Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Markus Püschel, David Steurer
Gleb Novikov, Tommaso d'Orsi, Ulysse Schaller, Rajai Nasser

11 October 2021

Algorithms & Data Structures — Exercise sheet 3 — HS 21

The solutions for this sheet are submitted at the beginning of the exercise class on October 18th.

Exercises that are marked by * are challenge exercises. They do not count towards bonus points. In this sheet, the only exercises that are counted towards bonus points are: 3.3.(a-c) and 3.4.(a-d). You can use results from previous parts without solving those parts.

Exercise 3.1 *Some properties of O-Notation.*

Let $f : \mathbb{R}^+ \to \mathbb{R}^+$ and $g : \mathbb{R}^+ \to \mathbb{R}^+$.

a) Show that if $f \leq O(g)$, then $f^2 \leq O(g^2)$. You can assume that $\lim_{x\to\infty} \frac{f(x)}{g(x)} = C \in \mathbb{R}_0^+$.

Solution:

$$\lim_{x\to\infty} \frac{f^2(x)}{g^2(x)} = \lim_{x\to\infty} \frac{f(x)}{g(x)} \cdot \lim_{x\to\infty} \frac{f(x)}{g(x)} = C^2 \in \mathbb{R}_0^+ ,$$

hence by Theorem 1 of Exercise sheet 1, $f^2 \leq O(g^2)$.

b) Give an example where $f \leq O(g)$, but $2^f \not\leq O(2^g)$.

Solution: Consider $f(n) = 2n$, $g(n) = n$. Obviously, $f \leq O(g)$. However,

$$\lim_{n\to\infty} \frac{2^{f(n)}}{2^{g(n)}} = \lim_{n\to\infty} \frac{2^{2n}}{2^n} = \lim_{n\to\infty} 2^n = \infty ,$$

hence by Theorem 1 of Exercise sheet 1, $2^f \not\leq O(2^g)$.

Another important example is $f(n) = \log_2 n$ and $g(n) = \log_4 n$. As we already showed, $f \leq O(g)$. However, $2^{f(n)} = n$ and $2^{g(n)} = \sqrt{n}$, so by Theorem 1 of Exercise sheet 1, $2^f \not\leq O(2^g)$.

Exercise 3.2 *Iterative squaring.*

In this exercise you are going to develop an algorithm to compute powers a^n, with $a \in \mathbb{Z}$ and $n \in \mathbb{N}$, efficiently. For this exercise, we will treat multiplication of two integers as a single elementary operation, i.e., for $a, b \in \mathbb{Z}$ you can compute $a \cdot b$ using one operation.

a) Assume that n is even, and that you already know an algorithm $A_{n/2}(a)$ that efficiently computes $a^{n/2}$, i.e., $A_{n/2}(a) = a^{n/2}$. Given the algorithm $A_{n/2}$, design an efficient algorithm $A_n(a)$ that computes a^n.

Solution:

Algorithm 1 $A_n(a)$

$x \leftarrow A_{n/2}(a)$

return $x \cdot x$

b) Let $n = 2^k$, for $k \in \mathbb{N}_0$. Find an algorithm that computes a^n efficiently. Describe your algorithm using pseudo-code.

Solution:

Algorithm 2 Power(a, n)

```
if n = 1 then
    return a
else
    x ← Power(a, n/2)
    return x · x
```

c) Determine the number of elementary operations (i.e., integer multiplications) required by your algorithm for part b) in O-notation. You may assume that bookkeeping operations don't cost anything. This includes handling of counters, computing $n/2$ from n, etc.

Solution: Let $T(n)$ be the number of elementary operations that the algorithm from part b) performs on input a, n. Then

$$T(n) \leq T(n/2) + 1 \leq T(n/4) + 2 \leq T(n/8) + 3 \leq \ldots \leq T(1) + \log_2 n \leq O(\log n)\,.$$

d) Let Power(a, n) denote your algorithm for the computation of a^n from part b). Prove the correctness of your algorithm via mathematical induction for all $n \in \mathbb{N}$ that are powers of two.

