Revision Exercises

Algorithms and Data Structures I

If you wish to replicate exam conditions for these exercises, answer questions by hand on paper without any extra resources or study materials. There are exercises that involve writing code, but first attempt the answer using pen and paper, and code it up on a computer afterwards to check your answer.

Solutions in blue.

Part A (50 marks)

Question (a): Consider the following piece of psuedocode:

```
1: a ← 2
2: a ← a^3
3: function SMALLERTHAN(n)
4:     if n<10 then
5:         return n
6:     end if
7:     return 10
8: end function
9: a ← SMALLERTHAN(a)
10: a ← a+1
11: a ← SMALLERTHAN(a)
```

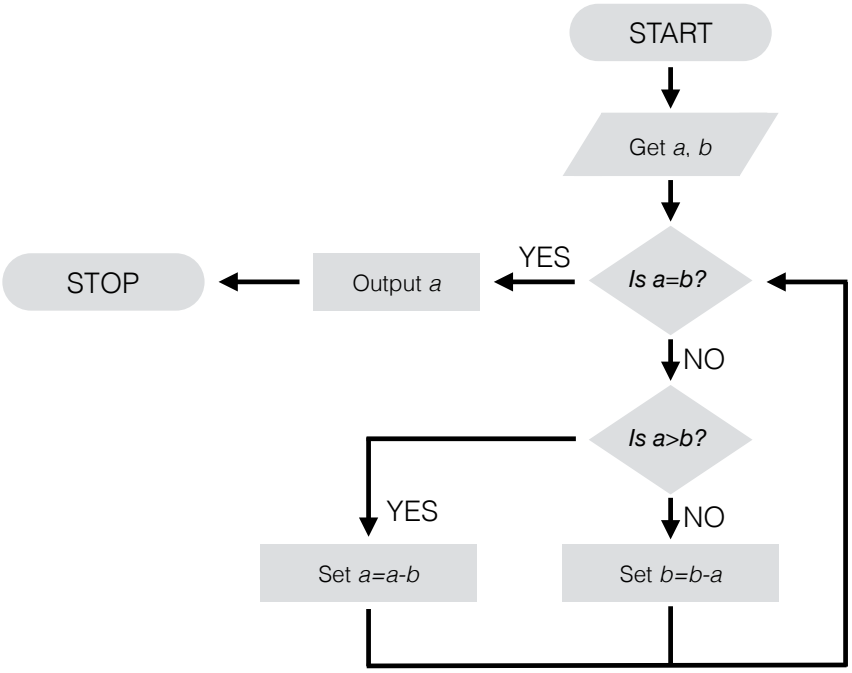
1. What is the value of *a* at the end of the second line? [1 mark]
   8
2. What is returned by SMALLERTHAN(9+3)? [1 mark]
   10
3. What is the value of *a* at the end of line 9? [1 mark]
   8
4. What is the value of *a* at the end of line 11? [1 mark]
   9

Question (b): Write a function in psuedocode that takes a number n as an input parameter (which is assumed to be an integer) and returns true if the number is perfectly divisible by three (leaves no remainders when divided by three) or returns false if it is not perfectly divisible by three. [4 marks]

Code should look something like this:

```
function DIVISIBLE(n)
    if n mod 3 = 0 then
        return true
    end if
    return false
end function
```

Question (c): Consider the following flowchart:



1. Write a function in pseudocode that implements the algorithm in this flowchart. The function should take two input parameters and return a single number. [5 marks]

   Code should look something like this:

```
function EUCLID(a,b)
    while a ≠ b do
        if a > b then
            a ← a-b
        else
            b ← b-a
        end if
    end while
    return a
end function
```

2. What does this algorithm compute? [2 marks]

   This computes the greatest common divisor of two numbers, a and b.

Question (d): Give one difference between a dynamic array and a stack. [2 marks]

Any of the following differences: in a dynamic array all elements are addressable, but not in a stack; a dynamic array has the length operation, but a stack does not; in a dynamic array elements can be added in various locations, but can only be added to the top of the stack.

