Department of Computer Engineering, SVNIT, Surat

M Tech I - 1^{st} Semester, Mid-Semester Examinations

CO 603: Algorithms and Computational Complexity

Solution to Mid-Sem Question paper

11:30 hrs to 13:30 hrs, 2^{nd} November 2020

Instructions:

1. *Write your Admission number clearly in the answer books along with the other details. Write page numbers on all pages that you use.*
2. *Assume any necessary data but giving proper justifications.*
3. *Be brief, precise, clear and to the point in answering the questions. Unnecessary elaboration WILL NOT fetch more marks.*
4. *This is an open-notes exam. Students are allowed to carry with them only the class handouts in the exam hall.*
5. *Maximum marks = 30, Duration = 2.0 hrs. Each MAIN question carries 10 marks, maximum.*
6. *All sub-questions carry equal marks unless stated otherwise.*

1. (a) Consider a typical cryptosystem used between Virat and Rahul for their confidential communication. The system parameters they use are as follows: the system modulus is n, the encryption exponent is e, the decryption exponent is d. The message M to be encrypted to a ciphertext C is done as $C = M^e \ mod \ n$. The exponent e is at least 2048 bits in size and hence the exponent operation is computationally a very costly operation. Specify what would be the complexity of finding M^e if the size of e is $|e|$. Design an algorithm $MyExponent(M, e)$ that computes the exponent with complexity $lg \ \ |e|$. Analyze mathematically and **prove with a mathematically sound proof** that the complexity of $MyExponent(M, e)$ is of logarithmic order. [4]

(a) The complexity of finding M^e if the size of e is $|e|$, using the conventional exponentiation operation would be given by $O(|e|)$, since the conventional exponentiation operation requires $|e| - 1$ multiplication operations.

(b) However, it is desired to have an algorithm with the complexity better as compared to $O(|e|)$. The algorithm $MyExponent(M, e)$ that we show below is based on binary exponentiation operation, wherein the exponent in treated as a binary number and appropriate further multiplications aare carried out. The algorithm is shown below:

```
Algorithm BINEXPONENT (x, m):
1.      let ans = 1
2.      divide 2 into m giving quotient q & remainder r
3.      if r = 1
4.              then ans = ans * x
5.      if q = 0
6.              goto exit
7.      let m = q
8.      let x = x * x
9.              goto step 2
10.     exit
```

(i) First, we note that the core operations in the algorithm are multiplication and division. The statements that involve these two operations are viz. 2,4 and 8. Rest other operations are assignments or initialization. Hence we can ignore the cost of execution of all other statements except for the statements viz. 2,4 and 8.

(ii) Also, as we see the statement numbers 4 and 8 are executed only if the statement no 2 is executed. Therefore, if the statement no 2 is executed k times, then the total no of times these three statements are executed is $3k$ times - to be specific, ...executed $3k - 1$ times; because in the last iteration, statement 8 is not executed.

(iii) Thus, in order to analyze the complexity of the algorithm, we need to figure out how many times statement no 2 is executed. That is, analyze how times the division by 2 in statement no 2 occurs.

(iv) First, we note that if r bits go into representation of m, then the number of times m can be divided by 2 is given by r. This can be shown by considering different numbers and relating the number to the bit representations that go into m and the number of times we divide the number.

(v) Secondly, we note that if r is the number of bits that go into representation of m, then we have the following expression as true:

$$2^{r-1} \leq m \leq 2^r$$
$$\Rightarrow 2^{r-1} \leq m$$
$$\Rightarrow r \leq lg\, m + 1$$
$$\Rightarrow r = \lfloor lg\, m + 1 \rfloor$$

(vi) But, we recollect that the number of bits that go into representation of a number m i.e. r denotes the number of times we divide m in statment 2 in the algorithm. Therefore, the number of times we divide in statement 2 is given by $\lfloor \lg\, m + 1 \rfloor$.

(vii) Thus, the complexity of the algorithm is given by $3\lfloor (lg\, m + 1) \rfloor$. That is, the asymptotic complexity of the algorithm is $O(lg\, m)$.

