Name (print):

Signature:

Instructions: This is a closed-book exam. Communicate your ideas *clearly* and *succinctly*. Write your solutions directly *and only* on this booklet. You may use other sheets of paper as scratch, but *do not submit anything other than this booklet*, as nothing else will be considered for grading. You may use either a pen or a pencil.

On problem	you got	out of
1		30
2		20
3		30
4		20
5		20
Total		120

►**Exercise 1.** Write an algorithm MOUNTAIN-SORT(A) that, given an array A of n numbers, sorts A in-place such that the left half of A is increasing and the right half is decreasing. More specifically, the values from $A[1]$ to $A[\lfloor n/2 \rfloor]$ are increasing and the values from $A[\lfloor n/2 \rfloor]$ to $A[n]$ are decreasing. Notice that the left and right subsequences share the element in the middle position $A[\lfloor n/2 \rfloor]$. Notice also that the resulting order is not unique. For example, for $A = [8, 2, 5, -12, 2, 11, -15, -8, -1, 12]$, MOUNTAIN-SORT($A$) might result in $A = [-12, -8, -1, 1, 12, 11, 8, 5, 2, -15]$. *(30)*

You must detail every algorithm you use in your solution. So, if you want to, say, sort the input array or any part of it, you must explicitly write the sorting algorithm. Also, analyze the complexity of your solution.

▷*Solution 1*

In essence, the resulting order must be such that the value in the middle position $A[\lfloor n/2 \rfloor]$ is maximal, and that the subsequence to the left of $A[\lfloor n/2 \rfloor]$ is increasing while the subsequence on the right is decreasing. Other than that, the two sides don't need to be balanced or otherwise correlated in any way. Also, there are no complexity constraints. So, we can develop a very simple solution based on INSERTION-SORT. The idea here is to first sort the whole sequence, with INSERTION-SORT, and then to invert the right half of A.

```
MOUNTAIN-SORT(A)
 1  n = A.length
 2  for i = 2 to n
 3      j = i
 4      while j > 1 and A[j − 1] > A[j]
 5          swap A[j − 1] ↔ A[j]
 6          j = j − 1
 7  i = ⌊n/2⌋
 8  j = n
 9  while i < j
10      swap A[i] ↔ A[j]
11      i = i + 1
12      j = j − 1
```

The complexity is $\Theta(n^2)$, which is the complexity of sorting the array.

Another approach could be to first put the maximal value in the middle, and then sort the left half in increasing order and the right half in decreasing order.

▶**Exercise 2.** Consider the following algorithm that takes an array A of n numbers.

```
ALGO-X(A)
n = A.length
x = 0
for i = 1 to n
    j = 1
    while j ≤ n and (i == j or A[i] ≠ A[j])
        j = j + 1
    if j > n
        x = x + 1
return x
```

Question 1: Explain what ALGO-X does. Do not simply paraphrase the code. Instead, explain the high-level semantics of the algorithm independent of the code. *(5)*

▷ *Solution 2.1*

ALGO-X returns the number of unique values in A

Question 2: Analyze the complexity of ALGO-X. Is there a difference between the best and worst-case complexity? If so, describe a best and a worst-case input of size n, as well as the behavior of the algorithm in each case. *(5)*

▷ *Solution 2.2*

The worst-case complexity is $\Theta(n^2)$. This is a case in which the algorithm must check that $A[i]$ is not equal to any other value $A[j]$ (with $i \neq j$). In the best case, the algorithm goes through each value $A[i]$ but then does not run the inner loop more than a constant amount of times. This is the case, for example, in which A contains n copies of the same value. In this case, the inner loop terminates immediately for every $A[i]$, and therefore the complexity is $\Theta(n)$.

Question 3: Write an algorithm called Better-Algo-X that does exactly the same thing as Algo-X with a strictly better time complexity. *(10)*

▷ *Solution 2.3*

We first sort A, and then go through the sorted data B, counting how many elements $B[i]$ are different from their adjacent elements $B[i-1]$ and $B[i+1]$.

```
Better-Algo-X($A$)
1  $B$ = copy of $A$ sorted in ascending order
2  $x = 0$
3  **for** $i = 1$ **to** $B.length$
4        **if** ($i == 1$ **or** $B[i] \neq B[i-1]$) **and** ($i == n$ **or** $B[i] \neq B[i+1]$)
5              $x = x + 1$
6  **return** $x$
```

►**Exercise 3.** An accounting system models a revenue transaction t as an object with two attributes, $t.date$ and $t.amount$, representing the date and amount of the transaction, respectively. Dates are represented as numbers of days since a reference initial date, such that $t_2.date - t_1.date$ is the number of days between transactions t_1 and t_2. Amounts are positive numbers. With that, consider the following ALGO-Y(T) that takes an array T of transactions:

```
ALGO-Y($T$)
$x = 0$
for $i = 1$ to $T.length$
    $l = T[i].amount$
    $r = T[i].amount$
    for $j = 1$ to $T.length$
        if $i \neq j$
            if $T[j].date \leq T[i].date$ and $T[i].date - T[j].date \leq 10$
                $l = l + T[j].amount$
            if $T[j].date \geq T[i].date$ and $T[j].date - T[i].date \leq 10$
                $r = r + T[j].amount$
    if $x < r$
        $x = r$
    if $x < l$
        $x = l$
return $x$
```

Question 1: Explain what ALGO-Y does. Do not simply paraphrase the code. Instead, explain the high-level semantics of the algorithm independent of the code. *(5)*

▷*Solution 3.1*

ALGO-Y returns the highest total sales in a period of ten days.

