Honors Algorithms/Written Qualifying Exam

Wednesday, Dec. 22, 2010

This is a $3\frac{1}{2}$ hour examination.
All questions carry the same weight.
Answer all six questions.

- Please *print* your name on the sticky note attached to the outside envelope, and nowhere else.
- Please *do* not write your name on the examination booklets.
- Please answer each question in a separate booklet, and *number* each booklet with that question number.
- Read the questions carefully. Keep your answers brief. Assume standard results, except when asked to prove them.
- When you have completed the exam, please reinsert the booklets back into the envelope.

Problem 1 **[10 points]** *New numbered exam booklet PLEASE*
Consider the following randomized procedure:

```
    function $Rand(n)$;
1       if $n \le 1$ then return(1)
2       else set $x$ to 0 with probability $\frac{1}{3}$, or 1 with probability $\frac{2}{3}$;
3           if $x = 0$ then return($3 \cdot Rand(\frac{n}{2}) + Rand(\frac{n}{2}) + 4 \cdot n \cdot n \cdot n$)
4           elseif $x = 1$ then return($2 \cdot Rand(\frac{n}{2}) + 2 \cdot Rand(\frac{n}{2}) + 7 \cdot n \cdot n \cdot n$)
5           endif
6       endif
    end_$Rand$;
```

(a) Let $R(n)$ be the expected value returned by the $Rand$ procedure on input n. Write an *exact* recurrence equation for $R(n)$ and solve it in the Θ-notation (do not bother about the base case).

Solution

$R(n) = 4R(n/2) + 6n^3$, $n > 1$; so $R(n) = \Theta(n^3)$.

(b) Let $M(n)$ be the expected number of multiplications needed to compute $Rand(n)$ (all multiplications use "$\cdot$" symbol in the code above). Write a recurrence equation for $M(n)$ and solve it in the Θ-notation (do not bother about the base case).

Solution

$M(n) = 2M(n/2) + \Theta(1)$, $n > 1$; so $M(n) = \Theta(n)$.

(c) Consider the following faster procedure $Fast(n)$. Notice that the expected value computed by $Fast(n)$ is the same as the expected value computed by $Rand(n)$ (which you also computed in part (a)). Let $F(n)$ be the expected number of multiplications used by $Fast(n)$. Write a recurrence equation for $F(n)$ and solve it in the Θ-notation (do not bother about the base case). Compare with your answer in part (b).

```
    function $Fast(n)$;
1       if $n \le 1$ then return(1)
2       else set $x$ to 0 with probability $\frac{1}{3}$, or 1 with probability $\frac{2}{3}$
3           if $x = 0$ then return($4 \cdot (Fast(n/2) + n \cdot n \cdot n)$)
4           elseif $x = 1$ then return($4 \cdot Fast(n/2) + 7 \cdot n \cdot n \cdot n$)
5           endif
6       endif
    end_$Fast$;
```

Solution

$F(n) = F(n/2) + O(1)$, $n > 1$; so $F(n) = \Theta(\log n)$, which is much faster than the time used in part (b). However, the random variable computed in (c) has a different distribution from that in part (b), although the expectations are the same.

Problem 2 **[10 points]** *New numbered exam booklet PLEASE*

Formal Languages

(a) Let $L = \{(a|b)^i(b|c)^j : i < j\}$. Prove that L is not regular.

Solution

Use the uvy pumping lemma. A convenient version says: for any RL L, there is a fixed p such that for any string ω in L with $|\omega| \ge p$, we can write: $\omega = uvy$ where $|uv| < p$, $|v| > 0$, and uv^iy is in L for all $i \ge 0$.

So choose $a^p b^{p+1}$, which is in L. v must be all as. Pump that v (or let $i = 2$). We get a string with more a's than bs and no cs, which is not in L. It follows that there is no such p for L, and L is therefore not a RL.

(b) Prove that L is a CFL.

Solution

A Grammar is as easy as anything for this: $S \to BC$; $B \to LBR|\epsilon$; $L \to a|b$; $R \to b|c$.
So the B derivations contribute equally to i and j. As for C, we have: $C \to C|b|c$, which will increase j by an arbitrary positive integer.

(c) For any two languages L and M, let

$\mathit{Shuffle}(L, M) = \{\ell_1\mu_1\ell_2\mu_2 \cdots \ell_k\mu_k$ where the ℓ_i and μ_j are strings, the string $\ell_1\ell_2 \cdots \ell_k$ is in L, and the string $\mu_1\mu_2 \cdots \mu_k$ is in M, for $k = 1, 2, 3, \ldots\}$.

