Exercise V, Algorithms 2024-2025

These exercises are for your own benefit. Feel free to collaborate and share your answers with other students. There **are many problems on this set, and it is not expected that you solve all of them during two hours**. Instead, train on the topics you find most difficult and also solve more problems at home! Also ask for help if you get stuck for too long. Problems marked * are more difficult but also more fun :).

1 Consider the problem of calculating the value of $f(n)$ where f is defined for non-negative integers recursively as follows:

$$f(n) = \begin{cases} 1 & \text{if } n = 0, \\ \sum_{i=0}^{n-1} f(i) & \text{if } n \geq 1. \end{cases}$$

1a Devise a dynamic programming algorithm to calculate $f(n)$ using the "top-down with memoization" approach.

Solution:

```
F-Top-Down(n)
1 let r[0..n] be a new array
2 r[0] = 1
3 for i = 1 to n
4     r[i] = −1
5 return Compute-F(n, r)
```

```
Compute-F(n, r)
1 if r[n] ≥ 0 then
2     return r[n]
3 s = 0
4 for i = 0 to n − 1
5     s = s+Compute-F(i, r)
6 r[n] = s
7 return r[n]
```

1b Devise a dynamic programming algorithm to calculate $f(n)$ using the "bottom-up" method.

Solution:

```
F-Bottom-Up(n)
1 let r[0..n] be a new array
2 r[0] = 1
3 for i = 1 to n
4     r[i] = 0
5     for j = 0 to i − 1
6         r[i] = r[i] + r[j]
7 return r[n]
```

1c What is the running time of your algorithms?

Solution: The "bottom-up" solution has two nested loops, where the number of iterations of the inner loop is:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Therefore, the running time is $\Theta(n^2)$. The "top-down" solution also solves each subproblem just once (immediately returning from the recursive call when it's already solved), so the number of iterations of the for loop forms the same sum as above, and the running time is again $\Theta(n^2)$.

Rod Cutting

2 (Exercise 15.1-2) Show, by means of a counterexample, that the following "greedy" strategy does not always determine an optimal way to cut rods. Define the ***density*** of a rod of length i to be p_i/i, that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length i, where $1 \leq i \leq n$, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.

Solution: Assume that the price function is given as:

$$p_i = \begin{cases} 1 & \text{if } i = 1, \\ 5 & \text{if } i = 2, \\ 7 & \text{if } i = 3 \end{cases}$$

The density function is then:

$$d_i = p_i/i = \begin{cases} 1 & \text{if } i = 1, \\ 2.5 & \text{if } i = 2, \\ 2.33 & \text{if } i = 3 \end{cases}$$

The greedy strategy will cut the rod of length 3 into pieces of size 2 and 1 for the total price of 6, while the optimal solution is to leave the rod of size 3 with the price of 7.

3 (Based on Exercise 15.1-3) Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a cost of c. The revenue associated with a solution is now the sum of prices of the pieces minus the costs of making the cuts.

3a Write the optimal revenue r_n of a rod of length n in terms of optimal revenues from shorter rods. In other words, give a recursive formulation of the optimal revenue.

Solution: The optimal solution is either the rod of length n or a combination of two optimal solutions whose length totals to n. Assuming that we view the decomposition as a piece of length i and the remainder of length $n - i$, the formula is:

$$r_n = \max(p_n, \max_{1 \leq i \leq n-1} (p_i + r_{n-i} - c))$$

3b Describe how to calculate r_n using the "top-down with memoization" approach. In addition, analyze asymptotically your algorithm's time and space complexity.

Solution:

```
MEMOIZED-CUT-ROD(p, n, c)
1  let r[0..n] be a new array
2  r[0] = 0
3  for i = 1 to n
4      r[i] = -∞
5  return MEMOIZED-CUT-ROD-AUX(p, n, c, r)

MEMOIZED-CUT-ROD-AUX(p, n, c, r)
1  if r[n] ≥ 0
2      return r[n]
3  q = p[n]
4  for i = 1 to n - 1
5      q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, c, r) - c)
6  r[n] = q
7  return r[n]
```

The solution solves each subproblem only once. Therefore the number of iterations of the for loop forms an arithmetic series, and the running time is $\Theta(n^2)$ iterations. The amount of extra space used is $\Theta(n)$ for the array r.

