Data Structures Homework #3

Ari Feiglin

Question 3.1:

You are provided with a circular doubly linked list where each node can be in of two states: `On` or `Off`. Formulate algorithms for every one of the following. These algorithms can use only a single pointer and counters. Compute/prove the time complexity of each algorithm as well.

(1) If it is known that all nodes are in the `On` state, find the length of the list.

(2) Given a parameter ℓ, test if the length of the list is at most ℓ.

(3) Find the length of the list in $\mathcal{O}\left(n^2\right)$ time.

(4) Find the length of the list in $\mathcal{O}\left(n\right)$ time.

Answer:

(1) Since we know the initial state of each node, we can change the state of one of the nodes, then traverse the list until we reach a node of that (changed) state again. This node must be the node that we started on, and by keeping track of how many nodes we traversed, we can compute the length of the list. This is best expressed in pseudocode:

```
Input  : A cyclic doubly-linked list with node node
Output: The length of the list
node.turnOff();
node ← node.next;
length ← 1; // Since we've already ``moved'' one node
while node.state ≠ off do
    node ← node.next;
    length ++;
end
return length;
```

This obviously executes in $\mathcal{O}\left(n\right)$ time as it merely traverses each node a constant amount of times.

(2) What we can do here is simply set the state of a node to `Off`, and then traverse the next ℓ nodes and change each of their states to `On`. Then we check if the first node's state has changed. The node's state changes if and only if we've encountered it a second time: which can only happen if the length of the list is less than or equal to ℓ.

```
Input  : A cyclic doubly-linked list with node node and an integer $\ell$
Output: If the length of the list is less than or equal to $\ell$
node.turnOff();
node ← node.next;
$i \leftarrow 1$;
while $i < \ell$ do
    node.turnOn();
    node ← node.next;
    $i$++;
end
while $i > 0$ do
    node ← node.prev;
    $i$--;
end
return node.state == On;
```

This algorithm just iterates over the first ℓ nodes in the list twice and thus has a time complexity of $\mathcal{O}(\ell)$.

(3) Let the previous subquestion's algorithm be the function `LengthLeq`.
What we can do is iterate over the integers and check if `LengthLeq(`ℓ`)` is true. The first integer we encounter where `LengthLeq(`ℓ`)` is true will be the length of the list. This is because the list's length is not less than or equal to $n-1$ (the length is greater than $\ell - 1$) but it is less than or equal to ℓ, which means the length is ℓ.

```
Input  : A doubly-linked list with node node
Output: The length of the list
1 $\ell \leftarrow 1$;
2 while not LengthLeq(node, $\ell$) do
3     $\ell$++;
4 end
5 return $\ell$;
```

At each iterationof the loop, we are doing $\mathcal{O}(\ell)$ work, so all in all this is:

$$\sum_{i=1}^{n} i = \frac{n}{2}(n-1) \in \mathcal{O}\left(n^2\right)$$

Work.

(4) The idea here is to slightly alter the previous algorithm from the previous subquestion. Notice that if we overshoot ℓ (that is we get $\ell \geq n$), all the nodes will be turned on. So the idea is to efficiently find an $\ell \geq n$ then execute the algorithm from the first subquestion. Let that algorithm be `SimpleLength`.
So now the question remains, how do we efficiently find such an ℓ? The idea is to slightly alter the previous subquestion's algorithm. While the previous algorithm iterated over every natural number less than n, we don't need to be careful that we're less than n here. Thus we can have larger jumps. So instead of incrementing by 1 each time, we can multiply the index by some amount (for example 2) each time.

```
Input  : A doubly-linked list with node node
Output: The length of the list
1 $\ell \leftarrow 1$;
2 while not LengthLeq(node, $\ell$) do
3     $\ell \times= 2$;
4 end
  /* At this point we know that all the nodes must be on, so we can use SimpleLength.
     */
5 return SimpleLength(node);
```

We know that `SimpleLength` $\in \mathcal{O}(n)$.
And the loop will loop $\log n$ times, and each iteration will take $\mathcal{O}\left(2^i\right)$ time. So the loop will have a time complexity of:

$$\sum_{i=0}^{\log n} 2^i = \frac{2^{1+\log n} - 1}{2-1} = 2 \cdot n - 1 \in \mathcal{O}(n)$$

So the algorithm as a whole has a time complexity of:

$$\mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$$

As required.

