CS161 Final Exam

Do not turn this page until you are instructed to do so!

Instructions: Solve all questions to the best of your abilities. You may cite any result we have seen in class or CLRS (or in any resource linked from the course website, except Piazza) without proof. You have **180 minutes** to complete this exam. You may use one two-sided sheet of notes that you have prepared yourself. You may not use any other notes, books, or online resources. There is one blank page at the end that you may tear off as scratch paper, and one blank page for extra work. Please write your name at the top of all pages.

Advice: If you get stuck on a problem, move on to the next one. Pay attention to how many points each problem is worth. Read the problems carefully.

The following is a statement of the Stanford University Honor Code:

1. *The Honor Code is an undertaking of the students, individually and collectively:*
   (1) *that they will not give or receive aid in examinations; that they will not give or receive unpermitted aid in class work, in the preparation of reports, or in any other work that is to be used by the instructor as the basis of grading;*
   (2) *that they will do their share and take an active part in seeing to it that others as well as themselves uphold the spirit and letter of the Honor Code.*
2. *The faculty on its part manifests its confidence in the honor of its students by refraining from proctoring examinations and from taking unusual and unreasonable precautions to prevent the forms of dishonesty mentioned above. The faculty will also avoid, as far as practicable, academic procedures that create temptations to violate the Honor Code.*
3. *While the faculty alone has the right and obligation to set academic requirements, the students and faculty will work together to establish optimal conditions for honorable academic work.*

By signing your name below, you acknowledge that you have abided by the Stanford Honor Code while taking this exam.

Signature: _____

Name: SOLUTIONS

SUNetID: _____

Section	1 (multiple choice)	2 (short answer)	3 (alg. design)	4 (alg. analysis)	Total
Score					
Maximum	20	32	33	15	100

1 Multiple Choice (20 pts)

No explanation is required for the questions in Section 1. Please clearly mark your answers; if you must change an answer, either erase thoroughly or else make it **very** clear which answer you intend. **Ambiguous answers will be marked incorrect.**

1.1. **(2 pt.)** Below, assume that all functions $f(n)$ map positive integers to positive integers. Which of the following functions $f(n)$ are $O(n)$? Circle all that apply.

(A) $f(n) = 2n$
(B) $f(n) = \sqrt{n}$
(C) $f(n) = n\log(n) + 10$
(D) Any $f(n)$ that satisfies $f(n) = \Omega(n)$ ← for example, $n^2 = \Omega(n)$ but is not $O(n)$
(E) Any $f(n)$ that satisfies the recurrence $f(n) \leq 2 \cdot f(\lceil n/2 \rceil) + n$ for $n \geq 2$.
(F) Any $f(n)$ that satisfies the recurrence $f(n) \leq 3 \cdot f(\lceil n/2 \rceil) + n$ for $n \geq 2$.
(G) Any $f(n)$ that satisfies the recurrence $f(n) \leq f(\lceil n/2 \rceil) + f(\lceil n/20 \rceil) + n$ for $n \geq 20$.

1.2. **(2 pt.)** Suppose we run DFS on a graph G. Suppose that v and w are vertices in G, and in our run of DFS we assign start and finish times to these vertices. Which of the following are possible? Circle all that apply.

(A) v.start ≤ w.start ≤ w.finish ≤ v.finish
(B) w.start ≤ w.finish ≤ v.start ≤ v.finish
(C) v.start ≤ w.start ≤ v.finish ≤ w.finish
(D) w.start ≤ v.start ≤ w.finish ≤ v.finish
(E) w.start ≤ v.finish ≤ v.start ≤ w.finish

1.3. **(2 pt.)** Which of the following can RadixSort sort in time $O(n)$? Circle all that apply. (Assume that you can represent any integer in any basis in time $O(1)$).

