Algorithms Written Qualifying Exam
Department of Computer Science
New York University
May 1, 2013

This examination is a *three hour* exam.
All questions carry the same weight.
Answer all of the following *six* questions.

Please be aware that to pass this exam you need to provide good answers to several questions; it is not sufficient to obtain partial credit on each question.

- 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 legible, and brief but precise. Assume standard results, except where asked to prove them.
- When you have completed the exam, please reinsert the booklets back into the envelope.

Problem 1 [10 points]
Consider the following Postal Route Reduction (PRR) Problem.

> Input: Directed graph G and a set $S = \{R_1, R_2, \ldots, R_l\}$ of routes (circuits) starting and ending at a vertex s (the depot), plus an integer k.
> Question: Is there a subset of k of the routes in S that between them include every vertex in the graph? (These k routes may include a vertex more than once.)

Show that PRR is NP-Complete.
Hint. You might do a reduction from Vertex Cover to PRR, or from 3-SAT to PRR.

Recall that Vertex Cover is the following problem.

> Input: Undirected graph $G = (V, E)$, and an integer k.
> Question: Does G have a vertex cover of size k?
> A vertex cover is a set $U \subset V$ such that every edge in E has at least one endpoint in U.

Problem 2 [10 points]
Let $G = (V, E)$ be a directed graph, where each vertex has an associated color. A path in G is called *k-alternating* if, as one traverses the nodes along the path, the colors of the nodes change exactly k times. For example, a path whose nodes (in order) are colored *red*, *red*, *blue*, *green*, *green* would be a 2-alternating path. Note that such a path need not be simple: it may or may not contain repeated nodes.

Your task is to design and analyze an efficient algorithm that takes such a graph G as input, and outputs the *maximum* value of k for which a k-alternating path exists in G, or ∞ is there is no maximum (which means that there are k-alternating paths in G for arbitrarily large values of k).

Your algorithm should run in time $O(|V| + |E|)$. You may assume the graph is represented in standard adjacency list form, and that the color of a node v is given by $C[v]$, where C is an array indexed by node indices.

Problem 3 [10 points]
The oil-rich country Sudway has build a single long highway. Now, a little late, it is planning to add rest stops on heavily traveled portions of the highway in the least expensive way possible.

In more detail, the input is a path $v_1, v_2, \ldots, v_n$ of n vertices. The input also includes for each vertex v_i a positive cost c_i, and a collection $\mathcal{I}$ of intervals $I_j = [v_{l_j}, v_{r_j}]$, with $l_j \le r_j$, for $1 \le j \le m$. Your task is to choose a minimum cost subset of the vertices on which to place rest stops subject to the constraint that each interval contains at least one rest stop. It will cost c_i to build a rest stop at vertex v_i.

More precisely, your task is to give a polynomial time algorithm to determine the minimum cost for building a collection of rest stops that meet this condition.

Go To the Next Page

Problem 4 **[10 points]** Consider a variation of the Towers of Hanoi problem in which there are 4 posts and n disks. All of the usual rules apply: the only difference is that there are now 4 posts instead of the usual 3.

Design and analyze a strategy for moving all n disks from one post to some other post. Your strategy should use at most $2^{O(\sqrt{n})}$ steps; that is, the number of steps should be bounded by $2^{c\sqrt{n}}$ for some constant c.
Hint: a. You are free to use the fact that the 3-post version of this puzzle can be solved using a strategy that uses $2^n - 1$ steps.
b. Note that $2^{\sqrt{n}} \times 2^{\sqrt{n}} = 2^{O(\sqrt{n})}$.

For completeness, we recall the details of the Towers of Hanoi puzzle. You may skip this description if you are already familiar with it. At the beginning of the puzzle, there are a number of disks, all of different sizes, stacked on one post, in order of decreasing size: the smallest disk is on the top of the stack, with the second smallest underneath that, followed by the third smallest, and so on, with the largest disk on the bottom of the stack. In each step of the puzzle, you may remove one disk from the top of a stack on one post and place it onto the top of a stack on another post, subject to the following constraint: you may only place a disk x on top of a stack that is either empty or whose topmost disk is smaller than x.

