CS101 Algorithms and Data Structures

Fall 2019

Homework 10

Due date: 23:59, December 1st, 2019

1. Please write your solutions in English.
2. Submit your solutions to gradescope.com.
3. Set your FULL Name to your Chinese name and your STUDENT ID correctly in Account Settings.
4. If you want to submit a handwritten version, scan it clearly. Camscanner is recommended.
5. When submitting, match your solutions to the according problem numbers correctly.
6. No late submission will be accepted.
7. Violations to any of above may result in zero score.
8. In this homework, all the algorithm design part need the four part proof. The demand is in the next page. If you do not use the four part proof, you will not get any point.
9. In the algorithm design problem, you should design the correct algorithm whose running time is equal or smaller than the correct answer. If it's larger than the correct answer, you cannot get any point.

Demand of the Algorithm Design

All of your algorithm should need the four-part solution, this will help us to score your algorithm. You should include **main idea, pseudocode, proof of correctness and run time analysis.** The detail is as below:

1. The **main idea** of your algorithm. This should be short and concise, at most one paragraph— just a few sentences. It does not need to give all the details of your solution or why it is correct. It should be enough that if you were to share it with a classmate, it would be a total giveaway to the problem. This is the single most important part of your solution. If you do a good job here, the readers are more likely to be forgiving of small errors elsewhere.

2. The **pseudocode** for your algorithm. The purpose of pseudocode is to communicate concisely and clearly, so think about how to write your pseudocode to convey the idea to the reader. Note that pseudocode is meant to be written at a high level of abstraction. Executable code is not acceptable, as it is usually too detailed. Providing us with working C code or Java code is not acceptable. The sole purpose of pseudocode is to make it easy for the reader to follow along. Therefore, pseudocode should be presented at a higher level than source code (source code must be fit for computer consumption; pseudocode need not). Pseudocode can use standard data structures. For instance, pseudocode might refer to a set S, and in pseudocode you can write things like "add element x to set S." That would be unacceptable in source code; in source code, you would need to specify things like the structure of the linked list or hashtable used to store S, whereas pseudocode abstracts away from those implementation details. As another example, pseudocode might include a step like "for each edge $(u, v) \in E$", without specifying the details of how to perform the iteration.

3. A **proof of correctness**. You must prove that your algorithm work correctly, no matter what input is chosen. For iterative or recursive algorithms, often a useful approach is to find an invariant. A loop invariant needs to satisfy three properties: (1) it must be true before the first iteration of the loop; (2) if it is true before the ith iteration of the loop, it must be true before the $i + 1$st iteration of the loop; (3) if it is true after the last iteration of the loop, it must follow that the output of your algorithm is correct. You need to prove each of these three properties holds. Most importantly, you must specify your invariant precisely and clearly. If you invoke an algorithm that was proven correct in class, you don't need to re-prove its correctness.

4. The asymptotic **running time** of your algorithm, stated using $O(\cdot)$ notation. And you should have your **running time analysis**, i.e., the justification for why your algorithm's running time is as you claimed. Often this can be stated in a few sentences (e.g.: "the loop performs $|E|$ iterations; in each iteration, we do $O(1)$ Find and Union operations; each Find and Union operation takes $O(\log |V|)$ time; so the total running time is $O(|E| \log |V|)$"). Alternatively, this might involve showing a recurrence that characterizes the algorithm's running time and then solving the recurrence.

0. Four Part Proof Example

Given a sorted array A of n (possibly negative) distinct integers, you want to find out whether there is an index i for which $A[i] = i$. Devise a divide-and-conquer algorithm that runs in $O(\log n)$ time.

Main idea:

To find the i, we use binary search, first we get the middle element of the list, if the middle of the element is k, then get the i. Or we seperate the list from middle and get the front list and the back list. If the middle element is smaller than k, we repeat the same method in the back list. And if the middle element is bigger than k, we repeat the same method in the front list. Until we cannot get the front or the back list we can say we cannot find it.

Pseudocode:

Algorithm 1 Binary Search(A)

```
low ← 0
high ← n − 1
while low < high do
  mid ← (low + high)/2
  if (k == A[mid]) then
    return mid
  else if k > A[mid] then
    low ← mid + 1
  else
    high ← mid − 1
  end if
end while
return -1
```

Proof of Correctness:

Since the list is sorted, and if the middle is k, then we find it. If the middle is less than k, then all the element in the front list is less than k, so we just look for the k in the back list. Also, if the middle is greater than k, then all the element in the back list is greater than k, so we just look for the k in the front list. And when there is no back list and front list, we can said the k is not in the list, since every time we abandon the items that must not be k. And otherwise, we can find it.