■

Question 3.2:

3SUM is the following problem: construct a data structure where given a set of numbers $S = \{x_1, \dots, x_n\}$, it allows for the following operations:

(1) Initialization

(2) Sum-querying: given a number x, checks if there exist 3 elements in S such that $x_i + x_j + x_k = x$

Assume there exists a data structure which solves 2SUM, which initializes in $\mathcal{O}(n \log n)$ time and queries in $\mathcal{O}(f(n))$ time.
Prove there exists a data structure which solves 3SUM with an initializing time of $\mathcal{O}(n \log n)$ and a querying time of $\mathcal{O}(n \cdot f(n))$.

Answer:

The algorithm to do this is actually quite simple. Proving why it works less so.
Firstly, during the initialization we must create the infrastructure to allow us to check how many of the same number there is and their indexes in $\mathcal{O}(1)$ time (so it doesn't affect the time complexity of the rest of the algorithm).
Now, suppose we're querying for a number a.
First, we check if there are three instances of $\frac{a}{3}$. This can be done by iterating through S and looking and counting the instances of $\frac{a}{3}$. This takes $\mathcal{O}(n)$ time, and is done once, so it won't affect the time complexity of the rest of the algorithm (as $n \cdot f(n) \geq n$).
Next we iterate over S and check if there is a two sum of $a - x_i$. Suppose there is: $x_j + x_k$. This means that $x_i + x_k + x_j = a$. If two of these indexes are equal (and at most only 2 can be, as explained above), suppose $i = j$, iterate over the numbers equal to x_i and swap x_j with the first one whose index is not equal to k (and not equal to i as well of course). If such a number exists, return this new triplet. Otherwise, continue the iteration.

```
Input  : An array arr
Output: Necessary infrastructure for the Three-Sum problem
twoSum.init(arr);                        // Initialize the twoSum data structure
/* sortWithIndexes gives an array of indexes sorted by the value of arr. This can be
   achieved by sorting the array $[1, \dots, \text{arr.len}]$ by comparing values of arr $[i]$, thus
   achieving $\mathcal{O}(n \log n)$ time.                                          */
sortedArr ← arr.sortWithIndexes();
indexes ← [];
currIndexes ← [];
foreach $i$ in sortedArr do
    if currIndexes == [] or arr[$i$] ≠ arr[currIndexes[1]] then currIndexes.append($i$);
    else
        indexes.append(currIndexes);
        currIndexes ← [];
    end
end
if currIndexes ≠ [] then indexes.append(currIndexes);
```
Algorithm 1: 3-Sum Initialization

What this does is initialize a **Two-Sum** data structure and create a list **Indexes**, which is a list of indexes grouped by if their respective xs are equal. That is:

$$\textbf{Indexes} = [[i_{1,1}, \dots, i_{1,\ell_1}], \dots, [i_{n,1}, \dots, i_{n,\ell_n}]]$$

Where $x_{i_{m,1}} = \dots = x_{i_{m,\ell_m}}$.
This has a time-complexity of $\mathcal{O}(n \log n)$ since initializing the Two-Sum data structure takes $\mathcal{O}(n \log n)$ time, sorting the array takes $\mathcal{O}(n \log n)$ time, and the loop iterates over the array which takes $\mathcal{O}(n)$ time. So all in all this is $\mathcal{O}(n \log n) + \mathcal{O}(n \log n) + \mathcal{O}(n) = \mathcal{O}(n \log n)$ time, as required.

