CHENNAI MATHEMATICAL INSTITUTE

M.Sc. / Ph.D. Programme in Computer Science

Entrance Examination, 15 May 2013

This question paper has 4 printed sides. Part A has 10 questions of 3 marks each. Part B has 7 questions of 10 marks each. The total marks are 100. Answers to Part A must be filled in the answer sheet provided.

Part A

1. Ball Mart has 10^7 different items in stock across all its stores worldwide. The company has collected billing data for 10^{10} customer transactions. Each individual bill has at most 10 distinct items in it.

   Ball Mart's CEO wants to optimize the company's inventory and has asked for a list of those items that appear in at least 2% of the billed transactions. Which of the following is the most precise upper bound one can compute for the number of such items, given the data?

   (a) 500 (b) 1000 (c) 5000 (d) 20000

   Answer:

   (a) 500 items.

   An item that is in 2% of the bills must appear in 2×10^8 bills. Across all bills, there are at most $(10^{10}) \times 10 = 10^{11}$ items mentioned. So at most $(10^{11})/(2 \times 10^8) = 500$ items can appear in 2% of the bills. The number of items in stock is irrelevant. ⊣

2. 10% of all email you receive is spam. Your spam filter is 90% reliable: that is, 90% of the mails it marks as spam are indeed spam and 90% of spam mails are correctly labelled as spam. If you see a mail marked spam by your filter, what is the probability that it really is spam?

   (a) 10% (b) 50% (c) 70% (d) 90%

   Answer:

   (b) 50%.

   Out of 100 mails, 10 are spam. The filter will label 9 of 10 spam as spam and 9 of 90 non-spam as spam. So 18 are labelled spam, of which 9 are actually spam. You can compute the same result more formally using conditional probabilities. ⊣

3. When a user submits a query, a search engine does the following. For every webpage that has been visited by the search engine, it computes a score indicating how relevant that page is to the query. Finally, it reports the pages with the top k scores on the screen, for a number k specified by the user. A good data structure for accumulating the scores and ranking them is:

   (a) a queue (b) a heap (c) a stack (d) a binary search tree

Answer:

(b) A heap.

Let n be the number of pages visited by the search engine at the time a query is submitted. Assume that it takes constant time to compute the relevance score for each page w.r.t. a query. Then it takes $O(n)$ time to compute the relevance scores, a further $O(n)$ time to build a heap of n relevance scores, and $O(k \cdot \log n)$ time for k delete-max operations to return the top k scores. ⊣

4. Consider the set of all words over the alphabet $\{x, y, z\}$ where the number of y's is not divisible by 2 or 7 and no x appears after a z. This language is:

   (a) regular

   (b) not known to be regular

   (c) context-free but not regular

   (d) recursively enumerable but not context-free

Answer:

(a) Regular.

Let $\Sigma = \{x, y, z\}$. The language in question is $(\Sigma^* \setminus (L_1 \cup L_2)) \cap L_3$ where

$$L_1 = \{w \in \Sigma^* \mid \text{ the number of } y\text{'s is divisible by } 2\},$$

$$L_2 = \{w \in \Sigma^* \mid \text{ the number of } y\text{'s is divisible by } 7\},$$

and

$$L_3 = \{w \in \Sigma^* \mid \text{no } x \text{ appears after a } z\}.$$

It suffices to show that all these three languages are regular, and appeal to the fact that regular languages are closed under boolean operations.

They are described by the following regular expressions, respectively.

$$r_1 = ((x + z)^* y (x + z)^* y (x + z)^*)^*$$

$$r_2 = ((x + z)^* y (x + z)^* y (x + z)^* y (x + z)^* y (x + z)^* y (x + z)^* y (x + z)^* y (x + z)^*)^*$$

$$r_3 = (x + y)^* (y + z)^*$$

⊣

5. You have n lists, each consisting of m integers sorted in ascending order. Merging these lists into a single sorted list will take time:

   (a) $O(nm \log m)$ (b) $O(mn \log n)$ (c) $O(m + n)$ (d) $O(mn)$

Answer:

(b) $O(mn \log n)$ time or (d) $O(mn)$ time.

We can merge two sorted lists of size k and ℓ in time $O(k + \ell)$. We begin by merging the lists in pairs to generate $\frac{n}{2}$ lists of size $2m$ each, in total time $O(mn)$. If we repeat this, we get $\frac{n}{4}$ lists of size $4m$ each, again in total time $O(mn)$. Thus, in $O(\log n)$ rounds, we converge to a single sorted list. Each round takes time $O(mn)$, so the total time is $O(mn \log n)$.

