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 $O(\cdot)$ 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 the a and b (a,b>0) the long side and short side length. first select all the valid solution

transform the phase $\frac{C^2}{S} = \frac{4(a+b)^2}{a*b} = 4 * \frac{a^2+b^2+2ab}{ab} = 4(\frac{a}{b}+\frac{b}{a}+2)$

the $\frac{a}{b}+\frac{b}{a}$ get to min only when a=b. we can firstly save all the valid solution and traverse the minimum

Pseudocode

```
Algorithm: GetMin(int n):
    a <- initializeData(n);
    sort(a+1, a+1+n)
    n <- unique(a+1, a+1+n) - a - 1
    count <- 0
    for i in range(1,n)
        if(mp[a[i]] >1)
            count++
            b[count] = a[i]
        endif
    int ans1, ans2
    temp <- 0x3f3f3f3f
    for i in range(2,cnt)
        cmp = (b[i]/b[i-1]) + (b[i-1]/b[i])
        if cmp < temp then
            temp = cmp
            ans1 = b[i-1]
            ans2 = b[i]
        end if
    end for
    return temp.

initializeData(int n):
    mp <- map<int, int>
    int ans <- 0
    for i in range(1,n)
        mp[a[i]] += 1;
        if (mp[a[i]] == 4) ans = a[i];
    endfor
    return a
```

proof. of correctness: Suppose the greedy strategy is A. and A is not optimal. so there exists O, satisfy the first k-1 th choice is the same, while kth is not the same

Denote A kth element is a[k] a[k-1] while O kth is a'[k] a'[k-1]

In my implementation $\{a'[k]\ a'[k-1]\}$ $\frac{C^2}{S} = 4(\frac{a_k}{a_{k-1}} + \frac{a_{k-1}}{a_k} + 2) \leq 4(\frac{a'[k]}{a'[k-1]} + \frac{a'[k-1]}{a'[k]} + 2)$

which triggers the contradiction. O doesn't exists. ∴ greedy strategy is optimal.

Run time analysis: the map find takes O(1) and takes n time. so it's O(n)

compare $\frac{a}{b}+\frac{b}{a}$ takes O(1) and takes n time. so it's O(n)

the time complexity is O(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: ① sort the children by expected sizes and sort the cakes by the size of piece. both ascending order.
② We can probe the solution by scanning the children from the one with the lowest expected size and the list of pieces of cake from the one with the smallest size. Once the piece of cake satisfies a child, we just move the child to the DoneList and remove all pieces of cake being scanned.
③ repeat the above procedure, until either child or cake list is empty finally output the length of DoneList.

Pseudocode

```
Algorithm: GiveExpectedCakes(int* children, int* cakes):
    n ← len(children)
    sort(children+1, children+1+n)
    sort(cakes+1, cakes+1+n)
    i ← 0
    while i ≤ n do
        j ← 1
        while j ≤ n do
            if children[i] <= cakes[j] then
                Donelist.append(children[i])
                j ← j+1
            end if
            if children.isempty() or cakes.isempty() then
                return Donelist
            end if
        end while
    end while
```

Proof of correctness Suppose that there exists an optimal solution which satisfies a child with higher expected size than those without higher expected size.
① If we take the cake for the child with large expected size to unsatisfied child. the number of satisfied children doesn't change. so, we get another optimal solution
② repeat the process and the greedy algorithm can be used to select the optimal children.
Suppose that there exists an optimal solution, that the children with less expected size are treated with larger pieces of cake and vice versa.
① If we invert the two pieces of cake, we'll see both are satisfied. so we just get to another optimal solution.
② Repeat the process until the greedy method in picking cake is optimal.

Runtime analysis. quick sort takes O(n log n).

in the loop the worst situation is do n times O(n), it's O(n)

So the overall time time complexity 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: ① Sort the set of n points $X:\{x_1, x_2, x_3, x_4 \cdots x_n\}$ that the 'check' program may be invoked.

② scan the set from x_1 one by one, once meeting x_i ($i \in [1,n]$) just put $[x_i, x_{i+1}]$ into the solution set S. meantime, remove all the points in the set covered by $[x_i, x_{i+1}]$

③ repeat ② till the last element in set and output S.

Psendocode:

```
Algorithm: OutputExecution(init X)
    n ← len(X)
    sort(X[1, x+n+1])
    for i in range(1,n)
      S.append([X_i, X_i+1])
      for e in X:
        if e is in [X_i, X_i+1])
          S.remove(e)
        end if
      end for
    end for
    return S
```

proof of correctness. suppose that there exists an optimal solution S', such that y_1 is covered by $[x', x'+1] \in S'$. and $x' < y_1$. ∵ y_1 is the left most smallest in the sorted set. there should not be any other point lying in $[x', y_1)$.

If we replace $[x', x'+1]$ in S. by $[y_1, y_1+1]$, we'll get another optimal solution

Repeat solving the remaining subproblem, we get the greed strategy to optimal

Run the analysis: quick sort takes $O(n \log n)$.

in the loop the worst situation is obo n^2 times $O(1)$, $\#1$ $O(n^2)$

So the overall time the time complexity is $O(n^2)$

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.

Question: every person has 2 indicators l_i, denothy ith person has at least l_i in the left of him, r_i denotes ith person has at least r_i empty places on his right.

Main idea. consider their indicators as (a, b) and (c, d), suppose they form the circle themselves respectively, then the answer must be $2+\max(a,b)+\max(c,d)$ If put them in a circle, the answer is $2+\max(a,d)+\max(b,c)$, so the overall answer is $n+\sum_{i=1}^{n}\max(l_i, r_i)$, we just need to minimize the rest part. so have a sort for l and r is the final answer.

Pseudo code

Algorithm findMinTable(int* l, int* r, int n):

```
sort (l+1, l+1+n)
sort (r+1, r+1+n)
count←n
for i in range(1,n)
    count← count + max(l[i], r[i])
end for
return count.
```

denote sort & max STL C++.

proof of correctness. Suppose $X=\{x_1, x_2 \cdots x_k\}$ is the optimized solution where the value is t_k, $X'=\{x'_1, x'_2 \cdots x'_k\}$ is the greedy solution where the value is t'_k,

if respectively ①, $t_k = 2+\max(a,b)+\max(c,d)$

if circle ②, $t_k = 2+\max(a,d)+\max(b,c)$

if ① $t'_k < 2+\max(a,b)+\max(c,d) \Rightarrow t'_{k+1} < t_{k+1}$, X' is not optimized, contradiction

if ② $t'_k < 2+\max(a,d)+\max(b,c) \Rightarrow t'_{k+1} < t_{k+1}$, X' is not optimized, contradiction

$\Rightarrow$ greedy strategy is the optimized.

Running time analysis the quick sort takes $O(n\log n)$

the greedy part takes $O(n)$

$\Rightarrow$ the overall time complexity is $O(n\log n)$