Dynamic Programming

Most material are from
http://cs360.cs.ua.edu/spring2014/schedule.htm

Dynamic Programming

- Use a recursive formula
- But recursion can be inefficient because it might repeat the same computations multiple times
- So use an array to store and lookup the subproblem results as needed

Step of the dynamic programming

1. Characterize the structure of an optimal solution
2. Define the value of an optimal solution recursively in terms of the optimal solutions to subproblems
3. Write a recursive algorithm based on recurrence to compute the minimum cost

Examples

- Fibonacci numbers
- 0-1 Knapsack
- Longest Common Subsequence (LCS)
- Floyd’s all-pairs shortest-paths

Fibonacci number

Fibonacci numbers:

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

F(0) = 1
F(1) = 1
F(n) = F(n–2) + F(n–1), when $n \geq 2$

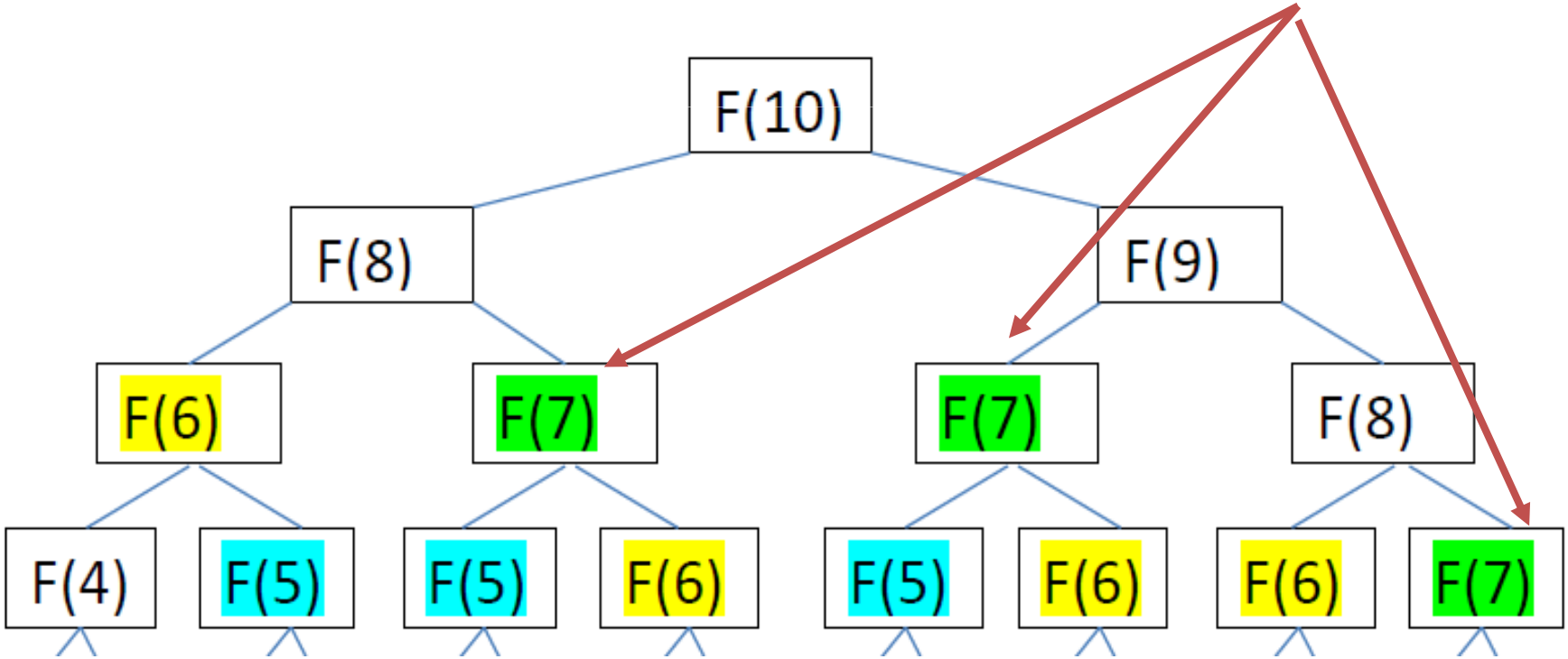
```
int F (int n) {
        if (n==0 || n==1) return 1;
        else return F(n–2) + F(n–1);
}
```

Fibonacci number

```
int F (int n) {
      if (n==0 || n==1) return 1;
      else return F(n–2) + F(n–1);
}
```