The querying algorithm is:

```
Input  : A number $a$
Output: The indexes of three numbers which sum to $a$
/* Here, count and findIndexes are functions which count and find the indexes of the
   instances of an element in an array respectively. These can be done by simply
   iterating over an array, and thus have a time complexity of $\mathcal{O}(n)$.      */
if arr.count($a$ / 3) $\geq 3$ then return arr.findIndexes($a$ / 3)[1,2,3];
foreach currIndexes in indexes do
    $i \leftarrow$ currIndexes [1];
    $i_1 \leftarrow$ currIndexes [2];
    $i_2 \leftarrow$ currIndexes [3];
    /* I'm not handling errors here, so errors are just ignored and execution is
       continued at the next line                                                     */
    if twoSum.query($a$-Arr[$i$]) $\rightarrow j, k$ then
        if $i == k$ then
            $j \leftrightarrow k$;      // Swap $j$ and $k$ so we only need to deal with the case that $i == j$.
        end
        if $i == j$ then
            /* Swap index $j$ with an $i_m$ that doesn't equal to $k$ (since $i_m \neq i$ we don't need
               to check that).                                                        */
            if $i_1 \neq k$ then return $i, i_1, k$;
            return $i, i_2, k$;
        else
            return $i, j, k$;
        end
    end
end
```

Algorithm 2: 3-Sum Querying

But does this algorithm work? Suppose we have the sum $x_{i_1} + x_{j_1} + x_{k_1} = a$, but any query does not lead to us getting these indexes.
I will denote querying the index t as:

$$x_t \mapsto x_n, x_m$$

Ie. querying t produces n and m.
Note that none of these xs can be equal. If they are, suppose $x_{i_1} = x_{j_1}$, then querying:

$$x_{i_1} \mapsto x_{i_1}, x_{i_2}$$

Since it cannot return a valid sum, so it must return the index i_1. But then x_{i_1} will be swapped with x_{j_1} (or a number equal to it) and we get the indexes i_1, j_1, i_2. But again, this cannot be a valid sum, so i_2 must be equal to j_1 (as it is returned by the query, along with x_{i_1}, it cannot be equal to i_1), then i_2 will be swapped with k_1 (or a number equal to x_{k_1}) since

$$x_{i_1} + x_{j_1} + x_{i_2} = a \implies x_{i_2} = x_{k_1}$$

So we will have a valid sum to a, in contradiction.
This means that:

$$x_{i_1} \longmapsto x_{i_1}, x_{i_2}$$

As this is the only way querying x_{i_1} does not give us a valid sum to a, and similarly:

$$x_{j_1} \longmapsto x_{j_1}, x_{j_2} \qquad x_{k_1} \longmapsto x_{k_1}, x_{k_2}$$

Notice that this means:

$$\begin{gathered}2x_{i_1} + x_{i_2} = a \\ 2x_{j_1} + x_{j_2} = a \\ 2x_{k_1} + x_{k_2} = a\end{gathered}$$

Which means that:

$$2(x_{i_1} + x_{j_1} + x_{k_1}) + x_{i_2} + x_{j_2} + x_{k_2} = 3a \implies 2a + x_{i_2} + x_{j_2} + x_{k_2} = 3a \implies x_{i_2} + x_{j_2} + x_{k_2} = a$$

And notice that these second-generation xs must all be distinct as well, since if $x_{i_2} = x_{j_2}$, then

$$a = \begin{cases} 2x_{i_1} + x_{i_2} = 2x_{i_1} + x_{j_2} \\ 2x_{j_1} + x_{j_2} \end{cases} \implies x_{i_1} = x_{j_1}$$

In contradiction.
So the sum:

$$x_{i_2} + x_{j_2} + x_{k_2} = a$$

Is a valid 3-sum. This means that querying these numbers must return indexes, but they cannot be valid.
Inductively, we see that forall m:

$$\begin{gathered}x_{i_m} \longmapsto x_{i_m}, x_{i_{m+1}} \\ x_{j_m} \longmapsto x_{j_m}, x_{j_{m+1}} \\ x_{k_m} \longmapsto x_{k_m}, x_{k_{m+1}}\end{gathered}$$