Suppose that L is regular and M is a CFL.
Must $\mathit{Shuffle}(L, M)$ be a CFL? Explain your answer.

Solution

Yes, $\mathit{Shuffle}(L, M)$ is a CFL. Let M_r be the FSM for L, and M_p be the PDA for M. The idea is that a slightly FSM-enhanced version of the PDA M_p should run for as many steps as it likes, while being ready to nondeterministically epsilon-transition back to a slightly FSM-enhanced M_r that can run for as many steps as it likes, and non-deterministically epsilon-transition back to the PDA. Each "epsilon-transition back" from machine M_x is to the state in the other machine $M_{other\ one}$ that executed the most recent transition into M_x. This way, each machine – when its switch-to-other transitions are omitted – will execute as it should on the subsequence of the string that it is supposed to recognize. As these words suggest, a state for the PDA must "remember" which state in the FSM caused the current sequence of execution steps by the PDA, and vice versa. We can record this by cloning $\#_p$ duplicates of M_r, where there are $\#_p$ states in the PDA finite state machine M_p. Likewise, we clone $\#_r$ duplicates of M_p. Let theses many states be written as $(previousPDAstate, currentFSMstate)$, where each such pair defines one new state and the first index in this pair is conceptually a "level number" for the altogether $\#_p$ levels of the current state "FSMstate," which belongs to the machine M_r. So each pair is a state in the new machine, and there are also the pairs $(previousFSMstate, currentPDAstate)$ that represent the different PDA levels which each remembers the name of the FSMstate that transitioned to the PDA execution.

A state $(previousPDAstate, z)$ will have all of the transitions from z that are defined for z in M_r. These transitions will be to the clones of the states out of z in M_r, and will reside on the level "previousPDAstate." In addition, $(previousPDAstate, z)$ will have an epsilon transition to $(z, previousPDAstate)$, which represents a return to the PDA recognition scheme, and the recording of the fact that z was the source of that return.

The analogous construction is used for the $(previousFSM, currentPDAstate)$ states.

The start state is $newS$, which has an epsilon transition to $(previousPDAstart, currentFSMstart)$, and another epsilon transition to $(previousFSMstart, currentPDAstart)$ (because ℓ_1 could be ϵ for those who care).

A state in this new machine is accepting if both the PDA and FSM states are accepting.

The machine is a PDA, and it is clear that it recognizes the shuffles as described.

Problem 3 **[10 points]** *New numbered exam booklet PLEASE*
A string is a palindrome if it is the same read left to right or right to left, e.g., $ababa$.

(a) The longest palindrome subsequence of a string x is its longest subsequence (of not necessarily consecutive symbols) that is a palindrome. Give an algorithm to determine in time $O(|x|^2)$ the length of the longest palindrome subsequence of a string x. A high-level program specification is sufficient, but it must have enough specificity to be correct, and have an operation count of $O(|x|^2)$. What is the space complexity (requirement) for your solution?

Solution

Solution 1: let $Len(i,j)$ be the length of longest palindrome in $x[i..j]$. then:

$$Len(i,j) = \begin{cases} 0 & \text{if } j < i, \\ 1 & \text{if } i = j, \\ 2 + Len(i+1, j-1) & \text{if } x[i] = x[j], \\ \max\{Len(i+1,j), Len(i,j-1)\} & \text{otherwise,} \end{cases}$$

and the cases are processed as being mutually exclusive. The exponential growth of a purely recursive solution is tamed by storing $Len(i,j)$ in an $|x| \times |x|$ array that is initialized to `Nil`. A call to compute $Len(i,j)$ only initiates the recursion if the corresponding table entry is `Nil`. If so, the answer is computed as above, and then written into the table prior to the return. Otherwise the table value is returned. Consequently, each value is computed just once for each index pair. It follows that the spatial and time complexity are $\Theta(|x|^2)$.

Cheap solution: Let y be the reverse of x, and apply the standard textbook Greatest Common Subsequence algorithm. Include a look-up table to keep the recursive computation efficient.