(b) For non-negative functions $f(n)$ and $g(n)$ indicate whether each of the following statements is true or false. Justify each alternative with a short proof or a simple counterexample. [4]

- If f(n)=O(g(n)), then log(f(n)) = O(log(g(n)))

Answer: This is not true for all the values of n. Observe that if f(n)=2 for all n and g(n)=1 for all n, then, f(n) ≤ cg(n) is true i.e. f(n)=O(g(n)), for c=3 and for all >n_0=1. However, log (f(n)) = log(2) = 1, but log(g(n))= log (1) = 0. Therefore, log(f(n)) is ! ≤ log(g(n)). And hence the statement is not true.

But the statement is true for all n > n_1 such that g(n) ≥ 2.
We can prove so as follows:
f(n) = O(g(n)). Therefore, f(n) ≤ c g(n) for all +ve constants c and n s.t. 0 ≤f(n) ≤cg(n)), $n \geq n_0$.
Therefore, log(f(n)) ≤ log (c.g(n)) = log c + log (g(n)). Thus, now we have to show that log c+ log g(n) ≤ c' log(g(n)) for some arbitrary positive constant c'
LEt us assume that g(n) → ∞ as n → ∞.
Therefore, log (g(n)) ≥ 1 for all sufficiently large n and for all n ≥ n_1 such that n = max $\{n_0, n_1\}$

- If f(n)=$\theta g(n)$) then $2^{f(n)} = \theta(2^{g(n)})$

This statement is false. This can be proven by taking the following counterexample.
Let f(n) = 2n and let g(n) = n.
Then, f(n) = $\theta g(n)$ is true.
However, $2^{f(n)} = 2^{2n} = 4^n$, whereas $2^{g(n)} = 2^n$. Obviously $2^{f(n)} = 2^{2n} = 4^n$ cannot be less than 2^n.

(c) An interger array $A[n]$ has $O(1)$ distinct elements. Explain the meaning of this statement with the help of a clear example. [2]

(i) The notation $f = O(1)$ indicates the asymtpotic growth rate of the function $f()$ is of the order of 1. That is it is constant and does not grow with the growth in the input.

(ii) This can be illustrated with the example as follows: Say we have an array $A[]$ with 110 elements, has 100 distinct elements. Now, if the number of distinct elements in the array $A[]$ is $O(1)$, then it means that even if the array size grows i.e. say the size of array grows to 1000, then also the number of distinct elements in this new array with elements 1000 would be the same as before i.e. 100.

2. (a) A number of websites use collaborative filtering to *identify* people with similar taste and then push the related contents to the entire collection. An algorithm *Counting-Inversion(A,B)* can be used to find how dis-similar the choices of two persons are with respect to a particular subject. Design an algorithm $Count - Inversion(A[])$ to solve this problem, that takes as argument the choices of one person illustrating how the approach used in the Merge-Sort can be used for the purpose. [Assume the choices of the other person are in ascending order already known] [5]

- The algorithm $Count - Inversion(A[])$ can be depicted as shown below. The algorithm is structured into two algorithms such that the algorithm Merge-and-count(A, B) is called by the main algorithm Sort-and-Count(L).

```
Algorithm Merge-and-count(A, B)
Maintain a Current pointer into each list,
        initialized to point to the front elements
Maintain a variable Count for the number of inversions,
        initialized to 0.
While both lists are nonempty {
        Let a_i and b_j be the elements pointed to
                by the Current pointer
        Append the smaller of these two to the output list
        If b_j < a_i then
                increment Count by the number of elements
                remaining in A
        Advance the Current pointer in the list from
                which the smaller element was selected.
        }

        Algorithm Sort-and-Count(L):
        \\Pre-condition: [Merge-and-Count]
A and B are sorted.
        \\ Post-condition. [Sort-and-Count]
L is sorted.
1.      if list L has one element
2.              return 0 and the list L
3.      Divide the list into two halves A and B
4.      (r_A, A) ← Sort-and-Count(A)
5.      (r_B, B) ← Sort-and-Count(B)
6.      (r, L) ← Merge-and-Count(A, B)
7.      return r = r_A + r_B + r and the sorted list L
```

- This algorithm is similar to the Mergesort algorithm in that it uses a similar procedure merge but uses count to count the no of inversions within the two subhalves and then counts the inversions across the two halves, while merging.

