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		30
3		30
4		30
Total		120

►**Exercise 1.** An array A of n numbers is sorted. Some elements are then set to 0. Write an algorithm RE-SORT(A) that takes such an array A and sorts it in-place and in time $O(n)$. *(30)*

▷ *Solution 1*

The input consists of a sorted sub-sequence of negative numbers (possibly empty) followed by a sorted sub-sequence of positive numbers (possibly empty), possibly with zeroes within and between the first and second sequence. So, all we have to do is pack the first subsequence of negative numbers towards the left side of A, then pack the subsequence of positive numbers towards the right side of A, and then set to 0 all the positions that are left in the middle.

```
RE-SORT(A)
 1  $n = A.length$
 2  $i = 1$
 3  $i_{base} = i$
 4  while $i \le n$ and $A[i] \le 0$
 5      if $A[i] < 0$
 6          $A[i_{base}] = A[i]$
 7          $i_{base} = i_{base} + 1$
 8      $i = i + 1$
 9  $j = n$
10  $j_{base} = j$
11  while $j \ge i$
12      if $A[j] > 0$
13          $A[j_{base}] = A[j]$
14          $j_{base} = j_{base} - 1$
15      $j = j + 1$
16  while $i_{base} \le j_{base}$
17      $A[i_{base}] = 0$
18      $i_{base} = i_{base} + 1$
```

►**Exercise 2.** Consider the following game: you start with two decks of n playing cards each (shuffled). At each round, you remove one or two cards as follows. If the two cards at the top of the two decks have the same suit or the same numeric value, you may remove both of them at no cost. If the two cards have different suits and numbers, or if you do not choose to remove both of them, you must choose to remove one of the two cards at a cost corresponding to its numeric value. If one of the decks is empty, you have no choice: you must remove the card on the remaining deck at the cost of its numeric value. The game ends when both decks are empty.

Now consider the following decision problem: given the two initial shuffled decks A and B and a maximal cost c, decide whether it is possible to play a game with a total cost less than c. A and B are arrays of cards; the functions $\mathit{suit}(x)$ and $\mathit{value}(x)$ return, in $O(1)$ time, the suit and numeric value of a card x, respectively. For example, $\mathit{suit}(A[i])$ returns the suit of the the i-th card on the A deck.

Question 1: Is this problem in NP? Show a proof of your answer. *(10)*

Hint: a decision problem is in NP when an example that shows that the answer is "yes" can be verified in polynomial time. Here, a sequence of game choices can be such an example.

▷*Solution 2.1*

The problem is in NP, since it is easy to play the game following a given set of choices S that serve as a "witness" for a TRUE answer.

```
VERIFY($A, B, c, S$)
n = A.length // assume A.length == B.length
i = 1
j = 1
k = 1
t = 0 // total cost of the game
while i ≤ n or j ≤ n
    if i ≤ n and j ≤ n
        if S[k] == DISCARD-BOTH
            if suit(A[i]) ≠ suit(B[j]) and value(A[i]) ≠ value(B[j])
                return FALSE
            i = i + 1
            j = j + 1
        elseif S[k] == DISCARD-A // discard from A
            t = t + value(A[i])
            i = i + 1
        else t = t + value(B[j]) // discard from B
            j = j + 1
        k = k + 1
    elseif i < n
        t = t + value(A[i])
        i = i + 1
    else t = t + value(B[j])
        j = j + 1
if t < c
    return TRUE
else return FALSE
```

Question 2: Is this problem in P? Show a proof of your answer. *(20)*

Hint: consider a dynamic-programming approach to find the minimal cost of a game.

▷ *Solution 2.2*

The problem is in P. We can decide by checking that the minimal cost of a game is less than the given cost limit c. We find the minimal cost of a game with a dynamic programming algorithm. The dynamic-programming solution is simply a coding of all the possible choices in the game.

```
SOLVE($A, B, c$)
if $DP(A, 1, B, 1) < c$
    return TRUE
else return FALSE
```

```
DP($A, i, B, j$)
if $i > A.length$ and $j > B.length$
    return 0
if $i == A.length$
    $t = 0$
    while $j \leq B.length$
        $t = t + valueB[j]$
        $j = j + 1$
    return $t$
if $j == B.length$
    $t = 0$
    while $i \leq A.length$
        $t = t + valueA[i]$
        $i = i + 1$
    return $t$
$t = \min\{\mathrm{DP}(A, i+1, B, j) + value(A[i]), \mathrm{DP}(A, i, B, j+1) + value(B[j])\}$
if $suit(A[i]) == suit(B[j])$ or $value(A[i]) == value(B[j])$
    $t = \min\{t, \mathrm{DP}(A, i+1, B, j+1)\}$
return $t$
```

Now, this solution is not really polynomial, since the combination of all possible choices given by the multiple recursion of the DP function leads to an exponential complexity. However, the algorithm can be readily turned into a polynomial one by using memoization, which is left as an exercise for the reader...