Technically, this answer is correct, but flawed. It is correct in the sense that the answer is numerically correct. It is flawed in the sense that the character locations for a GCS in x and $x^{reversed}$ does not have to reference the same sequence of physical characters. Example: ZABZA and AZBAZ has the GCS ABZ. Of course, in addition to this Bidendrome, ZABZA has four Palindromes of length 3. Exercise (that was not part of the problem): Prove that for any string x, the length of the longest palindrome is the same as the length of the GCS of x and $x^{reversed}$.

(b) Give an algorithm to determine in time $O(|x|^2)$ the length of the longest palindrome that is a sequence of consecutive symbols in a string x. What is the space complexity?

Solution

Done in c below.

(c) Give an algorithm to determine in time $O(|x|^2)$ the length of the longest palindrome that is a sequence of consecutive symbols in a string x, and which uses no more than $O(|x|)$ space. (Hint: think about the logic of a non-deterministic PDA that recognizes palindromes, but do not use the code for such a device.)

Solution

If we can guess the middle, we will only need to read pairs of letters to determine the length of this thing. In this case, a look-up table will be unnecessary.

A version of the code is as follows:

```
function Long(n,x[..n]);
   if $n$ equals 0 then return(Stupid case nobody cares about) endif;
   $x[0] \leftarrow \pi$;   { $x(0)$ is a sentinel to keep reads in bounds }
   $x[n+1] \leftarrow \rotatebox{180}{$\pi$}$;   { $x(n+1)$ is a sentinel to keep reads in bounds }
   $longest \leftarrow 1$;
   for $lmid \leftarrow 1$ to $n$ do
```

```
        $lltest \leftarrow lmid$;
        $rrtest \leftarrow lmid + 1$;
        $current \leftarrow 0$;
        while $x[lltest]$ equals $x[rrtest]$ do     { the even length cases }
          $current \leftarrow current + 2$;
          $lltest \leftarrow lltest - 1$;
          $rrtest \leftarrow rrtest + 1$
        endwhile;
        $longest \leftarrow \max\{longest, current\}$;
        $lltest \leftarrow lmid - 1$;
        $rrtest \leftarrow lmid + 1$;
        $current \leftarrow 1$;
        while $x[lltest]$ equals $x[rrtest]$ do     { the odd length cases }
          $current \leftarrow current + 2$
          $lltest \leftarrow lltest - 1$;
          $rrtest \leftarrow rrtest + 1$
        endwhile;
        $longest \leftarrow \max\{longest, current\}$
      endfor;
      return($longest$)
    end_Long;
```

The time is $O(|x|^2)$ and the additional space for this version is a constant number of words.

A clever problem reduction technique can improve the performance to $O(n \log n)$ operations with $\theta(|x|)$ additional words of storage, and the use of a standard, but reasonably sophisticated compact suffix tree data structure can reduce the operation count to $\Theta(|x|)$ and $\Theta(|x|)$ storage.

Problem 4 [10 points] *New numbered exam booklet PLEASE*

Value at risk

ScotiaMocatta, a global leader in the precious metals futures market, needs a data structure S to record its commitments to provide silver over time.

A record $R = (s, d, p)$ in S has three relevant fields: s is the selling date, which is when the contract is sold; d is the delivery date, at which time the actual silver must be delivered by ScotiaMocatta. p is the number of pounds of silver that must be supplied on the delivery date. You can denote them by $R.sell$, $R.del$, and $R.lbs$ if you wish.

The operations are insert R, delete R (for contracts that were not sold), and $PoundsatRisk(t)$.

$PoundsatRisk(t)$ is the number of pounds of silver that ScotiaMocatta, at date t, has already promised to deliver on that day or some time in the future. It equals the sum of the pounds of silver in the contracts with a selling date that is less than t, and a delivery date that is greater than or equal to t.

Formally, for the records (s, d, p) in the selling system S:

$$PoundsatRisk(t) = \sum_{\substack{(s,d,p)\ in\ S \\ s<t\leq d}} p.$$

To comply with financial regulations, the data structure must store records for old contracts that have been completed as well as contracts that ScotiaMocatta plans to sell in the future. Likewise, the query time t can be any time in the past, the present, or the future.

Explain how to implement a data structure S so that the $Insert(*)$, $Delete(*)$, and $PoundsatRisk(*)$ operations will all run in $O(\log n)$ time when S contains $n > 1$ records. There is no requirement that a record R have just one physical entry in S.

