Lecture 9

More DP Review

Exercises:

1. Compute the number of distinct binary search trees using the numbers \{1, 2, \ldots, n\}.

2. Suppose you are given a list of widths and heights of blocks that each has unit depth. You can put one block on top of another exactly when the top block has width and height at most as large as the block below it. What is the height of the tallest tower of blocks that can be made?

Solutions:

1. 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 \( n \)th Catalan number.

2. Sort the list of blocks to be ascending in width, and break ties using ascending heights. Run LIS on the heights of the sorted list.

Probability Preview: Dynamic Programming

As dynamic programming often allows us to solve interesting counting problems, it can help us with many discrete probability problems. For instance, in a situation where every outcome is equally likely, the probability is simply

\[
\frac{\text{desired outcomes}}{\text{total possible outcomes}}.
\]
Exercises:

1. Given a perfectly shuffled standard 52-card deck, you deal 5-card hands to \( n \) players \((1 \leq n \leq 10)\). What is the probability that at least one of the players has a flush (all 5 cards in their hand have the same suit)? Could you give a good approximate answer using only a calculator?

Solutions:

1. Firstly we show the DP solution. This counts the number of sequences of suits yielding at least one flush, and the total number of possible dealings. The result is the quotient. We use doubles since the numbers can get very large.

\[
14^4 \times 4 \times 6 = 921984 \text{ entries}
\]

```java
static double[] cache[] = new double[14][14][14][4][6]; // fill with -100

// Counts number of ordered dealings with at least 1 flush with 4 kinds of cards
static double solve(int[] cnts, int suit, int run, int cards, int totCards) {
    if (cards == totCards) return run == 5 ? 1 : 0;
    double v = cache[cnts[0]][cnts[1]][cnts[2]][cnts[3]][suit][run];
    if (v > -1) return v;
    double sum = 0;
    for (int i = 0; i < 4; ++i) {
        if (cnts[i] == 0) continue;
        int nRun = 0;
        if (run == 5) nRun = 5;
        else if (run == (cards % 5) && (run == 0 || suit == i)) nRun = run + 1;
        double c = cnts[i];
        cnts[i] -= c;
        sum += c * solve(cnts, i, nRun, cards + 1, totCards);
        cnts[i] += c;
    }
    cache[cnts[0]][cnts[1]][cnts[2]][cnts[3]][suit][run] = sum;
    return sum;
}
```

```java
static double[] tcache[] = new double[14][14][14][14]; // Fill with -100

// Total number of ordered dealings with 4 kinds of cards
static double total(int[] cnts, int cards, int totCards) {
    // Code here...
}
```

2
if (cards == totCards) return 1;

double v = tcache[cnts[0]][cnts[1]][cnts[2]][cnts[3]];
if (v > -1) return v;

double sum = 0;
for (int i = 0; i < 4; ++i)
{
    if (cnts[i] == 0) continue;
    double c = cnts[i];
    cnts[i]--;
    sum += c*total(cnts, cards+1, totCards);
    cnts[i]++;
}
tcache[cnts[0]][cnts[1]][cnts[2]][cnts[3]] = sum;
return sum;

We could also use a Monte Carlo approach. The program below estimates a 1 standard
deviation error-bar around the result.

import java.util.*;
public class PokerMonte
{
    static boolean hasFlush(ArrayList<Integer> list, int n)
    {
        for (int i = 0; i < n; ++i)
        {
            boolean hasRun = true;
            for (int j = 1; j < 5; ++j)
            {
                if (list.get(5*i+j) != list.get(5*i))
                    hasRun = false;
            }
            if (hasRun) return true;
        }
        return false;
    }

    public static void main(String[] args)
    {
        int N = Integer.parseInt(args[0]), n =
            Integer.parseInt(args[1]), cnt = 0;
        ArrayList<Integer> list = new ArrayList<Integer>();
        for (int i = 0; i < 13; ++i) for (int j = 0; j < 4; ++j)
            list.add(j);
        for (int i = 0; i < N; ++i)
        {
To try and estimate this probability by hand, we could add the (invalid) assumption that the probability that each hand is a flush is independent (i.e., drawn from different shuffled decks). The probability of a single deck getting a flush is

\[
\frac{4 \left( \frac{13}{5} \right)}{\left( \frac{52}{5} \right)}.
\]

Assuming independence, the required probability is

\[
1 - \left( 1 - \frac{4 \left( \frac{13}{5} \right)}{\left( \frac{52}{5} \right)} \right)^n,
\]

a pretty good approximation. For \( n = 10 \) the correct answer is

0.01958222694330079

while the approximation gives

0.0196323.

**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 \( O(n \log n) \). 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 \( O(n \log n) \) time. We can then answer range queries in constant time.

**Exercises:**

1. Show how to build the table for static maximum range queries in \( O(n \log n) \) time, and then show how to answer queries in \( O(1) \) time.
2. Suppose you have a $6 \times n$ grid of squares. Give an algorithm to count the number of ways to tile it with dominoes modulo 1000003.

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.

4. 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).

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.

```java
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:

```java
// Bits of S in column-major order
static int count(int k, int S)
{
    if (k == 0) return S==0?1:0;
    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 $O(n^{2^{12}})$.

3. We use memoization:

    static double dist(double[] p1, double[] p2)
    {
    }
    //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 $O(n^22^n)$.

4. 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.