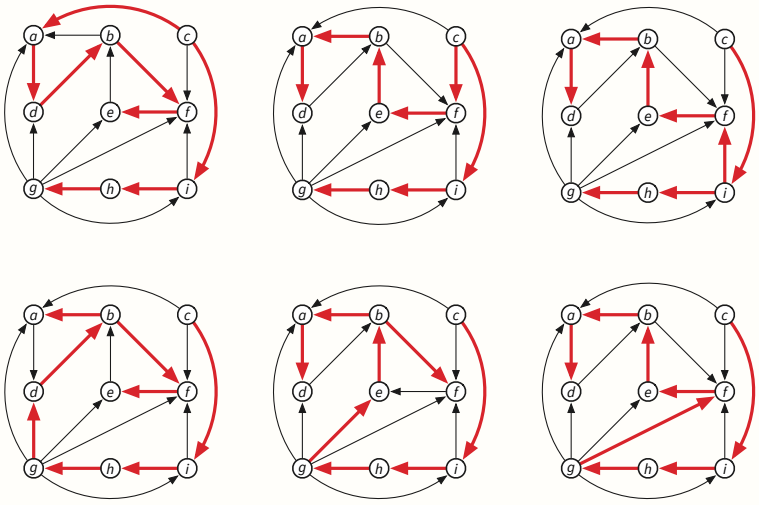
CS/ECE 374 A ✧ Fall 2019
Midterm 2$^{\otimes}$ Problem 1 Solution

Clearly indicate the following structures in the directed graph G...

(a) A depth-first search tree rooted at c.

Solution: There are six correct solutions:



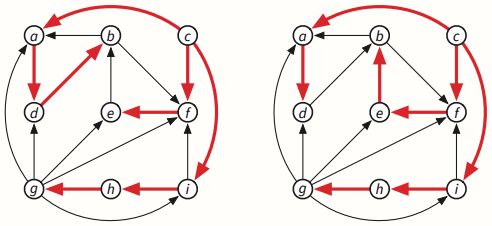
■

Rubric: 2½ points.

- No credit if the reported graph is not a spanning tree rooted at vertex a.
- −1 for each misplaced edge, compared to the closest correct solution

(b) A breadth-first search tree rooted at c.

Solution: There are two correct solutions:



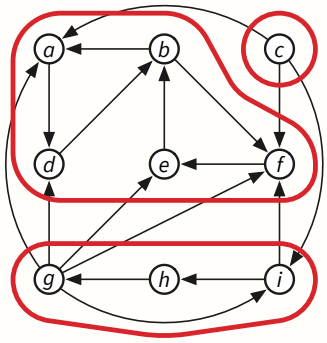
■

Rubric: 2½ points

- No credit if the reported graph is not a spanning tree rooted at vertex a.
- −1 for each misplaced edge

(c) Circle each strong component.

Solution:



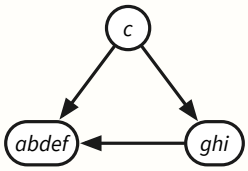
■

Rubric: 2½ points

- −½ for not circling the individual vertex c.
- No credit if circled subgraphs omit any vertex other than c.
- −1 for circling subgraphs abd and bef, either instead of or in addition to $abdef$.
- No credit if any circled subgraph is not strongly connected.
- No penalty if circles include only the vertices.

(d) Draw the strong-component graph of the example graph G.

Solution:



■

Rubric: 2½ points

- No credit if the reported graph is undirected or contains a directed cycle.
- −1 for each misplaced, omitted, or extra vertex or edge.
- No penalty for omitting the vertex labels.

CS/ECE 374 A ✧ Fall 2019
Midterm 2$^{\otimes}$ Problem 2 Solution

Rachel has a map of her home town in the form of an undirected graph G, whose vertices represent intersections and whose edges represent roads between them. A subset of the vertices are marked as bakeries; another disjoint subset of vertices are marked as coffee shops. The graph has two special nodes s and t, which represent Rachel's home and work, respectively.

