Algorithms and Data Structures
Spring 2018
Exercises for Units 1 and 2

1. This sequence of exercises is supposed to illustrate that certain restrictions that we put on our RAM model are really necessary. If they were not imposed then some problems that are believed to be very difficult could be solved easily. In particular you are to show that the problem of factoring a large integer into its prime components becomes quite easy, if no restrictions on the sizes of integers stored in a RAM are made (or if you allow a floor function for a RAM with real numbers). Many methods in cryptography, e.g. the security of the RSA crypto system, rely on the assumption that factoring is hard.

   The following sub-problems should lead to this result. The underlying model is an integer RAM with no size restriction and unit cost operations $+,-,*,$div.

   (a) Show that given integers A and N the number A^N can be computed in $O(\log N)$ time.

   (b) Show that given natural numbers N and K the binomial coefficient $\binom{N}{K}$ can be compute in $O(\log N)$ time.
   Hint: Consider $(A+1)^N$ *for large* A.

   (c) Show that given natural number N the number $N!$ can be computed in $O(\log^2 N)$ time.

   (d) Show that in $O(\log^2 N)$ time it can be tested whether N is a prime number.

   (e) Show that in $O(\log^3 N)$ time a non-trivial factor of N can be found, provided N is not prime. For this you may assume the existence of a routine that computes the GCD (Greatest Common Divisor) of two numbers X and Y in time $O(\log(\min\{X,Y\}))$.

   (f) Show that the prime factorization of N can be found in time $O(\log^4 N)$.

2. The following algorithm works on an array $A[1..n]$ that contains n integer numbers. Analyze the algorithm for its worst case behaviour and also analyze it for its expected behaviour under the probabilistic assumption that array $A[1..n]$ contains the integers from 1 to n as a random permutation with each permutation equally likely. In particular consider the following questions for these analyses:

   (a) How often is line 7 executed?

   (b) What is the overall running time?

```
1: for i = 2 to n do
2:   if A[i] < A[1] then
3:     x = A[i]
4:     for j = i downto 2 do
5:       A[j] = A[j − 1]
6:     end for
7:     A[1] = x
8:   end if
9: end for
```

Algorithms and Data Structures
Spring 2018
Exercises for Units 1 and 2

3. In the programming languages C and C++ an expression of the form $(a < b)$ returns the value 1 if a is indeed less than b and evaluates to 0 otherwise. Expressions involving other comparison operators, such as $(a >= b)$ have analogous semantics.

   Consider the following somewhat unusual way of rearranging the values in integer array $A[1..n]$ so that the small entries with value less than "pivot" x end up in part $A[1..j]$ whereas the large entries with value $\geq x$ end up in $A[j+1..n]$.

   1: $i = 1;\ j = n$
   2: **repeat**
   3: swap$(A[i], A[j])$
   4: $s = (A[i] < x);\quad t = (A[j] >= x)$
   5: $i = i + s;\quad j = j - t$
   6: **until** $j < i$

   Consider the number of swaps in line 3.

   (a) Does this algorithm indeed partition the array as advertised?

   (b) How many swaps can there be in the worst case and under what circumstances does this worst case happen?

   (c) Assume that every entry of $A[]$ has (independently of the others) equal probability of being smaller than pivot x and of being not smaller than x. What is the expected number of swaps performed by the algorithm?

   (d) You are invited to compare implementations of this strange partition procedure and of the traditional one as discussed in classe (or as given for instance in the Quicksort section of Cormen et al.). On my laptop this strange partition procedure is faster than the traditional procedure, inspite of the superfluous swaps. Why would this be the case?