COMP 3804 — Winter 2026
Solutions Problem Set 3

Some useful facts:

1. $1 + 2 + 3 + \cdots + n = n(n+1)/2$.
2. for any real number $x > 0$, $x = 2^{\log x}$.
3. For any real number $x \neq 1$ and any integer $k \geq 1$,
$$1 + x + x^2 + \cdots + x^{k-1} = \frac{x^k - 1}{x - 1}.$$
4. For any real number $0 < \alpha < 1$,
$$\sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}.$$

Master Theorem:

1. Let $a \geq 1$, $b > 1$, $d \geq 0$, and
$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ a \cdot T(n/b) + \Theta(n^d) & \text{if } n \geq 2. \end{cases}$$
2. If $d > \log_b a$, then $T(n) = \Theta(n^d)$.
3. If $d = \log_b a$, then $T(n) = \Theta(n^d \log n)$.
4. If $d < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$.

Question 1: You are given a min-heap $A[1 \dots n]$ and a variable *largest* that stores the largest number in this min-heap.

In class, we have seen algorithms INSERT(A, x) (which adds the number x to the min-heap and restores the heap property) and EXTRACTMIN(A) (which removes the smallest number from the heap and restores the heap property).

Explain, in a few sentences, how these two algorithms can be modified such that the value of *largest* is correctly maintained. The running times of the two modified algorithms must still be $O(\log n)$.

Solution:

- INSERT(A, x): The algorithm is the same as the one we have seen in class. At the end of this algorithm, we set $\mathit{largest} = \max(\mathit{largest}, x)$. The running time will be $O(\log n) + O(1) = O(\log n)$.
- EXTRACTMIN(A): The algorithm is the same as the one we have seen in class. Note that, if the heap is not empty afterwards, the value of *largest* does not change. If A is empty afterwards, then we set $\mathit{largest} = -\infty$. The running time will be $O(\log n) + O(1) = O(\log n)$.

Question 2: Let m be a large integer and consider m non-empty sorted lists $L_1, L_2, \dots, L_m$. All numbers in these lists are integers. Let n be the total length of all these lists.

Describe an algorithm that computes, in $O(n \log m)$ time, two integers a and b, with $a \leq b$, such that

- each list L_i contains at least one number from the set $\{a, a+1, \dots, b\}$ and
- the difference $b - a$ is minimum.

For example, if $m = 4$,

$$\begin{aligned} L_1 &= (2, 3, 4, 8, 10, 15) \\ L_2 &= (1, 5, 12) \\ L_3 &= (7, 8, 15, 16) \\ L_4 &= (3, 6), \end{aligned}$$

then the output can be $(a, b) = (4, 7)$ or $(a, b) = (5, 8)$.

As always, justify the correctness of your algorithm and explain why the running time is $O(n \log m)$.

Hint: Use a min-heap of size m and use Question 1. For the example, draw it like this, then stare at it until you "see" the algorithm:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	2	3	4				8		10					15	
1				5							12				
						7	8							15	16
		3			6										

Solution: After having stared at the figure long enough, you will have noticed that we want to compute the smallest interval that contains at least one element from each list. We will do this by "sliding" an interval $[a, b]$ from left to right and take care that this interval always contains at least one element from each list.

- For each $i = 1, 2, \ldots, m$, take the first element x_i in list L_i and remove it from L_i. Let H be the sequence $x_1, x_2, \ldots, x_m$. Let *smallest* be the smallest number in H, let *largest* be the largest number in H, and let *answer* = *largest* − *smallest*. Then the interval $[smallest, largest]$ contains at least one element from each list, and the length of this interval is *answer*.
- How do we "slide" this interval to the next one: We delete *smallest* from H. Let i be the index such that *smallest* was in L_i. Then we add the first element in L_i to H, and delete it from L_i. Now we recompute *smallest*, *largest*, and *answer*.
- We repeat this until one of the lists gets empty.
- What are the operations that we need?
  - We have to delete the smallest number in H.
  - We have to insert an element into H.
  - We have to keep track of the largest number in H.
- The first two operations suggest that we store H in a min-heap.
- For the third operation, we will use Question 1.

Here is a more formal description of the algorithm. We assume that every number "knows" the index of the list L_i it is part of.

Initialization:

- For $i = 1, 2, \ldots, m$, let x_i be the first element in L_i, and delete it from L_i.
- Build a min-heap H for the numbers $x_1, x_2, \ldots, x_m$.
- Set *smallest* = MIN(H).
- By scanning $x_1, x_2, \ldots, x_m$, compute the largest among these numbers and store it in the variable *largest*.
- Set *answer* = *largest* − *smallest*.

Repeat the following as long as all lists are non-empty:

- x = EXTRACTMIN(H); this updates *largest*, see Question 1.