Question (e):

1. In a coursework assignment you are assigned tasks. These tasks need to be completed in numerical order for the work to be completed correctly, with the first task being completed first, the second task being completed second and so on. The text of each task in your coursework can be stored as a string in memory, and then this string can be stored in a data structure: tasks will be stored if they are not yet completed, and removed when they are completed. Of the two following abstract data structures, which is the best way of storing the strings to make it easier to schedule your coursework tasks?

   - A stack
   - A queue

   [1 mark]

   A queue

2. Give a justification for your choice of abstract data structure in part 1 of this question; in doing so explain how data is added and removed from the abstract data structure in such a way that is suited to completing your coursework.

   [3 marks]

   A queue is a first-in-first-out (FIFO) abstract data structure so the first element put into the data structure (using the enqueue operation), which will be the head, is the first to leave. Thus the first task will be enqueued first, and it will be the first to leave when completed (using the dequeue operation). The last task will thus be stored in the tail.

Question (f): Consider the following piece of pseudocode:

```
1: new Queue q
2: ENQUEUE[10,q]
3: store←HEAD[q]
4: ENQUEUE[5,q]
5: DEQUEUE[q]
6: store←store+HEAD[q]
```

1. What is the value of *store* at the end of line 3?[1 mark]
   10

2. What is the value of *store* at the end of line 6?[1 mark]
   15

3. Write a function in pseudocode that takes a queue storing numbers as an input parameter and will return the sum of all of the numbers in the elements. If the input queue is empty the function will return the value null. [5 marks]

Code should look something like this:

```
function SUMQUEUE(queue)
    if EMPTY[q]=true then
        return null
    end if
    a ← HEAD[q]
    DEQUEUE[q]
    while EMPTY[q]=false do
        a ← a + HEAD[q]
        DEQUEUE[q]
    end while
    return a
end function
```

Question (g): When should we use the linear search algorithm instead of the binary search algorithm? [2 marks]

Linear search should be used on unsorted vectors, binary search should be used on sorted vectors.

Question (h):

1. What is the worst-case time complexity of insertion sort? [2 marks]

   $O(n^2)$

2. From the point-of-view of worst-case time complexity, which algorithm would you prefer, insertion sort or bubble sort? Briefly explain your answer. [3 marks]

   They are equally as good since the worst-case time complexity of bubble sort is also $O(n^2)$.

Question (i): You are trying to perform a task on a large database of N elements, where N will tend to get very large. One colleague tells you that they have an algorithm (called algorithm A) that takes at most $0.01N^2 + 2N + 4$ steps, another colleague says they found an algorithm (called algorithm B) that takes at most $1000N \log_2(N) + 2N$ steps.

1. Give the worst-case run-times of algorithm A and algorithm B in terms of 'Big O' notation. [2 marks]

   Worst-case run-time of A *is $O(N^2)$, and run-time of* B *is $O(N \log_2 N)$. You may use any base in the logarithm, or just write $O(N \log N)$.*

2. Which algorithm are you going to use? Explain your reasoning. [3 marks]

   You should use algorithm B*, as asymptotically (since N can get large) $O(N \log_2 N)$ is better (grows slower) than $O(N^2)$.*

Question (j): This question is about the Fibonacci numbers: $0, 1, 1, 2, 3, 5, 8, 13, ...$, where each new number is obtained by adding the previous two in the sequence. The nth Fibonacci number is denoted F_n, and thus $F_n = F_{n-1} + F_{n-2}$. By convention the first two Fibonacci numbers are $F_0 = 0$ and $F_1 = 1$.

