Homework Assignment #5

Due: Noon, 11th Dec, 2017
Submit **printed** solutions via eClass
Submit **handwritten** solutions at dropbox on CSC level 1

CMPUT 204
Department of Computing Science
University of Alberta

Note: All logs are base-2 unless stated otherwise.
You should answer all questions. Your max-grade is 110.

Exercise I. A lone traveler hikes in the desert. She has a single water bottle that allows her to walk k kms before she suffers from dehydration. She has a map of the n water-holes that are scattered along the desert, and can plan her hike in advance, especially the places where she stops to fill her bottle.

(a) Show that if she hikes on a single clear path (i.e., the desert is 1-dimensional) then the greedy algorithm where she stops along the furthest reachable waterhole to fill her bottle is the one that minimizes the number of stop. Describe how to implement the greedy algorithm in $O(n \log(n))$ time.
Answer. The algorithm works by sorting the waterholes according to location along the path, then picks the furthers waterhole which is still reachable from the current location.

```
Procedue Travel($A$) **$A$ is the array of waterhole locations
    Sort($A$)
    $loc \leftarrow 0$
    $stops \leftarrow 0$
    $i \leftarrow 0$
    while ($i \leq n$)
        while ($i < n$ and $A[i+1] \leq loc + k$)     ** As long as the next waterhole is reachable
            $i \leftarrow i + 1$
        $loc \leftarrow A[i]$
        $stops \leftarrow stops + 1$
```

First, sorting A takes $O(n \log(n))$ time using, say, MergeSort. Then it is evident that for any time we increment i we only do $O(1)$-work, so the rest of the algorithm takes $O(n)$. This results in an overall runtime of $O(n \log(n))$.
As for correctness:
The optimal substructure property is simple to see. Whichever waterhole we pick as first, we better make the minimal number of stops on the remaining waterholes in order to minimize the number of stops we make overall.
The substitution property is also quite simple. Let $A[i_1], ..., A[i_t]$ be a waterhole sequences such that $A[i_1] \leq k$ (we can reach the first stop from the origin) and such that any consecutive two waterholes are of distance $\leq k$. We show that we can replace the first stop with the furthest waterhole from the start. If $A[i_1]$ isn't the furthest waterholes with distance $\leq k$ then replace it with the $A[furthest]$ and remove from the sequence any stop that comes before $A[furthest]$. The new sequence, $A[furthest], A[i_j], ..., A[i_t]$ is still a solution to the problem — the new first stop is within distance k of the origin, and $A[i_j] - A[furthest] \leq A[i_j] - A[i_{j-1}] \leq k$ so any pair of consecutive stops is within distance k.

(b) Show that if she hikes in the wilderness (i.e., the desert is 2-dimensional) then the greedy algorithm doesn't minimize the number of stops made by the traveler.
Answer. Here's one counter example. The traveler begins at the origin $(0,0)$, and has one water hole at $(k,0)$, another waterhole at $(0,k-1)$, another waterhole at $(1,k+1)$ and the endpoint of $(k+1,k+1)$. The greedy algorithm first picks the stop which is the furthest away from $(0,0)$ yet still reachable with distance $\leq k$ — which is $(k,0)$ and then gets stuck and must travel back to the origin. The shortest path to the end point is through $(0,k-1)$, $(1,k+1)$ and $(k+1,k+1)$.

(c) Give an algorithm that minimizes the number of stops the traveler makes regardless of the desert being 2-dimensional, 3-dimensional (a space traveler) or through any other metric. Describe how to implement your algorithm in $O(n^2)$-time.
Answer. The algorithm does BFS on a graph whose set of nodes is the starting point, the end point, and all waterholes in the map. For every pair of nodes that are within distance k we put an edge in the graph. Note that populating this graph takes $O(n^2)$ time, since we need to check for the distance between any pair of nodes. Once we write down the graph, we do a BFS traversal that take $O(|V| + |E|)$-time,

where the number of nodes is $n+2$ and the number of edges is $O(n^2)$. Overall, the runtime is $O(n^2)$. As we know, BFS finds a shortest path from the origin to the destination, so this necessarily minimizes the number of stops the traveler makes.

Exercise II.

(a) Prove that in any graph with edge-weights that are distinct (no pair of edges exists with the same weight), then both Prim and Kruskal *must* output the same MST, regardless of the starting node.