Running time analysis:

The running time is $\Theta(\log n)$.

Since every iteration we give up half of the list. So the number of iteration is $\log_2 n = \Theta(\log n)$.

1. (★ 5')The special matrix

Let's define a special matrix as H_k, and these matrix satisfy the follow properties:

1. $H_0 = [1]$
2. For $k > 0$, H_k is a $2^k \times 2^k$ matrix.

$$H_k = \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right]$$

(a)Suppose that

$$v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

is a column vector of length $n = 2^k$. v_1 and v_2 are the top and bottom half of the vector, respectively. Therefore, they are each vectors of length $\frac{n}{2} = 2^{k-1}$. Write the matrix-vector product $H_k v$ in terms of H_{k-1}, v_1, and v_2 (note that H_{k-1} is a matrix of dimension $\frac{n}{2} \times \frac{n}{2}$, or $2^{k-1} \times 2^{k-1}$). Since H_k is a $n \times n$ matrix, and v is a vector of length n, the result will be a vector of length n.

$$H_k v = \begin{bmatrix} H_{k-1}(v_1 + v_2) \\ H_{k-1}(v_1 - v_2) \end{bmatrix}$$

(b) Use your results from (a) to come up with a divide-and-conquer algorithm to calculate the matrix-vector product $H_k v$, and show that it can be calculated using $O(n \log n)$ operations. Assume that all the numbers involved are small enough that basic arithmetic operations like addition and multiplication take unit time. You do not need to prove correctness.

If

$$v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \quad H_k v = \begin{bmatrix} H_{k-1}(v_1 + v_2) \\ H_{k-1}(v_1 - v_2) \end{bmatrix}$$

So we can split $H_k v$ into two small operation.

And let the operation $H_k v$ use as $T(n)$. We just need to compute $H_{k-1}v_1$ and $H_{k-1}v_2$, both of it takes $T(n/2)$ operations. And we need another two add and sub of these two matrix. And both top half add and bottom half sub takes $n/2$ steps.

And we can do the same operation to $H_{k-1}v_1$ and $H_{k-1}v_2$

So $T(n) = 2T(n/2) + n$

And using the Master Therom, $a = 2, b = 2, d = 1$, so $d = \log_b a$.

$T(n) = O(n \log n)$

2. (★★★ 10')Majority Elements

An array $A[1 \dots n]$ is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is $A[i] > A[j]$?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is $A[i] = A[j]$?" in constant time. Four part solutions are required for each part below.

(a) Show how to solve this problem in $O(n \log n)$ time.

Main idea:

Split the array into two sub-array with almost same length. And then doing recursion to get the majority of the two subsets. Then try to determined whether the majority of subsets is the majority of the whole array.

Psuedocode:

Algorithm 2 *majority*$(a[0, 1, \cdots, n])$

```
if n == 0 then
  return a[0]
else
  left ← majority(a[0, 1, ··· , n/2])
  right ← majority(a[n/2 + 1, ··· , n])
  if left and right and left == right then
    return left
  else
    leftCount ← the number of elements in a equals to left
    rightCount ← the number of elements in a equals to right
    if leftCount > ⌊(n + 1)/2⌋ then
      return left
    else if rightCount > ⌊(n + 1)/2⌋ then
      return right
    else
      return None
    end if
  end if
end if
```

Proof of Correctness:

In each recursion, we do it like this. The recursion ends when the list only have one element. If the array's left and right sub-array both have majority, and they are the same. We can get that the majority of the array is this. And if not, we just scan the whole array to get if the left or right majority has greater than $n/2$ elements in the array. If it has, then it's the majority of the array. If not, we cannot find the majority.

Runtime analysis:

In every recursion, we split the array into two sub array. And scan the array for at most two times. So

$T(n) = 2T(n/2) + O(n)$

$T(n) = O(n \log n)$

(b) Can you give a linear time algorithm? (You should not reuse the algorithm to answer part a)

Main Idea:

Firstly, we can find an element may be the majority. We scan the array get if the element is same as the majority then count adds 1, else count subs 1. And if count equals 0, we just select majority as the next element. And then, after scan the last element, we check whether this majority shows more than $n/2$ times.

Psuedocode:

Algorithm 3 $majority(a[0, 1, \cdots, n])$

```
count1 ← 0, count2 ← 0, majority
for i = 0 to n do
  if count1 == 0 then
    majority ← a[i]
  else if a[i] == majority then
    count1 ← count1 + 1
  else
    count1 ← count1 − 1
  end if
end for
for i = 0 to n do
  if a[i] == majority then
    count2+ = 1
  end if
end for
if count > n/2 then
  return majority
else
  return None
end if
```

Proof of Correctness:

Firstly, the majority of the array occurs more than $n/2$ times. So if the $count1$ is greater than 1, it has the probability of being majority, since it may occurs more than any other elements.

