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. 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 $\mathrm{O}(\cdot)$ 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.

Solution:

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

(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 use the four part proof.**

Solution:

Main idea: In each iteration, partition the matrix into four size-equivalent parts and multiply the top-left submatrix by the vectors $[v_1 + v_2]$ and $[v_1 - v_2]$. Iterate until the matrix has a size of 1×1.

Running time analysis: In each iteration, the size of the submatrix is the half of the original matrix, thus there will be $O(\log n)$ iterations. In each iteration, the total calculation of corresponding vectors is $O(n)$ since the vector is partitioned to two parts and calculated twice (once for addition and once for subtraction) thus the operation of this iteration's calculation is equal to the operation of previous one's and is equal to the first iteration's. Therefore, there will be $O(n \log n)$ operations.

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 proof are required for each part below.**

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

Solution:

Main idea: Partition array A into two subarrays A_l and A_r, each have a size of half of the size of array A. Recursively find the majority element of each subarray (if any) and then scan the whole array A to determine which is the proper majority element of array A (if any). The base case is the case that the array only have one entry, and the value of that entry is the majority of the array in base case.

Pseudocode:

Algorithm 2 findMajorityA(A)

```
if n == 1 then
  return A[0]
end if
A_l ← leftHalf(A), A_r ← rightHalf(A)
M_l ← findMajorityA(A_l), M_r ← findMajorityA(A_r)
count_l, count_r ← 0
while i < n do
  if A[i] == M_l then
    count_l ← count_l + 1
  else if A[i] == M_r then
    count_r ← count_r + 1
  else
    continue
  end if
  i ← i + 1
end while
if count_l > n/2 then
  return M_l
else if count_r > n/2 then
  return M_r
else
  return NULL
end if
```

Proof of correctness:

It is obvious that the majority of array in base case is the value of the only entry.

For other cases (i.e. $n > 1$), suppose none of the subarrays has majority element for itself, then for each element in each subarray, it occurs less or equal than $\frac{\frac{n}{2}}{2}$ times in the corresponding subarray and can only occur less or equal than $\frac{n}{2}$ times in the original array, thus no majority element for the original array. So if the original array has a majority element, it must be the majority element of one of the subarrays. To figure out which majority element of subarrays is the majority element of the original array, scan through the original array and keep counting the occurring times. The one with a count larger than $\frac{n}{2}$ is the one we want, since there can't be two majority elements for the same array or they would occur more than n times in total, which is impossible for an array of size n.

Therefore, The algorithm can find the majority element of the array.

Running time analysis:

Since the size of the array is halved in each iteration, thus there would be $O(\log n)$ iterations. In each iteration, comparing $A[i]$ with majority elements of subarrays need $O(1)$ so the comparing for every entry in A adds up to $O(n)$. Therefore, the algorithm needs $O(n \log n)$ in total.

(b) Can you give a linear time algorithm whose running time is $O(n)$? (You should not reuse the algorithm to answer part a)

Solution:

Main idea: From the first entry of the array A, pair first and last unvisited entries and compare them. If these two values of the corresponding entries are equal, add the value to a new array, otherwise, do nothing. Take one entry forward and repeat the previous procedure until the last two entries are visited. Invoke the algorithm on the newly created array until there are only two entries in the array (base case), if the two entries in the base case have same value, then that value is the majority element, otherwise, the array has no majority element.

Pseudocode:

Algorithm 3 findMajorityB(A)

```
if n == 2 then
  if A[0] == A[1] then
    return A[0]
  else
    return NULL
  end if
end if
Create a new array newArray[n/2]
while i < n/2 do
  if A[i] == A[n − 1 − i] then
    Copy A[i] to newArray
  else
    continue
  end if
  i ← i + 1
end while
return findMajorityB(newArray)
```

Proof of correctness:

For the base case, it's obvious that the majority element of the array in base case only exists when the only two entries have the same value, or the condition "more than half of the entries" can't be satisfied.

For other cases (i.e. $n > 2$), **if the majority element of the array exists, then the majority element of the newly created array exists and has the same value.** Denote the majority element and any other element as m and s respectively. In (m, m) situation, one majority element is copied; In (m, s) situation, one majority element and one non-majority element are discarded; In (s, s), situation, one non-majority element is copied; In (s_1, s_2) situation, both non-majority elements are discarded. Since there are more occurrence of m than the occurrence of non-majority elements, even though m might be discarded, the number of non-majority elements discarded is still larger than the number of m. Thus there will be more occurrence of m than the total occurrence of non-majority elements in the newly created array. **If there is no majority element in the newly created array, then there is no majority element in the original array.** Since the occurrence of each entry of the newly created array is less or equal than half of the size of new array, thus the occurrence of these entries in the original array cannot be larger than half of the size of the original array because distinct entries have been discarded.

Therefore, the algorithm can find the majority element of the array.

