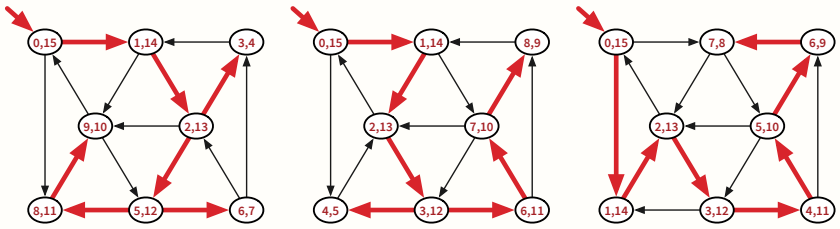
CS/ECE 374 A ✧ Spring 2018
Midterm 2◇ Problem 1

Clearly indicate the following structures in the directed graph below, or write NONE if the indicated structure does not exist. Don't be subtle; to indicate a collection of edges, draw a heavy black line along the entire length of each edge.

(a) A depth-first tree rooted at x.

Solution: There are several possibilities.



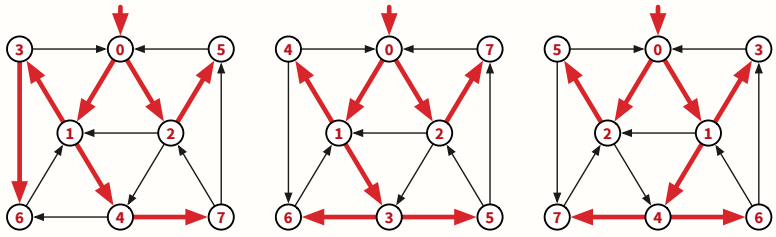
These are ***not*** the only correct answers! Only one correct tree is required for full credit, obviously. The numbers in the vertices show a preorder and postorder labeling consistent with each tree; these are ***not*** required for credit. ■

Rubric: 2½ points:

- No credit if the indicated graph is not a spanning tree of G.
- −1 for each misplaced (or misdirected) edge, compared to the closest correct tree.

(b) A breadth-first tree rooted at y.

Solution: There are several possibilities.



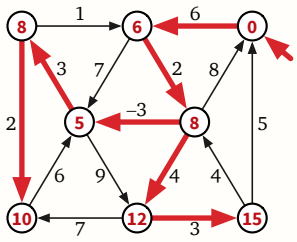
These are the only correct answers. Only one correct tree is required for full credit, obviously. The numbers in the vertices indicate the order they enter and leave the BFS queue; these are ***not*** required for credit. (The queue numbering for the first tree is unique, but not for the second and third trees.) ■

Rubric: 2½ points:

- No credit if the indicated graph is not a spanning tree of G.
- −1 for each misplaced (or misdirected) edge, compared to the closest correct tree.

(c) A shortest-path tree rooted at z.

Solution:



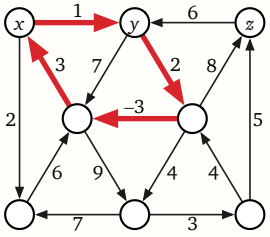
This is the only correct answer. The numbers in the nodes are shortest-path distances from z; these are ***not*** required for credit. ■

Rubric: 2½ points:

- No credit if the indicated graph is not a spanning tree of G
- −1 for each misplaced (or misdirected) edge

(d) The shortest directed cycle.

Solution:



This is the only correct answer. ■

Rubric: 2½ points:

- No credit if the indicated graph is not a directed cycle in G
- −2 for the wrong cycle

CS/ECE 374 A ✧ Spring 2018
Midterm 2♢ Problem 2

Suppose you are given a directed graph G where some edges are red and the remaining edges are blue. Describe an algorithm to find the shortest walk in G from one vertex s to another vertex t in which no three consecutive edges have the same color. (See the question handout for an example.)

Solution: We reduce to a shortest-path problem in a new graph $G' = (V', E')$, defined using the following "product" construction.

