Data Structures and Algorithms – (COMP SCI 2C03)
Winter, 2021
Assignment-I

Due at 11:59pm on February 14th, 2021

- **No late assignment accepted.**
- Make sure to submit a version of your assignment ahead of time to avoid last minute uploading issues.
- Submit one assignment solution as a PDF file per group on Avenue
- If the solution submitted by two groups is the same. Both teams will get a zero mark on the assignment.
- Present your algorithms in Java or Pseudocode (Pseudocode is preferred).
- It is advisable to start your assignment early.

This assignment consists of 10 questions, and is worth 75 marks.

Question 1 Given a doubly linked-list (L), and its head and tail:

(a) Write an algorithm that outputs the reverse of the list L, as a new list, in $\Theta(n)$ time, where n is the length of the linked list. Your solution must update both the *next* and *prev* pointers appropriately. [5 marks]

Answer:

procedure $Reverse\_List(L)$
  $L^R = NULL; count = 0$ ▷ count= no.of notes in L^R
  if $L.head = NULL$ **then**

```
    return L^R
x = L.head; L^R.head = x; L^R.tail = x; count = 1
L^R.tail.next = NULL; L^R.head.prev = NULL
while x.next ≠ NULL do
    x = x.next
    L^R.head.prev = x; x.next = L^R.head
    L^R.head = x; L^R.head.prev = NULL
    count = count + 1
    if count = 2 then
        L^R.tail.prev = x
```

(b) Explain your algorithm and its running time analysis. [5 marks]
Answer: The `Reverse_List` procedure take an input list L and outputs the reverse of this list as a new list L^R. If L is empty then it returns an empty reverse list. Otherwise, it reads/scans the input list from left to right, starting from its head ($L.head$). During this scan, when it reads a new node x in L, it appends this node at the beginning of L^R; that is, at its head.

The algorithm has a single `while` loop which reads each node (except the first) of the input list L of length n. Therefore, it executes $n-1$ times. The other assignment/conditional statements take constant time. Therefore, $T(n) = n + t_1$, where t_1 is a constant. $T(n) \in \Theta(n)$. By choosing $c_1 = 0.5$, $c_2 = t_1 + 1$ and $n_0 = 1$, $c_1 n \leq T(n) \leq c_2 n$.

Question 2 Given a non-negative integer array $A[1..n]$ (that is, A contains numbers from the set $\{0, 1, 2, 3, 4, \ldots\}$), the next smaller value of A (NSV_A) is an integer array of length n, where $NSV_A[i]$, $1 \leq i \leq n$, is defined as follows:

1. if $A[i] = 0$ then $NSV_A[i] = 0$
2. otherwise, $NSV_A[i] = j$, where
   (a) for every $h \in [1..j-1]$, $A[i] \leq A[i+h]$, and
   (b) either $i + j = n + 1$ or $A[i] > A[i+j]$.

In short, $NSV[i]$ shows how far one should trace forward from A[i] to reach a smaller value than $A[i]$. Write an algorithm to compute the NSV array in $O(n)$ time, using the stack data structure. Below is an example of an NSV array. [6 marks]

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
A	0	1	3	3	7	7	1	5	5	0	2	2	6	6	0	4	4
NSV_A	0	8	4	3	2	1	3	2	1	0	4	3	2	1	0	2	1

Answer: See Algorithm 1.

Algorithm 1 Algorithm to compute NSV_A

```
procedure COMPUTE_NSV_A(A)
    i = 1; S = NULL                                  ▷ stack S is empty
    while (i ≤ n) do
        if (S.isempty()) then S.push(i)
        else
            if (A[S.top()] ≤ A[i]) then S.push(i)
            else
                while A[S.top()] > A[i] do
                    NSV_A[S.top()] = i − S.top()
                    S.pop()
                S.push(i)
        i = i + 1
    while (!S.isempty()) do
        if (A[S.top()] == 0) then NSV_A[S.top()] = 0
        else
            NSV_A[S.top()] = n + 1 − S.top()
        S.pop()
```

Question 3 Implement the Queue data structure using the Stack data structure. For this question you need to implement all the Queue operations (Dequeue, Enqueue etc.) using ONLY the stack operations (PUSH, POP etc.). [6 marks]
Answer: See Figure 1.