Running time analysis:

The recurrence relation is as follows:

$$T(n) = T(\frac{n}{2}) + O(n),\ T(2) = O(1)$$

Therefore, the total running time can be calculated as:

$$T(n) = \frac{O(n) \cdot (1 - (\frac{1}{2})^{\log n})}{1 - \frac{1}{2}} = O(n)$$

The result holds when $n \to \infty$.

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. **Four part proof is required.**

Solution:

Main idea: In each iteration, create two arrays, then scan through the original array and partition it into two parts given the value of the highest bit (the $(\lceil \log_2(N+1) \rceil - 1)$th bit) of each integer. If the subarray that contains integers which has 0 on their $(\lceil \log_2(N+1) \rceil - 1)$ bits has a size less than $2^{\lceil \log_2(N+1) \rceil - 1}$, then invoke this algorithm on this subarray, otherwise, invoke on the other subarray. Iterate until the original array has only 1 integer (base case).

Pseudocode:

Algorithm 4 findMissing(A)

```
if N == 1 then
    b ← fetch the 0th bit of A[0]
    return 1 − b
end if
Create two arrays: smaller_array, larger_array
highest_bit ← ⌈log2(N + 1)⌉ − 1
while i < N do
    number_hbit ← fetch the highest_bitth bit of A[i]
    if number_hbit == 1 then
        Copy A[i] to larger_array
    else
        Copy A[i] to smaller_array
    end if
    i ← i + 1
end while
if smaller_array.size < 2^highest_bit then
    return 0 + findMissing(smaller_array)
else
    return 1 << highest_bit + findMissing(larger_array)
end if
```

Proof of correctness:

For the base case, $N = 1$, so either 0 is missing or 1 is missing, just check the 0th bit and if it's 1 then missing 0, if it's 0 then missing 1.

For other cases (i.e. $N > 1$), since m bits binary integers can composite at most 2^m decimal integers, thus for decimal integer N, its binary representation has $\lceil \log_2(N+1) \rceil$ bits and the highest bit is the $(\lceil log_2(N+1) \rceil - 1)$th bit (counting from 0). If there is a missing integer between 0 and N, then that integer is either smaller than $2^{\lceil \log_2(N+1) \rceil - 1}$ or larger or equal than $2^{\lceil \log_2(N+1) \rceil - 1}$, which makes $2^{\lceil \log_2(N+1) \rceil - 1}$

an entry to divide the array. Notice that there are exactly $2^{\lceil \log_2 (N+1) \rceil - 1}$ integers that are smaller than $2^{\lceil \log_2 (N+1) \rceil - 1}$, so if the size of the array that is aimed at storing integers which are smaller than $2^{\lceil \log_2 (N+1) \rceil - 1}$ (i.e. *smaller_array*) is smaller than $2^{\lceil \log_2 (N+1) \rceil - 1}$, then there must be an integer missing between 0 and $2^{\lceil \log_2 (N+1) \rceil - 1}$, and the highest bit of the missing integer in this iteration is 0, so we can search that integer by invoking the algorithm on *smaller_array* and then return the returned value added by 0; otherwise, the missing integer must be between $2^{\lceil \log_2 (N+1) \rceil - 1}$ and N, and the highest bit of the missing integer in this iteration is 1, so we can search that integer by invoking the algorithm on *larger_array* and then return the returned value added by $1 << (\lceil \log_2 (N+1) \rceil - 1)$, which keeps its highest bit.

Therefore, the final returned value is the value of the missing integer.

Running comlexity analysis:

In each iteration, scan through the array needs $O(N)$ and the size of the subarray is at most $2^{\lceil \log_2 (N+1) \rceil - 1}$. Denote $\lceil \log_2 (N+1) \rceil$ as K, then $O(N) = O(2^K)$. The recurrence relation can be written as:

$$\begin{aligned}
C(N) &= C(2^{K-1}) + O(2^K) \\
&= C(2^{K-2}) + O(2^K) + O(2^{K-1}) \\
&= \cdots \\
&= C(1) + O(\sum_{i=2}^{K} 2^i) \\
&= O(1) + O(\frac{2^2 \cdot (1 - 2^{K-1})}{1-2}) \\
&= O(2^{K+1}) \\
&= O(N)
\end{aligned}$$

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 smaller than the pivot, the elements equal to the pivot, and the elements that are larger 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 $A = [1, 2, ..., n]$ shuffled into some arbitrary order. What is the worst-case runtime of $Quickselect(A, \lfloor n/2 \rfloor)$ in terms of n? Construct the sequence of pivots which have the worst run-time.

Solution:

The worst-case runtime is $O(n^2)$.

The sequence of pivots is: $1, 2, \cdots, \lfloor n/2 \rfloor$ or $n, n-1, \cdots, \lfloor n/2 \rfloor$.

(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 $p \leq x$.

Solution:

Since p is also a median in one of the first $\lfloor n/5 \rfloor$ groups, consider the first iteration. Since p is the median of the first $\lfloor n/5 \rfloor$ medians, there are $\lfloor \frac{\lfloor n/5 \rfloor}{2} \rfloor$, or to say, at least $n/10$ groups' medians are larger or equal to p. In each of these groups, there are at three elements that are larger or equal to its own median, thus at least $3n/10$ elements x are larger or equal to p.