Problem 5 **[10 points]** A sequence $x_1, x_2, \ldots, x_n$ of n integers is said to be *smooth* if successive integers differ in value by at most 1; e.g. the sequence $1, 1, 2, 3, 3, 2$ is smooth, but the sequence $1, 2, 4, 3$ is not.

The sequence may be updated by a D&I (duplicate and increment) operation. Specifically, D&I(i) removes the ith element, x_i, from the sequence, and replaces it with two consecutive copies of $x_i + 1$. Thus, if the current sequence is $1, 1, 2, \mathbf{3}, 3, 2$, then D&I(4) will replace the first 3 in the sequence to produce the result $1, 1, 2, \mathbf{4}, \mathbf{4}, 3, 2$. Note that the resulting sequence is not smooth.

The smooth version of the operation, denoted by sD&I(i), after performing D&I(i), carries out the minimal number of D&I's to make the sequence smooth once again. Note that sD&I(i) is uniquely defined. For instance, starting with (1,1,2,3,2,2), the operation sD&I(4) will produce the smooth sequence $1, \mathbf{2}, \mathbf{2}, \mathbf{3}, \mathbf{3}, \mathbf{4}, \mathbf{4}, \mathbf{3}, \mathbf{3}, 2$.

Suppose that the initial input is a smooth sequence of n integers and suppose m sD&I operations are performed on this sequence. You are to show that a total of $O(m + n)$ D&I operations are performed.

Hint: Use an amortized analysis, either with a potential function or a credit argument.
a. Call a sequence of the form $x_i, x_i - 1, x_i - 2, \ldots, x_i - m, y$, where $y \geq x_i - m$, a length m *right chain* for x_i. Similarly, call the sequence $y, x_i - m, \ldots, x_i - 2, x_i - 1, x_i$ where $y \geq x_i - m$, a length m *left chain* for x_i.

Suppose that the operation sD&I(i) is performed, where x_i is the ith element in the sequence. What is the result of the operation sD&I(i) on the left and right chains, if any, starting at x_i?

b. What are the lengths of the chains resulting from the operation sD&I(i) from part a?

c. Provide charges (or credits), as needed, to some of the integers in a chain to pay for possible future D&I operations on these integers. Which integers should receive such charges? Now show that any m sD&I operations result in a total of $O(m + n)$ D&I operations being performed.

Problem 6 [10 points]

We view a binary search tree (BST) as storing a set of *items*, where an item is a (key, data) pair. Let T be a full binary search tree: full means each internal node has two children. Suppose that T also satisfies the following *BST property*: if u_L (resp. u_R) is a node in u's left subtree (resp. right-subtree), then

$$u_L.\text{key} < u.\text{key} \leq u_R.\text{key}. \tag{1}$$

We call T an *external* BST (EBST) if, in addition, the keys in the leaves are all distinct, the internal nodes do not hold any data, and each leaf u stores some data, u.data. Thus a leaf u holds the item $(u.\text{key}, u.\text{data})$. Keys in the internal nodes may be duplicates of the keys in the leaves; they are used for navigation.

a. Describe LookUp and Insert algorithms for an EBST. You must specify how the keys in the internal nodes are introduced and used. We suggest drawing figures to supplement your explanations.

b. Recall (but DO NOT describe) the splay operation which can be applied to nodes of a binary tree. Provide splay versions of the operations in Part(a) so as to achieve $O(\log n)$ amortized time for each operation (again, use the splay operation as needed without further description). Explain why this $O(\log n)$ time bound is achieved, assuming the results from a standard splay analysis.

c. Recall the Huffman coding problem: Given a string $s = a_1 a_2 \cdots a_m \in \Sigma^*$, where Σ is a finite set of symbols, we want a prefix-free code $C : \Sigma \to \{0,1\}^*$ such that the encoded string $C(s) = C(a_1)C(a_2)\cdots C(a_m)$ has minimal length. We will obtain the code C from an EBST T in which the symbols in Σ are stored as keys in the leaves of T: for $a \in \Sigma$, the code $C(a)$ is just the binary encoding (0=left, 1=right) of the path from the root to the leaf storing a.

