UNIVERSITY OF OTAGO EXAMINATIONS 2022

COMPUTER SCIENCE

COSC201

ALGORITHMS AND DATA STRUCTURES

Semester 1

Duration : Three hours

Online exam platform: Blackboard

Candidates should answer questions as follows:

Candidates should answer all questions.

Questions are worth various marks, and marks are shown this way: (5)
The total number of marks available is 80.

The following material is provided:

Nil.

Use of calculators:

All calculators are permitted.

The following resources are permitted:

This is an **open book** examination. Your responses **must** be in your own words. Copy and pasting material is not permitted.

Other Instructions:

Candidates MUST read all Instructions and Notices prior to starting the Exam.

You must complete your exam within its stated duration. At the end of that time you must submit your work immediately. An additional five minutes are allowed to submit/upload your answers.

1. *Union-find*

The purpose of the *union-find* data structure is to maintain a *dynamic set partition*, i.e., a partition of the elements $\{0, 1, 2, \ldots, n-1\}$ into groups, that supports:

- `union(x,y)` which joins the groups containing x and y (if they are not already in the same group), and
- `find(x)` which determines a representative element of the group that x belongs to.

The values of `find(x)` and `find(y)` must be the same whenever x and y belong to the same group (and only then).

(a) Suppose that a union-find instance `u` has been initialised with $n = 10$.

- At this point, before any calls to `union` have been made, what is the value of `find(4)`?

Now, the following calls to `union` are made: `union(4, 2)`, `union(7, 4)`, `union(6, 7)`, `union(5,8)`.

- What is the value of `find(0)`?
- What can be said about the value of `find(4)` (independent of the specific implementation)?
- How many different values of x are there for which `find(x)` and `find(4)` have the same value?
- What is the minimum additional number of calls to `union` that can be made before there is only one group, i.e., `find(x)` takes on the same value for all x (give a brief justification). (5)

(b) In our first union-find implementation (`UF1`) an array, `reps`, of representatives was maintained in such a way that the value of `find(x)` was just the element `reps[x]`.

- Why does that imply that, in `UF1`, the cost of a `find` operation is $\Theta(1)$?
- Why does that imply that, in `UF1`, the cost of a `union` operation is $\Theta(n)$? (2)

(c) By contrast, in `UF2` and `UF3` the array `reps` maintained *local representatives*, and the value of `find(x)` was determined by repeatedly replacing `x` by `reps[x]` until the value did not change.

- How does this affect the (worst-case) cost of a `find` operation?
- What change was made between the two implementations in order to reduce the worst-case cost of `find` from $O(n)$ (for `UF2`) to $O(log\, n)$ (for `UF3`)? (3)

2. *Induction*

(a) What are the four parts of a formal proof by induction? (2)

(b) Recall that $n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$. Using the fact that

$$4! = 4 \times 3 \times 2 \times 1 = 24 > 16 = 2 \times 2 \times 2 \times 2 = 2^4,$$

prove by induction that $n! > 2^n$ for all $n \geqslant 4$. (4)

(c) Suppose that the implementation of a recursive algorithm (with input parameter a non-negative integer n) has the following characteristics:

- The base cases include $n = 0$ and all operate in time at most C for some fixed constant C.
- Each recursive case makes a single call to an instance of the algorithm with a strictly smaller value of the parameter, and its execution time is at most time of that call plus some other constant D.

Prove, by induction, that, if $T(n)$ represents the execution time for the implementation at n, then $T(n) \leqslant C + D \times n$ for all non-negative integers n. (4)

3. *Recursion and Dynamic Programming*

The *Fibonacci* numbers are the sequence f_n defined recursively by: $f_0 = 1$, $f_1 = 1$, and $f_n = f_{n-1} + f_{n-2}$ for $n > 1$. The following four methods all correctly compute f_n as `fibX(n)` (replacing `X` by `A`, `B`, `C` or `D` as appropriate).

```
public long fibA(int n) {
  if (n <= 1) return 1;
  return fibA(n-1) + fibA(n-2);
}
```

```
public long fibB(int n) {
  return fibPair(1,1,n);
}

private long fibPair(long a, long b, int n) {
  if (n == 0) return a;
  return fibPair(b, a+b, n-1);
}
```

```
public long fibC(int n) {
  long a = 1, b = 1;
  for(int i = 0; i < n; i++) {
    long c = a + b; a = b; b = c;
  }
  return a;
}
```

```
public long fibD(int n) {
  long[] f = new long[n+1];
  f[0] = 1; f[1] = 1;
  for(int i = 2; i <= n; i++) {
    f[i] = f[i-1] + f[i-2];
  }
  return f[n];
}
```

Answer each of the following *with a brief justification* (i.e., not just ‘none’ or a list of letters).