Solution

Use two 23trees T_1 and T_2 that are enhanced to answer these range queries. Let T_1 store all records (s, d, p) in a sorted order according to the major key s and minor key d. Let T_2 store all records (s, d, p) in a sorted order according to the major key d and minor key s. In addition, let each internal node v in T_1 and in T_2 have a special field $v.pounds$, which stores the total number of pounds entries in the leaves of the subtree rooted by v.

So insertion and deletion is standard. The only enhancement is that during insertion, each vertex on the path from the root of each tree to the newly inserted leaf must have its pound field increase by the pounds entry in the newly inserted record. Deletes are managed analogously. It is straightforward to adapt the resulting splits and joins of vertices due to insertions and deletions to accommodate these enhancements.

Given the query time t, we traverse T_1 with respect to the major key s and minor key ∞. On the path from the root to that newly found location, we add up all of the $.pounds$ values in the vertices that are children of the vertices in the path, and lie to the left of the path. Let this sum be $Psold$, which equals the pounds of silver sold before – or at time t.

Do the same for T_2. Let this sum be $Pdelivered$. Then the answer to the query is $Psold - Pdelivered$.

Problem 5 [10 points] *New numbered exam booklet PLEASE*
Consider the following problems:

(a) Balancing Unfair Odds and Evens (BUOE)
Instance: a set S that contains n integers that are multiples of 32, and n odd integers.
Question: Can S be partitioned into the two disjoint sets M and W with sums that differ by at most 1?
Prove that BUOE is NP-Complete.

Solution

BUOE is NP-Complete.

Proof:

It is clear that BUOE is in NP: if the answer is yes, then the partition can be presented and verified in polynomial time.

We encode Partition as a BUOE problem. So let $A[1..n]$ be an array of n integers for the Partition problem. We encode the problem as the two arrays $TrtytuNA[1..n]$ and $Odd[1..n]$ where $Odd[1..n]$ are all equal to 1, and $TrtytuNA[i]$ is set to $32nA[i]$ for $i = 1, 2, \ldots, n$. So if A does have a partition, then so does $TrtytuNA$, and $Odd[1..n]$ can be split into two sets of $n \div 2$ ones, and $n - (n \div 2)$ ones. It is clear that one of the $TrthtuNA$ partitions and one of the Odd partitions can be combined as M, and the remaining data combined as W, and sums will differ by 1 or 0. Conversely, any partitioning of $TrtytuNA$ will give two sums that differ by some multiple of $32n$, so n ones cannot be distributed to the two sums in a way where each new sum will be within one of the other unless the two sums are equal without any ones.

It follows that Partition and its encoding as a BUOE problem have identical yes-no answers, and the encoding runs in polynomial time. Hence BUOE is NP-Complete. ∎

(b) Dominating Set (DS)
Instance: $G = (V, E)$ is an undirected graph of n vertices, and a target t.
Question: Does V contain a subset S of t vertices where every vertex in V is either in S or is adjacent to

some vertex in S? Prove that DS is NP-Complete.
Note: although DS might seem the same as vertex cover (VC), there are differences:
in the graph, w—x—y—z, the set $\{w, z\}$ is a DS, but is not a VC because $\{x, y\}$ does not have an endpoint in $\{w, z\}$.

Solution

DS is NP-Complete.

Proof:

It is clear that DS is in NP, since if the answer is yes, then there is such an S, and it is easy to verify any such S in polynomial time.

We reduce VC to DS. Given an undirected graph $H = (W, F)$, we build G as follows: For each w in W, insert a clone copy of w into V. For each edge $e = \{u, v\}$ in F, insert a clone copy of e into E, and create the new vertex ν_e. Insert ν_e into V, and the edges $\{\nu_e, u\}$ and $\{\nu_e, v\}$ into E. It is clear that a VC for H is a DS for G. To see the converse, let S be a DS for G. Then at least one of the vertices u, v, ν_e must be in S for each $e = \{u, v\}$ in E. In those cases where only ν_e is in S, we can replace it by one of the other two vertices, and the new S will still be a DS for G. It will also be a VC for H, since every edge in H will have at least one of its vertices in S.

It follows that VC and its encoding as a DS problem have identical yes-no answers, and the encoding runs in polynomial time. Hence DS is NP-Complete. ∎

Problem 6 **[10 points]** *New numbered exam booklet PLEASE*
Let $G = (V, E)$ be a weighted directed graph with vertices $V = \{1, 2, 3, \ldots, n\}$, and the real valued edge weight function $Eweight(i, j)$ for i, j in V. As is standard, we define the weight of a path p to be the sum of the $Eweight$ weights of the edges in p. Assume that G has no cycles with a negative weight, and that $Eweight(i, j) = \infty$ if the directed edge (i, j) is not in E.

