ALGORITHMS & DATA STRUCTURES

8th November

PLAN FOR TODAY

- Bonus Exercises
- Extra DP Exercises

EXERCISE 6.1

Consider the recurrence

$$F_1 = 1$$

$$F_n = \left(\min_{1 \le i < n} F_i^2 + F_{n-i}^2 \right) \bmod 3n \quad \text{for } n \ge 2,$$

where $a \bmod b$ is the remainder of dividing a by b.

a) Consider the following algorithm that computes F top-down

Algorithm 1 Computing $F(n)$

```
function F(n)
    if n = 1 then
        return 1
    else
        x ← F(1)^2 + F(n − 1)^2
        for i = 2 ... ⌊n/2⌋ do
            x ← min(x, F(i)^2 + F(n − i)^2)
        return x mod 3n
```

Lower bound the running time $T(n)$ of the above algorithm (i.e., give a simple function $g(n)$ such that $T(n) \ge \Omega(g(n))$) and show that it has an exponential running time.

Hint: *Prove by induction that* $T(n) \ge (3/2)^{n-1}$.

To show $T(n) \ge \left(\frac{3}{2}\right)^{n-1}$ (*)

$n=1$ $T(n) \ge \frac{3}{2}^0 = 1$ ✓

$n=2$ $T(n) \ge \frac{3}{2} = \frac{3}{2}^{n-1}$

IH (*) holds $\forall m \le n$ for a n

is $T(n+1) \ge \sum_{i=1}^{n} T(i)$

$\overset{IH}{\ge} \sum_{i=1}^{n} \left(\frac{3}{2}\right)^{i-1}$

$= \frac{\left(\frac{3}{2}\right)^n - 1}{\frac{3}{2} - 1} = 2 \cdot \left(\frac{3}{2}\right)^n - 2$

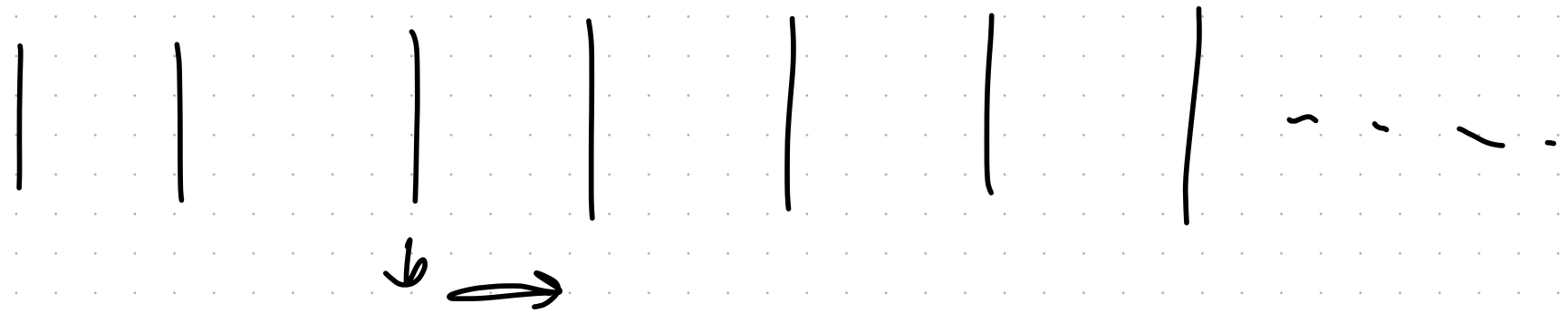
$= \frac{3}{2}^n + \underbrace{\frac{3}{2}^n - 2}_{\ge 0} \ge \left(\frac{3}{2}\right)^n$

$T(1) = a$

$T(n) = \sum_{i=1}^{n-1} T(i) + bn + c$

$T(n) \ge d \cdot 2^{n-1}$

$T(n) \ge \sum_{i=1}^{n-1} 1 \quad i)$



b) Improve the running time of the algorithm in (a) using memoization. Provide pseudo code of the improved algorithm.