(a) Which (if any) of the four methods make use of recursion? (2)

(b) Which (if any) of the four methods use dynamic programming techniques? (2)

(c) Which (if any) represent divide and conquer algorithms? (2)

(d) Which (if any) require $O(1)$ extra storage? (2)

(e) Which (if any) have run-time bounded by $O(n)$? (2)

4. *Sorting*

Three algorithms for sorting arrays are described briefly below.

- *InsertionSort*: Working from left to right, each element is inserted into the proper position relative to itself, and all the elements to its left (i.e., of smaller index).
- *QuickSort*: The first element of the array is chosen as a pivot. All elements smaller than it are placed before it, and all elements greater than or equal to it are placed after it. Then *QuickSort* is applied to the elements before it, and the elements after it.
- *MergeSort*: The first and second halves of the array are sorted using *MergeSort*. Then the resulting arrays are merged into a single one.

Let n represent the number of elements in an array to be sorted.

(a) Are there any arrays for which *InsertionSort* requires $\Theta(n^2)$ time? If so, what do they look like? (1)

(b) Are there any arrays for which *QuickSort* requires $\Theta(n^2)$ time? If so, what do they look like? (1)

(c) Are there any arrays for which *MergeSort* requires $\Theta(n^2)$ time? If so, what do they look like? (1)

(d) What does it mean to say that an implementation operates *in place*? Which (if any) of the three algorithms above can be implemented so that they operate in place? (2)

(e) What is the worst-case time complexity of *MergeSort*? Explain briefly how this can be derived. (3)

(f) Refined versions of *QuickSort* and *MergeSort* are used in Java's libraries. Why not choose just one or the other? (2)

5. *Heaps*

Suppose that we have an array-based implementation `Heap.java` that represents a max-heap of `String` elements. That is, it contains a data-field `heap` of type `String[]` and the following conditions hold:

- The element at index 0 is the greatest (i.e. last, closest to z, etc.) in alphabetical order of all the elements of the heap.
- The children (if any) of the element at index `i` are at indices `2*i+1` and `2*i+2` and are before (or equal) to it in alphabetical order.

We consider positions 0 through $n-1$ of the array to be *occupied* if the heap contains n elements (the remaining positions, if any, are free space).

The `add` operation places a new element at the first unoccupied index and then adjusts its position relative to its parent (and grandparent, etc.) until the conditions are restored.

The `remove` operation removes (and eventually returns) the element at position 0, by replacing it with the element at the last occupied index and then adjusting its position relative to its children (and grandchildren, etc.) until the conditions are restored.

(a) List (in increasing order of their index) the occupied positions of the array `heap` after the following sequence of operations. (2)

```
Heap h = new Heap();
h.add("cat");
h.add("dog");
h.add("ant");
h.add("kea");
```

(b) What is printed by the following? (2)

```
Heap h = new Heap();
String[] as = new String[] {"kit", "dog", "gnu",
  "moa", "pet", "boa", "rat", "elk", "cub", "eel"};
for(String a : as) h.add(a);
System.out.println(h.remove());
```

(c) List (in increasing order of their index) the occupied positions of the array `heap` after the following sequence of operations. (2)

```
Heap h = new Heap();
String[] as = new String[] {"yak", "boa",
  "ant", "roc", "eel", "kea"};
for(String a : as) h.add(a);
h.remove(); h.remove();
```

(d) How would you implement a *priority queue* data structure using a heap? In a priority queue, each element consists of a *value* and an associated *priority* and we add pairs consisting of a value and a priority, but when we remove an element all we want back is the value. (4)

6. *Binary search trees*

In a *binary search tree* (BST) of strings (for the sake of example) each node contains a string and has at most two children (also nodes), called *left child* and *right child*. The strings stored in the left child or any of its descendants are less than the string stored at the node, and the strings stored in the right child or any of its descendants are greater than the string stored at the node. Binary search trees support `add`, `remove` and `find` as fundamental operations.

