Written Qualifying Exam
Theory of Computation

Spring, 1998

Friday, May 22, 1998

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

- Please write your name on the outside envelope, but not on any if the exam booklets.
- Please answer each question in the numbered booklet provided for that question.

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

Problem 1 [10 points]
Consider the problem of sorting an array $A[1:n]$ of n distinct items, where each item is guaranteed to be within k places of its correct location in the sorted array; i.e. $A[h]$ belongs somewhere between $A[h-k]$ and $A[h+k]$ in the sorted ordering.

Consider the following algorithm for sorting A. It uses a heap H which can hold up to $k+1$ items.

```
procedure Sort_PartiallySorted(A, n)
  for i ← 1 to k + 1 do
    HeapInsert(H, A[i]) {* inserts A[i] into heap H *}
  endfor
  for i ← k + 2 to n do
    A[i − (k + 1)] ← Deleletemin(H)
    HeapInsert(H, A[i])
  endfor
  for i ← 1 to k + 1 do
    A[n − (k + 1) + i] ← Deletemin(H)
  endfor
end_Sort_PartiallySorted.
```

a. **3 points.** Argue that the above algorithm correctly sorts A if every item starts within k positions of its final location.

b. **2 points.** What is the running time of the above algorithm as a function of n and k? Justify your answer briefly.

c. **2 points.** Suppose the heap is stored in-place in $A[1:k+1]$. By slightly modifying the above algorithm, explain how to reorder the array so that $A[k+2] < A[k+3] < \cdots < A[n] < A[1] < A[2] < \cdots < A[k+1]$. It suffices to explain the changes in words.

d. **3 points.** Suppose $k+1$ divides n exactly. Give an $O(n)$ time algorithm to reorder the array from part (c) so that it is in standard sorted order ($A[1] < A[2] < \cdots < A[n]$). Your algorithm may only use $O(1)$ space in addition to the array A. Further, do not assume that k is a constant (so an $O(nk)$ time algorithm does not suffice).

Problem 2 [10 points]
Consider the following medical rationing problem.

There are k diseases. Each disease has a vaccine. The cost of the ith vaccine is \$$c_i$. The ith vaccine has an effectiveness e_i, versus an effectiveness f_i if the ith vaccine is not given. The effectiveness is the fraction of people that survive or avoid the disease in question. You may assume $e_i > f_i$ (for otherwise the vaccine is worthless).

Suppose \$$D$ can be spent per person on vaccines. Assume D and c_i, $1 \leq i \leq k$, are integers. Give an algorithm to determine a best choice of vaccines, i.e. a choice that achieves the highest survival rate. More precisely, suppose vaccines $j_1, \cdots, j_l$ are chosen, and vaccines $h_1, \cdots, h_{k-l}$ are not chosen. The goal is to maximize:

$$\prod_{i=1}^{l} e_{j_i} \cdot \prod_{i=1}^{k-l} f_{h_i} \text{ given that } \sum_{i=1}^{l} c_{j_i} \leq D$$

Your algorithm should run in time $O(kD)$.

Hint. Use Dynamic Programming.

Problem 3 [10 points]
The Gas Tank Problem.

Suppose a directed graph $G = (V, E)$ is given in which each edge is labelled with a real number cost (in gallons). Let $n = |V|$.

In the following problem you may use the $O(n^3)$ Floyd-Warshall all pairs shortest path algorithm for G without further elaboration.

a. **5 points.** Suppose that some subset $U \subseteq V$ of nodes are labelled as gas stations. Suppose that a car has a gas tank with capacity g gallons, and initially it is full. The problem is to determine, for each pair i, j of vertices in G, whether it is possible for the car to travel from vertex i to vertex j with at most one refuelling, and if so, to determine the most gas that can remain in the tank. Show how to solve this problem in $O(n^3)$ time.

b. **5 points.** Suppose any number of refuellings are allowed. Now, for each pair i, j of vertices, give an algorithm to determine if the car can travel from i to j assuming it starts with a full tank of gas, and if so determine the largest amount of gas that could remain in the gas tank. Again, seek an algorithm with an $O(n^3)$ running time.

