1) A *median-heap* is any data structure supporting the following operations:
   - emptiness test
   - insertion of a new integer
   - output and removal of the median (if the data structure contains an even number of elements, output and removal of the upper median).

   Propose an implementation with best possible amortized query times for each operation.

   We internally maintain a max-heap H1 and a min-heap H2 such that:
      i) the smallest $\lfloor n/2 \rfloor$ elements are stored in H1
      ii) the largest $\lceil \frac{n}{2} \rceil$ elements are stored in H2.
   Each operation can now be implemented as follows:

   - <u>Emptiness test</u>: we test whether both heaps are empty – **in worst-case O(1)**.

   - <u>Median</u>: We call getMin() + deleteMin() on H2. Indeed, the (upper) median is always the least element in H2. Let n' = n-1. If, after removal of the median, the number of elements in H2 becomes $\left\lceil \frac{n'}{2} \right\rceil - 1$ then we pick be the maximum element in H1 (obtained with getMax()), we remove it from H1 (operation deleteMax()) and we insert it in H2. – **in O(log(n)).**

   - <u>Insertion</u>: Let m denote the current median in the data structure (min. element in H2) and let n'=n+1. If the newly inserted element is smaller than or equal to m, then we insert it in H1; if after this insertion, the number of elements in H1 becomes$\left\lfloor \frac{n'}{2} \right\rfloor + 1$, then we pick the maximum element in H1, we remove it from H1 and we insert it in H2. Else (the newly inserted element is larger than m), we insert the new element in H2; if after this insertion, the number of elements in H2 becomes $\left\lceil \frac{n'}{2} \right\rceil + 1$, then we remove m from H2 and we insert it in H1. – **in O(log(n)) even if we use Fibonacci heaps** (because we sometimes need to delete an element).

2) Recall that a heap can be either a max-heap or a min-heap. Propose a linear-time algorithm in order to transform a max-heap into a min-heap.

   The trick consists in selecting the right implementation for a heap. In a Fibonacci min-heap the amortized complexity for inserting a new element is in O(1). So, we can simply add one by one all elements from the max-heap into a Fibonacci min-heap.

3) Propose a linear-time algorithm in order to transform a binary research tree into a max-heap.

   Solution 1: Output a postorder of the elements in the binary research tree. In doing so, we sorted all elements from largest values to smallest values. This sorted vector can be considered as a valid support for a binary max-heap.

   Solution 2: Insert one by one all elements in a Fibonacci max-heap (Insertion can be done in constant amortized time in a Fibonacci heap).

4) A min-max heap supports the following operations:
   - emptiness test
   - insertion of a new integer
   - output of the minimum element
   - deletion of the minimum element
   - output of the maximum element
   - deletion of the maximum element

   Propose an implementation with best possible amortized query times for each operation.

   We maintain a max-heap H1 and a min-heap H2 over the same set of elements. Each element in the max-heap H1 keeps a pointer to its current position in the min-heap H2, and vice-versa.
   - emptiness: test whether any of H1 or H2 is empty. **– in worst-case O(1).**
   - insertion: insert in both H1 and H2. **– in amortized O(1) if we use Fibonacci heaps (or in worst-case O(log(n)) if we either use binary heaps or binomial heaps).**
   - output min: output min in H2. **– in worst-case O(1).**
   - delete min: delete min in H2. Access to the current position of the removed element in H1. Deletion of this element in H1. – **in amortized O(log(n)) if we use Fibonacci heaps (or in worst-case O(log(n)) if we either use binary heaps or binomial heaps).**
   - output max: output max in H1 **– in worst-case O(1).**
   - delete max: delete max in H1. Access to the current position of the removed element in H2. Deletion of this element in H2. – **in amortized O(log(n)) if we use Fibonacci heaps (or in worst-case O(log(n)) if we either use binary heaps or binomial heaps).**

5) A number is ugly if and only if its only prime factors are amongst 2,3 and 5. In particular, the first ugly numbers are 1,2,3,4=2x2,5,6=2x3,8=2x2x2,… A number is beautiful if it is not ugly. In particular, 7 is the smallest beautiful number.

   Write algorithms with best possible complexity for outputting the nth smallest ugly number, resp. the nth smallest beautiful number.

