Homework 5: Due February 28 (11:55 a.m.)

Instructions

- **Answer each question on a separate page.**
- **Honors questions are optional. They will not count towards your grade in the course. However you are encouraged to submit your solutions on these problems to receive feedback on your attempts. Our estimation of the difficulty level of these problems is expressed through an indicative number of stars ($'*' =$ easiest) to ($'*****' =$ hardest).**
- **You must enter the names of your collaborators or other sources as a response to Question 0. Do NOT leave this blank; if you worked on the homework entirely on your own, please write "None" here. Even though collaborations in groups of up to 3 people are encouraged, you are required to write your own solution.**

Question 0: List all your collaborators and sources: ($-\infty$ points if left blank)

Question 1: Efficient Fibonacci (3+5+2=10 points)

1. Show that it takes $2^{\Omega(n)}$ time to compute the first n Fibonacci numbers if we use naive recursion. For simplicity, assume that adding any two numbers, regardless of the their size, takes one unit of time. (Hint: write out the recurrence for $T(n)$, the cost of computing the n-th Fibonacci number, and use induction.)
2. Describe an idea of reusing Fibonacci numbers that you have already calculated in order to output the first n Fibonacci numbers much faster (polynomial time).
3. How long does your algorithm take? State a Θ expression for the asymptotic run time of your algorithm. Again assume that adding any two numbers only takes one unit of time.

Question 2: Rod Cutting (4+5+1=10 points)

Suppose we have a rod of length n inches and we also have an array of prices P, where $P[i]$ denotes the selling price (\$) of a piece that is i inches long. (See CLRS Ch15.1 for reference, which gives an algorithm that uses dynamic programming to find a way of cutting the rod into pieces that maximizes the **revenue**). Suppose now we have to pay a cost of \$1 per cut. Define the *profit* we make as the revenue minus the total cost of cutting. We want an algorithm which finds a way to cut the rod that maximizes our **profit**. For your DP algorithm, use the name MaxProfit for the (potentially multi-dimensional) array which stores values of the subproblems.

1. Describe in words what MaxProfit should be and state its dimension. In addition, state the base cases and their values.
2. Give and justify the recurrence MaxProfit should satisfy.
3. Justify the run time of your algorithm to compute MaxProfit as a big-Θ expression.

Question 3: Subset Sum (2+6+2=10 points)

Let $A = \{a_1, \ldots, a_n\}$ be a set of n natural numbers. Given a number $t \in \mathbb{N}$, we say t is a *subset sum* of A if t can be written as the sum of elements from a subset of A. That is, if there is a subset of indices $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} a_i = t$. For example, if $A = \{1, 3, 5, 7\}$ then $12 = 5 + 7$ is a subset sum of A, but 2 is not. We want to design an algorithm that determines whether a given number $t \in \mathbb{N}$ is a subset sum of A. Specifically, the algorithm if given $A = \{a_1, \ldots, a_n\}$ and $t \in \mathbb{N}$ as input, should output 1 if t is a subset sum of A, and 0 otherwise. To this end, we will use dynamic programming. For your DP algorithm, use the name SubsetSum for the (potentially multi-dimensional) array which stores values of the subproblems.

1. Describe in words what SubsetSum should be and state its dimension. In addition, state the base cases and their values.

2. Give and justify the recurrence SubsetSum should satisfy.

3. Justify the run time of your algorithm to compute SubsetSum as a big-Θ expression. (Hint: you may want to make use of the quantity $M = \sum_{i=1}^{n} a_i$, which is the sum of all elements in A).

Question 4: Alternating Coins (2+5+6+2=15 points)

Consider a row of n coins of values $v_1 \ldots v_n$, where n is even. A turn based game is being played between 2 players where they alternate turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. We want an algorithm that determines the maximum possible amount of money you can definitely win if you move first (assume your opponent will play to maximize the amount they get). To this end, we will use dynamic programming. For your DP algorithm, use the name MaxGain for the (potentially multi-dimensional) array which stores values of the subproblems.

1. Now let's play this game against an extremely smart AI. Suppose $n = 6$, the coin values are 1 or 2, and the given coin arrangement, represented as an array A, is

   $$A = [1, 1, 1, 2, 2, 2] \ .$$

   If you move first and try to maximize your gains, how much money can you definitely win regardless of the opponent's moves? Describe in words your optimal play strategy for this particular instance.

2. Describe in words what MaxGain should be and state its dimension. In addition, state the base cases and their values.

3. Give and justify the recurrence MaxGain should satisfy.

4. Justify the run time of your algorithm to compute MaxGain as a big-Θ expression.

Honors Questions

Question 5: Honors

(**) Give an algorithm that computes the n-th Fibonacci number F_n in $O(\log n)$ time. (Here again we are assuming each arithmetic operation takes unit time; the n-th Fibonacci number is $\Theta(n)$ digits long, so if we took into account the word size, we would obviously need at least $\Omega(n)$ time to compute it)

Question 6: Honors

(**) Given an $m \times n$ size rectangle, we wish to divide it into non-overlapping square pieces, using the least possible number of pieces. For example a 4×5 rectangle needs at least 5 pieces: a big 4×4 square and 4 small 1×1 squares.