Exponential runtime

Repeated subproblem



Fibonacci number

- Dynamic programming ⇒ **use a table** (array) to **remember** the values that have previously been computed
- First method: Bottom-up (use a table instead of recursion)

```
allocate array A[0…n];
for (k=0; k<=n; k++)
    if (k==0 || k==1) A[k] = 1;
    else A[k] = A[k–2] + A[k–1];
return A[n];
```

Fibonacci number

OLD

```
int F (int n) {
   if (n==0 || n==1) return 1;
   else return F(n–2) + F(n–1);
}
```

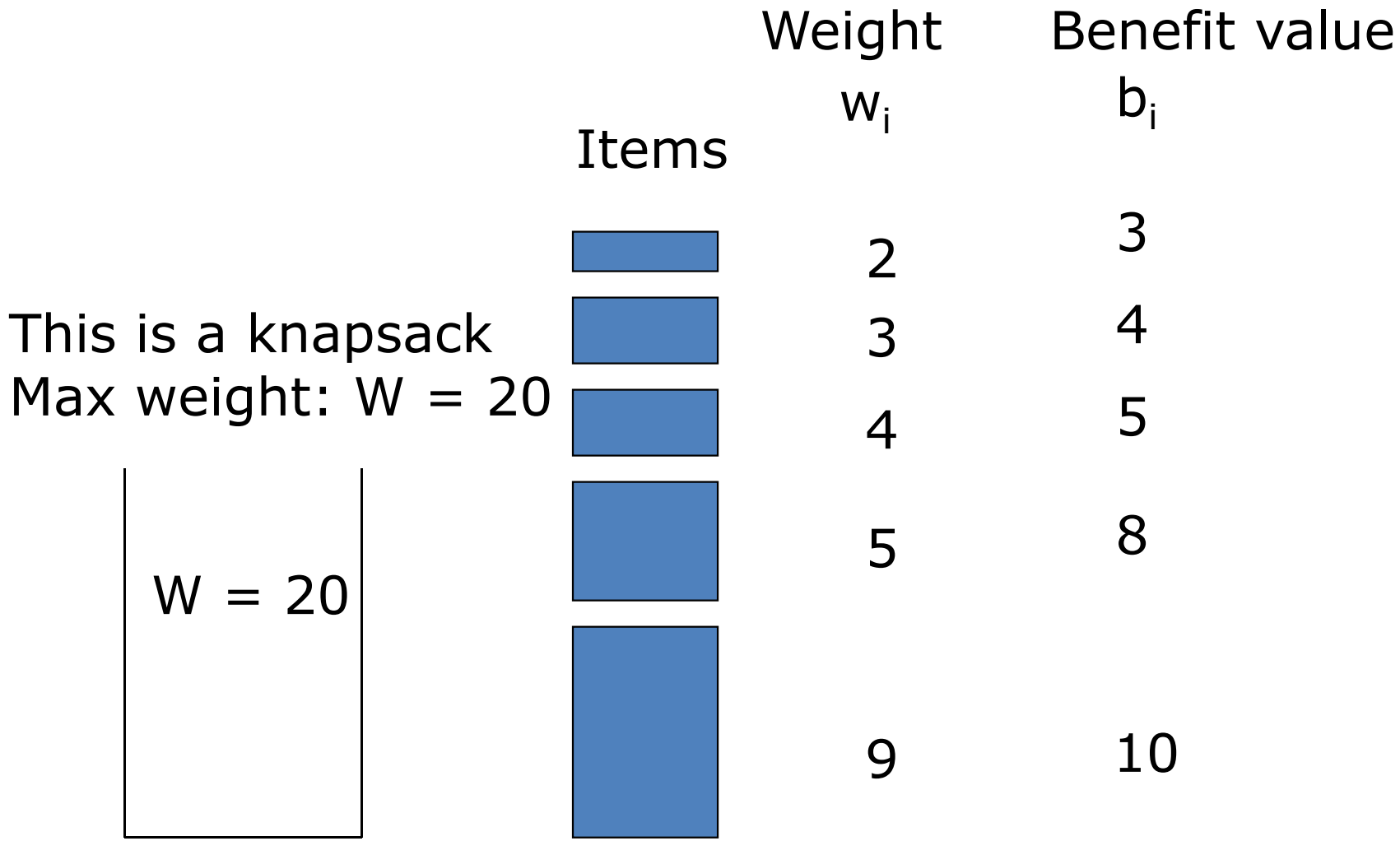
NEW

```
   allocate array A[0…n];
   for (k=0; k<=n; k++)
        if (k==0 || k==1) A[k] = 1;
        else A[k] = A[k–2] + A[k–1];
   return A[n];
```