(A) n positive integers of value at most 10 — $\lceil \log_n(10) \rceil \cdot \Theta(n) = \Theta(n)$
(B) n positive integers of value at most n — $\lceil \log_n(n) \rceil \cdot \Theta(n) = \Theta(n)$
(C) n positive integers of value at most n^{10} — $\lceil \log_n(n^{10}) \rceil \cdot \Theta(n) = \Theta(n)$
(D) n positive integers of value at most 10^n — $\lceil \log_n(10^n) \rceil \cdot \Theta(n) = \Theta(n^2/\log(n))$
(E) n positive integers of value at most n^n — $\lceil \log_n(n^n) \rceil \cdot \Theta(n) = \Theta(n^2)$

Running time is $O(\lceil \log_r(M) \rceil (n + r))$. (#rounds; passes items; initialize buckets.) Choose $r = n$ to balance the $n + r$ term.

1.4. **(2 pt.)** Let $G = (V, E)$ be a unweighted directed **acyclic** graph, and let $s \in V$. You want to design a dynamic programming algorithm to find the *longest path* in G that starts at s. You decide to fill in a table L, where $L[x]$ is the cost of the longest path from s to x. Which of the following is the correct recursive structure? Circle exactly one. **(Note: don't worry about how you would actually implement the DP algorithm, that's not part of this problem.)**

(A) $L[x] = \max\{L[u] + 1 : (u, x) \in E\}$
(B) $L[x] = \min\{L[u] - 1 : (u, x) \in E\}$
(C) $L[x] = L[u] + 1$, where u is the vertex that maximizes $L[u]$

1.5. **(8 pt.)** For each of the following quantities, fill in a single expression from the choices below that describes it. **Below, all graphs G have n vertices and m edges.**

1.5.1. The time it takes to search for an element in a red-black tree which is storing n items:

$\Theta(\log(n))$

1.5.2. The time it takes to deterministically sort n arbitrary comparable objects: $\Theta(n \log(n))$

1.5.3. The expected number of items hashed into any given bucket, when n items are hashed into n buckets using a hash function h that is chosen uniformly at random from a universal hash family:

$O(1)$

1.5.4. The expected number of times you need to look at a random element from an array A of length n containing distinct integers until you see the maximum element of A:

$\Theta(n)$

1.5.5. The time it takes to find an ordering $v_1, \ldots, v_n$ of the vertices in a directed acyclic graph $G = (V, E)$, so that for every directed edge $(v_i, v_j) \in E$, $i < j$:

$\Theta(n+m)$

1.5.6. The number of edges in a minimum spanning tree in a connected undirected graph:

$\Theta(n)$ (actually exactly $n-1$)

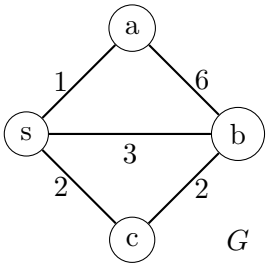
1.5.7. The worst-case running time of the Bellman-Ford algorithm: $\Theta(n \cdot m)$

1.5.8. The time it takes to determine if an unweighted undirected graph is bipartite: $\Theta(n+m)$

Choices (which may be used more than once or not at all):

- $\Theta(n)$
- $\Theta(n \log(n))$
- $\Theta(n^2)$
- $\Theta(nm)$
- $\Theta(\log(n))$
- $O(1)$
- $\Theta(n + m)$

1.6. **(4 pt.)** Consider the undirected weighted graph G shown below. For each algorithm on the right, draw a single line connecting it to the sub-tree on the left naturally associated with the algorithm. A tree may be used more than once or not at all.

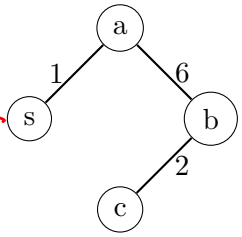


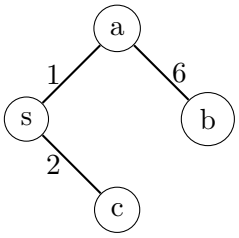
Dijkstra's algorithm, starting from s

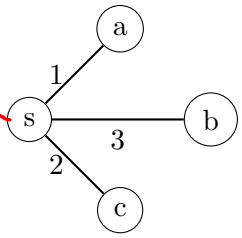
BFS, starting from s, breaking ties in alphabetical order and ignoring edge weights

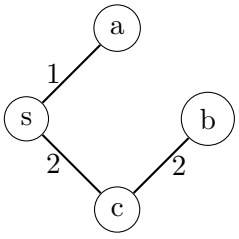
DFS, starting from s, breaking ties in alphabetical order and ignoring edge weights

Prim's algorithm, starting from s









2 Can it be done? (Short answers) (32 pts)

For each of the following tasks, either **explain briefly how you would accomplish it**, or else **explain why it cannot be done**. If you explain how to do it you do not need to justify why your answer is correct. You may use any algorithm or result we have seen in class as a black box.