Solution:

Algorithm 2 Computing $F(n)$ using memoization

```
memory ← array of size n filled with (−1)s
function F_mem(n)
    if memory[n] ≠ −1 then                    ▷ If F(n) is already computed.
        return memory[n]
    if n = 1 then
        return 1
    else
        x ← F_mem(1)^2 + F_mem(n − 1)^2
        for i = 2 ... ⌊n/2⌋ do
            x ← min(x, F_mem(i)^2 + F_mem(n − i)^2)
        memory[n] ← x mod 3n
        return memory[n]
```

When calling $F_{mem}(n)$, each $F(i)$ for $1 \leq i \leq n$ is computed only once and then stored in `memory`. This substantially enhances the running time of the algorithm.

Remark. The running time of the above memoization algorithm has the same asymptotic growth as the runtime of the algorithm in c) that uses dynamic programming, namely, $\Theta(n^2)$.

Dimensions of the DP table: The DP table is linear, its size is n.

Definition of the DP table: $DP[i]$ contains F_i for $1 \leq i \leq n$.

Calculation of an entry: Initialize $DP[1]$ to 1.

The entries with $n > 1$ are computed by

$$DP[n] = \left(\min_{1 \leq i \leq \lfloor \frac{n}{2} \rfloor} DP[i] \cdot DP[i] + DP[n-i] \cdot DP[n-i] \right) \bmod 3n.$$

Calculation order: We can calculate the entries of DP from smallest to largest.

Reading the solution: All we have to do is read the value at $DP[n]$.

Running time: Each entry $DP[i]$ can be computed in time $\Theta(i)$, so the running time is

$$\sum_{i=1}^{n} \Theta(i) = \Theta(n^2).$$

EXERCISE 6.2

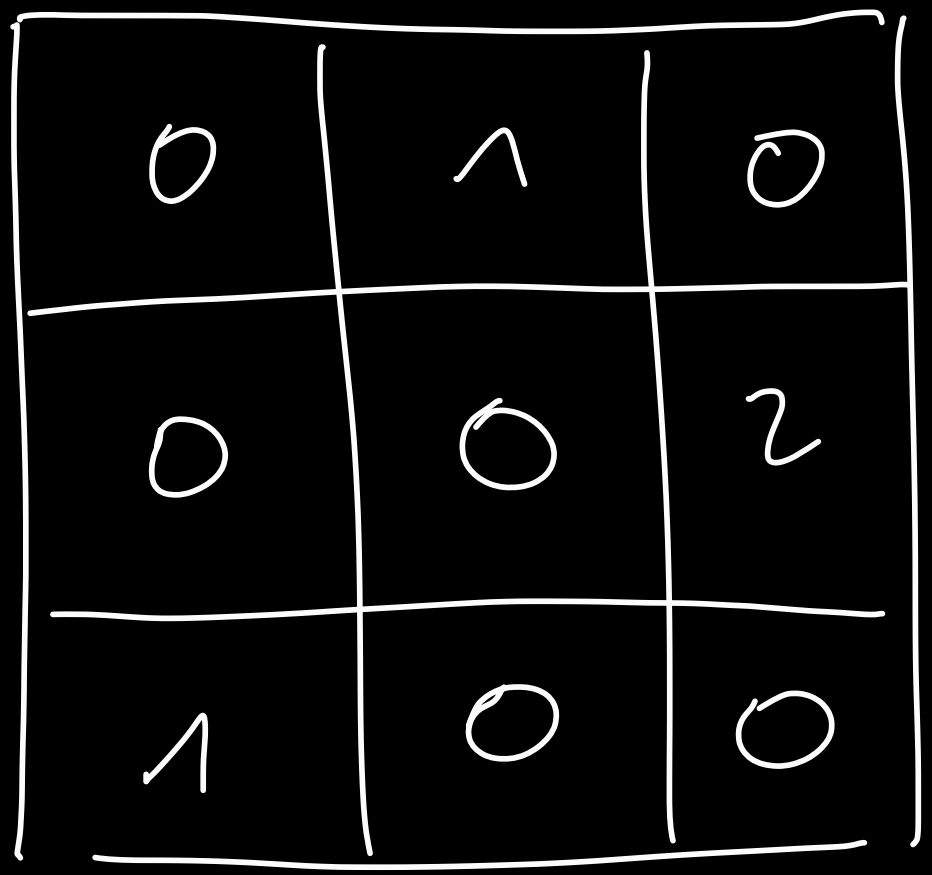
Let $\Sigma = \{a, b, c, \ldots, z\}$ denote the alphabet. Given two strings $\alpha = (\alpha_1, \ldots, \alpha_m) \in \Sigma^m$ and $\beta = (\beta_1, \ldots, \beta_n) \in \Sigma^n$, we are interested in the length of their longest common *substring*, which is the largest integer k such that there are indices i and j with $(\alpha_i, \alpha_{i+1}, \ldots, \alpha_{i+k-1}) = (\beta_j, \beta_{j+1}, \ldots, \beta_{j+k-1})$. Note that this problem is different from the longest common subsequence problem that you saw in the lecture. For example, the longest common substring of $\alpha = (a, a, b, c, b, a)$ and $\beta = (a, b, a, b, c, a)$ is (a, b, c), which is of length 3.

