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 $i + 1$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(·) 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.

Main idea:

Select those lengths that have more than one sticks. Sort these lengths. Pick two consecutive lengths that the ratio of them is cloest to 1. Pick 2 sticks of each length.

Pseudocode:

Algorithm 2 rectangle(Stick$[1, \ldots, n]$)

```
$Length \leftarrow SELECT\ lengths\ of\ more\ than\ one\ sticks\ from\ Stick$
$Sort\ Length\ in\ increasing\ order$
$max \leftarrow 0$
$maxIndex \leftarrow 0$
for each $i \in [1, size(Length) - 1]$ do
    if $Length[i]/Length[i+1] > max$ then
        $max \leftarrow Length[i]/Length[i+1]$
        $maxIndex \leftarrow i$
    end if
end for
return $Length[maxIndex],\ Length[maxIndex],\ Length[maxIndex+1],\ Length[maxIndex+1]$
```

Proof of correctness:

Suppose the length and width of the rectangle are a and b. So $\frac{C^2}{S} = \frac{(2a+2b)^2}{ab} = 4(\frac{a}{b} + \frac{b}{a} + 1)$. It's a Nike function. The closest $\frac{a}{b}$ to 1 makes $\frac{C^2}{S}$ min. That only happens to two consecutive lengths.

Running time analysis:

Sort and visit sticks to select repeated lengths. That takes $O(n \log n)$. Sort lengths. Size of lengths are $\frac{n}{2}$ at most. That takes $O(n \log n)$. Visit lengths to find max ratio. That takes $O(n)$. Thus, whole algorithm takes $O(n \log n)$.

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.

Main idea:
We first sort expected sizes and cake sizes in increasing order.
Initially, the counter is zero. Every step we satisfy the lowest expected pending size with the smallest cake that can statisfy it and the counter pluses one. If the biggest remaining cake cannot statisfy the lowest expected pengding size or if there's no cake remaining, our algorithm ends and the counter is the answer.

Pseudocode:

Algorithm 3 cake(children$[1,\dots,n]$, cakes$[1,\dots,n]$)

```
$count \leftarrow 0$
$Sort\ children,\ cakes\ in\ increasing\ order$
$i \leftarrow 1,\ j \leftarrow 1$
while $i <= n$ and $j <= n$ do
  if $children[i] <= cakes[j]$ then
    $i \leftarrow i+1,\ j \leftarrow j+1$
    $count \leftarrow count + 1$
  else
    $j \leftarrow j+1$
  end if
end while
return $count$
```

Proof of correctness:
We prove this algorithm is optimal following.
A. $\{A_1, \cdots, A_p\}$ denote the children' expected sizes statisfied by our algorithm. $\{B_1, \cdots, B_q\}$ denote the children' expected sizes statisfied by an optimal solution. $A_1 \le \cdots \le A_p$ and $B_1 \le \cdots \le B_q$. Apparently, $p \le q$.
B. Since our algorithm statisfies the lowest expected sizes, it's easy to know $A_1 \le B_1, \cdots, A_p \le B_p$. If there's a cake statisfying B_i, it can also statisfy A_i. $i = \{1, \cdots, p\}$. So we can reconstruct another optimal solution as $C_1, \cdots, C_p, B_{p+1}, \cdots, B_q$, where $C_i = A_i$ and $i = \{1, \cdots, p\}$.
C. Started with $i = 1$ iteratively, if the assigned cake of C_i is bigger than A_i's, we can assign C_i the same cake of A_i by swapping or assigning from remaining cakes. It won't affect the result. Our algorithm can ensure C_i's is bigger than or equal to A_i's when we have same assigned cakes of C_j and A_j where $j < i$. In this way, A_i and C_i can have same assignment.
D. Assume $p < q$, then there's a cake can statisfy $B_{p+1} > A_p$. It's a conflict according to our algorithm's end condition. So $p = q$, our solution is an optimal solution.
Running time analysis:

We sort two arrays. It runs in $O(max(m,n)log(max(m,n)))$ where m denotes the number of expected sizes and n denotes the number of cake size. Since our algorithm visit these two arrays once, it runs in $O(m{+}n){=}O(max(m{+}n))$. So the whole program runs in $O(max(m,n)log(max(m,n)))$. ($O(m{+}n)log(m{+}n)$ is also right)

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.