Answer. If all edge-weights are distinct, the MST is unique. Since there's a single MST, both Prim and Kruskal must output the only MST of the graph regardless of the starting node.

(b) Give an example of a graph where the edge-weights aren't distinct, when Prim, Kruskal and Dijkstra *must* output the same tree regarding of the starting node.

Answer. Let G be a graph which is a tree to begin with (and make all edges have the same weight to satisfy the required...) Therefore, there's a unique spanning tree, let alone a unique MST and a unique Shortest-Paths tree. All algorithms must produce this tree.

Hint: Each question can be answered with a 1-sentence-long answer.

Problem 1. (40 pts) For each of the following problems, describe a / the greedy-approach for a solution. If the greedy approach works and is *guaranteed to always* find an optimal solution – prove it; otherwise, give a counter example where the greedy approach fails to find an optimal solution.

(a) (10 pts) The Equal-Weight Integral Knapsack problem: Your input is a capacity W and a set of n non-breakable items (you either take them or leave them, no fractional takes) with values v_i and weight w_i, and your goal is to find a maximal set of items whose total weight is at most $\leq W$ with maximal profit. However, it turns out that all weights of all items are the same: $w_1 = w_2 = ... = w_n = w$.
Answer. The greedy algorithm is the same as the same for fractional knapsack (or even simple, as we can order the items in descending value order). Clearly, in this order, for any $i < j$ we have $\frac{v_i}{w_i} = \frac{v_i}{w} \geq \frac{v_j}{w} = \frac{v_j}{w_j} \geq \frac{v_j}{w_j}$. Therefore, we sort items according to the value/weight ratio and take them one by one until any single item cannot be added. (Note that if we can't add item i as its weight is too big for the knapsack, then we won't take any item $j > i$ as those have the same weight as w_i.)

The greedy algorithm for this problem does work. The problem satisfies the two required properties. Subproblem optimality is the same as before: given that we pick items $i_1, i_2, ..., i_k$, then the remaining items must be an optimal solution to the subproblem of Knapsack, with capacity: $W - kw$ among the remaining items. (Otherwise, we would find an even more profitable bundle of items of the remaining $n-k$ items and pad those with the k items chosen.)

Substitution property is also satisfied. Suppose A is an optimal solution to the problem that doesn't pick the first item in this order. If A is empty — and an optimal solution — it means that even the first item doesn't fit into the knapsack, and since all items have the same weight it means that $W < w$ thus our greedy algorithm produces an empty solution as well. If A isn't empty, if A contains item 1 (the most profitable item) then we're done; otherwise let us replace some item $k \in A$ with the first item. Since $w_k = w_1$ then the replacement doesn't exceed the knapsack's capacity; and since $v_1 \geq v_k$ then the replacement cannot decrease A's total profit.

(b) (10 pts) The Coin Change problem: In the country of New-Cland, currency is arranged in coins/bills of increasing C values (for a given integer $C > 1$). I.e., they have coins of 1¢, C¢, C^2¢, C^3¢, ... C^k¢ up to some k. (Think of $C = 10$ and a subset of our bills and coins.) You are a teller and you have to return to the costumer a total of x¢ using the smallest number of coins. Your input is x, and your output is a sequence $y_0, y_1, ..., y_k$ such that (i) $x = \sum_i y_i C^i$ and (ii) you minimize the number of coins used: $\sum_i y_i$.
Answer. The greedy algorithm for this problem fits as many of the highest value coin C^k so that won't exceed x, then recurses on the leftover change with the coins from 1 to C^{k-1}. And though it is described recursively, we can apply it using a loop.

```
procedure CoinChange($x, C, k$)
$coins$ ← new array of size $k+1$
for ($i$ from $k$ downto 0) do
    $coins[i+1] \leftarrow \lfloor \frac{x}{C^i} \rfloor$
    $x \leftarrow x - C^i \times coins[i+1]$        ** $x$ becomes the remainder of dividing $x$ by $C^i$
return $coins$
```

The greedy algorithm for this problem does work. The problem satisfies the two required properties. Subproblem optimality is, as ever, simple: take any minimal solution in terms of the number of coins, and pick any coins $y_1, y_2, .., y_t$ in this solution (whose worth is $C^{i_1}, C^{i_2}, ..., C^{i_t}$), and the remaining coins must be a minimal set of coins in which we can break $x - \left(C^{i_1} + C^{i_2} + ... + C^{i_t}\right)$. (Otherwise, there would exist a different set of even fewer coins so combining those with the t coins we took out we are left with a smaller set than an optimal set.)

