1. a) $A^1 = A$

$A^N = (A^{\lfloor N/2 \rfloor})^2$ if N even and ≥ 2

$A^N = (A^{\lfloor N/2 \rfloor})^2 \cdot A$ if N odd and ≥ 2

$\Rightarrow T(N) \leq T(\lfloor \frac{N}{2} \rfloor) + 2$

↓ multiplying $A^{\lfloor N/2 \rfloor}$, $A^{\lfloor N/2 \rfloor}$ and then multiplying by A

- we can half the problem at most $\log N$ times

$\Rightarrow$ in each step we do a constant number of operations and we have $\log N$ steps at most $\rightsquigarrow O(\log N)$

b) $\binom{N}{k} \leq 2^N$ so it can be written with N bits

$A := 2^N \Rightarrow (A+1)^N = \sum_{0 \leq k \leq N} \binom{N}{k} 2^{Nk}$

$(A+1)^N = \underbrace{\binom{N}{0}}_{} \underbrace{\binom{N}{1}}_{} \ldots \underbrace{\binom{N}{k}}_{} \ldots \underbrace{\binom{N}{N}}_{}$

$N+1$ blocks of N bits each

- want to "extract" the k-th block:

$\lfloor (A+1)^N : 2^{Nk} \rfloor = \lfloor \sum_{0 \leq R \leq N} \binom{N}{R} 2^{NR} : 2^{Nk} \rfloor$ → when $R < k$, these elements are lost in the rounding down (get smth. < 1 when dividing)

$= \binom{N}{k} + \binom{N}{k+1} 2^N + \binom{N}{k+2} \cdot 2^{2N} + \ldots + \binom{N}{N} 2^{N \cdot N}$

$\lfloor (A+1)^N : 2^{Nk} \rfloor \bmod 2^N = \binom{N}{k}$

$\Rightarrow T(N) = O(\log N) + 2$

↳ $\lfloor : 2^{Nk} \rfloor$, $\bmod 2^N$

c) $1! = 1$

$N! = \binom{N}{\frac{N}{2}} \left(\frac{N}{2}\right)! \left(\frac{N}{2}\right)!$ if N even and ≥ 2

$N! = \binom{N}{\frac{N-1}{2}} \left(\frac{N-1}{2}\right)! \left(\frac{N-1}{2}\right)! \left(\frac{N+1}{2}\right)$ if N odd and ≥ 2

$T(N) \leq T(\lfloor \frac{N}{2} \rfloor) + O(\log N) + O(1)$

↳ multiply together

$\Rightarrow$ we can half our problem at most $\log N$ times and at each level we need $O(\log N)$ time

$\Rightarrow T(N) = O(\log^2 N)$

d) need: N prime $\Leftrightarrow N \nmid (N-1)!$

$[\Rightarrow]$ if N is prime, then it obviously does not divide $(N-1)!$ because $N > 1, 2, \ldots, N-1$ and cannot be split into smaller factors that could divide some numbers from $1, 2, \ldots, N-1$

$[\Leftarrow]$ we prove N not prime $\Rightarrow N \mid (N-1)!$

(equivalent to $N \nmid (N-1)! \Rightarrow N$ prime)

assume 1) $N = \prod P_i^{e_i}$ and there is more than one prime factor $\Rightarrow P_i^{e_i} < N \Rightarrow$ contained in $(N-1)!$ $\forall i$

$\Rightarrow (N-1)!$ is a multiple of N ✓

2) $N = P^e \Rightarrow P, P^2, \ldots, P^{e-1} < N \Rightarrow$ all of them are contained in $(N-1)! \Rightarrow P \cdot P^2 \cdot \ldots \cdot P^{e-1}$ is contained in $(N-1)! \Rightarrow P^e = N$ contained in $(N-1)!$ ✓

test whether N is prime:

- just check whether N divides $(N-1)!$, for instance whether $\lfloor (N-1)! : N \rfloor \cdot N = (N-1)!$
- for this, we need to compute $(N-1)!$ and the rest is done in constant time $\Rightarrow T(N) = O(\log^2 N)$

e) If $N \mid Q!$, then $N \mid R!$ for any $R \geq Q$ (we are just adding more factors) and $N \nmid 1!$