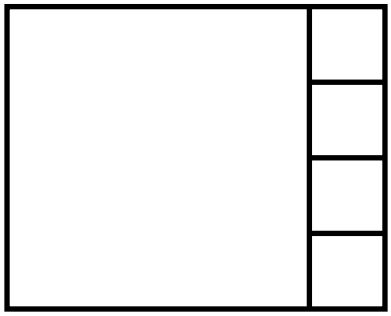

Figure 1: Dividing up a 4×5 rectangle using 5 squares

We might try to solve this using the idea of trying to cut the $n \times m$ size rectangle into two pieces either by a vertical or a horizontal cut. The following pseudo-code tries all possible horizontal and vertical cuts and picks the best case out of these.

MinTilings(m, n):
if $m == n$ **then**
  return 1.
end if
$H_{min} = \text{INT}_{\text{MAX}}$.
$V_{min} = \text{INT}_{\text{MAX}}$.
for $i = 1$ **to** $\lfloor m/2 \rfloor$ **do**
  $H_{\min} = \min(H_{\min}, \text{MinTilings}(i, n) + \text{MinTilings}(m - i, n))$.
end for
for $j = 1$ **to** $\lfloor n/2 \rfloor$ **do**
  $V_{min} = \min(V_{\min}, \text{MinTilings}(m, j) + \text{MinTilings}(m, n - j))$.
end for
return $\min(H_{\min}, V_{\min})$.

Use dynamic programming to improve this recursive method by avoiding re-computation for sub-problems that have already been solved. What is the running time of your algorithm?

Does this approach work to find the *optimal* tiling? Why or why not? (Hint: Consider the figure below)

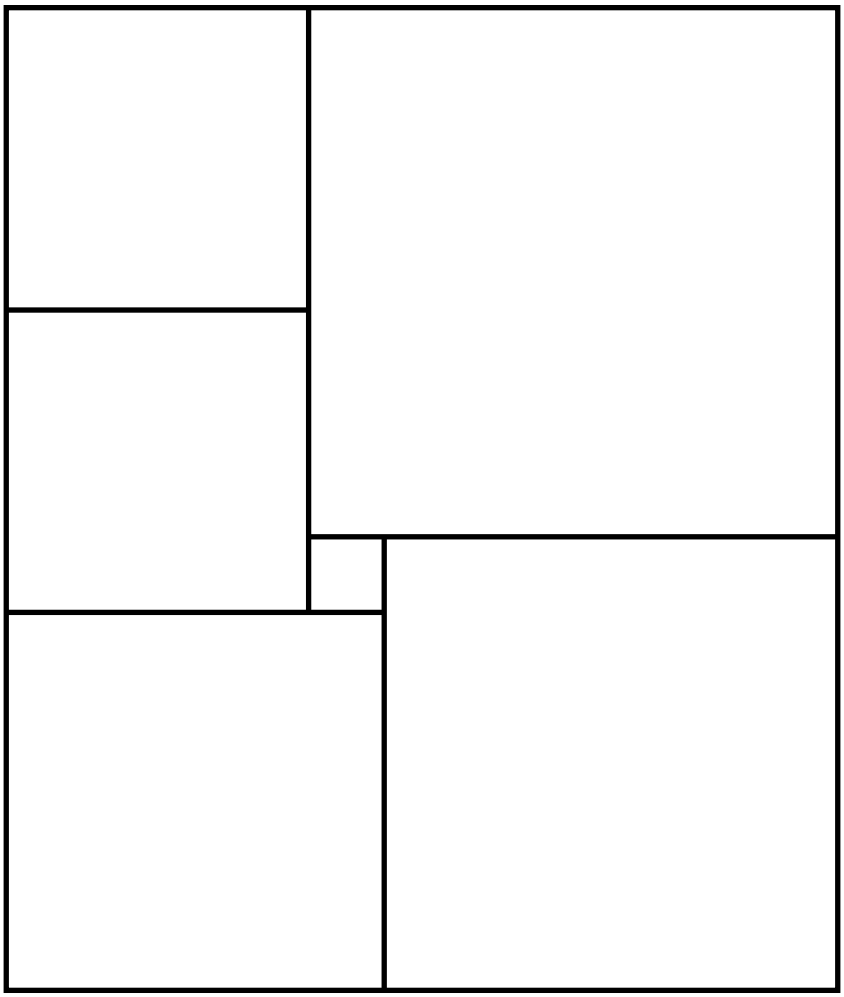

Figure 2: Dividing up a 11×13 rectangle using 6 squares (two 4×4 squares and one each of 1×1, 5×5, 6×6 and 7×7 squares)