CSC4120 Spring 2024 - Written Homework 9

Yohandi 120040025
Andrew Nathanael 120040007

April 17, 2024

Problem 1. Analyzing a game

There are N pots of gold arranged linearly. Alice and Bob are playing the following game. They take alternate turns, and in each turn they remove (and *win*) one of the two pots at the two ends of the sequence. Alice plays first and both players know the amount of gold in each pot.

(a) Design an algorithm to find the maximum amount of gold that Alice can *assure* herself of winning. In this case Alice maximizes the minimum amount she can guarantee to win *for any possible strategy* of Bob. This corresponds to a *max-min strategy* for Alice.

(b) Design an algorithm to find the maximum amount of gold that Alice can win assuming that Bob *acts also strategically* and solves the same problem of maximizing his total amount of gold assuming Alice is strategic. This is the case of a game and we will find a *Nash equilibrium strategy*!

In both cases write a recurrence relating the solution of a general subproblem to solutions of smaller subproblems. Then, analyze the running time of your algorithm, including the number of subproblems and the time spent per subproblem.

(a) In this case, Alice attempts to maximize the gold she can guarantee regardless of Bob's strategy. We define $\texttt{dp}[i][j]$ as the maximum gold Alice can guarantee when only the pots from index i to j are left.

The recursive relation can be written as:

$$
\begin{aligned}
\texttt{dp}[i][j] = \max(& \\
& \min(\texttt{dp}[i+1][j-1], \texttt{dp}[i+2][j]) + \texttt{pots}[i], \\
& \min(\texttt{dp}[i+1][j-1], \texttt{dp}[i][j-2]) + \texttt{pots}[j] \\
)&
\end{aligned}
$$

- The first term includes the possibility of Alice taking the pot on the left end, which is $\texttt{pots}[i]$. Now, the range of consideration for Bob is $[i+1, j]$, either taking $\texttt{pots}[i+1]$ or $\texttt{pots}[j]$. As we aim to maximize Alice's profits alone, it is unnecessary for us to consider Bob's maximizing his profits. In the worst-case scenario, Alice only gets the minimum of $\texttt{dp}[i+2][j]$ and $\texttt{dp}[i+1][j-1]$, corresponding to Bob's choice: $\texttt{pots}[i+1]$ or $\texttt{pots}[j]$, respectively.
- The second term includes a similar possibility, but this time for Alice taking the pot on the right end. The corresponding range consideration for Bob is now $[i, j-1]$; hence, the worst-case scenario for Alice is the minimum of $\texttt{dp}[i+1][j-1]$ and $\texttt{dp}[i+2][j]$.

The base cases:

- $\texttt{dp}[i][i] = \text{pots}[i]$: If there is only one pot left, Alice takes it.
- $\texttt{dp}[i][i+1] = \max(\text{pots}[i], \text{pots}[i+1])$: If there are two pots left, Alice picks the one with the maximum gold.

There are $\mathcal{O}(N^2)$ states with each transition takes $\mathcal{O}(1)$; hence, the overall dynamic programming algorithm takes $\mathcal{O}(N^2)$ operations.

(b) The same algorithm in part (a) can be used. The $\texttt{dp}[i][j]$ is defined as Alice, who has the current turn and, considering the range $[i, j]$, gets the maximum profits. Then, the same dynamic programming can be applied for Bob if range $[i, j]$ is the range that is being considered, and it is Bob's move.

Due to Bob's objective is to maximize his profits, the consequence of Alice getting the worst-case scenario (mentioned in part (a)) always happens; hence, the formula remains the same.

Problem 2.

A *contiguous subsequence* of a list S is a subsequence made up of consecutive elements of S. For instance, if S is $5, 15, -30, 10, -5, 40, 10$, then $15, -30, 10$ is a contiguous subsequence but $5, 15, 40$ is not. Give a linear-time algorithm for the following task:

Input: A list of numbers, $a_1, a_2, \ldots, a_n$.

Output: The contiguous subsequence of maximum sum (a subsequence of length zero has sum zero).

For the preceding example, the answer would be $10, -5, 40, 10$, with a sum of 55. Note that we never get a negative value since we can always have the zero value by choosing a subsequence of length zero.

The above is a well-known problem, named Maximum Subarray Problem, which involves finding the maximum sum contiguous subsequence in an array and it can be solved using Kadane's Algorithm in $\mathcal{O}(n)$. The algorithm is as follows:

- Initialize two variables: $\texttt{max\_current} = \texttt{max\_global} = A[1]$.
- For i being iterated from 2 to n:
  - Update $\texttt{max\_current}$ as $\max(\texttt{max\_current} + A[i], A[i])$.
  - Update $\texttt{max\_global}$ as $\max(\texttt{max\_global}, \texttt{max\_current})$.
- Return $\max(\texttt{max\_global}, 0)$ as our answer.

