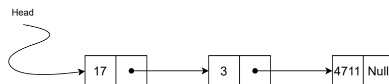
Suggested solutions for DA3018 exam on 2023-08-15

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

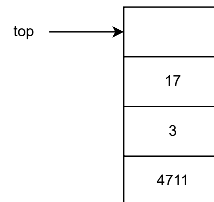
1. (a) To understand how resource needs (like time and memory) is affected by data-set size. It describes how the resource needs *scale*.

   (b) “Unit cost” means that we decides that the basic operations, that we use to measure the resource use, all have the same size/cost. For example, we say that an assignment takes the same time as a multiplication. The actual hardware may have different time needs for different operations, but a unit cost assumption simplifies analysis without losing much in usefulness.

   (c) A single-linked list is a datastructure built by elements with two cells, one for values and one for a pointer/reference to the next element. The next pointer in the last element is Null, or similar value. The figure illustrates a single-linked list with three elements, containing values 17, 3, and 4711. The first element, often named ”head” is indicated.

   (d) A stack is Last-In-First-Out datastructure where you have operations for putting an element on the ”top” of the stack, ”push”, and removing an element, ”pop”. The predicate ”empty?” is typically also defined. The illustration on the right shows a stack where 4711, 3, and 17 have been pushed onto the stack, in that order, meaning that 17 is at the top. The ”top” arrow shows where the next pushed element will end up.

2. (a)
   - Input: an array of n integers.
   - Output: an array $a = (a_1, a_2, \ldots a_n)$ with the input integers ordered such that $a_i \leq a_j$, for all $i, j \in [1, n]$.

   (b) $O(n \lg n)$ is expected when there are no specific assumptions on the input and you count the number of comparisons needed.

   (c) With a Python-inspired pseudocode:

```
mergesort(a):
    if len(a) < 2:
        return a
    else:
        mid = len(a)/2
        half1 = mergesort(a[:mid])
        half2 = mergesort(a[mid:])
        return merge(half1, half2)
```

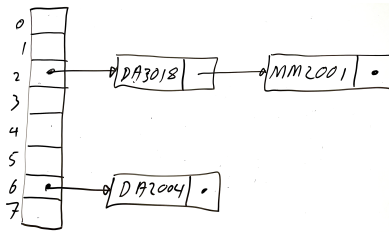
```
merge(lst1, lst2):
   if empty(lst1):
      return lst2
   elif empty(lst2):
      return lst1
   elif lst1[0] < lst2[0]:
      return lst1[0] ++ merge(lst1[1:], lst2)
   else:
      return lst2[0] ++ merge(lst1, lst2[1:])
```

The $++$ operation is "append".

3. (a) *Chaining* refers to the practice of resolving collisions, occurring due to more than one inserted element getting the same hash, by storing those elements in linked lists.

   (b) *Load* is the fraction n/m if n is the number of elements inserted into the hash table and m is the number of entries in the hash table. It is a measure of how full the datastructure is.

   (c) The following simple illustration shows a hash table with capacity 8. The three strings have been inserted, and two of the have been hashed to the same index (2), and the collision is handled using chaining.



   (d) A good hash function spreads elements uniformly over the hash table, and makes use of all bits in the elements to be inserted. This way, collisions are reduced. Say that we have m slots in the hash table. Since we are working with strings of limited size, we can easily loop over all characters to use the whole input element. One can treat the characters as integers and aggregate a large number to map to the range $[0, m-1]$.

```
hash(string s):
   int data = 0
   for c in s:   //  Loop over characters of s
      data = data + code(c)    // Adding character codes
   int hash = ((a * data + b) mod large_prime) mod m
   return hash
```

   Here `large_prime` is a prime $\gg m$ and $a > 0$ and b are some arbitrary integers.

   This is a simple hash function that makes use of the whole input string. The calculation of the hash value is done using a function that has proven to work well for hashing.

