PROGRAMMING ASSIGNMENT (Homework 10)

22C:44 Algorithms **Spring 2002**

Due Thursday, May 9, 2002

Introduction.
The well-known *bubblesort* algorithm sorts its input sequence by swapping adjacent elements whereas other sorting algorithms such as *quicksort* and *mergesort* sort their input by swapping arbitrary pairs of elements. It is possible to imagine algorithms that use operations quite different from swaps. For example, consider the following story of a disgruntled waiter (*American Mathematical Monthly*, 82(1), 1975, 1010):

> The chef in our place is sloppy, and when he prepares pancakes they come out all in different sizes. Therefore, when I deliver them to a customer, on the way to the table I rearrange them (so that the smallest winds up on top, and so on, down to the largest at the bottom) by grabbing several from the top and flipping them over, repeating this (varying the number I flip) as many times as necessary.

The operation the waiter is performing in this story could be called a "pancake flip," but in computer science literature this is called a *prefix reversal*. So the waiter aims to sort his stack of pancakes by using a series of prefix reversals.

Here is an example that shows a 10-element sequence being sorted by a series of prefix reversals. At each step, the underlined prefix of the sequence is reversed. It took me 12 prefix reversal operations to sort this sequence; can you do better?

6 9 1 0 4 2 3 7 8 5
0 1 9 6 4 2 3 7 8 5
5 8 7 3 2 4 6 9 1 0
2 3 7 8 5 4 6 9 1 0
9 6 4 5 8 7 3 2 1 0
7 8 5 4 6 9 3 2 1 0
4 5 8 7 6 9 3 2 1 0
9 6 7 8 5 4 3 2 1 0
6 9 7 8 5 4 3 2 1 0
8 7 9 6 5 4 3 2 1 0
7 8 9 6 5 4 3 2 1 0
9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9

Before you read any further, convince yourself that any sequence with n elements can be sorted with at most $2n$ prefix reversals. The specific problem that we would like you to solve is the following.

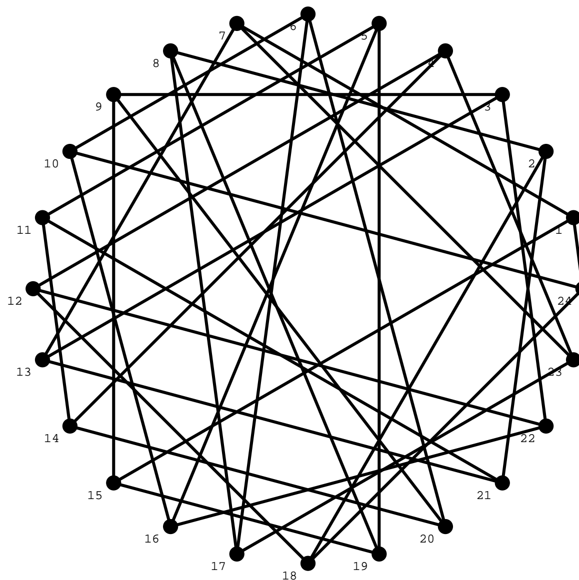
> **Minimum Prefix Reversals problem:**
> Given as input an arbitrary permutation P of the elements $1, 2, \ldots, n$, find the *smallest* number k of prefix reversals needed to sort P. Additionally, report a sequence of k prefix reversals that sorts P.

History.
The problem, as stated comes from the paper "Bounds for sorting by prefix reversal," *Discrete Mathematics*, 27 (1979), 47–57, by William H. Gates and Christos H. Papadimitriou. Yes, you guessed right, this is the same William Gates who founded the world's largest company in his spare time, when he was not trying to solve the pancake flipping problem. While the authors'

preface their paper with the story you see above, of the disgruntled waiter, the real motivation for the problem comes from computational biology. The general problem of finding the shortest way of getting from one permutation to another, using a limited set of operations is an abstraction of the problem of finding the evolutionary history of the genome. The contribution of the Gates-Papadimitrio paper was a proof that every n-permutation could be sorted in at most $(5n + 5)/3$ prefix reversals.

There is overwhelming evidence to indicate that the Minimum Prefix Reversals problem is impossible to solve efficiently. In his 1993 dissertation (University of Texas at Dallas), M. Heydari showed the problem to be NP-complete. For computer scientists, this essentially means that this problem cannot be solved efficiently.

