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		20
2		20
3		30
4		20
5		30
Total		120

►**Exercise 1.** Write an algorithm MAX-HEAP-INSERT(H, x) that inserts a value x in a max-heap H. Also, write the content of H (as an array) after the insertion of each of the following values, in the given order, starting from an empty max-heap: *(20)*

$$3, 7, 3, 2, 9, 5, 9, 8, 5, 2, 9, 4, 7, 3, 9$$

.

►**Exercise 2.** The following algorithm ALGO-X(A) takes an array A of n numbers.

```
ALGO-X(A)
for i = 1 to A.length
    s = 0
    for j = 1 to A.length
        if i ≠ j
            s = s + A[j]
    if A[i] == s
        return TRUE
return FALSE
```

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)*

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)*

Question 3: Write an algorithm called BETTER-ALGO-X that does exactly the same thing as ALGO-X in $O(n)$ time. *(10)*

►**Exercise 3.** The following algorithm ALGO-Y(A, r, c) operates on an $r \times c$ matrix of $n = rc$ elements, where r and c are the numbers of rows and columns of the matrix, and the matrix is stored row-wise in the given array A. This means that the first c elements of A are the c elements of the first row of the matrix, the following c elements of A are the c elements of the second row of the matrix, and so on.

```
ALGO-Y(A, r, c)
for i = 1 to rc
    for j = i + 1 to rc
        if A[i] == A[j]
            a = ⌊(i − 1)/c⌋ // integer division
            b = ⌊(j − 1)/c⌋ // integer division
            if a == b or a == b − 1
                if i − ac == j − bc or i − ac == j − bc + 1 or i − ac == j − bc − 1
                    return TRUE
return FALSE
```

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)*

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)*

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 wost case. Analyze the complexity of BETTER-ALGO-Y. *(20)*

►**Exercise 4.** Write an algorithm FIND-AVG-POINT(A) that takes an array of $n \geq 2$ numbers, and returns a position i where the values in A cross the average between the first and last element. More specifically, letting $m = (A[n] + A[1])/2$, FIND-AVG-POINT(A) must return an index i such that $A[i] \leq m \leq A[i+1]$ or $A[i] \geq m \geq A[i+1]$. FIND-AVG-POINT(A) must have a worst-case time complexity of $o(n)$, meaning strictly better than linear time. Also, analyze the complexity of FIND-AVG-POINT. (*Hint:* interpret the values in A as a series of points with coordinates $(i, A[i])$ connected by line segments. FIND-AVG-POINT(A) must return a position i where the segment crosses or touches the horizontal line at level m.) *(20)*

►**Exercise 5.** We say that an array A is in "e-top" order when $A[i] \leq A[j]$ for all i, j such that i is odd and j is even. Write an algorithm SORT-E-TOP(A) that sorts an array A in e-top order with an average-case time complexity of $O(n)$. You may want to use standard, well-known algorithms. However, you must explicitly write their pseudo-code. *(30)*

Solutions

▷ *Solution 1*

MAX-HEAP-INSERT(H, x)

$H.heap\text{-}size = H.heap\text{-}size + 1$
$i = H.heap\text{-}size$
$H[i] = x$
while $i > 1$ **and** $H[i] > H[\lfloor i/2 \rfloor]$
    swap $H[i] \leftrightarrow H[\lfloor i/2 \rfloor]$
    $i = \lfloor i/2 \rfloor$

[3]

[7, 3]

[7, 3, 3]

[7, 3, 3, 2]

[9, 7, 3, 2, 3]

[9, 7, 5, 2, 3, 3]

[9, 7, 9, 2, 3, 3, 5]

[9, 8, 9, 7, 3, 3, 5, 2]

[9, 8, 9, 7, 3, 3, 5, 2, 5]

[9, 8, 9, 7, 3, 3, 5, 2, 5, 2]

[9, 9, 9, 7, 8, 3, 5, 2, 5, 2, 3]

[9, 9, 9, 7, 8, 4, 5, 2, 5, 2, 3, 3]

[9, 9, 9, 7, 8, 7, 5, 2, 5, 2, 3, 3, 4]

[9, 9, 9, 7, 8, 7, 5, 2, 5, 2, 3, 3, 4, 3]

[9, 9, 9, 7, 8, 7, 9, 2, 5, 2, 3, 3, 4, 3, 5]

▷ *Solution 2.1*

ALGO-X checks whether A contains an element $A[i]$ that is equal to the sum of all other elements in A.