3c Describe how to calculate r_n using the "bottom-up" approach. In addition, analyze asymptotically your algorithm's time and space complexity.

Solution:

```
BOTTOM-UP-CUT-ROD(p, n, c)
1  let r[0..n] be a new array
2  r[0] = 0
3  for i = 1 to n
4      q = p[i]
5      for j = 1 to i - 1
6          q = max(q, p[j] + r[i - j] - c)
7      r[i] = q
8  return r[n]
```

The number of iterations of the inner loop forms an arithmetic series, so the running time is again $\Theta(n^2)$. The space complexity is again $\Theta(n)$.

3d How would you modify the algorithms to return not only the value but the actual solution too?

Solution: In order to be able to return the actual solution, along with the maximal revenue $r[i]$ for every length i, we need to also store $s[i]$, the size of the first piece to cut off.

```
BOTTOM-UP-CUT-ROD(p, n, c)
1  let r[0..n] and s[0..n] be new arrays
2  r[0] = 0
3  for i = 1 to n
4      q = p[i]
5      for j = 1 to i - 1
6          if q < p[j] + r[i - j] - c
```

```
7          q = p[j] + r[i − j] − c
8          s[i] = j
9      r[i] = q
10 return r and s

CUT-ROD-PRINT-SOLUTION(p, n, c)
1 (r, s) = BOTTOM-UP-CUT-ROD(p, n, c)
2 while n > 0
3     print s[n]
4     n = n − s[n]
```

Matrix-Chain Multiplication

4 (Based on Exercise 15.2-1) Consider the dynamic programming algorithm for matrix-chain multiplication taught in class. Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6\rangle$. Find the answer (and explain how the algorithm proceeds) by filling in the "memory" table in the same way as the algorithm.

Solution: From the sequence of dimensions given we can identify six matrices with the following dimensions:

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	5×10	10×3	3×12	12×5	5×50	50×6

The algorithm proceeds using the following formula:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{i<j.} \end{cases}$$

The table is computed starting with the diagonal of the matrix, i.e. it computes first $m[i,i] = 0$ for $i = 1, 2, ..., n$. Then this information is used to compute $m[i, i+1]$ for $i = 1, 2, ..., n-1$ so on and so forth. The resulting tables, m and s, will be filled as follows:

m=

i \ j	1	2	3	4	5	6
1	0	150	330	405	1655	2010
2		0	360	330	2430	1950
3			0	180	930	1770
4				0	3000	1860
5					0	1500
6						0

s=

i \ j	2	3	4	5	6
1	1	2	2	4	2
2		2	2	2	2
3			3	4	4
4				4	4
5					5

Therefore the optimal parenthesization is $((A_1A_2)((A_3A_4)(A_5A_6)))$.

5 (Exercise 15.2-6) Show that a full parenthesization of an n-element expression has exactly $n-1$ pairs of parentheses.

A product of matrices is *fully parenthesized* if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. For example, if the chain of matrices is $\langle A_1, A_2, A_3, A_4\rangle$, then we can fully parenthesize the product $A_1A_2A_3A_4$ in five

distinct ways:

$(A_1(A_2(A_3A_4)))$, $(A_1((A_2A_3)A_4))$, $((A_1A_2)(A_3A_4))$, $((A_1(A_2A_3))A_4)$, $(((A_1A_2)A_3)A_4)$.

(Hint: use induction on the number of elements)

Solution: Recall the definition of fully parenthesized matrices: A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrices surrounded by a parentheses.

Therefore by induction

Base case:

1. if $n \leftarrow 1$, there are no parentheses ($n - 1 = 0$ parentheses)

Assume that this relationship holds for $n - 1$ matrices, i.e., $n - 1$ matrices have $n - 2$ pairs of parentheses.

According to the definition, the product of two fully parenthesized matrices is fully parenthesized by surrounding them by a parentheses. Let the number of matrices in the first fully parenthesized matrices be i, so the number of matrices in the other one would be $n - i$. By the induction hypothesis the first one needs $i - 1$ parentheses, and second one needs $n - i - 1$, plus an additional parentheses to merge both parenthesizations. Hence we need in total $(i - 1) + (n - i - 1) + 1 = n - 1$ which concludes the proof.