COMP2123-Assignment 1

August 2, 2024

Notation Clarification

$a \leftarrow 0$	The value of variable 0 is assigned to variable a.
$[a : b]$	The sequence a, $a+1, \ldots, b-1$.
`boolean`	The data type representing integers.
`float`	The data type representing float.
`self`	a reference to the current instance of the class, and is used to access variables that belongs to the class
`real`	The data type representing the real numbers.
`void`	Used to show that a function does not return anything.
`null`	The variable representing nothingness.
`bool`	The data type representing a boolean value which is either true or false.

Problem 1

```
function ALGORITHM(A)
    $n \leftarrow$ length of $A$
    num_matches $\leftarrow 0$
    for $i \in [0 : n]$ do
        for $j \in [i + 1 : n]$ do
            if $A[i] == A[j]$ then
                num_matches $\leftarrow$ num_matches $+ 1$
    return num_matches
```

Algorithm 1: Pseudocode 1

a)

All the assignments in line 2 and 3 take $O(1)$ time, The loop in line 4 will run for n times, inside that loop, we have second loop at line 5 that will run $n - (i + 1)$ times, inside that loop, we perform at most one assignment that is going to cost O(1) time.

Let $T(n)$ be the running time of the algorithm with n is the length of input array A, we can bound it as below.

$$\begin{aligned}
T(n) &= O(1) + O(1) + \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} O(1) \\
&\leq O(1) + O(1) + \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} O(1) \\
&\leq O(1) + O(1) + \sum_{i=0}^{n-1} O(n) \\
&\leq O(1) + O(1) + \sum_{i=0}^{n-1} O(n) \\
&\leq O(1) + O(1) + O(n^2) \\
&\leq O(n^2)
\end{aligned}$$

$O(n^2)$ is the upperbound of this algorithm.

b)

To lowerbound the running time of this algorithm, we consider only the running time of algorithm during the last $\frac{n}{2}$ iteration. Since this is part of the full execution, analyzing only part gives a lower bound on the total running time.

Let $T(n)$ be the running time of the algorithm with n is the length of input array A, we can bound it as below.

$$\begin{aligned}
T(n) &= \Omega(1) + \Omega(1) + \sum_{i=\frac{n}{2}}^{n-1} \sum_{j=i+1}^{n-1} \Omega(1) \\
&\geq \Omega(1) + \Omega(1) + \sum_{i=\frac{n}{2}}^{n-1} \sum_{j=\frac{n}{2}+1}^{n-1} \Omega(1) \\
&\geq \Omega(1) + \Omega(1) + \sum_{i=\frac{n}{2}}^{n-1} ((n-1) - (\frac{n}{2}+1) + 1) \\
&\geq \Omega(1) + \Omega(1) + \sum_{i=\frac{n}{2}}^{n-1} (\frac{n}{2} - 1) \\
&\geq \Omega(1) + \Omega(1) + \frac{n^2}{4} - \frac{n}{2} \\
&\geq \Omega(n^2)
\end{aligned}$$

$\Omega(n^2)$ is the lower bound of this algorithm.

Problem 2

a)

Stack ADT that we want to extend will have the following operations:

- `void push(int e)`: inserts an element, `e`.
- `int pop()`: removes and return the last inserted element.
- `int top()`: returns the last inserted without removing it
- `int Count()`: returns the number of items on the stack.
- `bool isEmpty()`: indicates wheather no elements are stored.

Aditional operations:

- `int Sum()`: returns the sum of all elements on the stack.
- `float Average()`: returns the average of all elements on the stack.

The implementation of the stack is summarised as follows:

- We use singly linked list to implement stack ADT.
- The singly linked list class will have the following information.
    - The reference to the head of the node in singly linked list.

```
Class SINGLE_LL
    Node head                ▷ The first node in singly linked list, null if linked list is empty
```

Algorithm 2: Class Singly linked list

- To update first node of a singly linked list after we add an element or remove an element in singly linked list, we have to design a method naming **updateFirst(Node n)** to update the head of singly linked list, which will be really useful for us to implement push and pop function of Stack.

```
function UPDATEFIRST(self, Node n)
    self.head = n
```

Algorithm 3: updateFirst method

- As we can see in algorithm 3, we will update the head of linked list to **Node n**, and this method will run in $O(1)$ time because we just assign the reference of **Node n** to head.
- To push and pop in stack, which is implemented by linked list, so we have to push Node object instead of pushing value. Thus, we have to have the Node class.

