Answers and comments to DA3018 exam on 2020-05-25

1. (a) Here is an attempt:

```
def find_indices(A):
    S = ArrayList()
    S.append(0)
    for i from 1 to len(A):
        if A[i] == 0:
            S.append(i+1)
    return S
```

This helper function iterates through the elements of A. In each iteration, there is an if-statement taking $O(1)$ time and a possible addition of an element to the ArrayList which can be assumed to also take $O(1)$ time (on average). Since the preparations of creating an ArrayList takes constant time, the helper function's time complexity is $O(n)$.

(b) Mergesort on m elements takes $O(m \log m)$ time if comparisons take $O(1)$ time. We work with string comparisons in this assignment, and although we know there are m strings, there is no obvious way of using n to limit their length. So a worst-case analysis suggests that the string comparisons take $O(n)$ time. The yields a total time complexity of $O(mn \log m)$.

2. (a) Source vertices in a directed graph are recognised by having indegree 0. So we can start by identifying those.

Then we start an adapted DFS that collects all sink-vertices that are found from a starting sink and returns them.

Here is suggested pseudocode.

```
def find_sources_and_sinks(G):
    sources = find_sources(G)
    results = empty_list()
    for v in sources:
        sinks = find_sinks(v, G)
        results.append( (v, sinks) )
    return results

def find_sources(G):
    sources = empty_list()
    for v in G.vertices():
        if indegree(v) == 0:
            source.append(v)
    return sources

def find_sinks(v, G):
    for w in G:
        w.visited = False
    sinks = empty_set()
    find_sink_helper(v, sinks)
    return sinks
```

```
def find_sink_helper(v, sinks):
    v.visited = True
    if outdegree(v) == 0:
        sinks.append(v)
    else:
        for u in neighbors(v):
            if not u.visited:
                find_sink_help(u, sinks)
```

This algorithm is correct because for each source, we use DFS to exhaustively search the graph for sinks, and the sinks are collected in a set which is created empty for each source vertex.

(b) Identifying the source vertices goes in $O(|V|)$ time. Then there is an adapted DFS and because the added operations are constant time, that function is $O(|V| + |E|)$. The DFS is applied for each source, and there are in the worst case $O(|V|)$ sources. Consequently, the algorithm's time complexity is $O(|V|^2 + |V| \times |E|)$.

3. **Phone numbers lookup**. Please note that there is not One True Answer to this question. You do not need to make the same assumptions as I do, but they should be reasonable and based on what we discuss in the course and common computer knowledge.

   I will allocate 15 bytes for a phone number. Longer numbers, if they exist, are ignored. Names are given 85 bytes for storage. Longer names are truncated. This means that a phone record takes 100 bytes.

   Memory usage analysis is probably easier if I use open addressing for my hash table, because I can simply give 2 GB of memory to the hash table and do not need to worry about dynamic allocation of linked lists for chaining. There will then be 2 GB divided by 100 bytes per slot in the table results in 2×10^7 slots.

   To ensure an efficient system, I allow for a 75% load factor. Let n be the max number of records in my system, so $n = 0.75 \times 2 \times 10^7 = 15 \times 10^6$ is the maximum number of phone records that my system allows without bad performance. That might just barely work for the Swedish population.

   How many lookups per second can we hope for on this server? Say that there are 10^8 operations per second that we can achieve. Each hash-function to compute is on a 15-digit phonenumber or shorter. That phone number can be converted to a 15-digit integer, which in turn fits in a 64-bit integer (because $\log_2(10^{16}) < 54$) and that means efficient arithmetic. I pick a very simple hash function: $h(x) = (x \bmod p) \bmod m$, for some prime $p4$, $m << p << 10^{16}$, which can be computed in regular computer arithmetic using two modulo-operations. This means that the bottleneck for the hash function is the conversion from a phonenumber string to an integer. Since there are 15 digits, the conversion should be possible to do in some tens of CPU-instructions and memory accesses. A rough estimate is that 100 CPU-instructions are needed for hashing and memory access. (Channel your inner physicist: 10 instructions are too few, and 1000 instructions does not seem warranted if the conversion needs 10s of instructions). So

we should be able to compute roughly $10^8/100 = 10^6$ hash functions per second.

Summary: I estimate that my phonenumber lookup service can store 15 million records and serve up one million requests per second.

4. (a) **Quicksort**. I choose to pick the middle element of A as a pivot element in the partition function. Advantages with that is that it is trivial and fast to choose, and also results in a good partitioning if the input is sorted. The obvious disavantage is that it is an arbitrary choice, so there is no guarantee that we get good partitioning. In fact, one can construct input that forces $O(n^2)$ time complexity with this partitioning scheme.