Solution Approach.
Given that the problem has no efficient solution, we would like you to design a "somewhat slow" algorithm as opposed to a "painfully slow" algorithm. The most obvious approach to solving the problem is graph-theoretic. Define a graph $G_n = (V_n, E_n)$, where V_n is the set of all n-permutations (that is, permutations of elements 1 through n) and E_n is the set of edges that connects pairs of permutations that can be obtained from each other by a prefix reversal. The figure below shows G_4.



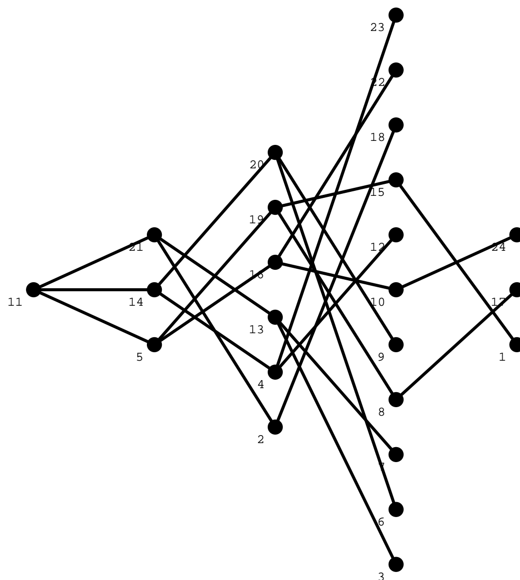
Since the number of 4-permutations is 24, G_4 has 24 vertices. In the above figure, these are labeled 1 through 24. These numbers correspond to permutations in the order given below. Read the list below row-by-row, left-to-right. So vertex 1 corresponds to $(1, 2, 3, 4)$, vertex 7 to $(2, 1, 3, 4)$, vertex 15 to $(3, 2, 1, 4)$, etc.

```
{1, 2, 3, 4}, {1, 2, 4, 3}, {1, 3, 2, 4}, {1, 3, 4, 2},
{1, 4, 2, 3}, {1, 4, 3, 2}, {2, 1, 3, 4}, {2, 1, 4, 3},
{2, 3, 1, 4}, {2, 3, 4, 1}, {2, 4, 1, 3}, {2, 4, 3, 1},
{3, 1, 2, 4}, {3, 1, 4, 2}, {3, 2, 1, 4}, {3, 2, 4, 1},
{3, 4, 1, 2}, {3, 4, 2, 1}, {4, 1, 2, 3}, {4, 1, 3, 2},
{4, 2, 1, 3}, {4, 2, 3, 1}, {4, 3, 1, 2}, {4, 3, 2, 1}
```

Each vertex in the graph has degree 3, corresponding to each of the 3 (non-trivial) prefix reversals possible on a 4-permutation. Note that reversing prefixes of length 1 corresponds to an edge from a vertex to itself; these are not relevant to us and so we ignore these.

So to compute the minimum number of prefix reversals needed to sort an input n-permutation P, we simply perform a *breadth first search* in G_n with source P, to find the shortest distance in this graph between P and the completely sorted sequence $(1, 2, \ldots, n)$. Breadth first search is, in general, known to be a very efficient algorithm. Specifically, on a graph $G = (V, E)$, the worst case running time of breadth first search is $\Theta(|V| + |E|)$. The problem with our problem is not the breadth first search algorithm, but the size of the graph G_n. You can easily verify that G_n has $n!$ vertices and $(n-1)n!/2$ edges. These quantities are functions of n that grow extremely rapidly. For example, 15! is 1307674368000 and so, if you use this approach to solve the minimum prefix reversals problem for a 15-permutation, in the worst case you might examine more than a trillion vertices!

The example below shows the breadth first search tree of G_4 rooted at permutation $(2, 4, 1, 3)$ (numbered 11 in our order). It is clear from this picture that getting to permutation $(1, 2, 3, 4)$ from $(2, 4, 1, 3)$ requires 4 hops, implying that the minimum number of pancake flips needed to sort $(2, 4, 1, 3)$ is 4.



