

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 i th 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 separate 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 elements in the front list are 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 elements in the back list are 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 say 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 iterations 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 .

(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.**

So we can apply the divide and conquer method

Let's say for col vector a with length n , let a_1 be the column with $\frac{n}{2}$ consisting of first $\frac{n}{2}$ coordinates of a

$$\text{Denote } \begin{cases} (H_k b)_1 = H_{k-1} b_1 + H_{k-1} b_2 = H_{k-1}(b_1 + b_2) \\ (H_k b)_2 = H_{k-1} b_1 - H_{k-1} b_2 = H_{k-1}(b_1 - b_2) \end{cases}$$

so we just need to compute $H_{k-1}(b_1 + b_2)$, $H_{k-1}(b_1 - b_2)$ recursively.

both solving the + and - takes $O(n)$ time

So the recursive time complexity could be written as

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

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

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 proof are required for each part below.

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

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

a) method: divide & conquer.

Main idea. If A has an element v , v must be the element in A_1 or A_2 or both $A_1 \& A_2$ and the counterproposition: If v is no bigger than half in some part, v will be no bigger than the half totally. ∴ If both A_1 and A_2 have the same element, it'll be the element of A . It'll take $O(n)$ to see whether it's the element in A until the base condition when $n=1$.

pseudocode:

Algorithm: majority1 (data)

```

n<len(data)
list other()
i ← 0
while i < n do
    if data(i) ≤ n/2
        else major ← major1(data[:i])
        else major ← major1(data[i+1:])
    end if.
    if data(i) = n/2 then
        return major
    end if.
endwhile
return None

```

proof of correctness

① the base condition $n=1$ the algorithm holds

② suppose $n=m$ the algorithm holds that is $\text{major} = \text{major1}(\text{data}[:2^m])$ or $\text{major} = \text{major1}(\text{data}[2^m:])$

\therefore If A has an element v , v must be the element in A_1 or A_2 or both $A_1 \& A_2$ and vice versa.

$\Rightarrow \text{major} = \text{major1}(\text{data}[:2^{m+1}])$ or $\text{major} = \text{major1}(\text{data}[2^m:])$ holds

Running time analysis

If 2 parts has the same element, you just search for 2 sets to find the correctness so it's $O(n)$.

If you do the above part recursively when can deduct the production rule $T(n) = 2T(n/2) + O(n)$, by Master Theorem, $O(n \log n)$

b) improvement to the algorithm in a: we can find a reduction technique for pairs. indicated in that L_1 is formed by self-partitioning L into pieces and include the element from each pair where 2 elements are the same.

Main idea: give a list data of $n > 0$ items, and a tie unlocker item. we first circle out special condition that is T be none. the element must be either in L or in T ($\text{if } T = \frac{1}{2}L$) so we can recursively define (L_1, T_1) by 1. L_1 is formed by self-partitioning L into 2 pieces, and include an element from each pair where the 2 elements are the same. 2. T_1 is the odd unpicked element of L if n is odd. and T_1 is T is n is even.

The tie unlocker is set as the part of sub-problems so it should be set as none. and with this flag, it can easily break the tie. It may give a positive contribution to the vote but can't overturn with at least one extra vote.

pseudocode:

Algorithm : majority2l(data, tie unlocker)

```

n < len(data)
if n == 0 then
    return tie unlocker
endif.
list other().
if n % 2 == 1 then
    tie unlocker = data[-1]
endif
i < 0
while i < n-1 do
    if data[i] == data[i+1] then
        other.insert(data[i])
    endif
    i += 2
end while
major < majority2(other, tie unlocker)
if major == None then
    return None
endif
temp = data.count(major)
if 2 * temp > n
    return major
endif
if 2 * temp == n then
    if major == tie unlocker then
        return major
    endif
endif
return None

```

proof of correctness: let M be the majority element, m be the number of the copy of M and k be the count of the rest be the excess, $m-k$.

lemma: with direct majority element means $e \geq 0$

- ① requirements $e > 0$ or $(e=0 \text{ and } M=T)$ (initial condition)
 - ② abandoning the unmatched pairs may lowered the n and m , but n decreases less than m . i.e. e will not decrease and element will not be invalidated
 - ③ if n is odd, there's no tie, so just consider 2 pairs and odd extra element from L_1 . The odd extra element just like the tie unlocker with L_1 pairs. so, each pair adds 2 votes. and the odd element adds up when the vote matches in L_1
 - ④ if n is even, e is even, too, in the next level. $(e \geq 0 \text{ and } M=T)$ or $(e=1 \text{ and } M \neq \text{the element})$
 - ⑤ if n is even, e is even, too, in the next level. $(e \geq 0 \text{ and } M=T)$ or $(e=1 \text{ and } M \neq \text{the element})$
- ⇒ the same majority can be found for (L_1, T_1) with constraints using above 2 transformation

