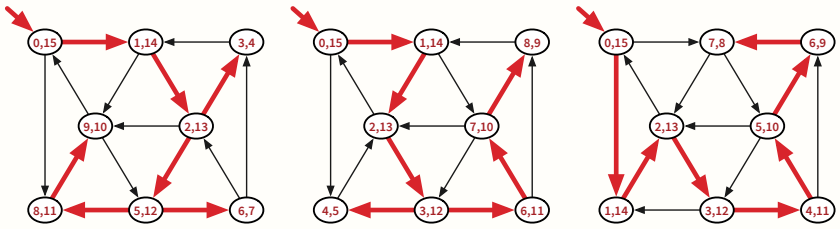
CS/ECE 374 A ✧ Spring 2018
Midterm 2♡ Problem 1

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.



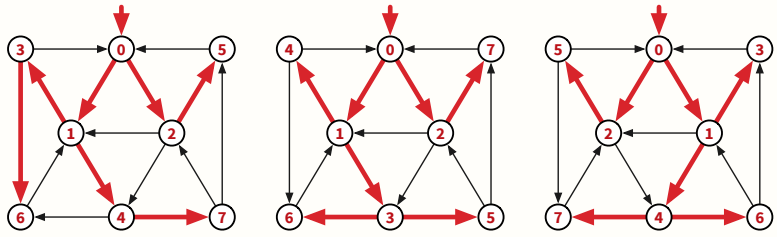
These are ***not*** the only correct answers! Only one correct tree is required for full credit, obviously. The numbers in the vertices show a preorder and postorder labeling consistent with each tree; these are ***not*** required for credit. ■

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.



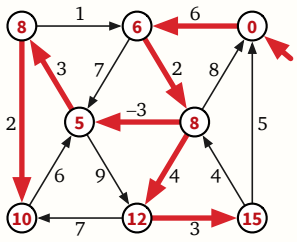
These are the only correct answers. Only one correct tree is required for full credit, obviously. The numbers in the vertices indicate the order they enter and leave the BFS queue; these are ***not*** required for credit. (The queue numbering for the first tree is unique, but not for the second and third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



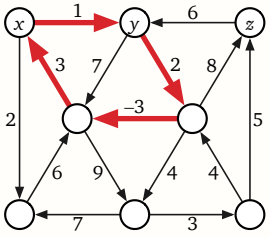
This is the only correct answer. The numbers in the nodes are shortest-path distances from z; these are ***not*** required for credit. ■

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

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 minimum span of a path in G with at least 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! Extending a path increases its span, so we don't actually need to consider paths with *more* than k edges.

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

$$\mathit{Highest}(v, i) = \begin{cases} h(v) & \text{if } i = 0 \\ \max\{\mathit{Highest}(w, i-1) \mid v \to w \in E\} & \text{otherwise} \end{cases}$$

- We can memoize this recurrence into an array $v.\mathit{Highest}[0..k]$ at each vertex. (Equivalently, if vertices are identified by integers between 1 and V, we can memoize into a two-dimensional array $\mathit{Highest}[1..V, 0..k]$.)
- 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 ***$O(k(V+E))$ time***.

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

If we choose the evaluation order more carefully, we only need to maintain a single integer $v.highest$ at each vertex. In particular, the inner loop *must* consider vertices in *forward* topological order; otherwise, we could inadvertently skip over some path lengths.

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

■

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 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*** $\boldsymbol{A[i]}$ ***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 4

Suppose you are given a directed graph G where some edges are red and the remaining edges are blue, and some vertices are marked *special*, 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 changes color only at special vertices.

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

- $V' = V \times \{r, b\}$. Each pair (v, c) indicates the current vertex and the current edge color.
- E' is the union of three sets:
  - red edges (Beszél): $\{(v, r) \to (w, r) \mid v \to w \text{ is red}\}$
  - blue edges (Ul Qoma): $\{(v, b) \to (w, b) \mid v \to w \text{ is blue}\}$
  - special edges (Copula Hall): $\{(v, r) \to (w, b) \mid v \text{ is special and } v \to w \text{ is blue}\} \cup \{(v, b) \to (w, r) \mid v \text{ is special and } v \to w \text{ is red}\}$
- We need to compute the shortest path in G' from either (s, r) or (s, b) to either (t, r) of (t, b).
- We can compute each of the four candidate shortest paths using breadth-first search. Then we return the shortest of these four candidate paths.
- The resulting algorithm runs in $O(V' + E') = \boldsymbol{O(V + E)}$ ***time***.