Main idea:
We can sort these programs by its finish time in increasing order. When we do this, we can use a priority queue. Initially, we assume the last check time is 0.
In each step, we dequeue the first node constantly until this node's start time is greater than the last check time. Then we insert a 'check' program at the first node's finish time and update the last check time. It's easy to know if a node's start time is not greater than the last check time it has been checked before. It's not necessary to care about these checked nodes.

Pseudocode:

Algorithm 4 program(Program$[1, \ldots, n]$)

```
$count \leftarrow 0$
$lastTime \leftarrow 0$
build a min heap myHeap of $n$ programs by comparing$Programs[1, \ldots, n].finishTime$
while $myHeap.hasElement() = True$ do
  if $myHeap.top.startTime <= lastTime$ then
    $myHeap.pop()$
  else
    $lastTime \leftarrow myHeap.top.finishTime$
    $count \leftarrow count + 1$
    $myHeap.pop()$
  end if
end while
return $count$
```

Proof of correctness:
We prove this algorithm is optimal following.
A. Giving a series of programs, we analysis the first insert time.
B. If the first insert time is behind our selection, i.e. it's behind a program's finish time, the result will miss some programs to check.
C. If the first insert time is before our selection, the remaining programs that haven't been checked will be a superset of ours. So the number of remaining 'check' programs can not be less than ours.
D. We can analysis the remaining programs' first insert time in the same way iteratively. Proved.
Running time analysis:
We build a heap running in $O(n)$. We dequeue n times and each time we maintain the heap running in $O(logn)$. So this algorithm runs in $O(nlogn)$.

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? You can only give the number of chairs. Note that you don't have to care the number of tables. Four part proof is required.

Main idea:
Sort array l, r in increasing order respectively which means r_k and l_k are not demands of the same person after sorting. The answer is $\Sigma_{k=1}^{n} max(l_k, r_k) + n$.

Pseudocode:

```
Algorithm 5 guests( l[1, ..., n], r[1, ..., n])
  Sort $l$
  Sort $r$
  $ans \leftarrow 0$
  for each $k \in [1, n]$ do
    $ans \leftarrow ans + max(l[k], r[k])$
  end for
  return $ans + n$
```

Proof of correctness:
In each solution, every guest i has a left neighbor j and a right neighbor k. If guest i's table has only 1 guest, $i = j = k$. If it has 2 guests, $j = k$. We use $Left[i] = j, Right[i] = k$ to record the neighbors of i. Since everyone is in a ring, we can only record it's left neighbors to represent a solution. The number of needed chairs of this solution is $\Sigma_{i=1}^{n} max(r_{Left[i]}, l_i) + n$.
Then we define a inversion in a solution is $\exists i_1, i_2 \in [1, \dots, n], r_{Left[i_1]} < r_{Left[i_2]}, l_{i_1} > l_{i_2}$. The optimal solution is a solution with no inversions. We prove that in the following way:
A. Suppose it's not the optimal solution. Then the optimal solution has at least one inversion.
B. Suppose one inversion is i_1, i_2. Then $r_{Left[i_1]} < r_{Left[i_2]}, l_{i_1} > l_{i_2}$.
C. Swap them. 2 cases:
a. If biggest two elements are $\{r_{Left[i_2]}, l_{i_1}\}$, swapping will decrease the result.
b. If biggest two elements are $\{r_{Left[i_1]}, r_{Left[i_2]}\}$ or $\{l_{i_1}, l_{i_2}\}$, swapping will remain the result.
D. Thus, by swapping all inversions, we can get that the result with no inversions is not bigger than the optimal. That's a conflict.

Running time analysis:
Sorting two array takes $O(n \log n)$. Calculating the answer takes $O(n)$. So the whole algorithm takes $O(n \log n)$.