Problem 4 is worth 40% of this problem set, and problems 1-3 constitute the remaining 60%.

For each problem where you are asked to give an algorithm, more points are given for asymptotically faster algorithms. In judging the number of points to award a correct solution, we only consider the running time in asymptotic notation, i.e. a writeup of an algorithm taking $1000n^2$ steps versus one taking $.01n^2$ steps would receive the same score — both would be simply treated as $\Theta(n^2)$-time solutions. For some problems we may also ask for memory consumption, and using asymptotically less memory similarly is awarded more points.

Each of problems 1 and 2 is a dynamic programming problem and should be viewed as having 3 parts. You should find a function $f$ which can be computed recursively so that evaluation of $f$ on a certain input gives the answer to the stated problem. Part (a) is to define $f$ in words (without mention of how to compute it recursively). You should clearly state how many parameters $f$ has, what those parameters represent, what $f$ evaluated on those parameters represents, and what parameters you should feed into $f$ to get the answer to the stated problem. Part (b) is to give a recurrence relation showing how to compute $f$ recursively. In part (c) you should give the running time and space for solving the original problem using computation of $f$ via memoization or bottom-up dynamic programming. If you need to use certain data structures to make computation of $f$ faster, you should say so. Note: if there are multiple solutions to solve the stated dynamic programming problem, you should describe the most time-efficient one you know. If there are multiple solutions with the same asymptotic time complexity, you should describe the implementation that gives the best asymptotic space complexity.

As an example, suppose the stated problem is that you are given a weighted directed graph with $n$ vertices $\{1, \ldots, n\}$, $m$ edges, and positive weights. The problem then might ask you to print an array of shortest-path distances $d_{u,v}$ for all vertices $u, v$. This can be solved with the Floyd-Warshall algorithm. A solution to part (a) could say that $f(u, v, k)$ is a function that takes as input vertex ID’s $u, v, k$, $f(u, v, k)$ represents the length of the shortest path from $u$ to $v$ in which all intermediate vertices should be from the set $\{1, \ldots, k\}$, and finally the answer $d_{u,v}$ is $f(u, v, n)$. Then part (b) should give a recurrence relation to compute $f$ (see the top of page 3 of “Notes 5” on the lecture notes website). Part (c) in this case should argue why the running time is $\Theta(n^3)$ and how to get space $\Theta(n^2)$ using bottom-up dynamic programming.
Problem 1

You are given an undirected tree (a connected and acyclic graph) in adjacency list form where each vertex \( v \) has a weight \( w(v) \). For a subset \( S \) of vertices, we define its weight \( w(S) \) to be \( \sum_{v \in S} w(v) \). A set \( S \) is called separated if for all \( u, v \in S \), the edge \((u, v)\) does not appear in the graph. Give an algorithm to find the largest \( w(S) \) achievable over all separated sets (you need not find the \( S \) achieving it, just \( w(S) \)).

Problem 2

There are four types of brackets: (, ), <, and >. We define what it means for a string made up of these four characters to be well-nested in the following way:

1. The empty string is well-nested.
2. If \( A \) is well-nested, then so are \(<A>\) and \((A)\).
3. If \( S, T \) are both well-nested, then so is their concatenation \( ST \).

For example, (), <>, (()), ()<>, and ()<() are all well-nested. Meanwhile, (, <, ), (), <>, and (<>) are not well-nested.

Devise an algorithm that takes as input a string \( s \) of length \( n \) made up of these four types of characters. The output should be the length of the shortest well-nested string that contains \( s \) as a subsequence. For example, if \(<()\) is the input, then the answer is 6; a shortest string containing \( s \) as a subsequence is ()<()>

Problem 3

In this problem you will develop an analysis for QuickSelect similar to that for QuickSort covered in lecture. Suppose we call QuickSelect(\(A, k\)), to find the \( k \)th smallest element in an array \( A[1..n] \) of size \( n \). Assume the elements of \( A \) are distinct. Note the running time of QuickSelect is proportional to the number of comparisons. Thus if we let \( X_{i,j} \) be a random variable which is 1 if \( i \)th smallest item and \( j \)th smallest item are ever compared throughout the execution of QuickSelect(\(A, k\)), then the running time is proportional to \( \sum_{i<j} X_{i,j} \).

(a) (6 points) For \( i < j \), give an exact expression for \( \mathbb{E} X_{i,j} \) in terms of \( i, j, k, n \). You may need to employ case analysis.

(b) (4 points) Using (a), show that \( \mathbb{E} \sum_{i<j} X_{i,j} = O(n) \).
Programming Problem


Hint: even if two algorithms have the same asymptotic complexity, one that uses significantly less memory may run faster in practice due to caching effects.

Note: The second batch of test cases does not have the constraint that $N \cdot B \leq 5 \cdot 10^6$ as the final batch does. We thus recommend special-casing test cases in the second batch in your code and doing something slightly different for them.