Below, all graphs have n vertices and m edges.

The first two have been done for you to give an idea of the level of detail we are expecting. Note that it is possible to get full credit on this section without writing any pseudocode, although you may write pseudocode if you like.

2.1. **(0 pt.)** Find the maximum of an unsorted array of length n in time $O(n \log(n))$.

I would use MergeSort to sort the array, and then return the last element of the sorted array.

2.2. **(0 pt.)** Find the maximum of an unsorted array of length n in time $O(1)$.

This cannot be done, because since the maximum could be anywhere, we need to at least look at every element in the array, which takes time $\Omega(n)$.

2.3. **(4 pt.)** Given a weighted directed graph G with non-negative edge weights, and given vertices s and t, deterministically find a shortest path from s to t in time $O((m+n)\log(n))$.

Use Dijkstra's Algorithm

2.4. **(4 pt.)** Given an undirected unweighted graph G with maximum degree[1] d and a vertex s, find all of the vertices v with distance at most 6 from s, in time $O(d^6)$.

Run BFS to depth 6.

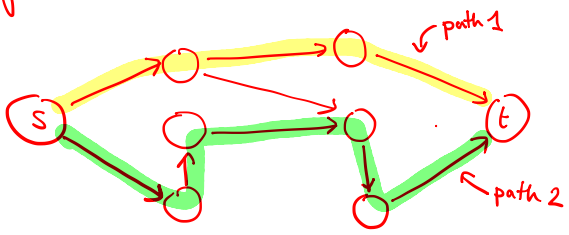
2.5. **(4 pt.)** Search for an element in a sorted array, which contains n distinct arbitrary comparable items, in time $O(1)$.

This cannot be done. The element could be in any of n locations, so this requires a decision tree of depth $\log(n)$. Hence, any algorithm must use $\Omega(\log(n))$ comparisons (hence time) in the worst case.

2.6. **(4 pt.)** Given a directed unweighted graph G and vertices s, t, find the largest number k so that there are k disjoint paths from s to t, in time $O(nm^2)$. (We say two paths are *disjoint* if they don't share any edges. It's okay if they share vertices). Assume that s only has outgoing edges and t only has ingoing edges.

Use Ford-Fulkerson to find the max flow in the graph where all the weights are 1.

eg,



, and the max flow is 2. (sending 1 unit of stuff on each path).

[1]Recall that the *degree* of a vertex is the number of edges leaving that vertex. The maximum degree of a graph is the maximum degree of any vertex in the graph.

2.7. **(4 pt.)** On a hypothetical final exam, there are n problems. Problem i is worth p_i points, and it takes t_i minutes to complete, where t_i is a positive integer. You have T minutes total to take the exam. The numbers (p_i, t_i) are released well before the exam starts. Given these numbers, in time $O(nT)$, find a set of problems that you can solve in T minutes in order to maximize the points that you receive during the exam. (Assume that there is no partial credit on this hypothetical exam, although there may be partial credit given on the real-life exam).

This is exactly the 0/1 knapsack problem.
Use the DP solution we saw in class.

2.8. **(4 pt.)** Suppose that you have access to a genie which does the following in time $O(n)$:

- Input: an array A of length n, containing arbitrary comparable elements.
- Output: with probability 1/2, a sorted version of A. With probability 1/2, it outputs an array which may not be sorted.

Design a randomized algorithm which uses the genie and sorts an array of arbitrary comparable elements in time $O(n)$, which succeeds with probability at least $1 - 2^{-10}$.

Run the genie 10 times, and check each time if the list is sorted.
If the list is sorted, return it.

