Homework 2 Solutions

Yusu Qian

7/14/2019

Exercise 1 *Substitution Method (4 pts)*

Consider the recurrence relation

$$T(n) = \begin{cases} T(\lceil \frac{n}{2} \rceil) + 2T(\lceil \frac{n}{4} \rceil) + n & \text{for } n > 4 \\ 1 & \text{for } n \leq 4. \end{cases}$$

Prove using the substitution method (i.e., by induction) that $T(n) = O(n \log n)$. If it helps, you may assume that n is a power of 2.

Solution 1

We guess the solution is $T(n) = O(n \log n)$. Now we begin the induction process to prove our guess is true. We assume that for values smaller than n $T(n) \leq cnlogn$.

$T(n) = T([\frac{n}{2}]) + 2T([\frac{n}{4}]) + n$
$= c * \frac{n}{2} + 2 * c\frac{n}{4} \log \frac{n}{4} + n$
$= \frac{cn}{2}[\log n - 1 + \log n - 2] + n$
$= cn \log n - \frac{3cn}{2} + n$
$\leq cn * \log n (\text{for c>= } \frac{2}{3})$

Now we show the basecase. $T(1) > c * 1 * \log 1 = 0$, so we revise our induction and only prove the statement for $n >= 2$.

Exercise 2 *Randomized Algorithms (6 pts)*

You have just purchased the contents of an abandoned storage locker, which is full of old bottles of soda. The labels are too worn and yellowed to read, so the only way to tell if two bottles are the same brand of soda is to do a side-by-side taste test. You are told that one particular brand is in the majority (more than half the bottles are of that brand) and you want to identify a bottle of the majority brand.

Below are three algorithms for finding a majority soda from a list of bottles L, each of which uses isMajority as a sub-routine; isMajority operates by running pairwise taste tests, i.e., calling tastesTheSame which counts as a single operation. For each of the three algorithms, give the best possible big-O bound on its worst-case running time (worst-case input, worst-case randomness) and expected running time (worst-case

input, random randomness). Justify your answers.

Algorithm 1: findMajoritySoda1(L)

```
while true do
    pick a random s in L
    if isMajority(s,L) then
        return s;
```

Algorithm 2: findMajoritySoda2(L)

```
for s in L do
    if isMajority(s,L) then
        return s;
```

Algorithm 3: findMajoritySoda3(L)

```
put the list of sodas L in a random order in O(n) time
for s in L do
    if isMajority(s,L) then
        return s
```

Algorithm 4: isMajority(s,L)

```
count = 0
for t in L do
    if tastesTheSame(s,t) then
        count += 1
if count >len(L)/2 then
    return true
else
    return false
```

Solution 2

Algorithm 1:
The worst case is it never finds the majority. O(infinite). The expected time is n * $[\frac{1}{2} * 1 + \frac{1}{2} * \frac{1}{2} * 2 + \frac{1}{2} * \frac{1}{2} * \frac{1}{2} * 3 + ...]$ = infinite. O(infinite).

Algorithm 2:
The worst case is all the bottles that do not belong to the majority are stored in the front. In this case, The majority will be found on the $(\frac{n}{2})^{th}$ attempt, $O(\frac{n}{2}) * n = O(n^2)$. The expected time is the same as the worst case, as this algorithm doesn't introduce randomness.

Algorithm 3:
The worst case after randomly reorder is the same as the worst case in alagorithm 2, thus $O(n^2) + O(n) = O(n^2)$. The expected time is $n * [\frac{1}{2} * 1 + \frac{1}{2} * \frac{1}{2} * 2 + \frac{1}{2} * \frac{1}{2} * \frac{1}{2} * 3 + ... + (\frac{1}{2})^{\frac{n}{2}} * \frac{n}{2}] \leq 2n$. Thus, O(n).

Exercise 3 *Birthdays (6 pts)*

Suppose you've traveled to a parallel-universe-version of Earth with strange birthday-related rituals. Your parallel-universe classmates in CS 161 give you an *unordered* list of n pairs of the form `(month, # birthdays in that month)`, but the months all have weird names, so your list might look like

`[(Novemuary, 2), (Febtober, 3), (Japril, 1), (Febtober, 1)]`.

(If a month appears multiple times in the list, then the total number of birthdays in that month is the sum of the number recorded in each appearance, so in the example above, there are 4 total birthdays in Febtober.) You need to calculate in what month the k'th birthday occurs, so you can carry out the appropriate k'th-birthday-month rituals.