Substitution property also holds. Fix x and let A be a minimal set of coins whose values sums to x. First we argue that due to optimality, then for any coin of type j (namely, C^j) — except for maybe the very largest one C^k¢ — we have that A has at most $C-1$ coins of that type. The reason is simple: had

A had $\geq C$ coins of type C^j¢ we can convert them into a single coin of type C^{j+1}¢ and by exchanging C coins into a single coin we have strictly decreased the size of A — a smallest set.

Suppose the greedy solution picks as the very first coin some C^i, as this is the largest coin such that $C^i \leq x$ (so $C^{i+1} > x$ unless $i = k$). ASOC that A doesn't have a C^i coin. Clearly A cannot have any coin in C^{i+1}¢, C^{i+2}¢,..., C^k¢as those exceed x, and can have at most $(C-1)$ of each coin of value $\leq C^{i-1}$¢. Hence, the total value of all coins in A is at most $\sum\limits_{j=0}^{i-1}(C-1)\cdot C^j = (C-1)\cdot\frac{C^i-1}{C-1} = C^i - 1 < C^i \leq x$ and so A cannot sum to x. Therefore, A too must have a C^i coin.

(c) (10 pts) The Maximal Weighted Matching problem: You are given an undirected bipartite graph, where the vertex-set is partitioned into R and L and every edge connects a node in R with a node in L, and where the edges have weights $w : E \to \mathbb{R}$. A *matching* M in this graph is a subset of the edges of the graph which are vertex-disjoint; namely, for every node u there is at most a single edge in M that is adjacent to it.
Think of this problem as a Kidney Transfer problem: on the left you have the kidney donors, on the right you have the kidney patients (who need a transplant), and each edge represents that chances that the transplant from this donor to this patient will succeed. Since each donor can give at most one kidney and each patient will receive at most one kidney, you are looking to find a matching that has the largest chances of supplying patients with functioning kidneys.

Your goal is to pick a maximal matching. The input is a graph G (including weights on the edges), and your output is a subset of the edges of the graph $M \subset E$ such that M is a matching and you maximize the total cost of all edges in the matching: $\sum\limits_{e\in M} w(e)$.

Answer. The greedy algorithm works by sorting all edges in the bipartite graph from largest to smallest, picks the largest edge (u, v) and removes all edges that either touch u or touch v. In fact, we don't really need to remove edges, the simplest way to check a node hasn't been touched (add a boolean flag per node).

```
procedure GreedyMaxMatching($G, w$)
foreach (node $u \in V(G)$) do
    $u.picked$ ←FALSE
Sort edges according to descending weight
$A \leftarrow \emptyset$          ** $A$ is the chosen set of edges.
foreach edge $e = (u, v)$ in the descending weights order
    if ($u.picked$ =FALSE and $v.picked$ =FALSE) do
        $A \leftarrow A \cup \{e\}$
        $u.picked$ ←TRUE
        $v.picked$ ←TRUE
return $A$
```

Unfortunately, the greedy algorithm fails to retrieve a maximal matching. Considering the graph with two right nodes a, b and two left nodes x, y and all possible 4 edges of weights: $w(a, x) = 7$, $w(a, y) = w(b, x) = 6$ and $w(b, y) = 1$. The greedy algorithm picks first the edge (a, x) and then is forced to pick the only possible non-conflicting edges (b, y), and ends with a matching of total weight $7 + 1 = 8$. The optimal solution however is to pick the two edges of weight 6 and get a total matching weight of 12.

(d) (10 pts) The Longest-Path problem: You are given an undirected graph with weights on the edges and a pair of vertices s, t. Your goal is to find a longest simple path from s to t.
(The greedy approach is a version of Dijkstra, where you pick the max-edge crossing the cut between the current set of vertices and the remaining set of vertices.)
Answer. The greedy algorithm for finding such longest-paths tree would be to simulate Dijkstra starting with the set $S = \{s\}$ and incrementing it one node at a time, where in each step we pick a node of maximal current length.