$\mathbb{P}\{\text{this algorithm fails}\} = \left(\mathbb{P}\{\text{genie fails}\}\right)^{10} = \left(\frac{1}{2}\right)^{10}$

Note: Problems 2.9 and 2.10 may be more difficult. You may wish to skip them and come back to them later.

2.9. **(4 pt.)** *(May be more difficult)* Say that an array A of odd length $n = 2r + 1$ is *oscillating* if $A[0] \leq A[1] \geq A[2] \leq A[3] \geq \cdots \leq A[2r-1] \geq A[2r]$. For example, the array $[5, 7, 1, 6, 2, 8, 4, 9, 3]$ is oscillating. Given an unsorted array B containing $n = 2r + 1$ distinct comparable items, output an oscillating array A that has all the same elements as B, in time $O(n)$.

Use MEDIAN to find the median m of the array.

Use PARTITION to split the array into $L \cup \{m\} \cup R$ s.t. $x < m \; \forall \; x \in L$, $y > m \; \forall \; y \in R$

Return

$[m, R[0], L[0], R[1], L[1], \ldots, R[r-1], L[r-1]]$

2.10. **(4 pt.)** *(May be more difficult)* Let $G = (V, E)$ be a directed unweighted graph. Say that G is "kind-of-connected" if for every $u, v \in V$, either there is a path from u to v in G, or there is a path from v to u in G, or both. Decide if G is kind-of-connected in time $O(n + m)$.

Use the SCC algorithm to find the SCC DAG G' of G.

Topologically sort the SCC DAG; say the SCC's are $A_1, A_2, \ldots, A_r$

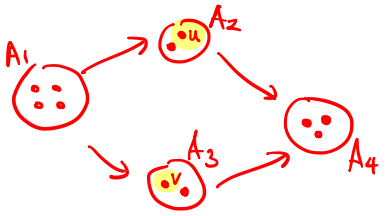
For $i = 1, \_, r-1$:

  if there is no edge from A_i to A_{i+1} in the SCC DAG:

    return FALSE

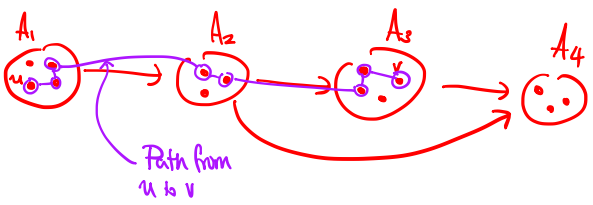
return TRUE

That is, if the SCC DAG looks like



then there's no path from u to v, so we should return FALSE

But if it looks like

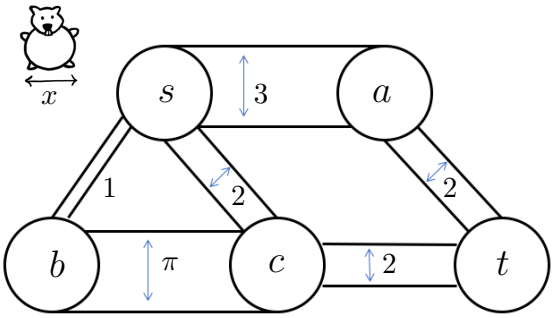


then there's a path from any u to any v.

3 Algorithm Design (33 pts)

3.1. **(11 pt.)** Let $G = (V, E)$ be an undirected, weighted graph, with n vertices and m edges, which represents a network of tunnels belonging to a family of groundhogs.[2] The positive weight $w(u, v)$ on an edge $\{u, v\}$ represents the *width* of the tunnel between u and v. A groundhog of width x can only pass through tunnels of width $\geq x$.

For example, in the graph below, a groundhog with width 2 could get from s to t (either by $s \to a \to t$ or by $s \to c \to t$). However, a groundhog of width 3 could not get from s to t.



3.1.1. **(5 pt.)** Given G, vertices $s, t \in V$, and a desired groundhog width $x > 0$, design an algorithm which finds a path from s to t that a groundhog of width x could fit through, or else reports that no such path exists. Your algorithm should run in time $O(n + m)$. You may use any algorithm we have seen in class as a subroutine.

