Dynamic Programming Basics

• In general DP algorithms proceed as follows:
  – Establish a recursive property that gives the solution to an instance of the problem in terms of the solution to an easier problem.
  – Solve an instance of the problem in a bottom-up fashion, by starting with the easy instances.

• Remember the Fibonacci numbers?
Wrapping Up Divide and Conquer

- Thresholds.
- When not to use divide and conquer.
Computing Binomial Coefficients

- Recall the binomial coefficient: \( \binom{n}{k} \)
  - This is the number of distinct ways to choose \( k \) things from a set of \( n \) things.

- Appendix A of our textbook gives the following equation:

\[
\binom{n}{k} = \frac{n!}{k!(n-k)!}
\]
Recursive Specification

- It can be shown that:

\[
\binom{n}{k} = \begin{cases} 
\binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\
1 & k = 0 \text{ or } k = n
\end{cases}
\]

- Great! We can easily turn this into a divide and conquer algorithm:

```c
int bin(int n, int k)
{
    if (k ==0 || n == k)
        return 1;
    else
        return bin(n-1, k-1) + bin(n-1, k);
}
```
Dynamic Programming Approach

- We will construct an array $B$, such that the entry $B[i][j] = \binom{i}{j}$.

- We know from the previous slide that
  - $B[i][j] = B[i-1][j-1] + B[i-1][j]$  

- So if we fill in the values from top to bottom, and left to right, we will always have the values we need to compute the next term.

- Let's do an example: $\binom{3}{2}$
Dynamic Programming Algorithm

```c
int bin2(int n, int k) {
    B[0..n][0..k];
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= min(i,k); j++) {
            if (j == 0 || j == i)
                B[i][k] = 1;
            else
                B[i][k] = B[i-1][j-i] + B[i-1][j];
        }
    }
    return B[n][k]
}
```
Shortest Path Problems

- One version requires finding a single shortest path in a directed graph.
  - I.e. of all the sequences of edges that lead from vertex A to vertex B, which sequence has the lowest summed weight.
  - Dijkstra's algorithm (which we will see later) allows us to solve this problem in $\Theta(n^2)$ steps.

- The every pairs shortest path problem requires finding the shortest path from every vertex to every other vertex in a graph.
  - One possibility would be to use Dijkstra's algorithm repeatedly.
Representing a Graph as a Matrix

- We can represent any graph as a matrix $W$ where

$$W[i][j] = \begin{cases} 
\text{weight on edge} & \text{if there is an edge from } v_i \text{ to } v_j \\
\infty & \text{if there is no edge from } v_i \text{ to } v_j \\
0 & \text{if } i = j.
\end{cases}$$

- Let's invent an example.
Floyd's Algorithm

- We want to specify a solution to the shortest paths problem in terms of the solution to an easier problem.
- To this end, we define a series of matrices $D^{(k)}$ where $0 \leq k \leq n$ and where
  - $D^{(k)}[i][j] = \text{length of the shortest path from } v_i \text{ to } v_j \text{ using only vertices in the set } \{v_1, v_2, \ldots, v_k\} \text{ as intermediate vertices.}$
  - For example $D^{(0)}$ is our original matrix $W$, $D^{(n)}$ is our matrix of shortest paths.
- We need an efficient algorithm for computing $D^{(k)}$ from $D^{(k-1)}$ (how efficient?)
Computing $D^{(k)}$ from $D^{(k-1)}$

- We already know the shortest paths using nodes \{\nu_1, \nu_2, \ldots, \nu_{k-1}\} as intermediate nodes. Can we find a shorter path from $\nu_i$ to $\nu_j$ if we throw in $\nu_k$?

- Two possibilities:
  - NO, in which case $D^{(k)}[i][j] = D^{(k-1)}[i][j]$.
  - YES, in which case $D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j]$.
  - Huh? Why?
\[ D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j] \] (Sometimes.)

- First notice that \( D^{(k)}[i][k] = D^{(k-1)}[i][k] \).
  - Because the path from \( v_i \) to \( v_k \) cannot use \( v_k \) as an intermediate node.
- We have assumed that the shortest path from \( v_i \) to \( v_j \) using the first \( k \) nodes includes \( v_k \).
  - The path from \( v_i \) to \( v_k \) on this shortest path can't be longer than \( D^{(k-1)}[i][k] \), otherwise we could replace it with a shorter path.
  - There can't be a path from \( v_i \) to \( v_k \) that's shorter than \( D^{(k-1)}[i][k] \) using \( \{v_j,v_2, \ldots, v_{k-1}\} \), by definition.
- The same reasoning applies to the path from \( v_k \) to \( v_j \).
Floyd's Algorithm

void floyd(int n, number W[][], number D[][])
{
    number[][] PrevD = W;
    for (int k = 1; k <= n; k++) {
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                D[i][j] = min(PrevD[i][j],
                               PrevD[i][k] + PrevD[k][j]);
            }
        }
        PrevD = D;
    }
}

(We can get away without using the PrevD matrix)
Floyd Analysis

- Time complexity?
- Space complexity?
Returning the Shortest Paths

- We can modify the algorithm to allow us to find the paths themselves, not just lengths:

```c
void floyd2(int n, number W[][], number D[][],
            index P[][])
{
    P[0..n][0..n] = 0;
    D = W;
    for (int k = 1; k <= n; k++) {
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (D[i][k] + D[k][j] < D[i][j]) {
                    P[i][j] = k;
                    D[i][j] = D[i][k] + D[k][j];
                }
            }
        }
    }
}
```
Returning a Shortest Path

- The following procedure then prints out the intermediate nodes on the shortest path:

```c
void path(index q, index r, index P[][[]])
{
    if (P[q][r] !=0) {
        path(q, P[q][r], P);
        print(P[q][r]);
        path(P[q][r], r, P);
    }
}
```

- Time complexity?
Principle of Optimality

- Dynamic programming works on optimization problems if the principle of optimality holds:
  - The optimal solution to an instance of a problem always includes the solution to all sub-instances.
  - If we think of a solution as a series of steps, the next step is a function only of our current position.
  - Another way of saying this:
    - A problem satisfies the principle of optimality if, after we remove one decision from an optimal solution, the remaining solution(s) is(are) optimal for the remaining problem(s).