Below is the pseudo-code of an algorithm that computes the length of the longest common substring of two strings $\alpha \in \Sigma^m$ and $\beta \in \Sigma^n$ using $\Theta(mn)$ elementary operations:

Algorithm 3 `LongestCommonSubstring`(α, β)

$L \leftarrow \mathbf{0}^{m \times n}$ an $m \times n$ matrix of zeros.
for $i = 1, \ldots, m$ **do**
  for $j = 1, \ldots, n$ **do**
    if $\alpha_i = \beta_j$ **then**
      if $i = 1$ or $j = 1$ **then**
        $L_{i,j} = 1$
      else
        $L_{i,j} = L_{i-1,j-1} + 1$
    else
      $L_{i,j} = 0$
    // Your invariant from a) must hold here.
return $\max\{L_{i,j} : 1 \leq i \leq m, 1 \leq j \leq n\}$

(a, b, c) and (c, a, b)

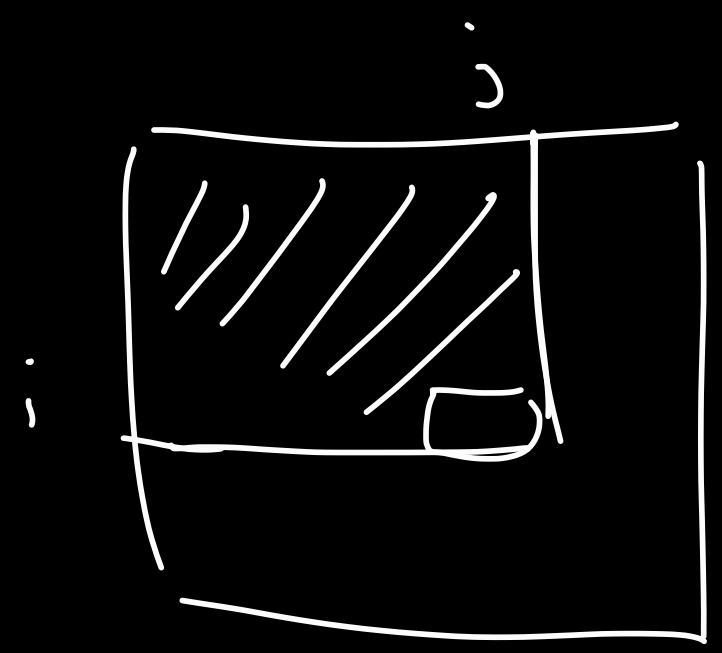


$L_{i,j}$ ist die Länge der longest common substring zwischen $\alpha_1 \ldots \alpha_i$ und $\beta_1 \ldots \beta_j$ die in i,j endet