And then we can scan the array to get whether it's the majority. If the count is larger than $n/2$, it's majority. Otherwise, this array do not have majority.

Runtime Analysis:

We have done two scan for the array. So $T(n) = 2n = O(n)$

3. (★★★ 5')Find the missing integer

An array A of length N contains all the integers from 0 to N except one (in some random order). In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the jth bit of $A[i]$". Using only this operation to access A, give an algorithm that determines the missing integer by looking at only $O(N)$ bits. (Note that there are $O(NlogN)$ bits total in A, so we can't even look at all the bits). Assume the numbers are in bit representation with leading 0s.

Main Idea:

Look at least significant bits and compare the 0s and 1s. Discard the numbers whose least significant bit is of the larger set. The bit of the missing number at this position will be the bit of the smaller set. Recursively apply the algorithm and build the missing number at each bit position.

Psuedocode:

Algorithm 4 $FindMissing(A)$

```
return FindMissingNum(A, 0)
```

Algorithm 5 $FindMissingNum(A, m)$

```
if length[A] = 0 then
    return m
end if
B ← value of A with LSB of 0
C ← value of A with LSB of 1
if length [B] ≤ length[A] then
    B ← B with LSB of all numbers removed
    m ← m with 0 prepend to least significant bit
    return FindMissingNum(B, m).
else
    C ← C with LSB of all numbers removed
    m ← m with 1 prepend to least significant bit
    return FindMissingNum(C, m).
end if
```

Proof of correctness:

Removing a number, m, creates an imbalance of 0s and 1s. If N was odd, the number of 0s in least significant bit position should equal the number of 1s. If N was even, then number of 0s should be 1 more than the number of 1s. Thus if all numbers are present, count(0s) ≥ count(1s). We have four cases:

1. If LSB of m is 0, removing m removes a 0
   - If N is even, count(0s) = count(1s)
   - If N is odd, count(0s) < count(1s)
2. If LSB of m is 1, removing m removes a 1
   - If N is even, count(0s) > count(1s)

- If N is odd, count(0s) > count(1s)

Notice that if count(0s) $\leq$ count(1s) m's least significant bit is 0. We can discard all numbers with LSB of 1 because removing m does not the count of 1s. If count(0s) > count(1s), m's least significant bit is 1. Likewise we can discard all numbers with LSB of 0. Assume that this condition applies for all bit positions up to k. If we look at the $k+1$th bit position, the condition above holds true. The elements at the $k+1$-th bit position have the same bit at the k-th position as m and thus are the only elements we are interested in at the $k+1$ position.

Running time analysis:

Since we care about the number of bits seen, creating the auxiliary arrays looks at $O(N)$ bits. In the worst case, the problem is reduced in half, so we have the recurrence $T(N) = T(N/2) + O(N)$, which, by master's theorem, gives us $T(N) = O(N)$

4. (★★ 10') Median of Medians

The $Quickselect(A, k)$ algorithm for finding the kth smallest element in an unsorted array A picks an arbitrary pivot, then partitions the array into three pieces: the elements less than the pivot, the elements equal to the pivot, and the elements that are greater than the pivot. It is then recursively called on the piece of the array that still contains the kth smallest element.

(a) Consider the array $A = [1, 2, \ldots, n]$ shuffled into some arbitrary order. What is the worst-case runtime of $Quickselect(A, \lfloor n/2 \rfloor)$ in terms of n? Construct an sequence of pivot which have the worst run-time.
A single partition takes $O(n)$ time on an array of size n. The worst case would be if the partition times were $n + (n-1) + \cdots + 2 + 1 = O(n^2)$.
This would happen if the pivot choices were $1, 2, 3, \cdots, \lfloor n/2 \rfloor - 1, n, (n-1), \cdots, \lfloor n/2 \rfloor$. In each of these cases, the partition happens so that one piece only has one element, and the other piece has all the other elements. To get a better runtime, we would like the pieces to be move balanced.

(b) Let's define a new algorithm Better-Quickselect that deterministically picks a better pivot. This pivot-selection strategy is called 'Median of Medians', so that the worst-case runtime of $Better-Quickselect(A, k)$ is O(n).

