1) In what follows, we are given a perfect random generator, that outputs a single bit (equal to 1 with probability ½, resp. equal to 0 with probability ½).

   a. Propose an algorithm in order to generate uniformly at random a number between 0 and some $2^p$ (power of two).

      It suffices to call the perfect generator in order to output p-1 different bits.

   b. Propose an algorithm in order to generate uniformly at random a number between some integer a and some integer b.

      First, we observe that it suffices to generate uniformly at random a number between 0 and b-a. Therefore, in what follows let us assume $a = 0$. Let p be the smallest integer such that $b \leq 2^p$. We generate uniformly at random a number between 0 and $2^p$ until we get a number less than b. Since $2^p < 2b$, the probability NOT to generate a number less than b is at most ½, and therefore we only need O(1) rounds in expectation.

   c. Propose an algorithm in order to generate uniformly at random an even number between some integer a and some integer b.

      We may assume a (resp., b) to be even, since otherwise we can replace it by a+1 (resp., b-1). We output uniformly at random a number between a/2 and b/2, then we double the outcome.

2) Given an n-size vector v, the element uniqueness problem asks whether all elements in v are pairwise different (i.e., $i \neq j \Rightarrow v[i] \neq v[j]$). Propose an algorithm to solve this problem in expected O(n) time.

   We use an auxiliary hash table. Specifically, we read all elements v[i] sequentially. We first check whether v[i] is present in the hash table in expected O(1) time. Otherwise, we add v[i] as a new key in this table, also in expected O(1).

3) Given an n-size vector v, the frequency of a given element e is its number of repetitions in v (i.e., number of indices i such that $v[i] = e$). Propose an algorithm to output an element of v with maximum frequency, in expected O(n) time.

   This is an easy variation of the previous exercise, but where we exploit the additional property that, to any key in a hash table, we can associate some arbitrary value. Now, we read all elements v[i] sequentially. We first check whether v[i] is present in the hash table in expected O(1) time. If that is NOT the case, then we add v[i] as a new key in this table, with associated value 1. Otherwise, we increment by one the value already associated to v[i] in this table. We end up scanning the hash table once in order to output a key with maximum associated value.

4) Consider an n-size vector v. Show that after pre-processing this vector in expected O(n) time, we can answer in expected O(1) time to the following type of queries q(e):``does there exist an i such that $v[i] = e$?''.

   It suffices to insert all elements of the vector in a hash table.

5) Consider an n-size vector v. Show that after pre-processing this vector in expected O(n) time, we can answer in expected O(1) time to the following type of queries q(e):``output the smallest/largest i such that v[i] = e, if it exists, and -1 otherwise.''

   Here also this is a variation of the previous exercise. We scan the elements v[i] from i = 0 to i=n-1 by increasing order (or by decreasing order for the largest variant of the problem). If v[i] is not already present in some auxiliary hash table, then we add it with associated value i (otherwise, we do nothing).

6) We still consider an n-size vector v. We consider the following (slightly more complex than before) range queries q(i,j,e): ``does there exist an index k between i and j such that v[k]=e?''.
   a. Show that after pre-processing the vector in expected O(n) time, we can answer to these above range queries in expected O(log(n)) time.

      We insert all elements of the vector in a hash table. To each key e in the table, its associated value is the sorted vector of all indices i such that v[i] = e. Note that we can construct ``for free'' these associated values while scanning the vector v (no need to use a sorting algorithm). Therefore, the whole pre-processing stage takes O(n) time in expectation. In order to answer a query q(i,j,e), we search for key e in the hash table, in expected O(1) time. Then, we search in its (sorted) associated vector the smallest index k greater or equal to i. It can be done in O(log(n)) time using binary search.

   b. In the offline variant of the problem, we are given in advance the m range queries $q(i_r,j_r,e_r)$, $0 \leq r \leq m-1$, to be answered. Propose an algorithm to solve the offline variant in expected O(n+m) time.