INVARIANT

$INV(i,j) :=$ $L_{i',j'}$ ist korrekt

$\forall i' \leq i, j' \leq j$



INDUCTION PROOF

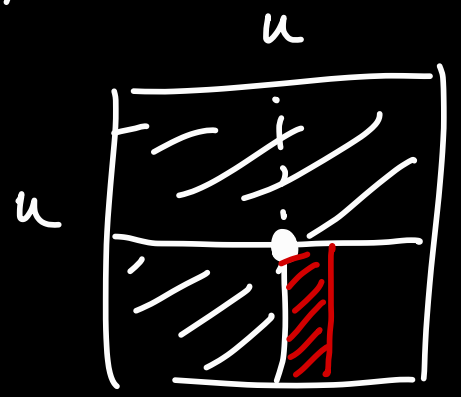
Induktion über $k := \min(i, j)$

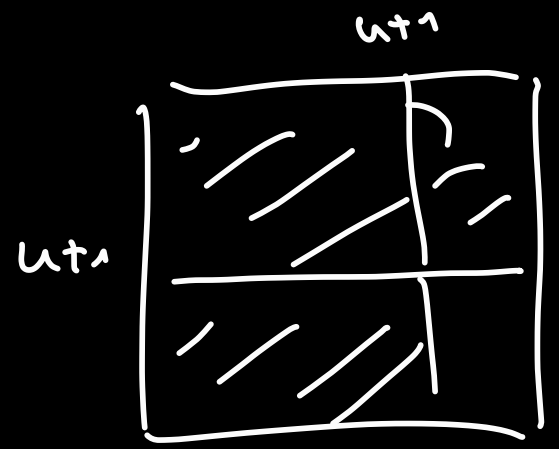
$k = 1$ $i = 1$ $L_{1,j}$ ist Lösung für α_1 und $\beta_1 \dots \beta_j \; \forall j$.

oder $j = 1$ ...

IH $INV(i, j)$ gilt $\forall i, j$ s.d. $\min(i, j) = k$

IS $k \to k + 1$



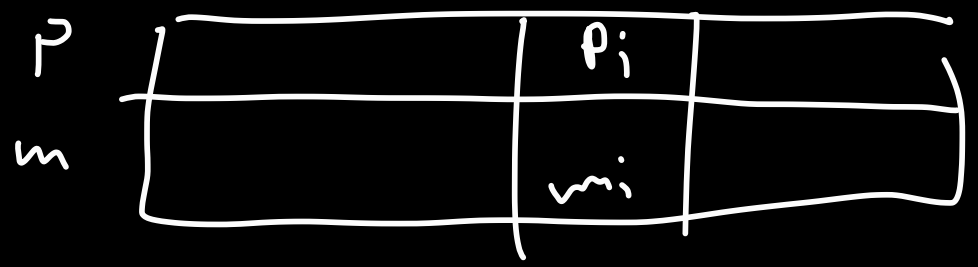


EXERCISE 6.5

Exercise 6.5 *Optimizing Starduck's profit* **(1 point)**.

The coffeeshop chain Starduck's is planning to open several cafés in Bahnhofstrasse Zürich. There are n possible locations $1, \ldots, n$ for their shops on Bahnhofstrasse, ordered by their distance to Zürich main station $m_1 < \ldots < m_n$. Opening a shop at location i would yield Starduck's a profit of $p_i > 0$.

However, they are not allowed to open cafés that are too close to each other, namely any two cafés should have distance at least d from each other, for some given value $d > 0$.