Question 4 Using ONLY the definition of $O(f(n))/\Theta(f(n))$ prove that the following statements are TRUE. Your proofs using Limits will not get a mark:

```java
class MyQueue {

    private Stack<Integer> stack1;
    private Stack<Integer> stack2;

    public MyQueue() {
        stack1 = new Stack<>();
        stack2 = new Stack<>();
    }

    public void enqueue(int x) {
        stack1.push(x);
    }

    public int dequeue() {
        if(stack2.isEmpty()){
            while(!stack1.isEmpty())
                stack2.push(stack1.pop());
        }
        return stack2.pop();
    }

    public boolean isEmpty() {
        return stack1.isEmpty() && stack2.isEmpty();
    }
}
```

Figure 1: Solution for Q3

(a) ~~$(65n^4 + 2n + 3)/(n^4 + 1) = \Theta(n^4)$ [3 marks]~~ - Question is incorrect. So we ignore it.

(b) $\log 2^{n^2} + n^2 + 1 = \Theta(n^2)$ [3 marks]
Answer: Let $T(n) = \log 2^{n^2} + n^2 + 1 = 2n^2 + 1$ – choose $c_1 = 1$, $c_2 = 3$, and $n_0 = 1$

(c) $1 + 2 + 3 + \ldots + n = O(n^2)$ [3 marks]
Answer: Let $T(n) = 1 + 2 + 3 + \ldots + n = n(n+1)/2 = n^2 + n/2$ – choose $c_1 = 4$, and $n_0 = 1$

Question 5 Using ONLY the definition of $O(f(n))/\Theta(f(n))$ prove that the following statements are FALSE. Your proofs using Limits will not get a mark:

(a) $(1000n^4 + n^2 + 4n)/(\log n) = \Theta(n^4)$ [3 marks]
Answer: The proof is by contradiction. Let $T(n) = (1000n^4 + n^2 + 4n)/(\log n) \in \Theta(n^4)$. Since for any function $f(n) \in \Theta(g(n))$;

$f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$. Then, by assumption $T(n)$ must be in $O(n^4)$. Let c, n_0 be the least constants, such that the below inequality holds for all $n \geq n_0$

$$T(n) \leq c \cdot n^4.$$

However, for all $n \geq 2$, $T(n) > c \cdot n^4$ – a contradiction. Therefore, $T(n) \notin \Theta(n^4)$.

(c) $\sqrt{n} = O(\log n)$ [3 marks]
Answer: The proof is by contradiction. Let c, n_0 be the least constants, such that the below inequality holds for all $n \geq n_0$

$$\sqrt{n} \leq c \cdot \log n.$$

However, for all $n \geq 2^{1/c}$, $\sqrt{n} > c \cdot \log n$ – a contradiction.

Question 6 Given a sequence $< x_1, x_2, \ldots, x_n >$ of integers, we want to find a subsequence of consecutive items that give the largest sum. Let LI and UI be the lower and upper indices defining such a subsequence of consecutive items, and let $MSum$ be the sum of items in this subsequence. Then your algorithm should report the following $< MSum, LI, UI >$. For example, if you are given the following sequence of integers $< -5, -3, 7, -2, 4, 6, -7 >$, then $< 15, 3, 6 >$ should be reported since the sum $x_3 + x_4 + x_5 + x_6 = 15$ gives the maximum. In the special case where all the integers are negative, $< 0, 0, 0 >$ should be reported.

(a) Give the simplest possible brute force $\Theta(n^3)$ algorithm for this problem. It should check all possible candidate sequences. [5 marks]
Answer: For each subrange i, j we compute the subsequence sum explicitly. Among all subranges giving the maximum sum, this algorithm returns the right most one, if multiple such subranges exist.

```
procedure MAX_SUM_BRUTEFORCE(x)
    minIndex = 0; maxIndex = 0; maxSum = 0
    for i = 1 to n do
        for j = 1 to n do
            sum = 0
            for k = 1 to n do
```

```
            sum = sum + x[k]
            if sum > maxSum then
                maxSum = sum
                minIndex = i
                maxIndex = j
        return < maxSum, minIndex, maxIndex >
```

(b) Modify your algorithm from (a) to reduce the running time to $\Theta(n^2)$ [6 marks]
Answer:
For each i we compute the cumulative sums for each subrange of the form i, j and test for maximality as we proceed. This algorithm returns the same result as the brute force algorithm.

```
procedure MAX_SUM_IMPROVED(x)
    minIndex = 0; maxIndex = 0; maxSum = 0
    for i = 1 to n do
        sum = 0
        for j = i to n do
            sum = sum + x[j]
            if sum > maxSum then
                maxSum = sum
                minIndex = i
                maxIndex = j
    return < maxSum, minIndex, maxIndex >
```

Question 7 Construct an algorithm that will sort four numbers in no more than 5 comparisons, regardless of the input. [5 marks]
Answer: The following is inspired by the mergesort algorithm. It outputs a, b, c, and d in sorted order. If you wish, the output statements can be replaced by assignment to 4 new variables or to an array. Note that the number of comparisons is either 4 or 5.