X not find

```
procedure GreedyLongPaths($G, s$)
foreach (node $u$) do
    $u.dist \leftarrow -\infty$
    $u.predec \leftarrow$NIL
$S = V$
$s.dist = 0$
while ($|S| > 0$) do
    pick $v \in S$ with maximal $dist$ and remove it from $S$
    foreach neighbor $u$ of $v$ do
        if ($u.dist < v.dist + w(u, v)$) then             ** a revised version of relax()
            $u.dist = v.dist + w(u, v)$
            $u.predec = v$
return $A$
```

Unfortunately, this algorithm fails to find such a longest-path tree. Consider a graph on three vertices s, t, a where $w(s, t) = 7$, $w(s, a) = 6$ and $w(a, t) = 5$. Our revised algorithm picks first s, sets $t.dist = 7$ and $a.dist = 6$, then picks t (using the edge (s, a), then might as well halt as it doesn't revise the tree it creates. Thus, the greedy algorithm pick a path $s \to t$ of length 7. Clearly the path (s, a, t) has length $6 + 5 = 11$ and it is therefore the longest $s \to t$ path.

Problem 2. (20 pts) Consider an edge weighted graph G with weights $w : E \to \mathbb{R}$, and a minimum spanning forest of the graph, $F = \{T_1, T_2, ..., T_k\}$. In this question you are asked to deal with a dynamic update of the edge structure of G: a new edge e with weight $w(e))$ might be added to the graph, or an existing edge e might be removed from the graph.
Given an Insertion / Deletion of an edge, describe a procedure that would find a new minimal spanning forest of the revised G. Your algorithm must always run faster than recomputing the MST of each connected component from scratch, i.e., $o(|E| \log |V|)$. Give the running time of your procedure.
Answer. Before we describe the update procedure, we start with a simple $O(n)$-time pre-processing — we traverse (say, using BFS) each tree $T_i \in F$ and label all of its nodes with i. We save this label per node under the field as $u.label$. We assume this was done initially, when F was constructed.

- Suppose an edge (u, v) is added such that $u.label \neq v.label$ (the two nodes belong to two different trees).

  Here we need to merge the two trees containing u and v into a single tree, and update the label of all the nodes in these two trees. We do this by constructing a graph containing solely the edges in trees T_i and T_j (where $u \in T_i, v \in T_j$) and the new edge (u, v).

  The reason is that we now T_i and T_j span their own respective connected-components, and since those are connected-components, and so G has no edge connecting any node in T_i with a node in T_j before this edge-insertion. Therefore, the new edge merges the two components into a single one, and moreover, to span all nodes in the new component we must use the edge (u, v) (the only possible edge that allows us to connect the nodes in T_i to the nodes in T_j).

- Suppose an edge (u, v) is added such that $u.label = v.label$ (the two nodes belong to the same tree).

  Denoting T_i as the tree that spans the component that both u and v belong to, then this tree might be revised. In the new graph, we might substitute (u, v) with a different edge. But which one? We know that u and v are connected by T_i, but since $(u, v) \notin T$ then u and v were connected by a simple path of length > 1. This path, together with (u, v) creates a cycle. Based on Kruskal's analysis, we can deduce that we must omit from this cycle a single edge (u', v') which is (one of) the heaviest edge(s) on this cycle. So, we find the path P from u to v on T and remove the edge which is the heaviest from $T \cup \{(u, v)\}$. Clearly, removing an edge from a cycle leaves us with a path that still connected all nodes on this cycle so we maintain connectivity. Moreover, the revise T_i is a MST because on every cycle closed by a non-tree edge, that non-tree edge is the heaviest.

  Finding the path connecting u and v on T_i takes $O(|T_i|) = O(n)$ as we must do a graph-traversal (it doesn't matter whether it is DFS or BFS, there's only one path from u to v on a tree), and since this path can have at most $|T_i| - 1$ edges, then finding the heaviest edge on this path and (u, v) takes $O(|T_i|) = O(n)$.

- Suppose an edge (u, v) is removed such that $u.label = v.label$ (the two nodes belong to the same tree), however, the edge doesn't belong to the tree that spans the connected component.

  This is the easiest case: we check that (u, v) isn't a tree edge (by looking at $u.predec$ or $v.predec$) in $O(1)$ time, and if so — we needn't do anything. T_i, the tree that spans all vertices in this connected component, was a min-weight tree out of all trees that spanned the component before the edge was removed. Now that the edge is removed, there are only fewer spanning tree and T_i, which remains intact, still has weight which isn't greater than any of them.

- Suppose an edge (u, v) is removed such that $u.label = v.label$ (the two nodes belong to the same tree), however, the edge *does* belong to the tree that spans the connected component.