In other words: show that Power$(a, n) = a^n$ for all $n \in \mathbb{N}$ of the form $n = 2^k$ for some $k \in \mathbb{N}_0$.

- **Base Case.**
  Let $k = 0$. Then $n = 1$ and Power$(a, n) = a = a^1$.
- **Induction Hypothesis.**
  Assume that the property holds for some positive integer k. That is, Power$(a, 2^k) = a^{2^k}$.
- **Inductive Step.**
  We must show that the property holds for $k + 1$.

$$\text{Power}(a, 2^{k+1}) = \text{Power}(a, 2^k) \cdot \text{Power}(a, 2^k) \overset{\text{I.H.}}{=} a^{2^k} \cdot a^{2^k} = a^{2^{k+1}}.$$

  By the principle of mathematical induction, this is true for any integer $k \geq 0$ and $n = 2^k$.

e)* Design an algorithm that can compute a^n for a general $n \in \mathbb{N}$, i.e., n does not need to be a power of two.

Hint: *Generalize the idea from part a) to the case where n is odd, i.e., there exists $k \in \mathbb{N}$ such that $n = 2k + 1$.*

Solution:

Algorithm 3 Power(a, n)

```
if n = 1 then
    return a
else
    if n is odd then
        x ← Power(a, (n − 1)/2)
        return x · x · a
    else
        x ← Power(a, n/2)
        return x · x
```

f)* Prove correctness of your algorithm in e) and determine the number of elementary operations in O-Notation. As before, you may assume that bookkeeping operations don't cost anything.

Solution: Let's prove correctness.

- **Base Case.**
  Let $n = 1$. Then $\text{Power}(a, n) = a = a^1$.
- **Induction Hypothesis.**
  Assume that the property holds for all positive integers $m < n$. That is, $\text{Power}(a, m) = a^m$.
- **Inductive Step.**
  We must show that the property holds for n. If n is even,

$$\text{Power}(a, n) = \text{Power}(a, n/2) \cdot \text{Power}(a, n/2) \overset{\text{I.H.}}{=} a^{n/2} \cdot a^{n/2} = a^n.$$

  If n is odd,

$$\text{Power}(a, n) = a \cdot \text{Power}(a, (n-1)/2) \cdot \text{Power}(a, (n-1)/2) \overset{\text{I.H.}}{=} a \cdot a^{(n-1)/2} \cdot a^{(n-1)/2} = a^n.$$

  By the principle of mathematical induction, this is true for any integer $n \geq 1$.

Let $T(n)$ be the number of elementary operations that the algorithm Power performs on input a, n. Let's prove by induction that $T(n) \leq 2 \log_2 n$.

- **Base Case.**
  Let $n = 1$. Then $T(n) = 0 \leq 2 \log_2 n$.
- **Induction Hypothesis.**
  Assume that the property holds for all positive integers $m < n$. That is, $T(m) \leq 2 \log_2 m$.
- **Inductive Step.**
  We must show that the property holds for n. If n is even,

$$T(n) \leq T(n/2) + 1 \overset{\text{I.H.}}{\leq} 2 \log_2 n/2 + 1 < 2 \log_2 n.$$

  If n is odd,

$$T(n) \leq T((n-1)/2) + 2 \overset{\text{I.H.}}{\leq} 2 \log_2 (n-1)/2 + 2 < 2 \log_2 n.$$

  By the principle of mathematical induction, this is true for any integer $n \geq 1$.

Exercise 3.3 *Counting Operations in Loops* **(1 Point)**.

For the following code fragments count how many times the function f is called. Report the number of calls as nested sum, and then simplify your expression in O-notation (as tight and simplified as possible) and prove your result.

More precisely, let $T(n)$ be the number of calls of function f in the snippet. Find as simple as possible function $g(n)$ such that $T(n) \leq O(g(n))$ and $g(n) \leq O(T(n))$ and prove these bounds.

