Algorithmic Topology and Groups
Exercises #1

1. Recall that a random access machine (RAM) is composed of an array R of registers indexed by $\mathbb{N}$, an accumulator α to perform arithmetic operations, a program counter κ and a list of numbered instructions among the followings

Instruction	**Semantics**	
LOAD j	$\alpha := R[j]$	$\kappa := \kappa + 1$
iLOAD j	$\alpha := R[R[j]]$	$\kappa := \kappa + 1$
STORE j	$R[j] := \alpha$	$\kappa := \kappa + 1$
iSTORE j	$R[R[j]] := \alpha$	$\kappa := \kappa + 1$
WRITE x	$\alpha := x$	$\kappa := \kappa + 1$
ADD j	$\alpha := \alpha + R[j]$	$\kappa := \kappa + 1$
SUB j	$\alpha := \alpha - R[j]$	$\kappa := \kappa + 1$
HALF	$\alpha := \lfloor \alpha/2 \rfloor$	$\kappa := \kappa + 1$
JZERO i		if $(\alpha = 0)$ then $\kappa := i$ else $\kappa := \kappa + 1$
JPOS i		if $(\alpha > 0)$ then $\kappa := i$ else $\kappa := \kappa + 1$
HALT		Stop the program

   - Describe a RAM that performs the multiplication of two input nonnegative integers given in the first two registers $R[0]$ and $R[1]$. The machine should halt with the resulting product written in $R[2]$.
   - What is the complexity of your algorithm? Here by complexity, we mean a function f such that the number of executed instructions before the RAM halts is bounded by $f(n)$ where n is the total number of bits in the binary encoding of the input integers.
   - What would be the complexity of your algorithm if it was to be implemented on a multitape Turing machine. Here, the complexity is expressed in terms of the number of transitions of the Turing machine before it stops.

2. Karatsuba 1960's algorithm for integer multiplication relies on the following simple formula. If x_0, x_1, y_0, y_1 are four digits in base B, then

$$(x_0 + B\,x_1)(y_0 + B\,y_1) = u + B\,v + B^2\,w \qquad \text{with}$$
$$u = x_0 y_0, \quad w = x_1 y_1, \quad v = (x_0 - y_1)(x_1 - y_0) + u + w$$

   Note that multiplication by B amounts to a left shift and a zero padding in the base B expansion of the number. The multiplication of the 2-digit numbers '$x_1 x_0$' and '$y_1 y_0$' thus requires 3 multiplications of 1-digit numbers, 3 shifts and 6 addition or subtractions. Shifts and additions or subtraction can easily be implemented to run in linear time on a (multitape) Turing machine.

Assuming that B is a power of 2, the above formula can be applied recursively until $B = 2$. Give a recursive formula for the time complexity of multiplication on a Turing machine based on this recursive algorithm.

There are more efficient algorithms for integer multiplication. The Schönhage–Strassen algorithm (1971) is based on the Fast Fourier Transform (FFT) to evaluate the product of two polynomials. Indeed, viewing the coefficients of a polynomial as a function over the naturals, the coefficients of the product becomes the convolution of the coefficient functions. The Fourier transform of the convolution is thus the component-wise product of the Fourier transforms. The Schönhage–Strassen algorithm runs in $O(n \cdot \log n \cdot \log\log n)$ time. Very recently (2019), Harvey and van der Hoeven announced a $O(n \log n)$ time algorithm for integer multiplication. This is the fastest known algorithm up to date.

Try to read and understand a full description of Schönhage–Strassen algorithm in order to explain it in class.

3. Show that **PSPACE** $\subseteq$ **EXP**.

4. Show that **NP** $\subseteq$ **PSPACE**.

5. Show that every non-trivial problem (proper subset of $\mathscr{A}^*$) in **P** is **P**-complete.

6. Suppose we are given a list of n elements from a totally ordered set and the list is stored in n consecutive cells of a linear array. The sorting problem is to rearrange the elements so that they appear in nondecreasing order in the array. Here, we count the complexity of a sorting algorithm as the maximal number of comparisons needed to sort any array of length n. Propose and analyse different sorting algorithms. Prove a lower bound on the complexity of any sorting algorithm.

7. Prove that the breath-first search algorithm as described in the course indeed computes the shortest path distance to the source vertex.

8. Suppose we are given a connected graph $G = (V, E)$ with a non-negative weight function $w : E \to \mathbb{R}_+$. A shortest path tree with source $s \in V$ is a spanning tree of G such that for every $v \in V$ the sum of the weights of the edges of the path from v to s in the tree is minimal among all paths in G with the same endpoints. When the weight function is constant, the previous exercise claims that breath-first search computes a shortest path tree. For non-constant weights, the famous Dijsktra's algorithm (1956) efficiently computes a shortest path tree. To describe the algorithm we assume that there exists an efficient data-structure, called a *heap*, to store items with associated key numbers allowing the following operations:

   - inserting a new item into the heap,
   - decreasing by any amount the key of an item present in the heap,
   - extracting (*i.e.*, finding and removing) an item with minimum key in the heap.

The *Fibonacci heap* is a well-known implementation that supports the above operations in constant time for inserting an item and decreasing a key, and in $O(\log n)$ time for extracting an item with minimum key, where n is the number of items in the heap at the time of extraction. For the Dijsktra's algorithm every vertex $v \in V$ is associated with three fields:

- a distance d_v intended to be an overestimation of the shortest path length from v to s,
- a connecting edge e_v intended to connect a parent of a vertex belonging to a shortest path from v to s,
- a binary status marked/unmarked set to marked if v is in the heap.

As usual, G is given as input by its adjacency lists.

Algorithm 1 Dijsktra's algorithm

```
1: Set d_s = 0 and d_v := +∞ for all v ∈ V \ {s} // Initialization
2: Insert all vertices in V in an initially empty heap
3: Mark all the vertices
4: while the heap is nonempty do
5:    extract v with minimum distance in the heap
6:    unmark v
7:    for Each edge e = uv incident with v do
8:       if d_u > d_v + w(e) then
9:          d_u := d_v + w(e) and e_u := e
10:         if u is marked (in the heap) then
11:            decrease d_u accordingly in the heap
12:         end if
13:      end if
14:   end for
15: end while
```

Prove that Algorithm 1 indeed computes a shortest path tree.

What is the complexity of this algorithm?