Write a recursive function in pseudocode called FIBONACCI(n), which takes the number n as an input parameter, and returns the nth Fibonacci number (assuming n is a non-negative integer). This new function should not use any form of iteration. [6 marks]

Code should look something like this:

```
function FIBONACCI(n)
    if n < 2 then
        return n
    end if
    return FIBONACCI(n-1) + FIBONACCI(n-2)
end function
```

Question (k): Consider the following piece of pseudocode that computes the factorial of a non-negative integer n:

```
function FACTORIAL(n)
    a ←1
    for 1 ≤ i ≤ n do
        a ← a × i
    end for
    return a
end function
```

Give a brief explanation why the number of operations in a standard implementation (in the RAM model) of FACTORIAL(n) is in the 'Big O' class $O(n)$ for the input n. [4 marks]

Before the for loop there is a constant number of operations. There are n iterations due to the for loop, and thus there are $O(n)$ iterations. In each iteration there is a constant number of operations (multiplication) so the total number of operations is $O(n)$.

Part B (90 marks)

Question 1: This question is about search algorithms.

Part (a) Consider the following array:

$$S = [0, 1, 16, 4, 9, 25]$$

You are tasked with sorting the array S in ascending order so the smallest value is in the first element. Directly run through the Quicksort algorithm by hand on S. Show explicitly each step taken in the algorithm. [7 marks]

Calculate pivot to be element 2 (floor of (0+5)/2)
Partition the array according element 2 to get
[0, 1, 9, 4, 16, 25]
Final location of pivot is in element 4. Right sub-array is already sorted. Need to sort left sub-array.
Pivot of left sub-array is element 1 (floor of (0+3)/2). Partition according to value at this element.
[0, 1, 9, 4, 16, 25]
Left sub-array consists of element 0 and is sorted. Right sub-array consists elements 2 and 3. Pivot is element 2.
Partition:
[0, 1, 4, 9, 16, 25]
Array is now sorted.

Part (b) You are now tasked with algorithmically searching the array S to see if it has an element with the value 2. Directly run through the binary search algorithm by hand on the sorted version of S. Show explicitly each step taken in the algorithm. [7 marks]

Start with sorted array [0, 1, 4, 9, 16, 25]
Start with `left` variable as 0, and `right` variable as 5
Calculate mid-point to be element 2 (floor of (`left`+`right`)/2)
Element 2 does not contain the value 2
Update `right` variable to be 1 (since 2 is smaller than 4)
Calculate mid-point to be element 0 (floor of (0+1)/2)
Element 0 does not contain the value 2
Update `left` variable to be 1 (since 2 is larger than 0)
Calculate mid-point to be element 1 (floor of (1+1)/2)
Element 1 does not contain the value 2
Update `left` variable to be 2 (since 2 is larger than 1)
Return "not found" since `left` is larger than `right`

Part (c) What is the worst-case time complexity in `n` of the binary search algorithm for an array of length `n`? [3 marks]

$O(\log n)$

Part (d) Consider the following problem:

Given that $x^2 = n$, where n is an integer, is x an integer?

We will use search algorithms to solve this problem. The first approach to this problem is to first generate an array of all integers squared and then see if n is in that array using linear search.

Consider the following piece of JavaScript that implements this solution approach.

```
function intSqr(n) {
  var a = [];
  for (var i = 0; i <= n; i++) {
    a[i] = i * i;
  }
  return linearSearch(a, n);
}
```

This function `intSqr(n)` calls the linear search algorithm `linearSearch(a, n)` for array `a` looking for the value `n`. Write the function `linearSearch(a, n)`, which will return `true` if `n` is in `a`, and return `false` otherwise. [5 marks]

Code should look something like this:

```
function linearSearch(a, n) {
  var l = a.length;
  for (var i = 0; i < l; i++) {
    if (a[i] == n) {
      return true;
    }
  }
  return false;
}
```

Part (e) The algorithmic approach of `intSqr(n)` first creates an array and then searches, but elements of the array could be 'created one at a time' and then checked to see if they are equal to `n`. Consider the following JavaScript code:

```
function intFind(n) {
  for (var i = 0; i <= n; i++) {
    if (i * i == n) {
      return true;
    } else if (i * i > n) {
      return false;
    }
  }
}
```

Both of these methods essentially use a form of the linear search algorithm by sequentially checking the square of integers `i` from `0` to `n`. Note that these squares of integers are generated in a 'sorted' manner, since $i^2 < (i+1)^2$, so we can adapt the Binary Search algorithm to look for an integer among the squares of integers. That is, we calculate the mid-point between `0` and `n * n` to compare it with `n`; depending on the value of this mid-point we will find the value we are looking for, or calculate a new mid-point. The implementation of this approach therefore should have time complexity $O(\log n)$.

