Homework 5: Due October 16 (11:59 p.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 to 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 n-th Fibonacci number F_n 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 (e.g., in polynomial time rather than the naive $2^{\Omega(n)}$).
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 that finds a way to cut the rod maximizing our **profit**. For your DP algorithm, use the name MaxProfit for the (potentially multi-dimensional) array which stores values of the subproblems.

1. What is the dimension of the array MaxProfit?
2. State the base cases and their values.
3. Give and justify the recurrence MaxProfit should satisfy.
4. Justify the run time of your algorithm to compute MaxProfit as a big-Θ expression.

Question 3: Subset Sum (1+1+1+5+2=10 points)

Let $A = [a_1, \ldots, a_n]$ be an array of n natural numbers. Given a number $t \in \mathbb{N}$, we say t is a *subset sum* of A 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 is a subset sum of A since $5+7 = 12$, 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, when given $A = [a_1, \ldots, a_n]$ and $t \in \mathbb{N}$ as input, the algorithm should output 1 if t is a subset sum of A, and 0 otherwise. The algorithm should output an answer (either 0 or 1) for *any* given $t \in \mathbb{N}$.

We will use dynamic programming to solve this problem. The subproblem in your DP algorithm should be SUBSETSUM, which is defined as

$$\textsc{SubsetSum}[i][t] = \begin{cases} 1 & \text{if } t \text{ is a subset sum of the first } i \text{ elements } A_{1:i} = [a_1, \ldots, a_i], \\ 0 & \text{otherwise.} \end{cases}$$

Also, let us define $M = \sum_{i=1}^{n} a_i$, which is the sum of all numbers in A. The quantity M may appear in the size of the DP array and the run-time of your algorithm.

1. Suppose $A = [2, 1, 3]$. Fill out the following table for A. Note that we define the sum over the empty set to be 0. The pre-filled entries in the top left corner correspond to $\textsc{SubsetSum}[0][0] = 1$ and $\textsc{SubsetSum}[0][1] = 0$.

i \ t	0	1	2	3	4	5	6
0	1	0					
1							
2							
3							

Table 1: Fill in the entries for $\textsc{SubsetSum}[i][t]$.

2. In general, what is the size of the DP array containing values for SUBSETSUM? (Hint: What is the largest number you could even conceivably obtain as a subset sum of A? You may want to make use of M for your expression.)

3. State the base cases and their values.

4. Give and justify the recurrence SUBSETSUM should satisfy. (Hint: Given inputs i and t, we can either use the i-th element a_i or not use it in the subset sum for t. If we use a_i, then the remaining subset sum is $t - a_i$ which is less than t.)

5. Justify the run time of your algorithm to compute SUBSETSUM as a big-Θ expression. You may want to use M for your expression.

Honors Questions

Question 4: 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 5: 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.

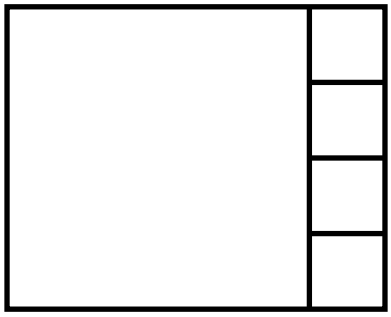

Figure 1: Dividing up a 4×5 rectangle using 5 squares

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)

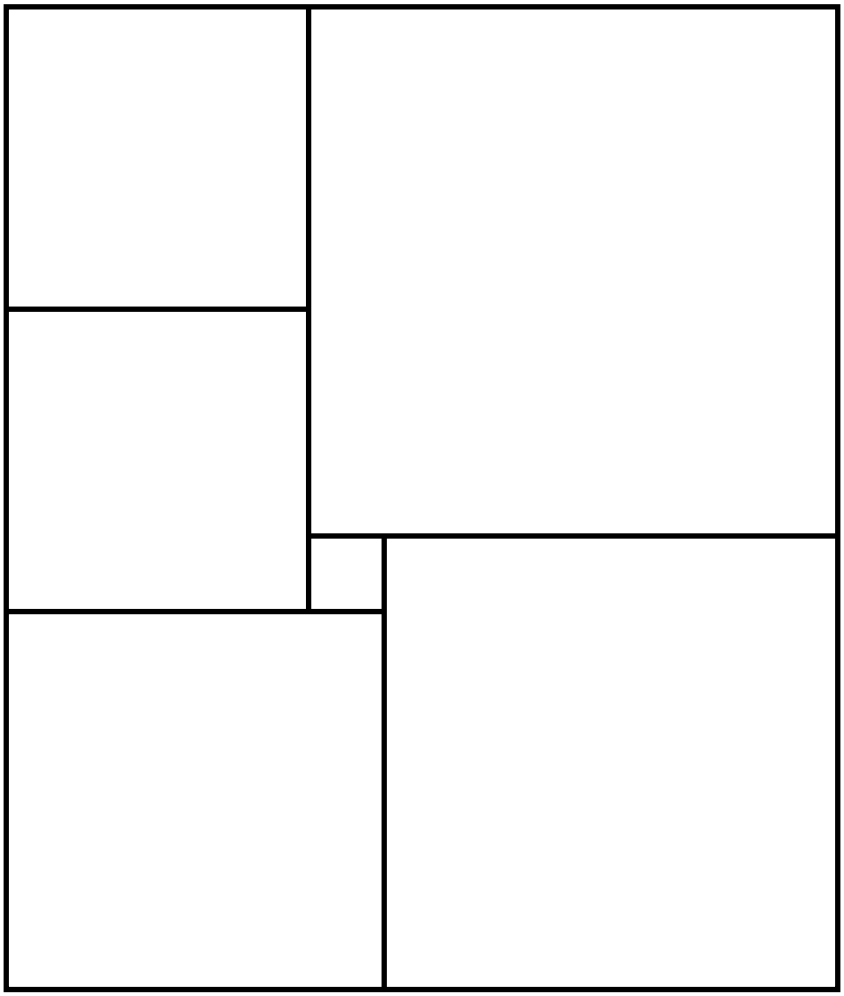

Figure 2: Dividing up a 11×13 rectangle using 6 squares (two 4×4 squares and one each of 1×1, 5×5, 6×6 and 7×7 squares)