  This is the most complicated case. We find a new tree in two steps.

  Step 1: we remove the edge from the tree T_i which spans this connected component, and break this component into two — by doing two BFS traversals on the two trees resulting from removing e from T_i. This takes $O(|T_i|) = O(n)$. Let's denote the two resulting trees as T_i^1 and T_i^2.

  Step 2: we now traverse all edges leaving either T_i^1 or T_i^2 and check whether they connect the two subtrees. If there are no edges connecting the two subtrees — we are done, we revise F by removing T_i and adding the two new trees. If there are such edges, we denote e' as the smallest of these edges and revise T_i to be $T_i^1 \cup T_i^2 \cup \{e'\}$. Finding the min-edge that does connect T_i^1 and T_i^2 takes $O(m)$ time in the worst case (or at least the time it takes to traverse all edges leaving a vertex in T_1^i or a vertex in T_i^2, which, in the worst case, is all m edges). Thus, the total runtime of this fix-up (both steps) is $O(n + m)$.

  The reason why the revised T_i is a MST of its connected component follows from the analysis of Prim: each edge is the min-edge on the cut it induces. (Note, for any edge other than e' we have that the set of edges crossing this cut is the same as before the removal, since all other edges leave u and v on the same side.)

- Lastly, observe that we cannot have a case where we remove an edge (u, v) such that $u.label \neq v.label$: The two nodes were connected by an edge, and therefore must reside in the same connected component in G (before the removal of an edge).

Problem 3. (20 pts) The MAX-BOTTLENECK PATH Problem: Given a weighted undirected graph with non-negative weights, suppose the weights represents the max-capacity of a message sent along an edge. (Alternatively, the edges are roads and the weights are the heights of the bridges above these roads.) Thus, the bottleneck along a path $(v_0, v_1, ..., v_k)$ is $\min_{1\leq i\leq k}\{w(v_{i-1}, v_i)\}$. A max-bottleneck path between u and v is a path whose bottleneck is the largest among all paths connecting u and v.

(a) (10 pts) Suppose we revise `init'()` and `relax'()` to the following functions.

```
init'(s)
foreach $v \in V(G)$ do
    $v.b \leftarrow 0$
$s.b \leftarrow \infty$
```

```
relax'(u,v)
if ($v.b < \min\{u.b, w(u,v)\}$) then
    $v.b \leftarrow \min\{u.b, w(u,v)\}$
```

Show that any algorithm that only accesses the bottle-neck estimation b via `init'()` and `relax'()` always uses *lower bounds* on the bottleneck. That is, show that for every vertex u, at any point of the algorithm it must hold $u.b \leq$bottleneck(s,u). Deduce that once such an algorithm sets $u.b =$bottleneck(s,u) then $u.b$ is never changed from that point on.

Answer. We claim that any algorithm that alters the b attribute using only `init'()` and `relax'()` must have that at any point of the execution and for any node v, we have $v.b \leq$bottleneck(s,v). The proof is by induction on the number of calls to `relax'()`.

In the base case, there are 0 calls to `relax'()`. This means we only have `init'()`, which sets $s.b = \infty$ — and indeed bottleneck$(s,s)= \infty$ as there are no obstacles on the 0-length path; and sets $u.b = 0$ for any u, which are clearly lower bounds on the bottleneck for each vertex u.

Assume the claim holds after the initial k calls for `relax'()`. We show it also holds after the $k+1$-call. Suppose we now invoke `relax'(u,v)`, that means we can only change $v.b$. If we haven't changed it, the claim still holds. If $v.b$ was altered, we now set it to

$$v.b = \min\{u.b, w(u,v)\} \leq \min\{\text{bottleneck}(s,u), w(u,v)\}] \leq \text{bottleneck(specific } s \to v \text{ path)} \leq \text{max-bottleneck}(s,v)$$

and the claim is proven.

An immediate corollary of this claim is that if and when our algorithm sets $v.b =$bottleneck(s,v) then it must not update $v.b$ anymore (as each update only increases $v.b$).

(b) (10 pts) Given a start vertex s, adjust Dijkstra's algorithm to find the max-bottleneck path from s to any other node in $V(G)$. (Use the adjusted `init'()` and `relax'()` functions.) Argue the correctness of your algorithm. What's the runtime of your algorithm?