►**Exercise 3.** Consider the following algorithm that takes two strings A and B. You may assume that characters have numeric codes between 0 and m for some relatively small constant m. For example, ASCII characters are encoded by numbers between 0 and 127.

```
ALGO-X($A, B$)
V = [] // empty array
for $i$ = 1 to $B.length$
    append 0 to $V$
for $i$ = 1 to $A.length$
    $x$ = FALSE
    $j$ = 1
    while $j \leq B.length$ and $x$ == FALSE
        if $A[i] == B[j]$ and $V[j] == 0$
            $x$ = TRUE
            $V[j] = 1$
        else $j = j + 1$
for $j$ = 1 to $B.length$
    if $V[j] == 0$
        return FALSE
return TRUE
```

Question 1: Explain what ALGO-X does. Do not simply paraphrase the code. Instead, explain the high-level semantics, independent of the code. Also, analyze the complexity of ALGO-X. *(10)*

▷*Solution 3.1*

ALGO-X returns TRUE if and only if the characters of B are a subset of those of A, considering also their multiplicity. So, for example, B =“aac” is a subset of A =“aabbcc”. The worst-case is when B does not contain any of the characters of A. For example, A =“aaa. . . ” and B =“bbb. . . ”. In fact, the outer loop (over A) is fixed, and the inner loop (over B) can only terminate when $A[i] == B[j]$ for some i and j. So, in the worst case, the complexity is $\Theta(n^2)$.

Question 2: Write an algorithm called BETTER-ALGO-X that does exactly the same thing as ALGO-X, but with a strictly better complexity. Analyze the complexity of BETTER-ALGO-X. Notice that if ALGO-X modifies the content of the input strings, then BETTER-ALGO-X must do the same. Otherwise, BETTER-ALGO-X must not modify A and B. *(20)*

Bonus: extra points if your BETTER-ALGO-X runs in linear time. *(5)*

▷ *Solution 3.2*

We must compare the sequences as multi-sets. We can do that by first sorting the two sequences, so that we can then compare them element-by-element as if we were performing a *merge* of the two (sorted) sequences.

```
BETTER-ALGO-X($A, B$)
    $C$ = sorted copy of $A$
    $D$ = sorted copy of $B$
    $j = 1$
    for $i = 1$ to $A.length$
        if $j > B.length$
            return TRUE
        elseif $C[i] == D[j]$
            $j = j + 1$
        elseif $C[i] > D[j]$
            return FALSE
    if $j \leq B.length$
        return FALSE
    else return TRUE
```

The main body of this algorithm runs in $O(n)$ time, so the overall complexity is $\Theta(n \log n)$ for sorting A and B.

Since the values of the characters in A and B are numbers from a fixed and small range, we can also develop an $O(n)$ solution:

```
BETTER-ALGO-X-LINEAR($A, B$)
    $n = A.length$
    $C$ = []
    $D$ = []
    for $i = 1$ to $m$ // $m$ is the size of the alphabet
        append 0 to $C$
        append 0 to $D$
    for $i = 1$ to $n$
        $C[A[i]] = C[A[i]] + 1$
        $D[B[i]] = D[B[i]] + 1$
    for $i = 1$ to $m$
        if $C[i] < D[j]$
            return FALSE
    return TRUE
```

►**Exercise 4.** Write an algorithm MINIMAL-ADDITIONAL-EDGES(G) that takes an undirected graph G and returns the minimal number of edges that must be added to G to make it connected. *(30)*

▷*Solution 4*

Notice that G can be seen as the union of c connected components, with $1 \le c \le n$, where each connected component is a maximal set of vertexes that form a connected subgraph of G. We can then connect those components by adding $c - 1$ edges to form a spanning tree of the connected components.

In practice, we can start from any vertex v_0, then visit all the vertexes reachable from v_0, directly or indirectly using BFS, then find the first vertex v_1 that was not already visited, and therefore implicitly count an additional edge (v_0, v_1), and again visit all the vertexes reachable from v_1 (BFS); then again find the next non-visited vertex v_2, implicitly count an additional edge (v_1, v_2), and so on until we visited every vertex in G.

```
MINIMAL-ADDITIONAL-EDGES($G = (V, E)$)
 1  $Visited = \emptyset$ // vertexes that were already visited
 2  $c = 0$ // number of connected components
 3  while $Visited \neq V$
 4      $u$ = any vertex that is not in $Visited$ // must exist, since $Visited \neq V$
 5      $c = c + 1$ // we now run a BFS starting from $u$
 6      $Q$ = empty queue
 7      enqueue $u$ in $Q$
 8      $Visited = Visited \cup \{u\}$
 9      while $Q$ is not empty
10          $v$ = dequeue vertex from $Q$
11          for $w \in Adj(v)$
12              if $w \notin Visited$
13                  enqueue $w$ in $Q$
14                  $Visited = Visited \cup \{w\}$
15  return $c - 1$
```