# CS161 Practice 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 lecture or CLRS without proof. You have **180 minutes** to complete this exam. You may use two two-sided sheets 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: | SOLUTIONS |
| Name: | SOLUTIONS |
| SUNetID: | SOLUTIONS |

---

<table>
<thead>
<tr>
<th>Section</th>
<th>1 (multiple choice)</th>
<th>2 (short answer)</th>
<th>3 (alg. design)</th>
<th>4 (alg. analysis)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum</td>
<td>20</td>
<td>32</td>
<td>33</td>
<td>15</td>
<td>100</td>
</tr>
</tbody>
</table>
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 \( f(n) = \Omega(n) \)
(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 \( n \geq 2 \).

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) \( v.\text{start} \leq w.\text{start} \leq w.\text{finish} \leq v.\text{finish} \)
(B) \( w.\text{start} \leq w.\text{finish} \leq v.\text{start} \leq v.\text{finish} \)
(C) \( v.\text{start} \leq w.\text{start} \leq v.\text{finish} \leq w.\text{finish} \)
(D) \( w.\text{start} \leq v.\text{start} \leq w.\text{finish} \leq v.\text{finish} \)
(E) \( w.\text{start} \leq v.\text{finish} \leq v.\text{start} \leq w.\text{finish} \)

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
(B) \( n \) positive integers of value at most \( n \)
(C) \( n \) positive integers of value at most \( n^{10} \)
(D) \( n \) positive integers of value at most \( 10^{n} \)
(E) \( n \) positive integers of value at most \( n^{n} \)

1.4. (2 pt.) Let \( G = (V,E) \) be an 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:
\[ \Theta(n) \quad \text{(actually exactly } n-1 \text{)} \]

1.5.7. The worst-case running time of the Bellman-Ford algorithm:
\[ \Theta(nm) \]

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 weighted graph \( G \), possibly with negative edge weights, decide whether or not the graph contains a negative cycle in time \( O(nm) \).

Run the Bellman-Ford algorithm. If the array that Bellman-Ford maintains keeps changing after \( n-1 \) iterations, there is a negative cycle.

---

\(^1\)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.
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 $L [m] \cup R$ s.t. $x < m \forall x \in L$

$y > m \forall y \in R$

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

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 $i = 1, \ldots, r - 1$:

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

return FALSE

That is, if the SCC DAG looks like

Then there is no path from $u$ to $v$, so we should return FALSE

But if it looks like

then there is a path from any $u$ to any $v$.
3 Algorithm Design (33 pts)

3.1. (11 pt.) Suppose that $A$ is an $n \times n$ array containing arbitrary positive integers. Design an algorithm that runs in time $O(n^2 \log(n))$ and finds the largest element that appears at least once in each row of $A$. For example, if

$$A = \begin{bmatrix}
4 & 1 & 2 & 8 \\
2 & 5 & 4 & 1 \\
3 & 4 & 6 & 3 \\
4 & 7 & 4 & 2
\end{bmatrix},$$

then your algorithm should return 4.

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

High-level description: First, sort all the rows. Then for every element in the first row (starting with the largest), use binary search to see if it is in all the other rows.

```
for i = 0, ..., n-1:
    Sort A[i, :] using MERGE.SORT.

for j = 0, ..., n-1:
    p = A[0, n-1-j]
    pGood = TRUE
    for i = 1, ..., n-1:
        Binary search for p in A[i, :]
        if p is not found:
            pGood = FALSE
            break
        if pGood:
            return p
    ```

The running time is $O(n^2 \log(n))$.

- First, we run Mergesort, which is $O(n \log(n))$, $n$ times.
- Then we run binary search, which is $O(\log(n))$, $O(n^2)$ times.
3.2. (11 pt.) There are \( n \) final exams today at Stanford; exam \( i \) is scheduled to begin at time \( a_i \) and end at time \( b_i \). Two exams which overlap cannot be administered in the same classroom; two exams \( i \) and \( j \) are defined to be overlapping if \( [a_i, b_i] \cap [a_j, b_j] \neq \emptyset \) (including if \( b_i = a_j \), so one starts exactly at the time that the other ends). Design a greedy algorithm which solves the following problem.

**Input:** Arrays \( A \) and \( B \) of length \( n \) so that \( A[i] = a_i \) and \( B[i] = b_i \).

**Output:** The smallest number of classrooms necessary to schedule all of the exams, and an optimal assignment of exams to classrooms.

**For example:** Suppose there are three exams, with start and finish times as given below:

\[
\begin{array}{c|c|c|c}
   i & 1 & 2 & 3 \\
   \hline
   a_i & 12pm & 4pm & 2pm \\
   b_i & 3pm & 6pm & 5pm \\
\end{array}
\]

Then the exams can be scheduled in two rooms; Exam 1 and Exam 2 can be scheduled in Room 1 and Exam 3 can be scheduled in Room 2.

Your algorithm should run in time \( O(n \log(n) + nk) \), where \( k \) is the minimum number of classrooms needed.

[We are expecting: Pseudocode AND a short English description. You do not need to prove that your algorithm is correct or justify the running time.]