(a) Fortunately, you have a calendar so you know all the month names and their order. Design a deterministic $O(n)$ time algorithm to find the month in which the k'th birthday occurs, using an existing *linear-time sorting algorithm* as a sub-routine. Give pseudocode, a brief English description of your algorithm, and a justification that the runtime is $O(n)$.

(b) Oh no, you lost your calendar! You don't remember how many months there are or their order, but you can ask your classmates questions of the form "Is [month A] before [month B]?" Design a (deterministic or randomized) algorithm to find the month in which the k'th birthday occurs that only requires you to ask $O(n)$ questions of your classmates in expectation, using a modified version of the *select algorithm*. (You may assume that the each month appears at most once in the list you are given, but you may *not* assume that the number of months is a constant.) Give pseudocode, a brief English description of your algorithm, and a justification that the expected number of questions is $O(n)$.

Solution 3

(a) The existing linear-time sorting algorithm I use is the bucket sort algorithm. The operation Bucketsort(birthday) puts the birthday into the correct month bucket.
Pseudocode:

```
def Bucketsort(birthday):
    for month in calendar:
        if birthday.month == month:
            month.append(birthday)
def findmonth(k, birthdays,calendar):
    for birthday in birthdays:
        Bucketsort(birthday)
    for month in calendar:
        month.count = sum(month.birthday)
     accumulate = 0
     for month in calendar:
        if accumulate + month.count < k:
            accumulate += month.count
        else:
            return month
```

(b)
```
def FindMonth2(k,birthdays):
    def FindFirstMonth(birthdays):
        FirstMonth = birthdays[0].month
        for birthday in birthdays:
            if birthday.month is before FirstMonth:
                FirstMonth = birthday.month
                FirstMonthCount = birthday.count
        return FirstMonth, FirstMonthCount
    accumulate = FindFirstMonth(birthdays)[1]
    FirstMonth = FindFirstMonth(birthdays)[0]
    while k<accumulate:
        birthdays -= birthdays[birthday.month == FirstMonth]
```

```
        accumulate += FindFirstMonth(birthdays)[1]
        LastMonth = FindFirstMonth(birthdays)[0]
    return LastMonth
```

explanation: the sub algorithm FindFirstMonth finds the first month in a list of months that haven't been considered yet. Then we start from the first 'first month' we find. The number of birthdays it contains is our initial 'accumulate' value, where we store how many birthdays we have accumulated so far. If the number of birthdays it contains is less than k, then we go ahead, remove the first month from the list of birthdays, find the new 'first month' in the new list, and add the number of birthdays it contains to 'accumulate'. Repeat these steps and we will find the k^{th} birthday by asking fewer than $k * n$ questions, as the first time we find the 'first month' we need to ask $(n-1)$ questions, the second time we ask $(n-2)$ questions... and the k^{th} time we ask $(n-k)$ questions. Thus, the expected number of questions is $O(n)$.

Exercise 4 *Sorting Seagulls (6 pts)*

Suppose that n seagulls are standing in a line. Each seagull has a political leaning: left, right, or center. You'd like to sort the seagulls so that all the left-leaning ones are on the left, the right-leaning ones are on the right, and the centrist seagulls are in the middle. You can only do two sorts of operations on the seagulls:

- POLL(i): Ask the seagull in position i about its political leanings.
- SWAP(i, j): Swap the seagulls in positions i and j.

However, in order to do either operation, you need to pay the seagulls to co-operate: each operation costs one stale hot dog bun. Also, you didn't bring a piece of paper or a pencil, so you can't write anything down and have to rely on your memory. Like many humans, you can remember up to seven integers between 0 and n at a time.[1]

(a) Design an algorithm to sort the seagulls which costs $O(n)$ stale hot dog buns, and uses no memory other than storing at most seven integers between 0 and n. Give pseudocode and a brief English description of your algorithm.

(b) Give an (informal) justification why your algorithm is correct, uses only $O(n)$ stale hot dog buns, and uses no memory other than storing at most seven integers between 0 and $n-1$.

Solution 4

(a)
```
def SortLeaning(seagulls):
    head = 0
    tail = n-1
    for i in range(head,tail):
        leaning = Poll(i)
        if leaning == left:
            swap(head,i)
            head += 1
            i += 1
        elseif leaning == right:
            swap(tail,i)
            tail -= 1
        else:
            i += 1
```

(b) In the above algorithm, when the i^{th} seagull answers left or middle, i increases by 1, when it

[1] See https://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two

answers right, the queue of seagulls of interest shortens by one. Thus, we have a maximum of n steps to take. Asking the seagulls about their leaning cost n hot dog buns and swapping them cost n hot dog buns too. Thus we use O(n) hot dog buns, and only need to memorize three integers(head and tail positions, and i position, all between 0 and n).