Breadth first search may be inefficient, but given the seemingly, inherent intractability of the problem, we may not have much of a choice. One approach to improving this solution is to make breadth first search somewhat "smarter" and get it to avoid examining large chunks of the graph during its search. For any permutation P, let $M(P)$ denote the minimum number of prefix reversals needed to sort P. Now suppose that for any permutation P, we are able to quickly compute $L(P)$ and $U(P)$, which are lower and upper bounds respectively, on $M(P)$. In other words, $L(P) \leq M(P) \leq U(P)$. Being able to compute $L(P)$ and $M(P)$ can help in the following way. Suppose that the current vertex being processed by breadth first search is some permutation Q. The algorithm then examines neighbors of Q and suppose that R and S are two neighbors such that $U(R) < L(S)$. This immediately implies that $M(R) < M(S)$ and this means that it is a waste of time to examine vertex S and its descendents in the breadth-first search tree. In this manner, we've avoided looking at a portion of the graph. Depending on how good the bounds $L()$ and $U()$ are, we might be able to avoid looking at large portions of the graph, thus speeding up our search.

Various other heuristics that prune portions of the graph are possible. For example, suppose we are asked to find the minimum number of prefix reversals to sort an n-permutation P whose

last k elements are $n-k+1, n-k+2, \ldots, n$. Then, there is no reason to ever move the last k elements and therefore our search can focus on the $(n-k)!$ permutations whose last k elements are fixed to be $n-k+1, n-k+2, \ldots, n$. How fast your program runs will depend, to a large extent, on how cleverly you can prune the graph or permutations.

Given how large the graph is, you may also have to deal with the problem of running out of memory. The typical implementation of the breadth-first search algorithm uses two data structures to help organize the search: (i) a set of vertices already visited and (ii) a queue of vertices being processed. It is possible, even likely, that after a point in the search these data structures will not fit in main memory. The solution to this problem is to save these data structures in secondary memory, once they get sufficiently large. However, accessing secondary memory is orders of magnitude slower than accessing main memory and so you'll have to design your program and data structures so that secondary memory is accessed only when absolutely necessary.

See the next page for specific details of the assignment and what you must turn in.

Program Specification
The input to you program will be an n-permutation. Your program should first read integer n, followed by n addtional integers representing the permutation. You should read all numbers from standard input. Don't worry about error checking, and don't both bother producing prompts.

It should run in two modes: an "exact answer" mode and an "approximate answer" mode. In the exact mode, for a given input permutation P, your program should output $M(P)$, the minimum number of prefix reversals needed to sort P. It should also produce as output a sequence of $M(P)$ prefix reversals that sort P. In the approximate mode, your program should produce a sequence of prefix reversals that sort P, but this sequence need not be of minimum length.

Your program may be written in any high-level language (`C`, `C++`, `Java`, etc.).

What to turn in
You should electronically submit: source code for your program, a `README` file, an executable that runs on Linux, and the write-up described below. Directions for using the "submit" command are included at the end of this document.

You must also turn in hardcopy of the write-up and source code.

The `README` file should contain instructions on how to compile your program and should also contain a list of known bugs.

The write-up should be approximately 2 pages, describing your experience with the program. Specifically, you should address the following issues.

(a) What heuristics you used to prune the graph and how helpful these were in improving the speed of the search and the memory required.

(b) How you organized your data structures and what techniques and tricks you used to get around the main memory bottleneck.

(c) How well your program performs when run in the approximate mode. In particular, how much longer is the sequence of prefix reversals produced by your "approximate" algorithm in the worst case, as compared to the optimal solution?

Grading
Most of your grade for this assignment will depend on three aspects of your submission:

(a) Correctness of your program.

(b) How fast your program runs in the exact mode.

(c) How good the solution produced in the approximate mode is.

(d) Your understanding of the issues involved, as shown by your write-up.

In the exact mode, your program will be tested on permutations of sizes in the range 10 through 15, while in the approximate mode, your program will be tested on permutations of sizes in the range 1000 through 1500.

Contest! To get extra-credit (and possibly, wonderful prizes), you can enter your program in a competition that will run at the end of semester to determine the best program. Details of this competition will be announced later.
Programming projects always take much longer than anticipated, so start early.

Using the 'submit' command Use the 'submit' command to submit source code, executable, README, and write-up. Put all the files for submission into one directory, e.g. 'hw9files'. To

submit the directory, execute 'submit hw9files' on one of the CS department's Unix workstations. The program will then ask you the course for which you are submitting. Respond with 'c044'. Then it will request a submission directory; you will be prompted for its name via "Choice:". Respond with "hw9".