Written Qualifying Exam
Theory of Computation

Spring, 1999

Friday, May 21, 1999

This is nominally a *three hour* examination, however you will be allowed up to four hours. There are six questions; answer all six questions. All questions carry the same weight.

• Please print your name on the back of your exam envelope next to your exam identification number. Do NOT write your name on the exam booklets.
• Use a separate booklet for each question. The exam booklets have been labelled by question number; pelase be sure to use the correct booklet for the question you are answering.

Read the questions carefully. Keep your answers brief. Assume standard results, except where asked to prove them.

Problem 1 [10 points]
An n-permutation is a reordering of the integers $1, \cdots, n$, which we will store in the array $P[1 : n]$ as follows: $P[i]$ denotes the number mapped to the ith position, $1 \leq i \leq n$. Thus the array $P[1 : n]$ should contain each integer $1, \cdots, n$ exactly once.

Consider the following algorithm for generating a random permutation.

```
initialize: $P[i] \leftarrow i$ for $= i, \cdots, n$
for $i = n$ downto 1 do
    generate an integer $j$ in the range $[1, i]$ uniformly at random;
    swap $(P[i], P[j])$
end
```

Show that this algorithm generates each possible permutation with probability $1/n!$.

Problem 2 [10 points]
The knapsack problem is the following. Given a knapsack of size S, and a collection of n items of sizes $s_1, s_2, \ldots, s_n$, respectively, determine if there is a subset $i_1, i_2, \ldots, i_k$ of the items such that

$$\sum_{h=1}^{k} s_{i_k} = S$$

assuming S and s_i, $i \geq 1$ are positive integers.

In general, the knapsack problem is NP-complete. However, suppose S is bounded by n^2; give a polynomial time algorithm for the knapsack problem in this case, and in the event that there is a solution, your algorithm should determine the subset $i_1, i_2, \ldots, i_k$. Your answer, in addition to describing an algorithm, should explain why the algorithm computes the desired result, and should give a brief analysis of its running time.

Problem 3 [10 points]
Let G be a directed graph in which each edge is colored, either red or blue. A red-blue path is a path in which the edges alternate in color, starting with a red edge and ending with a blue edge. Vertices u and v are said to be *red-blue strongly connected* if there are red-blue paths from u to v and from v to u.

a. Show that red-blue strongly connected is an equivalence relation.

b. Let the equivalence classes of the red-blue strongly connected relation be called red-blue strong components. By means of a reduction to strong connectivity, or otherwise, give a linear time algorithm to compute the red-blue strong components. You may assume a linear time strong components algorithm is given. Hint: For the reduction, making two copies of each vertex may be helpful.

Go to the next page

Problem 4 [10 points]
Consider the following machine: a 2-way p.d.a. It is similar to a p.d.a.; there are two differences. First, on each move, it can move its input head either left or right on its input, the only restriction being that it remain between end markers (# and \$) denoting the start and finish of its input. Second, it accepts by entering an accept state.

a. Describe a deterministic 2-way p.d.a. that accepts the language

$$L_1 = \{xx \mid x \in \{a, b\}^*\}$$

Do not give state transitions; simply explain what your machine does (e.g. "copy the input to the stack...").

b. Describe a deterministic 2-way p.d.a. that accepts the language

$$L_2 = \{a^x b^y c^{x \cdot y} \mid x, y > 0\}$$

Problem 5 [10 points]
Let $f(n)$ be a computable function. The xth Turing Machine clocked with respect to f, $M_x^{f(n)}$ is defined as follows: on input y, $|y| = n$, it first computes $f(n)$ and then simulates $M_x(y)$ for $f(n)$ steps. If $M_x(y)$ halts, $M_x^{f(n)}(y)$ accepts and otherwise it rejects (but it always halts).

a. Show that $L_2 = \{x \mid \exists y\ M_x^{n^2}(y) \text{ rejects}\}$ is r.e.

b. Show that $L_1 = \{x \mid \forall y\ M_x^{n^2}(y) \text{ accepts}\}$ is not r.e.

Problem 6 [10 points]

a. Let F be a 3-CNF formula. Function *count*(F) returns the number of different satisfying assignments for F. Suppose that *count* can be computed in polynomial time. Then show how to find a satisfying assignment in polynomial time, if there is one, for 3-CNF formula F.

b. Let F_1, F_2 be 3-CNF formulas. Function *Equal_count*(F_1, F_2) returns TRUE if the number of satisfying assignments for F_1 and F_2 are equal and FALSE otherwise. Suppose that *Equal_count* runs in polynomial time. Then show how to find a satisfying assignment in polynomial time, if there is one, for 3-CNF formula F. Hint: Let F^1 be F with variable x_1 set to TRUE and F^0 be F with x_1 set to FALSE (assuming F has variables $x_1, x_2, \cdots, x_n$). If F is not satisfiable, what is the value of *Equal_count*(F, F^1) and *Equal_count*(F, F^0)? What about if F is satisfiable?

