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 $\mathrm{O}(\cdot)$ 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. (★ 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:

Suppose p and q are adjacent sides of a rectangle. Then the circumference of the rectangle C can be expressed as $2 \times (p+q)$, and the square of the rectangle S can be expressed as $p \times q$. We can use p and q to express $\frac{C^2}{S}$:

$$\frac{C^2}{S} = \frac{(2 \times (p+q))^2}{p \times q} = 4\frac{p^2+q^2+2pq}{pq} = 4(\frac{p}{q} + \frac{q}{p} + 2)$$

In order to minimize it, p needs to get close to q as much as it can, which means $\frac{q}{p}$ needs to be as close to one as it can. Firstly, we need to sort the $2n$ lengths of the sticks into ascending order. Then we can traverse the array storing the lengths. If the number of one element is smaller than two, we just ignore it and consider the next element. The variable 'ratio' is used to record the optimal $\frac{q}{p}$, and 'P' and 'Q' is used to record the according p and q. The two adjacent considered elements are regarded as p and q, and we can calculate $|\frac{q}{p} - 1|$. If it is smaller than ratio, update ratio to the new $\frac{q}{p}$, and update P and Q at the same time. Otherwise, just keep on. After we iterate the whole array, we have got the optimal ratio which is sufficiently close to one. Then we can compute P and Q, which can minimize $\frac{C^2}{S}$. Notice that if the number of an element is greater than or equal to four, it must be the optimal solution and we can use four such sticks to minimize $\frac{C^2}{S}$. As soon as we get such element, the algorithm can be finished.

Pseudocode:(It is on the next page)

Proof of Correctness:

1. In the traversal, when we get an element whose number is greater than or equal to 4, it means that we can use 4 sticks of that length to create a square. In such case, $\frac{q}{p} = 1$, and we can never find a ratio more close to one as one. Hence, i is assigned $2n$ to jump out of the loop. Pick four sticks of length $Length[i]$ is the optimal solution, and the minimal $\frac{C^2}{S}$ is 16.

2. If an element whose number is one, we can not pick it because we need two pairs of sticks with each pair sharing the same length. Hence, in such case i is increased by one and we get into the next loop.

3. When we meet an element whose number is 2 or 3, it means that the sticks with such length have chances to be picked, and it will not be used to create a square. Then we can compare the ratio of them with the previous one, and if the current one is closer to one, we just update the *ratio* so that we can get a ratio most close to one.

4. Plus, because we have sorted the $2n$ lengths, the optimal ratio must be generated from the ratio of the adjacent elements.

Algorithm 2 Ractangle($l_1, l_2, ..., l_{2n}$)

Store $l_1, l_2, ..., l_{2n}$ into an array $Length$ of size $2n$.
QuickSort($Length$)
$p \leftarrow 0$
$ratio \leftarrow \infty$
$i \leftarrow 1$
while $i \neq 2n$ **do**
  if (count($Length[i]$)== 1) **then**
    $i \leftarrow i + 1$
  else if (count($Length[i]$)$\geq$ 4) **then**
    $P \leftarrow Length[i]$
    $Q \leftarrow Length[i]$
    $i \leftarrow 2n$
  else
    $q \leftarrow Length[i]$
    if ($p \neq 0$ and $|\frac{q}{p} - 1| < |ratio - 1|$) **then**
      $ratio \leftarrow \frac{q}{p}$
      $P \leftarrow p$
      $Q \leftarrow q$
    end if
    $p \leftarrow q$
    $i \leftarrow i+$ count($Length[i]$)
  end if
end while
return (P,Q)

Running time analysis:
The running time of QuickSort is $O(n \log n)$. The function count(x) is used to count the number of x. Because we have sorted the array, when we use count(), we are just keeping on traversing the array, so that in the worst case we traverse the array for one time. In best case, we get the solution immediately when we traverse the array because the number of the first element is no less than four. Moreover, one comparison or one update costs $O(1)$. Hence 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.

Main idea:
In this case we can use Greedy Algorithm. Suppose there are n children and m cakes. Array *Expected* is used to store children's expected sizes and array *Available* is used to store the cake pieces that I have. In order to make the most children satisfied, we need to satisfy those need least first. Firstly, sort the two arrays in ascending order. Then check whether the first cake can satisfy the first child. If not, remove the cake. If yes, satisfy this child and consider the next child with the same method.

Pseudocode:

Algorithm 3 Cake($Expected(e_1, e_2, ..., e_n), Available(a_1, a_2, ..., a_m)$)

```
QuickSort(Expected)
QuickSort(Available)
number ← 0
cake ← 1
for i from 1 to n do
  while Expected[i] > Available[cake] do
    cake ← cake + 1
    if (cake > n) then
      return number
    end if
  end while
  number ← number + 1
end for
return number
```

Proof of Correctness:
If all the m cakes are given out, then it must be optimal. Suppose only $m-1$ cakes are given out and the algorithm is not optimal, then it must exist a cake that can satisfy the m'th child. However, the m'th child can never be satisfied because there does not exist a cake that is larger than his expected size. This form a contradiction. Hence, the algorithm is correct.

Running time analysis:
QuickSort costs $O(n \log n) + O(m \log m)$. Checking whether the cake can satisfy the child costs $O(n)$. Hence, if $m > n$, the running time is $O(m \log m)$. Else, the running time is $O(n \log n)$.

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:
In order to invoke 'check' program as few as possible, we need to invoke it the time most number of programs overlap first. Firstly, we need to sort the finish time of the programs in ascending order. Then we can check the first finish time, pop it, delete all the program whose start time is before the minimum finish time. The left minimum finish time is the next time point to invoke 'check'. Apply this method until all the programs have been deleted.