[We are expecting: Pseudocode, a high-level description of your algorithm, and a short justification of the running time.]

```
def findFatPath(G, s, t, x):
    Let G' be a copy of G w/ all the tunnels of width < x removed.
        \\This takes time O(n+m) if the graph is represented with adjacency lists.
    Run BFS on G'   \\Time O(n+m)
    If BFS finds an s-t path:
        return that path
    Else:
        return "No such path"
```

Description: Remove all the thin edges from G, then run BFS.

Running Time: Copying the graph and running BFS both take time O(m+n).

[Continued on next page]

[2] A **groundhog** is a rodent that lives in North America and digs tunnels.

3.1.2. **(6 pt.) Using your answer to 3.1.1. as a subroutine,** design an algorithm which will find the path from s to t which accomodates the largest groundhog possible. Your algorithm should run in time $O((n+m)\log(m))$. You may use any algorithm we have seen in class as a subroutine.

Note: The tunnel widths are arbitrary real numbers: you cannot assume that they are bounded integers.

[We are expecting: Pseudocode, and a high-level description of your algorithm. Don't worry about floors and ceilings, we will not take off points for small errors like that.]

High-level idea: use binary search and 3.1.1. to find the largest x.
Note that the maximum width x must be equal to one of the edge weights.

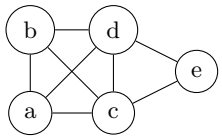
```
def findFattestPath(G, s, t):
    Sort the edges E by increasing width.    // Time O(m log(m))
    l=0, h=n-1      // searching the interval {l, l+1, ..., h}
    while h > l:
        mid = ⌈(h+l)/2⌉
        P = findFatPath(s, t, G, width(E[mid]))
        If P == "no such path":  // then mid was too wide! Search below it.
            h = mid-1
        else:                    // then mid worked, but there might be a wider solution.
            l = mid
    // after the while loop is done, h=l
    Return findFatPath(s, t, G, width(E[l]))
```

The running time is $O(\log(m)(n+m))$ b/c the while loop runs $O(\log(m))$ times, and each time findFatPath takes time $O(n+m)$.

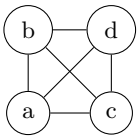
3.2. **(11 pt.)** Let $G = (V, E)$ be an undirected, unweighted graph. For a subset $S \subseteq V$, define the **subgraph induced by** S to be the graph $G' = (S, E')$, where $E' \subseteq E$, and an edge $\{u, v\} \in E$ is included in E' if and only if $u \in S$ and $v \in S$.

For any $k < n$, say that a graph G is k-well-connected if every vertex has degree at least k. (That is, if there are least k edges coming out of each vertex).

For example, in the graph G below, the subgraph G' induced by $S = \{a, b, c, d\}$ is shown on the right. G' is 3-well-connected, since every vertex in G' has degree at least 3. However, G is not 3-well-connected since vertex E has degree 2.



$G = (V, E)$



$G' = (S, E')$, for $S = \{a, b, c, d\}$

Observation: If G' is a k-well-connected subgraph induced by S, and $v \in V$ has degree $< k$, then $v \notin S$. This is because v would have degree $< k$ in the induced subgraph G' as well, and so G' couldn't be k-well-connected if it included v.

3.2.1. **(9 pt.)** Guided by the **observation** above, design a greedy algorithm to find a maximal set $S \subseteq V$ so that the subgraph $G' = (S, E')$ induced by S is k-well-connected.

In the example above, if $k = 3$, your algorithm should return $\{a, b, c, d\}$, and if $k = 4$ your algorithm should return the empty set.

You may assume that your representation of a graph supports the following operations:

- `degree(v)`: return the degree of a vertex in time $O(1)$
- `remove(v)`: remove a vertex and all edges connected to that vertex from the graph, in time $O(\text{degree}(v))$.

Your algorithm should run in time $O(n^2)$.

[We are expecting: Pseudocode and English description of what your algorithm is doing, as well as a justification of the running time. You do not need to prove that your algorithm is correct.]

Idea: greedily remove all the low-degree vertices.

```
def findWellConnectedSubgraph(G, k):
    while(TRUE):
        If there is a vertex with degree(v) < k:   \\ takes time O(n) to run degree(v) for all v.
            remove(v)   \\ O(deg(v)) ≤ O(k)
        Else:
            break
    Return G.
```

[More space on next page]

(Additional space for Problem 3.2.1.)

