CSE202 2019-2020 – FINAL EXAM

The 6 exercises are independent and can be treated in arbitrary order. Within an exercise, the answer to a question can be used in the next ones even if a proof has not been found.

The number of points indicated in front of each question is an indication of the relative difficulties of the questions. It is not necessary to solve all the questions to obtain the maximal possible grade.

Your descriptions of algorithms can be written in Python or in pseudo-code.

Exercise 1. Given two arrays A and B of integers, an inversion between A and B is a pair of indices i and j such that $A[i] > B[j]$.

(1) (2pts) Describe an algorithm that counts the number of inversions between two arrays A and B of at most n elements, each of them being sorted. Your algorithm should use $O(n)$ comparisons.

An inversion in an array A of integers is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$. For instance, there are 4 inversions in the array $[6, 4, 9, 2]$, found at the indices $(1, 2), (1, 4), (2, 4), (3, 4)$.

(2) (4pts) Describe and analyze an algorithm to count the number of inversions in an n-element array in $O(n \log n)$ comparisons. A solution in $O(n \log^2 n)$ comparisons gives you only half of the points.

Solution. 1. It is important to note that if all the elements of A are larger than all the elements of B, then the number of inversions is n^2 and thus an algorithm that would only increase by one its counter is bound to have a quadratic complexity. One way of achieving linear complexity is by a single loop as follows:

```
# A and B are assumed to be sorted
def InversionBetween(A,B):
    res = 0 ; i = len(A)-1; j = len(B)-1
    while (i>=0 and j>=0):
        if B[j]>=A[i]: j -= 1
        else: res += j+1; i -= 1
    return res
```

2. The complexity $O(n \log n)$ and the previous question both suggest a divide-and-conquer algorithm. If the array is split into two halves, then the number of inversions is given as the sum of the numbers of inversions in both halves, plus the number of inversions between the sub-arrays. This last number can be obtained by first sorting both sub-arrays and then using the previous question. However, if one sorts at each recursive step, the complexity increases to $O(n \log^2 n)$. So instead, one recovers the sorted arrays from the previous steps and merges them recursively as in MergeSort. The resulting algorithm is

```
def NumberofInversionsandSort(A):
    n = len(A)
```

```
if (n<=1): return 0,A
res1,A1 = NumberofInversionsandSort(A[0:n//2])
res2,A2 = NumberofInversionsandSort(A[n//2:n])
return res1+res2+InversionBetween(A1,A2),Merge(A1,A2)
```

It returns both the number of inversions and the sorted array, using an instruction `Merge`*, which is the same as in MergeSort. The number of comparisons performed by the algorithm then obeys* $C(1) = 0$ *and*

$$C(N) \leq 2C(\lceil N/2 \rceil) + 2N,$$

since at most N *comparisons are performed in* `Merge` *and in* `InversionBetween`*. The master theorem of divide-and-conquer then gives that* $C(N) = O(N \log N)$ *as desired.* □

Exercise 2. (2pts) In a binary search tree built from n random keys, what is the probability that the successful search cost for one of these keys is 2?

Solution. The search cost is 2 when the key is found at depth 2 in the tree, ie, as a child of the root. For all choices of roots except the largest and the smallest key, this happens for 2 of the keys, while in the last two cases, only one key is at depth 2. The final result is therefore

$$\underbrace{\frac{n-2}{n}}_{\mathbb{P}(root \in \{2,\dots,n-1\})} \frac{2}{n} + \underbrace{\frac{2}{n}}_{\mathbb{P}(root \in \{1,n\})} \frac{1}{n} = \frac{2}{n}\left(1 - \frac{1}{n}\right).$$ □

Exercise 3. If $P \in \mathbb{K}[X]$ is a polynomial of degree n with coefficients in a field $\mathbb{K} \subset \mathbb{C}$, and $\alpha_1, \dots, \alpha_n$ are its complex roots, the *Newton sums* s_k for $k \in \mathbb{N}$ are the sums of the kth powers of the α_i: $s_k = \alpha_1^k + \cdots + \alpha_n^k \in \mathbb{K}$. The aim of this exercise is to prove that $(s_0, \dots, s_N)$ can be computed in $O(\mathrm{Mul}(N))$ arithmetic operations in $\mathbb{K}$, without knowing the α_i's. Here $\mathrm{Mul}(N)$ is a bound on the number of coefficients operations needed to multiply two polynomials of degree at most N.

(1) (1pt) By considering the Taylor expansion of $1/(1-aX)$, give an algorithm for the case when the degree n of P is 1;
(2) (3pts) Let $Q(X) = X^n P(1/X)$ with roots $1/\alpha_i$, $i = 1, \dots, n$ and consider the partial fraction expansion of Q'/Q to design an efficient algorithm for arbitrary n. (Assume for simplicity that the roots α_i are distinct.) [It may help to consider first the case of a quadratic polynomial $(X-\alpha_1)(X-\alpha_2)$.]