Write a new JavaScript function called `intBin(n)` that will use the Binary Search algorithm to see if the square root of `n` is integer or otherwise. The function will take the integer `n` as input parameter and return `true` if the square root of `n` is integer, and `false` otherwise. The implementation of your function should have time complexity $O(\log n)$, so you need to avoid creating an array with `n` elements. [8 marks]

Code should look something like this:

```
function intBin(n) {
  // the imaginary array will be [0*0, 1*1, 2*2, 3*3, 4*4, ... , n*n], so it will have
    length n+1
  var length = n + 1;
  var left = 0;
  var right = length - 1;

  while (left <= right) {

    // now we try to find the mid-point of the imaginary array, and then square it
    var mid = Math.floor((left + right) / 2);
    if (mid * mid == n) {
      return true;
    } else if (mid * mid < n) {
      left = m + 1;
    } else {
      right = m - 1;
    }
  }

  return false;
}
```

Question 2: This question is about recursion and stacks.

Part (a) Consider the following piece of JavaScript:

```
1 function sum(n) {
2   var a = 0;
3   for (var i = 1; i <= n; i++) {
4     a = a + i;
5   }
6   return a;
7 }
```

Complete the following table of values returned by this function for particular inputs `n`:

n	sum(n)
0	*0*
1	*1*
2	*3*
3	*6*

[5 marks]

Part (b) Write a new JavaScript function called `recSum(n)` that is a recursive implementation of `sum(n)`, but does not use any form of iteration. It should take the same inputs as `sum(n)`, and produce the same output. [5 marks]

Code should look something like this:

```
1 function recSum(n) {
2   if (n == 0) {
3     return 0;
4   }
5   return n + recSum(n-1);
6 }
```

Part (c) The factorial of an integer n is written $n!$ and is defined as $n! = n \times (n-1) \times (n-2) \times ... \times 2 \times 1$, and $0! = 1$. Write a new function called `recFactorial(n)` that is a recursive implementation of the factorial in JavaScript, without using any form of iteration. It should take as input parameter the non-negative integer `n`, and return a number corresponding to the factorial of `n`. [5 marks]

Code should look something like this:

```
function recSum(n) {
  if (n == 0) {
    return 1;
  }
  return n * recSum(n-1);
}
```

Part (d) Write a new function called `recChoose(sign, n)`, which takes a string called `sign` and number `n` as input parameters, and returns a number. The function should return `recSum(n)` if `sign` is the string `"+"`, or return `recFactorial(n)` if `sign` is the string `"*"`. For any other value of the string `sign`, the function should return the string `"Invalid input"`. [5 marks]

Code should look something like this:

```
function recChoose(sign, n) {
  if (sign == "+") {
    return recSum(n);
  } else if (sign == "*") {
    return recFactorial(n);
  } else {
    return "Invalid input";
  }
}
```

Part (e) Consider the following piece of pseudocode:

```
function FactorialStack(n)
    new Stack s
    PUSH[1,s]
    if n > 0 then
        for 1 ≤ i ≤ n do
            MISSING
        end for
    end if
    return s
end function
```

The incomplete function FactorialStack(n) should create an empty stack and then push elements to it to get a stack of `n+1` elements where the top will store the factorial of `n`, the next element down will store the factorial of `n-1`, and so on. What should go in the place of MISSING to do this? [3 marks]

PUSH[i × TOP[s],s]

Part (f) When FactorialStack(n) is completed, for a stack f that is returned by the function call FactorialStack(4), what is returned by TOP[f]? [2 marks]

24

Part (g) What is the time complexity in 'Big O' notation for an implementation of your recursive function `recFactorial(n)`? Give a brief argument why. [5 marks]

The time complexity is $O(n)$. The recursive calls in the function are handled by a call stack; every time the

function is called, a stack frame is pushed to the stack containing the relevant data. There are $n-1$ of these function calls, so there will be $O(n)$ frames pushed to the stack. Then after all of these calls have been completed, each call returns, and the frame is popped from the stack, involving $O(n)$ operations. Within each stack frame a constant number of operations is carried out, therefore the number of operations is $O(n)$.