- $V' = V \times \{0, r1, r2, b1, b2\}$. Each pair (v, k) indicates the current vertex and the number of consecutive red or blue edges just traversed.
- E' contains the edges

$$(v,0)\to(w,r1), \quad (v,r1)\to(w,r2), \quad (v,b1)\to(w,r1), \quad (v,b2)\to(w,r2)$$

  for each red edge $v\to w \in E$, and the edges

$$(v,0)\to(w,b1), \quad (v,r1)\to(w,b1), \quad (v,r2)\to(w,b1), \quad (v,b1)\to(w,b2)$$

  for each blue edge $v\to w \in E$.
- We need to compute the shortest path in G' from $(s,0)$ to any vertex of the form $(t,\cdot)$.
- We can compute all such shortest paths using breadth-first search, and then return the shortest of these paths.
- The resulting algorithm runs in $O(V' + E') = \boldsymbol{O(V + E)}$ ***time***.

■

Rubric: 10 points: standard graph reduction rubric

CS/ECE 374 A ✧ Spring 2018
Midterm 2♢ Problem 3

Let G be an arbitrary (*not* necessarily acyclic) directed graph in which every vertex v has a distinct integer label $\ell(v)$. Describe an algorithm to find the longest directed path in G whose vertex labels define an increasing sequence. (See the question handout for an example.)

Solution: First delete any edge $u \to v$ where $h(u) \geq h(v)$, since no such edge can appear in the path we are searching for. The remaining subgraph H is a dag; specifically, sorting the vertices by their labels defines a topological order. (However, our algorithm will *not* sort the vertices by their labels, because that would take too long.) Topologically sort H, because that's how we do.

For any vertex v, let $LIP(v)$ denote the length of the *L*ongest *I*ncreasing *P*ath in H that starts at v. We need to compute $\max_v LIP(v)$. This function satisfies the following recurrence, where for convenience we define $\max \varnothing = 0$:

$$LIP(v) = \max\{1 + LIP(w) \mid v \to w \in H\}$$

We can memoize this function into the graph itself, by recording $LIP(v)$ in a new field $v.LIP$ at each vertex v. (Equivalently, if vertices are represented as usual by integers between 1 and V, we can memoize into an array $LIP[1..V]$.)

We can fill the memoization structure in reverse topological order in $O(V + E)$ time. (Alternatively, we can compute $LIP(v)$ for all v in postorder via depth-first search.) Finally, we compute $\max_v LIP(v)$ in $O(V)$ time. The overall running time of our algorithm is $\boldsymbol{O(V + E)}$. ■

Solution (smartass): Delete every edge $u \to v$ where $h(u) \geq h(v)$, because these edges are useless. Add a new source vertex s, with edges to every other vertex, and a new target vertex t, with edges from every other vertex. We need to compute the longest path from s to t in the resulting dag. Jeff did this in class in $\boldsymbol{O(V + E)}$ ***time***. ■

Rubric: 10 points: 2 for deleting downward edges + 8 for the dynamic programming algorithm. Yes, the smartass solution is worth full credit, but only because we added the source and target vertices.

CS/ECE 374 A ✧ Spring 2018
Midterm 2◇ Problem 4

Suppose you have an integer array $A[1..n]$ that *used* to be sorted, but k entries of A have been overwritten with random numbers. Describe an algorithm to determine whether your corrupted array A contains a given integer x. Your input consists of the array A, the integer k, and the target integer x.

Solution: When $k = 0$, we can clearly solve the problem in $O(\log n)$ time using binary search. For larger values of k, we use a modification of this standard algorithm.

First consider the special case $k = 1$. At most one of any three consecutive entries $A[i-1], A[i], A[i+1]$ in the input array is corrupted, and therefore at most one of the two consecutive pairs is out of order. There are three cases to consider:

- If $A[i-1] > A[i]$, then either $A[i-1]$ or $A[i]$ is corrupted, and therefore $A[i+1]$ is not corrupted. Thus, we can still use $A[i+1]$ to guide a binary search for x.
- If $A[i] > A[i+1]$, then either $A[i]$ or $A[i+1]$ is corrupted, and therefore $A[i-1]$ is not corrupted. Thus, we can still use $A[i+1]$ to guide a binary search for x.
- Finally, if $A[i-1] < A[i] < A[i+1]$, then we can use $A[i]$ to guide a binary search for x ***even if*** $\boldsymbol{A[i]}$ ***is corrupted***. Specifically, if $x < A[i]$, then x cannot lie in the suffix $A[i..n]$, and if $x > A[i]$, then x cannot lie in the prefix $A[1..i]$.

