Lecture 10

More DP Review

Exercises:

1. Let $F(n)$ denote the nth Fibonacci number. Compute $F(1000000)$ modulo 1000003.
2. Consider strings using only the letters a, b, c. Count the number of such strings that do not have the substring bba of length 35 (result fits in a 64-bit signed integer).
3. Compute the number of distinct binary search trees using the numbers $\{1, 2, \ldots, n\}$.

Solutions:

1. Firstly we show the DP solution:

```
int[] dp = new int[1000001];
dp[0] = 0; dp[1] = 1;
for (int i = 2; i < dp.length; ++i) dp[i] =
    (dp[i-1]+dp[i-2])%1000003;
```

As we only depend on the last 2 entries, we can solve this without a table:

```
int a = 0, b = 1;
for (int i = 2; i <= 1000000; ++i)
{
  int t = b;
  b = (a + b) % 1000003;
  a = t;
}
```

Finally, note that

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} F(n+1) \\ F(n) \end{pmatrix}.$$

Thus we have the equation

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} F(n+1) \\ F(n) \end{pmatrix}.$$

We can exponentiate matrices in time $\lg(n)$ giving our quickest algorithm.

2. First we give a memoized solution:

```
long count(int n, int bba)
{
  if (n==0) return 1;
  if (cache[bba][n] > -1) return cache[bba][n];
  long ret = count(n-1,0); //letter c
  ret += count(n-1,Math.min(bba+1,2)); //letter b
  if (bba < 2) ret += count(n-1,0); //letter a
  return cache[bba][n] = ret;
}
```

This could be looked at as performing DP on the states of a DFA. We could have also solved this using matrix exponentiation.

3. Firstly, we solve this using memoization:

```
long count(int n)
{
  if (n == 0) return 1;
  if (cache[n]>-1) return cache[n];
  long ret = 0;
  for (int root = 1; root <= n; ++root)
     ret += count(root-1)*count(n-root);
  return cache[n] = ret;
}
```

Interestingly, the answer is just $\frac{1}{n+1}\binom{2n}{n}$, the nth Catalan number.

Longest Increasing Subsequence

Earlier we saw a fairly easy dynamic programming/memoization algorithm for finding the longest increasing subsequence in $O(n^2)$ time. There is a more complicated DP algorithm that solves this problem in $O(n \lg m)$ time, where m is the length of the longest increasing subsequence. To do this, we maintain an array `int[] len2lastelt`. After processing indices in `[0..k]` the entry `len2lastelt[i]` will store the lowest possible element that ends an increasing subsequence of length i. The key observation is that this array will always be in increasing order. Thus, when we process element $k+1$, we can binary search our dp array to find largest entry it is greater than. This identifies exactly how to update `len2lastelt`.

Exercises:

1. Go through each step of the longest increasing subsequence algorithm on the list $3, 1, 2, 0, 4, 3, 4$. In addition to maintaining len2lastelt, also keeping track of the length of the longest increasing subsequence that ends with each element of the list.

2. Show how to reconstruct a longest increasing subsequence using the length array from the previous part.

3. Given two arrays **int**[] arr1, arr2, show how to compute the longest common subsequence. That is, the longest subsequence of arr1 that is also a subsequence of arr2.

4. You are given a starting string x and a target string y. Assuming you can change a letter, insert a letter, or delete a letter, each with a cost of one, compute the minimum cost to change x into y.

Solutions:

1. Below we show len2lastelt after processing each element, and the length for each element.

		Indices of len2lastelt				
k	lst [k]	1	2	3	4	5
0	3	3	X	X	X	X
1	1	1	X	X	X	X
2	2	1	2	X	X	X
3	0	0	2	X	X	X
4	4	0	2	4	X	X
5	3	0	2	3	X	X
6	4	0	2	3	4	X

k	0	1	2	3	4	5	6
lst [k]	3	1	2	0	4	3	4
len [k]	1	1	2	1	3	3	4

2. We process len from the previous part backwards looking for the largest length, and then the next largest, and so forth, always checking that we are producing an increasing sequence.

3. We solve the subproblem of computing the largest common subsequence of the suffixes $[p1, L1]$ and $[p2, L2]$ where $L1, L2$ are the lengths of arr1,arr2, respectively. The runtime is $O(n^2)$.

```
static int lcs(int[] arr1, int[] arr2, int p1, int p2)
{
  if (p1 == arr1.length || p2 == arr2.length) return 0;
  if (cache[p1][p2] > -1) return cache[p1][p2];
  int ret =
     Math.max(lcs(arr1,arr2,p1+1,p2),lcs(arr1,arr2,p1,p2+1));
  if (arr1[p1]==arr2[p2]) ret = 1+lcs(arr1,arr2,p1+1,p2+1);
  return cache[p1][p2] = ret;
}
static int lcs(int[] arr1, int[] arr2) { return
   lcs(arr1,arr2,0,0); }
static void printlcs(int[] arr1, int[] arr2, int p1, int p2)
{
    if (p1 == arr1.length || p2 == arr2.length) return;
    int val = lcs(arr1,arr2,p1,p2);
```

```
        if (val == lcs(arr1,arr2,p1+1,p2))
            printlcs(arr1,arr2,p1+1,p2);
        else if (val == lcs(arr1,arr2,p1,p2+1))
            printlcs(arr1,arr2,p1,p2+1);
        else
        {
            System.out.print(arr1[p1]+" ");
            printlcs(arr1,arr2,p1+1,p2+1);
        }
}
static void printlcs(int[] arr1, int[] arr2) {
    printlcs(arr1,arr2,0,0); }
```

