1) (*Seen in class*) Consider a k-bit binary counter initialized to 0. Prove that incrementing this counter can be done in amortized O(1).

   We choose as potential function the number of bits sets to 1. Incrementing the counter requires k'+1 operations, where k' is the number of consecutive low-order bits set to 1. Then, the potential function decreases by k'-1. Therefore, the amortized cost per increment is 2.

2) We still consider a k-bit binary counter initialized to 0. Prove that we cannot both increment and decrement this counter in amortized O(1).

   We increment this counter until the highest order bit is set to 1, and all others to 0 (power of 2). Then, we alternatively decrement and increment the counter. Each operation changes the values of all bits. Therefore, the amortized complexity of each operation is in O(k), that is nonconstant.

3) Consider a k-size vector v whose each element has value in {-1,0,1}. This vector is interpreted as the number $\sum_i$ v[i]*2^i.
   a. Show that there is no unicity in the representation (i.e., two different vectors can be interpreted as the same number).

      The number 1 is the interpretation of both [0,....,0,1] and [0,....,1,-1].

   b. A vector can be modified as follows:
      Increment: we read the values v[i], by increasing indices i, until we find v[i] $\neq$ 1. Then, we set v[i] = v[i]+1 (0 if v[i]=-1, 1 if v[i]=0), while for every j < i v[j] = 0.
      Decrement: we read the values v[i], by increasing indices i, until we find v[i] $\neq$ -1. We set v[i] = v[i]-1 (-1 if v[i]=0, 0 if v[i]=1), while for every j < i v[j] = 0.
      Prove that increment/decrement operations require amortized O(1) time.

      We choose as potential function the number of non-zero values in the vector. Incrementing the vector requires k'+1 operations, where k' is the number of consecutive indices (starting from 0) set to 1. Then, the potential function decreases by at least k'-1. Similarly, decrementing the vector requires k''+1 operations, where k'' is the number of consecutive indices (starting from 0) set to -1. Then, the potential function decreases by at least k''-1. In both cases, the amortized cost of each operation is at most 2.

4) Consider a sorted list of integers. In order to keep the list sorted, upon inserting a new value i, we iteratively remove from the bottom of the list all elements greater than i. Show that the amortized cost for an insertion is in O(1).

   We choose as potential function the size of the list.

5) A *Union-Find* data structure maintains a collection of disjoint subsets over n elements, while supporting the following operations:
   - Find(i): Returns an identifier of the subset to which element i belongs
   - Union(i,j): Merges the respective subsets containing i and j into one.

a. Consider a naive implementation, as a vector `rep` storing, for each i, the identifier of its subset. What are the respective complexities of Find and Union?

   Find is obviously in worst-case O(1) since we just need to read the ith element of the vector. However, Union is in O(n). This is because we need to scan the whole vector in order to list all elements in one group (say, the group containing element i) and then update the subset containing these elements.

b. Let us associate to each subset one of the elements it contains as its identifier. Then, in an n-size array of lists, we may store at position i all elements in the subset of identifier i, if it exists. In doing so, whenever we merge two groups into one, we can always choose to insert all elements of the *smallest* group into the *largest* group. What are now the respective complexities of Find and Union?

   Find can still be done in worst-case O(1). Furthermore, Union is still in worst-case O(n). However, the total time for all Union operations is upper bounded by O(nlog(n)). Indeed, each time an element changes her group (as the result of a Union operation), the size of her group doubles. This is because only the elements in the smallest group are changing for the largest group. As a result, a given element can change her group at most O(log(n)) times.

6) 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 $\geq$ i, of $B_{a-1}$ and we end up incrementing the result also by 1 for each even element v[q], $j \geq q$, of $B_{a+t+1}$. Since each block has at most $\sqrt{n}$ elements, it also takes $O(\sqrt{n})$ time.

7) 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 invertions between i and j, that is, the number of pairs (r,s) such that $i \leq r < s \leq j$ and $v[r] > v[s]$?''

First let us consider the case i = 0, j = n-1 (invertions in a vector, without any range restrictions). We can solve this problem in O(n*log(n)) time, as follows. 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.

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 $B_a, B_{a+1}, \ldots, B_{a+t}$ 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.

8) We are given as input an n-size integer vector v. Show that after a pre-processing in O(n*log(n)) time, we can answer the following type of range queries q(i,j,e): ``does there exist an index k between i and j so that v[k] = e?''.

We construct a sorted copy of v in O(n*log(n)) time, of which we remove all duplicates. Let w be the resulting vector. For each element e of w, we associate a sorted vector u[e] of all positions k such that v[k] = e. Now, in order to answer a query q(i,j,e), we first perform a binary search in w to find e. If e is found, then we perform a binary search in u[e] to find the smallest element greater or equal to i.

9) A word is a palindrom if it is equal to its mirror. In particular, ``aba'' is a palindrom but ``abbc'' is not. Prove that we can recognize palindroms in O(n) time, n being the length of the input word.

We insert all characters of the word in a stack. Note that in doing so, the stack now contains the mirror of the input word. We repeatedly compare each character in the input word to the top element of the stack (obtained using pop() operation) until either we find a discrepancy or we emptied the stack.

10) We are given three types of parentheses: ( and ), [ and ], { and }. A word is well-formed if: either it is the empty word, or it can be written as (u)w, [u]w or {u}w with u and w being also well-formed. Equivalently, each parenthesis must be closed by a parenthesis of the matching type. In particular, ``([[]]){}'' is well-formed, but ``{]'' and ``[[`` are not. Show that, given an n-length word, we can check whether it is well-formed in O(n) time.

We scan the input word and put all opening parentheses in a stack. Whenever we read a closing parenthesis, we check whether it matches the type of the top parenthesis in the stack (obtained using pop() operation). Once we ended scanning the input, we end up verifying whether the stack is empty.