Solutions

Solution to Problem 1

Consider a particular permutation π. It is obtained by first swapping the correct item into $P[n]$ and then correctly permuting the $n-1$ items now in $P[1:n-1]$. Inductively, the latter event occurs with probability $1/(n-1)!$ and the former event with probability $1/n$, given an overall probability of $1/n!$. For completeness, we should note that in the base case, $n = 1$, the one permutation occurs with probability $1 = 1/1!$.

Solution to Problem 2

Consider the following recursive algorithm for the knapsack problem.

```
**Procedure** Knapsack $(s, n, solution, Soln\_Part)$
(*$solution$ is a boolean variable indicating if there is a solution*)
(*$Soln\_Part$ is an array used for reconstructing a solution*)
  **if** $n = 1$ **then**
    **if** $S = s_1$ **then**
      $solution \leftarrow$ TRUE
      $Soln\_Part(S, 1) \leftarrow 1$
    **else**
      $solution \leftarrow$ FALSE
      $Soln\_Part(S, 1) \leftarrow 0$
  **else**
    Knapsack $(S, n-1, solution, Soln\_Part)$
    **if** $solution =$ TRUE **then**
      $Soln\_Part(S, n) \leftarrow Soln\_Part(S, n-1)$
    **else if** $S \geq s_n$ **then**
      Knapsack $(S - s_n, n-1, solution, Soln\_Part)$
      **if** $solution =$ TRUE **then**
        $Soln\_Part(S, n) \leftarrow n$
**end**
```

We employ dynamic programming to avoid repeated recursive calls. As $S \leq n^2$, there are $O(n^3)$ recursive calls, each of which takes $O(1)$ time to evaluate, giving an $O(n^3)$ running time overall. The above recursive algorithm tries the two options of including and not including the item of size s_n in the knapsack, and thus tries every distinct possibility.

To obtain the items forming a solution if one exists the following recursive procedure is used.

```
**Procedure** Print_Solution $(S, n, Soln\_Part)$
  $next\_item \leftarrow Soln\_Part(S, n)$
  **if** $next\_item > 0$ **then**
    Print $(next\_item)$
    Print $(S - s_{next\_item}, next\_item - 1)$
**end**
```

For the array entry $Soln\_Part(S,n)$ stores the highest index item in the solution to the (S,n) knapsack problem computed by our algorithm, if there is one. Clearly this procedure runs in $O(n)$ time.

Solution to Problem 3

a. We write $u \sim v$ if u and v are red-blue strongly connected. To show $\sim$ is an equivalence relation we note:

(i) $u \sim u$ (by using zero length paths from u to u).

(ii) $u \sim v$ if and only if $v \sim u$ (as the relationship is symmetric by inspection).

(iii) $u \sim v$ and $v \sim w$ implies $u \sim w$ (this follows by concatenating the pair of red-blue paths from u to v and from v to w and the pair from w to v and from v to u yielding red-blue paths from u to w and from w to u, respectively).

b. We build a directed graph $G' = (V_1 \cup V_2, E')$, where for each vertex v in G there are vertices $v^1 \in V_1$ and $v^2 \in V_2$. A blue edge (u,v) in E produces edge (u^1, v^2) in E', and a red edge (w,x) in E produces edge (w^2, x^1) in E'.

Let C be the vertices in a strong component of G^1. Suppose $C \cap V_1$ comprises the vertices $v^1_{i_1}, v^1_{i_2}, \cdots, v^1_{i_k}$. Then $v_{i_1}, v_{i_2}, \cdots, v_{i_k}$ form a red-blue component of G and conversely. For u^1 and v^1 are in the same strong component of G^1 if only if there are red-blue paths between u and v in G, i.e. u and v are in the same red-blue strong component.

G^1 is readily constructed in linear time; together with a linear strong components algorithm and the trivial linear time mapping of strong components of G^1 to red-blue strong components of G, this yields a linear time algorithm for finding red-blue strong components.

Solution to Problem 4

a.

Step 1. Push $|x|$ onto the stack.

