Question 1: Write your name and student number.

Solution: Wanda Maximoff, 2800

Question 2: Let $G = (V, E)$ be an undirected graph, and let $e$ be an edge in $E$. Describe an algorithm that decides, in $O(|V| + |E|)$ time, whether $G$ has a cycle containing the edge $e$. Justify your answer. You may use any result that was presented in class.

Solution: The algorithm does the following.

Step 1: Run algorithm DFS($G$) and classify each edge as a tree edge or a back edge.

Step 2: If $e$ is a back edge, return YES. Otherwise, $e$ is a tree edge, in which case we proceed with Step 3.

Step 3: Assume that $e$ is a tree edge. Let $u$ and $v$ be the two vertices of $e$, and assume that, in the DFS-forest, $v$ is in the subtree of $u$.

- Consider the DFS-tree that contains the tree edge $e$.
- We color “red”, all vertices on the path from the root to $u$.
- We color “blue”, all vertices in the subtree of $v$.
- Go through all back edges. If there is a back edge $\{a, b\} \neq e$, such that $a$ is red and $b$ is blue (or vice versa), return YES. Otherwise, return NO.

Step 1 takes $O(|V| + |E|)$ time. Step 2 takes $O(1)$ time. Step 3 takes $O(|V| + |E|)$ time. Thus, the overall running time is $O(|V| + |E|)$.

Why is this algorithm correct:

- First assume that the algorithm returns YES. There are two possibilities:
  - If $e$ is a back edge: Let $u$ and $v$ be the two vertices of $e$, and consider the DFS-tree that contains $e$. The tree edges on the path between $u$ and $v$ in this DFS-tree, together with the back edge $e$, form a cycle containing the edge $e$. Thus, in this case, the output of the algorithm is correct.
  - If $e$ is a tree edge: In this case, there is a back edge $\{a, b\} \neq e$, such that $a$ is red and $b$ is blue. Observe that the path between $a$ and $b$ in the DFS-tree contains the edge $e$. This path, together with the back edge $\{a, b\}$, form a cycle containing the edge $e$. Thus, in this case, the output of the algorithm is correct.

- Now assume that the graph $G$ has a cycle containing $e$. 

– If $e$ is a back edge, then the algorithm correctly returns YES.

– If $e$ is a tree edge, then there must be a back edge $\{a, b\}$, where $a$ is red and $b$ is blue. Thus, the algorithm correctly returns YES.

You can simplify the above algorithm in the following way: Let $u$ and $v$ be the two vertices of the edge $e$. In Step 1, i.e., when running algorithm $\text{DFS}(G)$, start by calling $\text{EXPLORE}(u)$. Then, $u$ is the root of a $\text{DFS}$-tree, implying that there is only one “red” node, namely node $u$.

Here is an even simpler solution: Let $u$ and $v$ be the two vertices of the edge $e$. Let $G'$ be the graph obtained by removing $e$ from the input graph $G$. Run algorithm $\text{EXPLORE}(u)$ on the graph $G'$. The original graph $G$ has a cycle containing $e$ if and only if $\text{EXPLORE}(u)$ encounters vertex $v$.

**Question 3:** Consider the following directed graph:

![Directed Graph](image)

(3.1) Draw the $\text{DFS}$-forest obtained by running algorithm $\text{DFS}$. This algorithm uses algorithm $\text{EXPLORE}$ as a subroutine.

Classify each edge as a tree edge, forward edge, back edge, or cross edge. In the $\text{DFS}$-forest, give the pre- and post-number of each vertex. Whenever there is a choice of vertices, pick the one that is alphabetically first.

(3.2) Draw the $\text{DFS}$-forest obtained by running algorithm $\text{DFS}$. Classify each edge as a tree edge, forward edge, back edge, or cross edge. In the $\text{DFS}$-forest, give the pre- and post-number of each vertex. Whenever there is a choice of vertices, pick the one that is alphabetically last.
Solution: We start by recalling algorithms DFS and EXPLORE:

Algorithm DFS(G):
for each vertex v
do visited(v) = false
endfor;
clock = 1;
for each vertex v
do if visited(v) = false
then EXPLORE(v)
endif
endfor

Algorithm EXPLORE(v):
visited(v) = true;
pre(v) = clock;
clock = clock + 1;
for each edge (v, u)
do if visited(u) = false
then EXPLORE(u)
endif
endfor;
post(v) = clock;
clock = clock + 1