Another strategy to achieve complexity $O(mn \log n)$ is to build a min-heap of size n, consisting the first element in each of the n lists. We then extract the minimum element from the heap to the output and replace it in the heap with the next element from the list from which the minimum came. Thus, it takes time $O(\log n)$ to generate each element in the output and there are $O(mn)$ output elements in all, so the overall time is $O(mn \log n)$.

On the other hand, if we can apply min() to n elements at a time, we can merge all n lists is parallel. Let the n lists be $\ell_1, \ell_2, \ldots, \ell_n$. We maintain indices $i_1, i_2, \ldots, i_n$ into these n lists, initially set to 1. At each stage we add $j = \min(\ell_1[i_1], \ell_2[i_2], \ldots, \ell_n[i_n])$ to the sorted output list and increment the corresponding index. Overall, we add mn elements to the sorted output and each element is generated in constant time, so the time taken is $O(mn)$.

⊣

6. A simple graph is one in which there are no self loops and each pair of distinct vertices is connected by at most one edge. Let G be a simple graph on 8 vertices such that there is a vertex of degree 1, a vertex of degree 2, a vertex of degree 3, a vertex of degree 4, a vertex of degree 5, a vertex of degree 6 and a vertex of degree 7. Which of the following can be the degree of the last vertex?

   (a) 3 (b) 0 (c) 5 (d) 4

   Answer:

   (d) 4.

   The number of odd degree vertices in any graph is even. Since we already have four vertices of odd degree, the degree of the last vertex cannot be 3 or 5. It cannot be 0 either, since there is one vertex with degree 7, which means that it is a neighbour to all the other vertices, which implies that there is no isolated vertex in the graph. Thus the only possible degree of the last vertex is 4. An actual graph that conforms to the specification is given in Figure 1.

   ⊣

7. Consider the following two statements.

   (A) There are infinitely many interesting whole numbers.

   (B) There are finitely many uninteresting whole numbers.

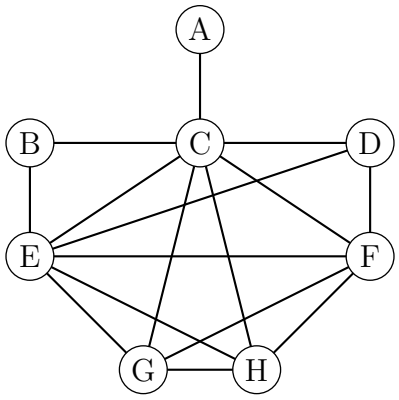

Figure 1: A graph meeting the constraints in Question 6.

Which of the following is true?

(a) Statements A and B are equivalent.

(b) Statement A implies statement B.

(c) Statement B implies statement A.

(d) None of the above.

Answer:

(c) Statement B implies statement A.

Since there are infinitely many numbers, if only finitely many are uninteresting, infinitely many have to be interesting. The converse is not true. It is possible that infinitely many numbers are uninteresting (say, all the even numbers) while at the same time infinitely many are interesting (say, all the odd numbers, since they are odd). Since A does not imply B, A and B are not equivalent. ⊣

8. In the passing out batch, 54 students know Java, 39 know Python and 43 know C++. Of these, 15 know both Java and Python, 17 know both Python and C++ and 23 know both Java and C++ and 11 know all three languages. If there are 100 students in the class, how many know none of these three languages?

(a) 3 (b) 8 (c) 17 (d) 19

Answer:

(b) 8.

Let J, P and C be the sets of students who know Java, Python and C++, respectively. Using the principle of inclusion-exclusion, the total number of distinct students in $J \cup P \cup C$ is given by $|J| + |P| + |C| - (|J \cap P| + |P \cap C| + |J \cap C|) + (|J \cap P \cap C|)$. This works out to $(54 + 39 + 43) - (15 + 17 + 23) + 11 = 92$. So there are 92 students who know at least one language and $100 - 92 = 8$ are left out. ⊣

The next two questions are based on the following program.

```
procedure mystery (A :  array [1..100] of int)
  int  i,j,position,tmp;
  begin
    for j := 1 to 100 do
      position := j;
      for i := j to 100 do
        if (A[i] > A[position]) then
           position := i;
        endfor
      tmp := A[j];
      A[j] := A[position];
      A[position] := tmp;
    endfor
  end
```

9. When the procedure terminates, the array `A` has been:
   (a) Reversed (b) Left unaltered
   (c) Sorted in descending order (d) Sorted in ascending order

   Answer:

   (c) Sorted in descending order.

