1 Algorithm Design

Describe an $\Theta(n \log n)$ time algorithm that, given an array $A$ of $n$ integers and another integer $x$, determines whether or not there are two elements in $A$ whose sum equals $x$ (Hint: Sorting!).

Solution. I will describe two different solutions. Solution 1 is the solution that I had in mind. During the exercise class, one of you came up with a simpler and much more elegant solution! This solution is presented as Solution 2.

Solution 1. We first sort the array $A$ in time $\Theta(n \log n)$. Assume from now on that $A$ is sorted. Next, we check whether $A$ contains two elements of value $x/2$ in time $\Theta(\log n)$ (using binary search). If there are such elements then we are done. Else, we know that if there is a solution then it consists of two elements $x_1, x_2$ with $x_1 < x/2$ and $x_2 > x/2$. Let $i$ be the position in array $A$ such that $A[i] < x/2$ and $A[i + 1] \geq x/2$. Let $j = i + 1$. Consider now the following loop:

- If $A[i] + A[j] > x$ then subtract 1 from $i$.
- If $A[i] + A[j] = x$ then we found a solution and we stop.

We stop this procedure once $i = -1$ or $j = n$ as we then have not found a solution. The runtime of this procedure is clearly $\Theta(n)$, since $i$ and $j$ together “walk” at most a distance of $n$.

To see why this works, let $k_1, k_2$ with $k_1 < k_2$ be the indices of a solution, i.e., $A[k_1] + A[k_2] = x$. Observe that, initially, we have

$$k_1 \leq i < j \leq k_2.$$  \hspace{1cm} (1)

If the algorithm “misses” the solution $k_1, k_2$, then there is moment when we updated either $i$ or $j$ and then Inequality 1 is no longer true, i.e., we either updated $i$ to become value $k_1 - 1$ or we updated $j$ to become value $k_2 + 1$.

Suppose first that variable $i$ was updated at this moment. This implies that the algorithm went from the configuration $(i = k_1, j)$ to the configuration $(i = k_1 - 1, j)$. By construction of the algorithm, this only happens if $A[k_1] + A[j] > x$. This however is a contradiction, since $A[k_1] + A[j] \leq A[k_1] + A[k_2] = x$ (since $j \leq k_2$).

Suppose next that variable $j$ was updated at this moment. This implies that the algorithm went from the configuration $(i, j = k_2)$ to the configuration $(i, j = k_2 + 1)$. By construction of the algorithm, this only happens if $A[i] + A[k_2] < x$. This however is a contradiction, since $A[i] + A[k_2] > A[k_1] + A[k_2] = x$ (since $i \geq k_1$).

The algorithm therefore cannot miss the configuration $(k_1, k_2)$. 


Solution 2. Again, we first sort the array $A$ in $\Theta(n \log n)$ time. Assume from now on that $A$ is sorted. Next, we walk through the array from left to right with a for loop (using variable $i = 0 \ldots n - 1$). In iteration $i$, we use a binary search to check whether the array $A$ contains an element with value $x - A[i]$. A binary search takes time $O(\log n)$. Since we do a binary search in each iteration, and there are $n$ iterations at most, the runtime is $O(n \log n)$. This is a very nice and elegant solution. Thanks to the student who came up with it.

✓

2 Bubblesort

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order:

Algorithm 1 Bubblesort

Require: Array $A$ of $n$ integers
1: for $i = 0$ to $n - 2$ do
2: for $j = n - 1$ downto $i + 1$ do
5: end if
6: end for
7: end for

1. What is the worst-case runtime of Bubblesort?

Solution. Observe that the operation in Line 4, i.e., exchanging two elements in the array, takes time $O(1)$. The runtime is therefore bounded by the number of times Line 4 is executed. The outer loop goes from $i = 0$ to $n - 2$, and the inner loop goes from $j = n - 1$ downto $i + 1$. We therefore compute:

$$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} O(1) = O(1) \cdot \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = O(1) \cdot \sum_{i=0}^{n-2} ((n - 1) - (i + 1) + 1)$$

$$= O(1) \cdot \sum_{i=0}^{n-2} (n - i - 1) = O(1) \cdot \left( (n - 1)^2 - \sum_{i=0}^{n-2} i \right)$$

$$= O(1) \left( (n - 1)^2 - \frac{(n - 2)(n - 1)}{2} \right) \leq O(1) \left( (n - 1)^2 / 2 \right)$$

$$= O(n^2).$$

✓

2. Consider the loop in lines 2 – 6. Prove that the following invariant holds at the beginning of the loop:

$$A[j] \leq A[k], \text{ for every } k \geq j$$

Give a suitable termination property of the loop.
Consider now the loop in lines 1−k. We are guaranteed that Termination: \( k \geq k \) holds). Concerning \( k \)

**Solution.**

**Initialization:** We need to show that the property is true prior to the first iteration of the loop. Let \( j = n - 1 \). Then the property translates to \( A[n - 1] \leq A[k] \) for every \( k \geq n - 1 \). This is trivially true since the only value for \( k \) such that \( k \geq n - 1 \) that also lies within the boundaries of the array is \( k = n - 1 \). It is of course true that \( A[n - 1] \leq A[n - 1] \). The property thus holds.

**Maintenance:** Suppose that the property is true before an iteration \( j \) of the loop, i.e., \( A[j] \leq A[k] \) holds for every \( k \geq j \). We will show that the property also holds before the next iteration. Observe that before the next iteration, the value of \( j \) is decreased. We thus need to show that after the current iteration, \( A[j - 1] \leq A[k] \) holds for every \( k \geq j - 1 \).