We start with (3.1). In case there is more than one choice, we pick the alphabetically smallest one. Thus, algorithm DFS(G) starts by calling EXPLORE(A). Since A has two outgoing edges, one to vertex B and the other to vertex F, algorithm DFS(G) will then call EXPLORE(B). Here is the resulting DFS-forest, which consists, in fact, of one tree:
Next we do (3.2). In case there is more than one choice, we pick the alphabetically largest one. Thus, algorithm DFS(G) starts by calling EXPLORE(H). Here is the resulting DFS-forest, which consists of two trees:

Question 4: In order to increase revenue, the owner of Michiel’s Taxi Company introduces the policy that all taxi drivers must always take the longest path in the directed graph $G = (V, E)$ representing the roads of Ottawa. For each directed edge $(u, v)$ in $E$, let $wt(u, v) > 0$ denote the length of this edge.
The following approach is suggested to compute the length of the longest path from a source vertex $s$ to each vertex of $V$: For each directed edge $(u, v)$ in $E$, define $wt'(u, v) = -wt(u, v)$. Run a shortest-path algorithm using the new weights $wt'$. In the output, replace each shortest-path length $L$ by $-L$.

In both of the following two parts of this question, assume that the directed graph $G$ is acyclic.

(4.1) Prove or disprove: Algorithm ShortestPathAcyclic, which we have seen in class, correctly computes, for every vertex $v$, the length of the longest path from $s$ to $v$. (Note: You have to run the algorithm exactly as given at the end of this assignment.)

(4.2) Prove or disprove: Dijkstra’s algorithm correctly computes, for every vertex $v$, the length of the longest path from $s$ to $v$. (Note: You have to run Dijkstra’s algorithm exactly as given at the end of this assignment. After a vertex $v$ has been deleted from the set $Q$, it may happen that the value of $d(v)$ gets decreased later in the algorithm.)

**Solution:** We start with (4.1). The algorithm is as follows:

**Algorithm** ShortestPathAcyclic($G, s, wt'$):
- topologically sort $G$;
- denote the resulting numbering of the vertices by $v_1, v_2, \ldots, v_n$;
- assume that $s = v_1$;
- $d(s) = 0$;
- for $i = 2$ to $n$
  - do $d(v_i) = \infty$;
- endfor;
- for $i = 1$ to $n$
  - do $u = v_i$;
    - for each edge $(u, v)$
      - do if $d(u) + wt'(u, v) < d(v)$
        - then $d(v) = d(u) + wt'(u, v)$
      - endif
    - endfor
- endfor

The claim in (4.1) is true: The proof consists of copying pages 93–95 of the notes available at


where you replace $wt$ by $wt' = -wt$. Note that the proof never uses that weights are positive.
Next we do (4.2). The algorithm is as follows:

Algorithm Dijkstra\((G, s, wt')\):
for each \(v \in V\)
do \(d(v) = \infty\)
endfor;
d\((s) = 0\);
\(S = \emptyset\);
\(Q = V\);
while \(Q \neq \emptyset\)
do \(u = \) vertex in \(Q\) for which \(d(u)\) is minimum;
delete \(u\) from \(Q\);
insert \(u\) into \(S\);
for each edge \((u, v)\)
do if \(d(u) + wt'(u, v) < d(v)\)
then \(d(v) = d(u) + wt'(u, v)\)
endif
endfor
endwhile

We show that the claim in (4.2) is not true. The reason is that the proof uses the fact that weights are positive.

Consider the following directed and acyclic graph with positive edge weights \(wt(u, v)\):

For each edge \((u, v)\), define \(wt'(u, v) = -wt(u, v)\). We run algorithm Dijkstra\((G, s, wt')\).

- At the start: \(d(s) = 0\), \(d(a) = d(b) = d(c) = \infty\), \(Q = \{s, a, b, c\}\).
- After the first iteration: \(d(s) = 0\), \(d(a) = -3\), \(d(b) = -2\), \(d(c) = \infty\), \(Q = \{a, b, c\}\).
- After the second iteration: \(d(s) = 0\), \(d(a) = -3\), \(d(b) = -2\), \(d(c) = -4\), \(Q = \{b, c\}\).
- After the third iteration: \(d(s) = 0\), \(d(a) = -3\), \(d(b) = -2\), \(d(c) = -4\), \(Q = \{b\}\).
- After the fourth iteration: \(d(s) = 0\), \(d(a) = -4\), \(d(b) = -2\), \(d(c) = -4\), \(Q = \emptyset\).
Thus, the algorithm tells us that the length of the longest path from $s$ to $c$ is equal to 4. This is wrong, because this longest path has length 5.