- Let i be the index such that x was in L_i.
- Let x_i be the first element in L_i and remove it from L_i.
- INSERT(H, x_i); this updates *largest*, see Question 1.
- Set *smallest* = MIN(H).
- Set *answer* = $\min(\textit{answer}, \textit{largest} - \textit{smallest})$.

After the loop has terminated: Return *answer*.

Running time:

- The initialization takes $O(m)$ time.
- Note that, at any moment, the heap stores m numbers. Therefore, each operation that we perform on the heap takes $O(\log m)$ time.
- There are at most n EXTRACTMIN-operations, at most n INSERT-operations, and at most n MIN-operations.
- Thus, the loop takes time $O(n \log m)$.
- We conclude that the total running time is $O(m+n \log m)$, which is $O(n \log m)$, because $m \leq n$.

Question 3: Let $G = (V, E)$ be an undirected graph. A *vertex coloring* of G is a function $f : V \to \{1, 2, \ldots, k\}$ such that for every edge $\{u, v\}$ in E, $f(u) \neq f(v)$. In words, each vertex u gets a "color" $f(u)$, from a set of k "colors", such that the two vertices of each edge have different colors.

Assume that the graph G has exactly one cycle with an odd number of vertices. (The graph may contain cycles with an even number of vertices.)

What is the smallest integer k such that a vertex coloring with k colors exists? As always, justify your answer.

Solution: Since the graph has an odd cycle, it cannot be colored using two colors. (This is the same as saying that the graph is not bipartite.)

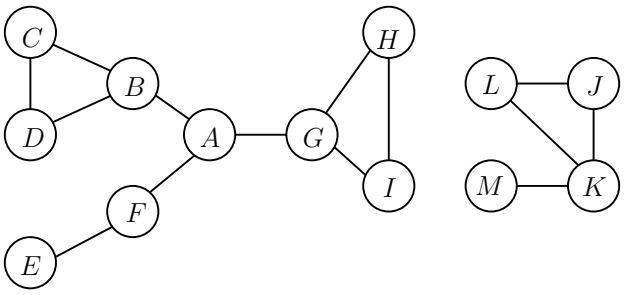
We will show that the graph can be colored using three colors. Let $(v_1, v_2, \ldots, v_k, v_1)$ be the unique odd cycle in G.

We remove one edge of this cycle, say $\{v_1, v_2\}$ from G, and denote the resulting graph by G'. (We only remove this edge, we do not remove the vertices v_1 and v_2.)

We observe that the graph G' does not contain any odd cycle. Therefore, G' is bipartite. Thus, we can split the vertex set V into two sets, say B and R, such that every edge in G' has one vertex in B and the other vertex in R.

We give every vertex of B the color blue, and every vertex of R the color red. This is a valid vertex coloring of G' using two colors. However, the vertices v_1 and v_2 have the same color. Thus, if we add the edge $\{v_1, v_2\}$ to G', we do not get a vertex coloring of the original graph G. We do get a vertex coloring of G, by giving v_1 a new color, say green.

Question 4: Consider the following undirected graph:



Question 4.1: Draw the DFS-forest obtained by running algorithm DFS on this graph. Recall that algorithm DFS uses algorithm EXPLORE as a subroutine.

In the forest, draw each tree edge as a solid edge, and draw each back edge as a dotted edge.

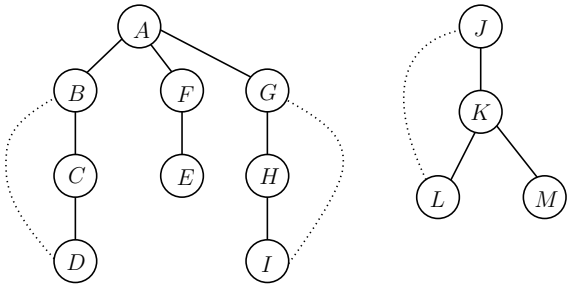
Whenever there is a choice of vertices (see the two lines labeled (*)), pick the one that is alphabetically first.

Question 4.2: Do the same, but now, whenever there is a choice of vertices (see the two lines labeled (*)), pick the one that is alphabetically last.