Hint. A trip from i to j had three parts:
a. The journey from i to a first gas station.
b. The journey from the first to the last gas station, possibly via intermediate gas stations.
c. The journey from the last gas station to j.

What is the "cost" of each of the parts?

Go to the next page

Problem 4 [10 points]

Let Σ be an alphabet of two or more characters. Let $L \subseteq \Sigma^*$. Strings $x, y \in \Sigma^*$ are *strongly equivalent* with respect to L if for all $w, z \in \Sigma^*$:

$$wxz \in L \iff wyz \in L$$

Let $C_x = \{y \mid x \text{ and } y \text{ are strongly equivalent}\}$.

It is easy to see that C_x is an equivalence class (you need not prove this). C_x is called x's class (w.r.t. L).

Show that if L is regular then there are finitely many classes of strongly equivalent strings with respect to L.

Hint. Consider a DFA M accepting L. Let M have state set Q. Consider strings x and y, and pairs of states $\delta(q, x)$ and $\delta(q, y)$, for states $q \in Q$, where δ is the transition function for M.

Problem 5 [10 points]

A **twin prime** is a pair of primes of the form $(p, p+2)$. Thus $(3,5), (5,7), (11,13)$ are the first three twin primes. Let $\langle M \rangle$ denote the standard encoding of Turing machine M. Consider the language B comprising all $\langle M \rangle$ such that for all twin primes $(p, p+2)$, M accepts p and also accepts $p+2$.

Classify the language B completely with respect to its recursiveness, recursive enumerability (r.e.), and co-recursive enumerability (co-r.e.); i.e., is B recursive, r.e., co-r.e., or none of these. You must justify your answers. NOTE: it is not known if there are infinitely many twin primes. You should consider both logical possibilities.

Go to the next page

Problem 6 [10 points]
In this question, assume probabilistic Turing machines (PTM) that halt on every path, and answer 'YES' or 'NO' upon halting. (In general, a PTM could also answer 'MAYBE'.) Let $e(n)$ be a function such that $(\forall\ n)\ 0 < e(n) < 1/2$. M has **error bound** $e(n)$ if:

- On $w \in L(M)$, the probability that M answers YES is $\geq 1 - e(|w|)$.
- On $w \notin L(M)$, the probability that M answers NO is $\geq 1 - e(|w|)$.

A $p(n)$**-strong** BPP**-machine** is a PTM that runs in polynomial time with error bound $e(n) = 1/p(n)$. A RP**-machine** is a PTM that runs in polynomial time, and for any inputs not in the language, the machine answers NO on every path.

Suppose SAT is accepted by a $p(n)$-strong BPP-machine M, for a sufficiently large polynomial $p(n)$. Consider the following procedure to test if a given Boolean formula F is satisfiable: let the Boolean variables in F be $x_1, \ldots, x_n$. We shall operate in n stages. At the start of stage k ($k = 1, \ldots, n$), we have already computed a sequence of Boolean values $b_1, \ldots, b_{k-1}$, and $F_{b_1 \cdots b_{k-1}}$ is the formula in which x_i is replaced by b_i ($i = 1, \ldots, k-1$).

STAGE k:
1. Call M on input $F_{b_1 \cdots b_{k-1} 0}$.
2. If M answers YES, then set $b_k = 0$ and go to DONE.
3. Else call M on input $F_{b_1 \cdots b_{k-1} 1}$.
4. If M answers NO again, answer NO and return.
5. Else set $b_k = 1$.
6. DONE: If $k < n$ go to stage $k + 1$.
7. Else answer YES if $F_{b_1, \ldots, b_n} = 1$, otherwise answer NO.

Prove that this procedure is an RP-machine for SAT, if $p(n)$ is a sufficiently large polynomial. Assume $|F_{b_1, \ldots, b_k}| = |F| \geq n$, $0 \leq k \leq n$. You will need to choose an appropriate p.

HINT: If $F_{b_1 \cdots b_{k-1}}$ is satisfiable, what is the probability of the following event: either the algorithm answers NO in stage k or the $F_{b_1 \cdots b_k}$ computed in stage k is not satisfiable.