And:

$$x_{i_m} + x_{j_m} + x_{k_m} = a$$

And:

$$x_{i_m} \neq x_{j_m} \neq x_{k_m}$$

Because there are finite xs, at some point $x_{i_{m',1}} = x_{i_{m,1}}, x_{j_{m',2}} = x_{j_{m,2}}, x_{k_{m',3}} = x_{k_{m,3}}$ $(m' < m)$. This means that querying these returns the same as their m' equivalents, so for every n:

$$x_{i_{m'_1+n}} = x_{i_{m_1+n}}$$

And similar. So we can take the maximum m', and let it be m':

$$x_{i_{m'}} = x_{i_{m_1}}$$

And similar, for some m_1 (not necessarily the same m_1 as before). We can just write this as:

$$x_{i_1} = x_{i_{m_1}}$$

By changing what we choose for i_1. And so we know that:

$$x_{i_1} = x_{i_{n \cdot m_1}}$$

For every natural n, as this is just like restarting the cycle. So by setting the appropriate ns (for i, set $n = m_2 \cdot m_3$), we get:

$$x_{i_1} = x_{i_{m_1 \cdot m_2 \cdot m_3}} \qquad x_{j_1} = x_{j_{m_1 \cdot m_2 \cdot m_3}} \qquad x_{k_1} = x_{k_{m_1 \cdot m_2 \cdot m_3}}$$

Let's simplify by letting $n := m_1 \cdot m_2 \cdot m_3$, and so $x_{i_1} = x_{i_n}$, etc.
But we know that:

$$x_{i_n} = a - 2x_{i_{n-1}}$$

Opening up this reccurrence, we get:

$$x_{i_n} = a - 2a + 4a - \cdots + (-2)^{m-1}a + (-2)^n \cdot x_{i_{n-m}}$$

This is a geometric sum, so it is equal to:

$$x_{i_n} = a \cdot \frac{1-(-2)^m}{3} + (-2)^n x_{i_{n-m}}$$

Setting $m = n-1$, we get:

$$x_{i_n} = a \cdot \frac{1-(-2)^{n-1}}{3} + (-2)^{n-1} x_{i_1}$$

Since we know $x_{i_n} = x_{i_1}$, we get:

$$x_{i_1}\left(1-(-2)^{n-1}\right) = a \cdot \frac{1-(-2)^{n-1}}{3} \implies x_{i_1} = \frac{a}{3}$$

But by doing an identical process to x_{j_1}, we get the same result:

$$x_{j_1} = \frac{a}{3} = x_{i_1}$$

But we know proved that $x_{i_1} \neq x_{j_1}$, in contradiction.
So, in short: this algorithm works.
The time complexity is simply $\mathcal{O}\left(n \cdot f(n)\right)$. This is because the time complexity of the loop is $\mathcal{O}\left(n \cdot f(n)\right)$: all operations other than querying are $\mathcal{O}\left(1\right)$, so the time complexity of each iteration is $\mathcal{O}\left(f(n)\right)$, and this is repeated n times, thus $\mathcal{O}\left(n \cdot f(n)\right)$.
And the time complexity of the if statement before the loop is $\mathcal{O}\left(n\right)$. And $n \cdot f(n) \geq n$, so in total, this gives us a time complexity of $\mathcal{O}\left(n \cdot f(n)\right)$, as required.

■

Question 3.3:

Create a data structure, **Minimum-Stack**, which provides the following methods normal stack methods:

- Initialization
- If empty
- Push
- Pop

As well as another:

- Find the minimum

All methods should be done in constant time.

Answer:

The issue here is finding the minimum in constant time. To do so, we keep track of the minimum and all past minimums. We do this by keeping another stack of minimums, and whenever we push a number smaller than the current minimum, we push it onto the minimum stack. And whenever we pop the current minimum, we must pop it from the minimum stack as well.

- To initialize, we just initialize two empty stacks.

```
primaryStack = new Stack();
minimumStack = new minimumStack();
```

  Since stack initialization is $\mathcal{O}(1)$, so is this.