$DP_1 \ldots DP_n$

$DP[1] \leftarrow p_1$

for $i = 2 \ldots n$

first $\leftarrow DP(i-1)$

second $\leftarrow p_i + DP(p(i))$

$DP(i) \leftarrow$ max (first, second)

return $DP(n)$

ALGO(i) binary search

if (M(i) ≠ -1) return M(i)

i = n return p_i

first $\leftarrow p_i$ + ALGO(s(i))

second $\leftarrow$ ALGO(i+1)

M(i) $\leftarrow$ max (first, second)

return max (first, second)

DYNAMIC PROGRAMMING EXERCISES

122. Best Time to Buy and Sell Stock II

Medium 5765 2220 Add to List Share

You are given an integer array `prices` where `prices[i]` is the price of a given stock on the `i^th` day.

On each day, you may decide to buy and/or sell the stock. You can only hold **at most one** share of the stock at any time. However, you can buy it then immediately sell it on the **same day**.

Find and return *the **maximum** profit you can achieve*.

Example 1:

Input: prices = [7,1,5,3,6,4]
Output: 7
Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.
Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.
Total profit is 4 + 3 = 7

```java
class Solution {
    public int algo(int[] hold, int[] not, int[] prices, int flag, int p){
        if(p >= prices.length) return 0;
        if(flag == 1 && hold[p] != -1) return hold[p];
        if(flag == 0 && not[p] != -1) return not[p];
        if(flag == 1){
            int first = algo(hold, not, prices, 1, p + 1);
            int second = prices[p] + algo(hold, not, prices, 0, p + 1);
            hold[p] = Math.max(first, second);
            return Math.max(first, second);
        }
        else{
            int first = algo(hold, not, prices, 0, p + 1);
            int second = -prices[p] + algo(hold, not, prices, 1, p + 1);
            not[p] = Math.max(first, second);
            return Math.max(first, second);
        }
    }
    public int maxProfit(int[] prices) {
        int n = prices.length;
        int[] hold = new int[n];
        int[] not = new int[n];
        for(int i = 0; i < n; i++){
            hold[i] = -1;
            not[i] = -1;
        }
        return algo(hold, not, prices, 0, 0);
    }
}
```

887. Super Egg Drop

Hard 1943 113 Add to List Share

You are given `k` identical eggs and you have access to a building with `n` floors labeled from `1` to `n`.

You know that there exists a floor `f` where `0 <= f <= n` such that any egg dropped at a floor **higher** than `f` will **break**, and any egg dropped **at or below** floor `f` will **not break**.

Each move, you may take an unbroken egg and drop it from any floor `x` (where `1 <= x <= n`). If the egg breaks, you can no longer use it. However, if the egg does not break, you may **reuse** it in future moves.

Return *the **minimum number of moves** that you need to determine **with certainty** what the value of* `f` is.

Example 1:

```
Input: k = 1, n = 2
Output: 2
```

```java
    public int algo(int[][] M, int k, int f){
        if(k == 1) return f;
        if(f == 1) return 1;
        if(k <= 0 || f <= 0) return 0;
        if(M[k][f] != -1) return M[k][f];

        int l = 0;
        int r = f;
        int best = Integer.MAX_VALUE;
        while(l < r){
            int mid = (l + r) / 2;
            int first = algo(M, k - 1, mid - 1);
            int second = algo(M, k, f - mid);
            best = Math.min(best, 1 + Math.max(first, second));
            if(first == second){
              M[k][f] = 1 + first;
              return 1 + first;
            }
            else if(first < second) l++;
            else r--;
        }
        M[k][f] = best;
        return best;
    }

    public int superEggDrop(int k, int n) {
        int[][] M = new int[k + 1][n + 1];
        for(int i = 0; i < k + 1; i++){
            for(int j = 0; j < n + 1; j++) M[i][j] = -1;
        }
        return algo(M, k, n);
    }
```

$$\min_{1 \le X \le N} \left[\max \; T(u-1, X-1), T(u, N-X) \right]$$

$T(\cdot, X)$ is an increasing function in X so...

(1) $\max(T(u-1, X-1), T(u, N-X)) = T(u, N-X)$

$\Rightarrow$ Decreasing X makes the maximum larger

(2) $\max(T(u-1, X-1), T(u, N-X)) = T(u-1, X-1)$

$\Rightarrow$ Increasing X does not help