   At iteration j of the outer loop, the inner loop determines the index *position* between j and 100 such that $A[position]$ is maximum, and $A[position]$ is interchanged with $A[j]$. Thus at the end, position $A[j]$ holds the j^{th} largest value in the array. ⊣

10. The number of times the test `A[i] > A[position]` is executed is:
    (a) 100 (b) 5050 (c) 10000 (d) Depends on contents of `A`

    Answer:

    (b) 5050.

    In iteration j, there are $100 - j + 1$ comparisons made. So in all there are

    $$100 + 99 + \cdots + 2 + 1 = 5050$$

    comparisons. ⊣

Part B

1. For a binary string $x = a_0 a_1 \cdots a_{n-1}$ define $val(x)$ to be the value of x interpreted as a binary number, where a_0 is the most significant bit. More formally, $val(x)$ is given by

   $$\sum_{0 \le i < n} 2^{n-1-i} \cdot a_i.$$

   Design a finite automaton that accepts exactly the set of binary strings x such that $val(x)$ is divisible by either 4 or 5.

Answer:

For n a natural number, let $L_n = \{x \in \{0,1\}^* \mid val(x) \text{ is divisible by } n\}$. Observe the following.

(i) For a binary string x, $x \in L_n$ iff $val(x) \mod n = 0$.

(ii) if $val(x) = i \mod n$ for a binary string x, then $val(xb) \equiv (2i + b) \mod n$, for $b = 0, 1$.

So to construct a DFA for L_n, it suffices to take n states $q_0, \ldots, q_{n-1}$, with q_i recording the fact that the value of the string read so far is $i \mod n$. Under this interpretation, q_0 will be the initial and (only) final state, and the automaton will transition from state q_i to q_j on reading letter $b \in \{0,1\}$, where $j = (2i + b) \mod n$.

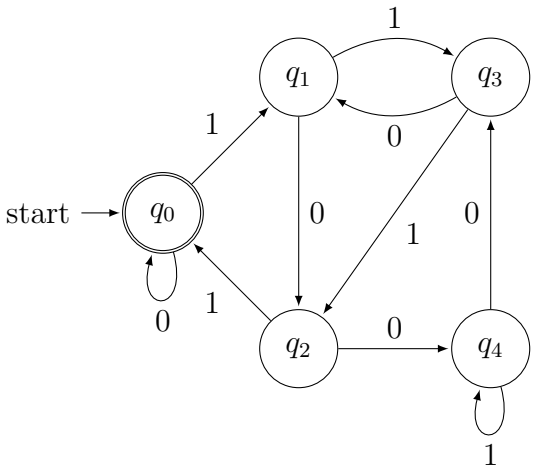

Figure 2: Automaton for strings whose value is divisible by 5.

For example, the automaton for L_5 is given in Figure 2. The reader can construct the automaton for L_4 in a similar manner. Or one could observe that $L_4 = \{0,1\}^*\{00\}$ and construct a simpler automaton.

Finally, the language in question is $L_4 \cup L_5$, for which one can appeal to the closure of regular languages under union (and provide the appropriate NFA). $\dashv$

2. A complete graph on n vertices is an undirected graph in which every pair of distinct vertices is connected by an edge. A simple path in a graph is one in which no vertex is repeated. Let G be a complete graph on 10 vertices. Let u, v, w be three distinct vertices in G. How many simple paths are there from u to v going through w?

Answer:

Let V be the set of all vertices. A path between u to v through w is formed by a subset V' of $V \setminus \{u, v, w\}$ and forming a permutation of $V' \cup \{w\}$. Now for each $i \leq 7$, there are $\binom{7}{i}$ subsets V' of size i, and $(i+1)!$ permutations of $V' \cup \{w\}$.

Thus the number of required paths is

$$8! \cdot \binom{7}{7} + 7! \cdot \binom{7}{6} + 6! \cdot \binom{7}{5} + 5! \cdot \binom{7}{4} + 4! \cdot \binom{7}{3} + 3! \cdot \binom{7}{2} + 2! \cdot \binom{7}{1} + 1! \cdot \binom{7}{0}.$$

This expands to

$$40320 \cdot 1 + 5040 \cdot 7 + 720 \cdot 21 + 120 \cdot 35 + 24 \cdot 35 + 6 \cdot 21 + 2 \cdot 7 + 1 \cdot 1 = 95901.$$

⊣

3. A simple graph is one in which there are no self loops and each pair of distinct vertices is connected by at most one edge. Show that any finite simple graph has at least two vertices with the same degree.

