Dynamic Programming

- Advantage of typical recursive algorithm is to divide domain into independent subproblems
- Some recursive problems do not divide into independent subproblems
- Use dynamic programming or memo-ization

Dynamic Programming

- Motivation
  - Eliminate costly recomputation in any recursive program, given space to store values of the function for arguments smaller than the call
  - Dynamic programming reduces the running time of a recursive function to be ≤ time required to evaluate the function for all arguments ≤ the given argument, treating the cost of a recursive call as a constant
Fibonacci Sequence

Definition
- \( F(0) = 0; F(1) = 1 \)
- \( F(n) = F(n-1) + F(n-2) \)

Find \( F(3) \)
- \( F(3) = F(2) + F(1); F(2) = F(1) + F(0) \)
  \[ \Rightarrow \]
- \( F(3) = (F(1) + F(0)) + F(1) = (1 + 0) + 1 \)

Recursive Implementation

```c
int FindFib(int i)
{
    if (i < 1) return 0;
    if (i == 1) return 1;
    return FindFib(i-1) + FindFib(i-2);
}
```

- Spectacularly inefficient recursive algorithm
- That is, the recursive algorithm is exponential (\( O(1.618^n) \))

Dynamic Programming

```c
int FindFibBU(int i)
{
    int F[maxN];
    F[0] = 0;
    F[1] = 1;
    for (k = 2; k <= i; k++)
        F[k] = F[k-1] + F[k-2];
    return F[i];
}
```

- Evaluate any function by
  - start at smallest function value
  - use previously computed values at each step to compute current value
- Must save all previously computed values
  - note that the values in F[] grow exponentially

Simple technique has converted exponential algorithm (\( O(1.618^n) \)) to linear (\( O(n) \))
Memo-ization

Fibonacci Sequence

```c
int FindFibTD(int i)

int knownF[maxN];

knownF[0] = 0;
knownF[1] = 1;

for (int j=2; j<maxN; j++)
    knownF[j] = unknown
return fibHelp(i, knownF);
```

```c
int fibHelp(int i, int knownF[])

if (knownF[i] == unknown)
    knownF[i]= fibHelp(i-1,knownF)
    + fibHelp(i-2,knownF)
return knownF[i];
```

- Same (or less) cost as bottom-up DP
- **FindFibTD**
  - initializations “scratch pad”
  - base cases
  - rest unknown
- **fibHelp**
  - Do I already know answer?
  - If not, calculate it, save it
  - return the answer
- Only calculates each needed answer once

"Same (or less) cost as bottom-up DP"

Binomial Coefficient

Def'n:

\[
\binom{n}{m} = \frac{n!}{m!(n-m)!}
\]

where \( n \) and \( m \) are non-negative integers

Binomial Coefficient

Naïve Approach

- Solve for \( n! \) recursively
  - \( \text{fact}(n) = n \times \text{fact}(n-1) \)
- Solve for \( (n-m)! \) recursively
- Solve for \( m! \) recursively

- but, \( 13! \) cannot be represented by a 32-bit integer
Binomial Coefficient Revisited

Recursive Def’n:

```c
int BiCoeff(int n, int m) {
    if (m==0 || m==n) return 1;
    else
        return BiCoeff(n-1,m-1) + BiCoeff(n-1,m);
}
```

Think about it!

Binomial Coefficient Memo-ization

- Start with:
  ```c
  int BiCoeff(int n, int m) {
      if (m==0 || m==n) return 1;
      else
          return BiCoeff(n-1,m-1) + BiCoeff(n-1,m);
  }
  ```
- Think about it!

Longest Common Subsequence

Def’n: subsequence of a given sequence is the given sequence with some elements (possibly none) left out

Say $X = \{A, B, C, B, D, A, B\}$
- $\{B, C, D, B\}$ is a subsequence
- $\{A, C, B, C, B\}$ is not
Longest Common Subsequence

Def'n: common subsequence of two given sequences (say X and Y) is a subsequence of both sequences

Say X = {A, B, C, B, D, A, B}, and Y = {B, D, C, A, B, A}.

- {B, C, A} is a common subsequence

Longest Common Subsequence

Def'n: longest common subsequence (LCS) of two given sequences (say X and Y) is the maximum length subsequence of both sequences

Say X = {A, B, C, B, D, A, B}, and Y = {B, D, C, A, B, A}.

- {B, C, A} is a common subsequence, but is not LCS
- {B, C, B, A} and {B, D, B, A} are LCS

Brute Force LCS

Describe the algorithm
Recursive LCS

**Defn:**

\[
\begin{align*}
c[i,j] = \\
0, & \text{ if } i = 0 \text{ or } j = 0 \\
\text{if } i,j > 0 \text{ and } x_i = y_j & \Rightarrow c[i-1,j-1] + 1 \\
\text{if } i,j > 0 \text{ and } x_i \neq y_j & \Rightarrow \max(c[i-1,j], c[i,j-1])
\end{align*}
\]

Algorithm LCS

**Algorithm LCS**(*X*, *Y*)

Input: String *X* of length *n* and String *Y* of length *m*

Output: 2D Array *c* containing length and 2D Array *b* containing arrows

\[O(nm)\]

**Nuances**

- Dynamic programming can be used any time Memo-ization is used
- Time and space requirements for Dynamic programming and Memo-ization may become prohibitively large
General Approach: Dynamic Programming

- Precompute values from base case up toward solution
- Loosely “non-adaptive”
  - will compute all smaller cases, needed or not

General Approach: Memo-ization

- Save known values as they are calculated
- Generally preferred because:
  - mechanical transformation of ‘natural’ (recursive) problem solution
  - order of computing subproblems takes care of itself
  - may not need to compute answers to all subproblems
- Adaptive
  - only compute needed subcases

When Good Approaches Go Bad

- When the number of possible function values is too high…
- More than a minor annoyance
- No good solution is known, and it is quite possible that no known solution exists
Summary: Dynamic Programming

- Advanced technique for difficult problems
- Trading space for time (when ample)
- Applied when solution domains are not independent
- Dynamic Programming
  - iteratively precompute all values ‘from the bottom’
- Memoization
  - recursively compute and save needed values ‘from the top’