Homework #4
Introduction to Algorithms/Algorithms 1
600.363/463
Spring 2015

Due on: Tuesday, February 24th, 5pm
Late submissions: will NOT be accepted
Format: Please start each problem on a new page.
Where to submit: On blackboard, under student assessment
Please type your answers; handwritten assignments will not be accepted.
To get full credit, your answers must be explained clearly,
with enough details and rigorous proofs.

February 26, 2015

1 Problem 1 (20 points)

1.1 Problem 1.1 (10 points)

Suppose you are a counselor at Bank of Mars that's concerned about fraud detection, and they come to you with the following problem. They have a collection of n bank cards that they've confiscated, suspecting them of being used in fraud. Each bank card is a small plastic object, containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many bank cards corresponding to it, and we'll say that two bank cards are $equivalent$ if they correspond to the same account.

It's very difficult to read the account number off a bank card directly, but the bank has a high-tech "equivalence tester" that takes two bank cards and, after performing some computations, determine whether they are equivalent.

Their question is the following: among the collection of n cards, is there a set of more than $n/2$ of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them into the equivalence tester. Show how to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester.

Answer:
First, let's consider if there exists a set of more than $n/2$ of them are are all equivalent to one another among n cards, we say this set is a majority set in n cards. We can use Divide and Conquer strategy to solve this problem. When considering the process of divide, you may discover the following fact: if there is a majority set in n cards and we divide n evenly into two $(n/2)$-subsets, at least one of the subsets will have a majority set, i.e. if none of the two subsets has a majority set, the total set will not contain a majority set. Therefore we can recursively solve the subproblem and finally get the result for the n cards problem. Label each card and denote the collection of n cards as array K.

```
FraudDetect($K$, l, h)
**if** $l == h$ **then**
    return $A[l]$;
**end**
**if** $l > h$ **then**
    **return** NULL;
**end**
**else**
    $p = \lfloor \frac{l+h}{2} \rfloor$;\\ Divide into two even subarrays
    lCandidate = FraudDetect($K$, l, p);
    rCandidate = FraudDetect($K$, p+1, h);
    Use equivalence tester to test lCandidate and rCandidate though
    $K[l, \cdots, h]$. If either lCandidate or rCandidate has an equivalence set of
    size larger than $(h - l + 1)/2$, return the right candidate. Otherwise,
    return NULL.
**end**
```

Time Complexity
Regarding the time complexity, we discover that the problem is divided into two half sided subproblems and needs additional $O(n)$ time to run the equivalence tester. Thus, $T(n) = 2T(n/2)+O(n)$, and by master theorem, $T(n) = O(n \log n)$.

Correctness

1. When $n = 1$, we set the indices $l, h = 0$. The algorithm will return $A[i]$ if $A[i] == A[j]$ or return NULL if $A[i] \neq A[j]$. Therefore, the algorithm is correct.

2. WLOG, let's assume when there are 2^k cards in K, FraudDetect(K,l,h) will return the correct result. We need to show the algorithm will return the cor-

rect result if there are 2^{k+1} cards in K. Since the algorithm is true when $n = 2^k$ and two FraudDetect(K,l,h) where $|K| = 2^k$, are needed, lCandidate and rCandidate are correct possible candidates of a majority set among 2^k cards. So after the equivalence tester, the right candidate will be returned and a majority set can be found correctly.

3. Our algorithm is correct.

1.2 Problem 1.2 (10 points)

The Tower of Hanoi is a mathematical puzzle. Given three rods and a stack of n disks arranged from largest on the bottom to smallest on the top placed on one of the rods, the puzzle asks for the minimum number of moves required to move the entire stack from one rod to another, where moves are allowed only if they place smaller disks on top of larger disks. Could you design an algorithm that provides an solution with minimum number of moves made? What is the minimum number of moves needed?

