1) A word is made of open `(` and closed `)' parentheses. A subword is made of consecutive letters. Prove that we can compute a longest well formed subword in O(n) time.

The key observation is as follows. Let us assume that we store a counter c, initially set to 0, which we increment/decrement after each open/closed parenthesis. Let i be the first index such that c becomes negative (if it exists). Then, a well-formed subword is fully between positions 0 and i-1 or it is fully between positions i+1 and n-1 (i.e., there is no overlap). In particular, we may restart the algorithm each time the counter c becomes negative, and then we never need to go back in the vector.

More precisely, the algorithm works as follows. We maintain a starting position s (initially, s = 0), an optimal length L (initially, L=0) and a counter c (initially set to 0). We increment/decrement c after each open/closed parenthesis. Let i denote our current position.
*** If c=0, then we found a new well-formed subword: L = max{L,i-s+1}**
*** If c < 0, then we reset c=0, s=i+1**

2) Let v[] be a vector of n positive integers. Being given an integer k, compute in O(n) time the minimum length of a subvector v[i..j] such that the sum of all its elements is at least k.

For every index i, let `min[i]` be the least index j such that `v[i...j]` satisfies the desired property (the sum of all its elements is at least k). Since every element is positive, we have: `min[i+1] >= min[i]`. Therefore, we can compute the vector `min[]` as follows:
*** we use a partial sum trick: for every index i, let `s[i]` be equal to `v[0]+v[1]+...+v[i]`. We can compute vector `s[]` in O(n) time by dynamic programming.**
*** set i = j = 0 (here, i represents the first index for which we do not know `min[i]`; j is a lower bound for `min[i]`).**

```
* while i < n do:
   while j < n and s[j]-s[i]+v[i] < k do:
     j=j+1
   min[i] = j
   i = i+1
```

*** finally, we iterate over all indices i such that `min[i] < n` and we output the minimum of `min[i]-i+1`.**

3) A frequency-stack is a data structure supporting the following operations:
* empty(): asserts whether the structure is empty
* push(e): add a new element e to the structure (possibly the repetition of a previous element)
* pop(): returns and deletes a most frequent element in the structure. <u>All occurences of e are removed at once.</u>
Describe an implementation such that every operation can be performed in expected O(1) time.

We store a ``list of list" F. More precisely, F is a doubly-linked list, whose every element is a pair (r,L(r)). The list L(r) contains every element repeated r times in the structure. Furthermore, F is ordered by increasing number of repetitions, i.e.: F=[r1,r2,…rq] with r1 < r2 <… < rq.

***empty(): we just check whether F is empty.**

***push(e): There are several cases and subcases**
a) Case e was never added to the structure before
Let r1 be the head of F
If r1=1 then we add e to L(1). Otherwise, we prepend 1 in F, and we set L(1) = [e]
b) Case e was already inserted r times to the structure before.
We remove e from L(r).
If r is the tail of F, or the successor r' of r in F is not r+1, then we insert r+1 in F immediately after r, and we set L(r+1) = [e]. Otherwise, we add e to L(r+1).

***pop(): Let rq be the tail of F. We remove any element from L(rq). If L(rq) becomes empty, then we remove**

rq from F.

Every operation can be done in O(1) time if, for any element e, we can:
- compute in O(1) time the number of repetitions of e in the structure (or decide that e was never added before)
- if there are r repetitions of e, access in O(1) time to the position of e in list L(r).
We can store this information in an auxiliary Hash-table. Thus, any operation can be done in expected O(1) time.

4) Let u[] and v[] be vectors of length n. We say that u and v are anagrams if they have the exact same elements (counted with multiplicities), but not necessarily in the same positions. Example: [1,1,2,3,1] and [1,2,1,3,1] are anagrams. Propose an algorithm in expected O(n) time in order to decide whether u,v are anagrams.

We create a Hash-table H whose keys are the elements of vector u, and such that H[e] is the number of repetitions of e in u. This can be done in expected O(n) time simply by scanning vector u once. Then, we consider each element of v sequentially, from j=0 to j=n-1. Let e = v[j] be the current element. If e is not a key of H, then we stop (e is in v but not in u, therefore u and v are not anagrams). Otherwise, we decrement H[e] and, if H[e] drops to 0, we remove e from H. If we end scanning v without stopping, then we accept.

5) Let v[] be a vector of n positive integers.
a) Compute the maximum length of a subvector v[i...j] such that all its elements are pairwise distinct.

We scan vector v from left to right. Let e = v[j] be the current element. During the scan, we store in an auxiliary Hash-table all elements e' encountered on previous positions 0,1,…,j-1 and the <u>last</u> position where each element was found. In particular, we have access to H[e], the largest index j' < j such that v[j'] = v[j] = e, with the convention that H[e] = -1 if no such index exists. During the scan, we compute a vector b[] such that, for every j, b[j] equals the least index i such that v[i...j] satisfies the desired property (all its elements are distinct). Note that computing b[] is actually sufficient in order to solve our problem. We compute all values b[j] by induction:
*** if j=0, then b[j] =0**
*** otherwise, b[j] = max{b[j-1],H[e]+1}, where e = v[j].**

b) Compute the maximum length of a subvector v[i..j] that satisfies Fibonacci recurrence, i.e.: for every k between i+2 and j, we must have v[k] = v[k-1] + v[k-2].

First we create an auxiliary vector fibo[] of length n-2 such that: fibo[k] = 1 if and only if v[k+2] = v[k]+v[k+1] (else, fibo[k] = 0). This can be done in O(n) time by scanning the vector v once.

Then, we compute max-fibo[], also of length n-2, such that max-fibo[i] equals the maximum length of a Fibonacci subvector that starts in position i.
*** If fibo[i] = 0, then max-fibo[i] = 2.**
*** If i=n-3 and fibo[i] =1, then max-fibo[i] = 3**
*** if i < n-3 and fibo[i] = 1, then max-fibo[i] = 1 + max-fibo[i+1]**

The total running time is in O(n).

<u>Remark</u>: The runtime for 5)b) is slightly better than for 5)a), for which the running time was also O(n) but only in expectation. The reason for that is the Fibonacci property is local, whereas checking that all elements are pairwise distinct is a global property. For global properties, it is often difficult to avoid using hash-table, whereas simpler tricks often work for local properties.

6) Let v[] be a vector of n positive integers. Compute, in expected O(n) time, the smallest positive integer e that is NOT an element of v.

It suffices to consider integers e between 1 and n+1 (since we only have n elements of e, the smallest missing integer must be in this range). For that, we store all elements of v in a hash-table H. Then, for e from 1 to n+1, we search for e in H.