For example, in the snippet

Algorithm 4

```
for k = 1, ..., n do
    f()
```

the function f is called $\sum_{k=1}^{n} 1 = n$ times. It is clear that in this example $T(n) \leq O(n)$. Moreover, this bound is tight, that is, $n \leq O(T(n))$.

a) Consider the snippet:

Algorithm 5

```
for j = 1, ..., n do
    for k = 1, ..., n do
        for m = 1, ..., n do
            f()
```

How many times is the function f called?

Solution: f is called

$$T(n) = \sum_{j=1}^{n} \sum_{k=1}^{n} \sum_{m=1}^{n} 1 = n^3$$

times. Hence $T(n) \leq O(n^3)$ and $n^3 \leq O(T(n))$. In other words, $T(n) = \Theta(n^3)$.

b) Consider the snippet:

Algorithm 6

```
for j = 1, ..., n do
    k ← min(j, 100)
    for l = 1, ..., k do
        f()
```

How many times is the function f called?

Solution: f is called

$$T(n) = \sum_{j=1}^{n} \sum_{k=1}^{\min(j,100)} 1 = \sum_{j=1}^{n} \min(j, 100) \leq \sum_{j=1}^{n} 100 = 100n \leq O(n)$$

times. On the other hand,

$$T(n) = \sum_{j=1}^{n} \min(j, 100) \geq \sum_{j=1}^{n} 1 = n,$$

so we also have

$$n \leq O(T(n)).$$

In other words, $T(n) = \Theta(n)$.

c) Consider the snippet:

Algorithm 7

```
for j = 1, ..., n do
    if j^2 ≤ n then
        for k = j, ..., n do
            f()
            f()
            f()
```

How many times is the function f called?

Hint: *You may use the following fact without proof: For every $n \geq 2$, we have $\lfloor \sqrt{n} \rfloor \geq \frac{\sqrt{n}}{2}$ and $n - \lfloor \sqrt{n} \rfloor + 1 \geq \frac{n}{2}$, where $\lfloor x \rfloor$ is the largest integer satisfying $\lfloor x \rfloor \leq x$.*

Solution: Note that the second loop is executed if and only if $j^2 \leq n \Leftrightarrow j \leq \sqrt{n}$. Therefore, f is called

$$T(n) = \sum_{j=1}^{\lfloor \sqrt{n} \rfloor} \sum_{k=j}^{n} 3 = \sum_{j=1}^{\lfloor \sqrt{n} \rfloor} (n - j + 1) \cdot 3 \leq 3 \cdot \lfloor \sqrt{n} \rfloor \cdot n \leq 3 \cdot \sqrt{n} \cdot n \leq O(n\sqrt{n})$$

times. On the other hand, for $n \geq 2$, we have

$$\begin{aligned} T(n) &= \sum_{j=1}^{\lfloor \sqrt{n} \rfloor} (n - j + 1) \cdot 3 \geq \sum_{j=1}^{\lfloor \sqrt{n} \rfloor} (n - \lfloor \sqrt{n} \rfloor + 1) \cdot 3 \\ &= 3 \cdot \lfloor \sqrt{n} \rfloor \cdot (n - \lfloor \sqrt{n} \rfloor + 1) \geq \frac{3}{4} \cdot \sqrt{n} \cdot n, \end{aligned}$$

so we also have

$$n\sqrt{n} \leq O(T(n)).$$

In other words, $T(n) = \Theta(n\sqrt{n})$.

d)* Consider the snippet:

Algorithm 8

```
for j = 1, ..., n do
    k ← 1
    l ← 0
    while k ≤ j do
        for m = 0, ..., l do
            f()
        k ← 13 · k
        l ← l + 1
```

How many times is the function f called?

