Code Jam 2008 - Round 1C

Analysis: Increasing Speed Limits

This was the hard problem for round 1C, with 398 people solving the small i/o set and 49 people solving the large i/o set during the contest.

The problem asks us to count the number of strictly increasing subsequences of the original sequence. This lends itself nicely to a dynamic programming solution for the small input.

Let f(x) be the number of strictly increasing subsequences that start with node x and S[x] be the value of the sequence at position x. At any node x we can either end the subsequence at x or connect it to any of the following nodes that are strictly greater than node x. Therefore, you could do something like this to find out how many strictly increasing subsequences start at each index.

```
f(x) = 1
for i = n to 1
  f(i) = 1
  for j = i + 1 to n
    if S[i] < S[j]
      f(i) = f(i) + f(j)
```

After doing that, solving the original problem is just a matter of summing the number of ways you make a strictly increasing subsequence starting at each position.

The key to solving the large i/o set is to realize we can make the inner loop run much faster. What we really want to do is sum all of the previous f(j)s for which S[j] > S[i]. This is a classic tree problem. There are a number of tree data structures you could use including a binary indexed tree, a segment tree, or a binary search tree, to name a few, that will allow you to solve the problem in O(n log n) time per case.

The first two types of trees are fairly easy to implement although they add a complication since they use an amount of memory proportional to the maximum value in the sequence. Fortunately this can be worked around by normalizing S to contain only values between 0 and n-1 without changing the the S[i] < S[j] property for any i and j. This can be done by transforming S[i] to be the first occurrence of S[i] in a sorted list of S.

Here is an implementation of these ideas in C++.

```
#define MAXN (1<<20)
int sum_bit[MAXN];

int sum_bit_get(int x)
{
  int ret = 0;
  for(int i = x | MAXN; i < 2 * MAXN; i += i & -i)
    ret = (ret + sum_bit[i ^ MAXN]) % 1000000007;
  return ret;
}

void sum_bit_add(int x, int v)
{
```

```
  for(int i = x | MAXN; i; i &= i - 1)
    sum_bit[i ^ MAXN] = (sum_bit[i ^ MAXN] + v) % 1000000007;
}

int S[1000000];
int S2[1000000];

int main()
{
  int T; cin >> T;
  long long X, Y, Z, A[100];
  for(int t = 1; t <= T; t++) {
    int n, m; cin >> n >> m >> X >> Y >> Z;
    for(int i = 0; i < m; i++) cin >> A[i];

    // Generate S
    for(int i = 0; i < n; i++) {
      S[i] = A[i % m];
      A[i % m] = (X * A[i % m] + Y * (i + 1)) % Z;
    }

    // Normalize S
    memcpy(S2, S, sizeof(S));
    sort(S2, S2+n);
    for(int i = 0; i < n; i++)
      S[i] = lower_bound(S2, S2+n, S[i]) - S2;

    // Calculate f(i) and sum them in to result.
    int result = 0;
    memset(sum_bit, 0, sizeof(sum_bit));
    for(int i = n - 1; i >= 0; i--) {
      int add = 1 + sum_bit_get(S[i] + 1);
      sum_bit_add(S[i], add);
      result = (result + add) % 1000000007;
    }

    cout << "Case #" << t << ": " << result << endl;
  }
  return 0;
}
```

In addition to trees, there was a "hacker's" O(n √n) solution. The idea behind it is similar to the idea mentioned in solution 1 of the Mousetrap editorial (from Round 1B) where, in addition to maintaining the value at each position, we maintain sums for ranges of length √N. These tables can be maintained in constant time and take √n time to query.

More Information:
Binary Indexed Tree