Question 8 Give traces, in the style explained in class, showing how the numbers $9, 1, 5, 2, 11, 18, 7, 23, 13, 0, 4$ are sorted with top-down mergesort. [5 marks]
Example top-down mergesort trace:
merge([9, 1, 5, 2, 11, 18, 7, 23, 13, 0, 4], 0, 0))
merge([1, 9, 5, 2, 11, 18, 7, 23, 13, 0, 4], 3, 3))

```
if a > b
  swap(a,b)
endif
if c > d
  swap(c, d)
endif
if a < c
  output a
  Merge (b, c, d)
else
  output c
  Merge (d, a, b)
endif
```

merge([1, 9, 5, 2, 11, 18, 7, 23, 13, 0, 4], 2, 2))
merge([1, 9, 2, 5, 11, 18, 7, 23, 13, 0, 4], 0, 1))
merge([1, 2, 5, 9, 11, 18, 7, 23, 13, 0, 4], 6, 6))
merge([1, 2, 5, 9, 11, 18, 7, 23, 13, 0, 4], 5, 5))
merge([1, 2, 5, 9, 11, 7, 18, 23, 13, 0, 4], 9, 9))
merge([1, 2, 5, 9, 11, 7, 18, 23, 13, 0, 4], 8, 8))
merge([1, 2, 5, 9, 11, 7, 18, 23, 0, 4, 13], 5, 7))
merge([1, 2, 5, 9, 11, 0, 4, 7, 13, 18, 23], 0, 4))
Final array: [0, 1, 2, 4, 5, 7, 9, 11, 13, 18, 23]

Question 9 Write an algorithm for the non-recursive version of quicksort based on a main loop where a subarray is popped from a stack to be partitioned, and the resulting subarrays are pushed onto the stack. Note : Push the larger of the subarrays onto the stack first, which guarantees that the stack will have at most $\log N$ entries. [7 marks]
Answer: See Figures 2 and 3

Question 10 (a) Is an array that is sorted in decreasing order a max-oriented heap? Justify your answer. [5 marks]
Yes. A max-oriented heap would have the largest elements at the front of the array, and similarly to a max-heap, A[0] would return the largest element in the array and this would continue

```java
class QuickSort {
    // Different partitioning logic could be used here
    static int partition(int numArray[], int low, int high) {
        int pivot = numArray[high];
        // smaller element index
        int i = (low - 1);
        for (int j = low; j <= high - 1; j++) {
            // check if current element is less than or equal to pivot
            if (numArray[j] <= pivot) {
                i++;
                // swap the elements
                int temp = numArray[i];
                numArray[i] = numArray[j];
                numArray[j] = temp;
            }
        }
        // swap numArray[i+1] and numArray[high] (or pivot)
        int temp = numArray[i + 1];
        numArray[i + 1] = numArray[high];
        numArray[high] = temp;
        return i + 1;
    }
```

Figure 2: Solution for Q9 - PART A

to hold true even if the current largest element is popped. However, counter-arguments may be presented discussing the fact that heaps do not ensure that all of the elements are sorted.

(b) For $N = 16$, give arrays of items that make heapsort use the most number of compares, and the least number of compares. Explain your answer. [5 marks]
Best case: any array where every value is the same. This is so that heapify does not need to be repeatedly called.
E.g. [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Worst case: any array with no duplicates that requires repeated calls to heapify.
E.g. [16, 15, 1, 14, 2, 13, 3, 12, 4, 11, 5, 10, 6, 9, 7, 8], assuming a max-heap is generated.

```
static void quickSort(int nums[], int low, int high) {
    int[] stack = new Stack < Integer > ();

    stack.push(low);
    stack.push(high);

    while (!stack.isEmpty()) {
        high = stack.pop();
        low = stack.pop();

        int pivot = partition(nums, low, high);

        int sizeLeft = 0;
        int sizeRight = 0;

        if (pivot - 1 > low) {
            sizeLeft = pivot - low
        }

        if (pivot + 1 < high) {
            sizeRight = high - pivot
        }

        if (sizeLeft > 0 && sizeRight > 0) {
            if (sizeLeft >= sizeRight) {
                stack.push(low);
                stack.push(pivot - 1);
                stack.push(pivot + 1);
                stack.push(high);
            } else {
                stack.push(pivot + 1);
                stack.push(high);
                stack.push(low);
                stack.push(pivot - 1);
            }
        } else if (sizeLeft > 0) {
            stack.push(low);
            stack.push(pivot - 1);
        } else if (sizeRight > 0) {
            stack.push(pivot + 1);
            stack.push(high);
        }
    }
}
```

Figure 3: Solution for Q9 - PART B