Answers and comments to exam on 2017-06-02 in DA3018

The *preliminary* point scoring rules is indicated for some questions. We may adjust the scoring.

1. (a) In asymptotic time complexity analysis, one often does not care how many nanoseconds are actually needed for various operations, so the simplification that all basic operations are the taking "one unit of time" is natural.

   (b) Accessing an arbitrary element i in an array means computing the memory address as "starting address of array" plus "the size of elements" times i. That computation is constant-time arithmetic. Referencing the actual memory address is also constant time on normal computers.

   (c) Since it is hard to figure out the exact time an algorithm might need, we look at asymptotic time complexity to see how resource needs (here: time) scales as input grows. This can help us estimate what is feasible or not to compute.

   (d) The worst-case time complexity of QuickSort is $O(n^2)$ and occurs when pivot elements are badly chosen. If one, for example, chooses the first element in the input array as pivot element, then having sorted input will yield very unbalanced subproblems which causes the bad performance.

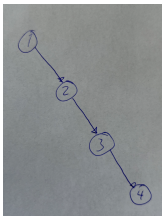
2. (a) Consider the following pseudocode:

```
def write_keys(T):
    if T:
        write_keys(T.left)
        print(T.key)
        write_keys(T.right)
```

   (b) On null input, nothing happens. When a tree node is given, then the left subtree is first processed, meaning the all the keys of that subtree are printed. Then, the current key is printed, before keys in the right subtree are printed using a second recursive call.

   (c) This is called *inorder traversal*.

   (d) A binary search tree is unbalanced if the height of subtrees differ substantially. A simple example is given on the right.
   A regular traversal, like the one implemented by `write_keys`, is not affected by balance, since every node will have to be traversed. Regardless of balance, the traversal takes $O(n)$ time, for n vertices.



3. (a) The breadth-first traversal can be illustrated as follows.

```
def bfs(G, start):
    for v in V(G):
```

```
        visited[v] = False

    queue = Queue()
    queue.enqueue(start)
    visited[start] = True

    while not queue.empty():
        v = queue.get_first()
        print(v)    # Show that we are traversing
            vertices!
        for w in G.neighbors(v):
            if not visited[w]:
                visited[w] = True
                queue.enqueue(w)
```

Scoring: deduct a point per fundamental mistake. We allow slight inefficiencies, but not asymptotic inefficiency.

(b) Here is the idea behind BFS:

- We keep track of which vertices we have visited using the array `visited`, which is initialized in the first loop.
- The starting vertex is initializing the queue, and then we start iterating over the contents of the queue.
- In each iteration, we extend the set of vertices to look at by adding the neighbors of the current vertex to the queue.
- Since we are using a queue, the vertices we work with (here: print them) are handled in the order they were found.

(c) The time complexity of $O(|V| + |E|)$, since we (1) iterate over all vertices and (2) by looking at neighbors actually look at each edge twice.

More carefully: The initial for-loop is looping over all the vertices doing constant-time operations, hence $O(|V|)$. We can then count the operations in the while loop by counting the number of times a line in the pseudocode might be executed. A vertex is added and removed from the queue at most once, so those operations are done at most $O(|V|)$ times. However, we are looking at the neighbors of each node and the number of "neighbor-relations" are exactly the number of edges. If $(v, w) \in E$, then that edge pops up both for v neighbors and w neighbors. It is reasonable to assume we implement the queue and visited-array such that operations on these take $O(1)$ time.

Scoring: 1p for observing cost of operations. 1p for noting complexity of first loop. 1p for counting edges in the second loop.

4. Say that the length of s and t is m and n, respectively.

(a) We can note that the internal helper function `at_position` takes time $O(m)$, because it loops over the length of s to verify that characters of s matches the substring of t starting at i. In each iteration, there are a constant number of arithmetic operations, assignments and single-character comparisons.

The main function, `is_substring`, iterates over possible starting points in t. Hence, there are $O(n)$ iterations containing some basic

constant-time operations and a call to `at_position`, yielding a total of $O(mn)$ operations.

Scoring: 1p for time complexity of `at_position`. 1p for main loop, and 1p for putting it together well.

