CSC263: Problem Set 1

September 17, 2019

1 Problem 1

(a) Let the size of array A be given by n. The costs of executing lines 2 and 5 of the sum function are constant, as in they run once, and so the computational costs associated with them would be c_1 and c_2. The cost of executing the for loop on line 3 of the sum function would be nc_3 because the number of times the loop runs is directly proportional to the length of the array, A. So with a length of n, each time the loop runs, there is a computation cost of c_3 associated with it, and it runs n times, thus nc_3. Additionally, the increment operation on line 4, also runs each time the for loop runs, and so, it too runs n times, and has a computational cost of nc_4 (there is a computational cost of c_4 associated with each time). Together we get $t_{sum}(A) = c_1 + c_2 + nc_3 + nc_4$. Combining c_1 and c_2 to a constant d and c_3 and c_4 to a constant b, we can simplify the equation to $t_{sum}(A) = bn + d$.

(b) Let the size of array A be given by n. Here we let $n > 3$ to consider the worst case. The for loop on line 11 executes at most $n - 2$ times. Each iteration has a computational cost of c_1 and so we get $(n - 2)c_1$. However, within the for loop exists an if condition on line 12 which is dependent on SUM(A[i:i+2]) $\neq 0$. The SUM operation runs at a cost of $3c_2$ because it iterates through A[i] to A[i+2], which is 3 elements, and adds them together, and c_2 is the computational cost of the SUM operation. The SUM operation is not directly proportional the length of array A, in the given code. Assuming that the if condition is not met (because otherwise the return statement on line 13 will cause the for loop to end prematurely), then the for loop continues to run. Finally, after the for loop, the return statement on line 14 executes and this runs in constant time with a computational cost of c_3. So we get $T(n) = (n-2)(c_1+3c_2)+c_3$. Letting $3c_2 + c_1 = a$ and $c_3 = b$, we get $T(n) = a(n - 2) + b$.

(c) Upper Bound:
Suppose A is an array of size $n > 3$. The for loop on line 11 executes at most n times. Each iteration takes at most c steps (for some constant c). The SUM operation on line 12 runs at most d times, where d is a

constant. There are at least b lines of code outside the for loop, where b is a constant. So $T(n) \leq (c+d)n + b \; \epsilon \; O(n)$.

Lower Bound:
The trivial solution is to consider an array A of size less than 3 because the if condition on line 7 is met and returns True which runs in constant time, with some computational cost c. And so we would get $T(n) \geq c \; \epsilon \; \Omega(1)$. So instead, suppose A is an array of size $n > 3$ and that the if condition on line 12 never comes true, i.e. SUM(A[i:i+2]) = 0, then the for loop on line 11 executes at most $n-2$ times, with a computational cost of c_1. Additionally, the SUM operation runs at most 3 times per iteration of the for loop, with a computational cost of c_2 so $3c_2(n-2)$ altogether. Finally the return statement on line 14 runs in constant time and so we can add a computational cost of 1. So $T(n) \geq (3c_2 + c_1)(n-2) + 1 \; \epsilon \; \Omega(n)$, since the function is linear.

Since $T(n) \; \epsilon \; O(n)$ and $T(n) \; \epsilon \; \Omega(n)$, then $T(n) \; \epsilon \; \Theta(n)$.

2 Problem 2

(a) For the Array approach: For a fixed length array, we take an array and store elements of the multi-set in it, in the same order. When the array is filled up, a new array of double its length is created, and the items from the original array are copied to the new array and then the old array is deleted.

For the Linked List approach: For a Linked List, either a singly linked or doubly linked list would work. In either case, we store elements of the multi-set in the same order, in nodes and create new nodes as needed, when adding more elements to the multi-set.

(b) For all operations, we assume that the given indices are valid.