Solution. 1. The polynomial is given as $p_0 + p_1 X$ *with* $p_1 \neq 0$ *since the degree is 1. Its root* $\alpha = -p_0/p_1$ *is obtained in* $O(1)$ *operations. The expansion is*

$$\frac{1}{1-\alpha X} = 1 + \alpha X + \alpha^2 X^2 + \cdots$$

and has for coefficients the Newton sums. Here a simple loop would give the first N *coefficients in* $O(N)$ *operations. Another way is to compute this expansino up to precision* N *by Newton iteration for division of power series, giving a complexity* $O(\mathrm{Mul}(N))$*.*

2. The polynomial Q *is obtained from the polynomial* P *by reordering the coefficients, thus at no arithmetic cost. The partial fraction decomposition of* Q'/Q

is

$$\frac{Q'}{Q} = -\frac{\alpha_1}{1-\alpha_1 X} - \cdots - \frac{\alpha_n}{1-\alpha_n X}.$$

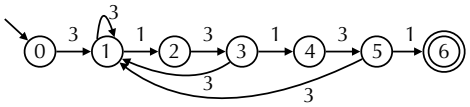
Its Taylor expansion is therefore

$$\frac{Q'}{Q} = -s_1 - s_2 X - \cdots - s_N X^{N-1} + O(X^N).$$

This can be computed by computing first $1/Q$ by Newton iteration to precision N in $O(\mathrm{Mul}(N))$ operations and by a final multiplication by Q' (itself obtained in $O(N)$ operations from Q) in $O(\mathrm{Mul}(N))$ operations again. □

Exercise 4. (1) (1pt) Give the KMP automaton for the pattern 313131, assuming a 4-character alphabet 0, 1, 2, and 3.
(2) (1pt) What is the sequence of transitions performed by the KMP algorithm to determine whether the text 1232033313230313131 contains this pattern?
(3) (1pt) How many comparisons of characters are needed for this?

Solution. 1. The automaton is



All the transitions that are not drawn point to the state 0.

2. On the given text, starting from the state 0 the transitions performed by the algorithm lead successively to the states 0010011123010123456.

3. The number of comparisons of characters performed by the KMP algorithm does not depend on the pattern: it does exactly one comparison per character in the text, here 19. □

Exercise 5. (2pts) Consider the following sorting algorithm: it takes as input a set S of n integers; initializes an empty binary search tree T; inserts the elements of S into T in random order; outputs the elements found by a depth-first search traversal of T, visiting recursively the left subtree, then the node, then the right subtree. Show that the expected number of operations performed by this algorithm is $O(n \log n)$.

Solution. The binary search tree is built using exactly the same comparisons as randomized QuickSort, thus in $O(n \log n)$ operations. The depth-first search traversal visits each edge twice (once in each direction) and there is exactly one edge per node except the root, so the traversal has cost $O(n)$, negligible with respect to the construction of the BST. □

Exercise 6. The partition problem is the following: given a list L of positive integers, partition the list into two lists A and B such that $\max(\sum_{a\in A} a, \sum_{b\in B} b)$ is minimal.

(1) (1pt) Show that the partition problem is NP-hard.
(2) (1pt) Consider the following algorithm:

```
def partition(X):
    suma = sumb = 0
    for i in sorted(X,reverse=True):
        if suma<sumb: suma += i
```

```
            else: sumb += i
    return max(suma,sumb)
```

Show that its complexity is polynomial in the size of the input.

(3) (2pts) Show that if the loop is run with `reverse=False` (ie, in increasing order), then the approximation ratio is at least $3/2$.

(4) (4pts) Show that the algorithm with `reverse=True` has approximation ratio at most $4/3$. [Hint: a possible proof is by contradiction.]

Solution. 1. We show that the problem is NP-hard by reducing the problem 2-partition to it (not the other way round!): we have seen that 2-partition is NP-complete so that such a reduction proves NP-hardness of the partition problem. The reduction is straightforward: if the list L input to the 2-partition problem is input to the partition problem and the result is an integer N, then compare N to half the sum of the elements of L (which can be performed in polynomial time) and return true if and only if they are equal. This occurs exactly when L can be 2-partitioned.

2. The complexity is clearly polynomial: sorting is in $O(n \log n)$ comparisons, each of which costs at most n bit operations, where n is the total bit size of L, next each comparison or addition is again polynomial in the size of the result, which is bounded by $\sum_{\ell \in \mathbb{L}}$, itself of at most linear size in n.

3. If all elements are equal to 1 and the last one is n, the optimum is $\max(n-1, n)$, which is found by the first algorithm. However, running in increasing order yields $\max(n/2, n + n/2) = 3n/2$ (assuming n is even).

4. Let L be an instance where the algorithm returns a solution larger than $4/3$ times the optimal value Ω and let S be the sum of the elements of L. Necessarily $\Omega \geq S/2$. Without loss of generality, assume that the result is the sum of the elements of A. The sum of the elements of A is larger than $4\Omega/3 \geq 2S/3$, so that the sum of the elements of B is smaller than $S/3$ and so was that of A before the last element was added to it. This implies that the last element is larger than $S/3$, which is incompatible with the previous sums being smaller than $S/3$ and the list being sorted by decreasing order. □