- The Node class will have the following information.
  - The value of the element.
  - The reference to the next node, `null` if it is the end node.

```
Class NODE
    int value                         ▷ The value store in the Node
    Node next                         ▷ The reference to the next node, null if it is the end node
```

Algorithm 4: Class Node

- The stack class holds the following information
  - The number of elements in the stacks, which is first assigned to 0.
  - The sum of all elements in the stacks, which is first assigned to 0.
  - The empty singly linked list, with the first Node is null.

```
Class STACK
    int size                          ▷ The number of element in the stack, which first assign to 0
    int sum                           ▷ The sum of all elements in the stack, which first assign to 0
    Single_ll collection              ▷ The empty singly linked list, with the first Node is null.
```

Algorithm 5: class Stack

With reference to the beginning node, both `push` and `pop` operations can be easily implemented such that they run in $O(1)$ time.

- `push(e)`

```
function PUSH(self, int $e$)
    $node \leftarrow$ new Node                                  ▷ Create a new object Node.
    $node.value \leftarrow e$                                   ▷ The node will store the element we want to push
    $node.next \leftarrow self.collection.first()$              ▷ The reference to the next Node will be the first Node in linked list
    $self.collection.updateFirst(node)$                         ▷ Update the head of the singly linked list
    $self.sum \leftarrow self.sum + e$                          ▷ Update sum
    $self.size \leftarrow self.size + 1$                        ▷ Update size
```

Algorithm 6: push operation

The idea of creating push function is that, first we will create a **Node** object, which stores the value of the element we want to push into the stack, and this object will points to the head of the singly linked list. After that, we have to update the head of the singly linked list to the node that we have just been created. Not only that, because each time we use this function, the number of elements in the stack and the sum of all elements in the stack have to be update each time when we call this function.

- Correctness
  - Following the the property of stack is **last in first out**. Thus, the push operation always push the value at the top of the stack. That is why the function push we have just created always insert the new element at the head of the linked list. After that, we will update the head of the singly linked list. Thus the push operation always correct because it always push the new value into the top of the stack.
  - As we can see in line 6 and 7 at **Algorithm 6**, each time we push an element into the stack, we will update the value of **sum** and **size**.
- Running time
  - From line 2 to 4, all of the assignments will run in 0(1) time. Not only that, the **(**updateFirst(node)) method also run in $O(1)$ time, and line 6 and will run in 0(1) time. Thus, this function will run in $O(1)$ time.

```
1: function isEmpty(self)
2:     if $self.size = 0$ then
3:         return True                  ▷ if the size of stack equals to 0, return True
4:     return False
```

Algorithm 7: isEmpty operation

IsEmpty function uses to check the stack is empty or not. The stack is empty if and only if there are no elements being stored on the stack. Thus, if the size of the stack equals to 0, the stack is empty, which is being cover in if statement in line 2.

The running time of this function is $O(1)$.

- `pop()`

```
1: function pop(self)
2:     if $self.isEmpty()$ then
3:         return null                  ▷ When the stack is empty, we can't conduct pop operation.
4:     int node_removed ← $self.collection.first().value$
5:     $self.collection.updateFirst(self.collection.first().next)$
6:     $self.size \leftarrow self.size - 1$
7:     $self.sum \leftarrow self.sum -$ node_removed
8:     return node_removed
```

Algorithm 8: Pop operation

The idea of creating pop function is that, first we use the condition statement to check the stack is empty or not by using function **isEmpty()**. If this function returns True, which means the stack is empty so we can not pop any element out in the stack. Thus we will return **null**. At vice versa, the function **isEmpty()** return False, which means the stack is not empty, then will assign the value of the first node in singly linked list (the first node of singly linked list also have to be the head of a stack). After that, we have to update the first element of singly linked list because we have to keep the head Node always be the top of the stack and the first of Node of Singly Linked list. When the element in the stack being popped out, the size of the stack and the sum of stack have to be updated. size will be decreased by 1 and sum will be decreased by the value of the node that being popped out of the stack.

- Correctness
  - Following the the property of stack is **last in first out**. Thus, the pop operation always pop the value at the top of the stack. That is why the function pop we have just created always take out element at the head of the linked list. After that, we will update the head of the singly linked list. Thus the pop operation always correct because it always pop the value at the top of the stack out.
  - As we can see at line 6 and 7 we also update the value of size and sum of the stack.
- Running time
  - The running time of this algorithm is $O(1)$ time.
- `top()`

```
function TOP(self)
    if self.isEmpty() then
        return null                                   ▷ When the stack is empty, the top of stack is null
    Node node_head ← self.collection.first().value
    return node_head
```

Algorithm 9: top operation

The idea of creating top function is that we will check the stack is empty or not. If the stack is empty, we will return **null** because when the stack is empty we don't have any elements in stack, the head of stack is null. Otherwise, we will return the head of stack.

