Mathematics 5365 (Analysis of Algorithms)

Final exam

You may use any algorithm from the lectures without reimplementing it, provided that you cite it correctly (i.e., state what it does exactly, and cite the correct running time). There is no guarantee that any problem can benefit from these algorithms, though.

Solve as many problems as you can. Choose problems that are easiest to solve for you. The first two or three problems are supposed to be the easiest ones, but there is no guarantee.

1. Input data: $n\colon \mathbf{N}$, $x\colon \mathbf{N}[n]$, $r\colon \mathbf{N}$, $x[i] \in [0,n)$ for any $i \in [0,n)$. Output data: $y\colon \mathbf{N}[n]$ such that for all $i \in [0,n)$ we have $y[i] = x[x[x[\cdots x[i]\cdots]]]$, where on the right side x occurs exactly r times. Worst-case running time $O(n)$. Notice that the running time does not depend on r.

2. Input data: $S\colon \mathsf{Order}$, $n\colon \mathbf{N}$, $x\colon S[n]$. Output data: $i,j\colon \mathbf{N}$ such that $i,j \in [0,n)$, $x[i] \le x[k]$ and $x[j] \ge x[k]$ for any $k \in [0,n)$. Thus, $x[i]$ is the smallest element of x and $x[j]$ is the largest element of x. Worst-case running time $O(n)$. The total number of comparisons of elements of x must be $3n/2 + O(1)$, i.e., there is $C > 0$ and n_0 such that for all $n \ge n_0$ the algorithm performs at most $3n/2 + C$ comparisons. Thus, you must find the smallest and largest elements of x using only $3n/2+O(1)$ comparisons, and no other operations on elements of x are permitted, i.e., you know nothing about its internal structure. You may assume, if you want, that all elements of x are distinct.

3. Input data: $S\colon \mathsf{OrdMon}$, $n\colon \mathbf{N}$, $x\colon S[n]$. (Here S is a totally ordered monoid, such as $\mathbf{Z}$, $\mathbf{Q}$, i.e., S is a monoid, equipped with a total ordering, so that $x \ge y$ implies $x+z \ge y+z$ and $z+x \ge z+y$ for all z.) Output data: $i,j\colon \mathbf{N}$ such that $0 \le i \le j \le n$ and $\sum_{k\in[i,j)} x[k]$ has the maximum possible value (with respect to the ordering on S). Worst-case running time $O(n)$. In other words, you have to find a (consecutive) subarray of x with the maximum sum.

4. Input data: $E\colon \mathbf{N}$, $F\colon \mathbf{N}$. Output data: $D\colon \mathbf{N}$. You are given E eggs, which you may drop from any of F floors in a building. There is a certain number $T\colon \mathbf{N}$ (the threshold; $0 \le T \le F$), which is not given to you a priori, with the following property: an egg dropped from the floor T or below will survive the fall (and can be reused again), whereas an egg dropped above the floor T will not survive the fall (and must be discarded). Your objective is determine T by dropping eggs as few times as possible. That is, you must compute the smallest possible number of droppings D that is guaranteed to determine T. This means that there is an algorithm that computes T using D droppings no matter what T is, but there is no algorithm that can do it using $D-1$ droppings. Example: if $E = 1$, there is only one way to do it: drop the egg from floor 1, 2, 3, ..., until it breaks at floor f (and then $T = f - 1$), so in this case $D = F$, because we may need to drop the egg from every floor. For $E > 1$ one can do it with fewer droppings: if the first egg breaks, the second one can still be used. Worst-case running time: any polynomial time in E and F will be awarded points, the maximum number of points will be awarded for the $O(E \log F)$ worst-case running time.

