ShanghaiTech University

CS101 Algorithms and Data Structures

Fall 2019

Homework 11

Due date: 23:59, December 8st, 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 + 1st 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(·) 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)

```
$low \leftarrow 0$
$high \leftarrow n - 1$
while $low < high$ do
  $mid \leftarrow (low + high)/2$
  if $(k == A[mid])$ then
    return mid
  else if $k > A[mid]$ 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. (★ 10’) Rectangle

There are $2n$ sticks of the given lengths. You have to pick exactly 4 of them to form a rectangle. We define C as the circumference of the rectangle and S as the square of the rectangle. How to pick these 4 sticks to make $\frac{C^2}{S}$ min? Four part proof is required.

Solution:

Main idea: First sort the given sticks according to their lengths, then scan through these sticks and looking for sticks that have at least two of the same length, always keep four sticks (two pairs of them and each pair have the same size or four sticks have the same length) that have the proportion most closely to 1. If these sticks cannot make a rectangle, then no solution.

Pseudocode:

Algorithm 2 minimumProportion(A)

```
Sort the sticks form the shortest to the longest: $quickSort(A)$
Use $S_a$ and $S_b$ to store two kinds of sticks, use $temp$ to temporarily store a new kind of stick
while $i < 2n$ do
  if $i + 1 < 2n$ and $A[i] == A[i + 1]$ then
    $tmep \leftarrow A[i]$
    $count \leftarrow 1$
    while $i + 1 < 2n$ and $A[i] == A[i + 1]$ do
      $+ + i$
      $+ + count$
      if $count == 4$ then
        return $\{temp, temp\}$
      end if
    end while
    if $S_a == NULL$ then
      $S_a \leftarrow temp$
    else if $S_b == NULL$ then
      $S_b \leftarrow temp$
    else if $S_b/S_a > temp/S_b$ then
      $S_a \leftarrow S_b$
      $S_b \leftarrow temp$
    end if
  end if
  $+ + i$
end while
if $S_a$ != $NULL$ and $S_b$ != $NULL$ then
  return $\{S_a, S_b\}$
else
  return $NoSolution$
end if
```

Proof of correctness:

If there is no pair of sticks that the numbers of both kind of sticks are at least two, then there is no rectangle.

If there exists rectangle, then denote the two edges as a and b and $b >= a$, then $C = 2(a+b)$ and $S = ab$, then

$$\frac{C^2}{S} = \frac{(2(a+b))^2}{ab} = \frac{4a^2+8ab+4b^2}{ab} = 4(\frac{a}{b}+\frac{b}{a})+8$$

In order to minimize $\frac{C^2}{S}$, $\frac{b}{a}$ should be as close to 1 as possible. If there exists four sticks of the same length, then $b = a$ and the proportion is minimized (to 1). If there is no four sticks of the same length, since this algorithm ensures $b > a$, only need to ditermine whether the length (say $temp$) of newly founded stick has a smaller proportion with b. If $temp/b$ is smaller than b/a, then it means the proportion of $temp$ and b is closer to 1, so we change a to b and change b to $temp$. There is no need to calculate the proportion of $temp$ and a since b/a will always be smaller than $temp/a$. After scanning through all sticks, the final b and a ensure the proportion of them is the minimal and we are done.

Running time analysis:

The *quicksort* need $O(2n\log 2n) = O(n\log n)$; While scanning through the sticks, comparing and exchanging and increasing all need $O(1)$; We only need to scan the sticks for once, that's $O(2n) = O(n)$.

Therefore, the running time is $O(n\log n)$.

2. (★★ 10') Cake

Assume you are going to give some pieces of cake to some children. However, you cannot satisfy a child unless the size of the piece he receives is no less than his expected cake size. Different children may have different expected sizes. Meanwhile, you cannot give each child more than one piece. For example, if the children's expected sizes are [1,3,4] and you have two pieces of cake with sizes [1,2], then you could only make one child satisfied. Given the children's expected sizes and the sizes of the cake pieces that you have, how can you make the most children satisfied? Four part proof is required.

Solution:

Main idea: Sort both children's expectations and cake pieces from the smallest to the largest. Start from the child that asks for the smallest piece and give each child the smallest piece you have that satisfies his/her expectation. If there's no piece of cake that satisfies one's expectation, then finished, and those who has no cake cannot be satisfied.

Pseudocode:

Let E denote the array of children's expectations and let S denote the array of cake pieces you have. Assume the size of E is m and the size of S is n.

Algorithm 3 distributeCake(E, S)

