Solutions for Exercise Sheet 11

Handout: November 28th — Deadline: December 5th

Question 11.1 (0.25 marks)

Consider a modification of the rod-cutting problem where in addition to a price p_i for each rod, each cut incurs a fixed cost c. The revenue for a solution is now the sum of the prices of the individual pieces minus the costs of making the cuts. Give a Dynamic Programming Bottom-Up algorithm to solve this problem.

Solutions: We need to account for costs of cutting the rod every time we work out in the inner loop that cutting the rod into smaller pieces $p[i]$ and $r[j-i]$ would lead to a higher revenue than the current best. This should happen in each iteration of the inner loop except for the last one when the rod is not cut at all i.e., when $j = i$ and $r[j-i] = 0$. An easy way to do this is to put the price of the uncut piece of size j directly in q at the beginning in line 4 and complete the inner loop one step early at $j-1$ instead of at j.

CUT-ROD-WITH-CUT-PENALTIES(p, n, c)

```
1: Let r[0, ..n] be a new array
2: r[0] = 0
3: for j = 1 to n do
4:       q = p[j]
5:       for i = 1 to j − 1 do
6:             q = max(q, p[i] + r[j − i] − c)
7:       r[i] = q
8: return r[n]
```

Question 11.2 (0.25 marks)

Modify MEMOIZED-CUT-ROD so that it also returns the actual solution rather than just its value.

Solutions:

EXTENDED-MEMOIZED-CUT-ROD(p, n)

```
1: Let r[0, ..n] and s[1..n] be new arrays
2: for i = 1 to n do
3:     r[i] = −∞
4:     s[i] = 0
5: q = EXTENDED-MEMOIZED-CUT-ROD-AUX(p, n, r, s)
6: Print q
7: while n > 0 do
8:     Print s[n]
9:     n = n − s[n]
```

EXTENDED-MEMOIZED-CUT-ROD-AUX(p, n, r, s)

```
1: if r[n] ≥ 0 then
2:     return r[n]
3: if n == 0 then
4:     q = 0
5: else
6:     q = −∞
7:     for i = 1 to n do
8:         t = p[i]+ EXTENDED-MEMOIZED-CUT-ROD-AUX(p, n − i, r, s)
9:         if q < t then
10:            q = t
11:            s[n] = i
12: r[n] = q
13: return r[n]
```

Question 11.3 (0.25 marks)

Provide the pseudo-code of an $O(n)$ time Dynamic Programming algorithm for calculating the n_{th} Fibonacci number. Draw the subproblem graph. How many vertices and edges does the subproblem graph contain?

Solutions:

FIBONACCI(n)

```
1: Let Fib[0, ..n] be a new array
2: Fib[0] = Fib[1] = 1
3: for i = 2 to n do
4:       Fib[i] = Fib[i − 1]+ Fib[i − 2]
5: return Fib[n]
```

There are $n + 1$ vertices in the graph: $v_0, v_1, \ldots, v_n$.

v_0 and v_1 have 0 leaving edges. The remaining have two leaving edges each (i.e., each v_i has an edge to v_{i-1} and one to v_{i-2} for all $2 \leq i \leq n$.

Question 11.4 (1 mark)

After retiring from a large company, Mr Mortimer wonders how much money he could have made if he had invested his life savings in shares of his company. He looks back at the share prices over the n days of his employment and asks himself how much his investment could have returned if he had—somehow—known the best time to buy and sell his shares. Mortimer doesn't like trading shares, so he would only ever buy once and sell once and assume that his life savings is a fixed amount.

Let $a_1, \ldots, a_n$ describe the difference of share prices over time: a_i is the amount of money Mr Mortimer would have gained if he had held on to shares on day i. Note that a_i can be negative, in which case day i would have been a loss. Assume $a_i \neq 0$. Let $f(i, j) = \sum_{k=i}^{j} a_k$ be the money earned if Mortimer had bought shares at the start of day i of his employment and sold them at the end of day j. Mortimer wants to find the maximum return $\max\{f(i, j) \mid 1 \leq i \leq j \leq n\}$.

Example: for $a_1, \ldots, a_n = +5, -3, -4, +8, -1, +12, -6, +4, +4, -14, +2, +8$, Mortimer's maximum return would have been $f(4, 9) = 8 + (-1) + 12 + (-6) + 4 + 4 = 21$ pounds.

(a) Design an algorithm in pseudocode that computes the maximum return using dynamic programming in time $O(n)$, following the hints below. Explain your solution.

(b) Show that your algorithm runs in time $O(n)$.

Hints: Use a bottom-up approach, tabulating for each day k:

- the maximum return A_k for an investment up to day k (buying and selling up to day k, formally $\max\{f(i, j) \mid 1 \leq i \leq j \leq k\}$) and
- the maximum return B_k up to day k for an *ongoing* investment: still holding on to shares on day k (formally $\max\{f(i, k) \mid 1 \leq i \leq k\}$).

Work out how A_1 and B_1 can be initialised, and work out Bellman equations showing how, for $k \geq 2$, A_k and B_k can be computed based on the input and values of A and B that you have tabulated earlier. Pay attention to the order in which to tabulate values. Don't forget to state how to compute the final output from the tabulated values.

Solutions: The only solution for A_1 is $i = j = 1$ with a return of a_1. So $A_1 = a_1$. The only solution for B_1 is the same, hence $B_1 = a_1$.

For $k \geq 2$, there are two ways of generating a solution that ends with holding shares on day k:

- using the best ongoing investment from day $k-1$, and holding on to shares on day k as well. The return of this solution is $B_{k-1} + a_k$, or
- buying shares on day k. This solution has a return of a_k.

The best return B_k is the maximum of these two returns: $B_k = \max\{B_{k-1} + a_k, a_k\}$.

The best value of a completed investment up to day k is either

- the best completed investment up to day $k-1$ with value A_{k-1} or
- the best ongoing investment up to day k with value B_k.

The best return A_k is the maximum of these two returns: $A_k = \max\{A_{k-1}, B_k\}$.

The algorithm uses the above formulas to compute B_k and A_k (in this order) for k increasing from 2 to n. The final output is then the best completed investment up to day n, A_n. In pseudocode:

MAXIMUM-RETURN$(a_1, \ldots, a_n)$

1: Let A, B be two arrays of n elements.
2: Let $A[1] = B[1] = a_1$.
3: **for** $k = 2$ to n **do**
4: Let $B[k] = \max\{B[k-1] + a_k, a_k\}$.
5: Let $A[k] = \max\{A[k-1], B[k]\}$.
6: **return** $A[n]$

The running time is $\Theta(n)$ as there is a single for loop executed $\Theta(n)$ times and one iteration runs in $\Theta(1)$ time.

Question 11.5 (0.5 marks)

Implement MEMOIZED-CUT-ROD(p, n) and BOTTOM-UP-CUT-ROD(p, n) such that they print both the optimal solution and its value. The algorithms take in input the number of different lengths n and an array $p[0..n]$ with the price of a rod of length i in position $p[i]$.