Pseudocode:

Algorithm 4 Program($P_1(s_1, f_1), P_2(s_2, f_2), ..., P_n(s_n, f_n)$)

```
array P ← HeapSort the n programs with the finish time f_i(1 ≤ i ≤ n) in ascending order.
S ← ∅
while P != empty() do
  S ← S ∪ {P[1].f}
  P ← P.pop(P[1])
  n ← n − 1
  for i from 1 to n do
    if (P[i].s ≤ P[j].f) then
      delete P[i]
      n ← n − 1
    end if
  end for
end while
return S
```

Proof of Correctness:

1. Whenever the 'check' program is invoked during the running time of a program, we tag this program True. For those start before $A[1].f$, they are tagged True.

2. Because we sort the programs with their finish time in ascending order, whenever we meet a program that starts after A[1], it means that it is the first-finish program excluding previous programs and it needs another invoking.

3. Assume that algorithm S1 is optimal, and the Algorithm 4, denoted by S2 is not optimal. For two adjacent invoking v_k and v_{k+1} in S2, and the corresponding adjacent invoking i_k and i_{k+1} in S1, $v_k = i_k$, $v_{k+1} > i_{k+1}$. A program may exist whose start time is later than i_{k+1}, and finish time is later than v_{k+1}, so S1 needs one more invoking. This forms a contradiction.

4. After the traversals, all the time point that contain most programs overlapping can be found in this way. Hence, the number of invoking can be reduced to least. The algorithm is right.

Running time analysis:
The heap sort costs $O(n \log n)$. In the worst case, we traverse the programs for n times, and each comparison costs $O(1)$. Hence, the running time is $O(n^2)$.

Another method:
Main idea:
In order to invoke 'check' program as few as possible, we need to invoke it the time most number of programs overlap first. Firstly, we need to sort the finish time of the programs in ascending order. Then we can check the first finish time, and among all the programs that contain this point, choose the latest finish time and check it. Apply this method until all the programs contain at least one checked point.

Pseudocode:

Algorithm 5 Program$((s_1, f_1), (s_2, f_2), ..., (s_n, f_n))$

```
array A ← Sort the n tuples with the finish time f_i(1 ≤ i ≤ n) in ascending order.
S ← {A[1].f}
j ← 1
for i from 2 to n do
  if (A[i].s > A[j].f) then
    S ← S ∪ {A[i].f}
    j ← i
  end if
end for
return S
```

Proof of Correctness:

1. Whenever the 'check' program is invoked during the running time of a program, we tag this program True. For those start before $A[1].f$, they are tagged True.

2. Because we sort the programs with their finish time in ascending order, whenever we meet a program that starts after A[1], it means that it is the first-finish program excluding previous programs and it needs another invoking.

3. After one traversal, all the time point that contain most programs overlapping can be found in this way. Hence, the number of invoking can be reduced to least. The algorithm is right.

Running time analysis:
The sorting costs $O(n \log n)$. We traverse the programs for one time, and each comparison costs $O(1)$. Hence, the 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.

Main idea:

Use an array G to store the need of the guests. Firstly, traverse G, and if for an element $l_i = r_i (i = 1, ..., n)$, put it in an empty table, and delete it from G. Use array L to store $l_1, l_2, ..., l_n$, and use array R to store $r_1, r_2, ..., r_n$. Sort L and R in descending order. Then traverse R. We denote that the guest in R is gr. Traverse L. We denote the guest in L is gl. If $max\{L_{gl}, R_{gl}\} + max\{L_{gr}, R_{gr}\} > max\{L_{gl}, R_{gr}\} + max\{L_{gr}, R_{gl}\}$, consider gl and gr as one guest gr such that $L_{gr} = L_{gr}$ and $R_{gr} = R_{gl}$. Finally, apply the previous step recursively until the inequality does not hold. Delete all the guests having been selected, and apply the previous two steps until all the guests are selected.

Pseudocode:

Algorithm 6 Guests(G, L, R)

```
Construct an array T.
m ← 0
for (i from 1 to n) do
  if (l_i == r_i) then
    T[m] ← guest k
    m ← m + 1
    A ← A that k is removed
  end if
end for
i ← 0
while R is not empty do
  for m from 1 to the size of L do
    if max{L_gl, R_gl} + max{L_gr, R_gr} > max{L_gl, R_gr} + max{L_gr, R_gl} then
      T[m] ← gr
      T[m++] ← gl
      R_gr ← R_gl
      delete gl
    end if
  end for
end while
```

Proof of Correctness:

If we let the guest whose needed number of chairs on the left is the largest and the guest whose needed number of chairs on the right is the largest, we can use smallest number of chairs to make everyone happy.

Running time analysis:
Due to the while and for loop, the running time is $O(n^2)$

Another method:
Main idea:
Use merge sort to sort the left and right empty chairs that are needed and store them respectively into two arrays L and R in ascending order. Then put $L[i].guest$ to the right of $R[i].guest$. Notice that if they are the same, put them to another table.

Pseudocode:

Algorithm 7 Guests(L, R)

```
MergeSort(L)
MergeSort(R)
for (i ≤ n and j ≤ n) do
  if (L[i].guest has been assigned) then
    i ← i + 1
  end if
  if (R[j].guest has been assigned) then
    j ← j + 1
  end if
  if (L[i].guest == R[j].guest) then
    a new table← L[i].guest
  else
    the right side of R[j].guest ← L[i].guest
  end if
end for
```

Proof of Correctness:
We can find that the sum of chairs == the sum of $\max\{pair_i\}$, and $pair_i = (L_i, R_j)$. We need to sort L and R and make the elements in same position into a pair. Thus, the algorithm is correct.

Running time analysis:
Merge sort costs $O(n \log n)$. The traversal costs $O(n)$. Hence, the running time is $O(n \log n)$.