(a) Present pseudocode for the standard all-pairs least-weight paths algorithm of Floyd-Warshall and state its space and time complexities. Include some explanation about how the least-weight path for any (i, j) pair can be recovered from the computation.

Solution

```
procedure FW(n,Eweight[1..n,1..n],Pcost[1..n,1..n],Intermediate[1..n,1..n]);
  for all pairs $(i,j)$ in $[1..n] \times [1..n]$ do
    $Pcost[i,j] \leftarrow Eweight[i,j]$;
    if $Pcost[i,j] < \infty$ then $Intermediate[i,j] \leftarrow j$ else $Intermediate[i,j] \leftarrow$ Nil endif
  endfor;
  for $k \leftarrow 1$ to $n$ do
    for all pairs $(i,j)$ in $[1..n] \times [1..n]$ do
      $temp \leftarrow Pcost[i,k] + Pcost[k,j]$;
      if $temp < Pcost[i,j]$ then
        $Pcost[i,j] \leftarrow temp$;
        $Intermediate[i,j] \leftarrow k$
      endif
    endfor
  endfor
end_FW;
```

Path recovery is performed via a DFS to print the leaf values of the implicit binary tree of subpaths with root $Intermediate[i, j]$.

(b) Now suppose that the weight of a path p is defined to equal the largest edge weight among the edges in p instead of the sum. Present pseudocode to solve the all-pairs least-weight paths problem for this modified definition of path weight.

Solution

Change line 7 to $temp \leftarrow \max\{Pcost[i,k], Pcost[k,j]\}$

(c) Suppose that all edge weights are non-negative. Explain how to solve the all-pairs least-weight paths problem for sparse graphs (where $|E| \ll |V|^2$) with a method that is more efficient than the Floyd-Warshall algorithm. Give the time complexity of your algorithm. You can use any standard methods, but be sure to name them and state their run-time costs.

Solution

Use n iterations of Dijkstra's algorithm, where the iterations sequence through the different vertices to use as the source. Time: $\Theta(n \times (|E| + n \log n))$. Karger, Koller, and Phillips (1993) integrated (and optimized) this approach by loading all n^2 initial path lengths into the priority queue, and executing the Dijkstra just-in-time update policies for each newly deleted min distance. They optimized the updates by (effectively) building new adjacency lists from the growing subset of edges that participate in shortest paths, and adapting the just-in-time updates to accommodate late (but not too late) discoveries of edges that are shortest paths. The operation count for their version is $\Theta(n \times (|F| + n \log n))$, where F is the set of edges that are also shortest paths.

(d) Now suppose that the objective is to solve not only the least-weight path costs for the standard definition of path weight as in part (a), but also to compute the largest edge weight on each solution path. If a pair of vertices i, j have several (equal) least-weight paths from i to j, the algorithm should find the path from i to j that not only has the least total path weight, but also has, among all such least-weight paths, a greatest weight edge that is as small as possible, which is to say that the greatest weight among its edges is less than or equal to the greatest weight edge in any of the other least-weight paths from i to j. Present pseudocode to solve this problem. Note that you will need two output arrays: one for least weight path for each (i, j) pair, and one for the largest weight edge on each such path.

Solution

```
procedure FWW(n,Eweight[1..n,1..n],Pcost[1..n,1..n],Epcost[1..n,1..n]);
  for all pairs $(i,j)$ in $[1..n] \times [1..n]$ do
    $Pcost[i,j] \leftarrow Eweight[i,j]$;
    $Epcost[i,j] \leftarrow Eweight[i,j]$
  endfor;
  for $k \leftarrow 1$ to $n$ do
    for all pairs $(i,j)$ in $[1..n] \times [1..n]$ do
      $temp \leftarrow Pcost[i,k] + Pcost[k,j]$;
      $etemp \leftarrow \max\{Epcost[i,k], Epcost[k,j]\}$;
      if $temp < Pcost[i,j]$ then
        $Pcost[i,j] \leftarrow temp$;
        $Epcost[i,j] \leftarrow etemp$
      elseif $temp$ equals $Pcost[i,j]$ then
        $Epcost[i,j] \leftarrow \min\{etemp, Epcost[i,j]\}$
      endif
    endfor
  endfor
end_$FWW$;
```