Similarly, since p is also a median in one of the first $\lfloor n/5 \rfloor$ groups, consider the first iteration. Since p is the median of the first $\lfloor n/5 \rfloor$ medians, there are $\lfloor \frac{\lfloor n/5 \rfloor}{2} \rfloor$, or to say, at least $n/10$ groups' medians are smaller or equal to p. In each of these groups, there are at three elements that are smaller or equal to its own median, thus at least $3n/10$ elements x are smaller or equal to p.

(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 $c > 0$

Solution:

An iteration of $Better-Quickselect$ (not the last iteration) contains three parts: 1. invoke $Median\ of\ Medians$;

2. partition the array into two parts according to the pivot; 3. invoke $Better-Quickselect$.

For the first part, the way to invoke $MedianofMedians$ is to recursively invoke $Better-Quickselect$ on an array of size $\lfloor n/5 \rfloor$, thus the time for this part is $T(\frac{n}{5})$. For the second part, the partition needs $O(n)$ since it needs to scan through the array. Here, we write the time as $a \cdot n$. For the third part, since we have shown that at least $3n/10$ elements are discarded, thus the running time for this part is at most $T(n-\frac{3n}{10}) = T(\frac{7n}{10})$. Therefore, the unequal recurrence relation is:

$$T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + a \cdot n$$

For $n = 1$, $T(1) \leq a \cdot 1$, satisfied;
For $n \geq 2$, assume $T(n) \leq c \cdot n$ for some $c > 0$,

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

Thus, if $\frac{9}{10} \cdot c + a \leq c$, i.e. $c \geq 10 \cdot a$, then $T(n) \leq c\dot{n}$, i.e. the running time is $O(n)$.

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). **Four part proof is required.**

Solution:

Main idea: In each iteration, find the median of the medians of remaining arrays (denote median of medians as M) and partition each remaining array into three parts: one for elements smaller than M, one for elements equal to M and one for elements larger than M. Combine all parts that are partitioned from remaining arrays into three arrays, corresponding to the relationship between their elements and M. Calculate each array's size and accumulate their sizes until the accumulated size is larger than t (t indicates the position of actual median in this iteration), invoke the algorithm on this array. Iterate until there are at most 3 elements left.

Pseudocode:

The following pseudocode assumes that the argument array A is an array of arrays.

Algorithm 5 mergedMedian(A, t)

```
if n ≤ 3 then
    insertionSort(A)
    return (n == 2) ? (A[0] + A[1])/2 : A[t]
end if
Create three two dimensional arrays: smaller_array, equal_array, larger_array
Create an array to store the medians of A's subarrays: medians
while i < k do
    medians.append(A[i][⌊l/2⌋])
end while
M ← medianOfMedians(medians)
while i < k do
    Use binarySearch(A[i]) to determine the last element smaller than M, the first element larger than M
    new array smaller_part ← part of A[i] that are smaller than M
    new array equal_part ← part of A[i] that are equal to M
    new array larger_part ← part of A[i] that are larger than M
    smaller_array.append(smaller_part)
    equal_array.append(equal_part)
    larger_array.append(larger_part)
    i ← i + 1
end while
if smaller_array.eleCount > t then
    return mergedMedian(smaller_array, t)
else if (smaller_array.eleCount + euqal_array.eleCount) > t then
    return M
else
    return mergedMedian(larger_part, t − smaller_array.eleCount − euqal_array.eleCount)
end if
```

Proof of correctness:

In each time, we find the median of medians M, the actual median of all elements can only be smaller than M or equal to M or larger than M. If the actual median is smaller than M, then it must be one element of *smaller_array* and the size of *smaller_arraty* must be larger than t which indicates the position of the actual median and is calculated by grade school mathematics, the correctness of this case can be proved by contradiction: If the *smaller_array* has a size larger than t and it doesn't contain the actual median, then there would be more elements that are larger or equal to M than elements smaller than M, which is contradictory to the definition of actual median. For other two cases (the actual median is equal to M or larger than M), the same logic holds and the proof is similar to the previous one.

Running time analysis:

In each iteration, finding medians and $medianOfMedians$ and partition needs $O(k)$, the $binarySearch$ needs $O(k \cdot \log l)$. Since each time there are at least $k \cdot \frac{l}{2} = \frac{n}{2}$ elements being discarded, thus the iteration needs $O(\log l)$ to reach the base case, which only needs $O(1)$ to finish. The recurrence relation is:

$$\begin{aligned} T(n) &\leq T(\frac{n}{2}) + O(k) + O(k \cdot \log l) \\ &\leq T(\frac{n}{2}) + O(k \cdot \log l) \\ &\leq T(\frac{n}{2}) + O(2k \cdot \log l) \\ &\leq \cdots \\ &\leq T(3) + O(\log l \cdot k \cdot \log l) \\ &= O(k \cdot \log^2 l) \\ &< O(n) \end{aligned}$$