Suggested answers and comments for the 2019-06-05 exam for DA3018

1. (a) Asymptotic complexity describe resource usage, for example time and space (memory), as a function of input size. We consider the asymptotic behaviour to simplify analysis and focus on the dominating terms. I.e., we are mostly interested in how resource usage grows and that is why complexity is expressed in Big-Oh notation.

   (b) A stack is a data structure with the standard operations PUSH (to insert an element), POP (remove and return an element), and IS_EMPTY (testing whether there are elements in the datastructure. The element most recently inserted using PUSH is the element removed with POP.

   (c) A topological sort means that you compute an order of the vertices (nodes) in a graph that respects the order given by the edges.

   (d) You traverse the vertices reachable from a start vertex v in an order such that the neighbors of v are passed before any other vertex is considered. Then vertices at distanve 2 from v, and so on.

2. (a) Sample implementation in Python:

```
def n_single_kids(t):
    if type(t) != tuple:        # Child
        return 0
    else:        # Internal node
        if len(t) == 1:
            return 1 + n_single_kids(t[0])
        elif len(t) == 2:
            left = t[0]
            right = t[1]
            return n_single_kids(left) +
                n_single_kids(right)
        else:
            raise Exception('Not␣a␣binary␣tree')
```

   We assume in this solution that trees are given as 1 or 2 tuples (containing 1 or 2 elements). The tree $(1,)$ is a tree with two vertices (one leaf) and `n_single_kids` should return 1. The function should also return 1 for the tree $((1,),(3,4))$.

   (b) The algorithm is correct because (1) its base case (a leaf) is correct, and (2) otherwise we will count 1 plus the number of single-children of the subtree in the single-child case, or for the two-child case we add up the single-children of the two subtrees.

   (c) The algorithm will traverse the vertices recursively. For each vertex, there are $O(1)$ work, so if the tree has n vertices, the time complexity will be $O(n)$.

3. (a) The algorithm is *not* suitable, because the problem formulation asks for all positions where p is found, but the algorithm returns only the first position of p in t.

(b) Let m be the length of t and n the length of p. The algorithm has a loop over the indices of t (with respect to p, assumed to be shorter) and within that an loop over indices in p. There is constant work done within the two loops. All in all, the time complexity is $O(mn)$.

4. (a) Our suffix array A contains all positions of suffices in S in an order that is a correct alphabetical sort of. One should use binary search in A.

   Here is an example implementation in Python:

```
def find_sa(t, A, p):
    left = 0
    right = len(t)
    while left < right:
        mid = (right + left) // 2
        pos = A[mid]
        if t[pos : pos + len(p)] == p:
            return A[mid]
        elif t[pos: pos + len(p)] < p:
            left = mid + 1
        else:
            right = mid
    return None
```

   Full credits are given for far simpler algorithm presentations than my Python code.

   To try the code, look at the standard suffix array example "banana" and its suffix array $A = [5, 3, 1, 0, 4, 2]$.

   A feature with suffix arrays is that you get easy access to the interval in A where a substring is found, and that interval then contains all positions in t for the substring.

   (b) Suppose t has length n. Binary search in A will then take $O(\log n)$ iterations. There are a limited number of string comparisons with p, each taking $O(m)$ time if p has length m, so the total time complexity is $O(m \log n)$.

5. We can represent the streets of Old Town with a weighted graph. Let intersections and addresses (and sightseeing positions) be vertices and the street segments in between are edges with the distance of the street segment as the weight of the edge.

   A street is then a path in the graph and the length of the street is the sum of the weights of the edges in the path.

   We can assume that the tourist business the friend works at has a position v that all guide trips start from and return to.

   A positional computational problem is:

   - Input: Given a weighted graph G for Old Town, and a set of vertices W for the positions we want to guide to.
   - Output: a walk of minimal weight starting and ending in v.

6. A hash function.

(a) I choose to use a hash function of the form $h(x) = (ax+b \bmod p) \bmod s$, where p is a prime.

```
hash(adress, s):
   key = 0
   for c in adress.street:
      key += char_to_int(c)
   key += adress.street_number
   for c in adress.town:
      key += char_to_int(c)
   p = first_prime_number_after(s)
   return (11 * key + 17 mod p) mod s
```

The help function `char_to_int` returns the code for a character in the character encoding in use (like: ASCI, UTF-8, et.c.). I use `first_prime_number_after` to choose a suitable prime, and assume that wehave access to a table of primes. I accept many "fuzzy" solutions to subproblems of this kind.

(b) Since I do not have control over the table size s, it is good to have a hash function that does not need the table size as a modulus. The suggested hash function returns a value in the interval $[0, s-1]$ and therefore uses the whole table. Experience shows that elements are nicely distributed using this hash function. Note that all address characters are used by the hash function, to reduce the risk of unnecessary collisions: two distinct addresses are expected to hash to different results.

7. BST-sortering

(a) One should use *inorder* traversal, since you then first get the vertices under the left subtree under a vertex v, then v, and finally the vertices in the right subtree under v. The main property if a binary search tree is that the key for all vertices in the subtree are smaller than v and the elements of the rigth subtree are larger than v, so the elements will then be accessed in sorted order.

(b) The algorithm consists of two parts: insertion and traversal. The latter part means that we visit each vertex in the tree and regardless of tree that takes linear time. Insertion is done in a loop over n elements. In every iteration an element is inserted into the BST and since it is not balanced the worst case (e.g., already sorted elements of a) the tree looks like a path and an insert takes $O(n)$ time. Altogether that is $O(n^2)$ time.

(c) Stability of sorting depends on the BST insert algorithm for identical elements. If you insert same elements into the left subtree, the last inserted element must be put above elements with the same key, and if you put them to the right, elements should be put below previous same-key elements.

8. Graph representation and DFS.

(a) If the average vertex degree is α, regardless of the number of vertices n, the number of edges in the graph are αn. Larger graphs are then more sparse than small graphs. This seems like a case for adjacency lists, since that should make for a memory-efficient representation. Using a adjacency matrix would have zero-elements grow as $O(n^2)$, which is inefficient for $O(n)$ edges.

(b) We have determined that we have n vertices and αn edges in the graphs we study. Simplifying the time complexity of a correctly implemented DFS, we get $O(|V| + |E|) = O(n + \alpha n) = O(n)$.

9. A possible example implementation:

```
def level_order(T):
   Q = empty_queue()
   Q.enqueue(root(T))
   while not Q.is_empty():
      v = Q.dequeue()
      print(v)
      if v.left:
         Q.enqueue(v.left)
      if v.right:
         Q.enqueue(v.right)
```

We disregard the empty tree.

The first vertex added to the queue is the root, which is first level. That vertex is also the first to be printed. Assume vertices up to level i have been printed correctly i level order. Their children, and other vertices, are in that case in Q, so when the loop continues the vertices at level $i+1$ are printed. This continues until Q is empty, which happens when all leaves are printed.

Every vertex will end up in Q and there are therefor $O(n)$ iterations for a tree with n vertices. In each iteration there are one or two queue operations, a print statement, and two if-statements, all in $O(1)$. So the total time complexity comes to $O(n)$.