Solutions

Solution to Problem 1

a. The smallest item in the array must lie among the first $k+1$ items in A and hence is correctly identified and written in $A[1]$. Suppose the first i items are correctly placed by the algorithm. Then the $(i+1)$st item must be drawn from the remaining k items in the heap (the remaining k items from $A[1] \cdots A[i+k]$) and $A[i+k+1]$. But these are the items in the heap following the insert of the $(i+1)$st step and thus the algorithm correctly identifies the $(i+1)$st item and places it in $A[i+1]$.

b. Each heap operation requires $O(\log(k+1))$ time (assuming $k \geq 1$). Thus the algorithm runs in $O(n \log k)$ time for $k \geq 2$, and $O(n)$ time for $k = 0, 1$.

c. Instead of outputting the sorted items to $A[1], A[2], \cdots, A[n]$ in turn, they are output to $A[k+2], A[k+3], \cdots, A[n], A[1], \cdots, A[k+1]$ in turn. Care must be taken to store $A[k+i+1]$ on the ith iteration, before it is overwritten by the ith smallest item. Further, the heap is stored "backward" with the minimum in $A[k+1]$, so that in the final stage as the heap shrinks in size, items can be written in $A[1]$, $A[2]$, $\cdots$, $A[k+1]$, in turn.

d. We repeatedly move blocks of $k+1$ items to their final locations, starting with the smallest $k+1$ items, followed by the next smallest $k+1$ items, followed by the next and third smallest set of $k+1$ items, and so on. In turn, each set of $k+1$ items is swapped with the block of the $k+1$ largest items, which are initially in the leftmost $k+1$ locations. One could think of this as a bubble sort, with a bubble of the $k+1$ largest items moving to the right, in steps of size $k+1$. Each step results in the next smallest $k+1$ items being correctly positioned.

The code is given below. Clearly, the algorithm takes $O(n/(k+1) \cdot (k+1)) = O(n)$ time.

```
procedure Reorder(A, n, k)
  for i ← 1 to n/(k+1) do
    for j ← 1 to k+1 do
      swap(A[(i−1)*(k+1)+j], A[i*(k+1)+j])
    endfor
  endfor
end_Reorder.
```

Solution to Problem 2

Let Effect(R, i) be a function that computes the effectiveness of a most effective choice of vaccines among the first i vaccines, with cost at most R.

Then, Effect(D, k) is defined recursively as follows:

```
procedure Effect(D, k))
  if k = 0 then return 1
  elseif D < c_k then return Effect(D, k − 1) · f_k
  else do
      use_k ← Effect(D − c_k, k − 1) · e_k
      not_use_k ← Effect(D, k − 1) · f_k
      if use_k ≥ not_use_k then return use_k
      else return not_use_k
      endif
    endif
  endif
end_Effect.
```

By using a table T of Dk entries, this recursive algorithm becomes a Dynamic Programming algorithm taking $O(1)$ time per recursive call and hence $O(Dk)$ time overall.

To determine the choice of vaccines, with each table entry, $T(R, i)$, the corresponding choice of vaccine needs to be recorded in a second table $V(R, i)$ (i.e., whether the ith vaccine is used or not). Then, by a standard backtracking, the best overall choice of vaccines can be determined in a further $O(k)$ time. The code follows.

```
forall R, i, 0 ≤ R ≤ k, O ≤ i ≤ k, initialize T(R, i) ← ∞
procedure Effect(D, k)
  if T(D, k) ≠ ∞ then return T(D, k)
  elseif k = 0 then answer ← 1
  elseif D < c_k then answer ← Effect(D, k − 1)·f_k
  else do
      use_k ← Effect(D − c_k, k − 1) · e_k
      not_use_k ← Effect(D, k − 1)·f_k
      if use_k ≥ not_use_k then V(D, k) ← 'use'; answer ← use_k
      else V(D, k) ← 'not use'; answer ← not_use_k
      endif
      T(D, k) ← answer
      return answer
    endif
  endif
end_Effect.
```

```
procedure ChooseVaccines(D, k)
  if k ≥ 1 then
    if T(D, k) = 'use' then Print(Use Vaccine k); ChooseVaccines(D − c_k, k − 1)
    else ChooseVaccines(D, k − 1)
    endif
  endif
end_ChooseVaccines.
```