(b) Given an array $a[0 : n-1]$ of n elements, it is required to determine the k^{th} smallest element, called a selection method. Derive a solution using divide and conquer method and perform complexity analysis. [5]

This can be done in O(N) time and O(1) space by the Quickselect Algorithm. The Quickselect Algorithm is a very useful divide-and-conquer based algorithm which rearranges the array such that the kth element is the kth smallest element in the array.
A step-by-step version of Quickselect is as follows:

1 Calculate the middle element for the given range.

2 Place all the elements greater than the mid element to its right.

3 Place all the elements lesser than the mid element to its left.

4 If 'k' is less than the index of the middle element, then recursively call Quickselect on the left half of the range.

5 If 'k' is greater than the index of the middle element, then recursively call Quickselect on the right half of the range.

Please see the example and explanation at the end of this document.

3. (a) Consider the use of Compact Discs (CD) that was prevalent till recent years. Note that the CDs are a Direct Access device unlike the sequential access devices, that predated the CDs. Assume that you are given a sequence of n songs, where the i^{th} song is l_i minutes long. You want to place all the songs on an ordered series of CDs (e.g., CD_1, CD_2, . . . , CD_k), where each CD holds m minutes. Furthermore,

- The songs must be recorded in the given order: $song_1$, $song_2$, . . . , $song_n$.
- All songs must be included. No song may be split across CDs.

Design a greedy approach algorithm to solve this problem and analyze its time complexity. [5]

- The outline of the algorithm to be used in this case is as follows:
- Record as many songs (from song 1 to song g) on the
  first CD as possible without exceeding the l_i minutes for the i^{th} song limit.
- Then recursively repeat this procedure for the remaining CDs.
- Since this just requires keeping a running sum in which each of the n song lengths are included once, clearly the above is an $O(n)$ algorithm.
- This approach can be formalized as follows:
  - First, find the largest set of songs that will fit on the first CD, i.e., find the largest integer j such that $\sum ji = 1li \leq m$.
  - Put the first minj,n songs on CD 1. If n $\leq$ j, you have added all of the songs and you are done.
    Otherwise, recurse on songs j + 1, j + 2, . . . and CDs 2, 3, ...
- We can also prove the approach used is correct by following the proof techniques for proving the greedy algorithms correct i.e. by showing that this algorithm exhibits, the greedy choice and the optimal substructure properties, as follows:
  - *Greedy Choice Property:* Let S be an optimal solution in which the
    first CD holds songs 1 to *opt*.
  - The greedy algorithm puts songs 1 to g on the
    first CD.
  - We now prove that there is an optimal solution which puts the
    first g songs on the
    first CD. If $k = g$ then S is such a solution.
  - Now suppose that $k \neq g$. Since the greedy algorithm puts as many songs on the
    first CD as possible, it follows that $opt < g$.
  - We construct a solution S' by modifying S to move all songs from $opt + 1$ to g onto the
    first CD.
  - We now argue that S' is a legal solution which is optimal.
  - First note that the number of CDs used by S' is at most the number of CDs used by S. All that remains is to argue that S' is legal (i.e. no CD holds more than m minutes of songs). By definition of the greedy choice the
    first CD can hold songs 1 to g. Since S is a legal solution all of the CDs in it hold at most m minutes. For all but CD 1 the number of songs is only reduced and hence must still hold at most m minutes. Hence S' is legal.
  - *Optimal Substructure Property:* Let P be the original problem with an optimal solution S. Then after putting songs 1 to g on CD_1, let P' be the subproblem of songs $(g+1)$ to n. Let S' be an optimal solution to P'. Since, $cost(S) = cost(S') + 1$, clearly an optimal solution to P includes within it an optimal solution to P'.