      We scan the vector from left to right. Upon reading the element v[i], we check whether it is already present in some auxiliary hash map. If not, then we insert it with associated value 1. Otherwise, we increment its associated value by 1. Note that at any step i of this algorithm, for any key e in the hash map, its associated value e.val equals its number of repetitions between 0 and i.

      Therefore, before starting step i of the algorithm, we first consider all queries $q(i_r,j_r,e_r)$ such that $i_r = i$. We store in some auxiliary variable u[r] the value: 0 if $e_r$ is not present in the hash table, and $e_r$.val otherwise. In the same way, after ending step i of the algorithm, we now consider all queries $q(i_r,j_r,e_r)$ such that $j_r = i$. By comparing u[r] with $e_r$.val, we can answer to the query $q(i_r,j_r,e_r)$.

7) Consider an n-size vector v. The 2-sum problem asks whether there exist i < j such that v[i]+v[j] = 0. Propose an algorithm to solve this problem in expected O(n) time (For your information, the 3-sum problem, where we ask whether there exist i < j < k such that v[i]+v[j]+v[k]=0 is conjectured to require essentially $O(n^2)$ time).

   We insert all elements of the vector in a hash table. Then, for each element v[i], we search for its negation –v[i] in the table.

8) Propose a data structure in which all following three operations can be done in O(1) time: i) emptyness test, ii) insertion of a new element, iii) output (and removal from the

structure) of the kth eldest element (i.e., exactly k-1 elements amongst those in the structure were added before it); if there are less than k elements in the structure, then output the eldest element (latest added to the structure).

Solution 1: maintain the k eldest elements in a stack and all other elements in a queue. Whenever we output the kth eldest element (using pop() operation), we replace it (if the latter is nonempty) by the top element in the queue. Note that by reverting the respective roles of stack and queue, we can also output the kth youngest element of the structure.

Solution 2: maintain all elements in a doubly linked list, ordered by insertion time. We just maintain one more pointer to the kth eldest element in the structure. Note that we can easily update this pointer after the removal of an element.

9) Consider an n-size boolean vector v, whose values are initially 0. We want to answer to the following types of requests:

- SET i: set v[i] = 1
- MIN i: outputs the smallest index $j \geq i$ such that v[i] = 1.

Show that the offline variant of this problem, where we are given in advance m requests to be answered, can be solved in O(m+n*log(n)) time.

Let $R_1$, $R_2$, ... $R_m$ be the m requests. We first scan each request and, if it is under the form $R_k$ = SET $i_k$, we set v[$i_k$]=1. It takes O(m) time. In a second phase of our algorithm, we now consider all the requests by decreasing index (from k=m to k=1). If $R_k$ = SET $i_k$, then this time we set v[$i_k$]=0. Indeed, this is because the only requests left to be answered are for indices k' < k (before we set v[$i_k$]=1). Let us now consider the case $R_k$ = MIN $i_k$.

We use an auxiliary union-find data structure, where initially each position in the vector is put in a separate set. For each subset S in the data structure, we further retain max(S), that is the maximum index j such that j is in S. The union-find data structure can be trivially initialized in O(n) time.

Let S = Find($i_k$) be the subset to which $i_k$ belongs. Our algorithm ensures that S is an interval, i.e., it contains all r between some i' and max(S); moreover, v[r] = 0 for every r in S, except maybe if r = max(S). These properties are trivially true at the initialization of the union find data structure. Then, if v[max(S)] = 1, then we answer to the request $R_k$ = SET $i_k$ with j = max(S). Otherwise, we call Union(Find($i_k$),Find(max(S)+1)), while still calling S the output. We continue this process until either max(S) = 1 or max(S) = n-1. Overall, since we know how to do all union operations in total O(n*log(n)) time, the total runtime that we achieve is in O(m+n*log(n)).

Remark: This is actually faster than what we claim, if we use an optimized version of Union-Find. In particular, we may achieve O(n+m) time.