- Correctness
  - Because in method pop and push, we also update the head of the stack. Thus, the head of stack always point to the correct location. That is why the method count always return the top of the stack without removing it.
- Running time
  - The conditional statement will run in $O(1)$ time and all the assignments and returning of the method will run in $O(1)$ time so the running time of this methods is $O(1)$.
- `Count()`

```
function COUNT(self)
    return self.size
```

Algorithm 10: Count operation

The **Count** method will return the size of the Stack.

- Correctness
  - In push and pop methods, we will increase the size by 1 for push method and decrease the size by 1 for pop methods. Thus, the size of the Stack will always true.
- Running time
  - Because this method just only use to return the size of the stack, which also similar to the number of an element in the stack, so the running time is only $O(1)$.
- `Sum()`

```
function SUM(self)
    return s.sum
```

Algorithm 11: Sum operation

The **Count** method will return the sum of all elements in the Stack.

- Correctness
  - In push and pop methods, we will increase the sum by the value of **Node** that being pushed into the stack for push method and decrease the sum by the value of the **Node** that being popped for pop methods. Thus, the sum of all elements on the Stack will always true.
- Running time
  - Because this method just only use to return the sum of the stack, so the running time is only $O(1)$.
- `Average()`

```
function AVERAGE(self)
    average ← $self.Sum()/self.Count()$
    return average
```

Algorithm 12: Average operation

Recall that the average is the total sum of the elements divided by the number of elements. Since we already store the size of the stack and the sum of the elements. When a new element get pushed or popped, we also update both of them.

- Correctness
  - The correctness follows directly from the definition of average. Because we also proved that sum is true and size is true, average equals to $\frac{\text{sum}}{\text{size}}$, so average have to be true.
- Running time
  - Because this method only calculate the division of sum and size so it will cost only $O(1)$.

Problem 3

a) DESIGN ALGORITHM

```
function PAIRS_SUM(array B, int aim)
    i ← 0
    j ← $B.size() - 1$
    valid_pairs ← 0
    for $i < j$ do
        if $B[i] + B[j] > aim$ or $B[i] + B[j] = aim$ then
            valid_pairs ← valid_pairs + $(j - i)$
            j ← j − 1
        else
            i ← i + 1
    return valid_pairs
```

Algorithm 13: Ex 3

We will utilize the two-pointer technique to solve this problem. Initially, one pointer will scan from left to right, starting at the first element of the sorted array B, while the second pointer will begin at the last element of the ascending array B. We'll set the variable for valid pairs to 0, which we'll return later. Within the loop at line 5, we'll iterate through each element in array B once. Inside this loop, we'll have a conditional statement to check if:

$$B[i] + B[j] \geq \text{aim}$$

If this condition holds true, it means that every element from the ith to the $(j-1)$th position in the array will have a sum with $B[j]$ that is larger than or equal to the aim. Consequently, we'll increment the count of valid pairs by $(j - i)$, then update the second pointer by decrementing it by 1. Conversely, if the condition is false, we'll move the first pointer up by 1 position.

b) CORRECTNESS: We will use loop invariant to prove the correctness of this algorithm.

- **Loop invariant:** The invariant used in this algorithm is :

$$\text{For each iteration with } i < j \text{ such that } B[i] + B[j] \geq aim \tag{1}$$

- Before the loop run, the invariant is correct because we don't have any pair i and j.
- Inside a loop we will have a conditional statement, so we will have two situations.
  - $B[i] + B[j] \geq aim$ so we will count all pair from i to j - 1. The reason for that is for all element after the index i the value B[i] will be the smallest with $B[i] + B[j] >= aim$. Thus all other pairs have to be larger than aim. That is why the invariant is still correct.
  - In the second case, we will shift the index of i by 1 because at this case $B[i] + B[j] < aim$ so we have to increase the value of B[i] to make it larger so we shift the index by 1 to the right. As it does not miscount any pair, the invariant is maintained.
  - When the loop finishes, which means i == j, it runs through all possible of (i, j) pairs, hence it counts every indices that will have the sum larger than aim.

- In conclusion: The maintainance of the invariant throughout the execution ensures that no valid pair is missed. Thus the count is accurate, leading to the right of the algorithm.

c) RUNNING TIME ALGORITHM

- All the assignments in line 2, 3, 4 will run in $O(1)$ time.
- At while loop at line 5, we can see that inside that loop, we have two condition and only one pointer will move at each turn until i and j meet together(i = j). We call n will be the length of array B, The first time the distance two pointer will be $n - 1$. However, because each iteration the distance between of them will decreased by 1, and the loop will terminate when the distance betweeen them equals to 0. Thus the running time of the algorithm will be $O(n)$. Thus the running time of this algorithm will be $O(n)$.