4. We solve this using memoization:

```
static int cost(String x, String y, int xpos, int ypos)
{
  if (xpos == x.length() || ypos==y.length()) return
     Math.abs(x.length()-y.length());
  if (cache[xpos][ypos] > -1) return cache[xpos][ypos];
  int cost = cost(x,y,xpos+1,ypos) + 1;
  cost = Math.min(cost, cost(x,y,xpos,ypos+1) + 1);
  int match = x.charAt(xpos)==y.charAt(ypos) ? 0 : 1;
  cost = Math.min(cost,cost(x,y,xpos+1,ypos+1)+match);
  return cache[xpos][ypos] = cost;
}
static int cost(String x, String y) { return cost(x,y,0,0); }
```

The runtime is $O(mn)$ where m, n are the two lengths.

Some Other Applications of Dynamic Programming

When studying segment trees, we had a fixed-size list whose entries were updated, and we wanted to repeatedly query for the maximum or minimum value in a range. Suppose that the entries were fixed as well. Is there some precomputation we can do to speed up range queries? With a trivial $O(n^3)$ brute force algorithm (using $O(n^2)$ space), we can precompute all possible range queries. This can be improved to $O(n^2)$ by using a straightforward dynamic programming approach. By using a slicker dynamic programming method, we can improve the runtime and memory usage to $O(n \lg n)$. The idea is pretty straight forward. For each index k and each length $L = 2^k$ we compute the maximum of the range begining in position k of length L. Using dynamic programming, and because we only use lengths that are powers of two, this requires $O(n \lg n)$ time. We can then answer range queries in constant time.

Another interesting application of dynamic programming is counting tilings. For example, suppose we have a $6 \times n$ grid of squares, and we want to count the number of ways of tiling it with dominoes modulo 1000003. To make this amenable to dynamic programming, consider the following subproblem: compute the number of tilings of a $6 \times k$ grid where some subset S of the cells in the first 2 columns are occupied. The number of states is thus $n2^{12}$ as there are n possible values for k, and 2^{12} possible values for S.

Exercises:

1. Show how to build the table for static maximum range queries in $O(n \lg n)$ time, and then show how to answer queries in $O(1)$ time.
2. Give code to solve the tiling problem above.
3. You are given 12 points in the plane with coordinates $(x_0, y_0), \ldots, (x_{11}, y_{11})$. Compute the shortest route (in Euclidean distance) beginning at (x_0, y_0) that visits each of the 12 points exactly once.

Solutions:

1. The code follows (could be improved slightly using Math.getExponent). We have just returned the max value, but could have returned the index of the max value with little extra code.

```
int[][] buildTable(int[] arr)
{
  int n = arr.length, m =
     (int)(Math.log(n)/Math.log(2)+1+1e-9);
  int[][] tab = new int[n][m];
  for (int i = 0; i < n; ++i) tab[i][0] = arr[i];
  for (int j = 1, L = 2; L <= arr.length; L<<=1, ++j)
    for (int a = 0; a+L-1 < arr.length; ++a)
      tab[a][j] = Math.max(tab[a][j-1],tab[a+L/2][j-1]);
  return tab;
}
//Query for indices in interval [a,b]
int maxQuery(int[][] tab, int a, int b)
{
  int L = b-a+1, lgL =
     (int)Math.floor(Math.log(L)/Math.log(2)+1e-9);
  return Math.max(tab[a][lgL],tab[b+1-(1<<lgL)][lgL]);
}
```

2. The code follows:

```
//Bits of S in column-major order
static int count(int k, int S)
{
  if (k == 0) return S==0?1:0;
```

```java
    if ((S & 0x3F) == 0x3F) return count(k-1,S>>6);
    if (cache[k][S] > -1) return cache[k][S];
    int lowOff = (~S)&(S+1), below = lowOff<<1,
        right = lowOff<<6;
    int ret = 0;
    if ((S & below)==0 && below < (1<<6))
      ret = (ret + count(k,S|lowOff|below))%1000003;
    if ((S & right)==0)
      ret = (ret + count(k,S|lowOff|right))%1000003;
    return cache[k][S] = ret;
  }
  static int count(int k) { return count(k,0); }
```

Instead of bit twiddling, we could have looped looking for the lowest off bit. The runtime is $O(n2^{10})$.

3. We use memoization:

```java
  static double dist(double[] p1, double[] p2)
  {
    return
       sqrt((p1[0]-p2[0])*(p1[0]-p2[0])+(p1[1]-p2[1])*(p1[1]-p2[1]));
  }
  //Assuming we have a path from 0 to last using vertices in S,
     compute
  //the cost of completing the path
  static double solve(double[][] ps, int S, int last)
  {
    if (S == (1<<ps.length)-1) return 0;
    if (cache[S][last] > -1) return cache[S][last];
    double min = Double.POSITIVE_INFINITY;
    for (int next = 0; next < ps.length; ++next)
    {
      int b = 1<<next;
      if ( (b & S) != 0 ) continue;
      min = Math.min(min,
          solve(ps,S^b,next)+dist(ps[last],ps[next]));
    }
    return cache[S][last] = min;
  }
  static double solve(double[][] ps) { return solve(ps,1,0); }
```

There are $O(n2^n)$ states with $O(n)$ runtime per state giving $O(n^2 2^n)$.