Now assume $\Sigma = \{0,1\}^8$ represents the set of ASCII characters. In *dynamic Huffman coding*, we encode only those symbols that have been read so far. As more symbols are read, the corresponding EBST T will change. Describe a protocol for online encoding of the string s using the EBST T from part (b).
Hint: You need not describe the decoding protocol, though you ought to keep it in mind. Assume that T initially stores only one special character $\mathbf{0} \notin \Sigma$ and that on extending the natural sorted order on Σ to include $\mathbf{0}$, $\mathbf{0}$ is defined to precede every character in Σ.

What must you transmit if a character a_i is not in T?

Solution to Problem 1 First, we note that PRR is in NP: the certificate comprises k routes that between them include all the vertices in the graph. To verify a certificate it suffices to check that each route is indeed a path from s back to s, and via a sort, that the collection of vertices in these paths includes all the vertices in G. Clearly, the verification takes time polynomial in the size of G and the certificate, and further the certificate has size at most $k \cdot n$, where n is the number of vertices in G.

To show NP-hardness, we give a reduction from Vertex Cover (VC). Let (G, k) be an input instance for VC with $k \leq |V|$ (for otherwise the VC instance is trivially solvable). We build an instance (H, S, h) of the PRR problem such that $(G, k) \in \text{VC} \iff (H, S, h) \in \text{PRR}$, as follows.

Recall that $G = (V, E)$. H has the set of vertices $\{s\} \cup V_E$ where V_E has a vertex v_e for each edge $e \in E$, and s is an additional vertex not in V_E. We call V_E the set of edge vertices. For simplicity, we make H a complete graph, i.e. every possible edge in both directions. For each vertex $u \in V$, we create a route R_u in S. This route starts and ends at s and includes in addition exactly those vertices v_e for which e is incident on u. The order (to be specific) is the same as the order in the adjacency list in the graph G. Finally, we set $h = k$. Clearly, this instance of the PRR problem can be built in time polynomial in (G, k).

Now if G has a vertex cover $\{u_1, u_2, \ldots, u_k\}$, then the routes that solve the PRR instance are $R_{u_1}, R_{u_2}, \ldots, R_{u_k}$, for they include s plus all the vertices corresponding to edges covered by $u_1, u_2, \ldots, u_k$, i.e., $V_E \cup \{s\}$.

While if the PRR problem can be solved by the k routes $R_{u_1}, R_{u_2}, \ldots, R_{u_k}$, then these routes include all the edge vertices in V_E, and correspondingly $\{u_1, u_2, \ldots, u_k\}$ forms a Vertex Cover of G.

Thus $(G, k) \in \text{VC} \iff (H, S, h) \in \text{PRR}$, as required.

Solution to Problem 2 To determine k, the algorithm runs as follows. First, it computes the strongly connected components, together with a topological sort of the component graph, using standard linear time algorithms.

Now, if any one component contains nodes of different colors, then $k = \infty$, and we are done. Otherwise, $k < \infty$, and we determine k as follows. Since every component has nodes only of one color, let us assign to each node of the component graph this common color. Assume that the nodes of the component graph in topological order are $v_1, \ldots, v_k$, where $k \leq n$, the number of nodes in the input graph. For each vertex v_i we compute the length of the longest path starting at v_i, where length is measured as the number of color changes along the path; this value will be reported in array $T[i]$.

We initialize an array $T[i] \leftarrow 0$ for $i = 1, \ldots, n$, and proceed as follows:

```
for $i \leftarrow n$ down to 1 do
    for each successor $v_j$ of $v_i$ do
        if $v_j$ and $v_i$ have different colors then
            $T[i] \leftarrow \max(T[i], T[j] + 1)$
        else
            $T[i] \leftarrow \max(T[i], T[j])$
```

Finally, output $\max(T[1], \ldots, T[n])$.