(b) Given n activities each with a defined start time s_i and finish time f_i, the problem is to select a maximal set of mutually compatible activities - wherein

- if each activity i occurs during the half open intveral $[s_i, f_i)$, then they are compatible if, $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.

Design a greedy algorithm to solve this problem. Give a formal proof that your algorithm actually works and analyze its time complexity. [5]

```
Algorithm Simple_ActivitySelection(Job, s_i, f_i)
/*A=Set of selected mutually compatible jobs*/
1.Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.
2.A ← φ
3.for j = 1 to n
4.        if f_j ≤ s_(j+1) /*(job (j+1) is compatible
with A)*/
5.        A = A ∪ {j}
6.return Selected_Jobs
```

Proof that the algorithm works :
Statement: For the interval scheduling problem, let the set A=i1,i2,i3,ik with —A—=k be the set of intervals returned by the our own algorithm as the answer and the set O=j1,j2,j3,jm be the optimal set of intervals returned by a random oracle algorithm that cannot go wrong and that always returns the correct answer. Prove that the our own algorithm returns the optimal answer Proof:

- First let us try to formulate the problem statement clearly before starting with our proof. That is, Let us try to reason as to what should be our goal to prove that our interval scheduling algorithm is optimal ?
- Well, the goal must be to prove that k = m..since the optimal answer is always correct and therefore returns the maximum number of activities.
- Now, observe, in this context, that our algorithm sorts the input activities in non-decreasing order of their finish times. This means that out of the entire INPUT job mix from which both our own algorithm and the optimal algorithm - would try to select their respective sets of activities, if we simply compare the finish times of the first activities in both the sets A and O, f(i1) ≤ f(j1). Thus, our greedy rule guarantees that f(i1) ≤ f(j1).
- Hence, eventually we should show that for every r ≥ 1, the rth request in the algorithms schedule finishes no later than the one in the optimal schedule. What is this equivalent to ?
- This is equivalent to showing that for every rth request, r ≥ 1, f(ir) ≤ f(jr). Thus, we restate the statement of the theorem to be proved as follows:

- Theorem: For the interval scheduling problem, if the set A=i1,i2,i3,ik with —A—=k is the set of intervals returned by the algorithm as the answer and the set O=j1,j2,j3,jm be the optimal set of intervals retruned by a random oracle, then prove that : For all indices rk, we have f(ir)f(jr)
- Proof: We shall use induction on r to prove the result as follows:
- Basis: We can easily state that f(i1)f(j1). This is so, based on our earlier observation that since our algorithm sorts the input activities in non-decreasing order of their finish times, out of all the jobs in the input job mix, the one selected by our algorithm must have the least finish time.
- Next, as per the induction hypothesis we assume that for any r¿=1, f(ir-1) *leq* f(jr-1)
- However, we can assume the above subject to the condition that we shall prove that f(ir)≤f(jr)
- Let us first observe that f(jr-1) ≤ s(jr). This has to be so, because the set O is the set of the optimal requests and so two activities selected in O must be compatible to each other.
- Now, since from IH we have, f(ir-1) ≤ f(jr-1) and from above observation we have f(jr-1) ≤ s(jr).
- Therefore, it must be the case that f(ir-1) ≤ s(jr).
- Therefore, this implies that interval jr is in the set R of available intervals to the algorithm at the time the greedy algorithm selects ir.
- Now, if interval ir and jr both are available for the algorithm to select, which out of the two, the greedy algorithm A would select ?

Naturally, since our algorithm sorts the jobs in terms of their finish times, out of the two jobs in contention viz. i_r and j_r - to be selected (that are compatible to each other), it would select the one with the least finish time. And we know, from the given data that our algorithm has indeed selected i_r.
Therefore, it must be the case that f(ir) ≤ f(jr). Therefore, proved.

*****

***

*