Order and the Lack Thereof

Nick Haliday

2012-04-24

1 Intro

The topics for this lecture are all focused on the ordering of elements in arrays. There's much more to this than sorting! We will cover fast kth element queries (order statistics), inversion counting and other problems involving inversions, and generating permutations.

Any permutation can be decomposed into some sequence of transpositions, swaps of two not necessarily adjacent elements.

2 Inversions

An inversion of a permutation σ is a pair (i, j), with $i < j$ and $\sigma(i) > \sigma(j)$. The number of inversions in a permutation, which we call $\text{inv}(\sigma)$, measures how "out-of-order" a permutation is. We can compute $\text{inv}(\sigma)$ in $O(n \log n)$ with any number of methods: an augmented comparison sort or a binary indexed tree for example. A BIT is likely the easiest way to code it.

A useful notation for permutations is cycle notation. (1 2 4) denotes the cycle that sends 1 to 2, 2 to 4, and 4 to 1. Any permutation can be expressed uniquely as the composition of disjoint cycles. These cycles can be broken down further into transpositions (swaps of two elements), and any transposition can be written as a product of adjacant transpositions (swaps of adjacent elements). $\text{inv}(\sigma)$ counts the minimal number of adjacent transpositions needed to produce a permutations. Thus it also counts the minimal number of adjacent transpositions needed to sort an array.

3 Order Statistics

Order statistics tell us the kth largest element in a (possibly unsorted) array. There is a rather complicated worst-case $O(n)$ algorithm that also has a large constant (the median-of-medians algorithm). Instead we'll look at a simple, expected $O(n)$ time algorithm modeled after quicksort.

The basic idea is: randomly choose a pivot, partition the algorithm in linear time based on that pivot k, then see if our partition's order statistic is k, $< k$, or $> k$ and recurse in the last two cases. The first two parts are exactly the same as what you do in a randomized quicksort. Recursing on only one part of the array keeps the expected runtime linear (the details are little messy but you don't have to do anything clever to prove this). Quickselect is available as `nth_element` in `algorithm` in C++.

```
1  // RAI = Random Access Iterator, like vector iterator or pointer to array
2  // templates allow this to work w/ arrays and iterators of any type
3  template <typename RAI>
4  RAI partition(RAI first, RAI pivot, RAI last) {
5      swap(*pivot, *(last - 1));
6      pivot = last - 1;
7      RAI store = first;
```

```
    while (first < pivot) {
        if (*first < *pivot) {
            swap(*store++, *first);
        }
        ++first;
    }
    swap(*store, *pivot);
    return store;
}

template <typename RAI>
RAI random_partition(RAI first, RAI last) {
    if (first == last) return first;
    RAI pivot = first + rand() % (last - first);
    return partition(first, pivot, last);
}

template <typename RAI>
void quickselect(RAI first, RAI nth, RAI last) {
    RAI pivot;
    do {
        pivot = random_partition(first, last);
        if (nth < pivot)
            last = pivot;
        else if (nth > pivot)
            first = pivot + 1;
    } while (pivot != nth);
}
```

4 Generating Things

A useful thing to know how to do is generate all permutations and combinations. Here's some relatively straightforward code from Python s itertools page to generate all r-length permutations (this basically just does what you would do with nested loops or recursion):

```
def combinations(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = range(r)
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:<D-j>
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
```

```
        yield tuple(pool[i] for i in indices)
```

Try modifying this to allow combinations with replacement.

An algorithm for generating the lexicographically next permutation is pretty simple:

1. Find the largest index k such that $a[k] < a[k+1]$. If no such index exists, the permutation is the last permutation.
2. Find the largest index l such that $a[k] < a[l]$. Since $k+1$ is such an index, l is well defined and satisfies $k < l$.
3. Swap $a[k]$ with $a[l]$.
4. Reverse the sequence from $a[k+1]$ up to and including the final element $a[n]$

This gives an average runtime of $O(1)$.

The following code for r-permutations on the other hand is pretty subtle:

```
def permutations(iterable, r=None):
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return
    indices = range(n)
    cycles = range(n, n-r, -1)
    yield tuple(pool[i] for i in indices[:r])
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
```

5 Problems

1. The Lehmer code or inversion vector of a permutation σ is defined by $I[i] = |\{j > i : \sigma(j) < \sigma(i)\}|$. So $\text{inv}(\sigma)$ is the sum of I. Give an algorithm for computing σ given I.
2. The factoradic number system gives place value $i!$ to the ith value from the right starting at zero. Consider factoradic numbering applied to a Lehmer code. What does the resulting number mean?
3. (USACO, FEB11S) FJ has N ($1 \leq N \leq 20$) cows conveniently numbered from 1 to N that like to line up in different orders. He wants to answer K queries ($1 \leq K \leq 10,000$) of two types. One gives a permutations of the N cows and asks for its position in the lexicographic order (ordered like strings) of all permutations. The other asks for the permutation given the position.

4. (USACO, DEC11G/S) FJ has N ($1 \leq N \leq 20,000$) cows. Each cow has a unique ID number. They were originally lined up in an unknown order A. Five times FJ tried to take a photo, and 5 times some group of cows backed out of the line and reinserted themselves at some different position, with the other cows sliding over to replace them. Each cow only moves in zero or one of the photos. Find the original sequence A given the 5 other sequences.

5. (USACO, NOV10G) Given some sequence cows numbered from 1 to N ($1 \leq N \leq 100,000$), find the minimal number of swaps of adjacent cows to convert them to a sequence of the form $k, k+1, k+2, \ldots, N, 1, 2, \ldots, k-1$ (including the sequence $1, 2, 3, \ldots, N$.

6. (USACO, NOV11G) FJ has lined up his N ($1 \leq N \leq 100,000$) cows and knows their heights H_i. Find all contiguous subsequences with median height greater than or equal to a given threshold X. The median is defined as the $\lceil \frac{N}{2} \rceil$ height in the sorted subsequence.

7. (CLRS) Let $X[1..n]$ and $Y[1..n]$ be two arrays, each containing n numbers already in sorted order. Give an $O(\log n)$-time algorithm to find the median of all $2n$ elements in arrays X and Y.