Exercise 5 *Measuring Moas (6 pts)*

Suppose that n moas are standing in a line, ordered from shortest to tallest. You have a stick of a certain height, and you would like to identify a moa which is the same height as the stick, or else report that there is no such moa. The only operation you are allowed to do is COMPARETOSTICK(j), which returns `taller` if the j'th moa is taller than the stick, `shorter` if the j'th moa is shorter than the stick, and `same` if the j'th moa is the same height as the stick. As in Exercise 4, you forgot to bring a piece of paper, so you can only store up to seven integers in $\{0, 1, \ldots, n\}$ at a time. And, as in Exercise 4, you have to pay a moa one stale hot dog bun in order to compare it to the stick.

(a) Give an algorithm which either finds a moa the same height as the stick, or else returns "No such moa", which uses $O(\log n)$ stale hot dog buns. Give pseudocode and a brief English description of your algorithm.

(b) Prove that any (deterministic) algorithm in this model of computation must use $\Omega(\log n)$ stale hot dog buns.

Solution 5

(a)
```
def findmoa(array):
    L=1
    U=n
    while L < U:
        M = (L+U)/2
        result1 = comparetostick(array(M))
        if result1 = same:
            return M
        elseif result = taller:
            U = M
        elseif result = shorter:
            L = M
    return "no such moa"
```

(b) The initial array has a length of n, in our case, we have n moas. Each step reduces the length of the array by half. Thus, to get to the bottom level, we make $\log n$ steps. In each step there are $\Omega(1)$ operations, thus, we must use $\Omega(\log n)$ stale hot dog buns.

Exercise 6 *Amortized Analysis (4 pts)*

The rare dododo bird can only thrive when living in a flock whose size is a power of 2. For example, they are happy to live on their own, in pairs, or in groups of 4, but NOT flocks of 3. On your dododo farm, you have k enclosures each of which can hold one flock of dododo birds. Because the birds are so rare, you can only acquire them one at a time, so you have to move them around a lot to make sure every flock always has size a power of 2! Each time you move a dododo bird to a new enclosure (including when you first acquire a new bird), you have to bribe it with one stale hot dog bun. You can move multiple birds simultaneously.

Suppose your enclosures are numbered $\{0, 1, \ldots, k-1\}$, and consider the following algorithm for adding

a new dododo bird.

Algorithm 5: addBird()

i = the smallest index such that enclosure i is empty
move all the birds from enclosures $0, 1, \ldots, i-1$, and the new bird, to enclosure i

(a) Give an (informal) justification that addBird only creates flocks whose sizes are powers of 2, when starting from k empty enclosures and adding fewer than 2^k birds total.

(b) Prove that addBird has an amortized cost of $O(k)$ stale hot dog buns, when called $n < 2^k$ times starting from k empty enclosures.

(c) **(extra credit—not required!!)** Can you do better? Either prove that any way of adding $n = 2^k - 1$ birds one-at-a-time requires $\Omega(nk)$ stale hot dog buns, or give an algorithm that doesn't! The teaching team can't help you with this one because (1) it's extra credit and (2) we don't know the answer.

Solution 6

(a) In this solution, $k_i = i$. Starting from the first enclosure($k = 0$). We can see that the maximum number of dododo birds the first enclosure can hold is 1, which equals to 2^{k_0}. The second enclosure can hold a maximum of 2 birds, which equals to $2^{k_0} + 1 = 2^{k_1}$. The reason is that when the second enclosure receives the first dododo bird from the first enclosure, it also receives a new bird. When the first enclosure fills up with one dododo bird again, both these first two enclosures need to transfer their dododo birds to the third enclosure. We can easily see that the third enclosure holds $2^{k_1} + 2^{k_0} + 1$ dododo birds or none. The pattern is, a new enclosure will only be filled up when all the enclosures before it have been filled up, thus the k-th enclosure holds either none or $1 + 2^{k_0} + 2^{k_1} + ... + 2^{k-2} = 2^{k-1}$ dododo birds. In conclusion, all the flocks created are powers of 2.

(b) When called $n < 2^k - 1$ times, the dododo birds are stores in every enclosure before the k^{th} enclosure. To achieve this result, each dododo bird has been moved for at least $k - 1$ times. Plus when it was first added into the enclosures, the maximum operations an individual dododo bird has been given are k. Thus, the amortized cost is $\frac{n*k}{n} = k$. Thus addBird has an amortized cost of $O(k)$ stale hot dog buns, when called $n < 2^k$ times starting from k empty enclosures.