```
Sort children's expectations form the smallest to the largest: $quickSort(E)$
Sort cake pieces you have form the smallest to the largest: $quickSort(S)$
$count \leftarrow 0$
while $i < m$ do
  while $j < n$ do
    if $S[j] < E[i]$ then
      $++j$
      continue
    else
      $++count$
      $++j$
      break
    end if
  end while
  if $j == n$ then
    break
  end if
  $++i$
end while
return $count$
```

Proof of correctness:

By contraction: The number of satisfied children is equal to the number of distributed cake pieces, so we focus on the cake pieces distributed. Assume this greedy *distributeCake* algorithm is not optimal, meaning there are more cake pieces distributed in another algorithm (here we call that algorithm optimal for simplicity). Let $[i_1, i_2, \ldots, i_k]$ denote the array of cake pieces distributed in this *distributeCake* algorithm and let $[j_1, j_2, \ldots, j_m](m > k)$ denote the array of cake pieces distributed in optimal algorithm with

$\{i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r\}$ for the largest possible value of r. i_{r+1} exists (or this algorithm have the same outcome with optimal) and is no larger than j_{r+1} (j_{r+1} exists because $m > k$), which means we can replace j_{r+1} with i_{r+1} and save the larger piece for someone else (if any) that expects larger piece. Here we meet a contradiction: The two algorithms now have $\{i_1 = j_1, i_2 = j_2, \ldots, i_{r+1} = j_{r+1}\}$, r is not the largest value for the situation we mentioned before while the revised optimal algorithm is still optimal. Therefore, the contradiction means $distributeCake$ algorithm is optimal.

Running time analysis:

$quickSort(E)$ needs $O(m \log m)$, $quickSort(S)$ needs $O(n \log n)$ and Searching for the smallest piece of cake satisfies one child's expectation needs $O(n)$ since the algorithm is finished once a child's expectation cannot be satisfied.

Therefore, the total running time is $O(m \log m + n \log n)$ where m is the number of children and n is the number of cake pieces.

3. (★★ 10') Program

There are some programs that need to be run on a computer. Each program has a designated start time and finish time and cannot be interrupted once it starts. Programs can run in parallel even if their running time overlaps. You have a 'check' program which, if invoked at a specific time point, can get information of all the programs running on the computer at that time point. The running time of the 'check' program is negligible. Design an efficient algorithm to decide the time points at which the 'check' program is invoked, so that the 'check' programis invoked for as few times as possible and is invoked at least once during the execution of every program. Four part proof is required.

Solution:

Main idea:

First sort these programs according to their start times from the earliest to the latest. Then scan forward from the earliest start program and find all programs that starts before than or equal to the earliest end time, set the time point at the earliest end time of these programs. Start scanning from the program that starts earliest after the previous time point and redo previous procedure.

Pseudocode:

Assume the number of programs is n. Assume the ending time point is part of the execution.

Algorithm 4 setTimePoint(A)

```
Sort programs according to their start times from the earliest to the latest: $quickSort(A)$
$time\_points \leftarrow NULL$
while $i < n$ do
    $start\_time \leftarrow A[i].start$
    $end\_time \leftarrow A[i].end$
    $time\_points \leftarrow NULL$
    while $(i+1) < n$ and $A[++i].start \le end\_time$ do
        if $A[i].end \le end\_time$ then
            $end\_time \leftarrow A[i].end$
        end if
    end while
    $time\_points.append(end\_time)$
end while
return $time\_points$
```

If the end time point of a program is not considered as part of the execution, then the actual appended time point should be a little earlier than *end_time* in the above pseudocode, but shouldn't be earlier than that program's start time.

Proof of correctness:

If each time point includes as many as it can then the number of time points is the least, otherwise, there will be at least one program that no time point includes it. So we need to prove that in each time point determined by the above algorithm, the number of check programs is the most. In each scanning procedure,

1. If we set the time point before the start time of the earliest ended program, then this time point will not include the earliest ended program. Since that program starts after and ends before any other programs scanned in this procedure, move the time point to a point between its start time and end time will still

include all the other programs, which makes the time point in if statement inoptimal (meaning not including the most programs).

2. If we set the time point after the end time of the earliest ended program, then this time point will not include the earliest ended program. Since that program starts after and ends before any other programs scanned in this procedure, move the time point to a point between its start time and end time will still include all the other programs, which makes the time point in if statement inoptimal (meaning not including the most programs).

Therefore, any time point that lies between the start time and end time of the earliest ended program includes most of the programs and the previous algorithm satisfies the demand.

Running time analysis:

The *quickSort* needs $O(n \log n)$. Since we only scan these programs for exactly once and every operation in scanning procedure is $O(1)$, the time for scanning is $O(n)$.

The total running time is $O(n \log n)$.

4. (★★★ 15') Guests

n guests are invited to your party. You have n tables and many enough chairs. A table can have one or more guests and any number of chairs. Not every table has to be used. All guests sit towards these tables. Guest i hopes that there're at least l_i empty chairs left of his position and at least r_i empty chairs right of his position. He also sits in a chair. If a guest has a table to himself, the chairs of his two direction can be overlap. How can you use smallest number of chairs to make everyone happy? Note that you don't have to care the number of tables. Four part proof is required.

Solution:

Main idea:

Sort all guests' left empty chair requirements and right empty chair requirements from the most to the least respectively. Then each time match the max requirement in left remaining requirements with the max requirement in right remaining requirements and move on to the next, continue matching until all requirements are matched.

Pseudocode:

Assume there are m guests and $L[i]$ includes guest i's left requirement and $R[i]$ includes guest i's right requirement.

Algorithm 5 assignChairs(L, R)

```
Sort left requirements from the most to the least: $quickSort(L)$
Sort right requirements from the most to the least: $quickSort(R)$
$chairs \leftarrow m$
while $i < m$ do
    $chairs += max(L[i], R[i])$
end while
return $chairs$
```

Proof of correctness:

Consider the base case where there's only one guest, then he only need one table and the number of empty chairs is determined by the max value of his left and right requirements, and add that number by 1 is the number of the total chairs.

When there are two guests, if they sit on different tables, then the total number of chairs can be written as

$$1 + max(L[0], R[0]) + 1 + max(L[1], R[1])$$

If the two guests sit on the same table, then the total number of chairs can be written as

$$2 + max(L[0], R[1]) + max(L[1], R[0])$$

After reviewing these two expressions, we can recognize the second situation as the situation where each guest sits on his/her own table after exchanging one of his/her requirements with the other guest's opposite requirement.

For more guests, we can use induction to match every two guests (or one guest with him self) and make the most use of overlapping. Hence the total number of chairs for m guests can be written as

$$m + \sum_{i=0}^{m-1} max(L[i], R[i])$$

Where L and R are two sorted arrays of left requirements and right requirements respectively.

Running time analysis:
Both two *quickSort* need $O(m \log m)$ and both scannings (through L and R) need $O(m)$.
Therefore, the total running time is $O(m \log m)$, where m is the number of guests.