(b) Looking for a substring in t can be seen as looking for the prefix of a suffix of t. Since A contains sorted suffices, we can do binary search over the suffices of t and check whether s is a prefix.

```
def is_substring(s, t, A):
    m = len(s)
    left = 0
    right = len(A)
    while left < right:
        mid = (right + left) // 2
        if s == t[A[mid] : A[mid] + m]:
            return True
        elif s < t[A[mid] : A[mid] + m]:
            right = mid
        elif s > t[A[mid] : A[mid] + m]:
            left = mid+1
    return False
```

Since this is a binary search, where the search interval is halved in each iteration, there will be at most $O(\lg n)$ iterations ($n = |A|$). In each iteration, there are a fixed number of arithmetic operations and assignments, plus the relatively expensive operation of comparing with s, which is $O(m)$. Hence, in the worst case there are $O(m \lg n)$ operations, which is significantly faster than $O(mn)$.

As a side note, string comparisons of this kind can be done in $O(m+n)$ time using several algorithms.

Scoring: 2p for correct algorithm which is faster than $O(mn)$, with additional 1p for explanation. 2p for time complexity.

5. One attempt to formulate a computational problem is here. There are more ways of doing it!

   The *Forest Thinning Problem* is defined as:

   - Input: a set of coordinates $C = \{(x_0, y_0), (x_1, y_1), \ldots\}$ and a minimum distance α.
   - Output: The largest subset $C' \subseteq C$ such that for any two points $p_i, p_j \in C'$, the distance from p_i to p_j is at least α and $|C'|$.

   Scoring: 2p for clearly stating input and output. 1p for stating an optimization problem. 1p for formalizing something like coordinates and 1p for noting a distance constraint.

6. (a) First note that there are five values to store for each observation in the data. It is given that the time stamp takes four bytes, and since it is reasonable that coordinates and intensity measurement are stored as 32-bit floats, we can assume $5 \times 4 = 20$ bytes per observation.

   Using an array to store these elements has no memory overhead at all: the datastructure needs 20×10^9 bytes, which is 20 GB and can

actually easily fit in relatively inexpensive computers (at least relative to Physics department expenses).

(b) Selection sort has quadratic time complexity, which is really bad on large datasets. In this case, quadratic time means on the order of 10^{18} operations and with a modern computer we can assume one operation per 10^{-9} seconds, so the sorting would take roughly 10^9 seconds which is something like 32 years.

(c) We need a sorting algorithm that runs in $O(n \lg n)$ time, because that would mean a running time like $Cn \lg n$. Since $\lg 10^9 = 9 \times \lg 10 \approx 9 \times 3.3 \approx 30$, running time would be measured in minutes rather than years.

An algorithm that runs *in place* avoids extra memory allocation. Heapsort is such an algorithm and would probably suit this problem very well. So zero extra memory.

(d) Instead of 20 bytes per element, we have to add to references to children. Depending on implementation, we might need memory for a reference to memory objects too, but one can avoid that and I consider this an unnecessary complication. In a memory-saving implementation, we would let the key (time stamp) and observation data be **`float`** and **`int`** attributes in the BST nodes. Therefore, the extra memory we should count is the memory for references. A memory address (the reference) is 8 bytes on a modern computer (but I will accept 4 bytes too, if clearly stated), so it is an additional 16×10^9 bytes, or an increase of $16/20 \approx 80\%$.

7. Create a vertex v_g for each gene g and w_i for each experiment i, giving vertices $V = v_g$ and $W = w_i$ such that $|V| = n$ and $|W| = m \leq n$. The edge set E contains edges for those vertices for which $x_{g,i} = 1$, i.e., the pairs g, i given in the input. Note that $|E| \leq cn$.

   If $x_{a,j} = x_{b,j} = 1$, then we wanted a and b to be in the same partition. In our graph $G = (V \cup W, E)$, those genes' vertices will be connected through edges (v_a, w_j) and (v_b, w_j). Transitively, if $x_{a,k} = 0$ and $x_{c,j} = 0$, a and c could be in the same partition if there is an experiment connecting c with b.

   A connected component in G would define the gene partitions we are looking for. If v_a and v_b are in the same component, then we now that there is one or more experiments connecting a and b. If v_a and v_b are in different components, then such experiments are missing and a and b should be in different partitions.

   Creating the graph, effectively reading input, takes time $O(cn) = O(n)$ if we use an adjacency list format. Connected components can be computed using DFS (which works nicely with adjacency lists) in time $O(|V|+|W|+|E|) = O(n + m + cn) = O(n)$. Translating components to gene sets is easily done in $O(n)$ time. Altogether, we keep within the requested time complexity.

   I encountered a problem like this when looking at solutions for a research question a couple of weeks ago! Recognizing it as a graph problem, connected components, enabled me to immediately see that solutions could

be computed fast and made implementation easy (I could use an existing routine in SciPy).

Scoring: 2p for a good formalization, with justification for the computational objective. 2p for a fast algorithm. 2p for correct time complexity analysis.