Describe an algorithm that computes the shortest path in G from s to t that visits both a bakery and a coffee shop, or correctly reports that no such path exists.

Solution: We construct a new *directed* graph $G' = (V', E')$ as follows:

- $V' = V \times \{\text{TRUE}, \text{FALSE}\} \times \{\text{TRUE}, \text{FALSE}\}$ — Each vertex (v, b, c) represents Rachel reaching vertex v, holding a croissant iff $b = \text{TRUE}$, and holding a cappuccino iff $c = \text{TRUE}$.
- E' contains three types of directed edges:
  - Edges into bakeries: $\{(u, b, c) \to (v, \text{TRUE}, c) \mid uv \in E \text{ and } v \text{ is a bakery}\}$
  - Edges into coffee shops: $\{(u, b, c) \to (v, b, \text{TRUE}) \mid uv \in E \text{ and } v \text{ is a coffee shop}\}$
  - Edges into neither: $\{(u, b, c) \to (v, b, c) \mid uv \in E \text{ and } v \text{ is unmarked}\}$

We need to compute the shortest path from $(s, \text{FALSE}, \text{FALSE})$ to $(t, \text{TRUE}, \text{TRUE})$. 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. No penalty for $O(E \log V)$-time algorithm using Dijkstra's algorithm instead of breadth-first search. (The problem statement didn't actually *say* the graph was weighted, but it's a reasonable assumption in context.)

CS/ECE 374 A ✧ Fall 2019
Midterm 2$^{\otimes}$ Problem 3 Solution

An undirected graph $G = (V, E)$ is ***bipartite*** if its vertices can be partitioned into two subsets L and R, such that every edge in E has one endpoint in L and one endpoint in R. Describe and analyze an algorithm to determine, given an undirected graph G as input, whether G is bipartite. *[Hint: Every tree is bipartite.]*

Solution: Let G be the input graph. The algorithm runs in three phases:

- First, compute a spanning tree T of G using whatever-first search, starting at an arbitrary vertex s. (If G is disconnected, consider each component of G separately, and return TRUE if and only if every component is bipartite.)
- Color s black. Using a preorder traversal of T, color every vertex except s the opposite color of its parent.
- Finally, scan through all edges in G by brute force; if any edge joins two vertices with the same color, return FALSE, and if all edges join vertices of both colors, return TRUE.

This algorithm runs in $O(V + E)$ time. ■

Solution: Let G be the input graph. The algorithm runs in two phases:

- Perform a breadth-first search of the graph, starting at an arbitrary source vertex s, labeling every vertex v with its shortest-path distance $dist(v)$ from s. (If G is not connected, consider each component of G separately, and return TRUE if and only if every component is bipartite.)
- Then for every edge uv in the graph, if $dist(u) = dist(v)$, return FALSE; if every edge joins vertices with two different distances, return TRUE.

This algorithm runs in $O(V + E)$ time.

This algorithm relies on a specific property of shortest-path distances in *unweighted, undirected* graphs: For every source vertex s and every edge uv, the distance to from s to u and the distance from s to v differ by at most 1. ■

Solution: The following variant of depth-first search either colors every vertex black or white, so that every edge has one endpoint of each color, or fails because the input graph is not bipartite.

```
2COLOR(G):
  for all vertices v
      v.color ← NONE
  for all vertices v
      if v.color = NONE
          2COLORDFS(v, WHITE)
```

```
2COLORDFS(v, mycolor):
  if mycolor = WHITE
      yourcolor ← BLACK
  else
      yourcolor ← WHITE
  v.color ← mycolor
  for each edge vw
      if w.color = mycolor
          fail gracefully
      else if w.color = NONE
          2COLORDFS(w, yourcolor)
```

The algorithm runs in $O(V + E)$ time. ■

Solution: Assume without loss of generality that the input graph $G = (V, E)$ is connected. (Otherwise, consider each component of G separately, and return TRUE if and only if every component of G is bipartite.) Construct a new undirected graph $G' = (V', E')$ as follows:

- $V' = V \times 0, 1$
- $E' = \{(u, 0)(v, 1) \mid uv \in E\} \cup \{(u, 1)(v, 0) \mid uv \in E\}$

G' has exactly twice as many vertices and twice as many edges as G. Now perform a whatever-first search in G' starting at an arbitrary vertex. If this search marks every vertex of G', then report that G *is not* bipartite; if at least one vertex remains unmarked, report that G *is* bipartite. This algorithm runs in ***$O(V + E)$ time***.

