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.

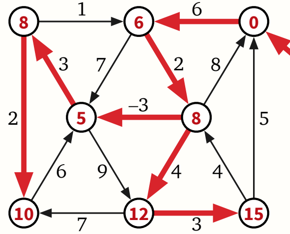
These are the only correct answers. Only one correct tree is required for full credit, obviously. The numbers in the vertices indicate the order they enter and leave the BFS queue; these are ***not*** required for credit. (The queue numbering for the first tree is unique, but not for the second and third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



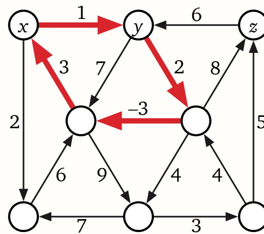
This is the only correct answer. The numbers in the nodes are shortest-path distances from z; these are ***not*** required for credit. ■

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 an ***undirected*** graph $G = (V, E)$, where each vertex $v \in V$ has a height $h(v)$ and each edge $e \in E$ has a positive length $\ell(e)$, along with a start vertex s, a target vertex t, and a maximum length L. Describe and analyze an algorithm to compute the maximum height reachable by a walk from s to t with total length at most L.

Solution: We first compute shortest-path distances $dist(s, v)$ and $dist(t, v)$ for every vertex v using Dijkstra's algorithm. There is a walk from s to t with length at most L that visits another vertex v if and only if $dist(s, v) + dist(t, v) \le L$. We find the maximum height among all such vertices by brute force.

```
MaxReachableHeight(V, E, h, ℓ, s, t, L):
  dist(s, ·) ← Dijkstra(V, E, ℓ, s)
  dist(t, ·) ← Dijkstra(V, E, ℓ, t)
  best ← −∞
  for every vertex v
      if dist(s, v) + dist(t, v) ≤ L
          best ← max{best, h(v)}
  return best
```

The algorithm runs in $O(E \log V)$ ***time***. ■

Rubric: 10 points:

+ 2 for recognizing this as a shortest path problem!
+ 4 for computing $dist(s, v) + dist(t, v)$ at every vertex v.
+ 2 for other algorithm details
+ 2 for running time

- Max 7 points for $O(VE)$ time (Bellman-Ford instead of Dijkstra)
- Max 7 points for $O(VE \log V)$ time (Dijkstra from every vertex)
- Max 5 points for $O(V^3)$ time (Floyd-Warshall instead of Dijkstra)

- Max 2 points for finding the maximum height $h(v)$ such that $dist(s, v) \le L/2$ and $dist(v, t) \le L/2$
- Max 2 points for returning the maximum height along the shortest path from s to t

− 1 for regurgitating Dijkstra's algorithm (or Bellman-Ford, or Floyd-Warshall, or whatever) instead of just writing "Dijkstra's algorithm" (or "Bellman-Ford", or "Floyd-Warshall", or "whatever")

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

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 $A[i]$ 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 4

Let G be a ***directed*** graph, where every vertex v has an associated height $h(v)$, and every edge $u \to v$ satisfies the inequality $h(u) > h(v)$. Describe and analyze an algorithm to find the maximum span of a path in G with at most k edges, given the graph G and the integer k as input. Report the running time of your algorithm as a function of ***V, E,*** and k.

Solution (dynamic programming): The input graph G happens to a dag—sorting the vertices by decreasing height gives us a topological order—but this is not actually important!

- For any vertex v and any integer $i \geq 0$, let $\boldsymbol{Lowest(v, i)}$ denote the minimum height among all vertices reachable from v by a path of length at most i.
- The maximum span of a path with at most k edges that starts at vertex v is $h(v) - Lowest(v, k)$. Thus, we need to compute $\min\{h(v) - Lowest(v, k)) \mid v \in V\}$.
- The *Lowest* function satisfies the following recurrence (assuming as usual $\min \varnothing = \infty$):

$$Lowest(v, i) = \begin{cases} h(v) & \text{if } i = 0 \\ \min \left\{ \begin{array}{c} Lowest(v, i-1) \\ \min\{Lowest(w, i-1) \mid v \to w \in E\} \end{array} \right\} & \text{otherwise} \end{cases}$$

- We can memoize this recurrence into an array $v.Lowest[0..k]$ at each vertex. (Equivalently, if vertices are identified by integers between 1 and V, we can memoize into a two-dimensional array $Lowest[1..V, 0..k]$.)
- We can fill the memoization structure in $O(kE)$ time by decreasing i in the outer loop and considering v in arbitrary order in the inner loop, or (because G is a dag) considering v in postorder in the outer loop and i in arbitrary order in the inner loop.
- The resulting algorithm runs in $\boldsymbol{O(k(V+E))}$ ***time***.

```
MaxSpan(V, E, h, k):
  for all vertices v
      v.Lowest[0] ← h(v)
  for i ← 1 to k
      for all vertices v
          v.Lowest[i] ← v.Lowest[i − 1]
          for all edges v→w
              v.Lowest[i] ← min{v.Lowest[i], w.Lowest[i − 1]}
  maxspan ← 0
  for all vertices v
      maxspan ← max{maxspan, v.h − v.Lowest[k]}
  return maxspan
```

If we choose the evaluation order more carefully, we only need to maintain a single integer $v.lowest$ at each vertex. In particular, the inner loop *must* consider vertices in *forward* topological order; otherwise, we could inadvertently consider paths with too many edges.

```
MaxSpan(V, E, h, k):
  maxspan ← 0
  for all vertices v
      v.lowest ← h(v)
  for i ← 1 to k
      for all vertices v in topological order
          for all edges v→w
              v.lowest ← min{v.lowest, w.lowest}
          maxspan ← max{maxspan, h(v) − v.lowest}
  return maxspan
```

Rubric: 10 points: standard dynamic programming rubric. This is not the only correct solution. Don't be picky about notation, but do be picky about evaluation order.

- No penalty for reporting $O(kE)$ running time (implicitly assuming G is connected)
- At most 7 points for an algorithm that runs in $O(VE)$ time even when k is small, for example, running k levels of breadth-first search from each vertex v.

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

Suppose you are given a directed graph G where some edges are red and the remaining edges are blue, along with two vertices s and t. Describe an algorithm to compute the length of the shortest walk in G from s to t that traverses an even number of red edges and an even number of blue edges.

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

- $V' = V \times \{0,1\} \times \{0,1\}$. Each triple (v, r, b) indicates the current vertex, the number of red edges traversed mod 2, and the number of blue edges traversed mod 2.
- $E' = \{(v,r,b) \rightarrow (w, 1-r, b) \mid v \rightarrow w \text{ is red}\} \cup \{(v,r,b) \rightarrow (w, r, 1-b) \mid v \rightarrow w \text{ is blue}\}$.
- We need to compute the shortest path in G' from $(s, 0, 0)$ to $(t, 0, 0)$.
- We can compute this shortest path using breadth-first search.
- The resulting algorithm runs in $O(V' + E') = \boldsymbol{O(V + E)}$ ***time***.

Rubric: 10 points: standard graph reduction rubric