We conclude that the following recursive algorithm correctly searches for x. The algorithm treats the input array $A[1..n]$ as a global variable; the top-level call is Find1Corrupt$(1, n, x)$. The new red lines are the only difference from a standard binary search.

```
Find1Corrupt(lo, hi, x):
  if hi − lo < 100
      use brute force
  else
      m ← ⌈(lo + hi)/2⌉
      if A[m] < A[m − 1]
          m ← m + 1
      else if A[m] > A[m + 1]
          m ← m − 1
      if x < A[m]
          return Find1Corrupt(lo, m − 1, x)
      else if x > A[m]
          return Find1Corrupt(m + 1, hi, x)
      else
          return True
```

The running time of this algorithm satisfies the usual binary-search recurrence $T(n) \le T(n/2) + O(1)$. Thus, the algorithm runs in $\boldsymbol{O(\log n)}$ ***time***.

To generalize this idea to arbitrary k, we consider a window of $2k+1$ consecutive entries $A[m-k..m+k]$. If any entry in this window is equal to x, we immediately return True. Otherwise, we use the *median* element of the window (computed using the linear-time selection algorithm described in class) to guide the binary search.

```
FindkCorrupt(lo, hi, x):
  if hi − lo < 100k
      use brute force
  m ← ⌈(lo + hi)/2⌉
  for i ← m − k to m + k
      if A[i] = x
          return True
  med ← Median(A[m − k .. m + k])
  if x < med
      return FindkCorrupt(lo, m − k − 1, x)
  else
      return FindkCorrupt(m + k + 1, hi, x)
```

We argue that this algorithm is correct, ***even when the median element of the window is corrupted***, as follows. If x is not in the input array, the algorithm correctly returns False. If x lies in the window $A[m-k\,..\,m+k]$, the algorithm correctly returns True. Without loss of generality, suppose x lies in the prefix $A[lo\,..\,m-k-1]$; the other remaining case is symmetric. The median of the window is greater than or equal to $k+1$ entries in the window (by definition of "median"), so at least one of these entries is not corrupted. For any such entry $A[i]$, we have $x < A[i] \le med$, so the algorithm correctly recurses in the prefix.

The running time of this algorithm satisfies the following recurrence:

$$T(n) \le \begin{cases} O(k) & \text{if } n < 100k \\ T(n/2) + O(k) & \text{otherwise} \end{cases}$$

The recursion "tree" for this recurrence is just a path, with one node with value $O(k)$ at every level, and the number of levels is $\log_2(n/100k) = O(\log(n/k))$. Thus, the algorithm runs in $\boldsymbol{O(k \log(n/k))}$ ***time***. ■

Solution: We use a modification of binary search (the obvious algorithm when $k = 0$):

In each step of the modified binary search, instead of comparing x with a single entry $A[m]$, we consider a window of $2k+1$ consecutive entries $A[m-k\,..\,m+k]$. If any entry in this window is equal to x, we can immediately return True. Otherwise, we exploit the fact that at most k entries *in this window* are corrupt, as follows.

- If more than k entries in $A[m-k\,..\,m+k]$ are smaller than x, at least one *uncorrupted* entry in $A[m-k\,..\,m+k]$ is smaller than x, which implies that x cannot lie in the prefix $A[1\,..\,m+k]$.
- Similarly, if more than k entries in $A[m-k\,..\,m+k]$ are larger than x, at least one *uncorrupted* entry in $A[m-k\,..\,m+k]$ is larger than x, which implies that x cannot lie in the suffix $A[m-k\,..\,n]$.

So at each iteration of our binary search, we count the number of entries smaller and larger than x in a window of size $2k+1$, and (unless we find x in that window) recurse on one side or the other.

```
FindkCorrupt(lo, hi, x):
  if hi − lo < 100k
      use brute force
  m ← ⌈(lo + hi)/2⌉
  less ← 0
  more ← 0
  for i ← m − k to m + k
      if x < A[i]
          less ← less + 1
      else if x > A[i]
          more ← more + 1
      else ⟨⟨x = A[i]⟩⟩
          return True
  if less > more
      return FindkCorrupt(lo, m − k − 1, x)
  else
      return FindkCorrupt(m + k + 1, hi, x)
```

Just like the previous solution, this algorithm runs in $O(k \log(n/k))$ *time*. ■

Rubric: Max 10 points. These are not the only correct solutions. Both solutions are *considerably* more detailed than necessary for full credit.