(continued from previous page)

Running time:

The while loop runs at most n times.

Each iteration uses $O(n + k) = O(n)$ time.

(n: to find a vertex with degree $< k$; k: to delete that vertex)

So the total running time is $O(n^2)$

3.2.2. **(2 pt.)** You do not need to formally prove why your algorithm is correct, but give an informal but convincing justification. (A few sentences should be enough).
[We are expecting: a few sentences about why your algorithm is correct.]

This algorithm is correct because at each step we are removing a vertex that could not possibly be in a set S so that $G|_S$ is k-well-connected. That is, our greedy choices do not rule out success.

3.3. **(11 pt.)** Suppose that A is an $n \times n$ array that contains only zeros and ones. Your goal is to find the largest square in A that is all ones. That is, you want to find i, j and ℓ so that ℓ is as large as possible and so that the sub-array $A[i-\ell+1 : i+1][j-\ell+1 : j+1]$ is entirely ones. (Notice that this requires $\ell \leq i+1$ and $\ell \leq j+1$.

For example, if A were the matrix

	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$
$i=4$	1	0	0	0	0
$i=3$	0	1	1	0	1
$i=2$	0	1	1	1	1
$i=1$	1	1	1	1	1
$i=0$	0	1	1	1	1

then you should return $i = 2, j = 4, \ell = 3$, corresponding the the box drawn above. (Here, the lower-left corner of the matrix is indexed as $(0, 0)$).

In the questions on the next two pages, you will design an algorithm to perform this task.

[Problems on next page]

3.3.1. **(5 pt.)** Define a function $F(x, y)$ to be the side length of the largest all-one square whose upper-right corner is (x, y).

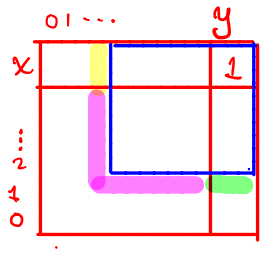
In the example above, $F(2, 4) = 3$, while $F(0, 0) = 0$.

Give an equation[3] that expresses $F(x, y)$ in terms of $F(x-1, y), F(x, y-1), F(x-1, y-1)$ and $A[x, y]$, and explain why your equation is correct.

[We are expecting: An equation and an informal but convincing argument that it is correct.]

First, if $A[x,y] = 0$, then $F(x,y) = 0$.

If $A[x,y] = 1$, then let $\ell = F(x,y)$ →

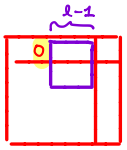


ℓ is the side length of the biggest all-ones square.

There are 3 cases (non-exclusive):

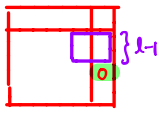
1. There is a zero here.

   In that case, we have $\ell - 1 = F(x, y-1)$



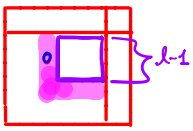
2. There is a zero here

   In that case, we have $\ell - 1 = F(x-1, y)$



3. There is a zero somewhere here

   In that case, we have $\ell - 1 = F(x-1, y-1)$



Notice that we always have

$\ell - 1 \leq F(x-1, y-1)$,
$\ell - 1 \leq F(x, y-1)$,
$\ell - 1 \leq F(x-1, y)$,

and these 3 cases say that at least one of these is an equality.

Between all of these,

$$F(x,y) = A[x,y] \cdot \left(\min\{F(x,y-1), F(x-1,y), F(x-1,y-1)\} + 1 \right)$$

[Another part on next page]

[3] For example, a completely incorrect answer that has the correct type would be

$$F(x, y) = \max\{F(x, y-1), F(x-1, y)\} + F(x-1, y-1)^2.$$

3.3.2. **(6 pt.)** Give an algorithm to return the largest all-one square. Your algorithm should take as input an $n \times n$ array A, and should return i, j, ℓ as described above. Your algorithm should run in time $O(n^2)$.