Solution: f is called

$$T(n) = \sum_{j=1}^{n} \sum_{l=0}^{\lfloor \log_{13} j \rfloor} \sum_{m=0}^{l} 1 = \sum_{j=1}^{n} \sum_{l=0}^{\lfloor \log_{13} j \rfloor} (l+1) \leq \sum_{j=1}^{n} \sum_{l=0}^{\lfloor \log_{13} j \rfloor} (\lfloor \log_{13} j \rfloor + 1)$$
$$\leq \sum_{j=1}^{n} \sum_{l=0}^{\lfloor \log_{13} n \rfloor} (\lfloor \log_{13} n \rfloor + 1) = n \cdot (\lfloor \log_{13} n \rfloor + 1)^2.$$

times. Notice that for $n \geq 13$, we have $\lfloor \log_{13} n \rfloor + 1 \leq \log_{13} n + \log_{13} n = 2 \log_{13} n$, hence,

$$\sum_{j=1}^{n} \sum_{l=0}^{\lfloor \log_{13} j \rfloor} \sum_{m=0}^{l} 1 \leq n \cdot (2 \log_{13} n)^2 = 4n \cdot (\log_{13} n)^2 \leq O(n \cdot (\log n)^2).$$

On the other hand,

$$T(n) = \sum_{j=1}^{n} \sum_{l=0}^{\lfloor \log_{13} j \rfloor} (l+1) = \sum_{j=1}^{n} \sum_{l=1}^{\lfloor \log_{13} j \rfloor + 1} l = \sum_{j=1}^{n} \frac{(\lfloor \log_{13} j \rfloor + 1) \cdot (\lfloor \log_{13} j \rfloor + 2)}{2}$$
$$\geq \sum_{j=\lceil n/2 \rceil}^{n} \frac{(\log_{13} j) \cdot (\log_{13} j)}{2} \geq \frac{n}{2} \cdot \frac{(\log_{13} \frac{n}{2})^2}{2} = \frac{n \cdot (\log_{13} \frac{n}{2})^2}{4}.$$

Now for $n \geq 13^2$, we have $\log_{13} n \geq 2$ and

$$\log_{13} \frac{n}{2} = \log_{13} n - \log_{13} 2 \geq \log_{13} n - 1 \geq \log_{13} n - \frac{1}{2} \log_{13} n = \frac{\log_{13} n}{2},$$

which implies that

$$\sum_{j=1}^{n} \sum_{l=0}^{\lfloor \log_{13} j \rfloor} \sum_{m=0}^{l} 1 \geq \frac{n \cdot (\log_{13} \frac{n}{2})^2}{4} \geq \frac{n \cdot (\frac{1}{2} \log_{13} n)^2}{4} = \frac{n \cdot (\log_{13} n)^2}{16},$$

so we also have

$$n \cdot (\log n)^2 \leq O(T(n)).$$

In other words, $T(n) = \Theta(n \log^2 n)$.

e)* Consider the snippet:

Algorithm 9

```
for j = 1, ..., n do
    for k = 1, ..., j do
        for ℓ = 1, ..., k do
            for m = ℓ, ..., n do
                for o = 1, ..., 100 do
                    f()
```

How many times is the function f called?

Solution: f is called

$$T(n) = \sum_{j=1}^{n}\sum_{k=1}^{j}\sum_{l=1}^{k}\sum_{m=l}^{n}\sum_{o=1}^{100} 1 \leq \sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n}\sum_{m=1}^{n} 100 = 100 \cdot n^4 \leq O(n^4)$$

times. Notice that for $n \geq 4$

$$T(n) = \sum_{j=1}^{n}\sum_{k=1}^{j}\sum_{l=1}^{k}\sum_{m=l}^{n}\sum_{o=1}^{100} 1 \geq \sum_{j=\lceil \frac{2n}{3} \rceil}^{n}\sum_{k=\lceil \frac{n}{3} \rceil}^{\lceil \frac{2n}{3} \rceil}\sum_{l=1}^{\lceil \frac{n}{3} \rceil}\sum_{m=\lceil \frac{n}{3} \rceil}^{n} 100 \geq 100 \cdot \left(\frac{n}{3} - 1\right)^4 \geq \frac{100 \cdot n^4}{12^4},$$

so we also have

$$n^4 \leq O(T(n)).$$

In other words, $T(n) = \Theta(n^4)$.

Exercise 3.4 *Investing in the stock market* **(2 Points)**.