Running time analysis

We can get the recursive function $T(n) = T(\frac{n}{2}) + O(n)$. by master theorem $T(n) = O(n)$

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 $O(N \log N)$ 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.**

Main idea: To determine the missing integer in $O(n)$ time. we can apply the divide and conquer idea to call the least significant bit as 0-th bit and most significant bit of n is $\lceil \lg(n+1) \rceil - 1$ of the whole bits.
pseudo code:

Algorithm: findMissing (item n)

```

if n == 1 then
    return 1 - len(list) // this means the 0-th of missing number.
endif
list leftlist()
list rightlist()
pos <=  $\lceil \lg(n+1) \rceil - 1$ 
for number in list
    if number [pos] == 0
        leftlist.insert(number)
    else
        rightlist.insert(number)
    end if
end for
If leftlist.size < pos/2 then

```

findmissing (leftlist, pos/2) // the pos bit is 0, and recursively call leftlist with pos/2
else abandon the right one

findmissing (rightlist, pos/2) // the pos bit is 1, and recursively call rightlist with pos/2
end if

Python:
version

```

def findmissing(items, i):
    if len(items) == 0:
        return 0

    numone = numzero = 0
    itemone = []
    itemzero = []
    for e in items:
        if (e >> i) & 1 == 0:
            numzero += 1
            itemzero.append(e)
        else:
            numone += 1
            itemone.append(e)

    if numone >= numzero:
        #remove 0
        return 2 * findmissing(itemzero, i+1)
    else:
        #remove 1
        return 2 * findmissing(itemone, i+1) + 1

```

Proof of correctness: All of the integer between 0 to n. $2^{\lceil \lg(n+1) \rceil - 1}$ numbers are less than $2^{\lceil \lg(n+1) \rceil - 1}$ because if we look at the $(\lceil \lg(n+1) \rceil - 1)$ bit of $[0, n]$ number, we can determine whether the number is $< 2^{\lceil \lg(n+1) \rceil - 1}$, then, we can look at the $(\lceil \lg(n+1) \rceil - 1)$ bit in $A[i, n]$ just within n loopup, we can form the right list and left list. that is smaller than $2^{\lceil \lg(n+1) \rceil - 1}$ the right list is vice versa.

use induction law,

- ① $n=1$, just one element in {0, 1} which is correct using the algorithm above
- ② suppose m works. that is the length of subproblem is no bigger than m. we have $2^{\lceil \lg(n+1) \rceil - 1} \leq n \leq 2^{\lceil \lg(n+1) \rceil - 1}$ in $m+1$ case we have $2^{\lceil \lg(n+1) \rceil - 1} < n$, $n - 2^{\lceil \lg(n+1) \rceil - 1} \leq 2^{\lceil \lg(n+1) \rceil - 1} - 2^{\lceil \lg(n+1) \rceil - 1} = 2^{\lceil \lg(n+1) \rceil - 1} - 1 < n$

the previous equation holds
 \therefore the most significant bit of the missing number and the list in which to look for its rest part.

Running time analysis.

$$T(n) \leq T(2^{\lceil \lg(n+1) \rceil - 1}) + cn$$

let t be a function corresponding to \geq $T[\lg(n+1)] - 1 = t$

$$T(n) = T(2^t - 1) + cn$$

$$\leq T(2^t - 1) + c(\underbrace{2^t - 1}_{n \leq 2^t - 1})$$

$$= T(2^t - 1) + c((2^{t+1} - 1) - (2^t - 1)) \text{ // accumulated addition}$$

$$\leq T(2^2 - 1) + c(2^{t+1} - 1 - (2^2 - 1))$$

$$= T(1) + c(2^{t+1} - 2) = T(1) + 2c(2^t - 1) \leq T(1) + 2cn$$

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

4. (★★ 10') Median of Medians

The $Quickselect(A, k)$ algorithm for finding the k th 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 k th smallest element.

(a) Consider the array $A = [1, 2, \dots, n]$ 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.

(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 at least $3n/10$ elements x we have that $p \geq x$, and that for at least $3n/10$ elements we have that $p \leq x$.