Correctness and running time should be clear.

Solution to Problem 3 We use dynamic programming.

Note that if an interval I is fully contained in another interval J then any rest stop in the interval I will also be in the interval J. So we can safely ignore any such intervals J. A preliminary pair-wise comparison of every pair of intervals suffices to eliminate all such intervals J. This takes $O(m^2)$ time. (More efficient algorithms are possible, but are not needed for the purposes of answering this question.)

We sort the remaining m' intervals by their right endpoints, and for simplicity we denote the sorted order by $I_1, I_2, \ldots, I_{m'}$. As the right endpoints can be converted to the integers $1 \ldots n$ by traversing the path, we can use a radix sort. (In fact, this gives the same order as sorting by the left endpoint, as none of the remaining intervals is fully contained in another interval.)

Let $C(k)$ be the minimum cost for placing rest stops so as to handle all the intervals $I_1, I_2, \ldots, I_k$.

We compute $C(k+1)$ as follows. Clearly, one rest stop is needed in the interval I_{k+1}. If this rest stop is on node v_j, further rest stops are needed to cover all the intervals whose right endpoint precedes v_j. To help capture this constraint, we define $s_j = \max_i\{i \mid r_i < j\}$.

Then $C(k)$ is given by the following recursive formula:

$$\begin{aligned} C(k) &= \min_{l_k \le j \le r_k} \{c_j + C(s_j)\} \qquad k \ge 1 \\ C(0) &= 0 \end{aligned}$$

If we use dynamic programming, there are m' terms to compute, and aside recursive calls, each takes $O(n)$ time to compute, for a total of $O(n \cdot m')$ time.

In addition, we need to sort the intervals by their right endpoint, which takes $O(m')$ time; we need to compute the values s_j, which requires a traversal of the sorted right endpoints, and takes a further $O(m' + n)$ time.

Thus the total cost of this algorithm is $O(m^2 + m' \cdot n) = O(m^2 + mn)$. (In fact, $m' = O(n)$.)

Solution to Problem 4 Consider the following recursive algorithm, to move n items from post A to post B (suppose A, B, C, D are the names of the 4 posts). The algorithm uses a parameter k, which is determined below, and which remains fixed throughout the recursion.

Move(n, A, B, C, D):

1. Recursively move the top $n-k$ disks from post A to post C.
2. Use the 3-post algorithm to move the bottom k disks from post A to post B (post C, where the top $n-k$ disks now reside cannot be used during this step).
3. Recursively move the top $n-k$ disks to post B, the post where the bottom k disks now reside.

Note that the recursion stops when we have k or fewer disks, in which case we just use the 3-post algorithm.

At every level of recursion, the number of disks decreases by k, and so the total number of levels is $\lceil n/k \rceil$. Thus, there are $O(2^{n/k})$ nodes in the recursion tree. Also, each node in the recursion tree (i.e., each recursive invocation) contributes a cost of (at most) $2^k - 1$, which arises from running the the 3-post algorithm.

It follows that the total cost of the recursive algorithm is

$$O(2^{n/k+k}),$$

and setting $k = \lfloor \sqrt{n} \rfloor$ yields the desired result.

Solution to Problem 5 a. Suppose the right chain is $x_i, x_i - 1, \ldots, x_i - m, y$ of length m. Then sD&I(i) will perform D&I(i), followed by m additional D&I operations to modify the right chain into

$$x_i + 1, x_i + 1; x_i, x_i; x_i - 1, \ldots, x_i - m + 2; x_i - m + 1, x_i - m + 1; y$$

If the left chain has length $\ell \geq 0$, we perform ℓ further D&I operations (for a total of $1 + m + \ell$ operations).

b. If $m \geq 0$, then the right chain will be transformed into m right chains of length 1, plus possibly another chain of the form $x_i - m + 1, y, \ldots$. This last chain has length 0 if $y = x_i - m + 1$; otherwise $y = x_i - m$ and the last chain will have length 1 more than the right chain length of y.

