ESO207: Data Structures and Algorithms
Quiz 1 06/02/2019

Time 45 minutes Max Marks 25

Question 1. [Marks 8+7].

Let array A contain the integers of a set S_1 and B contain those of integer-set S_2. Algorithm 2 computes their intersection. Since S_1 is a set, no number is repeated. Same is true for S_2.

Input: Integers n, m and arrays A, B containing sets of n and m integers respectively
Output: Computes intersection of the two sets and return it in array C

```
k := 0;
for i := 0 to n − 1 do
    toadd := false;
    j := 0;
    while j < m AND toadd = false do
        if A[i] = B[j] then
            toadd := true;
        else
            j := j + 1;
        end
    end
    if toadd then
        C[k] := A[i];
        k := k + 1;
    end
end
return C;
```

Algorithm 1: Problem 1: Intersection of integer-sets

(a) Determine the time complexity of Algorithm 2. Show all the steps.

(b) Assume that the numbers in A and B are sorted in increasing order. Modify the algorithm **minimally** to improve the time complexity using the fact that A and B are sorted. *Do not perform time complexity analysis*.

Solution

(a) In the i-th pass of the For-loop we search array B to see if integer $A[i]$ is present in it. At each $B[j]$ there is a fixed set of operations to be performed so it take $O(1)$ time. So if k locations of B are visited in the While-loop, then the time cost will be proportional to $O(k)$.

The worst case will occur when the number $A[i]$ is not present in B and search takes time proportional to the length of B, i.e., $O(m)$. So the time to complete the For-loop is $c.m.n = O(n.m)$.

(b) If A and B both are sorted in increasing order, then we can avoid many useless searches. To understand this consider this situation. Suppose $a_i = A[i]$ and $b_j = B[j]$ for all i, j. While searching a_i suppose we reach j-the slot in B and we find that $a_j < b_j$. Then we do not have to search beyond. Also $a_{i+1} > a_i$ so we do not have to start the search at any location before j-th slot because they are all smaller than a_i so they are also smaller than a_{i+1}. SO the search for a_{i+1} must start from b_j.

Based on this observation we have the following modified algorithm.

Input: Integers n, m and sorted arrays A, B containing sets of n and m integers respectively
Output: Computes intersection of the two sets and return it in array C

$k := 0;$
$j := 0;$
for $i := 0$ *to* $n-1$ **do**
  $toadd := false;$
  while $j < m$ *AND* $A[i] > B[j]$ **do**
    $j := j+1;$
  end
  if $A[i] = B[j]$ **then**
    $C[k] := A[i];$
    $k := k+1;$
    $j := j+1;$
  end
end
return C;

Algorithm 2: Problem 1: Intersection of integer-sets

Analysis Although the analysis was not required, here we will do so to understand how good is this algorithm.

The total cost of this algorithm can be bounded by a constant times the total number of comparisons performed. Suppose The search for $A[i]$ starts at $B[j]$ and ends at $B[j+r]$ where $r \geq 0$. To compute a bound for the total number of comparisons let us charge 1 rupee to $A[i]$ for the first comparison, i.e., with $B[j]$. Subsequently for each remaining comparison charge 1 rupee each to $B[k]$ for $j+1 \leq k \leq j+r$.

Clearly each $A[i]$ will be charged 1 rupee so total charge to array A is n. Now consider any B location. If $B[j]$ is compared by $A[t], A[t+1], \ldots, A[t+p]$. Then $B[j]$ was the last slot with which $A[t]$ was compared. So 1 rupee was charged to $B[j]$ for it. But for $A[t+1], \ldots, A[t+p]$ the comparison with $B[j]$ was their first slot so no money was charged to $B[j]$ for these comparisons. Hence each B-slot will be charged at most 1 rupee. Hence total cost is $n+m$. Thus the time complexity is $O(n+m)$.

Question 2. [Marks 1+3+6].

Given k sorted sequences $S_1, S_2, \ldots, S_k$ of integers in array $A_1, \ldots A_k$ respectively. The elements in A_i are stored in $A[0 : n_i - 1]$. Assume that the sequences are sorted in increasing order.

Consider Algorithm 3.

(a) What does Algorithm 3 compute?

(b) Derive the worst case time complexity of this algorithm.

(c) If a sorting algorithm is designed (do not write the algorithm) using Algorithm 3, then what will be its worst case time complexity? Derive it accurately showing all steps.

Solution

(a) It merges the given k sorted sequences in a single sorted sequence.

(b) Initially k items (first item from each array) is inserted in the heap. Subsequently if the removed item is from A_i then we place the next item of A_i in the heap, if p_i has not reached the end of A_i. So at all times the heap has at most k items.

Let us analyze the code. The first and the second For-loops take $O(k)$ time. In the last For-loop we perform one $Remove-Top$ and at most one $Insert$ operation. Each operation will take $O(\log_2 k)$ time. So total time for this loop is $O(N. \log_2 k)$.

(c) A sorting algorithm can be designed using this "k-way" merge just as we did with simple or "2-way" merge.

In this algorithm we will create n/k groups of k elements and merge each group. In the second step we will have n/k^2 groups each will have k sorted sequences computed in the first step. In j-th step there will be n/k^j groups. Each group will have kb sequences and each sequence will be a sorted sequence of k^{j-1}

```
Input: n_1, ..., n_k, A_1, ..., A_k
Create MinHeap H;
/* Heap stores items of the form (r,i) where r is a number extracted from A_i.
   And (r_1,i_1) ≤ (r_2,i_2) if r_1 ≤ r_2                                      */
for j := 1 to k do
|   p_j := 0;
|   Insert(H, (A_j[p_j], j));
end
N := 0;
for j := 1 to k do
|   N := N + n_j;
end
for j := 0 to N − 1 do
|   (a,i) := RemoveTop(H);
|   B[j] := a;
|   if p_i < n_i − 1 then
|   |   p_i := p_i + 1;
|   |   Insert(H, (A_i[p_i], i));
|   end
end
```

elements.

The cost of merging one group will be $c.k^j.\log_2 k$. So total cost of merging n/k^j groups will be $c.n.\log_2 k$. So we see that the cost of j-th step is $c.n.\log_2 k$. There will be r step where $k^{r-1} < n \leq k^r$. So $r = \lceil \log_k n \rceil$. So the total worst case time complexity is $O(r.n.\log_2 k) = O(n.\log_k n.\log_2 k) = O(n.\log_2 n)$.