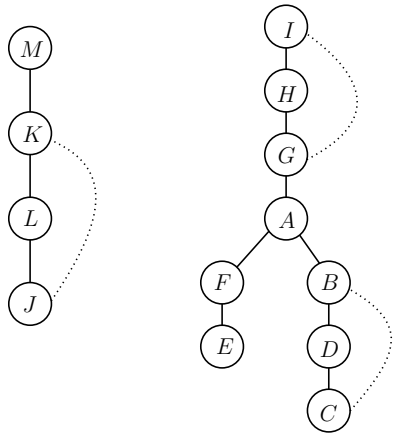
```
Algorithm DFS(G):
for each vertex u
do visited(u) = false
endfor;
cc = 0;
for each vertex v        (*)
do if visited(v) = false
    then cc = cc + 1
         EXPLORE(v)
    endif
endfor
```

```
Algorithm EXPLORE(v):
visited(v) = true;
ccnumber(v) = cc;
for each edge {v, u}        (*)
do if visited(u) = false
    then EXPLORE(u)
    endif
endfor
```

Solution: We start with **(4.1)**. In case there is more than one choice, we pick the alphabetically smallest one. Thus, algorithm DFS(G) starts by calling EXPLORE(A). The vertices in each adjacency list are sorted in increasing order. Here is the resulting DFS-forest, which consists of two trees:



Next we do **(4.2)**. In case there is more than one choice, we pick the alphabetically largest one. Thus, algorithm DFS(G) starts by calling EXPLORE(M). The vertices in each adjacency list are sorted in decreasing order. Here is the resulting DFS-forest, which consists of two trees:



Question 5: Prove that an undirected graph $G = (V, E)$ is bipartite if and only if G does not contain any cycle having an odd number of edges.

Solution: Assume that G is bipartite, i.e., the vertex set V of G can be partitioned into two sets L and R, such that each edge has one vertex in L and one vertex in R. We have to show that G does not contain any odd cycle. Assume there is an odd cycle

$$v_1, v_2, v_3, \ldots v_{2k+1}, v_1.$$

We may assume that v_1 is in L. Then v_2 is in R, v_3 is in L, v_4 is in R, $\ldots$, v_{2k-1} is in L, v_1 is in R. This is a contradiction.

Now we assume that G does not contain any odd cycle. We have to show that G is bipartite. We run algorithm DFS(G) and classify each edge as a tree edge or a back edge. Consider the tree defined by the tree edges. Using this tree, we partition the vertex set V into two subsets L and R: The vertices at the even levels are added to L, whereas the vertices at the odd levels are added to R. It is clear that for every tree edge, one vertex is in L and the other vertex is in R. Assume, by contradiction, that there is a back edge $\{u, v\}$ such that both u and v are in L (the case when they are both in R is symmetric). We may assume that v is in the subtree of u. Consider the following cycle in G:

- Start at u, follow tree edges to v, then follow the back edge to u.

Since both u and v are in L, this cycle has an odd number of edges, which is a contradiction.

Question 6: Since Katy Perry and Justin Trudeau miss each other very much, they decide to meet. Katy and Justin live in a connected, undirected, non-bipartite graph $G = (V, E)$. Katy lives at vertex k, whereas Justin lives at vertex j.

Katy and Justin move in steps. In each step, Katy must move from her current vertex to a neighboring vertex, and Justin must move from his current vertex to a neighboring vertex.

Both Katy and Justin have complete knowledge of the graph. They are in constant communication so that each person knows where the other person is.

Prove that there exists a moving strategy such that Katy and Justin meet each other at the same vertex.

Hint: While moving around in the graph, each of Katy and Justin may visit the same vertex more than once, and may traverse the same edge more than once.

If the graph G consists of the single edge $\{k, j\}$, then they will never be at the same vertex. But in this case G is bipartite.

If the graph contains a path having 8 edges, Katy lives at one end-vertex, and Justin lives at the other end-vertex, will they ever be at the same vertex?

Question 5 is useful.

Solution: To get some intuition, assume there is a path $(1, 2, 3, 4, 5, 6, 7)$ with 6 edges, Katy lives at vertex 1 and Justin lives at vertex 7.

- In step 1, Katy moves to 2, and Justin moves to 6.
- In step 2, Katy moves to 3, and Justin moves to 5.
- In step 3, Katy moves to 4, and Justin moves to 4. Both Katy and Justin are happy!

This works as long as the number of edges on the path is even. In other words, if there is a path with an even number of edges between the vertices k and j, then there is a strategy such that Katy and Justin meet at the same vertex. In fact, this path may be a *walk*, i.e., a sequence of vertices such that (i) the first vertex is k, the last vertex is j, and there is an edge between any two consecutive vertices in this sequence. Note that in a walk, vertices may be visited more than once and edges may be traversed more than once.

If we can prove that there always is a walk with an even number of edges between the vertices k and j, then we have answered the question.

Since the graph G is not bipartite, we know from Question 5 that there is a cycle C in G with an odd number of edges. Let c be an arbitrary vertex on this cycle C. Since the graph G is connected, there is a path P from k to c, and there is a path Q from c to j.

- If the concatenation PQ (which is a path or a walk) has an even number of edges, then we are done.
- Assume that PQ has an odd number of edges. Consider the walk W that starts at k, follows P to c, then walks along the cycle C back to c, and then follows Q from c to j. Since C has an odd number of edges, the number of edges on W is "odd plus odd", which is even.