**CS141, Fall 2022, Practice Midterm**
**Name:** **Student ID:**

---

This is a practice exam of CS141. The real exam problems will be in similar format, but will be **harder** and contain **more problems**. All the problems in this sample exam are covered in class, quizzes, discussions, the textbook, or in existing homework assignments.

# 1 Multiple Choice

For each problem, there is **only one correct choice**.

(1). If $g(n) = n$, $f(n) = n^2$, then:

A. $f(n) = \Theta(g(n))$
B. $f(n) = o(g(n))$
C. $f(n) = \omega(g(n))$
D. None of the above

(2). $T(n) = 2T(n/2) + n$, $T(n) = \Theta(\_\_\_)$

A. $n^2$
B. $n \log n$
C. $n$

(3). Which of the following algorithm does NOT use divide-and-conquer?

A. Quicksort
B. Mergesort
C. Selection sort
D. Strassen's Algorithm

(4). Which of the following is **wrong**?

A. Huffman coding can be encoded using a greedy algorithm.
B. Huffman coding can be decoded using a greedy algorithm.
C. Huffman code is fixed size

(5). In the kayaking algorithm we discussed in class, we will assign the lightest leftover student with the heaviest possible partner in one kayak. We can justify that putting these two students in a specific kayak will be able to give us an optimal solution, because _____.

A. We can show that this greedy choice is good
B. We can show this problem has optimal substructure

(6). In the kayaking algorithm we discussed in class, we will assign the lightest leftover student with the heaviest possible partner in one kayak. We can justify that repeatedly making such arrangements can give us an optimal solution, because _____.

A. We can show that this greedy choice is good
B. We can show this problem has optimal substructure

## 2 Short Answer

In the unlimited knapsack problem, the state is: _____

The recurrence is: _____

The Final answer is: _____

## 3 Voting for the Programming Contest (22pts)

In UCR, to vote for a proposal, each department will first vote independently, and the majority wins. The $n$ departments will then vote based on the results in the corresponding department, and the majority in this vote wins.

Our goal is to know the fewest number of supporters we need to get the proposal approved. For example, assume there are three departments, with 5, 7, and 3 faculty members, respectively. We only need 3 supporters in the first department and 2 from the third department. In that case, 2 out of 3 departments will vote for yes and win. 5 is the minimum number of supporters we need.

Notes: The number of people in each department and the number of departments in UCR are guaranteed to be odd numbers. For simplicity, we assume the numbers of people in all departments are distinct.

Before implementing the program, let's first design an algorithm and prove its correctness here.

We start with two easy arguments.

1. (1pt) For $n$ departments, we need at least $k =$_____ departments to vote for yes to get a proposal approved.

2. (1pt) To let a department $i$ of $a_i$ people vote for yes, we need $f(a_i) =$_____ supporters.

We then need to find $k$ departments to vote for yes ($k$ is your answer in 3.1). For a selected department with $a_i$ people, we need $f(a_i)$ supporters ($f(a_i)$ is your answer in 3.2). So if we decide to select a set of departments $T$, the total number of supporters we need can be denoted as $answer(T) = \sum_{a_i \in T} f(a_i)$.

A greedy solution is to **repeatedly pick the department with the fewest faculty members for $k$ times**. Assume the array $a_1 < a_2 <, \ldots, < a_n$ has been sorted from the smallest to the largest. In the following, we will prove that the algorithm gives you the optimal solution.

3. (14pts) Let's now prove the part of "greedy choice".

   Your greedy choice is _____(a)_____.

   We will prove this is a good choice, i.e., there exist an optimal solution that chooses this department.

   Fix an optimal solution $S = \{a_{x_1}, a_{x_2}, \ldots, a_{x_k}\}$. If any $x_j = 1$, then $a_1$ has been chosen in $S$, and we already found the solution containing $a_1$.

   Otherwise, this means $x_j \neq 1$ for all $j$, and $a_1$ is not chosen in $S$. We will prove this is impossible because another solution

   $S' =$_____(b)_____

   is better.

   We first prove $S'$ is feasible. This is because _____(c)_____.

   We then prove $S'$ is better than $S$.

   This is because the number of supporters we need for solution $S$ is

   $answer(S) =$_____(d)_____.

   The number of supporters we need for solution $S'$ is

   $answer(S') =$_____(e)_____.

   We can prove

$answer(S') - answer(S) =$ _____(f)_____ $\leq 0$ (please show the simplest form), which is because _____(g)_____.

This proves that $S'$ is a better solution, which is a contradiction. Therefore there must exist an optimal solution containing your greedy choice.

4. (6pts) Let's now prove the part of "optimal substructure".

   What we need to prove is, the best solution containing $a_i$ is $S = \{a_i\} \cup T_1$, where $T_1$ is "the best solution for a subproblem of _____(a)_____".

   Assume to the contrary that the best solution is $S'$, where $S' = \{a_i\} \cup T_2$, and $T_2$ is not the optimal solution of the subproblem you filled in above.

   This will lead to a contradiction since

   $$answer(S) = f(a_i) + answer(T_1)$$

   and

   $$answer(S') = \text{\_\_\_\_\_(b)\_\_\_\_\_}$$

   This indicates $answer(S) \leq answer(S')$ because _____(c)_____. Therefore, $S$ is a better solution than $S'$. This contradicts the assumption that $S'$ is the best solution.

Combining the conclusions of greedy choice and optimal substructure, we proved that our greedy strategy is optimal!

# 4 Longest Increasing Subsequence (LIS) (30 + 5pts)

One important property that we should keep in mind when learning dynamic programming is that many classic problems have lots of variants, and it is crucial that we can identify them and solve them by modifying the solutions to the classic problems. In class, we talked about the knapsack problems, similarity between LCS and edit distance, variants of no-boss party, etc.

In this problem, we consider a variety of different LIS algorithms/problems. We know that LIS can be computed using the following recurrence:

$$L_i = \max \begin{cases} 1 \\ 1 + \max_{1 \leq j \leq i, A_j < A_i} \{L_j\} \end{cases}$$

where $\{A_i\}$ is the input sequence, and $L_i$ denotes the longest LIS that ends at $A_i$. The answer for LIS is $\max_{1 \leq i \leq n} \{L_i\}$.

Now let's consider the variants of LIS. In all cases, your algorithm should spend no more than $O(n^2)$ time.

**Questions:**

1. (10pts) Compute the longest unimodal subsequence. Namely, you want to compute the longest subsequence $B$ from $A$ such that $B_1 < B_2 < \cdots < B_{k-1} < B_k > B_{k-1} > \cdots > B_m$.
2. (10pts) Compute the total number of LIS. Namely, if the length of LIS is $m$, you want to count the number of subsequences with length $m$.
3. (10pts) Compute the lexicographically smallest LIS and prove the correctness of your algorithm.
4. (bonus 5pts) Design $O(n \log n)$ algorithms for all three problems mentioned above.