Simply read across the input and for every two symbols read push one onto the stack (if there are an odd number of symbols in the input, reject).

Step 2. Move the read-write head distance $|x|$ from the left end of the input.

Use the stack contents to count distance $|x|$ from the left end of the input: move the read-write head to the left end of the input; then repeatedly pop the stack and move the read-write head one position to the right, until the stack is empty.

Step 3. Push a copy of the second x onto the stack.

Simply copy the right half of the input onto the stack.

Step 4. Verify the left half of the input is also x.

As in Step 1 and 2, move the read-write head to the middle of the stack (instead of the stack bottom, in Step 2, use a marker such as #). Now, after popping the marker, repeatedly pop the stack (which holds x^R) and match with the first x on the input, being read from the right and to the left.

b. In turn, push $x, 2x, \cdots, x \cdot y$ a's onto the stack, on top of $y-1, y-2, \cdots, 0$ b's, respectively.

Initially, y b's are placed on the stack.

The general iteration proceeds as follows:

> Move the read-write head to the rightmost b. Use the z a's on the stack to move the read-write head distance z to the right by popping all the a's (if the read-write head cannot move that far, then reject).
>
> Pop one b.
>
> Push z a's onto the stack by moving the read-write head back to the rightmost b.
>
> Push x a's onto the stack (copy the string a^x).

The iteration in which the string of b's on the stack is emptied will end with $x \cdot y$ a's on the stack. Now check that this is the length of the string of c's.

Solution to Problem 5

a. Consider the following T.M. M which as we will show accepts the language L_2. $M(x)$ in turn simulates $M_x^{n^2}(y)$ for $y = 0, 1, 2, \cdots$ until a value of y is found for which $M_x^{n^2}(y)$ rejects, in which case $M(x)$ accepts (otherwise $M(x)$ does not halt). Clearly $M(x)$ halts exactly if $x \in L_2$, and so L_2 is r.e.

b. We give a reduction of $\bar{K} \leq L_1$ or equivalently of $K \leq L_2$. As $\bar{K}$ is not r.e. this shows L_1 is not r.e. also.

The reduction is carried out by the computable function f defined as follows.

$$\begin{aligned} M_{f(x)}(y) \quad = \quad & \text{simulate } M_x(x) \text{ for } y \text{ simulation time;} \\ & \text{if it does not accept, then accept and otherwise reject} \end{aligned}$$

Clearly $M_{f(x)}$ runs in linear time, so $M_{f(x)}^{n^2}$ computes identically. If $x \in \bar{K}$ then $M_{f(x)}^{n^2}(y)$ accepts on all inputs y, and otherwise it rejects for large enough y (those y's that permit $M_x(x)$ to be simulated to acceptance). Thus $\bar{K} \leq L_1$ as claimed.

Solution to Problem 6

Suppose we have a polynomial time computable function $Satis(F)$ that returns TRUE if F is satisfiable and FALSE otherwise. Then, if F is satisfiable, a satisfying assignment can be found in polynomial time as follows.

Let $x_1, x_2, \cdots, x_n$ be the variables in F and let $F(x_n = \textsc{True})$ and $F(x_n = \textsc{False})$ denote F with x_n set to True and False respectively.

A satisfying assignment is found recursively as follows.

```
Procedure $Sat\_Assign(F)$
    if $n = 0$ then return
    if $Satis(F(x_n = \textsc{True}))$
        then return$(Sat\_Assign(F(x_n = \textsc{True})) \cup (x_n = \textsc{True}))$
        else return $(Sat\_Assign(F(x_n = \textsc{False})) \cup (x_n = \textsc{False}))$
```

a. $Satis(F)$ is implemented as follows: $Satis(F) = F$ if F has no variables; otherwise, $Satis(F) = (Count(F) > 0)$.

b. Note that $count(F) = count(F^0) + count(F^1)$ (if F has at least one variable). If F is not satisfiable then $count(F) = count(F^0) = count(F^1) = 0$ and so $Equal\_count(F, F^0) = \textsc{True} = Equal\_count(F, F^1)$. Otherwise, as at least one of $count(F^0)$ and $count(F^1)$ is greater than zero, and they cannot both be equal to $count(F)$, either $Equal\_count(F, F^0) = \textsc{False}$ or $Equal\_count(F, F^1) = \textsc{False}$ (or possibly both).

Thus $Satis(F)$ is implemented as follows: $Satis(F) = F$ if F has no variable; otherwise, $Satis(F) = \textbf{not}(Equal\_count(F, F^0)$ **and** $Equal\_count(F, F^1))$.