+ 2 for base case. Yes, "If $n = O(k)$ then brute force" is enough.
  - −1 for "if $n = O(1)$ then brute force", unless the resulting boundary issues for large k are addressed some other way, for example, by adding a buffer of k $-\infty$s to the start of the array and a buffer of k ∞s to the end of the array.
+ 3 for "binary search using a window of size $2k + 1$ for each comparison"
  - – no penalty for window of size $O(k)$ but greater than $2k + 1$
  - −1 for window of size $2k$.
  - −2 for window of size $O(k)$ with no explicit constant
  - −3 for window of size ck for some $c < 2$.
+ 1 for finding x in the window
+ 2 for correctly finding x outside the window
+ 2 for time analysis
  - −½ for $O(k \log n)$ instead of $O(k \log(n/k))$.

- Proof of correctness is not required.
- Detailed pseudocode is not required.
- Regurgitating binary search is not required.

- max 5 points for $O(\log n)$-time solution for $k = 1$; scale partial credit.
- max 3 points for brute-force $O(n)$ time.

CS/ECE 374 A ✧ Spring 2018
Midterm 2♢ Problem 5

Describe and analyze an algorithm to determine, given a weighted undirected graph G and a spanning tree T of G, whether T is in fact a *shortest-path* tree in G. Assume all edge weights are non-negative.

Solution: At the top level, for every vertex s in the graph, we check whether T is a shortest-path tree rooted at s as follows.

First set $dist(s) \leftarrow 0$. Perform a whatever-first search of T, starting at s, to assign a parent to every vertex, effectively directing the edges of T away from s. Whenever the search assigns $parent(v) \leftarrow p$, we also set $dist(v) \leftarrow dist(p) + \ell(pv)$. When the search concludes, $dist(v)$ is the length of the unique path in T from s to v. This part of the algorithm requires $O(V)$ time, because T has exactly $V-1$ edges.

After all these distances are computed, we check for tense edges in G (not just in T) by brute force. An edge uv is tense if either $dist(v) > dist(u) + \ell(uv)$ or $dist(u) > dist(v) + \ell(uv)$. If there are no tense edges in G, then T is a shortest path tree rooted at s, so we return TRUE. On the other hand, if we discover a tense edge, we can immediately reject s as a possible source. This part of the algorithm requires $O(E)$ time.

Overall, deciding whether T is a shortest-path tree *rooted at* s takes $O(V+E)$ time. Because we are given a spanning tree, we know that G is connected, so we can simplify this time bound to $O(E)$.

If we reject every vertex of G as a possible source, we return FALSE. The entire algorithm runs in $\boldsymbol{O(VE)}$ ***time***. ■

Solution (8/10): Fix a potential source vertex s. For each vertex v, we define two distances:

- $dist_G(v)$ is the length of the shortest path from s to v in G
- $dist_T(v)$ is the length of the *unique* path from s to v in T

T is a shortest-path tree rooted at s if and only if $dist_G(v) = dist_T(v)$ for every vertex v. We can compute both of these distances for all v in $O(E \log V)$ time using Disjktra's algorithm, and then check for equality in $O(V)$ time.

By repeating the previous algorithm for all vertices s, we can determine whether T is a shortest-path tree in $\boldsymbol{O(VE \log V)}$ ***time***. ■

Rubric: 10 points
- \+ 2 for considering all possible source vertices
- \+ 4 for correctly computing distances in T from a fixed source in $O(V)$ time
  - −½ for writing "depth" or "breadth" instead of "whatever"
  - −2 for computing distances using Dijkstra instead of WFS
- \+ 2 for correctly identifying tense edges in $O(E)$ time.
- \+ 2 for time analysis ("$O(V^2 + VE)$" is fine.)

- Max 8 points for $O(VE \log V)$ time.
- Max 7 points for $O(V^3)$ time (Floyd-Warshall)
- These are not the only correct solutions with these running times.