Answer.

```
procedure bottleneck($G, w, s$)
foreach $v$ in $V(G)$
    $v.b \leftarrow 0$          ** Our Init'()
    $v.predec \leftarrow$NIL
$s.b \leftarrow \infty$
Build PQ $Q$ on $V(G)$ with $key = b$
while ($Q$ isn't empty) do
    $u \leftarrow$ Extract-Max($Q$)
    foreach $v$ neighbor of $u$
        if ($v.b < \min\{u.b, w(u,v)\}$) then
            $v.b \leftarrow \min\{u.b, w(u,v)\}$        ** Our Relax'()
            $v.predec \leftarrow u$
            Increase-Key($Q, v, v.b$)
```

The correctness of the algorithm follows for the same argument as of Dijkstra. We argue that for any $u \notin Q$ we have set $u.b$ =bottleneck(s, u). Initially we take out s, which is set with $s.b = \infty$ so it must be true; and at the end Q is empty, so all the nodes have the right bottleneck estimations. We show that for every u we remove from Q the procedure sets the right $u.b$.

ASOC that this is not the case. Let u be the first vertex taken out of Q for which $u.b <$max-bottleneck(s, u) (recall, $u.b$ is always a lower-bound), and set Q to be the queue just as we are taking out u. Let P be a $s \to u$ path with max-bottleneck, and let (x, y) be the first edge on this path such that $x \notin Q$ and $y \in Q$ (such an edge must exist as $s \notin Q$ and $u \in Q$). Clearly:

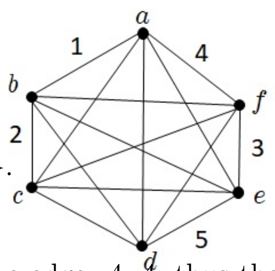
$$\begin{aligned}\text{max-bottleneck}(s,y) &\geq \text{bottleneck}(s \to y \text{ on the path } P)\\ &\geq \text{bottleneck}(s \to u \text{ on the path } P)\\ &= \text{max-bottleneck}(s,u)\end{aligned}$$

Moreover, as x has already been taken out of Q we know $x.b$ =bottleneck(s, x), and that we've invoked `relax'`(x, y), thus we set $y.b \geq \min\{x.b, w(x, y)\}$ $\geq$bottleneck$(s \to y$ on the path $P) \geq$ max-bottleneck(s, u). Thus, $y.b$ $\geq$max-bottleneck$(s, u)>$ $u.b$. So if $y = u$ we get an immediate contradiction, and if $y \neq u$ then `Extract-Max(`Q`)` should have returned y rather than u. Contradiction.

The runtime of the algorithm is just as the runtime of Dijkstra (in the adjacency list model) — $O((n + m) \log(n))$.

Problem 4. (30 pts) Boruvka's MST algorithm: given a n-nodes and m-edges weighted undirected and connected graph G, here's Boruvka's algorithm for finding a MST.

```
procedure Boruvka($G$)
$A \leftarrow \emptyset$          ** $A$ is a set of edges that starts as empty and ends up being the MST.
while ($A$ induces more than one connected-component) do
    $B \leftarrow \emptyset$
    foreach connected-component $C$ formed by $A$ do
        find $e_C \leftarrow$ min-weight edge connecting $C$ with $V \setminus C$
        $B \leftarrow B \cup \{e_C\}$
                    ** Be careful: $e_C$ could have already been added to $B$, you need to avoid duplicates
    $A \leftarrow A \cup B$.
```



(a) (5 pts) Exhibit how Boruvka's algorithm runs on the following example on the right. Show all intermediate steps.
(Any edge whose weight wasn't explicitly stated has weight of 6.)
Answer. We start with 6 singleton components: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}$. We pick each component's respective edge: 1, 1, 2, 5, 3, 3, thus the set B in the first iteration contains the edges: $\{1, 2, 3, 5\}$. Merging those we get two components $\{a, b, c\}, \{d, e, f\}$. We pick each component's respective edge: 4, 4, thus the set B in the second iteration contains the edge $\{4\}$. Merging the two components we end up with a single connected component having chosen the only 5 edges in this graph whose weight is < 6.

