Rushil Gupta September 25, 2024

CS-1204 Problem Set 1

Collaborators: *none*

Problem 1

```
x = S.pop()
if S.top() > x:
    x = S.pop()

# assert: x is the largest element in S with probability 2/3
```

Proof: We know that we have a total of 3 integers in the stack. Since we do not know anything about them, we can claim that all 3 integers are equally likely to be the largest. Thus, the probability of the largest integer being at the top of the stack is $\frac{1}{3}$. Now, in the algorithm, we are comparing the first 2 integers, and use x as a proxy to compare. So, we know that x stores the maximum of the first 2 integers with probability $\frac{2}{3}$.

Problem 2

1. If $d(n)$ is $O(f(n))$, then $ad(n)$ is $O(f(n))$ for any constant $a > 0$.

   By definition, $d(n)$ is $O(f(n)) \iff \exists c > 0, n_0 \geq 0$ s.t. $\forall n \geq n_0, 0 \leq d(n) \leq cf(n)$.

   My multiplying both sides by a, we get $0 \leq ad(n) \leq acf(n)$. Thus, $ad(n)$ is $O(f(n))$.

2. If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n) + e(n)$ is $O(f(n) + g(n))$.

   By definition, $d(n)$ is $O(f(n)) \iff \exists c_1 > 0, n_1 \geq 0$ s.t. $\forall n \geq n_1, 0 \leq d(n) \leq c_1 f(n)$.
   Similarly, $e(n)$ is $O(g(n)) \iff \exists c_2 > 0, n_2 \geq 0$ s.t. $\forall n \geq n_2, 0 \leq e(n) \leq c_2 g(n)$.

   Then, take $n = \max(n_1, n_2)$ and $c = c_1 + c_2$. We have:

   $$\begin{aligned} 0 \leq d(n) + e(n) &\leq c_1 f(n) + c_2 g(n) \\ &\leq (c_1 + c_2)(f(n) + g(n)) \\ &\leq c(f(n) + g(n)) \end{aligned}$$

   This, by definition, means that d(n) + e(n) is O(f(n) + g(n)).

3. If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n)e(n)$ is $O(f(n)g(n))$.

   By definition, $d(n)$ is $O(f(n)) \iff \exists c_1 > 0, n_1 \geq 0$ s.t. $\forall n \geq n_1, 0 \leq d(n) \leq c_1 f(n)$.
   Similarly, $e(n)$ is $O(g(n)) \iff \exists c_2 > 0, n_2 \geq 0$ s.t. $\forall n \geq n_2, 0 \leq e(n) \leq c_2 g(n)$.

   Then, take $n = \max(n_1, n_2)$ and $c = c_1 c_2$. We have:

$$\begin{aligned} 0 \leq d(n)e(n) &\leq c_1 f(n) c_2 g(n) \\ &\leq c_1 c_2 f(n) g(n) \\ &\leq c(f(n)g(n)) \end{aligned}$$

   This, by definition, means that d(n)e(n) is O(f(n)g(n)).

4. If $d(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$, then $d(n)$ is $O(g(n))$.

   By definition, $d(n)$ is $O(f(n)) \iff \exists c_1 > 0, n_1 \geq 0$ s.t. $\forall n \geq n_1, 0 \leq d(n) \leq c_1 f(n)$.
   Similarly, $f(n)$ is $O(g(n)) \iff \exists c_2 > 0, n_2 \geq 0$ s.t. $\forall n \geq n_2, 0 \leq f(n) \leq c_2 g(n)$.

   By taking the 2 inequalities, we get:

$$\begin{aligned} 0 \leq d(n) &\leq c_1 f(n) \\ &\leq c_1 c_2 g(n) \\ &\leq c(g(n)) \quad \forall n \geq \max(n_1, n_2) \end{aligned}$$

   Where, $c = c_1 c_2$. This, by definition, means that d(n) is O(g(n)).