Define $\texttt{max\_current}$ in iteration i as the maximum sum of subarrays that include the i-th element. The choice of $\texttt{max\_current} + A[i]$ and $A[i]$ open up the possibility of considering i as a pointer for a new starting array. The choice of $A[i]$ is made when $\texttt{max\_current} < 0$, meaning that it is best to cut off the last included element and start over. The overall time complexity is $\mathcal{O}(n)$ and the space complexity is $\mathcal{O}(1)$.

Problem 3.

Suppose two teams, A and B, are playing a match to see who is the first to win n games (for some particular n). We can suppose that A and B are equally competent, so each has a 50% chance of winning any particular game. Suppose they have already played $i+j$ games, of which A has won i and B has won j ($i, j \leq n$). Give an efficient algorithm to compute the probability that A will go on to win the match. For example, if $i = n-1$ and $j = n-3$ then the probability that A will win the match is 7/8, since it must win any of the next three games.

Given that team A and team B each have a 50% chance of winning any game, the probability that team A will win the match from a state (i, j), where i and j are the number of games won by A and B respectively, can be calculated using dynamic programming. The recursive formulation is:

$$P(i,j) = \begin{cases} 1 & \text{if } i = n \\ 0 & \text{if } j = n \\ 0.5 \times P(i+1,j) + 0.5 \times P(i,j+1) & \text{otherwise} \end{cases}$$

, where:

- $P(n, j) = 1$ for all $j < n$ (A has already won the match).
- $P(i, n) = 0$ for all $i < n$ (B has already won the match).

The probabilities are stored in a 2D array $dp[i][j]$ with the size $(n+1) \times (n+1)$, and the array is filled in reverse order from known winning conditions. The final answer, $P(0, 0)$, gives the probability that A will win the match starting from no games won.

Problem 4.

You are given a rectangular piece of cloth with dimensions $X \times Y$, where X and Y are positive integers, and a list of n products that can be made using the cloth. For each product $i \in [1, n]$, you know that a rectangle of cloth of dimensions $a_i \times b_i$ is needed and that the final selling price of the product is c_i. Assume a_i, b_i, and c_i are all positive integers. You have a machine that can cut any rectangular piece of cloth into two pieces either horizontally or vertically. Design an algorithm that determines the best return on the $X \times Y$ piece of cloth, that is, a strategy for cutting the cloth so that the products made from the resulting pieces give the maximum sum of selling prices. You are free to make as many copies of a given product as you wish, or none if desired.

This problem can be approached using dynamic programming. Let's define $f(x, y)$ as the maximum selling price obtainable from a rectangle of dimensions $x \times y$. The solution is found by calculating $f(X, Y)$, using the following recursive formula:

$$f(x,y) = \max\left(\max_{1\le i\le n}\{c_i \mid (a_i \le x \wedge b_i \le y) \vee (b_i \le x \wedge a_i \le y)\},\right.$$
$$\left.\max_{k=1}^{x-1}\{f(k,y)+f(x-k,y)\}, \max_{k=1}^{y-1}\{f(x,k)+f(x,y-k)\}\right)$$

The first term inside the max between two terms is about fitting one product inside the current $x \times y$ cloth. The second term refers to all possibilities of cuts (horizontal cuts that divide $x \times y$ to $k \times y$ and $(x-k) \times y$ for all $k \in \{1,\dots,x-1\}$ and vertical cuts that divide $x \times y$ to $x \times k$ and $z \times (y-k)$ for all $k \in \{1,\dots,y-1\}$).

The base cases are:

$$f(1,y) = \max(0, \max_{1\le i\le n}\{c_i \mid (a_i \le 1 \wedge b_i \le y) \vee (b_i \le 1 \wedge a_i \le y)\})$$

and

$$f(x,1) = \max(0, \max_{1\le i\le n}\{c_i \mid (a_i \le x \wedge b_i \le 1) \vee (b_i \le x \wedge a_i \le 1)\})$$

, for any x, y.

Let `dp`$[x][y]$ represents the value of $f(x,y)$. Then, the dynamic programming implementation involves filling `dp`. The table is initialized to zero and updated for each subproblem:

```
function maxProfit(X, Y, n, products):
    let dp = new array [1...X][1...Y] filled with zeroes
    for x from 1 to X:
        for y from 1 to Y:
            for each product (a, b, c) in products:
                if (a <= x and b <= y) or (b <= x and a <= y):
                    dp[x][y] = max(dp[x][y], c)
                for k from 1 to x-1:
                    dp[x][y] = max(dp[x][y], dp[k][y] + dp[x-k][y])
                for k from 1 to y-1:
                    dp[x][y] = max(dp[x][y], dp[x][k] + dp[x][y-k])
    return dp[X][Y]
```

The above algorithm works in $\mathcal{O}(X \times Y \times \max(X, Y, n))$ time, with a space complexity of $\mathcal{O}(X \times Y)$.