Mustafa Kemal Kücük mustafak@uni-bremen.com Übungsgruppe 51
Darius Dolha dolha@uni-bremen.de
Vasu Yaduvanshi vasu@uni-bremen.de

1 Übung 1.1

We are given an array representing daily changes in a stock price. The objective is to identify the best days to buy and sell in order to achieve the highest possible profit. Kadane's algorithm is employed to efficiently solve this problem in $O(n)$ time.

1.1 Pseudocode

Algorithm 1 Calculating Maximum Profit Using Kadane's Algorithm

```
1: $maxProfit \leftarrow 0$
2: $currentProfit \leftarrow 0$
3: $buy\_day \leftarrow 0$, $sell\_day \leftarrow 0$
4: $temp\_buy\_day \leftarrow 0$
5:
6: for $k = 0$ to $n - 1$ do
7:     if $currentProfit + A[k] > A[k]$ then
8:         $currentProfit \leftarrow currentProfit + A[k]$
9:     else
10:        $currentProfit \leftarrow A[k]$
11:        $temp\_buy\_day \leftarrow k$
12:    end if
13:    if $currentProfit > maxProfit$ then
14:        $maxProfit \leftarrow currentProfit$
15:        $buy\_day \leftarrow temp\_buy\_day$
16:        $sell\_day \leftarrow k$
17:    end if
18: end for
19: return $(buy\_day, sell\_day)$
```

1.2 Explanation

Kadane's algorithm determines the maximum profit by continuously checking the current profit and restarting whenever it becomes negative. The rationale behind this approach is that a negative profit from a subarray would diminish the potential total profit if included in the calculations.
To analyze the efficiency mathematically, we define the current profit P_k at day k as follows:

$$P_k = \sum_{i=0}^{k} A[i]$$

Where $A[i]$ represents the daily price change. If the accumulated profit P_k is negative, we reset the current profit:

$$P_k = \begin{cases} P_{k-1} + A[k] & \text{if } P_{k-1} + A[k] > 0 \\ A[k] & \text{otherwise} \end{cases}$$

Now, the algorithm checks at each step if the current profit exceeds the maximum profit found so far:

$$maxProfit = \max(maxProfit, P_k)$$

As the algorithm processes the array in a single loop, we can see that the overall time complexity is $O(n)$ due to the single pass through the array, making it significantly more efficient than a brute-force method, which would involve evaluating all pairs of buy and sell days and has a time complexity of $O(n^3)$.
This algorithm reliably identifies the days that yield the maximum profit, utilizing the properties of continuous subarrays and the mathematical nature of summation.

2 Übung 1.2

2.1 Description

Let A be an n-dimensional sorted array $A = [a0, a1, a2, ..., an]$ and k a natural number. We are going to choose two pointers i pointing at the beginning of the array and j pointing at the end of the array. In other words i and j are two indexes; $i = 0$ and $j = n - 1$ where $n = length(A)$. Let s be the sum of the two elements represented by the indexes. $s = A[i] + A[j]$.

The algorithm is going to use the following observation:

- if $s = k$ we return the pair of indexes (i, j)
- if $s < k$ we increment the value of i, in order to increase the sum s (we know that the array is sorted, so the next element would be bigger)
- if $s > k$ we decrement the value of j, in order to decrease the sum

If $i \geq j$, no such index pair exists, and -1 shall be returned.

2.2 Pseudocode

Algorithm 2

Data: sorted Array with n-elements $A = [a0, a1, a2, ..., an]$ and a natural number k
Result: pair of indexes (i,j) or -1

```
$i \leftarrow 0$
$j \leftarrow length(A) - 1$
while $i < j$ do
    $s \leftarrow A[i] + A[j]$
    if $s = k$ then
        return $(i, j)$
    else
        if $s < k$ then
            $i \leftarrow i + 1$
        else
            $j \leftarrow j - 1$
        end if
    end if
end while
return $-1$
```

2.3 Worst Case

In the worst case, the two-pointer algorithm checks all possible pairs without finding a match, resulting in n operations. Starting with pointers i (beginning) and j (end), the algorithm iterates, adjusting i or j based on whether their sum is less than or greater than k. If no valid pair is found, i and j eventually meet in the middle, ending the search. Each step moves one pointer closer to the other, ensuring each element is visited only once, making the algorithm efficient with linear time, even in the worst case. The running time for this scenario is going to be $an + b$ since it is required to to take at least n steps in order to realize that there is no solution.

2.4 Best Case

In the best scenario, the solution will be exactly $(0, n-1)$ leading us to linear running time and therefore the time complexity will be constant.

3 Übung 1.3

3.1 Best Case - Minimale Anzahl an Vergleichen

um das beste Case Szenario zu erreichen, muss das Array bei Eingabe schon sortiert sein. Das heißt, $a[j+1] > a[j]$ gilt sich bei jedem Durchlauf von der Schleife und für jedes Element. Das heißt, das Array soll so aussehen: $A = [a_0, a_1, ..., a_{n-1}]$ mit $a_0 \geq a_1 \geq ... \geq a_{n-1}$. Die äußere Schleife bei der Sortierung nach dem Einfügen prinzip beginnt bei $i = 1$ und geht bis i gleich $n-1$ ist und weil nichts ersetzt werden musste, wird die äußere Schleife insgesamt nur $n-1$ Mal durchlaufen.

3.2 Worst Case - Maximale Anzahl an Vergleichen

Das Array ist komplett rückwärts sortiert, also sieht so aus $A = [a_0, a_1, ..., a_{n-1}]$ mit $a_0 \geq a_1 \geq ... \geq a_{n-1}$. Für jedes Element $A[j+1]$ muss Insertion Sort A[j+1] mit $A[j]$ vergleichen, um die richtige Position zu finden. Für das erste Element gibt es keine Vergleiche. Für das zweite Element gibt es 1 Vergleich, für das dritte 2 Vergleiche und so weiter, bis zum letzten Element (das $n-1$ Vergleiche erfordert). Dies bildet eine Reihe: $1 + 2 + 3 + ... + (n-1)$, das summiert sich als $n(n-1)/2$.