Problem 1

1. Heapsort is not stable: Consider the array of key-value pairs $[(0, a), (1, a), (1, b)]$. This array already represents a min-heap. One call to `deleteMin` will return $(0, a)$ and the heap will be updated to $[(1, b), (1, a)]$. Next calls to `deleteMin` return $(1, b)$ and then $(1, a)$. Therefore the output of Heapsort is $[(0, a), (1, b), (1, a)]$ and Heapsort is not stable.

2. QuickSort1 is not stable: Consider the array of key-value pairs $[(2, a), (1, a), (1, b)]$. QuickSort1 chooses $(2, a)$ as a pivot. The call to partition will only execute the final swap, leading to an array $[(1, b), (1, a), (2, a)]$ after partitioning. This array is sorted but switched the order of elements of key 1.

Problem 2

`Solution 1:` Partition the array $A = [B_0|B_1|\cdots]$ in blocks of size $\ell := \lceil\log(n)\rceil$, *i.e.* first block is $B_0 = A[0, \ldots, \ell - 1]$, second block is $B_1 = A[\ell, \ldots, 2\ell - 1], \ldots$. In general, i-th block B_i is $A[i\ell, \ldots, (i + 1)\ell - 1]$ and there are $\Theta(n/\log n)$ such blocks. Note that the last block may have less than ℓ elements if n is not a multiple of ℓ.

The hypothesis $A[i] \geq A[i - j]$ for all $j \geq \log n$ guarantees that the elements of a block B_i are all greater than or equal to any element of B_j as long as $i \geqslant j + 2$. Our algorithms starts by sorting the array $[B_0|B_1]$ and cut the sorted array in two blocks $[B'_0|B'_1]$ of size ℓ.

We assert that smallest ℓ elements of A are to be found in B'_0 in order. Indeed, these ℓ smallest elements can not be found in $B_2, B_3, \ldots$ because all the ℓ elements of B_0 are smaller than any element of $B_2, B_3, \ldots$. So the smallest ℓ elements are in $[B_0|B_1]$, and therefore in B'_0 after sorting.

Then we recursively call our algorithm on the subarray $[B'_1|B_2|B_3|\cdots]$. The important thing is that this subarray still satisfies the property "the elements of a block B_i are all greater or equal to any element of B_j as long as $i \geqslant j + 2$". Indeed, the elements of B'_1 are smaller than any element in $B_3, B_4, \ldots$ because B'_1 is made of elements of B_0 and B_1.

Let us look at the cost of our algorithm. Sorting two consecutive blocks of size ℓ takes $O(\log n \log\log n)$. The cost of our sorting algorithm on an array of r blocks is $T(r) = O(\log n \log\log n) + T(r - 1)$. Since the number of blocks is $\Theta(n/\log n)$, the overall time complexity is $O(n \log\log n)$.

`Solution 2:` If we consider every $\ell = \lceil\log(n)\rceil$-th element, that is $A[0], A[\ell], A[2\ell], \ldots$, the elements will be in sorted order. Similarly for $A[1], A[1 + \ell], A[1 + 2\ell], \ldots$ or in general $A[i], A[i + \ell], A[i + 2\ell], \ldots$ for $i < \ell$. Therefore, we can first obtain ℓ sorted sequences, each consisting of $\Theta(n/\log n)$ elements. Now we can do a ℓ-merge similar to Assignment 2 to get the sorted array. That is, insert the smallest element from each sequence into a min heap, extract the minimum say m, and insert the next smallest element of the sequence where m comes from into the heap.

Notice that heap size is always $O(\log n)$. Therefore heap operations takes only $O(\log\log n)$ time and the overall complexity is $O(n \log\log n)$.

lajaneck

This document is for the exclusive use of lajaneck.