You have 100 CHF and you are considering investing it in the stock market. You heard from your friends that a particular stock is promising but you are not sure. You decided to analyze the performance of this stock during the recent past.

You have a friend that had invested in this stock for n consecutive days in the recent past. You asked your friend about how much money she invested in this stock and how much her total investment worth was progressing every day. You gathered this information in two arrays $A = (A_1, \ldots, A_n)$ and $I = (I_1, \ldots, I_n)$. Here, A_i represents the additional amount of money that your friend invested on the i-th day. More precisely, if your friend bought on the i-th day then A_i would be positive, and if she sold on the i-th day, then A_i would be negative. The value of I_i is the total worth of her investment in the stock on the i-th day. Here is an illustrating example with $n = 4$:

i	1	2	3	4
A_i	150	150	200	-100
I_i	150	350	500	400

In the above example, your friend had invested for 4 days. She invested $A_1 = 150$ CHF on the first day. Since she had not invested anything prior to that day, we must have $I_1 = A_1 = 150$ CHF. On the second day, she invested an additional $A_2 = 150$ CHF and the total worth of her investment was $I_2 = 350$ CHF. This means that right before buying the additional 150 CHF, her total investment in the

stock was worth $350 - 150 = 200$ CHF, which is an indication of an increase in the price of the stock from the first to the second day. On the last day of investment ($i = 4$), she sold 100 CHF worth of her investment, and the remaining total worth of her investment after the selling operation was 400 CHF.

a) Let $1 \leq i \leq n-1$. Suppose that you had invested 1 CHF on the i-th day and sold the entire investment on the $(i+1)$-th day. Show that you would have got $\frac{I_{i+1} - A_{i+1}}{I_i}$ CHF in return.

Solution: On day i, your friend had a total investment of I_i. On the other hand, on day $i+1$, right before the additional investment of A_{i+1}, the total investment was worth $I_{i+1} - A_{i+1}$. Therefore, if you had invested 1 CHF on day i, this investment would be worth $\frac{I_{i+1} - A_{i+1}}{I_i} \times 1\,\text{CHF} = \frac{I_{i+1} - A_{i+1}}{I_i}$ CHF on day $i+1$.

b) Let $1 \leq i \leq j \leq n$. Suppose that you had invested 100 CHF on the i-th day and sold the entire investment on the j-th day. Show that your profit is equal to

$$100 \cdot \prod_{k=i}^{j-1} \frac{I_{k+1} - A_{k+1}}{I_k} - 100.$$

Note that in the above equation, we adopt the convention that if $i = j$, then $\prod_{k=i}^{j-1} \frac{I_{k+1} - A_{k+1}}{I_k} = 1$.

Solution:

From a) it follows that on day j the total investment would be worth

$$\left(\prod_{k=i}^{j-1} \frac{I_{k+1} - A_{k+1}}{I_k} \right) \times 100\,\text{CHF} = 100 \cdot \prod_{k=i}^{j-1} \frac{I_{k+1} - A_{k+1}}{I_k}.$$

Therefore, your profit is

$$100 \cdot \prod_{k=i}^{j-1} \frac{I_{k+1} - A_{k+1}}{I_k} - 100.$$

You are interested in finding the maximum profit that you could have made in a single buy-sell operation by investing 100 CHF. Here, you would buy 100 CHF worth of the stock on some day i where $1 \leq i \leq n$ and then sell the entire investment another day j where $i \leq j \leq n$.

We first assume that all $A_1, \ldots, A_n$ and $I_1, \ldots, I_n$ are positive, and that $I_k > A_k$ for every k.

c) Describe how you can use the maximum subarray-sum algorithm that you learned in class in order to devise an algorithm that computes the maximum profit in $O(n)$ time. You can assume that arithmetic operations (such as addition, subtraction, multiplication and division) as well as logarithms and exponentials are elementary. This means that the computation of $\log$ and $\exp$ take one unit of time each.