Problem 3

This is assuming that we want to check the 3 types of brackets: (), , []. The assignment has a small typo, hence this clarification is being made.

```
function check_parenthesis(string s):
    stack S

    for (int i = 0; i < s.length(); i++):
        char c = s[i]
        if (c == '(' || c == '{' || c == '['):
            S.push(c)
        else if (c == ')' || c == '}' || c == ']'):
            if (S.empty()):
                # Assert: got closing bracket without any opening bracket
                return false

            char top = S.pop()
            if ((c == ')' && top != '(') || (c == '}' && top != '{') || (c == ']' &&
                # Assert: got bracket mismatch
                return false

    # Assert: S is empty <=> All brackets were popped
    return S.empty()
```

The code works by looping over all the characters of the string. Whenever we see an opening bracket, we push it onto the stack (this is because we can ignore all characters that are not brackets).

Whenever we see a closing bracket, we pop the top of the stack and check if the popped element is the corresponding opening bracket of the closing bracket. If it is not, we return false since we have a mismatch.

The time complexity of this code can be understood by the recurrence relation below:

$$T(n) = T(n-1) + O(1) \quad \text{(Since we are doing constant work at each step)}$$

Solving this, we get $T(n) = O(n)$.

Problem 4

(a) Before writing the recurrence relation, it is essential to breifly explain the algorithm, which is as follows:

- Given an array of scores, we iterate over it once to score the frequence scores in an array, where arr[i] stores the frequency of score i.
- Know, leveraging the fact that the median is at exactly n/2, we can simply find then the sum of the frequencies of the scores until we reach n/2.
- This is the score that is the median (+ constant time to account for cases where n is even).

The recurrence relation for this algorithm can be written as:

$$T(n) = T(n-1) + O(1) \quad \text{(We are just incrementing arr[m] by 1, so constant work)}$$

Solving this, we get $T(n) = O(n)$.

Now, for finding the median, the worst case it searching for $n/2$ elements (when everyone has a different score). Thus, the time complexity of finding the median is $O(n)$.

This means the time complexity of finding the median score in a particular section is $O(n)$.

(b) Code submitted on gradescope

Problem 5

(a) Code submitted on gradescope

(b) **Claim:** For n disks, it takes a minimum of $2^n - 1$ moves to solve the Tower of Hanoi problem.

Proof: We can prove this by induction. For the base case, $n = 1$, we know that it takes 1 move to 1 disk from A to C. Thus, the base case holds.

Now, for the inductive step, assume that the claim holds for some $n = k$, i.e., for k disks, it takes $2^k - 1$ moves. We need to prove that it holds for $n = k + 1$.

When $n = k - 1$, we can do the following steps:

- Move the top k disks from A to B using $2^k - 1$ moves (by the inductive hypothesis).
- Move the bottom disk from A to C using 1 move.
- Move the k disks from B to C using $2^k - 1$ moves (by the inductive hypothesis).

This takes us a total of $2(2^k - 1) + 1 = 2^{k+1} - 1$ moves. Thus, the claim holds for $n = k + 1$.

Therefore, by induction, the claim that for n disks, it takes a minimum of $2^n - 1$ moves to solve the Tower of Hanoi problem holds. Now, we will move on to show how the code maintains the invariants.

Firstly, we only pop from the stack when either: we are considering a problem of size 1, or we have solved the subproblem of size $n - 1$. This ensures that we are always popping the correct disk.

Moreover, since we first solve the smaller problem, the rule that we cannot place a larger disk on top of a smaller disk is never violated (because for a sub-problem of size $k < n$, the larger disk is equivalent to not having a disk at all since the large disk is never popped).

(c) As we saw earlier, it takes $2^n - 1$ moves to solve the Tower of Hanoi problem. More formally, in terms of defining a recurrence relation, we have to solve problems of sizes $n - 1$, 1, and $n - 1$. Thus, the recurrence relation can be written as:

$$T(n) = 2T(n - 1) + 1$$

Solving this, we get $T(n) \in O(2^n)$.