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.

1.Main idea

1) put the lengths of the 2n sticks into an array A.
2) sort the array with heap in ascending sequence.
3) traverse the array and get the elements which have exists twice or more, put them into 2-dimension array B.(B[i][0] is the element, B[i][1] is the number of existence of the element)
4) (i) find whether for any i, B[i][1]>=4, if true, then pick 4 sticks with length B[i][0], and C^2/S is min.
(ii) since the elements are sorted, traverse the array B (from the second element to the last element), calculate B[i][0]/B[i-1][0], sort them and find the minimum of them (suppose it is B[m][0]/B[m-1][0]) ,then pick 2 sticks with length B[m][0] and 2 sticks with length B[m-1][0].

2.Pseudocode

```
A[2n]
B[2n]
input the length of the 2n sticks into A
heap_sort(A) (with ascending sequence)
for ( i =0 to 2n-1)
{
    if(A[i]==A[i+1])
    {
        if (A[i] is not in B){put A[i] into B}
        if (A[i] is in B){B[k][1]++} //suppose B[k]==A[i]
    }
}

for (i=0 to the size of B-1)
{
    if ( B[i][1]>=4)
        {
            Return:Pick 4 sticks of length B[i][0]
        }
    else if( 0<i )
        {
        put B[i][0]/B[i-1][0] into array C
        }
}

heap_sort(C)
pop(C)
Return: Pick 2 sticks of length B[m-1][0] and 2 sticks of length B[m][0]
//Supposing the minimum of C is B[m][0]/B[m-1][0]
```

3. Proof of correctness

Denote X,Y are the lengths of 2 adjacent sides.
S=X*Y, C=2(X+Y)

$$\frac{C^2}{S^2} = \frac{4(X+Y)^2}{XY} = 4 \cdot \frac{X^2+Y^2+2XY}{XY}$$

When |X-Y| is the minimum, $\frac{C^2}{S^2}$ is minimum.
We can use heap sort to find the minimum |X-Y|.

4. Running time analysis

1. put elements into array A is O(n)
2. heap sort the array A is O(n log n)
3. traverse array A and implement array B is O(n)
4. traverse array B and implement array C (if necessary) is O(n)
5. heap sort array C is O(n log n)

So, the total time complexity 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.

1.Main idea

1) input the expectations of children into array A, the sizes of the cake into array B.
2) heap sort the array A and array B.(by descending sequence)
3) compare the elements in A and B, starting from A[0] and B[0].if A[i]>B[m], then compare A[i+1]and B[m].if A[i]≤B[m], count+1, then compare A[i+1] and B[m+1].

2.Pseudocode

input the expectations of children into array A, the sizes of the cake into array B.
count=0
heap_sort (A) (by descending sequence)
heap_sort (B) (by descending sequence)
while(m<size of B, i<size of A)
{
if ($A[i] \leq B[m]$) then{count++,i++,m++}
else {i++}
}
Return : count

3.Proof of correctness

to see if the algorithm in main idea is optimal.
supposing the optimal is :
denote the children who get the cakes(in ascending sequence) are $a_1, a_2....a_i$, the cakes given them are $b_{k_1}, b_{k_2}....b_{k_i}$.
if there exists a_m and a_n, such that $a_m \leq a_n$ and $b_{k_m} \geq b_{k_n}$
then you can swap b_{k_m} and b_{k_n}, since $b_{k_m} \geq b_{k_n} \geq a_n \geq a_m$. which is still optimal.
therefore, the algorithm in main idea is optimal.

4.Running time analysis

supposing the given size of children's expectations and cake pieces are O(n).
heapsort array A and B are O(n log n).
comparing the elements in A and B is O(n).
Therefore, the total 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.

1.main idea

1) implement array A with the programs, heap sort the array by ascending sequence of their finish time.
2) delete all the program whose start time is before the minimum finish time in array A. the minimum finish time is the time to invoke "check " program, pop(A)
3) recursively call step 2, until A is empty.

