LEC16: INTRODUCTION TO DYNAMIC PROGRAMMING

CSE 373 Analysis of Algorithms
Fall 2016
Instructor: Prof. Sael Lee

Lecture slide courtesy of Prof. Steven Skiena
Dynamic programming is a very powerful, general tool for solving optimization problems on left-right-ordered items such as character strings.

Once understood it is relatively easy to apply, it looks like magic until you have seen enough examples.

Floyd’s all-pairs shortest-path algorithm was an example of dynamic programming.
Greedy algorithms focus on making the best local choice at each decision point. In the absence of a correctness proof such greedy algorithms are very likely to fail.

Dynamic programming gives us a way to design custom algorithms which systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid recomputing (thus providing efficiency).
A recurrence relation is an equation which is defined in terms of itself.

They are useful because many natural functions are easily expressed as recurrences:

- Polynomials: \( a_n = a_{n-1} + 1; \ a_1 = 1 \rightarrow a_n = n \)
- Exponentials: \( a_n = 2a_{n-1}; \ a_1 = 2 \rightarrow a_n = 2^n \)
- Weird: \( a_n = na_{n-1}; \ a_1 = 1 \rightarrow a_n = n! \)

Computer programs can easily evaluate the value of a given recurrence even without the existence of a nice closed form.
$F_n = F_{n-1} + F_{n-2}; F_0 = 0; F_1 = 1$

- Implementing this as a recursive procedure is easy, but slow because we keep calculating the same value over and over.
\[ \frac{F_{n+1}}{F_n} \approx \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61803 \]

Thus \( F_n \approx 1.6^n \).

Since our recursion tree has 0 and 1 as leaves, computing \( F_n \) requires \( \approx 1.6^n \) calls!
WHAT ABOUT DYNAMIC PROGRAMMING?

We can calculate $F_n$ in linear time by storing small values:

+ $F_0 = 0$
+ $F_1 = 1$
+ for $i = 1$ to $n$
  
  \[ F_i = F_{i-1} + F_{i-2} \]

Moral: we traded space for time.
BENEFITS OF DYNAMIC PROGRAMMING

- Dynamic programming is a technique for efficiently computing recurrences by storing partial results.
- Once you understand dynamic programming, it is usually easier to reinvent certain algorithms than try to look them up!
- Dynamic programming to be one of the most useful algorithmic techniques in practice:
  - Morphing in computer graphics.
  - Data compression for high density bar codes.
  - Designing genes to avoid or contain specified patterns.
The trick to dynamic program is to see that the naive recursive algorithm repeatedly computes the same subproblems over and over and over again.

If so, storing the answers to them in a table instead of recomputing can lead to an efficient algorithm.

Thus we must first hunt for a correct recursive algorithm – later we can worry about speeding it up by using a results matrix.
The most important class of counting numbers are the binomial coefficients, where $\binom{n}{k}$ counts the number of ways to choose $k$ things out of $n$ possibilities.

**Committees** – How many ways are there to form a $k$-member committee from $n$ people? By definition, $\binom{n}{k}$.

**Paths Across a Grid** – How many ways are there to travel from the upper-left corner of an $n \times m$ grid to the lower-right corner by walking only down and to the right? Every path must consist of $n + m$ steps, $n$ downward and $m$ to the right, so there are $\binom{n+m}{n}$ such sets/paths.
Since \( \binom{n}{k} = \frac{n!}{(n - k)! \, k!} \), in principle you can compute them straight from factorials.

However, intermediate calculations can easily cause arithmetic overflow even when the final coefficient fits comfortably within an integer.
In Pascal’s Triangle, each number is the sum of the two numbers directly above it:

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
A more stable way to compute binomial coefficients is using the recurrence relation implicit in the construction of Pascal’s triangle, namely, that

\[
\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}
\]

It works because the nth element either appears or does not appear in one of the \( \binom{n}{k} \) subsets of k elements.
BASIS CASE

- No recurrence is complete without basis cases.
- How many ways are there to choose 0 things from a set?
  - Exactly one, the empty set.
- The right term of the sum drives us up to \( \binom{k}{k} \). How many ways are there to choose \( k \) things from a \( k \)-element set?
  - Exactly one, the complete set.
long binomial coefficient(n,m)
int n,m;  /* compute n choose m */
{
    int i,j;  /* counters */
    long bc[MAXN][MAXN]; /* table of binomial coefficients */
    for (i=0; i<=n; i++) bc[i][0] = 1;
    for (j=0; j<=n; j++) bc[j][j] = 1;
    for (i=1; i<=n; i++)
        for (j=1; j<i; j++)
            bc[i][j] = bc[i-1][j-1] + bc[i-1][j];
    return( bc[n][m] );
}
THREE STEPS TO DYNAMIC PROGRAMMING

1. Formulate the answer as a recurrence relation or recursive algorithm.
2. Show that the number of different instances of your recurrence is bounded by a polynomial.
3. Specify an order of evaluation for the recurrence so you always have what you need.