Question 2: Analyze the complexity of ALGO-Y. Is there a difference between the best and worst-case complexity? If so, describe a best and a worst-case input of size n, as well as the behavior of the algorithm in each case. *(5)*

▷*Solution 3.2*

The complexity is $\Theta(n^2)$. The nested loops perform complete iterations over T without any shortcut. So, the complexity is the same also in the best case.

Question 3: Write an algorithm called BETTER-ALGO-Y that does exactly the same thing as ALGO-Y, but with a strictly better complexity in the worst case. Analyze the complexity of BETTER-ALGO-Y. *(20)*

▷ *Solution 3.3*

We first sort the set of transactions by date, and then we simply scan the set of transactions maintaining the total (net) gain for a window of transactions that are all within 10 days of each other.

```
BETTER-ALGO-Y($T$)
 1  $S$ = copy of $T$ sorted by $date$
 2  $i = 1$
 3  $j = 1$
 4  $v = 0$
 5  $m = 0$
 6  while $j \le S.length$
 7      if $S[j].date - S[i].date \le 10$
 8          $v = v + S[j].amount$
 9          $j = j + 1$
10          if $m < v$
11              $m = v$
12      else $v = v - S[i].amount$
13          $i = i + 1$
14  return $m$
```

After sorting T, at a cost of $\Theta(n \log n)$, BETTER-ALGO-Y performs a linear scan of the sorted array. The overall complexity is therefore $\Theta(n \log n)$.

►**Exercise 4.** Consider the following array

$$H = [3, 5, 8, 6, 10, 9, 5, 6, 7, 20, 11, 17, 6, 9, 10]$$

Question 1: Does H contain a valid *min heap?* If so, extract the minimum value, rearranging H again as a minheap, and then write the resulting content of the array. If not, turn H into a min heap by applying a minimal number of swap operations, and write the resulting content of the array. Justify your answer. *(5)*

▷*Solution 4.1*

H is not a valid min heap because the value $H[3] = 8$ should be less than or equal to both the values $H[6] = 9$ and $H[7] = 5$. So, $H[7] < H[3]$ violates the min-heap property. Similarly, $H[13] = 6 < H[6] = 9$ also violate the same property. A simple fix is to swap those two pairs of values: $H[7] \leftrightarrow H[3]$ and $H[13] \leftrightarrow H[6]$. The resulting content of the array is:

$$H = [3, 5, 5, 6, 10, 6, 8, 6, 7, 20, 11, 17, 9, 9, 10]$$

Question 2: Write an algorithm MIN-HEAP-ADD(H, x) that adds a new value a min heap H. *(10)*

▷ *Solution 4.2*

```
MIN-HEAP-ADD($H$)
1  append $x$ to $H$
2  $i = H.length$
3  while $i > 1$ and $H[i] < H[\lfloor i/2 \rfloor]$
4      swap $H[i] \leftrightarrow H[\lfloor i/2 \rfloor]$
5      $i = \lfloor i/2 \rfloor$
```

Question 3: Execute MIN-HEAP-ADD$(H, 4)$ using the algorithm you wrote as a solution to Question 2. In this case, the input H contains the min-heap resulting from your solution to Question 1. Illustrate the execution of MIN-HEAP-ADD$(H, 4)$ by writing the full content of the array H at the beginning of each iteration of the algorithm, as well as at the end of the algorithm. *(5)*

▷ *Solution 4.3*

$$H = [3, 5, 5, 6, 10, 6, 8, 6, 7, 20, 11, 17, 9, 9, 10, 4]$$

$$H = [3, 5, 5, 6, 10, 6, 8, 6, 7, 20, 11, 17, 9, 9, 4, 10]$$

$$H = [3, 5, 5, 6, 10, 6, 4, 6, 7, 20, 11, 17, 9, 9, 8, 10]$$

$$H = [3, 5, 4, 6, 10, 6, 5, 6, 7, 20, 11, 17, 9, 9, 8, 10]$$

►**Exercise 5.** Write an algorithm SQUARE-ROOT(n) that, given a non-negative integer n, returns $\lfloor \sqrt{n} \rfloor$. SQUARE-ROOT(n) may only use the basic arithmetic operations of addition, subtraction, multiplication and division (integer), and must run in $O(\log n)$ time. *(20)*

▷*Solution 5*

We can compute the square root using a straightforward binary search.

```
SQUARE-ROOT($n$)
$h = n + 1$
$l = 0$
while $l + 1 < h$
    $m = \lfloor (l + h)/2 \rfloor$
    if $m \cdot m > n$
        $h = m$
    elseif $m \cdot m < n$
        $l = m$
    else return $m$
return $m$
```