2.pseudocode

implement array A with the programs.
implement array B to store the time to invoke "check"
impletement heap(A)
while(A is not empty)
{
put top(A) into array B.
delete the elements whose start time $\leq$ the finish time of top(A)
pop(A)
}

3.proof of correctness

supposing algorithm $S*$ is optimal, which needs to invoke equal to less times of "check" than the algorithm in main idea (S).
for 2 adjacent invoking d_k and d_{k+1} in S, and corresponding 2 adjacent invoking e_k and e_{k+1} in $S*$
$d_k = e_k$, $d_{k+1} > e_{k+1}$
it may exist a program, whose start time is larger than e_{k+1},and finish time is ~~smaller~~ larger than d_{k+1},then $S*$needs 1 more invoking.
Therefore, S is the optimal.

4.running time analysis

1.implement array A and B is O(n)
2.implement heap is O(n log n)
3. delete all the program whose start time is before the minimum finish time in array A. the minimum finish time is the time to invoke "check " program, is O(n)(which will be recursively called O(n)times)
4.pop(A) is O(n log n) (which will be recursively called O(n)times)
The total 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.

1.main idea

1) implement all the need of the guests into array A.

2) traverse the array, for all the guests such that $l_k = r_k (k = 0, 1, ...n-1)$, set each of them in an empty table, and delete them from A.

3) implement array B with $l_0, l_1, ..., l_{n-1}$, implement array C with $r_0, r_1, ..., r_{n-1}$

merge sort B,C in descending sequence.

4)start from C[0] (suppose the guest is K),traverse guests in order of descending sequence of necessary left empty chairs(suppose the guest is G).if $max\{l_G, r_G\} + max\{l_K, r_K\} > max\{l_G, r_K\} + max\{l_K, r_G\}$ merge G and K into one guest K such that $l_K = l_K$,$r_K = r_G$.

5)recurse step 4, until no G such that $max\{l_G, r_G\} + max\{l_K, r_K\} > max\{l_G, r_K\} + max\{l_K, r_G\}$ then delete all the guests who have been selected, and switch to a new table, and recurse step 4,5 until all the guests are selected.

2.persudocode

implement array T(to store the guests settled in each table)
implement all the need of the guests into array A.
m=0
for (k=0 to n-1)
{
if($l_k = r_k$)
put guest k into T[m]
m++
delete guest k from A
}
}
i=0 while (C is not empty)
{
supposing C[i] is corresponding to guest K.

for(m=0 to the size of B))
{
supposing B[m] is corresponding to guest G.

if ($max\{l_G, r_G\} + max\{l_K, r_K\} > max\{l_G, r_K\} + max\{l_K, r_G\}$)
{
set G,K into T[m].(if K is merged by K and G before, then don't set K into T[m])

$r_K = r_G$
delete G
}
}
}

3.proof of correctness

separate the tables with the settled guests (or maybe empty), according to the algorithm in main idea into 2 parts S and $S*$.

supposing the optimal algorithm is T and $T*$, and there only exists a guest G in S, such that G is in $T*$.apart from G, S and T have same elements.

denote in S, G_L is the left guest of G,G_R is the right guest of G. In $T*$, $P_L$is the left guest of G,P_R is the right guest of G.

denote $|S|$ is the number of chairs in S.

$|T| = |S| - 1 + max\{r_{G_L}, l_{G_R}\} - max\{r_{G_L}, l_G\} - max\{l_{G_R}, r_G\}$

$|T*| = |S*| + 1 - max\{r_{P_L}, l_{P_R}\} + max\{r_{P_L}, l_G\} + max\{l_{G_R}, r_P\}$

$|T| + |T*| = |S| + |S*| + max\{r_{G_L}, l_{G_R}\} - max\{r_{G_L}, l_G\} - max\{l_{G_R}, r_G\} - max\{r_{P_L}, l_{P_R}\} + max\{r_{P_L}, l_G\} + max\{l_{G_R}, r_P\} \geq |S| + |S*|$

4.running time analysis

the total time complexity is O(n^2)