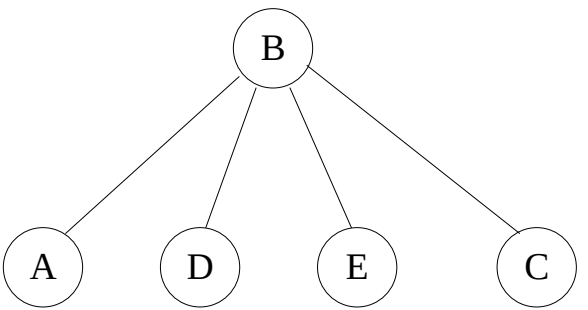
(c) In what order are the edges added in Kruskal's algorithm?

- (B,D); (D,E); (A,B); (B,C)

(d) Show the union-find trees at the end of Kruskal's algorithm (in case of a tie in rank, the lexicograpfically first node becomes root).

- Union(B,D) puts B as the parent (by the lexicographical constraint). Union(D,E) links E to B since Find(D) returns B. Union(A,B) and Union(B,C) also link A and C to B. This yields:

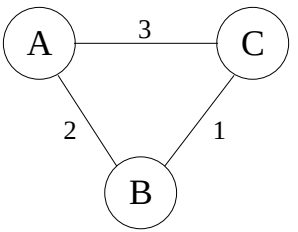


increasing order. Consider the first edge $e_i$ that belongs to $T_1$ and not to $T_2$, and the first edge $e_j$ that belongs to $T_2$ and not to $T_1$. Suppose, without loss of generality, that $e_i$ is shorter than $e_j$. Adding $e_i$ to $T_2$ creates a unique cycle in $T_1$. There must be some edge $e_k$ in the cycle such that $k > i$, since otherwise $T_1$ would have a cycle. But in this case $w(e_k) > w(e_i)$ so we can remove $e_k$ and get a lighter tree $T = T_2 \cup \{e_i\} - \{e_k\}$. So $T_2$ was not an MST.

Again, there are a number of ways to prove this. However, it is not sufficient to simply say that Kruskal's algorithm only finds one MST when the edge weights are distinct.

4. (*3 points)* Suppose all edge weights are different. Then the shortest path from A to B is unique.

   ▶ **False**. Two disjoint sets of integers can easily sum up to the same number. A simple counter-example is the following graph (for shortest paths from A to C):




5. (*3 points)* There is an efficient algorithm for finding the longest (highest length) path in a dag.

   ▶ **True**. Negate all the edges. The shortest path in the resulting graph is the longest path in the original graph. Since the graph is a DAG, there can be no negative cycles in the resulting graph. Thus, all we need to do is run Bellman-Ford on the graph with negated edges. This yields a reasonably efficient $O(VE)$ running time.

8. (*3 points)* Any operation in the union-find data structure takes at most $O(\log n)$ time.

   ▶ **True**. Every operation in the union-find data structure takes time at most a constant factor times the running time for the find operation. The find operation's running time is $O(h)$, where $h$ is the height of the tree it searches, since find($x$) begins at element $x$ and follows the path to $x$'s parent, and then to its grandparent, and so on, until it reaches the root. Since we make unions carefully, by always making the root node of lesser rank point to the root node of greater rank, the height of any tree is bounded by $\log n$.

9. (*3 points)* Even without path compression, any operation in the union-find data structure takes at most $O(\log n)$ time.

   ▶ **True**. The argument in part (8) did not rely on path compression. Path compression only makes the path that the find operation follows shorter.

10. (*3 points)* Any operation in the union-find data structure takes at most $O(\log^* n)$ time.

    ▶ **False**. This is the amortized cost, not an upper bound on the cost of any operation. A single operation, even with path compression, could take as long as $O(\log n)$.

```
changekey(v,dist(v))
```

The runtime will be the same as dijkstra's $O(V \log V + E)$ since in either case, we are only making a constant change per iteration of the inner loop.

We may argue correctness by considering, for any destination $t$, the shortest path that has the minimum num of hops. We want to say that when we pop a node along this path from the heap, that it has the same $dist(v)$ and $hops(v)$ values as it does on this path. We show this inductively: it is certainly true for $s$. Now assume that we pick a point on the path. We know by the inductive hypothesis, that the node before it on this path, $u$, had the correct $dist(u)$ and $hops(u)$ values when popped from the heap. This means that $dist(v)$ must be at most $dist(u) + l(u, v)$ and $hops(v)$ at most $hops(u) + 1$. They also are at least these values, or else there would be a better path than the one we were considering. Now we must show that v has not been popped from the heap already. But this follows for the same reason that it was true in the original dijkstra's algorithm (see lecture notes for the argument).

Note: many students didn't change the condition in the 'if' statement, and just put $hops(v) = hops(u) + 1$. This will end up returning the length of the shortest path that you would find in the original algorithm which may not be the shortest one.

(b) This was a tough one. No one got this completely right. Many people got the basic gist: add a very small value to each edge to bias toward shorter paths. Yes, this means that if two paths were of equal weight before, then the shorter one ill be chosen in the new graph. However, you have to show that the increments are small enough so that you don't get a new shortest path that just happened to have a very small amount of extra weight from the old one, but was dramatically shorter. Take the following example:

$\texttt{maxsuff}[i+1] =$

- $\max\{\texttt{maxsuff}[i] + a_{i+1}, 0\}$

(c) What other data structure do you need in order to recover in the end the beginning and end of the maximum contiguous sum?

- The most common approach was to use two pointers to keep track of the beginning and the end of the best sum seen so far. However, one also needs a pointer to keep track of the beginning of the best suffix in order to be able to update the two pointers properly.

  It is possible to find the beginning and the end without keeping track of any new information. Once we are done, we can search the `maxsuff` array for the element that equals $\texttt{maxsum}[n]$. This marks the end of the desired sum. Then we can start at this point in the original sequence and proceed backwards, adding terms until we reach the desired sum.