[We are expecting: Pseudocode along with an English description of the idea of your algorithm, as well as a short justification of the running time.]

Idea: We use the formula from the previous part to fill in an $n \times n$ table.

```
def findBigSquare(A):
    F = n×n array full of 0's
    for i = 0, ..., n-1:
        F[i, 0] = A[i, 0]
        F[0, i] = A[0, i]
    for x = 1, ..., n-1:
        for y = 1, ..., n-1:
            F[x, y] = A[x, y] · (min{F[x-1, y], F[x, y-1], F[x-1, y-1]} + 1)   (*)
    x*, y* = argmax_{x,y} F[x, y]
    Return x*, y*, F[x*, y*]
```

initialize these entries.

Running time: The running time is dominated by the two for loops, and (*) can be done in $O(1)$ time, so the running time is $O(n^2)$.

4 Algorithm Analysis (15 pts)

4.1. **(10 pt.)**

Consider the following algorithm for finding a minimum spanning tree in a connected, weighted, undirected graph $G = (V, E)$.

```
def newMST(G):
     while there is a cycle in G:
          let C be any cycle in G
          remove the largest-weight edge from C
     return G
```

That is, while the algorithm can find a cycle in G, it deletes the edge with the largest weight in that cycle. When it can no longer find a cycle, then it returns whatever is left.

4.1.1. **(5 pt.)** Use the following lemma to write a proof by induction that `newMST` is correct: that is, that it always returns an MST of G. **You do not have to prove the lemma (yet).**

> **Lemma:** Let C be any cycle in G, and let $\{u, v\}$ be the edge in that cycle with the largest weight. Then there exists an MST of G that does *not* include edge $\{u, v\}$.

[We are expecting: Your inductive hypothesis, base case, and conclusion, and a description of how the lemma can establish the inductive step.]

Inductive hypothesis: After removing the t^{th} edge and getting G_t, there is an MST T of G so that $T \subseteq G_t$.

Base case: $G_0 = G$, so the inductive hyp. holds by def. for $t = 0$.

Inductive Step: Suppose the inductive hyp. holds for $t-1$, and let $T \subseteq G_{t-1}$ be an MST of G. Then T is an MST of G_{t-1} as well, since $T \subseteq G_{t-1}$ is a spanning tree of G_{t-1}, and it must be minimal or we'd have a smaller spanning tree for G. By the Lemma, there exists an MST T' of G_{t-1} that does not include the removed edge $\{u,v\}$. Then $\text{cost}(T') = \text{cost}(T)$ (since both are MSTs of G_{t-1}), hence T' is an MST of G as well. Then T' is an MST of G, so that $T' \subseteq G_t$, and this establishes the inductive hypothesis for t.

GRADING NOTE:
You needed to use the inductive assumption in your inductive step to get full credit.
(For example, "by the lemma, there exists an MST $\subseteq G_t$" is not correct, because the lemma is about an MST of G_{t-1}, not of G. The inductive hyp. says that it's also an MST of G.)

Conclusion: When the while loop terminates, G_t is a tree, and the inductive hyp. implies that there is an MST T of G s.t. $T \subseteq G_t$. But since both T, G_t are trees (and in particular have $n-1$ edges) this implies that $T = G_t$.

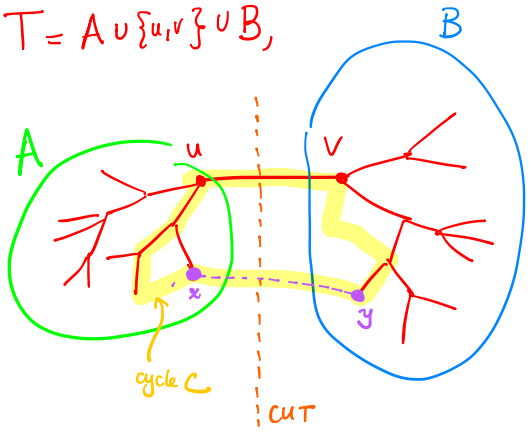
[More space and more problems on next page]

(More space for Problem 4.1.1.).

(This space intentionally blank)

4.1.2. **(5 pt.)** Prove the lemma. (**Hint:** this problem is similar to one from your homework.)

