Written Qualifying Exam
Analysis of Algorithms
(Fall 2005)

- This is a *three hour* exam.
- All *six* questions carry the same weight; however, to pass the exam, you need to provide good answers to several questions: it is not sufficient to obtain partial credit on each question.
- Print your name and SID on the front of the envelope only (not on the exam booklets).
- Answer each question in a *separate* booklet, and *number* each booklet with the corresponding question number.
- Read each question carefully. Keep your answers legible, and brief but precise.
- Assume standard results: cite standard theorems as necessary, and use standard algorithms as subroutines where appropriate.
- Good luck!

1. You are given an array $A[1 \ldots n]$ of items. An item x is called a *majority element* if it appears more than $n/2$ times in A. Clearly, a majority element, if it exists, must be unique.

   You are to design an algorithm that determines if A has a majority element, and if so, determines its value. However, your algorithm should observe the following restrictions:

   - The array A is stored in *read-only memory*.
   - Your algorithm should use at most $O(\log n)$ words of additional (read/write) memory (each word can hold numbers bounded by $O(n)$, and the output of your algorithm should be an index into A that indicates a position at which the majority element is located).
   - Your algorithm should run in time $O(n \log n)$, assuming comparisons of items take time $O(1)$.
   - Your algorithm should be deterministic.

   Solution. Divide and conquer: see if either $A[1 \ldots n/2]$ or $A[n/2+1 \ldots n]$ has a majority, and if so test if either of these is a majority for $A[1 \ldots n]$.

2. Consider the following recursive, probabilistic algorithm A. It takes as input list L of n items $x_1, \ldots, x_n$.

```
if n ≤ 1 then
    return
else
    for i ← 1 to 3 do
        initialize an empty list L_i
        for j ← 1 to n do
            flip a fair coin: if it comes up "heads," append x_j to L_i
        flip a fair coin: if it comes up "heads," recursively call A on input L_i
```

What is the expected running time of this algorithm? Give a careful justification of your claim.

Solution: $O(n)$. For $j = 0, 1, \ldots,$ let X_j be the number of items that are processed at level j. It is not hard to show that $E[X_{j+1}] = (3/4)E[X_j]$, and if $X = \sum_{j \geq 0} X_j$, it follows that $E[X] = O(n)$.

3. Given two strings $x, y \in \Sigma^*$, a *shuffle* of x and y is a string z such that

$$z = u_1 v_1 \cdots u_k v_k,$$

for some strings $u_i, v_i \in \Sigma^*$ with $u_1 \cdots u_k = x$ and $v_1 \cdots v_k = y$. That is, a shuffle is formed by concatenating symbols from x and y, while preserving the relative order of symbols in each of the strings x and y. For example, *a1bc23* is a shuffle of *abc* and *123* while *a2bc13* is not.

Also recall that for strings $z, w \in \Sigma^*$, we say that z is a *subsequence* of w if $z = z_1 \cdots z_k$, $w = w_1 \cdots w_\ell$, and there exist indices $i_1 < \cdots < i_k$ such that $z_1 = w_{i_1}, \ldots, z_k = w_{i_k}$.

Design and analyze an algorithm that takes three strings as input, $x, y, w \in \Sigma^*$, and determines if some shuffle z of x and y appears as a subsequence of w. Your algorithm should run in time $O(|x||y||w|)$.

Solution: Dynamic programming. Suppose $x = x_1 \cdots x_\ell$, $y = y_1 \cdots, z_m$, and $w = w_1 \cdots w_n$. The subproblems are $P(i, j, k)$, meaning that some shuffle of $x_i \cdots x_\ell$ and $y_j \cdots y_m$ appears as a subsequence of $w_k \cdots w_n$. The recursion is given by

$$\begin{aligned} P(i,j,k) \iff & (i = \ell + 1 \wedge j = m + 1) \\ & \vee (i \le \ell \wedge k \le n \wedge x_i = w_k \wedge P(i+1, j, k+1)) \\ & \vee (j \le m \wedge k \le n \wedge y_j = w_k \wedge P(i, j+1, k+1)) \\ & \vee (k \le n \wedge P(i, j, k+1)). \end{aligned}$$

The corresponding subproblem graph has $O(\ell m n)$ vertices, and each vertex has degree $O(1)$.

