Suggested solutions for DA3018 exam on 2023-05-29

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

1. (a) Space complexity describes how much memory an algorithm needs as a function of the input size.

   (b) The heap property states that an element is larger or smaller, depending on whether it is a min- or max-heap, than its children.

   (c) Divide-and-conquer is an algorithm strategy in which the input is first partitioned into smaller subsets (often two), that are analyzed independently, and then the results from the subsets is combined together. A classic example of divide-and-conquer is Merge sort.

   (d) Open addressing is used to manage collisions in hash tables. When it is discovered that there is a collision in the hash table, other indexes are probed systematically (for example adjacent subsequent indexes).

2. Use a single linked list and keep track of the first and last element, so a queue Q gets a `first` and `last` reference. With `first` and `last` we can immediately access the elements we need to maintain the queue.

   Suggested implementation below. For `empty?` we check whether "first" points to an element or not.

   ```
   def empty?(Q):
       return Q.first == null
   ```

   When inserting an element, `enqueue(Q, e)`, we should update the last element to point to e and then register e as the new last. Care has to be taken with null values, because Q may be empty when enqueue. In that case, e is set to both first and last in the queue.

   ```
   def enqueue(Q, e):
       v = new Node()
       v.next = null
       if Q.last:
           Q.last.next = v
       else:
           Q.first = v
       Q.last = v
   ```

   For `dequeue(Q)`, we make sure that there is an element to return and in that case update `first` reference. If there is only one element in Q, then the last reference also needs to be updated.

   ```
   def dequeue(Q):
       if Q.first == null:
           return null
       else:
           e = Q.first
           if Q.last == Q.first:
               Q.last = null
   ```

```
Q.first = e.next
e.next = null // To avoid later problems
return e
```

Scoring for (c) and (d): 2p for getting the concept right with constant-time operations and decent pseudocode, and 1p for also getting the details right.

3. Note: there are many ways to solve this problem.

   (a) The guests are given in a guest list L, a set of names.

   I suggest to quantify guest relations using a weight $w(g_1, g_2)$ for two guests g_1 and g_2. Seating structure is unclear, but let us be flexible and define a graph for each party room: each chair is a vertex and there is an edge between to vertices (chairs) if guests on those chairs are "close" (according to Caleb). One long table would mean a connected graph and a set of k small round tables would imply a graph with k components (probably "cliques"). We make sure that there are as many chairs as there are guests.

   A *seating* of a guest list L is a mapping s from V to L, i.e., from the chairs to the guest list, such that $\forall u, v \in V,\ s(u) \neq s(v)$, meaning that no guest is assigned two seats.

   To get happy guests, we want a good seating, and that is assumed to be given by the sum of weights with people around guests at the table. Let us define a score for happiness! The score of a seating can be defined as $\sum_{(u,v)\in E} w(s(u), s(v))$, the sum of relation weight for guests sitting close to each other.

   This score is not perfect. For example, it can ignore seating two people that hate each other nearby, if everyone else sits next to their favorite people. But Caleb did not state any such specifics.

   A computational problem:

   - **Input:** A guest list L, a guest relation weighting w, and a room graph $G = (V, E)$, such that $|V| = |L|$.
   - **Output:** A seating s maximizing $\sum_{(u,v)\in E} w(s(u), s(v))$.

   Grading:

   - 1p for input and output.
   - 1p for clear formulation of input and output.
   - 2p for a clear formulation

   (b) An instance to the problem in (a) can be:

   - $L = [Ali, Bo, Cia]$, just three guests.
   - $w(Ali, Bo) = 1, w(Ali, Cia) = 1, w(Bo, Cia) = 1$, they all like each other,
   - $V = \{u, v, w\}$, $E = \{(u, v), (v, w)\}$, so there are three chairs in a line, like at a bar disk.

   For 1p, you have to give example input to the computational problem you defined.

   Note: the empty party, with no guests and no chairs, is technically an instance here, but how fun is that?

4. First note that Selection sort is $O(n^2)$ and the alternative sorts are $O(n \lg n)$. I use the notation that $\lg = \log_2$.

   (a) If sorting, and in particular its comparisons, dominate the run time then we can assume the time for the algorithm on n elements is $T(n) = Cn^2$, where C is the time for a comparison. Disa noticed that $T(10^3) = 60$s, so we get the equation $10^6 C = 60$, giving $C = 6 \times 10^{-5}$.
   So a comparison takes about 0.06 ms.
   Grading:
   - Clear and reasonable assumptions: 1p.
   - A calculation giving a reasonable answer: 1p.

   (b) Replacing Selection sort with a fast sort means that the algorithm's sorting probably is *probably* still dominated by the sorting. We may have a hidden factor that becomes relevant with faster sorting, but the only information we have is that sorting is a "crucial part" and an observation that sort has "dominated" computing time. With comparisons taking 0.06 ms, it is reasonable to assume sorting can still be expensive.
   If $T'(n) \approx Cn \lg n$, then the time needed will be about $C \times 10^3 \times \lg 10^3$. The log-factor is annoying here, but there are two ways to deal with it.
   - We can use that $2^{10} = 1024$, and some of us computer nerds just know that. Therefore, $\lg 1000 \approx 10$.
   - We can use our log-laws and note that $\lg 10^3 = 3 \lg 10$. And since $2^3 = 8$ and $2^4 = 16$, we can notice that $3 < \lg 10 < 4$. That suggests a guesstimate of $\lg 10 \approx 3.3$, and then $3 \lg 10 \approx 10$. Even with under- or overestimates (from $3 < \lg 10 < 4$), we will get something close to 10, so nothing to worry about.