(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 $c > 0$

a) take the worst situation into consideration. partition on the group takes $O(n)$ and take n times the time complexity $n + (n-1) + \dots + 2 + 1 = O(n^2)$

This could happen if pivot is the sequence or the converse sequence of the group index, (1...n...n...1)

In each of the test case, the worst case is one piece has one element and the other has the rest.

b) we have to make the pieces to be more balanced to get better runtime

In the Better-Quicksort algorithm, at least $\frac{n}{5}$ has a median m s.t. m is no bigger than the pivot. similarly, $\frac{2}{5}$ of the elements are at most the median. \therefore more than $\frac{3n}{10}$ elements are at least of the size of the median.

c) From the PPT, we get $T(n) \leq T(n/5) + T(7n/10) + dn -$ construct the array takes $O(n)$

You select $n/5$ to call Better-Quicksort to get $7n/10$ we have to calculate the the get piece which is $\frac{1}{5} \times \frac{1}{2} = \frac{1}{10}$ and there's 3 pieces. (explained in (b))
quicksort that will find the median of the array So the total size of the partition piece is always at most $7n/10$.

$$T(n) \leq T(n/5) + T(7n/10) + dn \leq \frac{1}{3}cn + \frac{7}{10}cn + dn = \frac{9}{10}cn + dn$$

$$\text{let } \frac{9}{10}c + d \leq c \Rightarrow c \geq 10d$$

$$\therefore T(n) \leq cn \text{ for some } c > 0 \Leftrightarrow T(n) = O(n)$$

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 $O(n)$. **Four part proof is required.**

Main idea: calculate the n medians m_1, \dots, m_k of the $ar_i[l] (i \in [0, k-1])$ respectively

- 1) if $m_1 = \dots = m_k$ then we are done. return any one of them (like m_1)
- 2) if $m_1 \geq m_{i+1}$ then median is present in one of the below: from first element of ar_i to m_1 or from m_{i+1} to the last element of ar_i
- 3) if $m_1 < m_{i+1}$ then median is present in one of the below: from first element of ar_i to m_1 or from m_{i+1} to the last element of ar_i
- 4) repeat the 2 until the size of all the subarray becomes 2
- 5) use the formula to calculate the median, $\text{median} = \frac{\max(ar_1[0], ar_2[0], \dots, ar_k[0]) + \min(ar_1[l], \dots, ar_k[l])}{2}$

Pseudo code:

Algorithm: Merged Median ($ar_1, ar_2, \dots, ar_k, k$)

```

let  $ar_k$ 
for  $i$  in range  $(1, k)$ 
     $m_i = ar_i[\lfloor i/2 \rfloor]$ 
end for
if  $m_1 = m_2 = m_3 = \dots = m_k$  then
    return  $m_1$ 
end if
while  $\text{len}(ar_k) \neq 2$  do
    for  $i$  in range  $(1, k)$ 
         $m_i = ar_i[\lfloor i/2 \rfloor]$ 
    end for
    for  $i$  in range  $(1, k-2)$ 
        if  $m_i \geq m_{i+1}$  then
             $ar_i = [:m_i]$ 
             $ar_{i+1} = [m_{i+1}:]$ 
        else
             $ar_i = [m_i:]$ 
             $ar_{i+1} = [:m_{i+1}]$ 
        end if
    end for
end while
return  $(\max(ar_1[0], ar_2[0], \dots, ar_k[0]) + \min(ar_1[l], \dots, ar_k[l])) / 2$ 

```

Proof of correctness

when $k=2$ what we do calculation between the medians m_1 and m_2 of the input arrays $ar_1()$ and $ar_2()$
 if m_1 and m_2 both are equal we are done. return m_1 (means the first condition)

① $\text{len}(ar_k) \neq 2$, the last situation holds that Median must be $\frac{\max(ar_1[0], ar_2[0]) + \min(ar_1[l], ar_2[l])}{2}$

② let $(ar_1[i], i \in [1, l_1-1])$ if $ar_1[i]$ is in $ar_1[i_1:i_2]$ then it must be in $ar_1[i_1:\lfloor \frac{i_1+i_2}{2} \rfloor]$ or $ar_1[\lceil \frac{i_1+i_2}{2} \rceil:i_2]$

it also holds if $ar_1[i]$ is in $ar_1[i_1:i_2]$ then it must be in $ar_1[i_1:\lceil \frac{i_1+i_2}{2} \rceil]$ or $ar_1[\lceil \frac{i_1+i_2}{2} \rceil:i_2]$

and it also holds for ar_2 , so the production rule is correct, with prove when $k=2$ correctness for k arrays just runs a loop with will check for concatenated arrays. and the algorithm is ready. the correctness is obvious

Run time analysis

for $k=2$'s condition. proved in quiz is $O(lgn)$
 so the k th condition is $O(\frac{k}{2} \cdot lgn)$ so the time complexity is $O(lgn)$