   Answer:

   Let n be the number of vertices in a simple graph. The maximum possible degree is $n-1$. Let us observe that it is not possible for there to simultaneously be two vertices u and v such that u is of degree 0 and v is of degree $n-1$ (since in that case u would have no neighbours, while every vertex other than v – in particular, u – is v's neighbour, which is a contradiction). Therefore there are only $n-1$ possible degrees for the n vertices, which means that two vertices should have the same degree, by the pigeonhole principle. ⊣

4. You are given two sorted lists of integers of size m and n. Describe a divide and conquer algorithm for computing the k^{th} smallest element in the union of the two lists in time $O(\log m + \log n)$.

   Answer: Note that k is treated as a constant in this problem.

   First of all, the question is interesting only if the lists have duplicate elements. Otherwise, we just merge the two lists as usual and after k steps we have the answer.

   Let us first solve a simpler problem and find the k^{th} smallest number in a single sorted list of length n. The first number in the list is clearly the smallest number. Let us call this number x_1. The next number must be at least x_1+1, so we search for x_1+1 in the list using binary search. (Recall that binary search is an example of divide and conquer.) Either we find the number, in which case the second smallest number x_2 is x_1+1, or we fail to find it, in which case we will be at a boundary between the last occurrence of x_1 and the first occurrence of the next smaller number, which is x_2. We now search for x_2+1 and discover x_3. Repeat this $k-1$ times to find x_k. Each binary search takes time $\log n$, so overall this procedure takes time $k \log n$, which is $O(\log n)$ if we treat k as a constant.

   If we have two lists, we can find the first k numbers in the first list, which takes time $O(\log m)$ and then find the first k numbers in the second list, which takes time $O(\log n)$ and then merge these in time $O(k)$ (which is a constant!) to find the k^{th} smallest number overall.

   In fact, we can reduce the number of binary searches to k and avoid the merge step. Maintain indices i_1 and i_2 for the two lists, pointing initially to the first element in lists

1 and 2, respectively. At each stage, the smaller of the numbers pointed to by i_1 and i_2 is the next number to be enumerated. We then advance i_1 or i_2, as the case may be, using binary search as described above. After k such steps, we would have found the k^{th} smallest number overall. Some steps may involve a binary search in list 1, which takes time $O(\log m)$ and others may involve a binary search in list 2, which takes time $O(\log n)$, so each step is bounded by $\max(O(\log m), O(\log n))$. This gives an overall complexity of $\max(O(\log m), O(\log n))$, which is equivalent to $O(\log m + \log n)$.

$\dashv$

5. You are going abroad and you have to complete a number of formalities before you leave. Each task takes a full day to complete. Fortunately, you have an army of friends to help you and each task can be done by either you or any of your friends, so you can complete as many tasks as possible in parallel, on the same day.

   Some tasks depend on others: for instance, you cannot purchase foreign exchange till you have bought your ticket. If task B depends on task A, you can start B only after you complete A. A set of tasks with no pending dependencies can be completed in parallel.

   You are given a list of n such tasks to be completed, where each task comes with a set of other tasks that it depends on. The set of tasks is feasible: there are no circular dependencies. You want to compute the minimum number of days needed to complete all the tasks, given the constraints.

   (i) Model this problem formally using graphs.

   (ii) Describe an efficient algorithm for the problem and analyze the worst-case complexity of your algorithm.

   Answer:

   (i) Construct a graph in which the tasks are the vertices and there is an edge (i, j) if task i must be completed before starting task j. This is a directed graph without cycles, so it is a directed acyclic graph (dag). Any path $i_1 i_2 \ldots i_n$ of tasks would take a minimum of n days to complete because each edge in the path describes a dependency. Hence, the problem is one of finding the longest path in a dag.

   (ii) In any dag, there must be some vertex with no incoming edge (indegree 0). Any vertex of indegree 0 represents a task that has no pending dependencies and can hence be immediately completed. Once it is completed, we can remove it from the graph and operate on the tasks that remain.

   The algorithm is thus the following.

   - Initialize a counter c to 0.
   - While the dag is not empty
     - Remove all vertices of indegree 0
     - Recompute indegrees of remaining vertices
     - Increment c

The final value of c is the answer we seek.