Answer:

Assume three rods are: A, B, C, and n disks are originally put on rod A. Consider a divide-and-conquer strategy: if we want to move the entire n disks from A to C, we should move the top $n-1$ disks to B first and then move the nth disk to C. Therefore, the current problem is to move the rest $n-1$ disks to another rod. We exactly use the same method and the size of the problem reduces to $n-2$. After recursively solve the subproblems, the entire n disks can be moved to C. Consider A as the original rod, B as the backup rod, and C as the destination rod, we design the following algorithm:

```
Hanoi(A,B,C,n)
if $n \geq 1$ then
    Hanoi(A,C,B, n-1); \\move top n-1 disks to the backup rod B
    Move $n^{th}$ disk from A to C;
    Hanoi(B,A,C,n-1);\\ move the n-1 disks to rod C
end
```

We need to solve the recurrence to get the minimal number of steps. From the above algorithm, $T(n) = 2T(n-1) + O(1)$. By solving the recurrence, $T(n) = 2^n - 1$.

Correctness

Use inductive method to prove the correctness:

1. When $n = 1$, Hanoi(A,B,C,1) will move the only one disk from A to C, which is a correct moving.

2. Assume when $n = k > 1$, Hanoi(A,B,C,k) will correctly move k disks from A to C. We need to show that Hanoi(A,B,C,k+1) will correctly move k+1 disks from A to C. Let's look into Hanoi(A,B,C,k+1), first it will execute Hanoi(A,C,B,k), which correctly move top k disks from A to B. Second, $(k+1)$th disk has been moved to C. Finally, Hanoi(B,A,C,k) correctly move the rest k disks from B to C. Till now, the entire k+1 disks have successfully been moved from A to C. Therefore, Hanoi(A,B,C,k+1) will correctly move k+1 disks from A to C.

3. Thus, Hanoi(A,B,C,n) is correct.

2 Problem 2 (20 points)

2.1 Problem 2.1 (10 points)

Take a sequence of $2n$ real numbers as input. Design an $O(n \log n)$ algorithm to partition the $2n$ points into n pairs, with the property that the partition minimizes the maximum sum of a pair. For example, if given a sequence $(2, 3, 1, 9)$, the possible partitions are $((2, 3), (1, 9))$, $(1, 2), (3, 9)$, and $(2, 9), (3, 1)$. Then the sums of the pairs are $(5, 10), (3, 12), (11, 4)$. Thus, the first partition has 10 as its maximum sum, which is the minimum over the three partitions.
Answer:
We can design an algorithm with two steps to solve this problem. Let denote the $2n$ real numbers as an array $A[1, \cdots, 2n]$. First step, sort the $2n$ real numbers in ascending order by merge sort. Second step, make pairs by putting the head and tail together one by one, i.e. $(A[1], A[2n]), (A[2], A[2n-1]), \cdots, (A[n], A[n+1])$.

Now let's talk about why this algorithm works. We can prove the correctness by inductive method.

1. When $n = 1$, there is only one pair and the algorithm is clearly correct.

2. Now let's assume when $n = k > 1$, the algorithm is correct. We need to show that when $n = k + 1$, the algorithm is still correct. By the algorithm above, the $k + 1$ pairs are $(A[1], A[2(k + 1)]), (A[2], A[2(k + 1) - 1]), \cdots, (A[k + 1], A[2(k + 1) - (k + 1) + 1])$. More formally, the pairs are $(A[i], A[2n + 1 - i])$ where $i = 1, \cdots, n$.

Since merge sort needs $O(n \log n)$ time and making pair takes $O(n)$, the total time needed is $O(n \log n)$.

Correctness Proof

I eliminated the proof here. An introduction proof might be a common way to show the correctness of this problem.

2.2 Problem 2.2 (10 points)

Please give an efficient algorithm to rearrange an array of n keys so that all the negative keys precede all the nonnegative keys. Your algorithm must be in-place, which means you cannot allocate another array to temporarily store the items. How fast is your algorithm?

Answer:

Since the problem requires in-place, the algorithm that solves the problem should be similar to the partition process in quick sort. Denote K as the input array of size n.

```
Rearrange(K)
Initilizaiton: i = 0
for j = 0 to $(K.size() - 1)$ do
    if $K[j] < 0$ then
        swap(A[i],A[j]);
        i++;
    end
end
return K;
```

The time complexity of the above algorithm is $O(n)$ since the for loop has n iterations and each iteration needs $O(1)$ to be executed.

Correctness Proof

I eliminated the proof here. Please refer to the correctness of quick sort.

3 Optional Exercises

Solve the following problems and exercises from CLRS: 8.2, 9.2, 2.2, 2.3-5, 2.3-7.