Suppose G is not bipartite. Then G contains an odd cycle $v_1 \to v_2 \to \cdots \to v_{2k+1} \to v_1$, and therefore H contains the cycle

$$(v_1, 0) \to (v_2, 1) \to \cdots \to (v_{2k+1}, 0) \to (v_1, 1) \to (v_2, 0) \to \cdots \to (v_{2k+1}, 1) \to (v_1, 0).$$

Because G is connected, G contains a path from v_1 to every other vertex u of G. It follows that H contains a path from $(v_1, 0)$ to every vertex (u, i) of H, obtained either by following some $v_1 \rightsquigarrow u$ path in G directly, or by going around the cycle to $(v_1, 1)$ and then following some $v_1 \rightsquigarrow u$ path in G. We conclude that H is connected.

On the other hand, suppose G is bipartite. Color the vertices of G alternately black and white. Consider any two vertices u and v of G. Any path between $(u, 0)$ and $(v, 0)$ must have even length; but any path between a black vertex and a white vertex must have odd length. Thus, $(u, 0)$ and $(v, 0)$ lie in the same component of H if and only if u and v have the same color. Similarly, $(u, 0)$ and $(v, 1)$ lie in the same component of H if and only if u and v have different colors. We conclude that H consists of two disjoint copies of G, one with 0 = white and 1 = black, the other with 0 = black and 1 = white. ■

Rubric: 10 points = 8 points for the algorithm (if recursive: 2 point for base case + 6 points for recursive case) + 2 point for time analysis. These are not the only correct solutions. No penalty if the algorithm works only for connected graphs. A proof of correctness is not required. Max 7 points for an algorithm that runs in $O(E \log V)$ time; max 4 points for any slower correct algorithm.

CS/ECE 374 A ✧ Fall 2019
Midterm 2$^{\otimes}$ Problem 4 Solution

Describe an efficient algorithm that computes the largest number of students that Satya can host for testing in a sequence of n rooms without demolishing more than k walls. To prevent cheating, Satya can host students in two adjacent rooms i and $i+1$ only if he demolishes the wall between them. The input to your algorithm is the integer k and an array $S[1..n]$, where each $S[i]$ is the number of students that can fit in room i. (See the question handout for more details.)

Solution: For any index i, any integer ℓ, and any boolean *prev*, let $MaxHost(i, \ell, prev)$ denote the maximum number of students that can be hosted for testing in rooms i through n, assuming we can demolish at most ℓ walls and that room $i-1$ is occupied iff $prev = \textsc{True}$. We need to compute $MaxHost(1, k, \textsc{False})$.

This function can be described by the following recurrence:

$$MaxHost(i,\ell,prev) = \begin{cases} 0 & \text{if } i > n \\ MaxHost(i+1, 0, \textsc{False}) & \text{if } prev \text{ and } \ell = 0 \\ \max\left\{\begin{matrix} S[i] + MaxHost(i+1, \ell-1, \textsc{True}) \\ MaxHost(i+1, \ell, \textsc{False}) \end{matrix}\right\} & \text{if } prev \text{ and } \ell > 0 \\ \max\left\{\begin{matrix} S[i] + MaxHost(i+1, \ell, \textsc{True}) \\ MaxHost(i+1, \ell, \textsc{False}) \end{matrix}\right\} & \text{if } \neg prev \end{cases}$$

We can memoize this function into an array $MaxHost[1..n, 0..k, \textsc{False}..\textsc{True}]$, which we can fill by decreasing i in the outer loop and considering the other two coordinates in any order in the inner loops. The resulting algorithm runs in ***O(kn) time***. ■

Solution: For any index i, any integer ℓ, and any boolean *prev*, let $MaxHost(i, \ell, prev)$ denote the maximum number of students that can be hosted for testing in rooms i through n, assuming we can demolish at most ℓ walls and that room $i-1$ is occupied iff $prev = \textsc{True}$. We need to compute $MaxHost(1, k, \textsc{False})$.

