Midsem CS213M 2014

36 Marks, Closed Notes, 2 Hours. I have made every effort to ensure that all required assumptions have been stated. If absolutely necessary, do make more assumptions and state them very clearly. Whenever possible, describe your algorithm using pseudo code.

1. A program "HelloWorld" written by one of the cs213m TAs uses an implementation of the sequence ADT as its main component. It performs **atRank**(r) (returns element at rank r), **insertAtRank**(r) (inserts element at rank r) and **remove** (removes element from the end of the sequence) operations in some unspecified order. It is known that "HelloWorld" performs n^2 **atRank** operations, $2n$ **insertAtRank** operations, and n **remove** operations. Which implementation of the sequence ADT should the TA use in the interest of efficiency: the array-based one or the one that uses a doubly-linked list? Explain.

   Solution: Refer to class notes for running times of these methods.

	Array-based	**List-based**
atRank	$O(1)$	$O(n)$
insertAtRank	$O(n)$	$O(n)$
remove	$O(n)$	$O(1)$
Total time	$n^2O(1) + 2nO(n) + nO(n) =$ $O(n^2 + 2n^2 + n^2) =$ $O(n^2)$	$n^2O(n) + 2nO(n) + nO(1) =$ $O(n^3 + 2n^2 + n) =$ $O(n^3)$

   Since list-based implementation runs in $O(n^3)$ worst-case time, and array-based one runs in $O(n^2)$ time, we prefer the array-based one.

   (4 Marks)

2. For each of the following problems, you are allowed to use stacks or queues as auxiliary data structures. But you must use them only if the need be and use the minimum possible number of auxiliary data structures. All your algorithms must be non-recursive.

   (a) Describe in pseudo-code a linear-time algorithm for reversing a queue Q. To access the queue, you are only allowed to use the methods of

queue ADT.

SOLUTION: We empty queue Q into an initially empty stack S, and then empty S back into Q.

(2 Marks)

(b) Suppose you have a stack S containing n elements. Describe an efficient algorithm to scan S to see if it contains a certain element x, with the additional constraint that your algorithm must return the elements back to S in their original order. Analyse your algorithm for its time complexity.

SOLUTION: Use an auxiliary queue data structure Q. The solution is to actually use the queue Q to process the elements in two phases. In the first phase, we iteratively pop each the element from S and enqueue it in Q, and then we iteratively dequeue each element from Q and push it into S. This reverses the elements in S. Then we repeat this same process, but this time we also look for the element x. By passing the elements through Q and back to S a second time, we reverse the reversal, thereby putting the elements back into S in their original order. Time complexity is $O(n)$.

(2 Marks)

(c) Suppose you have a queue Q containing n elements. Describe an efficient algorithm to scan Q to see if it contains a certain element x, with the additional constraint that your algorithm must return the elements back to Q in their original order. Analyse your algorithm for its time complexity.

SOLUTION: In this case, all you need to do is keep dequeuing and enqueuing the dequeued element as many times as the size of the queue. Additionally, just check of the dequeued element is x. No auxiliary data structure is required. Time complexity is $O(n)$.

(2 Marks)

(d) You are given an array A of n distinct elements. You would like to generate an array B of all possible subsets of elements in A. Thus, the length of B will be 2^n. Each element of B can, in turn be stored as an array or linked list (of a subset of elements from A) or in any other convenient representation.

```
A = { a, b, c}
binary number  => resulting subset
000            => {     }  // no 'a',  no 'b', no 'c'
001            => {    c}
010            => {  b  }
011            => {  b,c}
100            => {a    }
101            => {a,  c}
110            => {a,b  }
```

```
111            => {a,b,c}


or

     set is {1, 2, 3, 4, 5}
count from 0 to 31:
count = 00000 => print {}
count = 00001 => print {1} (or 5, the order in which you do it really shouldn't matter)
count = 00010 => print {2}
        00011 => print {1, 2}
        00100 => print {3}
        00101 => print {1, 3}
        00110 => print {2, 3}
        00111 => print {1, 2, 3}
        ...
        11111 => print {1, 2, 3, 4, 5}
```

(4 Marks)

3. Consider a cryptographic scheme where a message is represented by a large prime number p. The message is encrypted using another large prime $q > p$, so that the actual message sent over the network is $r = p \cdot q$. Unfortunately, your message is intercepted by someone (hereafter referred to as the interceptor). Suggest an efficient decryption algorithm that the interceptor might use and present its worst-case time complexity.

   Now suppose you encrypted the message using $q > p$ such that q was the smallest integer larger than p that was co-prime[1] with p. And suppose the interceptor knew this. Could he have come up with a more efficient decryption algorithm? If yes, present the algorithm. If no, explain why.

   In both cases, please do consider that the numbers p, q and r are very large and hence the cost of addition, subtraction, division and multiplication of these large numbers will depend on the size of the numbers. For example, the schoolbook addition and subtraction methods take $O(n)$ time for two n-digit numbers.

   SOLUTION: The message can be decrypted using the following decryption algorithm:

   > For every integer p such that $1 < p < r$, check if p divides r. If so, stop and return p as the decrypted message, otherwise continue.

   What is the worst-case time complexity of this algorithm? Since the input r is stored as a string of bits, consider the input size n to be the number

[1] In number theory, two integers a and b are said to be co-prime if their greatest common divisor is 1.

of bytes needed to store r, that is, $n = (\log_2 r)/8$, and assume that each division takes time $O(n)$.

