UNIVERSITY OF OTAGO EXAMINATIONS 2024

SCHOOL OF COMPUTING

COSC201

Algorithms and Data Structures

Semester One

(TIME ALLOWED: 3 HOURS)

This examination paper comprises 9 pages

Candidates should answer questions as follows:

Candidates should answer **All** questions

Marks are shown thus: (2)

The total number of marks for this exam is 80

The following material is provided:

Nil

Use of calculators:

Any model of calculator provided that is battery powered, silent, truly portable and free of communication capabilities

Candidates are permitted copies of:

No additional material

Other Instructions:

Nil

TURN OVER

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 has been initialised with $n = 10$.

- At this point, before any calls to `union` have been made, what is the value of `find(6)`? (1)

Now, the following calls to `union` are made: `union(2, 5)`, `union(7, 4)`, `union(6, 7)`, `union(5,8)`. All of the following refer to the state of the union-find instance at this point.

- What is the value of `find(0)`? (1)
- What are the possible values of `find(6)` (independent of the specific implementation)? (1)
- How many different values of x are there for which `find(x)` and `find(4)` have the same value? (1)
- 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). (1)

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

(c) By contrast, in `UF2` and `UF3` the array `reps` maintained *local representatives*.

- In `UF2` and `UF3` how was the result `find(x)` computed using the local representatives? (1)
- 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`)? (2)

2. *Induction*

(a) What is the *induction step* in a proof by induction and what does it accomplish? (2)

(b) Prove by induction that for all positive integers n,

$$1+2+3+\cdots+n > n^2/2.$$

(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 new parameter that is strictly less than one half of the original parameter, and its execution time is at most time required for that call plus some other constant D.

Prove, by induction, that, if $T(m)$ represents the execution time for the implementation at m and n and k are any positive integers with $k \leqslant 2^n$, then $T(k) \leqslant C + D \times n$.

(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) {
  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];
}
```

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

```
public long fibD(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);
}
```

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 $\Theta(n)$ 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) For each of the three algorithms, how much additional memory might be required in order to sort an array of size n? (2)

(e) *MergeSort* is presented above as a divide and conquer recursive algorithm. Explain briefly how it can be implemented in a bottom-up, non-recursive fashion. (3)

(f) A refined version of *MergeSort* called *TimSort* is used in Java's library to sort arrays of reference type. Describe two ways in which it differs from basic *MergeSort*. (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("eel");
h.add("bat");
h.add("gnu");
h.add("kea");
h.add("cat");
```

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

```
Heap h = new Heap();
String[] as = new String[] {"rats", "and", "bats", "and",
    "alley", "cats"};
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. (3)

```
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 of the element of greatest priority. (3)

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 from a sorted list by first adding the middle element, then the elements on either side of that, then the elements on either side of those 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). If a tree has the maximum possible number of nodes at depth k must it also have the maximum possible number of nodes at depth $k-1$? Give a brief justification of your answer. (2)

(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) Explain how to take the *preorder traversal* of a binary search tree. If we have a binary search tree A and construct a new binary search tree, B, by adding elements to it in the order produced by the preorder traversal of A, how, if at all, will the structure of A and B be related? (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? (2)

7. *Hashing*

A collection of various `String` objects in Java are shown below along with their hashcodes:

`"ant"`	96743
`"cat"`	98262
`"dog"`	99644
`"hog"`	103488
`"kea"`	106055
`"moa"`	108287
`"rat"`	112677
`"yak"`	119395

Suppose that we constructed a hash set of these elements using an underlying array of 10 buckets, choosing the index by taking the remainder of the hashcode modulo 10, and added the elements in alphabetical order.

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

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

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

(d) Suppose that we are using linear probing, that we remove `"dog"`, replacing it by a tombstone, and then add `"yyy"` with hashcode 120153. At what index would it be stored? (1)

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).

(e) The intended application is to store a character's inventory in a role-playing game. Each possible item has an associated weight, and each character can only carry a limited total weight. (2)

(f) The intended application is to provide access to items in a shop in a role-playing game. The most common use case is to require a list of all available items whose cost lies in a particular range (i.e., to answer questions like "What can I buy for between 100 and 150 gold pieces?"). (2)

(g) The intended application is to store the names of all the individual characters in a role-playing game. (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.

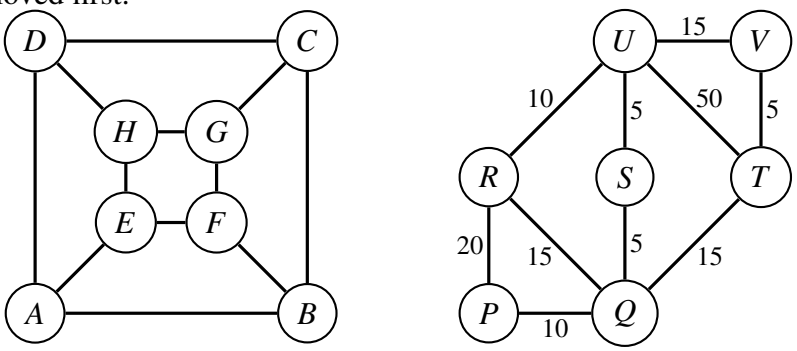
Breadth-first

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

Depth-first

```
s.push(v), v.seen ← true
while s is not empty do
    w ← s.pop()
    for n in w.neighours do
        if n.seen = false then
            s.push(n)
            n.seen ← 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 H. (1)

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

(e) Using the weights shown, list the vertices and their distances from P in the right-hand graph above in order of their distance from P. (2)

(f) In the right-hand graph above is it possible for the edge TU to belong to a minimum spanning tree? Give a brief justification of your answer. (2)

(g) In the right-hand graph above is it possible for both edges QR and RU to belong to a minimum spanning tree? Give a brief justification of your answer. (2)

END