Similar remarks apply to the left chain of x_i.

c. We use a potential function argument. A unit will cover the cost of one D&I operation.

Suppose that $v_i, v_i - 1, \ldots, v_i - m, w$ is a maximal right chain (so the item u to the left of v_i satisfies $u \leq v_i$). Then we say v_i is the starter item for a right chain, and the successive items to its right are the first, second, third, ..., items on this right chain; w is called the last item on the chain. An analogous definition is used for the left chain.

We define $\phi(i) = 1$ if the ith item is on a chain, but is neither the starter item, the first item, nor the last item; otherwise, $\phi(i) = 0$. We will sometimes write $\phi(\text{“}x_i\text{”})$ instead of $\phi(i)$ when this is the clearer way of identifying a particular item. For the example maximal chain, $\phi(\text{“}v_i\text{”}) = \phi(\text{“}v_i - 1\text{”}) = \phi(\text{“}w\text{”}) = 0$, and $\phi(\text{“}v_i - 2\text{”}) = \ldots = \phi(\text{“}v_i - m\text{”}) = 1$. The potential ϕ is given by the sum $\sum_{i=1}^{n} \phi(i)$.

When the operation sD&I(i) is performed, D&I operations will be performed on items $x_i, x_i - 1, \ldots, x_i - m$ in the right chain, but not on item y, and similarly for the left chain. If operation D&I(j) is performed and $\phi(j) = 1$ then this unit of potential is used to pay for the D&I(j) operation. The only items for which a unit of potential might not be available are x_i and the items immediately to its left and right. Paying for the D&I operations on these items costs at most 3 units. By part (b), the only items that may gain a unit of potential are the new second items on the chains immediately beyond x_i's chains. This causes a further charge of up to 2 units. Thus the total cost is 5 units.

Note that initially, the potential ϕ for the whole sequence is at most n. At the end of a sequence of m operations, the potential is at least 0. Therefore the decrease in potential is at most n. Thus the total amount of work (i.e., the number of D&I operations) is at most the decrease in potential plus $5m$. The total cost of this sequence of operations is therefore at most $n + 5m$.

Solution to Problem 6 a. LookUp is easy: we perform a standard binary tree search, except that we must not stop until we reach a leaf. At this point, we can report success if and only if the leaf contains the key being looked up.

Insert: To insert the item (K, D), we do a Lookup on key K. This leads to a leaf u with u.key $= K'$. If $K = K'$, then the insert fails. Otherwise, we now give u two children, u_L and u_K to store the keys K and K' (in the correct order). For instance, if $K < K'$, then u_L.key $\leftarrow K$, u_R.key $\leftarrow K'$, u_L.data $\leftarrow D$ and u_R.data $\leftarrow u$.data. Moreover, u.key $\leftarrow u_R$.key.

b. Splay version of Lookup: After we have reached a leaf u, we splay the parent of u. Splay version of Insert: After we have reached a leaf u, regardless of whether the insertion succeeds or fails, we splay the parent of u.

Why does this achieve $O(\log n)$ amortized cost? Well, the splaying cost is Θ of the cost of the Lookup or Insert operation, and we know that each splay operation has an amortized $O(\log n)$ cost.

c. The transmission protocol is as follows: assume that we have a EBST T that stores all the ASCII characters seen so far. T also stores a $\mathbf{0}$ that is smaller than any character in the string. We scan the input string $a_1 a_2 \cdots a_m$ and for each a_i, we do a Lookup on a_i. If a_i is found, we transmit the binary encoding of the path from the root to the leaf containing a_i; and we also splay the parent of the leaf. If a_i is not found, we transmit the path to $\mathbf{0}$, then transmit the ASCII code for a_i. Finally, we splay the parent of $\mathbf{0}$ and then insert a_i, followed by splaying the parent of a_i.

It should be clear that the receiver protocol can uniquely recover the string $a_1 \cdots a_m$.