This function can be described by the following recurrence:

$$MaxHost(i,\ell,prev) = \begin{cases} -\infty & \text{if } \ell < 0 \\ 0 & \text{if } i > n \\ \max\left\{\begin{matrix} S[i] + MaxHost(i+1, \ell-1, \textsc{True}) \\ MaxHost(i+1, \ell, \textsc{False}) \end{matrix}\right\} & \text{if } prev \\ \max\left\{\begin{matrix} S[i] + MaxHost(i+1, \ell, \textsc{True}) \\ MaxHost(i+1, \ell, \textsc{False}) \end{matrix}\right\} & \text{otherwise} \end{cases}$$

We can memoize this function into an array $MaxHost[1..n, 0..k, \textsc{False}..\textsc{True}]$, which we

can fill by decreasing i in the outer loop and considering the other two coordinates in any order in the inner loops. The resulting algorithm runs in ***O(kn) time***. ■

Solution: For any index i and any integer ℓ, let $MaxSt(i,\ell)$ denote the maximum number of students that can be hosted for testing in rooms i through n, assuming we can demolish at most ℓ walls and that room $i-1$ is definitely unoccupied.

As a helper function, let $Total(i,c)$ denote the total number of students that can be placed in the c consecutive rooms starting at room i. This function obeys the following recurrence:

$$Total(i,c) = \begin{cases} 0 & \text{if } c = 0 \\ -\infty & \text{if } c > 0 \text{ and } i > n \\ S[i] + Total(i+1, c-1) & \text{otherwise} \end{cases}$$

Our recurrence for $MaxSt$ will need $Total(i,c)$ for all $1 \le i \le n$ and $0 \le c \le k$. We can memoize these values into a 2d array $Total[1..n, 0..k]$, which we can fill by decreasing i in the outer loop and increasing c in the inner loop in $O(nk)$ time.

This this helper function in hand, the $MaxSt$ function can be described by the following recurrence:

$$MaxSt(i,\ell) = \begin{cases} 0 & \text{if } i > n \\ \max\left\{ \begin{array}{c} Total(i,c) + \\ MaxSt(i+c+1, \ell - \max\{0, c-1\}) \end{array} \;\middle|\; 0 \le c \le \min\left\{ \begin{array}{c} \ell+1 \\ n-i+1 \end{array} \right\} \right\} & \text{otherwise} \end{cases}$$

For each valid value of c, the recurrence tries occupying c consecutive rooms i through $i+c-1$, and then recursively considers rooms $i+c+1$ through n. Occupying c consecutive rooms requires smashing $\max\{0, c-1\}$ walls. The upper bounds on c ensure that Satya does not smash too many walls, and that he does not try to occupy rooms numbered higher than n.

We can memoize this function into an array $MaxSt[1..n, 0..k]$, which we can fill by decreasing i in the outer loop and considering ℓ in any order in the inner loop. The resulting algorithm runs in ***O(kn) time***. ■

Rubric: 10 points: standard dynamic programming rubric. These are not the only correct solutions. In particular, these are not the only correct recurrences for these functions, or the only correct evaluation orders for these recurrences. Watch carefully for off-by-one errors and incorrect boundary cases.

- −1 for reporting running time as $O(n)$, treating k as a constant.
- −½ for reporting running time as $O(n^2)$.

CS/ECE 374 A ✧ Fall 2019
Midterm 2$^{\otimes}$ Problem 5 Solution

Suppose you are given an array $A[1..n]$ of numbers.

