

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  $\leftarrow$  0
high  $\leftarrow$   $n - 1$ 
while low < high do
  mid  $\leftarrow$  (low + high)/2
  if ( $k == A[\textit{mid}]$ ) then
    return mid
  else if  $k > A[\textit{mid}]$  then
    low  $\leftarrow$  mid + 1
  else
    high  $\leftarrow$  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 .

(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.**

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)

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 j th 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.**

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 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 $c > 0$

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.**