Let C be a cycle in G, and let $\{u,v\}$ be the heaviest edge in C.
Let T be any MST of G.
If T does not contain $\{u,v\}$, we are done.
If T does contain $\{u,v\}$, then T can be written as $T = A \cup \{u,v\} \cup B$, where A, B are disjoint trees.



Consider the cut given by $\{A, B\}$.
Since C is a cycle and $\{u,v\}$ crosses the cut, there must be some other edge $\{x,y\} \in C$ that crosses the cut.
$\{x,y\} \notin T$, since T has no cycles.
Consider T' formed from T by swapping $\{u,v\}$ and $\{x,y\}$.
Then T' is still a tree (we did not create any cycles)
T' still spans (we didn't change the set of vertices we touched)
$\text{cost}(T') \leq \text{cost}(T)$ (since $\text{cost}(x,y) \leq \text{cost}(u,v)$).
So T' is an MST of G, that does not contain $\{u,v\}$.

4.2. **(5 pt.)** Let A be an array of n items from a universe $\mathcal{U}$, and suppose that $\mathcal{H}$ is a universal hash family so that the elements of $\mathcal{H}$ are functions $h : \mathcal{U} \to \{0, \ldots, M-1\}$, where M will be specified below.

Your goal is to decide if there are any repeated elements in A, and your friend comes up with the following randomized algorithm.

Algorithm 1: `isThereARepeat`(A)

Choose $h \in \mathcal{H}$ uniformly at random.
Initialize an array B of length M to all zeros.
for $i \in \{0, \ldots, n-1\}$ **do**
  if $B[h(A[i])] == 1$ **then**
    return `True`
  $B[h(A[i])] \leftarrow 1$
return `False`

The algorithm chooses a random hash function $h \in \mathcal{H}$, and hashes all of the elements of A. If there is ever a collision, the algorithm returns `True`, meaning that it guesses that there was a repeated element. Otherwise, it returns `False`, meaning that there was no repeated element.

Suppose that $M = 10n^2$. Prove that the algorithm is correct with probability at least 9/10. More precisely, prove that (a) if A has a repeated element, then `isThereARepeat`(A) always returns `True`; and (b) if A does not have a repeated element, then `isThereARepeat`(A) returns `False` with probability at least 9/10.

(a) If there is a repeated element, $A[i] = A[j]$, (say $i<j$) then $h(A[i]) = h(A[j])$, and so $B[h(A[i])]$ will be set to 1, and then $B[h[A[j]])] = 1$, so the alg will return TRUE.

(b) If there is no repeated element, then for each $i \neq j$,

$$\mathbb{P}_{h \in \mathcal{H}} \{ h(A[i]) = h(A[j]) \} \leq 1/M$$

So $\mathbb{P}_{h \in \mathcal{H}} \{ \exists (i,j) \in \{0, \ldots, n-1\}^2, i \neq j \text{ so that } h(A[i]) = h(A[j]) \} \leq \binom{n}{2} \cdot \frac{1}{M}$

$\leq n^2/M$

$\leq 1/10$ using the def. of M.

ASIDE

- If $\mathcal{U}$ is REALLY big (like 2^n) then this alg is not so bad: the input has size about n^2 (n items that need n bits each) and the running time is $O(n \cdot (\text{time to evaluate } h) + n^2)$ (initialize B) which is plausibly also $O(n^2)$.
- But if $\mathcal{U}$ is smaller, this seems wasteful. Could we maybe take $M = 10n$ instead of $10 \cdot n^2$?
- As a fun post-exam problem [which was left off of the exam due to length concerns] show that if you take $M = 10n$ then it WON'T work: that is, with high probability the alg. will fail. (for some universal hash family).

This is the end!

This page intentionally blank for extra space for any question.
Please indicate in the relevant problem if you have work here that you want graded, and label your work clearly.

This page intentionally blank for extra space for any question.
Please indicate in the relevant problem if you have work here that you want graded, and label your work clearly.

This page is for **scratch space**. You may tear off this page. Nothing on this page will be graded.

This page is for **scratch space**. You may tear off this page. Nothing on this page will be graded.