- To check if the stack is empty, we just check if the primary stack is empty:

```
Output: Whether or not the stack is empty
return primaryStack.isEmpty();
```

  This just calls an $\mathcal{O}(1)$ operation so it to is $\mathcal{O}(1)$.

- When pushing, we need to check if what we are pushing is less than or equal to the current minimum (the top of the minimum stack). If it is, push it onto the minimum stack.

```
Function getTop(stack)
    Input  : A stack stack
    Output: The top of the stack, without changing the stack
    top ← stack.pop();
    stack.push(top);
    return top;
end
Input  : An element $x$ to push
primaryStack.push($x$);
if minimumStack.isEmpty() or $x \leq$ getTop(minimumStack) then minimumStack.push($x$);
```

  All operations here are constant-time, so this is constant-time as well.

- To pop, all we need to do is pop from the primary stack, check if it's equal to the current minimum, and if it is, pop it from the minimum stack.

```
**Output:** The top of the stack
top ← primaryStack.pop();
if top == getTop(minimumStack) then minimumStack.pop();
return top;
```

  Again, all operations are constant-time, so so is this.

- To get the minimum, we just return the top of the minimum stack.

```
**Output:** The minimum of the stack
return getTop(minimumStack);
```

  This is also constant-time.

The purpose of the minimum stack is to create a chronology of all the minimums added (including duplicated), so the top of the minimum stack will always be the minimum and the element before it is the minimum that was the minimum before the current minimum was pushed.

■

Question 3.4:

An **Expanded Array** supports the following methods:

(1) Initialization

(2) Appendment to the end of the array

(3) Appendment to the start of the array

(4) Erasure at the end of the array

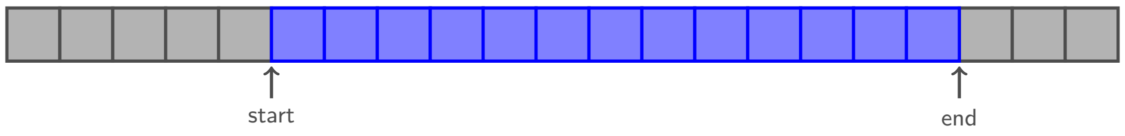
(5) Erasure at the start of the array

(6) Random-access indexing

Provide algorithms for each of them, and find the amortized cost of each algorithm using the accounting method.

Answer:

My idea for the **Expanded Array** data structure is to have a data structure in the form of:



Essentially the **Expanded Array** will consist of an allocated area of memory (the grey and blue squares), and "used" area of memory (the blue squares) which is pointed to by pointers at its start and end.
We can thus append elements to the start and end of the array by using these pointers. We can erase elements by simply moving the pointers.
In order to implement this data structure, we need the following fields:

- A pointer to the start of the allocated memory
- The current length of the allocated memory
- And pointers to the start and end of the "used" memory.

If we try to append/prepend an element to the array past the limits of the allocated memory, we simply reallocate the allocated memory to a greater length and copy the current "used" memory to the center of reallocated section. Then we can append/prepend the element.
Explicitly, here are the algorithms:

(1) We must initialize the fields explained above:

```
allocatedLength ← 2;
allocatedData ← allocate(allocatedLength);
start ← allocatedData;
end ← start+1;
```

Algorithm 1: Initializing the *Extend Array*.

This initializes an empty array of length 2, with the start pointer pointing to the first index in the array, and the end pointer is one index past.
Note that the start and end pointers will always point at the index that the next element should be appended/prepended to, meaning they point just past the actual allocated area.
Furthermore, this has time complexity $\mathcal{O}(1)$ since every operation here takes constant time.