   In the following, $<$ represents $\prec$.

```
def partition(A, left, right):
    mid = (left + right) div 2  integer division.
    pivot = A[mid]
    bottom = left
    top = right
    while bottom < top:
        while A[bottom] <= pivot:
            bottom++
        while A[top] > pivot:
            top--
        if bottom < top:
            swap(A, bottom, top)
    return top               // The last element of
     the first partition
```

   With a "fresh" array A, partition is called as `partition(A, 0, len(A)-1)`. At every iteration, we locate a pair of elements that are larger and smaller (by $\prec$) than the pivot element. Since the search for a large element starts from `left` and increases, while the search for a small element starts at `right` and decreases, we identify pairs to swap, as long as they are indeed in the "wrong" order (bottom $<$ top).

   In each iteration of the outer **while** loop, we either move at least one of `bottom` and `top`, or swap two elements. After a swap, the `bottom` and `top` will "move" in the next iteration. Hence, at most $O(n)$ swaps will be made, so the time complexity of partition is $O(n)$.

   (b) My partition implementation returns the position with the last element that is $\prec$ or $=$ to the pivot element.

```
def quicksort(A, left, right):
    if left < right:
        mid = partition(A, left, right)
        quicksort(A, left, mid)
        quicksort(A, mid+1, right)
```

   To sort, this function should be called as `quicksort(A,0,len(A)-1)`

   (c) **Visualisation of Quicksort**.

The input array A is

HIF	IFKN	OFK	KFF	IKS	AIK	VBoIS	BKH	DIF	FFF	HIF	IFKG	MAIF	IFE	OSK	MFF

and `quicksort(A,0,15)` will be called.

Then, a call to `partition(A, 0, 15)` will be made. The pivot element is chosen as BKH and the call will change A to

BKH	**AIK**	OFK	KFF	IKS	**IFKN**	VBoIS	**HIF**	DIF	FFF	HIF	IFKG	MAIF	IFE	OSK	MFF

and `top` $= 1$ is returned and assigned to `mid`. Two recursive calls in quicksort follows:

- `quicksort(A, 0, 1)`
- `quicksort(A, 2, 15)`

I will follow the second call, which starts with a call to `partition (A, 2, 15)`. It chooses FFF as a pivot element, changes the array to

BKH	AIK	**FFF**	**DIF**	IKS	IFKN	VBoIS	HIF	**KFF**	**OFK**	HIF	IFKG	MAIF	IFE	OSK	MFF

and returns 3 as the last element of the first partition.

Two recursive calls in quicksort follows:

- `quicksort(A, 2, 3)`
- `quicksort(A, 4, 15)`

I will follow the second call, which starts with a call to `partition (A, 4, 15)`. It chooses OFK as a pivot element, changes the array to

BKH	AIK	FFF	DIF	IKS	IFKN	**MFF**	HIF	KFF	OFK	HIF	IFKG	MAIF	IFE	OSK	**VBoIS**

The index 13 is returned and quicksort continues recursively into

- `quicksort(A, 4, 13)`
- `quicksort(A, 14, 15)`

We can notice that the array is gaining order, even though the partitions have not been evenly made at all.

5. A suggested formulation is as follows.

   We assume there is a map and an associated list of coordinates with marked map points, possible also known as good checkpoints. We have a function d that computes the Euclidian distance on the map between points. We also assume that is the distance the club refers to, and not a running distance in the terrain.

   The orienteering problem:

   - Input: a list L of coordinates, an integer n, and a starting checkpoint position $v \in L$.
   - Output: a subset $L' \subset L$ of size n such that $\forall u, v \in L' : d(u, v) \leq 500$, and $\sum_{u,v \in L'} d(u, v)$ is maximised.

   In this solution,

   - we use the positions given by the orienteering club,
   - pick a subset of them,
   - ensure they are not too far apart, while also

- spreading the points out by maximising the sum of pairwise distances.

Since a starting checkpoint is part of the input, we can make sure that there is some variation in the courses that are output. Hence, we have fullfilled the wishes from the orienteering club.

Note that there are many possible solutions to the problem.

6. (a) The given grammar is not LL(k), because you cannot consider a production and know which one to apply. For example, if the next terminal to parse is VERTEX, then you cannot know (for any fixed k terminals you look ahead at) decide whether to use the `VERTEX "-"one_or_more_edges` production or the `VERTEX "-"VERTEX ";"` production.

   (b) A possible solution is given in Figure 1. Several rules have been rewritten such that productions start with a distinct terminal. For example, the `"}"` terminal is moved from being last in the starting production, `graph`, to being a possible first terminal in `arc_list` and `edge_list`.

```
graph = "digraph" graph_name "{" arc_list ; (* directed *)
      | "graph"   graph_name "{" edge_list; (* undirected *)
      ;

arc_list = "}"  (* the empty list *)
         | arc_list one_or_more_arcs
         ;

one_or_more_arcs =  VERTEX "->" VERTEX more_arcs ;

more_arcs = ";"
          | "->" VERTEX more_arcs
          ;

one_or_more_edges = VERTEX "-" VERTEX more_edges ;

more_edges = ";"
           | "-" VERTEX more_edges
           ;
```

Figur 1: Suggested solution for 6b. Commenst are written as `(* comment here *)`.