(b) (1 pts) Given a subset of the edges A and the set of connected-components it induces $\mathcal{C} = \{C_1, C_2, ..., C_k\}$, let $E(C_i)$ be the set of all edges with one node in C_i and the other in $V \setminus C_i$. Prove that every edge belongs to at most 2 sets $E(C_i)$ and $E(C_j)$. Deduce that no matter how many connected components A induces and regardless of their size, it always holds that $\sum_i |E(C_i)| \leq 2m$.
Answer. Since every edge (u, v) connects two nodes, there can be two cases: either the two nodes belong to the same connected component C_i or to two different components ($u \in C_i, v \in C_j$). In the first case the edge doesn't belong to any of the sets $E(C_i)$; in the latter case the edge belongs to $E(C_i)$ and to $E(C_j)$. Therefore, the summation $\sum_i |E(C_i)|$ can count each edge at most twice, and cannot exceed $2m$.

(c) (4 pts) Argue that in each iteration of the `while`-loop the number of connected components is cut down in at least half. Deduce that we iterate through the `while`-loop at most $\log(n)$ times.
Answer. In each iteration, from every connected-component we pick one edge that leaves this component and ends up at some other component. Each edge can be chosen at most twice — once for each of the two components its two endpoints lie in. Thus, if we have c connected component at the beginning of the ith iteration, then the set B must have at least $c/2$ edges. Each edge causes a merge between its two components, and so the number of connected components decreases by at least $c/2$. Thus, in the next iteration, we can have at most $c/2$ components. Since we start with n components, then within $\log(n)$ iterations we have reduced the number to $\leq \frac{n}{2^{\log(n)}} = 1$ causing us to end the `while`-loop.

(d) (20 pts) Show how to implement Boruvka's algorithm in $O(m \cdot \log(n))$-time. You are free to do it anyway you'd like, however, you're advised to base your implementation using a combination of Union-Find and Priority-Queues (and maybe Lists too).
Hint 1: You want to design your use of the two data-structures so that whenever we check whether the two endpoints of an edge belong to the same connected component, then either we remove the edge and it will be forever disregarded or this edge becomes the first edge in one of the Priority-Queues (and therefore, in the next round...)
Hint 2: Imagine you have t Priority-Queues, each with n_i elements. While there are several ways to merge them into a single Priority-Queue in time $O(n_1 + n_2 + ... + n_t)$, one simple way is to use Array-to-List conversion and List-to-Array conversion. (What's the runtime for merging two lists?)

Answer. First, let's augment the Union-Find data-structure with a function `GetAllRoots()`:

```
procedure UnionFind-GetAllRoots()
$roots$ ←new list of nodes
foreach node $u$ do
   if ($u.predec = u$) then            ** $u$ is a root
      Insert($roots, u$)
return $roots$
```

`GetAllRoots()` returns all the current roots in the Union-Find data-structure in $O(n)$-time.

Secondly, we will have for each node *which is a root in the Union-Find* its own Priority-Queue on the edges that leave its component. Let's denote this as the attribute $u.AE$ (u's Adjacent Edges). We will start with this Priority-Queue as holding all edges in u's adjacency list. We will also convert these into a List (and from a List back into a Priority-Queue), so let's store the list in the field $u.L$.

Thirdly — and it is a very important point — for each edge e that we assign to B, we will check whether the two end-points of e belong to two different components or not. Only if it is not, then we will add the edge to A, merge the two connected components and merge the *two lists* of edges into a single edge, stored at the root of the one merged component. The reason why we do this check is because there could be multiple min-weight edges: suppose from C_1 we picked a min-edge e connecting it to C_2, and from C_2 we picked a different edge e' connecting it to C_1 (that is possible if $w(e) = w(e')$). Now that both edges are in B, the first one will cause us to merge the two components C_1 and C_2, but the second will be ignored — seeing as now, after the two components were merged already, it no longer connects two different components.

```
procedure Boruvka($G$)
foreach $u \in V(G)$ do
   $u.L \leftarrow u$'s adjacency list
   UnionFind-CreateSingleton($u$)
$A$ ←new array of size $n-1$
while ($A.size < n-1$) do
   $roots$ ←UnionFind-GetAllRoots()
   $B \leftarrow$ new list of edges
   foreach ($u$ in $roots$) do
      $u.EA \leftarrow$ PriorityQueue-Initialize($u.L$)
      $e = (x,y)$ ←PriorityQueue-ExtractMin($u.AE$)
      while ( UnionFind-FindRoot($x$)=UnionFind-FindRoot($y$) ) do
         $e = (x,y)$ ←PriorityQueue-ExtractMin($u.AE$)
      Insert($B, e$)
      $u.L \leftarrow$ Build a List from $u.EA$      ** delete the current $u.L$ and then just iterate through
                                                    ** the array $u.EA$ and insert each element into $L$.
   foreach ($e = (u,v)$ in $B$) do
      Insert($A, e$)
      $x$ ←UnionFind-FindRoot($u$)
      $y$ ←UnionFind-FindRoot($v$)
      if ($x \neq y$) then      ** We could have merged the two components already, using a different edge in $B$
         UnionFind-Merge($x, y$)
         $z$ ←UnionFind-FindRoot($x$)      ** either $z$ remains a root in the following iteration
         $z.L$ ←Lists-Merge($x.L, y.L$)     ** or $z$ is further merged with other nodes due to other edges in $B$
```

