1) Consider an n-size vector v. Show that after pre-processing this vector in O(n*log(n)) time, we can answer in O(log(n)) time to the following type of queries q(i,j):``what is the minimum value stored in the subvector v[i...j] ?''.

We precompute q(i, $i+2^k-1$), for every index i and every exponent k such that $i+2^k-1 < n$. For that, we proceed by induction:

if k=0, then q(i,i) = v[i]

if k > 0, $q(i, i+2^k-1) = \min\{ q(i,i+2^{k-1}-1), q(j, j+2^{k-1}-1) \}$ with $j = i + 2^k$

Since we only need to go until k = log(n), the preprocessing time is in O(n*log(n)).

In order to answer to a query q(i,j), we partition the interval [i;j] in many subintervals. More precisely, let $i_0 := i$, and let r:=0. While $i_r < j$, we compute the largest exponent k_r such that $i_r + 2^{kr} -1 \leq j$. We set $i_{r+1} := i_r + 2^{kr} -1$. Note that we stop after at most O(log(n)) steps. Since for all subintervals $[i_s;i_{s+1}]$ we precomputed $q(i_s,i_{s+1})$, we are done in O(1) time per subinterval, and so in O(log(n)) time.

2) 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.

3) 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.

4) A word is made of open `(` and closed `)' parentheses. It is well-formed if: either it is the empty word, or it can be written as (u)w, with u and w being also well-formed. Equivalently, each open parenthesis must be closed. For instance, `((()))()(())' 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 use an auxiliary counter, initially equal to 0. We scan the word once. Each time we read an open parenthesis, we increment the counter by one. Each time we read a closed parenthesis, we decrement the counter by one. If at some point the counter becomes negative, then there are too many closed parentheses and we reject. Finally, once we ended scanning the input, we end up verifying whether the counter equals zero (all parentheses were closed).

5) Let w be a fixed word of size n, which is made of open `(` and closed `)' parentheses. Show that after preprocessing this word in O(n*log(n)) time, we can answer in O(log(n)) time to the following type of queries q(i,j): ``is the subword w[i...j] well-formed?''

We first combine the algorithm of the previous question with a classic ``partial sum'' trick. Specifically, we create an auxiliary vector s[n] such that:

s[0] = 1 if w[0] = `(`, and s[0]=-1 if w[0]=`)`

s[i] = s[i-1]+1 if w[i]=`(` and s[i]=s[i-1]-1 if w[i]=`)`.

It can be done in O(n) time.
Second, we preprocess vector s as explained in question 1). It takes O(n*log(n)) time.

In order to answer to a query q(i,j), let the offset c(i) be such that: c(i)=0 if i=0, and c(i)=s[i-1] if i > 0 (intuitively, this is the value of our counter before we start scanning the subword). It suffices to check whether:
s[j] = c(i) (otherwise we reject)
the minimum element of s[i...j] is c(i) (otherwise, the counter goes below c(i) at some point, i.e., there are too many closed parentheses, and we reject).
It takes O(log(n)) time to compute the minimum element of s[i...j].

<u>Remark</u>: there exists an optimal algorithm with O(n) preprocessing time and O(1) query time. It is the same algorithm as above, but where we used a faster solution for question 1), using so-called ``Cartesian trees'' (more on that later in the course).

6) **Generalization.** 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.

7) 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.

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

We sort the vector then we check whether two consecutive elements are equal.

   b. Propose an algorithm to solve this problem in expected O(n) time.

Reminder about Hash Tables: insertion, deletion and lookup in expected O(1) 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).

9) 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).
   a. Propose an algorithm to output an element of v with maximum frequency, in O(n*log(n)) time.

We sort the vector, and we count the number of repetitions of each element.

   b. Let us assume (only for this question) that all values are between –n and 99*n. Propose an algorithm to output an element v with maximum frequency in O(n) time.

Use a frequency vector of size O(n).

   c. 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.

10) Consider an n-size vector v. The 2-sum problem asks whether there exist $i < j$ such that $v[i]+v[j] = 0$.
   a. Propose an algorithm to solve this problem in O(n*log(n)) time.

This what is sometimes called the ``meet in the middle'' method. We sort the vector v in O(n*log(n)) time. Then, we have two varying indices i,j, initially set to 0 and n-1. While i < j, we compare v[i] with v[j]:
*** if v[i]+v[j] = 0, then we are done**
*** if v[i]+v[j] > 0, then j = j-1 (we need smaller values to reach 0)**

*** if v[i]+v[j] < 0 then i=i+1 (we need bigger values)**

b. Propose an algorithm to solve this problem in expected O(n) 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.

11) 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.

12) We still consider an n-size vector v. We consider the following range queries q(i,j,e): ``does there exist an index k between i and j such that v[k]=e?'' They were already discussed at a previous seminar: we presented a solution with O(n*log(n)) preprocessing time and O(log(n)) query time. In what follows, we improve this result:

.

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 $q(i_r,j_r,e_r)$.