Question 3: This question is about linked lists.

Part (a) Explain why it is preferable to use a linked list to implement a stack rather than use an array data structure (note that this is distinct from a JavaScript array). [3 marks]

A stack is a dynamic abstract data structure: elements containing data are added and removed. It is easier to add and remove data from a linked list than from an array, since arrays are of a fixed size.

Part (b) Consider the following piece of pseudocode:

new Stack *s*
PUSH[1,*s*]
PUSH[3,*s*]
POP[*s*]
PUSH[2,*s*]
PUSH[5,*s*]

Draw a linked list of the final form of the stack *s* at the end of the pseudocode, with the top of the stack being the first element of the list. [5 marks]

The linked list should look like this:

Part (c) Consider the following piece of JavaScript:

```
1 function LLNode(data) {
2   this.data = data;
3   this.next = null;
4 }
5 var head = new LLNode(5);
6 console.log(head.data);
7 console.log(head.next);
```

This is a function that creates an object that creates a node of a linked list with the value `data` stored in the node. On line 5, the variable `head` will then store the object corresponding to the node.

1. What is printed to the console for line 6 of this code? [1 mark]
   5

2. What is printed to the console for line 7 of this code? [1 mark]
   null

3. In another piece of JavaScript code, construct the final linked list that was drawn in part (b) of this question. Assume that the function `LLNode(data)` is already defined and use it to construct the linked list. Store the linked list as an object in the variable `head` (assume that head is not defined at the beginning of your code). [5 marks]

   Code should look like this:

```
1 var head = new LLNode(5);
2 head.next = new LLNode(2);
3 head.next.next = new LLNode(1);
```

Part (d) The linear search algorithm can be amended to be applied to linked lists, by inspecting every node of the list until a value is found. Write a function in JavaScript that traverses a linked list from the front to the end searching for a value in the nodes. Call the function `searchLL(list, item)`: it takes as inputs the linked list as an object called `list`, and `item`, which is the value that is being searched. The function should return `true` if the value `item` is in the list, and return `false` otherwise. You may assume that the function `LLNode(data)` is defined and you can call it. [6 marks]

Code should look like this:

```
function searchLL(list, item) {
  var temp = list;
  while (temp !== null) {
    if (temp.data === item) {
      return true;
    }
    temp = temp.next;
  }
  return false;
}
```

Part (e) Consider the following piece of JavaScript:

```
function swapLL(point) {
  if (point.next !== null) {
    var store = point.data;
    point.data = point.next.data;
    point.next.data = store;
    return point;
  }
  return false;
}
```

This function will swap the data values on neighbouring nodes in a linked list when the pointer to the first node is given as an input to the function. The function will return `false` if the next pointer of the node at `point` points to null.

The goal is now to write a function that will implement a version of the `Bubble Sort` algorithm on a linked list (that contains only numbers) to sort that list. On each pass, neighbouring nodes are compared and then sorted according to ascending order. Write a function in JavaScript called `bubbleLL(head)` that takes as input parameter a reference to the list (`head`) and returns the list with all its elements sorted in ascending order. You may assume that `swapLL(point)` is already defined and you can call it in your function. [9 marks]

Code should look like this:

```
function bubbleLL(head) {
  if (head === null) {
    return head;
  }
  while (true) {
    var count = 0;
    var store = head;
    while (store.next !== null) {
      if (store.data > store.next.data) {
        swapLL(store);
        count++;
      }
      store = store.next;
    }
    if (count === 0) {
      return head;
    }
  }
}
```