**Question 5:** Let $G = (V, E)$ be a connected undirected graph in which each edge has a positive weight. You may assume that no two edges have the same weight.

(5.1) Prove or disprove: The edge with the second smallest weight is an edge in the minimum spanning tree of $G$.

(5.2) Prove or disprove: The edge with the third smallest weight is an edge in the minimum spanning tree of $G$.

**Solution:** We start with (5.1) and show that the claim is true: Consider what happens when we run Kruskal’s algorithm. First, the algorithm sorts the edges in increasing order of their weights. Then it initializes a forest consisting of $|V|$ trees, each one consisting of one single vertex.

- The algorithm considers the shortest edge $\{u, v\}$. At this moment, $u$ and $v$ are not in the same tree. Therefore, the algorithm adds the edge $\{u, v\}$ to the minimum spanning tree, and forms the union of $\{u\}$ and $\{v\}$.

- The algorithm then considers the second shortest edge $\{u', v'\}$. At this moment, the forest consists of one tree for $\{u, v\}$, and $|V| - 1$ trees of size one. It follows that $u'$ and $v'$ cannot be in the same tree. As a result, the algorithm adds the edge $\{u', v'\}$ to the minimum spanning tree.

We now show that the claim in (5.2) is not true: Let $G$ be the complete graph with three vertices and three edges. In other words, $G$ is a triangle. These edges have weights 1, 2, and 3. The minimum spanning tree consists of the two edges having weights 1 and 2; it does not contain the edge of weight 3.

In case you do not like this example (because you may think that the claim is true as soon as your graph gets sufficiently large), define a graph $G$ that consists of a triangle whose edges have weights 1, 2, and 3, plus as many additional vertices as you like. Add edges to the graph so that it is connected, and give each of these edges a weight that is larger than 3. In the resulting graph, the third smallest edge has weight 3 and is not contained in any minimum spanning tree.
**Question 6:** In class, we have seen a data structure for the Union – Find problem that stores each set in a linked list, with the header of the list storing the name and size of the set. Using this data structure, any operation Find\(x\) takes \(O(1)\) time, whereas any operation \(\text{Union}(A, B, C)\) takes \(O(\min(|A|, |B|))\) time.

Consider the same data structure, except that the header of each list only stores the name of the set (and not the size). Show that, in this new data structure, any operation Find\(x\) can be performed in \(O(1)\) time, and any operation \(\text{Union}(A, B, C)\) can still be performed in \(O(\min(|A|, |B|))\) time.

**Solution:** Each set \(A\) is stored in a list:

- The header stores the name of the set.
- Each other node stores one element of the set, a pointer to the next node in the list, and a pointer to the header.
- There are pointers to the header and tail of the list.

For \(\text{Find}(x)\), we get a pointer to the node storing \(x\). We follow the pointer to the header of the list and return the name of the set stored at the header. This obviously takes \(O(1)\) time.

For \(\text{Union}(A, B, C)\), we do the following:

\[
\begin{align*}
a &= \text{header of the list storing } A; \\
b &= \text{header of the list storing } B; \\
\text{while } a \neq \text{tail and } b \neq \text{tail} & \text{ do } a = \text{next}(a); \\
& \quad b = \text{next}(b) \\
\text{endwhile}; \\
\text{if } a = \text{tail} & \text{ then } // \text{ comment: We know that } |A| \leq |B| \\
& \quad \text{add the list storing } A \text{ at the end of the list storing } B \text{ and make changes} \\
& \quad \text{as we did in class} \\
\text{else } // \text{ comment: We know that } |A| > |B| & \text{ add the list storing } B \text{ at the end of the list storing } A \text{ and make changes} \\
& \quad \text{as we did in class} \\
\text{endif}
\end{align*}
\]

In the while-loop, we simultaneously walk along the lists storing \(A\) and \(B\). The running time of the loop is determined by the number of steps made by the first of \(a\) and \(b\) to reach the end of its list. This means that the loop takes \(O(\min(|A|, |B|))\) time. As we have seen in class, the rest of the algorithm, in which we perform the actual Union-operation, also takes \(O(\min(|A|, |B|))\) time.
**Question 7:** You are given two binary strings \( A = a_1a_2\ldots a_m \) and \( B = b_1b_2\ldots b_n \). A common substring of \( A \) and \( B \) is a binary string that occurs consecutively both in \( A \) and in \( B \).