Hint: *You can use the fact that the logarithm is a strictly increasing function that turns products into sums.*

Solution: From b) we can see that the maximum profit that you could have made in a single buy-sell operation is:

$$\max_{1\le i\le j\le n}\left\{100\cdot\prod_{k=i}^{j-1}\frac{I_{k+1}-A_{k+1}}{I_k}-100\right\}=100\cdot\left(\max_{1\le i\le j\le n}\prod_{k=i}^{j-1}\frac{I_{k+1}-A_{k+1}}{I_k}\right)-100,$$

which means that it is sufficient to compute $\max_{1\le i\le j\le n}\prod_{k=i}^{j-1}\frac{I_{k+1}-A_{k+1}}{I_k}$.

Let $G=(G_1,\ldots,G_{n-1})$ be a size $n-1$ array such that $G_k=\frac{I_{k+1}-A_{k+1}}{I_k}$ for every $1\le k\le n-1$. Note that since $A_1,\ldots,A_n$ and $I_1,\ldots,I_n$ are positive, and since $I_k>A_k$ for every k, we must have $G_k>0$ for every $1\le k\le n-1$. Now, the problem is reduced to computing

$$\max_{1\le i\le j\le n}\prod_{k=i}^{j-1}G_k, \tag{1}$$

which can be thought of as the maximum subarray product of G. Since the logarithm is a strictly increasing function that turns products into sums, we have:

$$\begin{aligned}\max_{1\le i\le j\le n}\prod_{k=i}^{j-1}G_k&=\exp\left(\log\left(\max_{1\le i\le j\le n}\prod_{k=i}^{j-1}G_k\right)\right)=\exp\left(\max_{1\le i\le j\le n}\log\left(\prod_{k=i}^{j-1}G_k\right)\right)\\&=\exp\left(\max_{1\le i\le j\le n}\sum_{k=i}^{j-1}\log(G_k)\right).\end{aligned}$$

Therefore, we can solve the problem using the following algorithm:

Algorithm 10 Computation of Maximum Profit

```
procedure MaxProfit(A, I)
    for 1 ≤ k ≤ n − 1 do
        G_k ← (I_{k+1} − A_{k+1})/I_k
    MaxProdG ← MaxSubarrayProduct(G)
    MaxProfit ← 100 · MaxProdG − 100
    return MaxProfit

procedure MaxSubarrayProduct(G)
    for 1 ≤ k ≤ n − 1 do
        L_k ← log(G_k)
    MaxSumL ← MaxSubarraySum(L)
    MaxProdG ← exp(MaxSumL)
    return G
```

d) Now assume that the logarithm and exponential operations are expensive so that we would like to avoid using them. Explain how you can modify the maximum subarray sum algorithm in order to solve the problem using only elementary arithmetic operations such as addition, subtraction, multiplication and division. The running time of the algorithm should remain in $O(n)$.

Solution: We only need to modify the MaxSubarrayProduct procedure. We can solve the maximum subarray-product problem using a very similar method to that of the maximum subarray-sum:

Algorithm 11 Computation of maximum subarray-product

procedure MaxSubarrayProduct(G)
  $P \leftarrow G_1$
  MaxProd $\leftarrow \max\{1, P\}$
  for $2 \leq j \leq n-1$ **do**
    $P \leftarrow \max\{G_j, G_j \cdot P\}$
    MaxProd $\leftarrow \max\{\text{MaxProd}, P\}$
  return MaxProd

It is easy to see that the above algorithm runs in $O(n)$ time.

e)* Explain why your algorithm in part d) is correct.

Solution: In order to see why the algorithm of part d) is correct, we first rewrite the maximum subarray-product of Equation (1) as

$$\begin{aligned} \max_{1\leq i\leq j\leq n} \prod_{k=i}^{j-1} G_k &= \max\left\{1, \max_{1\leq i<j\leq n} \prod_{k=i}^{j-1} G_k\right\} \\ &= \max\left\{1, \max_{1\leq i\leq j\leq n-1} \prod_{k=i}^{j} G_k\right\} = \max\left\{1, \max_{1\leq j\leq n-1} P_j^{(\max)}\right\}, \end{aligned} \tag{2}$$