Solution to Problem 3

a. First, the all pairs shortest path problem is solved on graph G. Suppose the solution for vertex pair (i, j) is stored in ShortestDirect(i, j). Then ShortestNoStop is computed as follows:

```
procedure ShortestNoStop(i, j)
  if ShortestDirect(i, j) ≤ g
    then ShortestNoStop(i, j) ← ShortestDirect(i, j)
  else ShortestNoStop(i, j) ← ∞
  endif
end_ShortestNoStop.
```

This gives the least amount of gas $\leq g$ needed to travel from i to j, and is ∞ if there is no route using at most g gallons.

To determine the amount left in the tank if up to one refuelling is allowed, all paths involving one stop at a gas station are considered, thus:

```
procedure ShortestOneStop(i, j)
  if ShortestDirect(i, j) ≤ g
    then ShortestOneStop(i, j) ← ShortestDirect(i, j)
  else ShortestOneStop(i, j) ← ∞
  endif
  for each u ∈ U do
    if ShortestDirect(i, u) ≤ g
      then ShortestOneStop(i, j) ←
                  min{ShortestDirect(u, j), ShortestOneStop(i, j)}
    endif
  endfor
end_ShortestOneStop.
```

Finally, we compute GasRemaining(i, j) to be the difference of g and ShortestOneStop(i, j), unless ShortestOneStop(i, j) is ∞, in which case there is no route from i to j with just one refuelling.

Since $|U| \leq n$, this procedure requires $O(n^3)$ time over all vertex pairs i, j. Clearly, it considers all paths involving at most one refuelling.

b. We create a new graph G' which augments G. The following new edges with length 0 are added to G: edge (i, u) for each $u \in U$ such that ShortestNoStop$(i, u) \leq g$.

If there are duplicate edges, only the 0-weight edge is kept.

The Floyd-Warshall algorithm is run on G'. As G' has n vertices this takes $O(n^3)$ time.

Clearly, a path from i to gas station u that uses at most g gallons will leave the tank full after refuelling at u. Likewise, paths between gas stations of length at most g, will also leave the gas tank full, after subsequent refuellings. Thus, the cost, in fuel, of a path from i to j, which uses the new 0-cost edges, is the cost in fuel of travelling from the last gas station to j, where all the paths between successive gas stations use at most g gallons, as does the path from i to the first gas station. But this is what the algorithm computes. As in part (a), the amount of gas left in the tank is the difference between g and the length of the shortest path (except where there is no shortest path, which indicates that there is no route that can be managed with a gas tank holding only g gallons).

Solution to Problem 4

Let M be a dfa accepting L. Let Q be the set of states for M and let δ be the transition function for M. Suppose that $\delta(q,x) = \delta(q,y)$ for all $q \in Q$. Then, for all strings w, z, $\delta(q_1, wxz) = \delta(q_1, wyz)$, where q_1 is the initial state of M, and thus $wxz \in L$ if and only if $wyz \in L$. In other words, x and y are strongly equivalent. But this is a finite partitioning: there are only $|Q|^{|Q|}$ collections $(q_1, q_{i_1}), (q_2, q_{i_2}), \cdots, (q_{|Q|}, q_{i_{|Q|}}))$, where $1 \le q_{i_j} \le |Q|$, for $1 \le j \le |Q|$; with each collection we associate the set of strongly equivalent strings such that $\delta(q_j, x) = q_{i_j}$, for $1 \le j \le |Q|$. As each string must belong to one of these collections, we conclude that a regular language L has only finitely many strongly equivalent sets.

Solution to Problem 5

It is convenient to write

$$\pi_1, \pi_2, \pi_3, \ldots, \tag{1}$$

for the sequence of twin primes. Thus $\pi_1 = (3,5)$, $\pi_2 = (5,7)$, etc. We may use the fact that the function $i \mapsto \pi_i$ is computable. Consider the two logical possibilities.