↝ find the smallest K s.t. $N \mid K!$ and $N \nmid (K-1)!$ using binary search
- $GCD(K, N)$ is a factor of N

runtime: $T(N) \leq O(\log N) \cdot O(\log^2 N) + O(\log N)$

($\log N$: binary search; $\log^2 N$: each time computing a factorial; $\log N$: GCD)

$$= O(\log^3 N)$$

f) We find a divisor K like in e) and then do factorizations of K and N/K.

$$T(N) \leq \underbrace{T(N/K)}_{\text{one subproblem}} + \underbrace{T(K)}_{\text{other}} + \underbrace{O(\log^3 N)}_{\text{time needed to divide into these two subproblems}}$$

- we can divide our problem in this way for at most $\log N$ times (K always ≥ 2) and for each division need $O(\log^3 N) \Rightarrow T(N) = O(\log^4 N)$

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

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
```

   The program maintains the following invariant:

   - Whenever line 1 is executed the current $A[1..i-1]$ is a permutation of the initial $A[1..i-1]$ that has the minimum element in $A[1]$

   This means that the yes-branch of the if statement starting in line 2 is executed if $A[i]$ is the smallest of the elements $A[1..i]$ (and this branch shifts the contents of $A[1..i]$ in a circular way so that the smallest element, i.e. $A[i]$, which is x, is stored in $A[1]$ – this is exactly line 7).

   (a) In the worst case, when $A[1..n]$ is sorted in decreasing order, line 7 is executed for every i, which means $n-1$ times.

   For the expected case, let M_i be a 0-1 random variable, indicating whether line 7 is executed in the loop interation for i. We know that $M_i = 1$ iff $A[i]$ is the smallest among the elements in $A[1..i]$. By the random permutation assumption this happens with probability $1/i$, and hence the expected value of M_i is $1/i$. Therefore the expected number of times that line 7 is executed is

$$\sum_{1<i\leq n} \mathrm{E}x[M_i] = \sum_{1<i\leq n} 1/i = H_{n-1} \approx \ln n\,.$$

   (b) Since the circular shift takes $O(i)$ time, in the worst case you can incur $\sum_{1<i\leq n} O(i) = O(n^2)$ time.

   For the expected case note that the work to be done during the iteration i of the loop, is either $O(i)$ if a new minimum was found, or $O(1)$ otherwise. Let D_i denote the expected work of iteration i. We have

$$D_i = \mathrm{P}r(M_i = 1)\cdot O(i) + \mathrm{P}r(M_i = 0)\cdot O(1) = (1/i)\cdot O(i) + (1-1/i)\cdot O(1) = O(1)\,.$$

   Thus the expected work over all iterations is $\sum_{1<i\leq n} O(1) = O(n)$.

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?

   Let us call a number *small* iff it is less than x and *large* otherwise.

   (a) The algorithm maintains the following invariant:

   - whenever the repeat loop is started all numbers in $A[1..i-1]$ are small and all numbers in $A[j+1..n]$ are large.

   Since the program does never decreases i nor increases j, if the program terminates the invariant implies that it delivers the desired result.

   (b) Let us count the number of times an element is moved. Since only swaps move elements (and one swap moves two elements) the total number of swaps will be half the total number of moves.

   $A[i]$ will me moved in line 3. If it is large, j will be decreased and it will not be moved again, otherwise it will be moved again and i will be increased. Symmetric statements can be made for $A[j]$. This means an element of $A[]$ can be moved at most twice. This happens to all elements if the first $n/2$ ones of $A[]$ are all small, and the other ones are all large. The total number of moves in this case is $2n$ and the total number of swaps is n.

   (c) Again counting moves we see that an element is moved once with probability $1/2$ and moved twice with probability $1/2$. This means its expected nubmer of moves is $3/2$ Summing over all elements we get $3n/2$ expected moves and hence $3n/4$ expected swaps.

   (d) Such an implementation turns out to be advantageous on processors that do branch prediction and speculative execution, since this implementation avoids branches.