4. Consider the two languages:

$$A = \{a^m b^n c^{m+n} : m, n \in \mathbb{Z}_{\geq 0}\} \quad \text{and} \quad B = \{a^m b^n c^{mn} : m, n \in \mathbb{Z}_{\geq 0}\}.$$

(a) Show that A is not regular.

(b) Give a context free grammar for A.

(c) Show that B is not context free.

Solution.

(a) Let p be the pumping length from pumping lemma. Set $s = a^p b^p c^{2p} \in A$. The pump handle must be contained in the "a region" of s, so pumping up or down changes the number of a's, but not the number of b's or c's, so we get a string outside of A.

(b) Here is a CFG:

$$S \to aSc, S \to T, T \to bTc, T \to \epsilon.$$

(c) Let p be the pumping length from pumping lemma. Set $s = a^p b^p c^{p^2}$. The pumping handles can span only two consecutive regions, and each individual handle must lie in one region (otherwise, we pump our way out of $a^*b^*c^*$). If we only pump in the a and b regions, then the number of c's will clearly be wrong. So we must pump in the b and c regions, and both handles must be non-empty. Suppose one pump handle consists of k b's and the other consists of ℓ c's, where $0 < k < p$ and $0 < \ell < p$. Pump up once: we should have $p(p+k) = p^2 + \ell$, which implies $pk = \ell$, which implies ℓ is a multiple of p, which is impossible.

5. The **maximum revenue** problem is this. You are given a directed graph $G = (V, E)$, where each node in the graph has a certain amount of money sitting on it. More precisely, vertices $v \in V$ are labeled with values $p(v) \in \mathbb{Z}_{\geq 0}$, where $p(v)$ represents the number of dollars sitting on vertex v. You are also given vertices $s, t \in V$. The goal is to start at s and follow edges in G to reach t, picking up as much money along the way as you can. You may visit the same node twice, but once you have picked up the money at that node, it is gone. Define $M(G, p, s, t)$ to be the maximum number of dollars you can pick up on any path from s to t (paths need not be simple).

   (a) Design and analyze an algorithm to compute $M(G, p, s, t)$. Your algorithm should run in time $O(|V| + |E|)$.

   (b) Explain how to enhance you algorithm so that it computes and prints out a path (any path) from s to t with $M(G, p, s, t)$ dollars on it. Full credit goes for an $O(|V|^2)$-time algorithm, and partial credit for an $O(|V||E|)$-time algorithm. Just a very high-level English description of your algorithm will do — do not write detailed pseudo-code.

Solution: For part (a), first solve the problem on a DAG: this can be done directing using the known algorithm DAG-SHORTEST-PATHS in CLRS, converting vertices to edges and negating the p-values; alternatively, do a topological sort, and process nodes in reverse topological order (starting at t and ending at s), and computing the max of the sums. For general graphs, compute SCCs.

For part (b), we need to compute a path that sweeps out each SCC. Chose any vertex v in the component C and precompute paths from v to all $w \in C$ and from all $w \in C$ to v. This will take time linear in $|C|$ plus the edges of G connecting vertices in C. Now glue together paths, always returning to v.

It turns out that in the worst case, the shortest path that sweeps out an SCC may contain $\Omega(|C|^2)$ vertices, so you really can't do better (well, you can do it in linear time if you print out paths using a compact representation).

6. Consider the following modification to the maximum revenue problem stated in Question 5, called the **maximum profit** problem. The setup is the same as in Question 5, except now edges $e \in E$ have *costs* $c(e) \in \mathbb{Z}_{\geq 0}$. For any path from s to t, define its *net value* to be the sum of all money sitting on the *distinct* vertices in the path (again, the money is gone once you take it), minus the cost off *all* the edges in the path (you are charged $c(e)$ dollars each and every time you cross edge e). Let $N(G, p, c, s, t)$ be the largest net value of any path from s to t.

   Your task is to show that computing $N(G, p, c, s, t)$ is NP-complete, or more precisely, that the language

$$L = \{\langle G, p, c, s, t, k\rangle : N(G, p, c, s, t) \geq k\}$$

   is NP-complete.

   Hint: this can be solved using a very simple reduction.

   Solution: restriction to *HAMPATH*. Given an instance $\langle G, s, t\rangle$ of the Hamiltonian path problem, construct an instance $\langle G, p, c, s, t\rangle$ of the maximum profit problem as follows: set costs on edges to 1 dollar, put n dollars on every vertex, and set $k = n^2 - n + 1$ (or alternatively, 2 dollars per vertex suffices, with $k = n + 1$).