(1) **There are finitely many twin primes.** Then B is r.e., but not recursive. To see that it is not recursive, we can invoke Rice's theorem. To see that it is r.e., we can construct a TM M_B that, on input $\langle M \rangle$, simply checks if M accepts each prime in the sequence (1).

(2) **There are infinitely many twin primes.** Then B is neither r.e. nor co-r.e.

(2.1) To see that B is not r.e., we give a many-one reduction of co-A_{TM} to B. [Note: the set A_{TM} comprises all pairs $\langle M, w \rangle$ such that M is a TM that accepts w.] Given $\langle M, w \rangle$, we construct a TM N with the following property: on input x, N will run M on w for $|x|$ steps. If M accepts within $|x|$ steps then N rejects. Otherwise N accepts. Thus N has this property:

– If M rejects w, then N accepts all inputs (and so all twin primes).

– If M accepts w, then N rejects all inputs after some point.

Equivalently, $\langle M, w \rangle \notin A_{TM}$ iff $\langle N \rangle \in B$. If B is r.e., then co-A_{TM} is r.e., a contradiction.

(2.2) Suppose B is co-r.e. We derive the contradiction that co-A_{TM} is r.e. by using

another reduction: on input $\langle M, w\rangle$, we construct a TM N with the following property: on input x, N will accept unless $x = 3$. If $x = 3$, N will simulate M on w (accepting iff M accepts). Thus N has this property:
– M rejects w iff N does not accept all primes. Equivalently, $\langle M, w\rangle \notin A_{TM}$ iff $\langle N\rangle \notin B$. Thus if B is co-r.e., then co-A_{TM} is r.e., a contradiction.

Solution to Problem 6

To show the procedure is an RP-algorithm, we need to show 3 properties: (a) the procedure is polynomial time, (b) if F is unsatisfiable, the answer is always NO, and (c) the probability of accepting a satisfiable formula is $> 1/2$.

Property (a) is obvious. To see property (b), note that the answer YES occurs only at the end of stage n, and this answer is never wrong. This implies that when F is unsatisfiable, the answer is NO on every path.

Finally, to see property (c), assume F is satisfiable. Write F_k for $F_{b_1,\ldots,b_k}$, assuming that $b_1, \ldots, b_k$ are defined. Let the event A_k correspond to "no mistakes up to stage k", i.e., F_k is defined and satisfiable. Similarly, let event E_k correspond to "first mistake at stage k", i.e., $E_k = A_{k-1} \cap \overline{A_k}$.

CLAIM: $\Pr(E_k) \leq 2^{-|F|+1}$.

Proof: Note that $\Pr(E_k) \leq \Pr(E_k|A_{k-1})$. We will bound $\Pr(E_k|A_{k-1})$. Assuming A_{k-1}, we consider 2 cases:

(A) CASE $F_{b_1 \cdots b_{k-1}0}$ is not satisfiable. Then $F_{b_1 \cdots b_{k-1}1}$ is satisfiable. With probability $\geq (1 - 1/p(n))$, the procedure will (correctly) answer NO the first time we invoke M. Then with probability $\geq (1 - 1/p(n))$, it will (correctly) answer YES the second time. So $\Pr(A_k|A_{k-1}) \geq (1 - 1/p(n))^2$ and

$$\Pr(E_k|A_{k-1}) \leq 1 - (1 - 1/p(n))^2 \leq 2/p(n).$$

(B) CASE $F_{b_1 \cdots b_{k-1}0}$ is satisfiable. This case is even easier, and yields $\Pr(E_k|A_{k-1}) \leq 1/p(n)$. This proves the claim.

To conclude, the probability of making a mistake at any stage is at most

$$\sum_{k=1}^{n} \Pr(E_k) \leq n \cdot 2/p(n) = 2n/p(n).$$

This is less than 1/2 if $p(n) \geq 4n$. Hence F will be accepted if $p(n) \geq 4n$.