Give a dynamic programming algorithm (in pseudocode) that computes, in \( O(mn) \) time, a longest common substring of \( A \) and \( B \). Argue why your algorithm is correct.

For example, for the input strings \( A = 0001010111 \) and \( B = 01010101000 \), the output will be the string 010101.

**Hint:** Define \( L[i, j] \) to be the length of a longest common substring of \( A \) and \( B \) that ends at \( a_i \) and \( b_j \). Observe that \( L[i, j] = 0 \) if \( a_i \neq b_j \).

**Solution:** The hint tells us that we have to consider the values \( L[i, j] \). (Why: because for these variables, it is easy to get a recurrence.) First of all, how do these values help us in finding the length \( L \) of a longest common substring of the strings \( A \) and \( B \)? Here is the answer: Consider the last symbol in the longest common substring of \( A \) and \( B \). This symbol must occur at some position, say \( i \), in \( A \), and it must occur at some position, say \( j \), in \( B \). This implies that

\[
L = \max\{L[i, j] : 1 \leq i \leq m, 1 \leq j \leq n\}.
\]

We want to apply dynamic programming, so we have to go through the three steps, as we did in class.

**Step 1:** Show that there is optimal substructure. Let \((c_1, c_2, \ldots, c_k)\) be a longest common substring of the strings \( A \) and \( B \). Look where the last symbol \( c_k \) occurs in \( A \) and \( B \). That is, let \( i \) and \( j \) be the indices such that \( c_k = a_i = b_j \). Then \( L[i, j] = k \).

- If \( k \geq 2 \), then \((c_1, c_2, \ldots, c_{k-1})\) must be a longest common substring that ends at \( a_{i-1} \) and \( b_{j-1} \).

This shows that there is optimal substructure.

**Step 2:** Set up a recurrence relation for the \( L[i, j] \)-values. For \( 0 \leq i \leq m \) and \( 0 \leq j \leq n \):

\[
L[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0, \\
0 & \text{if } i \geq 1 \text{ and } j \geq 1 \text{ and } a_i \neq b_j, \\
1 + L[i - 1, j - 1] & \text{if } i \geq 1 \text{ and } j \geq 1 \text{ and } a_i = b_j.
\end{cases}
\]

**Step 3:** Solve the recurrence, in a bottom-up order. While doing this, maintain the largest \( L[i, j] \)-value seen so far in a variable \( L \).
We start with a preliminary algorithm that returns the length of a longest common substring of $A$ and $B$:

```
for $i = 0$ to $m$ do $L[i, 0] = 0$ endfor;
for $j = 1$ to $n$ do $L[0, j] = 0$ endfor;
$L = 0$;
for $i = 1$ to $m$
  do for $j = 1$ to $n$
    do if $a_i = b_j$
      then $L[i, j] = 1 + L[i - 1, j - 1]$  
      else $L[i, j] = 0$
    endif;
    $L = \max(L, L[i, j])$
  endfor;
endfor;
return $L$
```

The first for-loop takes $O(m)$ time; the second for-loop takes $O(n)$ time. The nested for-loop takes $O(mn)$ time. Thus, the overall running time is $O(mn)$.

We now modify the algorithm so that it also returns a longest common substring of $A$ and $B$. Besides the variable $L$, we will maintain a variable $p$ that has the following property: Note that $L$ is the length of a longest common substring that the algorithm has found so far. The corresponding common substring will be $a_{p-L+1}, \ldots, a_p$. (Note: If $L = p = 0$, then this common substring is the empty string.)

```
for $i = 0$ to $m$ do $L[i, 0] = 0$ endfor;
for $j = 1$ to $n$ do $L[0, j] = 0$ endfor;
$L = 0$;
p = 0;
for $i = 1$ to $m$
  do for $j = 1$ to $n$
    do if $a_i = b_j$
      then $L[i, j] = 1 + L[i - 1, j - 1]$  
      else $L[i, j] = 0$
    endif;
    if $L[i, j] > L$
      then $L = L[i, j];$
      $p = i$
    endif
  endfor;
endfor;
return $L$ and $a_{p-L+1}, \ldots, a_p$
```

The total running time of this algorithm is still $O(mn)$.