The complexity of this algorithm depends on how we represent the graph. For $G = (V, E)$ let $|V| = n$ and $|E| = m$. If we use an adjacency matrix, for each vertex we remove, we have to scan a row of the matrix to determine which indegrees to decrement, so it will take time $O(n^2)$. If we use adjacency lists, for each vertex we delete, we can scan its list of outgoing edges and directly decrement indegrees for its outgoing neighbours. Across the n vertices we delete, we scan each of the m edges once, so the overall time is $O(n + m)$.

$\dashv$

6. Your final exams are over and you are catching up on watching sports on TV. You have a schedule of interesting matches coming up all over the world during the next week. You hate to start or stop watching a match midway, so your aim is to watch as many complete matches as possible during the week.

   Suppose there are n such matches scheduled during the coming week and you know the starting and finishing time for each match.

   (i) Describe an efficient algorithm to compute the following: for each match, what is the next match whose starting time is strictly later than the finishing time of the current match? Analyze the worse-case complexity of your algorithm.

   (ii) Develop an algorithm based on dynamic programming to compute the maximum number of complete matches you can watch next week. Analyze the worse-case complexity of your algorithm.

   Answer:

   (i) We can accumulate information about the matches in the order in which they appear in the TV schedule. Hence, we can assume that the starting times and ending times of the matches are recorded in two arrays $B[1..n]$ and $E[1..n]$, where $B[i]$ and $E[i]$ are the beginning and ending time, respectively, of match i and B is sorted in ascending order.

   For each match i, we use binary search in B to find the earliest match j such that $E[i] \leq B[j]$ and set $Next[i] = j$.

   We do n binary searches, so this takes time $O(n \log n)$.

   (ii) Let $Best[i]$ denote the maximum number of matches you can watch in the set $\{i, i + 1, \ldots, n\}$. The overall answer we are looking for is $Best[1]$.

   Consider the set of matches $\{i, i + 1, \ldots, n\}$. We have two options initially, and we must pick the better of the two.

   - Watch match i. In this case, we get to watch one match immediately and as many matches as we can manage if we start with the match $Next[i]$, so the overall number of matches is $1 + Best[Next[i]]$.
   - Skip match i. In this case, the number of matches we get to watch is $Best[i + 1]$.

Clearly $Best[n] = 1$ since there is no advantage in skipping the last match.

We can express this as a recurrence for $Best[i]$ as follows, where $Best[n]$ is the base case.

$$\begin{aligned} Best[i] &= \max(1 + Best[Next[i]], Best[i+1]) \\ Best[n] &= 1 \end{aligned}$$

We can now start with the base case $Best[n] = 1$ and work backwards to compute $Best[n-1]$, $Best[n-2]$, ..., $Best[1]$ using dynamic programming, in time $O(n)$.

⊣

7. Consider the code below, defining the function *mystery*:

```
mystery(a,b){
   if (a < 0 or b < 0) return 0;
   else if (a == 0) return b+1;
   else if (b == 0) return mystery(a-1,1);
   else return mystery(a-1, mystery(a,b-1));
}
```

(i) Express $mystery(1, n)$ as a function of n.

(ii) Express $mystery(2, n)$ as a function of n.

(iii) Compute $mystery(3, 2)$ and $mystery(3, 3)$.

Answer:

(i) $mystery(1, n) = n + 2$. We prove this by induction on n.
Clearly $mystery(1, 0) = mystery(0, 1) = 1 + 1 = 2 = 0 + 2$. Assuming that $mystery(1, n-1) = n + 1$, $mystery(1, n) = mystery(0, mystery(1, n-1)) = mystery(0, n+1) = (n+1) + 1 = n + 2$.

(ii) $mystery(2, n) = 2n + 3$, and we again prove this by induction on n.
Clearly $mystery(2, 0) = mystery(1, 1) = 1 + 2 = 3 = 2 \cdot 0 + 3$. Assuming that $mystery(2, n-1) = 2(n-1)+3 = 2n+1$, $mystery(2, n) = mystery(1, mystery(2, n-1)) = mystery(1, 2n+1) = (2n+1) + 2 = 2n + 3$.

(iii) $mystery(3, 0) = mystery(2, 1) = 2 + 3 = 5$.
$mystery(3, 1) = mystery(2, mystery(3, 0)) = mystery(2, 5) = 2 \cdot 5 + 3 = 13$.
$mystery(3, 2) = mystery(2, mystery(3, 1)) = mystery(2, 13) = 2 \cdot 13 + 3 = 29$.
$mystery(3, 3) = mystery(2, mystery(3, 2)) = mystery(2, 29) = 2 \cdot 29 + 3 = 61$.

⊣