Problem 3

1. ```
   a) i := 0
   b) while (i < n) do
   c)   if (A[i] == i) then
   d)     i := i+1
   e)   else
   f)     Swap(A[A[i]],A[i])
   ```

2. At the beginning of each loop iteration $A[0], A[1], \ldots, A[i-1] = 0, 1, \ldots, i-1$. Each iteration of the loop either increment i or increases by at least one the number of array elements that are in correct position. Thus, the number of iterations is between n and $2n$, giving a running time in $\Theta(n)$.

   More details (not required for full marks): The only possible modifications made to A are to swap two elements. Thus, $A[0, \ldots, n-1]$ remains a permutation of $0, \ldots, n-1$ throughout the execution.

   The first time the loop iterates, we have $i = 0$, so the loop invariant

   $$A[0], A[1], \ldots, A[i-1] = 0, 1, \ldots, i-1 \tag{1}$$

   holds trivially. Induction on the number of times the loop iterates shows that (1) holds throughout execution. One the one hand, if line d) is executed, then $A[i] = i$ and upon incrementing i, invariant (1) still holds. On the other hand, after line f) is executed, we get $A[j] = j$ for $j = A[i]$. Since $A[j] \neq j$ for $j \in \{i, A[i]\}$ beforehand, we have increased by at least one the number of array elements in correct position.

Problem 4

This algorithm was explained in class: it corresponds to solution 4, slide 18 of module 2. There are three steps :

i. Perform an in-place heapify so that $A[0, \ldots, n-1]$ is a min-heap.

   Running time: $O(n)$

ii. Perform k `deleteMin` (or equivalently `extractMin`) operations to obtain the k smallest integers in the array. Using the implementation `heapDeleteMax` of slide 14 of module 2 for Algorithm `deleteMin`, we obtain the minimum in position $A[n-1]$, the second minimum in position $A[n-2], \ldots,$ and the k-th minimum in position $A[n-k]$.

   Running time: $O(k \log n)$ which is $O(n)$ if $k \in O(n/\log n)$.

iii. Reverse the order of the elements in the array by interchanging $A[i]$ and $A[n-1-i]$ for $i = 0, 1, \ldots, \lfloor (n-1)/2 \rfloor$.

   Running time: $O(n)$.

This document is for the exclusive use of lajaneck.
   ```

Problem 5

1. Each player is weak or strong, but the two cases where all players are either all weak or all strong are excluded. The total number of possible outcomes is thus $2^n - 2$. Each contest has exactly 3 possible outcomes, so if an algorithm performs at most k contests, the number of different answers the algorithm could return is at most 3^k. So we must have $3^k \geqslant 2^n - 2$; solving for the integer k yields the lower bound of $\lceil \log_3(2^n - 2) \rceil$ contests.

2. Using the algorithm of part 3, we need 3 contests when $n = 4$. This matches the bound

$$\lceil \log_3(2^4 - 2) \rceil = 3.$$

3. Perform exactly $n - 1$ contests between the pairs for players P_1 and P_i for $2 \leqslant i \leqslant n$. P_1 is weak if and only if all contest outcomes result in either a tie or loss for P_1, with at least one loss because there is at least one strong player; the weak players are those that tied P_1 and the strong players are those that won against P_1.

   Similarly, P_1 is strong if and only if all contest outcomes result in either a tie or win for P_1, with at least one win; the strong players are those that tied P_1 and the weak players are those that lost against P_1.

   The algorithm takes $(n-1) \in O(n)$ contests. Observe that for $n \geqslant 2$

$$\lceil \log_3(2^n - 2) \rceil \geqslant \lceil \log_3(2^{n-1}) \rceil = (n-1)\log_3(2)$$

   so the lower bound from part 1 is $\Omega(n)$. Therefore this algorithm is asymptotically optimal in the number of contests.

Problem 6

Algorithm is based on Counting Sort and has three steps:

i. Count the number of occurrences of each key through a linear scan of the array and store the result in an array C of size k (similar to the first step of counting sort).

   This takes $O(n)$ time.

ii. Next, update C to store the number of items smaller or equal to each key; this can be done by a linear scan of C (similar to the second step of counting sort.

   This takes $O(k)$ time.

iii. The number of items in the range $[a; b]$ is equal to $C[b]$ (*i.e.* the number of items smaller or equal to b) minus $C[a-1]$ (*i.e.* the number of items smaller than a).

   This takes $O(1)$ time.

lajaneck

This document is for the exclusive use of lajaneck.