t, determines whether there exists a directed path of even length from s to t.

Full credit: The algorithm should run in $O(E+V)$ time in the worst case, where V is the number of vertices and E is the number of edges in G.

Partial credit (at least half): The algorithm should run in $O((E+V)E)$ time in the worst case. Additionally, *the algorithm is only required to output* `YES` *when there exists an even-length path from s to t that is edge-disjoint from any odd-length path between s and t* (i.e., it shares no edges with any odd-length path). That is, the algorithm must

- output `NO` if there is no directed path of even length from s to t.
- output `YES` if there exists a directed path of even length from s to t that is edge-disjoint from all odd-length paths between s and t.
- output either `YES` or `NO` for all other digraphs.

For example, the algorithm may return either `YES` or `NO` for the digraph above with vertices s and t, as the only even-length path from s to t (in orange) shares the edge $2 \to t$ with the odd-length path. However, the algorithm must return `YES` for the following digraph with vertices s and t, as there exists an even-length path from s to t (in orange) that does not share any edge with an odd-length path.



Choose one option to attempt:

- ◯ Full-credit solution.
- ◯ Partial credit solution.

Give a concise English description of your algorithm. Feel free to use code or pseudocode to improve clarity. You may also add a sketch to illustrate your algorithm.

This page is intentionally blank. You may use this page for scratch work.