   Combine this with our estimate $C \approx 6 \times 10^{-5}$ comparisons per second: $C \times 10^3 \times \lg 10^3 \approx 6 \times 10^{-5} \times 10^4 \approx 0.6$ seconds.
   Is the answer reasonable? We are "almost" removing a factor n from the original algorithm, and we go from 60 seconds to 0.6 seconds, so yes that is reasonable when we have $n = 1000$. *Grading*:
   - Clear and reasonable assumptions: 1p.
   - A reasonable calculation: 2p.

5. The time complexity analysis here is based on addition and multiplications taking unit time.

   (a) The first algorithm does some preparations on lines 2 and 3, constant time assignments, and is iterating n times. In each iteration a constant number of unit-time operations are done. Hence, the time complexity is $O(1) + (n + 1) \times O(1) = O(n)$.
   Grading: clear assumptions and noting iteration 1p; conveying that iteration brings the $O(n)$ term is another 1p.

   (b) In the second algorithm, we have basically the same computations, but must notice that power function is $O(i)$ because it computes x^i and that requires i iterations. So in iteration i, we are doing $O(i)$

work. Adding up the time complexity, lines 2 and 3 are $O(1)$ and then in each iteration there is $O(i) + O(1) = O(i)$ work, so in total it is $O(1) + \sum_{i=0}^{n} O(i) = O(n^2)$.

Grading: noting that the extra function is additional work: 1p; making a reasonable argument of the amount of extra work: 1p; a correct logic on full time complexity: 1p.

(c) For Horner's method, we can let $T(n)$ describe the time complexity for an n-degree polynomial. A call to the function `horners_eval` has an array lookup, an addition, and a multiplication with the result of Horner's method on a $n-1$-degree polynomial.

This can be described as $T(0) = O(1)$ and, for $n > 0$, $T(n) = O(1) + T(n-1)$. One can note that we "remove" one term of the polynomial per recursive call and will therefore eventually reach the base case. In that case there has been n recursive calls, each invoking $O(1)$ unit time operations, giving $O(n)$ time complexiy.

Grading:

- Clear and reasonable assumptions, and identifying recursion in a constructive way: 1p.
- Reasoning logically about recursion: 2p.
- Getting the correct answer: 1p.

(d) In the "competing" methods, say `poly_eval`, there are two multiplications (line 5 and 6) per iteration, but with Horner's method there is only one multiplication. The number of array/list accesses and additions is the same, but if multiplication is costly, then Horner's method is faster.

One may worry about the cost of recursion in Horner's method, but one can rewrite the algorithm to a plain iteration instead. Also, in some programming languages and compilers, the cost of using recursion here is zero or close to zero.

Grading:

- Observing and counting the multiplications: 2p

(e) A single-linked list needs space for the value and a pointer to the next element. Using a single-linked list for polynomial coefficients, we should store floats as values.

A Java float is 8 bytes (I looked it up). References (or pointers) on a modern computer also uses 8 bytes. (It has been pointed out to me that my calculations for 32-bit computers are outdated!)

An n-degree polynomial needs $n+1$ coefficients, so in total we need $(n+1) \times (8+8)$ bytes $= 16n + 16$ bytes.

I do not think it makes sense to bring in the storage for the reference to the list here (that would be another 8 bytes), but I accept that some would like to add that.

Grading:

- Reasonable space assumptions: 1p
- Correct estimate: 1p.
- Deducting 0.5p if missing the $+1$ for the zero-degree term.

6. (a) Given that Max is using a graph for the game map, with vertices representing "points of interest" and the graph will therefore probably be constrained to reflect geography (for the enthusiasts: it will be planar and Euclidian). To easily computing the distance from the current vertex v (where the game player is located), we can adapt Breadth-First Search to compute the distance from v. This runs in time linear in number of vertices and edges in the graph. Having the distance d (as number of edges away) allows for mapping to a brightness factor, perhaps something like $1/d$.

   We can decide to iterate through pairs of vertices and compute d as the geographical distance on the map. This implies a time complexity of $O(|V|^2)$ which may be worse than using BFS.

   With a large map, maybe we should combine these methods: using BFS we find the nearby vertices efficiently and spend some extra time computing geographical distance for those.

   (b) What Max is worried about here is whether the graph for the game map is *connected*. We can adapt Depth-First Search to compute connected components, as noted during the course. This is efficient, because the computation is again $O(|V| + |E|)$. However, according to the problem, Max is only interested in a *decision* of connectedness, not which components exists. The latter is of course useful when determining where there might be a problem, but may not be necessary.

   In the BFS algorithm, we keep track of whether a vertex has been visited or not. This indicator can of course be used, after BFS is done, to see what vertices has not been visited.

7. A suggested solution below.

   I suggest using nodes with child references *left* and *right*. We will traverse the tree recursively and return the subtrees we want to have. So nodes are removed by not being returned. Instead, its child is returned.

   - The null-tree is a good base case, so that we can handle all possible input.
   - If exactly one child is null, then the current vertex (the input to the current function call) should be removed and we implement that by not returning it. Instead, whatever is left after recursing on its child.
   - Otherwise we either have a leaf or a node with two children. Proceed recursively to simplify the children. In case of a leaf, both children are null references, but the base case will handle that for us.

```
def simplify_tree(t):
    if t == null:
        return null
    else:
        if t.left != null and t.right == null:
            # One child: skip current node
            return simplify_tree(t.left)
        else if t.left == null and t.right != null:
            # One child: skip current node
            return simplify_tree(t.right)
        else:
            # Two children or a leaf
            t.left = simplify_tree(t.left)
            t.right = simplify_tree(t.right)
            return t
```

   Grading: Base-score is 5, and then we deduct for mistakes. Some examples:

   - -2p for no base case.
   - -1p for unsuitable base case, for example not handling the empty tree well as input.
   - -1p for broken recursive calls.
   - -1p for forgotten case.