To analyze the runtime of this algorithm, we first analyze all edge-based operations that utilize the Union-Find / Priority-Queues data-structures. Note that each edge appears at most in two Priority-Queues (which might later be merged, so it may appear twice in the same Priority-Queue), but all in all, in all Priority-Queues together, it appears at most twice. And so each edge might be (1) extracted at most twice from priority queues, (2) added at most twice into B and once into A, (3) responsible for $O(1)$ many `FindRoot()` operations (including those done during a Union-Find `Merge`). We know that (1) takes $O(\log(m)) = O(\log(n))$ because any Priority-Queue has at most $2m$ edges overall; (2) takes obviously $O(1)$; and (3) takes $O(\log(n))$ as it is the worst-case runtime of `FindRoot()` in a Union-Find over n elements (the nodes in the graph). So, per edge — in the entire code execution — we only do $O(\log(n))$-work. Hence, this part of the code (lines 10-13,16-23) takes $O(m\log(n))$.

The rest of the code is composed of the initialization part (lines 1-3), which is clearly done in $O(n+m)$-time; and everything inside the first `while`-loop (lines 5-23) that we have not considered yet. So we show that — except for the part of the codes we already discussed in the last paragraph — the rest of the code in lines 6-23 takes $O(n+m) = O(m)$. Since the loop iterates $\log(n)$-times (due to previous article), then this part of the code will take atmost $O(m\log(n))$ to run as well. Therefore, the entire runtime of the whole code will be $\underbrace{O(m\log(n))}_{\text{dealing with edges}} + \underbrace{O(m\log(n))}_{\text{rest of code}} = O(m\log(n))$.

`GetAllRoots()` takes $O(n)$, and line 7 takes $O(1)$. Lines 09 and 14, where we convert Lists into Priority-Queues and back, both take $O(t)$ time given that the List / Priority-Queue has t elements. Based on the previous article the total number of elements in all lists of all roots at any iteration sums to at most $2m$ (because we never add edges to the Priority-Queues, just extract edges out of the Priority Queues). Therefore, the overall time for creating all of the Priority-Queues / Lists for all roots takes $O(m)$-time. Lastly, `Merge()` for two Lists takes $O(1)$ time — and that is why we converted the Priority-Queues into Lists! Thus, since B has at most $n-1$ edges, then the $|B|$ List-merges take altogether $O(n)$-time. Thus, all the code that doesn't extract edges from the Priority-Queues or find roots in the Union-Find inside the `while`-loop takes $O(n+m)$-time.

If we had stuck to maintaining just Priority-Queues, consider the case where B contains ℓ edges where ℓ is proportional to n (say $\ell = \frac{n}{2}$), so $B = \{(u, v_1), (v_1, v_2), (v_2, v_3), ..., (v_{\ell-1}, v_\ell)\}$ and furthermore we have that $u.EA$ contains linear number of edges (say $\frac{m}{4}$ edges) whereas and for each v_i we have that $v_i.EA$ contains only a constant number of edges. Naïvely merging these arrays would mean that the numerous elements that were in $u.EA$ would be copied repeatedly ℓ times, and we would have a runtime of $\frac{m}{4} \cdot l = O(mn)$. By converting all Priority-Queues into Lists first and then doing the merging we have that the runtime of all those merges is $O(m) + O(\ell) = O(m+n)$. But this is also because we relied solely on pairwise merging, we could have, alternatively, chosen to first traverse all edges in B and infer what set of Priority-Queues is going to get merged into a single Priority-Queue and then write a generic function that merges all of those together — but finding those would be done using constructing a graph whose edge-set is B and doing a graph traversal, and then writing the code for merging a set of Priority-Queues (neither task is hard, just tedious).