(2) To append, we must first check if the end pointer is past the bounds of the allocated space. If not, just place the element at the index pointed to by it. Otherwise, we must reallocate the memory, and copy the current memory into it. Then we can append.
Let us first create a procedure for reallocating the memory. It will triple the size of the current memory (essentially doubling the space in both directions).

```
Output: The reallocated Extended Array.
Function ReallocateExtendedArray()
    oldStart ← start;
    pointerDiff ← end-start;
    allocatedData ← allocate($3 \times$ allocatedLength);
    start ← allocatedData + allocatedLength ;    // Set start to be one-third the way down
        the new allocated space.
    end ← start + pointerDiff;
    allocatedLength $\times=$ 3;
    /*                          Begin copying                          */
    for $i$ from 1 to pointerDiff - 1 do
        *(start + $i$)← *(oldStart + $i$);
    end
end
```

This has time complexity $\mathcal{O}(n)$ since it just iterates over the "used" area of memory.
Now, onto actually appending an element:

```
Input: An element $x$ to append to the Extended Array.
if end - allocatedData $\geq$ allocatedLength then ReallocateExtendedArray();
*end ← $x$;
end++;
```

Algorithm 2: Appending an element to the *Extended Array*.

If reallocation takes place, this has complexity $\mathcal{O}(n)$, otherwise $\mathcal{O}(1)$.

(3) For prepending an element, the procedure is much the same as appending one:

```
Input: An element $x$ to prepend to the Extended Array.
if start < allocatedData then ReallocateExtendedArray();
*start ← $x$;
start--;
```

Algorithm 3: Prepending an element to the *Extended Array*.

If no reallocation takes place, this has $\mathcal{O}(1)$ complexity. If reallocation does take place, it has $\mathcal{O}(n)$.

(4) To erase an element from the end, we just decrement the end pointer:

```
if end - 1 > start then end--;
```

Algorithm 4: Erasing an element from the end of the *Extended Array*.

This takes constant time.
We must verify that the new end pointer will still be before the start pointer, hence the if statement.

(5) Erasing an element from the beginning of the Extended Array is very similar to doing so from the end:

```
if start + 1 < end then start++;
```

Algorithm 5: Erasing an element from the beginning of the *Extended Array*.

This takes constant time.

(6) To index the Extended Array, we just dereference that index plus the start pointer (since the index 1 corresponds to the first index, we don't need to add 1).

```
Input  : An index $i$.
Output: The element at the $i$th index in the Extended Array.
1 return*(start + $i$);
```
Algorithm 6: Indexing the *Extended Array*.

This takes constant time.

I will define the amortized values of each of these algorithms to be:

$$\hat{c}_1 \coloneqq 1 \quad \hat{c}_2 \coloneqq 3 \quad \hat{c}_3 \coloneqq 3$$
$$\hat{c}_4 \coloneqq 1 \quad \hat{c}_5 \coloneqq 1 \quad \hat{c}_6 \coloneqq 1$$

For every constant operation (1,4,5,6), the amortized value is 1 simply because these operations are "closed", the "pay" for themselves.
For appendment, the value is 3 so that:

- If no reallocation takes place then the new element pays for it own appendment (which takes constant time) with the first coin, and keeps the second.
  The third coin is then given to one of the elements in the middle which don't have their own. If all the elements have their own, then it doesn't matter what we do with the coin.
- If the array has been reallocated, then every element being copied must have (at least) one coin.
  We knos this because after being copied, the coins in the previous "used" area had no coins. But as one of the sides filled up (and one had to in order to call for a reallocation), it filled up this previous "used" area with coins (there is an equal amount of new elements to fill up on one side as their are in the previous "used" area. Each new element gives one of these "used" elements a coin, so in the end they all have their own coin.)
  So before reallocation, each element has their own coin, which means they can afford for themselves to be reallocated (which takes constant time and thus costs a single coin).

So the amortized value of each operation is:

Operation	**Amortized Value**
Intialization	1
Appendment	3
Prependment	3
Post-Erasure	1
Pre-Erasure	1
Indexing	1

(Post-Erasure and Pre-Erasure means erasure from the end and start of the array respectively.)
So all in all, the amortized complexity of each of the operations is $\mathcal{O}(1)$.

■