Median of Medians

1. Group the array into $\lfloor n/5 \rfloor$ groups of 5 elements each (ignore any leftover elements)
2. Find the median of each group of 5 elements (as each group has a constant 5 elements, finding each individual median is O(1))
3. Create a new array with only the $\lfloor n/5 \rfloor$ medians, and find the true median of this array using Better-Quickselect.
4. Return this median as the chosen pivot

Let p be the chosen pivot. Show that for least $3n/10$ elements x we have that $p \geq x$, and that for at least $3n/10$ elements we have that $p \leq x$.
Let the choice of pivot be p. At least half of the groups $(n/10)$ have a median m such that $m \leq p$. In each of these groups, 3 of the elements are at most the median m (including the median itself). Therefore, at least $3n/10$ elements are at most the size of the median. The same logic follows for showing that $3n/10$ elements are at least the size of the median.

(c) Show that the worst-case runtime of Better-Quickselect(A, k) using the 'Median of Medians' strategy is $O(n)$. **Hint**: Using the Master theorem will likely not work here. Find a recurrence relation for $T(n)$, and try to use induction to show that $T(n) \leq c \cdot n$ for some $c > 0$
We end up with the following recurrence:

$$T(n) \leq \underbrace{T(n/5)}_{(A)} + \underbrace{T(7n/10)}_{(B)} + \underbrace{d \cdot n}_{(C)}$$

(A) Calling Better-Quickselect to find the median of the array of medians

(B) The recursive call to Better-Quickselect after performing the partition. The size of the partition piece is always at most $7n/10$ due to the property proved in the previous part.

(C) The time to construct the array of medians, and to partition the array after finding the pivot. This is $O(n)$, but we explicitly write that it is $d \cdots n$ for convenience in the next part.

We cannot simply use the Master theorem to unwind this recurrence. Instead, we show by induction that $T(n) \leq c \cdot n$ for some $c > 0$. The base case happens when BetterQuickselect occurs on one element, which is constant time.
For the inductive case,

$$\begin{aligned} T(n) &\leq T(n/5) + T(7n/10) + d \cdot n \\ &\leq c(n/5) + c(7n/10) + d \cdot n \\ &\leq (\frac{9}{10}c + d)n \end{aligned}$$

We pick c large enough so that $(\frac{9}{10}c + d) \leq c$, i.e $c \geq 10d$. Then we can get $T(n) \leq cn$ for constant c, $T(n) = O(n)$

5.(★★★★★ 10') Merged Median

Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the $n = kl$ elements. Your algorithm should run asymptotically faster than O(n).

Main Idea:

In this problem, I designed a algorithm which get the sth element in the k sorted array.

Using the search method like binary, every time erase almost half of the elements. After one iteration, we use weighted median instead of median of median.

Psuedocode:

Algorithm 6 $sth(s, a_1[1, \cdots, n_1], a_2[1, \cdots, n_2], \cdots, a_k[1, \cdots, n_k])$

```
if n_1 ≤ 1 and n_2 ≤ 1 and ··· and n_k ≤ 1 then
    a ← Sorted([a_1[1], a_2[1], ··· , a_k[1]])
    return a[s]
end if
for i = 1 to k do
    if n_i ≥ 1 then
        b_i = Median(a_i)
    end if
end for
median ← Weighted_Median([b_1, b_2, ··· , b_k, n_1, n_2, ··· n_k])    //Weighted median is the function that sort the median first and add up the size of the previous lists until they add up to half.
for i = 1 to k do
    c_i = BinarySearch(median, a_i)
end for
if Sum(c_i) < s then
    return sth(s − Sum(c_i), a_1[c_1, ··· , n_1], a_2[c_2, ··· , n_2], ··· , a_k[c_k, ··· , n_k])
else
    return sth(s, a_1[1, ··· , c_1], a_2[1, ··· , c_2], ··· , a_k[1, ··· , c_k])
end if
```

Proof of Correctness:

Firstly, I assume the end stituation is all the array has only one or no element. Then I can sort these elements and get the sth from these.

Secondly, I got the weighted medians from the k arrays. And using binary search to get how many elements in every array in smaller than median of medians. If the total number of the elements which is smaller than median is smaller than s, then we just erase them and find the $(s - sum)$th element in the rest arrays, since the sth element is sure not smaller than median, so we just need to look it up in the rest of the arrays. Else, we just erase the element larger than s, then find the sth element in the rest arrays, since the sth element is smaller than median, so we just need to look it up in the rest of the arrays.

Runtime Analysis:

Since the weighted median is between the $[n/4, 3n/4]$ items, and the weighted median algorithm has $O(k)$ complexity, the binary search algorithm has $O(k \log l)$ complexity.

So $T(n) \leq O(k \log l) + T(3n/4)$. And the recursion can be over in $O(\log l)$.

So $T(n) = O(k \log^2 l)$