Mathematics and Algorithms

Study Point Assignment

Zimmermann, Tobias
cph-tz11@cphbusiness.dk

March 3, 2023

Contents

1 Linear Search Algorithm (Task 1)

It is important to understand that generally when analyzing the complexity of some algorithm we usually focus on the worst case scenario of the algorithm. However, this does not mean that the best case and middle case are not important, and they should be analyzed so that the developer using your algorithm can select the best algorithm for their specific use case.

Time complexity analysis of the code block:

```
1 pub fn arr_includes(arr: Vec<String>, test: String) {
2     for ele in arr {
3         if ele == test {
4             true
5         }
6     }
7     false
8 }
```

We are given the task of analyzing a function so that we can prove that it is indeed a linear search algorithm. An algorithm is linear if the complexity of an algorithm increases linearly with the size of n

Lets go through the code block, and understand why this algorithm is linear.

- Line 1: We are given an array of size n, because we don't know it's size, and a string which is the one we are searching for.
- Line 2: We iterate through the array. $(O(n))$
- Line 3-5: Check if the string we want to find is the current element in the array, and return true if that was the case. $(O(1))$
- Line 7: Return false in the end because no match was found. $(O(1))$

So to sum up. We go through the array, if a match is found, we return early otherwise we keep going through the whole array.

From what was discussed earlier, we know that there are three types of time complexities we should take into account.

1. Worst-case: $O(n)$ because the highest complexity found in the code block analysis is O(n).
2. Best-case: $O(1)$ because it is possible that the first element in the array is the one we are searching for, in which case we return early.

3. Average-case: $O(n)$ because if each say that each element in the array has the same likelihood of being searched, then the average case can be said to be $\frac{n+1}{2}$ which in Big O sense should be analyzed as just n because we take away the constants.

2 Bubble Sort (Task 2)

Time complexity analysis of the code block:

```
1  pub fn bubble_sort(arr: &mut Vec<usize>) {
2      for i in 0..arr.len() {
3          for j in i..arr.len() {
4              if arr[i] > arr[j] {
5                  let i_val = arr[i];
6                  arr[i] = arr[j];
7                  arr[j] = i_val;
8              }
9          }
10     }
11 }
```

We are given the task of analyzing the time complexity of the bubble sort sorting algorithm.

Lets go through the code block, analyzing each operation happening.

- Line 1: We are given a mutable array. This tells us that we don't have to allocate a new array, but instead we should be mutating the provided array. (More related to space complexity)
- Line 2: We iterate through the array $O(n)$
- Line 3: Inside the same array, we iterate through the array again, starting from where our outer loop is currently at. $O(n)$
- Line 4-8: Swap i with j if $i > j$, keeping the smallest numbers in the front. $O(1)$

From this we can gather information about the different types of time complexities for our algorithm:

1. Worst-case: $O(n^2)$ Because we are going through the same array one time and the ones more **inside** the same array.
2. Best-case: $O(n^2)$ Because no matter what, we still have to go through the whole array twice. There are variations of this algorithm, where can be made to have a best of $O(n)$ when the provided array is already sorted.
3. Average-case: $O(n^2)$ because if the both the upper bound and lower bound of our algorithm is the same, then the average must be the same as well.

3 Queue (Task 4)

Time complexity analysis of the code block:

```
const SIZE: usize = 10;

pub struct CircularQueue<T>
where
    T: Copy,
{
    front: usize,
    rear: usize,
    items: [Option<T>; SIZE],
    size: usize,
}

impl<T> CircularQueue<T>
where
    T: Copy,
{
    pub fn new() -> Self {
        Self {
            items: [None; SIZE],
            size: 0,
            front: 0,
            rear: 0,
        }
    }

    pub fn enqueue(&mut self, item: T) -> bool {
        if self.size == SIZE {
            return false;
        }
        self.items[self.rear] = Some(item);
        if self.rear == SIZE - 1 {
            self.rear = 0;
        } else {
            self.rear += 1;
        }
        self.size += 1;
        true
    }

    pub fn dequeue(&mut self) -> Option<T> {
        if self.size == 0 {
```

```
42             return None;
43         }
44         let item = self.items[self.front];
45         self.items[self.front] = None;
46         self.front += 1;
47         self.size -= 1;
48         item
49     }
50 
51     pub fn peek(&self) -> Option<T> {
52         self.items[self.front]
53     }
54 }
```

We are given the task of analyzing the time complexity of a queue.

There are many different types of queues, in my example i choose to implement a circular queue. A circular queue basically means that your rear pointer, can circle back to behind the front.

Lets go through the methods **enqueue**, **dequeue**, and **peek** respectively and understand their time complexity.

3.0.1 Enqueue

- Line 26: We are given an item for us to put at the end out our queue, and the methods returns boolean for whether or not the item could be stored in the queue.
- Line 27-29: Check if we can fit another item $O(1)$
- Line 30: Add the item to the end of our queue. $O(1)$
- Line 31-35: Set the new rear $O(1)$
- Line 36-37: Increase the size, and return true $O(1)$

Because we only have $O(1)$ operations the time complexity of enqueue is $O(1)$.

3.0.2 Dequeue

- Line 41-43: Check if there are items in the queue. $O(1)$
- Line 44: Get the next item in the queue. $O(1)$
- Line 45-48: Remove the the item from the queue, update the pointers, and return our item $O(1)$

Because we only have $O(1)$ operations the time complexity of dequeue is $O(1)$.

3.0.3 Peek

- Line 52: return the next item in the queue. $O(1)$

Because we only have $O(1)$ operations the time complexity of peek is $O(1)$.

3.0.4 Data structure complexity

So from gathering information about each of the methods that have been implemented, we can gather that the time complexity of our circular queue is $O(1)$ because that is the average of all our previous findings.