(a) If we construct a new binary search tree and then add elements to it in sorted order, e.g., `"ant"`, then `"bat"`, then `"cat"`, then `"dog"` and so on, what does the structure of the tree look like? (1)

(b) The *depth* of a node in the tree is the number of links from it through its parent to the root (so the root is depth 0, its children are depth 1, their children are depth 2, and so on). What is the maximum possible number of nodes at depth k? (1)

(c) The *height* of a tree is the maximum depth of any node in the tree. If we have a tree of height h, explain why each of the operations can be implemented with time complexity $O(h)$. (2)

(d) A *fully balanced* tree of height h has the property that, at every depth less than h there are as many nodes as possible of that depth (i.e., layer by layer it is full, except possibly in the final layer). Given a sorted array of strings, how could we construct a fully balanced BST containing exactly those strings? (3)

(e) In *self-balancing* trees, the dynamic operations (`add` and `remove`) include, as side-effects, modification of the tree structure that maintain some level of balance. Why is this desirable? Why don't the library implementations generally produce fully-balanced trees? (3)

7. *Hashing*

(a) The hashcodes of various `String` objects in Java are shown below:

`"ant"`	96743
`"boa"`	97716
`"eel"`	100300
`"kea"`	106055
`"roc"`	113094
`"yak"`	119395

Assume that we are constructing a hash set of these elements using an underlying array of 10 buckets. The elements are to be added in alphabetical order.

(i) Would any collisions occur when we add these elements to the array? If so, what is the first addition that causes a collision? (1)

(ii) Show the contents of the underlying buckets if we use chaining to deal with collisions. (1)

(iii) Show the contents of the underlying buckets if we use linear probing. (1)

(iv) Suppose that we are using linear probing, that we remove `"boa"`, replacing it by a tombstone, and then add `"dog"` with hashcode 99644. At what index would it be stored? (1)

(b) In each of the situations below, which of the Java library classes `HashSet`, `TreeSet`, `HashMap`, or `TreeMap` would be the most appropriate representation of the data? Include a brief justification (of a sentence or two in each case).

(i) The intended application is to implement an autocomplete feature for words, so the data is a list of strings (valid words) and the problem in general is to list all the possible completions of a word from a given prefix. (2)

(ii) The intended application is to return a student's record from their ID number. (2)

(iii) The intended application is a web-crawling program. Given an initial page it will visit all pages linked from it, then all linked from there etc. (with some vaguely defined completion criterion). The information from a page does not need to be associated with the page, but the application does need to be able to avoid visiting a given page more than once. (2)

8. *Graphs*

Pseudocode for breadth-first and depth-first traversal from a given vertex v in a graph is given below. We will assume that the vertices have distinct `String` labels and that the list of neighbours of a vertex is returned in alphabetical order.

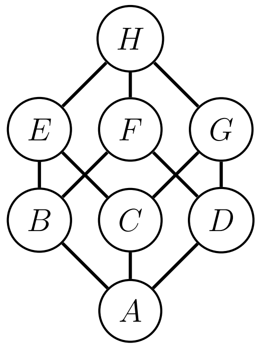
Breadth-first

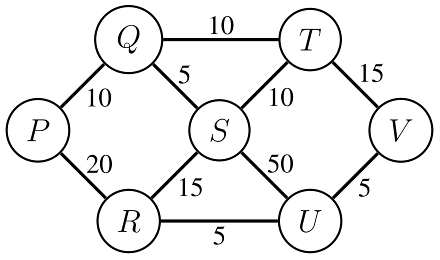
```
$q.add(v)$, $v.seen \leftarrow$ true
while $q$ is not empty do
    $w \leftarrow q.remove()$
    for $n$ in $w.neighours$ do
        if $n.seen$ = false then
            $q.add(n)$
            $n.seen \leftarrow$ true
        end if
    end for
end while
```

Depth-first

```
$s.push(v)$, $v.seen \leftarrow$ true
while $s$ is not empty do
    $w \leftarrow s.pop()$
    for $n$ in $w.neighours$ do
        if $n.seen$ = false then
            $s.push(n)$
            $n.seen \leftarrow$ true
        end if
    end for
end while
```

Note that the only difference is the use of a queue (q) in the breadth-first case and a stack (s) in the depth-first case. For clarity about the order: in a three vertex graph with vertices A, B and C and edges AB and AC only, the breadth-first traversal from A in visiting order is ABC while the depth-first traversal is ACB (because C is added to the stack after B and hence removed first).





(a) List the breadth-first traversal of the left-hand graph above, from vertex A. (1)

(b) List the depth-first traversal of the left-hand graph above, from vertex A. (1)

(c) List the breadth-first traversal of the left-hand graph above, from vertex D. (1)

(d) List the depth-first traversal of the left-hand graph above, from vertex D. (1)

(e) Find the shortest path using Dijkstra's algorithm in the right-hand graph above from P to V where the numbers denote the edge weights. Show your work by listing (in priority order) the elements of the priority queue at each step as triples (c, p, w) where c is the current (i.e., end) vertex, p is the previous vertex, and w is the total weight of the corresponding path from P to c. (4)

(f) Suppose that G is a weighted connected graph all of whose edge weights are different. Why must the edge of minimum weight belong to *every* minimum spanning tree? (2)

END