where

$$P_j^{(\max)} = \max_{1\leq i\leq j} \prod_{k=i}^{j} G_k.$$

Note that $P_j^{(\max)}$ represents the maximum product of a nonempty subarray that ends at index j. Notice also that the values of $\left(P_j^{(\max)}\right)_{1\leq j\leq n-1}$ satisfy the following recursive relation: For every $2 \leq j \leq n-1$, we have

$$\begin{aligned} P_j^{(\max)} = \max_{1\leq i\leq j} \prod_{k=i}^{j} G_k &= \max\left\{\prod_{k=j}^{j} G_k, \max_{1\leq i<j} \prod_{k=i}^{j} G_k\right\} = \max\left\{G_j, \max_{1\leq i<j}\left(G_j \cdot \prod_{k=i}^{j-1} G_k\right)\right\} \\ &\overset{(*)}{=} \max\left\{G_j, G_j \cdot \max_{1\leq i\leq j-1} \prod_{k=i}^{j-1} G_k\right\} = \max\left\{G_j, G_j \cdot P_{j-1}^{(\max)}\right\}, \end{aligned} \tag{3}$$

where $(*)$ is true because G_j is positive. The algorithm that we described in part d) is just an implementation of Equations (2) and (3).

f)* Now we consider the general case where $A_1, \ldots, A_n$ and $I_1, \ldots, I_n$ can be either positive or negative[1]. Describe an algorithm that computes the maximum profit that you could have made in a single buy-sell operation by investing 100 CHF. Does your algorithm run in linear time $O(n)$?

[1]If you are wondering how it is possible that the numbers in $I_1, \ldots, I_n$ can be negative, check the oil price crash of 2020.

Solution: Similarly to c) and d), we will first compute $G_k = \frac{I_{k+1} - A_{k+1}}{I_k}$ for every $1 \leq k \leq n-1$, and it is sufficient to compute

$$\max_{1 \leq i \leq j \leq n} \prod_{k=i}^{j-1} G_k = \max\left\{1, \max_{1 \leq j \leq n-1} P_j^{(\max)}\right\}, \qquad \text{where } P_j^{(\max)} = \max_{1 \leq i \leq j} \prod_{k=i}^{j} G_k.$$

Unlike c) and d) where G_k is positive for every k, the values of $(G_k)_{1 \leq k < n}$ can now be either positive or negative. Therefore, the recursive relation of Equation (3) does not necessarily hold because G_j can be negative. Nevertheless, we can write the following:

- If $G_j \geq 0$, we have

$$\begin{aligned} P_j^{(\max)} &= \max\left\{G_j, \max_{1 \leq i < j}\left(G_j \cdot \prod_{k=i}^{j-1} G_k\right)\right\} \\ &= \max\left\{G_j, G_j \cdot \max_{1 \leq i \leq j-1} \prod_{k=i}^{j-1} G_k\right\} = \max\left\{G_j, G_j \cdot P_{j-1}^{(\max)}\right\}, \end{aligned}$$

  exactly as in d).

- If $G_j < 0$, we have

$$\begin{aligned} P_j^{(\max)} &= \max\left\{G_j, \max_{1 \leq i < j}\left(G_j \cdot \prod_{k=i}^{j-1} G_k\right)\right\} \\ &= \max\left\{G_j, G_j \cdot \min_{1 \leq i \leq j-1} \prod_{k=i}^{j-1} G_k\right\} = \max\left\{G_j, G_j \cdot P_{j-1}^{(\min)}\right\}, \end{aligned}$$

  where

$$P_{j-1}^{(\min)} = \min_{1 \leq i \leq j-1} \prod_{k=i}^{j-1} G_k.$$

In fact, we can combine the following two cases in one equation:

$$P_j^{(\max)} = \max\left\{G_j, G_j \cdot P_{j-1}^{(\max)}, G_j \cdot P_{j-1}^{(\min)}\right\},$$

which is true regardless whether $G_j \geq 0$ or $G_j < 0$. Using a very similar argument, we can show that $P_j^{(\min)}$ can also be computed from $P_{j-1}^{(\min)}$ and $P_{j-1}^{(\max)}$: For every $2 \leq j \leq n-1$, we have

$$P_j^{(\min)} = \min_{1 \leq i \leq j} \prod_{k=i}^{j} G_k = \min\left\{G_j, G_j \cdot P_{j-1}^{(\max)}, G_j \cdot P_{j-1}^{(\min)}\right\}.$$