5. Compute the average number of comparisons performed by the following algorithm. Input data: $S\colon \mathsf{Order}$, $n\colon \mathbf{N}$, $x\colon S[n]$, $q\colon \mathbf{N}$, all elements of x are distinct. Output data: x rearranged in the some arbitrary order so that the element $x[q]$ is in its correct position if the array was sorted, but other elements can occur in any order. (This element is known as the qth *order statistic* of x. Individual cases are known as medians, deciles, quartiles, etc.) The algorithm runs $ord(0,n)$, where the subroutine *ord* is defined as follows:

```
ord(i, j: N)
  if j − i < 2
    return
  k ← split(i, i, j)
  if q < k
    ord(i, k)
  else
    ord(k + 1, j)
```

Here $split(p,i,j)$ takes as an input $0 \le i \le j \le n$ and $p \in [i,j)$. Denote $a = x[p]$. The subroutine *split* rearranges the elements of $x[i,j)$ and returns $k \in [i,j)$ such that all elements of $x[i,k)$ are at most a, $x[k] = a$, and all elements of $x(k,j)$ are at least a. The operation *split* performs exactly $j-i-1$ comparisons. Compute the average number of comparisons performed by this algorithm under the assumption that each of the possible orderings of x is equally likely to occur.

6. Input data: $n\colon \mathbf{N}$, $a\colon \mathbf{R}[n][2][2]$. Output data: $A\colon \mathbf{R}$, where A is the total area of

$$\bigcup_{i\in[0,n)} \{(x,y) \mid a[i][0][0] \le x \le a[i][1][0] \wedge a[i][0][1] \le y \le a[i][1][1]\}.$$

In other words, you must compute the area of union of solid squares with coordinates a. Worst-case running time $O(n \log n)$.

7. Input data: $n\colon \mathbf{N}$, $x\colon \mathbf{B}[n]$ (as usual, $\mathbf{B} = \{0,1\}$). Output data: a compressed suffix trie for x. Recall that a trie is the data structure used in the Aho–Corasick algorithm for multiple string matching. A *suffix trie* stores all suffixes of x. For convenience, assume that x ends in a character \$ that does not occur elsewhere in x. If $x =$ banana\$, then the suffix trie will contain the following strings: banana\$, anana\$, nana\$, ana\$, na\$, a\$, \$. Furthermore, we require that the trie is compressed: each node is either a leaf or has at least two children. (If a node has just one child, then the labels on the incoming and outgoing edge of this node can be concatenated and the node itself can be deleted.)

- Prove that the total number of nodes in the trie is at most $2n$.
- Explain how to compute the total number of different substrings in x in $O(n)$ worst-case running time using a suffix trie for x.
- Modify the Aho–Corasick algorithm to compute the suffix trie in $O(n)$ worst-case running time. Hint: the unmodified Aho–Corasick algorithm needs about n operations to insert each suffix, which yields n^2 operations in total. One can avoid rescanning the whole new suffix by jumping directly to some interior node of the tree that is known to occur in the scanning process. To this end, the following "suffix link" function may be useful: given a node p in the trie, which encodes some string $s = at$, where a is the first character of s, the suffix link of the node p points to the node q that encodes the string t. Another useful observation: if we know in advance that a certain string is contained in a trie, then we can find the corresponding node by only inspecting the first letter of each label (other letters are guaranteed to match), which gives running time proportional to the number of nodes inspected, not to the number of letters in the string.

8. Consider the following algorithm. Input data: $p\colon \mathbf{N}$, p is an odd prime, $s\colon \mathbf{F}_p$. Output data: $x\colon \mathbf{F}_p$ such that $x^2 = s$, if such an x exists at all. Algorithm: choose a random element $a \in \mathbf{F}_p^\times$ and terminate if $a^2 - s \notin (\mathbf{F}_p^\times)^2$, otherwise keep choosing a new random element until found. Then compute $x = (a + \sqrt{a^2 - s})^{(p+1)/2}$ in the field $\mathbf{F}_p(\sqrt{a^2 - s}) = \mathbf{F}_p[y]/(y^2 - (a^2 - s))$, i.e., the splitting field of the polynomial $y^2 - (a^2 - s)$. If $x \in \mathbf{F}_p$, then x is the answer, otherwise s has no square root.

- Prove that this algorithm is correct. Make sure to prove that if $x \in \mathbf{F}_p$, then $x^2 = s$ and that if $x \notin \mathbf{F}_p$, then s has no square root. Also make sure to prove (with details of probabilistic arguments) that the algorithm terminates with probability 1.
- How would you implement this algorithm? More precisely: (1) How would you verify that an element of $\mathbf{F}_p$ is not a square? (2) How would you perform the computation of x in the splitting field?
- Compute the average number of choices of a before the algorithm terminates.