(a) Describe and analyze an algorithm that either returns two indices i and j such that $A[i]+A[j]=374$, or correctly reports that no such indices exist. Do ***not*** use hashing.

Solution: First sort the array A; then for each index i, binary search for an index j such that $A[j]=374-A[i]$.

```
2SUM374(A[1..n]):
  sort A
  for i ← 1 to n
      j ← FINDINDEX(A, 374 − A[i])    ⟨⟨binary search⟩⟩
      if j ≠ NONE
          return (i, j)
  return NONE
```

(The subroutine FINDINDEX(A,x) finds an index i such that $A[i]=x$, or returns NONE if there is no such index, using binary search.) The algorithm runs in ***$O(n \log n)$ time***.

If we're really being pedantic, we should return indices into the *original* input array, rather than indices into the *sorted* input array. We can do that by creating an index array $I[1..n]$ where $I[i]=i$ for all i, permuting I as we sort A, and eventually returning $(I[i], I[j])$ instead of (i,j). ■

Solution: After sorting the array A, we essentially merge A with a second array B, where $B[i]=374-A[i]$; if we ever discover a duplicate value $A[i]=B[j]$, we return the pair (i,j). But in fact, we don't need to build the new array B explicitly, and we don't need to record the merged array.

```
2SUM374(A[1..n]):
  sort A
  i ← 1;  j ← n
  while i < j
      if A[i] + A[j] < 374
          i ← i + 1
      else if A[i] + A[j] > 374
          j ← j − 1
      else ⟨⟨A[i] + A[j] = 374⟩⟩
          return (i, j)
  return NONE
```

The algorithm runs in ***$O(n \log n)$ time***; the running time is dominated by the initial sort. ■

Rubric: 5 points = 4 for algorithm + 1 for time analysis. Max 2 points for brute-force $O(n^2)$-time algorithm. No penalty for returning indices into the sorted input array. No credit for an $O(n)$-time algorithm that uses hashing; the instructions ruled that out. No credit for algorithms that only work when the input numbers are integers, or even worse, positive integers.

(b) Describe and analyze an algorithm that either returns three indices i, j, and k such that $A[i]+A[j]+A[k]=374$, or correctly reports that no such indices exist. Again, do ***not*** use hashing.

Solution (4/5): Intuitively, for each index k, we look for indices i and j such that $A[i]+A[j]=374-A[k]$, using a variant of the algorithm from part (a).

```
3SUM374(A[1..n]):
  sort A
  for i ← 1 to n
      for j ← 1 to n
          k ← FINDINDEX(A, 374 − A[i] − A[j])    ⟨⟨binary search⟩⟩
          if k ≠ NONE
              return (i, j, k)
  return NONE
```

(Again, FINDINDEX(A,x) finds an index i such that $A[i]=x$, or returns NONE if there is no such index, using binary search.) The algorithm performs $O(n^2)$ binary searches and therefore runs in $\boldsymbol{O(n^2 \log n)}$ ***time***. ■

Solution: First sort the array. Then for each index k, we look for indices i and j such that $A[i]+A[j]=374-A[k]$, using the same "merge" strategy from part (a).

```
2SUM374(A[1..n]):
  sort A
  for j ← 1 to n
      i ← 1;  k ← n
      while i < k
          if A[i] + A[j] + A[k] < 374
              i ← i + 1
          else if A[i] + A[j] + A[k] > 374
              k ← k − 1
          else ⟨⟨A[i] + A[j] + A[k] = 374⟩⟩
              return (i, j, k)
  return NONE
```

The algorithm runs in $\boldsymbol{O(n^2)}$ ***time***. ■

Rubric: 5 points = 4 for algorithm + 1 for time analysis. Max 4 points for $O(n^2 \log n)$-time algorithm. Max 2 points for brute-force $O(n^3)$-time algorithm. No credit for an $O(n^2)$-time algorithm that uses hashing; the instructions ruled that out. No credit for algorithms that only work when the input numbers are integers, or even worse, positive integers.