Therefore, we can compute the maximum subarray product using the following algorithm

Algorithm 12 Computation of max subarray-product

procedure MAXSUBARRAYPRODUCT(G)
  $P_1^{(\min)} \leftarrow G_1$
  $P_1^{(\max)} \leftarrow G_1$
  MaxProd $\leftarrow \max\{1, P_{\max}\}$
  for $2 \le j \le n-1$ **do**
    $P_j^{(\max)} \leftarrow \max\left\{G_j, G_j \cdot P_{j-1}^{(\max)}, G_j \cdot P_{j-1}^{(\min)}\right\}$
    $P_j^{(\min)} \leftarrow \min\left\{G_j, G_j \cdot P_{j-1}^{(\max)}, G_j \cdot P_{j-1}^{(\min)}\right\}$
    MaxProd $\leftarrow \max\{\text{MaxProd}, P_j^{\max}\}$
  return MaxProd

It is easy to see that the above algorithm runs in $O(n)$ time.

Exercise 3.5* *Maximum-Submatrix-Sum.*

Provide an $\mathcal{O}(n^3)$ time algorithm which given a matrix $M \in \mathbb{Z}^{n\times n}$ outputs its maximal submatrix sum S. That is, if M has some non-negative entries,

$$S = \max_{\substack{1\le a\le b\le n\\ 1\le c\le d\le n}} \sum_{i=a}^{b}\sum_{j=c}^{d} M_{ij},$$

and if all entries of M are negative, $S = 0$.

Justify your answer, i.e. prove that the asymptotic runtime of your algorithm is $\mathcal{O}(n^3)$.

Hint: *You may want to start by considering the cumulative column sums*

$$C_{ij} = \sum_{k=1}^{i} M_{kj}.$$

How can you compute all C_{ij} efficiently? After you have computed C_{ij}, how you can use this to find S?

Solution: We start with the computation of a matrix of cumulative column sums

$$C_{ij} = \sum_{k=0}^{i} M_{kj}.$$

Then for each pair of rows a and b, $a \le b$, we compute an array of column sums inside the stripe between a and b, that is

$$A_j = \sum_{i=a}^{b} M_{ij} = C_{bj} - C_{a-1j}, \quad 0 \le j < n.$$

(If $a = 0$, $A_j = C_{bj}$).

Then we use a procedure MaxSubarraySum(A) which returns maximal subarray sum of A in time $O(n)$. Maximal subarray sum of A is equal to

$$P(a,b) = \max_{0\le c\le d<n} \sum_{i=a}^{b}\sum_{j=c}^{d} M_{ij}.$$

To find maximal submatrix sum, we maximize $P(a,b)$. For more details, see the pseudocode below.

Algorithm 13 Computation of max submatrix sum

```
procedure MaxSubmatrixSum(M)
    C ← M
    for 1 ≤ i < n do
        for 0 ≤ j < n do
            C_ij ← C_{i-1j} + M_ij
    S ← 0
    for 0 ≤ a < n do
        for a ≤ b < n do
            for 0 ≤ j < n do
                if a = 0 then
                    A_j ← C_bj
                else
                    A_j ← C_bj − C_{a-1j}
            S ← max{S, MaxSubarraySum(A)}
    return S
```

Computing cumulative column sum matrix takes time $O(n^2)$.
There are $O(n^2)$ pairs of rows (a, b) and for each pair we perform $O(n)$ operations: $O(n)$ operations to compute the array of column sums, $O(n)$ operations to find maximal subarray sum and $O(1)$ operations to compare maximal subarray sum with the current maximal value.
Hence the total number of operations is $O(n^3)$.