DUE: Monday, June 24, 6 PM. DO NOT COPY. ACKNOWLEDGE YOUR SOURCES.

Please read http://www.student.cs.uwaterloo.ca/~cs341 for general instructions and policies. Also read Assignment section of course outline for clarification of what “justify” means.

Note: All logarithms are base 2 (i.e., \( \log x \) is defined as \( \log_2 x \)).

Note: For all algorithm design questions, you must give the algorithm, argue the correctness, and analyze time complexity.

1 Warmup... reductions ... and analysis

1.1. [5 marks] **Target3Sum by reduction to 3Sum.** The **Target3Sum** decision problem is: given an array \( A \) of \( n \) distinct integers, \( A = [A[1], ..., A[n]] \) and a target integer \( T \), find out if there exist three (not necessarily distinct) elements in \( A \) whose sum equals \( T \). It is easy to modify any algorithm solving the **3SUM** problem so it solves the **Target3Sum** problem. For example in **Fast3SUM** algorithm from lecture notes we could replace \( x = \text{Sorted2SUM}(A,-A[k]) \) by \( x = \text{Sorted2SUM}(A,T-A[k]) \).

However, you are not allowed to modify existing algorithm **Fast3SUM** and required to use reduction. Design a reduction based algorithm for the **Target3Sum** problem, which calls **Fast3SUM** only one time to produce the desired result. Your algorithm takes an instance of **Target3Sum**, creates a suitable instance of **3SUM** problem, and returns the outcome of the call to **Fast3SUM** as the result.

Justify correctness of you reduction, i.e. show that it always produces correct result, assuming **Fast3SUM** is correct. Analyze the running time of your algorithm.

1.2. [5 marks] **Primality testing analysis.** Consider the primality testing algorithm that tests if a given \( k \)-bit number \( n \) is prime by attempting to divide \( n \) by \( i \) for \( 2 \leq i \leq \sqrt{n} \), \( i \in \mathbb{N} \).

Analyze the run time of this algorithm as a function of input size \( k \). To analyze run-time, do not use the word-RAM model. Instead, assume that you have a division method that runs in time \( \Theta(j^2) \) on \( j \)-bit integers. (This is called the “bit-complexity model”. ) Does the algorithm run in polynomial time? Why or why not?

2 Greedy again ...

[10 marks] In the **JobSelection** problem, the input is a positive integer \( t \) and a sequence of \( n \) pairs of positive integers \((r_1,p_1),(r_2,p_2),\ldots,(r_n,p_n)\) that correspond to the reward \( r_i \) you earn if you complete job \( i \) and the penalty \( p_i \) that you must pay if you do not complete job \( i \). You can only complete \( t \) jobs; a valid solution to the problem is a subset \( S \subseteq \{1,2,\ldots,n\} \) of \( |S| = t \) jobs that maximizes the profit

\[
\text{profit}(S) = \sum_{i \in S} r_i - \sum_{j \notin S} p_j
\]
earned by completing the set $S$ of jobs.

Design a greedy algorithm to solve this optimization problem. Prove that it always returns an optimal solution. Justify correctness and analyze running time.

3 Now we have two knapsacks ...

[10 marks] Consider a variation of the Knapsack problem. There are two knapsacks that have capacity $W_1 > 0$ and $W_2 > 0$, respectively. There are $n$ items $1, 2, \ldots, n$. Item $i$ has weight $w(i) > 0$ and two values $v_1(i) > 0$ and $v_2(i) > 0$. Here $v_k(i)$ is the value one gains by putting item $i$ into knapsack $k$ ($k = 1, 2$). The “Two Knapsacks Problem” is to find two disjoint subsets of items $S_1$ and $S_2$, such that

1. $\sum_{i \in S_1} w(i) \leq W_1$,
2. $\sum_{i \in S_2} w(i) \leq W_2$, and
3. $V = \sum_{i \in S_1} v_1(i) + \sum_{i \in S_2} v_2(i)$ is maximized.

Give a dynamic programming algorithm to find the maximum value $V$. Your algorithm does not need to find the sets $S_1$ and $S_2$. Clearly indicate what your subproblems are, and the order in which you solve them. Justify correctness of your algorithm, and analyze its running time. Is your algorithm a polynomial-time algorithm? Why or why not?

4 “DP” for Longest Oscillating Subsequence.

[10 marks] A sequence $s_1, s_2, \ldots, s_k$ of integers is oscillating sequence if $s_1 < s_2 > s_3 < s_4 > \ldots s_k$ or $s_1 > s_2 < s_3 > s_4 < \ldots s_k$.

Design a DP algorithm that finds a longest oscillating subsequence in given list of $n$ integers $A[1..n]$ in time $O(n^2)$. Clearly indicate what your subproblems are, and the order in which you solve them. Justify correctness of your algorithm, present DP-recurrence, and analyze its running time.