All arrays in the array operations have indices starting at 0.
Array Operations:

```
Insert(L,x,i):
    curr = L[i]
    count = i + 1
    L[i] = x
    while type(L[count]) is int do
        new = L[count]
        L[count] = curr
        curr = new
        count = count + 1
```

```
Pop(L,i):
    popped = L[i]
    count = i
    flag = True
```

```
    while flag is True do
        if type (L[count]) is int then
            L[count] = L[count+1]
            count = count + 1
        else then
            flag = False
    return popped
```

```
Remove(L,x):
    for i = 0 to Size(L)-1 do
        if L[i] == x then
            count = i
            flag = True
            while flag is True do
                if type(L[count]) is int then
                    L[count] = L[count+1]
                    count = count + 1
                else then
                    flag = False
```

```
Size(L):
    counter = 0
    while counter >= 0 do
        if type(L[counter]) is int then
            counter = counter + 1
        else then
            break
    return counter
```

```
Sort(L):
    last = Size(L) - 1
    sorted = False
    while not sorted do
        sorted = True
        for i = 0 to (last-1) do
            if L[i] > L[i+1] then
                swap L[i] and L[i+1]
                sorted = False
        last = last - 1
```

All linked lists in the linked list operations have indices starting at 0.
Linked List Operations:

```
Insert(L,x,i):
    new = Node(x)
    if i==0
        new.next = L.head
        L.head = new
        return
    node = L.head
    for y=0 to i-1
        node = node.next
    new.next = node.next
    node.next = new
```

```
Pop(L,i):
    node = L.head
```

```
    for y=0 to i-1
        node = node.next
    x = node.next.data
    node.next = node.next.next
    return x
```

```
Remove(L,x):
    node = L.head
    if node.data == x
        L.head = node.next
        return
    while node.next != null
        if node.next.data == x
            node.next = node.next.next
            break
        node = node.next
```

```
Size(L):
    x = 1
    node = L.head
    while node.next != null
        x = x + 1
        node = node.next
    return x
```

```
Sort(L):
    node = L.head
    size = Sort(L)
    for x=0 to size
        for y=0 to size-1
            if node.data > node.next.data
                swap(node.data,node.next.data)
            node = node.next
```

(c) Linked List

Linked list is preferable when we want to perform an operation to the head of the list. To perform these operations we just need to access the head node (see Ex F1) and not worry about making any changes to the rest of the list.

If we used an array it would have been more computationally expensive because when we insert, remove or pop from the front of the list then the remaining integers in front of the index have to be moved down by one index (in the case of remove or pop) and moved up by one index (in the case of insert). Since just making changes to head node takes $O(1)$ time compared to array which take $O(n)$, therefore in this situation linked list are more preferred

```
F1
Insert(L,x,i): #Linked List
    new = Node(x)
    if i==0
        new.next = L.head
        L.head = new
```

```
7           return
8       node = L.head
9       for y=0 to i-1
10          node = node.next
11      new.next = node.next
12      node.next = new
13
14  Insert(L,x,i): #Arrays
15      curr = L[i]
16      count = i + 1
17      L[i] = x
18      while type(L[count]) is int do
19          new = L[count]
20          L[count] = curr
21          curr = new
22          count = count + 1
```

In this situation to insert an integer for a linked list at index 0 will run lines 3-7 but in an array it will run line 15-22 as all the other integers have to move up one index.

Array
Array list is preferable when performing an operation which is at the end of the list. When making changes to the end of the linked list, you would first have to transverse through the whole list to reach the last node and then make changes to that node. But in an array list we can just access the array at the last index and make changes to the array. (see Ex F2)

```
1   F2
2   Insert(L,x,i): #Linked List
3       new = Node(x)
4       if i==0
5           new.next = L.head
6           L.head = new
7           return
8       node = L.head
9       for y=0 to i-1
10          node = node.next
11      new.next = node.next
12      node.next = new
13
14  Insert(L,x,i): #Arrays
15      curr = L[i]
16      count = i + 1
17      L[i] = x
18      while type(L[count]) is int do
19          new = L[count]
20          L[count] = curr
21          curr = new
22          count = count + 1
```

In this situation to insert an integer for a linked list at the end will run lines 3-12 as we need to find the last node but in an array it will run line 15-17 but since we are at the last index then line 18 will not run.