```python
scheduleRooms(A, B):
    n ← len(A)
    C = [(A[i], i) for i=1,...,n]
    C.sort()  // increasing by start time
    rooms = []
    endTimes = []
    for i=1,...,n:
        for r=1,...,len(rooms):
            if C[i][1] > endTimes[r]:
                rooms[r].append(C[i][2])
                endTimes[r] = B[i]
                break
        else:  // did not break.
            rooms.append([C[i][2]])
            endTimes.append(B[i])
    return rooms
```

**English description:**

We greedily choose the exam with the earliest starting time. If we can fit it in an existing room, we do that. Otherwise we make a new room.
3.3. \textbf{(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 $\ell$ so that $\ell$ is as large as possible and so that the $\ell \times \ell$ subarray of $A$ whose upper-right corner is $(i, j)$ is entirely ones.

For example, if $A$ were the matrix

\begin{align*}
\begin{pmatrix}
0 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 0
\end{pmatrix}
\end{align*}

then you could 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.
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\(^2\) 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.]

\[^2\text{For example, a completely incorrect answer that has the correct type would be } F(x, y) = \max\{F(x, y - 1), F(x - 1, y)\} + (x - 1, y - 1)^2.\]
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 \( i, j, \ell \) 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.

```python
def findBigSquare(A):
    F = n x 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* = \arg\max_{x, y} F[x, y]
    return x*, y*, F[x*, y*]
```

Running time: The running time is dominated by the two for loops, and \((x)\) 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 \( G = (V, E) \).

```python
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 hypothesis holds by def. for \( t = 0 \).

**Inductive step:** Suppose the inductive hypothesis 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 \( G_t \), so that \( T' \subseteq G_t \), and this establishes the inductive hypothesis for \( t \).

**Conclusion:** When the while loop terminates, \( G_t \) is a tree, and the inductive hypothesis 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 for Problem 4.1.1.)

(This space intentionally blank)

4.1.2. *(5 pt.) Prove the lemma.

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:

- This is b/c \( \{x,y\} \) connects the disjoint trees \( A \) and \( B \), so it doesn't create a cycle
  (and it does result in a connected graph).
- \( T' \) still spans (we didn't change the set of vertices we touched)
- \( \text{cost}(T') \leq \text{cost}(T) \) (since \( \text{cost}(xy) < \text{cost}(u,v) \)).

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 $U$, and suppose that $H$ is a universal hash family so that the elements of $H$ are functions $h: U \to \{0, \ldots, b - 1\}$, where $b$ 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 $h \in H$ uniformly at random.
Initialize an array $B$ of length $b$ 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 $h \in H$, 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 $b = 10n^2$. 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, $A[i] = A[j]$, 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 $i \neq j$,

$$\Pr_{h \in H} \left\{ h(A[i]) = h(A[j]) \right\} \leq \frac{1}{M}$$

So

$$\Pr_{h \in H} \left\{ \exists i, j \in \{0, \ldots, n-1\}, i \neq j \text{ so that } h(A[i]) = h(A[j]) \right\} \leq \binom{n}{2} \cdot \frac{1}{M} \leq n^2 \frac{1}{M} \leq \frac{1}{10} \text{ using the def. of } M.$$
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.