There are $\sqrt{r}$ iterations: $p \leq \sqrt{r}$ because the encryption key q is known to be larger than p. Since each iteration involves one division taking $O(n) = O(\log_2 r)/8 = O(\log r)$ time, the total worst-case running time is $O(\sqrt{r} \log r)$. In terms of n, this is $O(n2^n)$ because $r = 2^{8n}$.

(While it is true to say there are $O(r)$ iterations, and thus the running time is $O(r \log r)$, this is a case of not being as accurate as possible with the big-Oh notation.)

For the second part, q must be $p + 1$. So it amounts to solving $r = p^2 + p$ for known r and unknown p. The answer will be $p = \frac{-1+\sqrt{1+4r}}{2}$. This can be solved in time that is $O(\log_2 r)$.

(6 Marks)

4. Suppose you are given an n-element array A containing distinct integers that are listed in increasing order. Given a number k, describe a recursive algorithm to find two integers in A that sum to k, if such a pair exists. What is the running time of your algorithm?

   Solution: The solution makes use of the function FindPair(A, i, j, k) below, which given the sorted subarray $A[i..j]$ determines whether there is any pair of elements that sums to k. First it tests whether $A[i] + A[j] < k$. Because A is sorted, for any $j' \leq j$, we have $A[i] + A[j'] < k$. Thus, there is no pair involving $A[i]$ that sums to k, and we can eliminate $A[i]$ and recursively check the remaining subarray $A[i + 1..j]$. Similarly, if $A[i] + A[j] > k$, we can eliminate $A[j]$ and recursively check the subarray $A[i..j - 1]$. Otherwise, $A[i] + A[j] = k$ and we return true. If no such pair is ever found, eventually all but one element is eliminated ($i = j$), and we return false.

   (6 Marks)

5. Suppose that each column of an $n \times n$ array A consists of 1's and 0's such that, in any column of A all the 1's come above any 0's in that column. Assuming A is already in memory, describe an efficient algorithm for finding the column of A that contains the most 1's. What is the time complexity of your algorithm?

   Solution: The idea is to keep a running value of the max_row index of each column. Start at the top right of the matrix. Walk down the first column until a 0 is found. When the first 0 is found, set the max_row index to the current row index, then step up to the next column (without resetting the row position). If the next column at the max_row index has a 0, continue onto the next column. Else walk down the next column until a 0 is found and set the max_row index to the current row index. Continue this procedure until the rightmost column is traversed.

   The algorithm will run in $O(n)$ time.

(4 Marks)

6. Given a matrix (two dimensional array) $X[1,\ldots,n][1,\ldots,n]$ the span $S[i][j]$ of $X[i][j]$ is the maximum number of consecutive elements $X[p][q]$ immediately preceding[2] $X[i][j]$ (and including $p = i$ and $q = j$) such that $X[p][q] \leq X[i][j]$. Given a two dimensional array $X[1,\ldots,n][1,\ldots,n]$, present the most efficient algorithm you can, for computing the two dimensional array of spans $S[1,\ldots,n][1,\ldots,n]$. What is the time complexity of your algorithm.

   Hint: You could make use of one or more instances of the stack data structure.

   SOLUTION: Extend the solution on slide 20 of `http://www.cse.iitb.ac.in/~cs213m/notes/eNotes/2014_02_07.pdf` to two dimensions. I think the time complexity cannot be reduced below $O(n^4)$. Brute force algo is $O(n^4)$ and using two stacks (like in the class exercise) – one for row and another for column – can help ignore some blocks but cannot reduce complexity below $O(n^4)$. Following solution is courtesy Mandar and you can use a stack instead of the arrays

```
The earlier idea was to calculate row and column spans for each element X[i][j]
using stacks. The row span of any X[i][j] is the maximum number of consecutive
elements X[i][p] (p <= j) such that X[i][p] <= X[i][j]. Likewise, the column span
of any X[i][j] is the maximum number of consecutive elements X[q][j] (q <= i)
such that X[q][j] <= X[i][j]. Now the 2D span-matrix of X[i][j] cannot be larger
than the matrix that has (i - colspan(X[i][j]) + 1), (j - rowspan(X[i][j]) + 1)
as the top left corner and (i, j) as the bottom right corner. Within this sub-matrix,
we launch a brute-force search for our solution.

The search procedure is as follows.

For each X[q][j] such that q varies for i - 1 to (i - colspan(X[i][j]) + 1)
    Count the number of elements immediately to the left of X[q][j] (in row q)
    which are less than X[i][j]. Call this numElemY_q

For each X[i][p] such that p varies for j - 1 to (j - rowspan(X[i][j]) + 1)
    Count the number of elements immediately above X[i][p] (in col p)
    which are less than X[i][j]. Call this numElemX_p

S[i][j]  = min_q (numElemY_q) * min_p (numElemX_p)

However, it's easy to see that we do not need to pre-compute row and column spans
```

[2] That is, $S[i][j]$ will be the number of elements in the largest submatrix ending with $X[i][j]$ at the right hand bottom corner such that all elements in the submatrix are less than or equal to $X[i][j]$.

apriori. This can be done while doing the search itself.

(6 Marks)