▷ *Solution 2.2*

The worst-case complexity is $\Theta(n^2)$. In such a case, the algorithm goes through each one of the n elements, computes the sum of all the other $n-1$ elements in n steps, and then returns FALSE. The best-case complexity is instead $\Theta(n)$, which happens when the first element equals the sum of all other elements, which the algorithm computes in $\Theta(n)$ steps.

▷ *Solution 2.3*

If there is an element x such that the sum of every other element is x, then the total sum of all elements must be $2x$. So, we can simply compute the total sum s, in $\Theta(n)$ time, and then look for $s/2$ in A, also in $\Theta(n)$ time.

```
BETTER-ALGO-X(A)
s = 0
for i = 1 to A.length
    s = s + A[i]
for i = 1 to A.length
    if A[i] == s/2
        return TRUE
return FALSE
```

▷ *Solution 3.1*

ALGO-Y checks whether any two adjacent positions in the matrix contain equal elements. Adjacent means different positions whose column and row indexes differ by at most one.

▷ *Solution 3.2*

The complexity is $\Theta(n^2)$. The worst case is when there are no two equal elements, so the two loops go through all the $\binom{n}{2}$ pairs of elements, only to return FALSE at the end. Conversely, the best-case complexity is $O(1)$, which happens when the first two elements of the first row of the matrix are equal.

▷ *Solution 3.3*

For each element i, j in the matrix, which we denote here as $M_{i,j}$, there are at most 6 neighbors, namely $M_{i,j\pm1}$, $M_{i\pm1,j}$, and $M_{i\pm1,j\pm1}$. We can therefore scan all those pairs of adjacent positions in $\Theta(n)$ time. (Recall that the size of the matrix is $rc = n$.)

```
BETTER-ALGO-Y(A, r, c)
for i = 1 to r - 1
    for j = 1 to c
        if A[ic + j + 1] == A[(i + 1)c + j + 1] // M_{i,j} == M_{i+1,j}
            return TRUE
for i = 1 to r
    for j = 1 to c - 1
        if A[ic + j + 1] == A[ic + j + 2] // M_{i,j} == M_{i,j+1}
            return TRUE
for i = 1 to r - 1
    for j = 1 to c - 1
        if A[ic + j + 1] == A[(i + 1)c + j + 2] // M_{i,j} == M_{i+1,j+1}
            return TRUE
        if A[(i + 1)c + j + 1] == A[ic + j + 2] // M_{i+1,j} == M_{i,j+1}
            return TRUE
return FALSE
```

▷ *Solution 4*

The average $m = (A[n] + A[1])/2$ is such that either $m = A[1] = A[n]$, in which case the algorithm can immediately return $i = 1$ or $i = n$, or $A[1] < m < A[n]$ or $A[1] > m > A[n]$. In both these latter cases, we can proceed with a binary search. We just have to make sure that we run the binary search consistently with the specific relative order between $A[1]$ and $A[n]$.

```
Find-Avg-Point(A)
r = A.length
if A[1] == A[r]
    return 1
m = (A[r] + A[1])/2
ℓ = 1
while ℓ + 1 < r
    c = ⌊(ℓ + r + 1)/2⌋
    if A[c] > m
        if A[ℓ] > m
            ℓ = c
        else r = c
    elseif A[c] < m
        if A[ℓ] < m
            ℓ = c
        else r = c
    else return c
return ℓ
```

▷ *Solution 5*

The e-top order requires that all the elements in the even positions are less than or equal to all the elements in the odd positions. Since there are about $n/2$ even positions and $n/2$ odd positions in the array—more specifically, there are exactly $n/2$ even and $n/2$ odd positions if n is itself even, or $(n-1)/2$ even and $(n+1)/2$ odd positions if n is odd—the e-top order is equivalent to partitioning the array by the *median* value $m \in A$.

```
Sort-E-Top(A)
n = A.length
if n is even
    k = n/2
else k = (n + 1)/2
m = Selection(A, k)
i = 1
j = 2
while i ≤ n or j ≤ n
    if A[i] ≤ m
        i = i + 2
    elseif A[j] > m
    j = j + 2
else swap A[i] ↔ A[j]
    i = i + 2
    j = j + 2
```

```
Selection(A, k)
n = A.length
L = empty array
M = empty array
R = empty array
v = pick an element at random from A
for i = 1 to n
    if A[i] < v
        append A[i] to L
    elseif A[i] > v
        append A[i] to R
    else append A[i] to M
if k ≤ L.length
    return Selection(L, k)
elseif k ≤ L.length + M.length
    return v
else return Selection(L, k − L.length − M.length)
```