   Ugly numbers: We keep in a min-heap a subset of the smallest ugly numbers not enumerated yet. Initially, we just store the number 1 (first ugly number) in the heap. Upon extracting from the heap the smallest next ugly number (minimum element in the heap), call it u, we replace it into the heap by the elements 2*u, 3*u and 5*u. We need to check that none of these elements got inserted into the heap before. This can be done by keeping track of all ugly numbers inserted into the heap in a Hash table. The runtime is in O(n*log(n)) because we never have more than 2*n numbers in the heap.
   Observation: we can do without using a Hash table, which is a little bit better because the latter is randomized. For that, let us consider the number 2*u. Suppose that u = 3*v. Then, we enumerated 2*v before u, and so we needn't add 2*u = 3*(2v) in the heap. Same if u is a multiple of 5, etc.

   Beautiful numbers: Denote by U(i) the ith ugly number. We modify the above algorithm for computing ugly numbers so that it also computes N(i): the number of beautiful numbers smaller than U(i). Note that N(0) = 0 and N(i) = N(i-1) + U(i) – U(i-1) – 1. Therefore, computing also N(i) induces no overhead in the runtime of the algorithm. Now, to compute the nth smaller beautiful number, it suffices to compute ugly numbers U(i) until we obtain N(i) ≥ n. Then, the nth beautiful number is exactly U(i-1) + (n-N(i-1)). To analyse this algorithm, we start observing that for any integer k, only $O(\log^3(k))$ ugly numbers are smaller than k. Indeed, this is because there are only O(log(k)) powers of 2 (resp.,3,5) smaller than k. Therefore, if $k - O(\log^3(k)) \geq n$, then we only need to compute $O(\log^3(k))$ ugly numbers. For n large enough, we may simply choose k = 2n. Then, the runtime is in $O(\log^3(n)*\log\log(n))$.

6) Modify a min-heap implementation so that we have the following tie-break rules:
 a) Rule 1: if there are many elements of minimum value in the heap, always output the oldest one.
 b) Rule 2: if there are many elements of minimum value in the heap, always output the youngest one.

Solution 1: We keep in a Hash-table, for each element in the heap, an abstract data structure where we store all its repetitions inserted into the heap. Upon deleting the minimum element, we access to its associated data structure in the Hash-table, of which we remove the top element. If this data structure becomes empty (the element is no more in the heap), then we also call the deleteMin() operation.
a) We use as associated data structure a queue
b) We use as associated data structure a stack

Solution 2: we keep an internal counter c, initially set to 0. This counter must be incremented after each insertion (it is never decremented).
a) Upon inserting a new element with priority p, we assign it priority (p,c).
b) Upon inserting a new element with priority p, we assign it priority (p,-c).

Remark: Solution 2 is just the usual simulation of stacks/queues using a priority queue.

7) Propose an O(n+klog(k))-time algorithm in order to extract the k smallest elements in a vector.

This can actually be done faster, in O(n). Indeed, it suffices to apply quickselect in order to extract the kth order statistic ☺. We pay an additional O(k*log(k)) time if we further want to sort these k elements.

8) Propose an O(n*log(k))-time algorithm in order to merge k sorted lists into one sorted list.

We store in a min-heap the smallest element of each list. Whenever we extract the root of the min-heap, we delete it from its original list (which can be done in O(1) time because the list is sorted) and we put the new head of this list in the heap. Overall, since there are always at most k elements in the min-heap, the runtime is O(log(k)) per element.

9) Consider k sorted lists, whose respective sizes sum up to n. A valid range is an interval [a;b] such that every of the k lists contains an element between a and b. Propose the fastest possible algorithm for computing a valid range of minimum length.

This is essentially the same approach as for the previous exercise. We store in a min-heap the respective smallest elements (= heads) of each list. Consider the smallest element in the heap, call it i. Let j denote the maximum element in the heap (which can be stored and maintained in an auxiliary variable). Then, the minimum range which contains i and an element of each of the other k-1 lists is exactly [i;j]. We remove i from the min-heap and from its original list, then we add the new head of this list in the heap. We repeat this process until one of the lists becomes empty. Since there are at most k elements at any time in the heap, the total runtime is in O(n*log(k)).