Considering the algorithm, there are two cases: Either the if-condition evaluates to true, or it evaluates to false.

**Case 1:** \( A[j] \geq A[j - 1] \). (i.e., the if evaluates to false)

In this case nothing happens to the array elements. We need to show that \( A[j - 1] \leq A[k] \), for every \( k \geq j - 1 \). We already know that \( A[j] \leq A[k] \) for every \( k \geq j \). Since \( A[j - 1] \leq A[j] \), the loop invariant is thus also true.

**Case 2:** \( A[j] < A[j - 1] \). (i.e., the if evaluates to true)

In this case, \( A[j] \) is exchanged with \( A[j - 1] \). We need to show that after the exchange \( A[j - 1] \leq A[k] \) for every \( k \geq j - 1 \). Consider thus the state of the array after the exchange. Concerning \( k = j - 1 \), this is trivially true (i.e, \( A[j - 1] \leq A[j - 1] \) clearly holds). Concerning \( k = j \), this is also true due to the if-statement evaluating to true and the fact that we exchanged the two elements. Concerning all other values of \( k \), i.e., \( k \geq j + 1 \), this follows from the loop invariant being true at the beginning of the iteration.

**Termination:** We are guaranteed that \( A[i] \leq A[k] \), for every \( k \geq i \).

3. Consider now the loop in lines 1−7. Prove that the following invariant holds at the beginning of the loop:

The subarray \( A[0, i] \) is sorted.

Give a suitable termination property that shows that \( A \) is sorted upon termination.

**Solution.** We will prove the even stronger statement: “At the beginning of iteration \( i \), the subarray \( A[0, i] \) is sorted and \( A[0, i - 1] \) consists of the \( i - 1 \) smallest elements of \( A \).

**Initialization:** We need to show that the property is true prior to the first iteration of the loop. At the beginning of the first iteration we have \( i = 0 \). Then the property translates to “the subarray \( A[0\ldots 0] \) is sorted and \( A[0, -1] \) consists of the \( i - 1 \) smallest elements of \( A \)”.

This is trivially true, since \( A[0\ldots 0] = A[0] \) consists of a single element, and \( A[0\ldots -1] \) is empty.

**Maintenance:** Suppose that the property is true before an iteration \( i \) of the loop, i.e., \( A[0\ldots, i] \) is sorted and \( A[0\ldots i - 1] \) are the \( i - 1 \) smallest elements of \( A \). We will show that the property also holds before the next iteration. By the termination property stated in the last exercise, we have that \( A[i] \leq A[k] \), for every \( k \geq i \), or, in other words, \( A[i] \) is the smallest element in \( A[i, n - 1] \). By the loop invariant, \( A[0,\ldots, i - 1] \) are the \( i - 1 \) smallest elements in increasing order. Hence, the subarray \( A[0,\ldots, i] \) contains the \( i \) smallest elements in \( A \) in increasing order. This implies further that the subarray \( A[0, i+1] \) is sorted (note that no matter which element is at position \( i + 1 \), the array is sorted).

**Termination:** We are guaranteed that \( A \) is sorted.
3 Proofs by Induction (optional and difficult!)

Let \( n \) be a positive number that is divisible by 23, i.e., \( n = k \cdot 23 \), for some integer \( k \geq 1 \). Let \( x = \lfloor n/10 \rfloor \) and let \( y = n \% 10 \) (the rest of an integer division). Prove by induction on \( k \) that 23 divides \( x + 7y \).

**Example:** Consider \( k = 4 \). Then \( n = 92 \), \( x = 9 \) and \( y = 2 \). Observe that the quantity \( x + 7y = 9 + 7 \cdot 2 = 23 \) is divisible by 23.

**Solution.** We prove the statement by induction over \( k \). To this end, let \( x_i \) be the value of \( x \) when \( n = i \cdot 23 \), and similarly, let \( y_i \) be the value of \( y \) when \( n = i \cdot 23 \).

**Base case:** \( (k = 1) \)

In this case, \( n = 1 \cdot 23 \), \( x_1 = 2 \) and \( y_1 = 3 \). The quantity \( x_1 + 7y_1 = 23 \), which is divisible by 23.

**Induction Hypothesis:** Suppose that \( x_i + 7y_i \) is divisible by 23.

**Induction Step:** We will show that \( x_{i+1} + 7y_{i+1} \) is also divisible by 23. We conduct a case distinction:

- Suppose that \( y_i \leq 6 \). Then \( y_{i+1} = y_i + 3 \) and \( x_{i+1} = x_i + 2 \). We obtain:
  \[
  x_{i+1} + 7y_{i+1} = x_i + 2 + 7(y_i + 3) = x_i + 7y_i + 2 + 21 = x_i + 7y_i + 23 .
  \]
  Since \( x_i + 7y_i \) is divisible by 23 and 23 is of course divisible by 23, we have \( x_{i+1} + 7y_{i+1} \) is divisible by 23.

- Suppose that \( y_{i+1} > 6 \). Then, \( y_{i+1} = y_i - 7 \) and \( x_{i+1} = x_i + 3 \). We obtain:
  \[
  x_{i+1} + 7y_{i+1} = x_i + 3 + 7(y_i - 7) = x_i + 7y_i + 3 - 49 = x_i + 7y_i - 46 .
  \]
  Again, since \( x_i + 7y_i \) is divisible by 23 and 46 is divisible by 23, we have \( x_{i+1} + 7y_{i+1} \) is divisible by 23.

\( \checkmark \)