1) Let v be an n-size vector of integers. Show that after a pre-processing in O(n) time, we can answer to the following type of queries q(i,j) in O(1)-time: is the subvector v[i…j] sorted?

<u>Solution</u> 1: For every index k, let inv[k] denote the number of indices s < k such that v[s] > v[s+1]. It can be constructed in O(n) by scanning the vector v once. Now, to answer to q(i,j), we output YES if either i=j or inv[j] – inv[i] = 0.

<u>Solution 2</u>: We partition the vector in maximal sorted sub-vectors. It can be done in O(n) by scanning once vector v. Now, to answer to q(i,j), we just need to check in O(1) whether i,j are in the same range of this partition.

<u>Comment</u>: we could also check whether v[i…j] is either sorted by non-increasing values or sorted by non-decreasing values.

2) We are given as input an n-size integer vector v. We are allowed to pre-process this vector so as to answer, as fast as possible, to the following type of queries q(i,j): ``what is the number of elements v[p], p between i and j, such that v[p] is an even number?''.

   a. Show that, if v is fixed, then we can achieve O(n) pre-processing time and O(1) query time.

   We use a classic `partial sum' trick. Let u be the n-size boolean vector so that u[i] = 1 if and only if v[i] is even. In an auxiliary vector w, we store in w[i] the number of even elements v[q], for q between 0 and i. Since w[0] = u[0] and w[i] = w[i-1]+u[i], then we can compute the vector w in O(n) time by dynamic programming. Now, for answering a query q(i,j), it suffices to output w[j]-w[i]+u[i].

   b. We now allow the vector v to be dynamically modified: at any time step, one can change the value stored in an arbitrary position i of the vector. Show that we can achieve O(n) pre-processing time and O($\sqrt{n}$) query time.

   We use Mo's trick. Specifically, we partition the vector in $\sqrt{n}$ contiguous blocks of size $\sqrt{n}$ each. In a separate $\sqrt{n}$-size vector s, we store the number of even elements in each block. Note that, upon modifying the value of v[i], if i lies in the jth block then we only need to modify s[j] (we decrement this value by 1 if the former value of v[i] was an even number, then we increment this value by 1 if the new value of v[i] is even), that only takes O(1) time. Now, in order to answer to a query q(i,j), we do as follows: let B_a, B_{a+1}, ..., B_{a+t} the blocks fully between i and j (we can find these blocks simply by looking at the blocks to which i and j belong, respectively). We start summing all values s[a+p], for p between 0 and t. Since t is at most $\sqrt{n}$, the latter can be done in O($\sqrt{n}$) time. Then, we increment the result by 1 for each even element v[q], q ≥ i, of B_{a-1} and we end up incrementing the result also by 1 for each even element v[q], j ≥ q, of B_{a+t+1}. Since each block has at most $\sqrt{n}$ elements, it also takes O($\sqrt{n}$) time.

3) Presentation of Merge Sort (not seen yet in class).

4) We are given as input an n-size integer vector v.

   a. Show that we can compute in O(n*log(n)) the number of <u>invertions</u>, that is, the number of pairs (r,s) such that r < s and v[r] > v[s].

   It suffices to adapt Merge sort. We cut the vector in two halves and we apply recursively our algorithms on both halves so that: a) we counted the number of invertions in both halves; and b) we sorted both halves. Then, during a classical merge of both halves in

one sorted vector (interclasare), we can count the number of invertions with one element in each half. For that, consider the ith element of the left half. Let it be put in position i+j in the final sorted vector. Then, j elements on the right half were smaller than it. Therefore, we count j invertions with the right half for this element.

b. Show that after pre-processing the vector in O(n*log(n)), we can answer to the following type of queries q(i) in O(1): ``what is the number of invertions between i and n-1, that is, the number of pairs (r,s) such that $i \leq r < s$ and $v[r] > v[s]$?''

We apply the algorithm for counting the number of invertions in a vector (see the previous question). It runs in O(n*log(n)) time. Doing so, we actually computed something stronger: namely, we computed for every index i the number inv[i] of indices j > i such that v[i] > v[j]. Now, we apply a classic ``partial sum trick'', namely: let $u[i] = \sum inv[k]$, k=0...i. In order to answer to a query q(i), it suffices to output u[n-1] – u[i] + inv[i]. The pre-processing time is in O(n*log(n)), while the query time is in O(1).

c. We are allowed to pre-process this vector so as to answer, as fast as possible, to the following type of queries q(i,j): ``what is the number of invertions between i and j, that is, the number of pairs (r,s) such that $i \leq r < s \leq j$ and $v[r] > v[s]$?''

To solve our range query problem, we can now combine the above algorithm with Mo's trick. Specifically, we partition the vector in $\sqrt{n}$ contiguous blocks of size $\sqrt{n}$ each.
i) We create a first $\sqrt{n}$ x $\sqrt{n}$ matrix M_0, so that: if $i \leq j$, then $M_0[i,j]$ is the number of pairs (r,s) such that r < s is in block i, s is in block j, and v[r] > v[s]. If i=j, then we can compute $M_0[i,i]$ in O($\sqrt{n}$ *log(n)) time by using our above algorithm. If i < j, then we count in O($\sqrt{n}$ *log(n)) time the number inv(i,j) of invertions in the vector obtained from the concatenation of the ith and jth block; then, $M_0[i,j] = inv(i,j) - M_0[i,i] - M_0[j,j]$ (we could also easily adapt our previous algorithm to this case). The total runtime is in O(n*$\sqrt{n}$ *log(n)).
ii) Then, we use a classic partial sum trick. Specifically, for $i \leq j$, let $M_1[i,j]$ be the sum of all values $M_0[i,j']$, $i \leq j' \leq j$. We can compute the matrix M_1 in O(n) time by dynamic programming.
iii) Now, we create another n x $\sqrt{n}$ matrix M_2 so that $M_2[i,j]$ is the number of elements smaller than v[i] in the jth block. Being given a sorted copy of B_j we can compute $M_2[i,j]$ in O(log(n)), simply by doing a binary search for v[i]. Therefore, the total runtime is in O(n*$\sqrt{n}$ *log(n)).
iv) Finally, we create the n x $\sqrt{n}$ matrix M_3 so that $M_3[i,j]$ is the sum of all values $M_2[i,j']$, $0 \leq j' \leq j$. We can compute the matrix M_3 in O(n$\sqrt{n}$) time by dynamic programming.

In order to answer to a query q(i,j), we do as follows: let B_a, B_{a+1}, ..., B_{a+t} the blocks fully between i and j (we can find these blocks simply by looking at the blocks to which i and j belong, respectively). We start summing all values $M_1[a+p,a+t]$, for p between 0 and t. Since t is at most $\sqrt{n}$, the latter can be done in O($\sqrt{n}$) time. For each element v[q], $q \geq i$, of B_{a-1} we increment the output by $M_3[q,a+t] – M_3[q,a-1]$. Similarly, for each element v[q], $j \geq q$, of B_{a+t+1}, we increment the output by $M_3[q,a+t] – M_3[q,a-1]$. We are left computing the number of invertions in $[i,j] \cap B_{a-1} + B_{a+t+1}$ that can be done in O($\sqrt{n}$*log(n)) time.