CORE EXAMINATION
Department of Computer Science
New York University
May 14, 2004

This is the common examination for the M.S. program in CS. It covers core computer science topics: Languages and Compilers, Operating Systems, and Algorithms. The exam has two parts. The first part lasts three hours and covers the first two topics. The second part, given this afternoon, lasts one and one-half hours, and covers algorithms.

Use the proper booklet for each question. Each booklet is marked with the Area and Question number, in the form PL&C1, PL&C2, PLC&C3, OS1, OS2, ALGS1, ALGS2, ALGS3. Use the appropriate booklet for each question. DO NOT put your name on the exam booklet. Instead, your exam number must be on every booklet.

You will be graded according to your exam number, shown on the envelope containing the booklets. Remember your exam number: when grades are given out, they will be published according to this number, not by name.

Make sure your name and signature are on the envelope. This is the only place where your name appears. Please include all the booklets inside the envelope. You can keep the exam.

Good luck!

Basic Algorithms

Question 1

This problem concerns the cost to glue wooden blocks together to form longer blocks. The rules for gluing blocks are as follows.

- Blocks can only be glued in pairs; the resulting object is a new block that is available for further gluing.
- If block A has length x and block B has length y, then the two blocks, when glued together, result in a block of length $x + y$.
- The cost to glue two blocks together is the length of the resulting block.

For parts 1 and 2 below, carefully describe your algorithm, and **explain** why it is correct and why the total cost is as requested.

1. You are given n blocks of length 1.

   Show how to glue then together with a total cost of $O(n \log n)$.

   Solution

   For expositional convenience, we can suppose that n is a power of 2. We imitate Mergesort. Glue the blocks together in $\log n$ rounds as follows. At round 0 you have n blocks of length 2^0. At round $k+1$, take the $\frac{n}{2^k}$ blocks of length 2^k, pair them up for gluing to get $\frac{n}{2^{k+1}}$ blocks of length 2^{k+1}. The number of blocks halves at each round, and each block participates in just one gluing at each round. Moreover, every block participates in a gluing event at each round. So the cost of a round is n, and the number of rounds in $\log_2 n$. If n is not a power of 2, there are $\lceil \log_2 n \rceil$ rounds. When there is an odd number of blocks, we omit gluing some longest block. Anyhow, there are less than $1 + \log n$ rounds and each round costs at most n. So the cost is still $O(n \log n)$.

2. You are given n blocks with the varying lengths $1,\ 2,\ 4,\ 8,\ \ldots,\ 2^n$.

   Show how to glue them together with a total cost of $O(2^n)$.

   Solution

   For each of $n-1$ rounds, glue together the two shortest blocks.

   Each round decrements number of blocks (at the least possible cost). There are $n-1$ rounds of gluing, which guarantees that the algorithm succeeds.

   The abstract gluing rule has the following pragmatic implication: After round k, the k (initially) shortest blocks are glued together to form a block of length $2^{k+1}-1$, and the remaining blocks have length 2^{k+1}, 2^{k+2}, …, 2^n. At the next round, the block with (original) length 2^{k+1} is glued to the composite block. The cost for the full $n-1$ rounds of gluing is $2^2 - 1 + 2^3 - 1 + 2^4 - 1 + \cdots + 2^{n+1} - 1 = 2 + 4 + 8 + \cdots + 2^{n+1} - (n-1)$. The total cost is exactly $2^{n+2} - 2 - (n-1) = 2^{n+2} - n - 1$, which is $O(2^n)$.

Question 2

There are n provinces numbered from 1 to n and each province is different. An advertising company has developed a program $Buyem(i, d)$ that, for province number i, computes the expected number of votes that a candidate will receive from province i if the candidate spends d advertising dollars in the province.

This year, candidate Kushberry has L dollars, and wants to spent them in a way that will maximize the total number of votes that she will get. Give a high level dynamic programming program specification to solve this problem. A recursive formulation (or recurrence equation) is sufficient, provided you state, in a sentence or two, how to make the code efficient. If there is no advertising for province j, the number of votes, of course, will be $Buyem(j, 0)$, so the problem is fully specified. You can assume that the function $Buyem(i, d)$ has been precomputed for all values of i and d, so that its value can be returned in constant time.

Your solution should have an operation count that is polynomial in L and n (i.e. bounded by $O(L^r n^s)$ for suitably small constants r and s.)

Solution

Most students assumed that L dollars should be spent somewhere, which is to say that advertising always helps. No points were deducted for (otherwise correct) solutions even if this assumptions was not explicitly stated. We will initially make this assumption, and then remove it.

Let $Winnings(J, D)$ be the expected number of votes that Kushberry will get from provinces 1 through J by spending D dollars in those J provinces in the best possible way.

A recursive specification for $Winnings(J, d)$ is:

$$Winnings(J, D) = \begin{cases} Buyem(1, D) & \text{if } J = 1, \\ \max_{0 \le d \le D}\{Winnings(J-1, D-d) + Buyem(J, d)\} & \text{if } J > 1 \end{cases}.$$

This formulation says that if there is just one province, then spend all D dollars there. Otherwise spend, for the best value of d, $D - d$ dollars in the first $J - 1$ provinces and d dollars in the J^{th} province. Use the recursive solution to determine the best way to spend the $D - d$ dollars in those first $J - 1$ provinces.

The solution is computed as $Winnings(n, L)$. If spending less money might win more votes, the solution is $\max_{0 \le \ell \le L}\{Winnings(n, \ell)\}$.

Dynamic programming ensures that the computation is efficient. That is, we make a table $Win[1..n, 0..L]$, and store in $Win[J, \ell]$ the solution for $Winnings(J, \ell)$. Each value is computed just once, and is subsequently accessed by table look-up. There are nL table entries. The recursive formulation above shows that each table entry is computed as the best of at most L different cases that each require a single access to Win, and a single call to $Buyem$. So the total work is $O(nL \times L) = O(nL^2)$.

Question 3

The Dagnabit problem is the following. Let G be a directed acyclic graph $G = (V, E)$ that is specified any way you like. Each vertex v in V has three fields. $v.val$ is a positive number that is already stored in the field. The fields $v.best$ and $v.where$ are initially undefined.

a) Write an efficient program that inputs G and, for each vertex x in G, stores, in $x.best$, the largest $.val$ entry in the portion of the graph reachable from x. That is, $x.best = d$ if either $x.val$ equals d, or there is vertex y that is a descendant of x where $y.val = d$, and d is the largest such value where this is true. (In a DAG, y is a descendant of x if there is a path in the DAG from x to y). For full credit, the program should solve the problem for all vertices of the graph with an operation count of $\Theta(|V| + |E|)$ (the number of vertices plus edges). Less efficient solutions will receive partial credit.
Hint: think about how to solve this problem for a tree.

b) Present the code that solves part a) and that also stores, for $x.where$, the name of the vertex that has its $.val$ number stored in $x.best$. For full credit, your solution should not use post-processing, and should run in the same time as your solution in part a.

Solution parts a and b combined

```
Procedure Driver;
    foreach vertex $v$ in $V$ do
        if $v$ not marked, then $DFS(v)$ endif
    endfor
end_Driver;


Procedure DFS(v);
    mark $v$;
    $v.best \leftarrow v.val$;
    $v.where \leftarrow v$;
    foreach vertex $w$ in $v$'s adjacency list do
        if $w$ not marked then $DFS(w)$ endif;
        if $w.best > v.best$ then
            $v.best \leftarrow w.best$;
            $v.where \leftarrow w.where$
        endif
    endfor
end_DFS;
```