4. Consider the following algorithm in pseudo-Python. It ended up being more general than working with only degree k polynomials.

```
def multiply_polynomials(p, q):
   deg_p = degree(p)
   deg_q = degree(q)
   deg_product = deg_p + deg_q // degree of output polynomial
   product = array(size= (deg_product + 1)) // output array

   // Initialize output array:
   for i from 0 to deg_product:
      product[i] = 0

   // Do the term-wise multiplication
   idx_p = 0  // Index of current p coefficient
   for p_coefficient in p:
       idx_q = 0  // Index of current q coefficient
       for q_coefficient in q:
           idx = idx_p + idx_q   // Index of output term
           product[idx] += p_coefficient * q_coefficient
           idx_q += 1
       idx_p += 1
   return product
```

The first three lines will take $O(1)$ time, if arrays are given in some simple way. Initializing the output array involves looping over all its $k+k+1$ elements. With a simple assignment in the loop, initialization is $O(k)$. We are then using a double for-loop, causing $(k+1)\times(k+1)$ executions of the innermost lines where we have addition, multiplication and assignment, all unit cost operations. Hence, the double loop is $O(k^2)$.

5. (a) A binary search tree (BST) is a tree where each node has at most two children and contains a key. The BST property demands that nodes in the "left" subtree of a node v have keys smaller than the key in v, and nodes in the "right" subtree have keys that are larger.

   (b) A vertex contains a geographical position and string. The position can probably be represented using two floats in double precision, using 8 bytes. An old-school string (using ASCII or similar encoding) can store a character in a byte. Using a modern encoding like UTF-8 might require more, so let us assume 1 byte/char for simplicity. Ensuring that we can manage strings with 100 characters means that we allocate about 100 bytes for the string (we might need a stop byte in some programming languages or store the string length in others, but that detail can probably be ignored here).

   The BST also requires that nodes have pointers/references to its children. A modern computer uses 8 bytes per pointer. We can now add up how many bytes we are allocating for node: two references to children, two floats, and a string, which is $2\times 8 + 2\times 8 + 100 = 132$ bytes.

   If we are storing n data points, our BST is going to use about $132n$ bytes.

   (c) Example solution:

```
def check_key(bst, x):
   if not bst:  # The null tree: nothing here
      return False
   else:
```

```
if key(bst) == x:          # Found it
    return True
elif key(bst) < x: # Possibly in the right
     subtree
    return check_key(right(bst), x)
else:
    return check_key(left(bst), x)
```

The base cases are when the input tree is Null or when we have found the key in the root of the (sub-) tree. If there tree is there but the root does not contain x, then we look in subtree where it ought to be due to the BST property and returns the result.

6. (a) Let $n = |V(T)|$. The main function, `get_center_vertex`, makes n iterations and calls to the helper function `get_eccentricity(v,T)`, and some constant-time operations (assignments and comparisons). We can recognize the helper function as a breadth-first search, implying a $O(|V| + |E|)$ time complexity, with an added loop to find the maximum eccentricity in $O(n)$. Since the BFS is run in a tree, we have $|V| = |E| + 1$ and $O(|V| + |E|) = O(n)$.
   The whole algorithm is therefore $n \times O(n) = O(n^2)$.

   (b) Breadth-first search, or BFS.

   (c) The algorithm is making n calls to `get_eccentricity(v, T)`, meaning that almost the same function call is reapeated n times. To improve on $O(n^2)$ one should try to find a way to avoid the repetition.
   Suppose u and v are the vertices that are furthest apart from each other in T and let p_{uv} be the path from u to v. Let c be a vertex on p_{uv} such that $d(c, u) - d(c, v) \leq 1$, implying that c is a center on p_{uv}, because it is in the middle of the path. One can see that c must be a center also for T, because otherwise there must be another vertex w being the center and it would be one path longer than p_{uv}, contradicting that u and v are furthest apart.
   It is therefore sufficient to find the vertices with maximum eccentricity, because the center will be in the middle on the path between them. This can be done in two steps. First, pick an arbitrary vertex s and call a modified version of `get_eccentricity(s, T)` that also returns the vertex with maximum eccentricity. Call that vertex v'. Then call `get_eccentricity(v, T)` to get the vertex u' furthest away from v'. Claim: $(u, v) = (u', v')$, which can be proved as follows. Assume without loss of generality that $d(s, v) \geq d(s, u)$. Can we have $d(s, v') \geq d(s, v)$? Let w be a vertex on p_{uv} where the path from s to v intersects. Since $d(s, v) \geq d(s, u)$, then $d(w, v) \geq d(w, u)$. If $d(s, v') \geq d(s, v)$, then $d(v', v) \geq d(u, v)$, and p_{uv} would not be the longest path in T.
   Calling `get_eccentricity` twice and finding a center on its path takes $O(|V|)$ steps, so linear time.