■

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

- $V' = V \times \{r, b\}$. Each pair (v, c) indicates the current vertex and the current edge color.
- E' is the union of three sets:
  - red edges (Beszél): $\{(v, r) \to (w, r) \mid v \to w \text{ is red}\}$
  - blue edges (Ul Qoma): $\{(v, b) \to (w, b) \mid v \to w \text{ is blue}\}$
  - special edges (Copula Hall): $\{(v, r) \to (v, b), (v, b) \to (v, r) \mid v \text{ is special}\}$
- Red and blue edges have weight 1; special edges have weight 0.
- We need to compute the shortest path in G' from either (s, r) or (s, b) to either (t, r) of (t, b).

- We can compute each of the four candidate shortest paths using the following modification of breadth-first search. Whenever we pull a vertex $(v, \cdot)$ from the queue, where v is special, we immediately visit both (v, r) and (v, b). (Alternatively, we can run the ZEROONEDIJKSTRA algorithm from the HW8 solutions.) Then we return the shortest of these four candidate paths.
- The resulting algorithm runs in $O(V' + E') = \boldsymbol{O(V + E)}$ ***time***.

■

Rubric: 10 points: standard graph reduction rubric. These are not the only correct solution.

- −1 for using Dijkstra (in $O(E \log V)$ time) instead of breadth-first search
- −1 for regurgitating breadth-first search (or Dijkstra or whatever) instead of just writing "breadth-first search" (or "Dijkstra" or "whatever")
- −3 for using the second graph construction, but using off-the-shelf BFS (or equivalently, giving the special edges weight 1)

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

Let G be a directed graph with weighted edges, in which every vertex is colored either red, green, or blue. Describe and analyze an algorithm to compute the length of the shortest walk in G that starts at a red vertex, visits a green vertex, then visits a blue vertex, and finally ends at a red vertex. Assume all edge weights are positive.

Solution: Intuitively, we divide the walk into three *stages*. The first stage is a path from a red vertex to a green vertex; the second is a path from a green vertex to a blue vertex; and the third is a path from a blue vertex to a red vertex.

We reduce to a shortest-path problem in a new graph $G' = (V', E')$, defined as follows.

- $V' = V \times \{1, 2, 3\} \cup \{s, t\}$. Each pair (v, i) indicates the current vertex and the current stage.
- E' is the union of four sets:
  - Normal edges: $\{(u, i) \to (v, i) \mid u \to v \in E \text{ and } i \in \{1, 2, 3\}\}$
  - Start edges: $\{s \to (v, 1) \mid v \text{ is red}\}$
  - Green transition edges: $\{(v, 1) \to (v, 2) \mid v \text{ is green}\}$
  - Blue transition edges: $\{(v, 2) \to (v, 3) \mid v \text{ is blue}\}$
  - End edges: $\{(v, 2) \to t \mid v \text{ is red}\}$
- Each normal edge $(u, i) \to (v, i)$ has weight $u \to v$. All other edges have weight 0.
- We need to compute a shortest path in G' from s to t.
- We can compute this shortest path using Dijkstra's algorithm.
- The resulting algorithm runs in $O(E' \log V') = \boldsymbol{O(E \log V)}$ ***time***.

■

Rubric: 10 points: standard graph reduction rubric. This is not the only correct solution.

- Max 7 points for $O(EV)$ time (Bellman-Ford instead of Dijkstra)
- Max 7 points for $O(EV \log V)$ time (Dijkstra from every red vertex)
- Max 5 points for $O(V^3)$ time (Floyd-Warshall).
- Any correct and correctly analyzed algorithm is worth at least 3 points.
- −1 for regurgitating Dijkstra (or Bellman-Ford or Floyd-Warshall or whatever) instead of just writing "Dijkstra" (or "Bellman-Ford" or "Floyd-Warshall" or "whatever")