Facit och kommentarer till tentamen 2017-06-02 i DA3018

1. (a) Good general sorting algorithms use $O(n \log n)$ comparisons, and other operations are proportional to the number of comparisons. Since the length of the strings is a parameter and can be substantially larger than n, we should take sequence length into account. Each string comparison can be performed in a loop over strings, yielding time complexity $O(L)$, assuming that we compare character by character and a character comparison is $O(1)$. In total, $O(Ln \log n)$.

   (b) Finding the smallest element in a list can be computed by introducing a variable for the smallest element and then looping through the list and compare each element against the current-lowest element. Looping through a list of n elements and performing a limited number of operaitions is $O(n)$. No sorting is needed.

   (c) Constructing a min-heap takes $O(n)$ time for n elements. Checking the smallest element in a min-heap means accessing the first element in an array, which is $O(1)$ time. In all, finding the minimum this way is $O(n)$. Compared to the previous method, this is probably slower because more work is done: the constant "hidden" in the $O()$ notation is larger.

   (d) The game Yahtzee is played in 15 rounds by rules in games in Swedish toy stores, and 13 rounds according to the US rules on Wikipedia. In each round there are at most three rolls with the dice per player. Since all parameters of the game is bounded by a constant, simulating Yahtzee has time complexity $O(1)$.

2. Note that there are n^2 elements to be inserted into M. By inserting them in different order, we get different matrices. There are $n^2!$ different permutations of a list of n^2 elements, so we can construct $n^2!$ different M.

   Checking whether a matrix is magic reduces to checking whether each row, column, and diagonal has the same sum. There are n rows, n columns, and 2 diagonals in M, and summing up n elements takes $O(n)$ time, so "checking for magic" takes $O(n^2)$ time.

   Altogether, the time complexity of the algorithm is $O(n^2 \cdot n^2!)$. This is a worst case analysis and we have ignored how many magic squares there are. We will not need to go through all permutations to find a magic square.

3. When you store a pair (k, v) in a hash table, the key k is used for two things, (i) computing the hash value which is the base for placing the pair in the table, and (ii) comparing elements during lookup to determine whether you have found your element or not. We always expect collissions in hash tables, so if one changes the key, the corresponding value will not be found.

4. (a) Since it is stated that most of the computation time is spent in the function f, I will ignore other computational costs. The heuristic needs $O(n \log n)$ calls, but that is an asymptotic statement. I will

assume that it is exactly $n \log n$ calls to f. (In my experience, that is a very common case.) Furthermore, given that most algorithms give time complexities with the logarithm in base 2, we assume that is the case here too.

Our savings will then be $1 - \frac{n \log_2 n}{n^{3/2}}$. For $n = 10^4$:

$$1 - 10^4 \log_2 10^4 / 10^6 \approx 87\% \tag{1}$$

(b) The drawback with heuristics is that they do not guarantee accuracy or performance, so whereas we save a lot of time using the heuristic, we migth not get the optimal/correct answer.

5. For example:

```
def count_n_reps(t):
    if is_leaf(t):
        return 1
    else:
        l = left_subtree(t)
        r = rigth_subtree(t)
        return 2 * count_n_reps(r) * count_n_reps(r)
```

This algorithm correctly notes that there is only one way to convert a single leaf to a string, and if there is a branching, then there are two ways to proceed (write left tree first, or right tree first, but we are only counting so this is represented by "2*") before choosing a way to convert the two subtrees. We will have to count how many times the subtrees can be converted.

6. This problem was unfortunately damaged by an indentation bug. The pseudocode I *wanted* to give you is:

```
frequencies(T):   // T is a list of strings
   f = new FrequencyTable() // Made-up datastructure
   for line in T:
       w1 = null
       words = line.split() // splits at non-letters
       for w2 in words:
           w2 = lowercase(w2)
           if w1:
               f.update(w1, w2)   // Increments pair w1
                   ,w2
           w1 = w2
   return f
```

The difference is that `w1 = w2` is always executed, and not conditionally on w1.

With the code above, this is my answer:

> The algorithmic bug is that the problem statement is about word-pair frequences in the *text*, while the pseudocode counts word-pair frequencies in *lines*. Hence, a word-pair that straddles a linebreak will not be counted. For example, applying to the algorithm to this paragraph would miss the pair ('frequencies', 'in').

Without a code correction the answer is that w1 is always null, so nothing happens.

7. I would pick Mergesort.

   - All mentioned algorithms have time complexity $O(n \log n)$, but for Quicksort is only expected. Even if choosing the median as pivot element, which is somewhat expensive to compute, you can end up with very uneven subproblems and $O(n^2)$ time complexity.
   - **Heapsort:** It is positive that Heapsort has a guaranteed time complexity, but we cannot easily implement Heapsort and keep the heap on disk. A call to heapify will need almost random (as in *arbitrary*) file access patterns.
   - Both **Mergesort** and **Quicksort** partition the input and we can make use of that when sorting a large file.
   - Both **Mergesort** and **Quicksort** are recursive algorithms with a base case of some size. We can choose to use a base-case sorting when the partitioned data fits the memory in the computer, and choose any fast and easy-to-use algorithm we want for the base case.
   - **Mergesort** has the advantage that we can decide beforehand how many and how large partitions the recursive algorithm will make, so one can make a single pass through the large input file and create "base-case files". The recursive portion of Mergesort is then managing in which order we will merge datafiles as well as file-by-file merging.
   - For **Quicksort** one needs to choose pivot elements. Regardless what principle we choose, it will mean a lot of rearranging the data from the input file, and it will *a priori* be difficult to choose pivot elements (even with sophisticated methods) such that the base-case files are appropriately large. However, if that is possible, Quicksort has the advantage that the final step would only consist of merging files together.

   It seems to me that Mergesort will incur less data access and will be easier to implement.

8. What we in this question called *Recent relative problem* is more commonly known as *Lowest common ancestor* (LCA)in the CS literature (and sometimes "Least" or "Last" instead of "Lowest"), and *Most recent common ancestor* (MRCA) in Biology.

   (a) The algorithm starts with two vertices that may or may not be dominating (both are leaves, for example) each other. In the latter case, $a \neq b$ and a will not dominate b in the loop, so a will be iteratively assigned its parent until a dominates b (in the extreme case a reaches the root and dominates all vertices. In the following, b will be assigned to its parent vertex, until it reaches a and the while loop ends.

   The return value, a equal to b, is a recent relative (or LCA) because this is the first vertex to dominate both starting vertices.

(b) The algorithm will iteratively call `parent` until the recent relative (LCA) is found. For each call, a comparison (`!=`) and a call to `dominates` is done. The comparison should be $O(1)$ in a reaonable computational model and if vertices in the tree have integer identifiers such that all children have a larger integer than a parent, then `dominates` is also $O(1)$. Solutions stating that `dominates` take $O(n)$ are accepted.

One can then note that if there are n vertices in T, then the LCA is computed in $O(n)$ time, because there are at most $O(n)$ calls to `parent`. One can also note that if the LCA is at height h, defined as the number of edges above a and b, then the time complexity is $O(h)$.