0-1 Knapsack problem

- Given a knapsack with maximum capacity *W*, and a set *S* consisting of *n* items
- Each item *i* has some weight w_i and benefit value b_i (all w_i , b_i and *W* are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

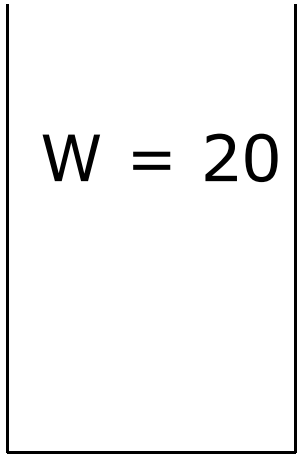
0-1 Knapsack problem: a picture



Can we use Greedy?

Pick largest benefit first

B = 10	W = 9
B = 10 + 8	W = 9 + 5
B = 18 + 5	W = 14 + 4
B = 23 + 3	W = 18 + 2
B = 26	W = 20

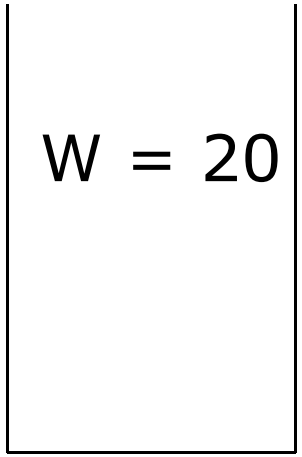


Weight w_i	Benefit value b_i
2	3
3	4
4	5
5	8
9	10

Can we use Greedy?

Pick largest benefit first
B = 10 W = 19

Better solution is
B = 8 +5+4+3 = 20
W = 9+4+3+2 = 18



Weight w_i	Benefit value b_i
2	3
3	4
4	5
9	8
19	10

0-1 Knapsack

Given:

n = number of objects

M = maximum weight capacity

W[i] = weight of object i

V[i] = value of object i

You want to find T(n,M) such that

T (n, M) = max possible total profit such that we choose any subset of objects {1…n} and maximum weight capacity is M

0-1 Knapsack

T (n, M) = max possible total profit such that we choose any subset of objects {1…n} and maximum weight capacity is M

General case: for any j and k such that $0 \leq j \leq n$ and $0 \leq k \leq M$ we will define T(j,k) for each condition

T (j, k) = max possible total profit such that we choose any subset of objects {1…j} and maximum weight capacity is k

1)If j = 0 or k = 0 then T(j,k) = 0

0-1 Knapsack

T (j, k) = max possible total profit such that we choose any subset of objects {1…j} and maximum weight capacity is k

1)If j = 0 or k = 0 then T(j,k) = 0

2)If j > 0 then there are two cases

2.1) W[j] > k cannot take object j

Then, T(j,k) = T(j-1,k)

2.2) W[j] ≤ k there are two cases

O-1 Knapsack

T (j, k) = max possible total profit such that we choose any subset of objects {1…j} and maximum weight capacity is k

1)If j = 0 or k = 0 then T(j,k) = 0

2)If j > 0 then there are two cases

2.1) W[j] > k cannot take object j

Then, T(j,k) = T(j-1,k)

2.2) W[j] ≤ k there are two cases

→ Take j

→ Do not take object j

We will choose the option that results in the maximum value

0-1 Knaksack

1)If j = 0 or k = 0 then T(j,k) = 0

2)If j > 0 then there are two cases

2.1) W[j] > k cannot take object j

Then, T(j,k) = T(j-1,k)

2.2) W[j] ≤ k there are two cases

→ Take j

T(j,k) = T(j-1,k-W[j]) + V[j]

→ Do not take object j

T(j-1,k)

Max(T(j-1,k-W[j]) + V[j], T(j-1,k))

0-1 Knapsack

Given: n = number of objects

M = maximum weight capacity

W[i] = weight of object i

V[i] = value of object i

```
allocate array T[0…n][0…M];
for j = 0 to n
    for k = 0 to M
        if (j==0 || k==0) T[j][k] = 0;
        else if (k < W[j]) T[j][k] = T[j–1][k];
        else T[j][k] = max {T[j–1][k], T[j–1][k–W[j]] + P[j]};
```

Problem

- **624 – CD**
- **562 - Dividing coins**
- **10130 – SuperSale**
- **10465 - Homer Simpson**

Longest Common Subsequence

Application: comparison of two DNA strings

Ex: X= {A B C B D A B },

Y= {B D C A B A}

Longest Common Subsequence:

X = A **B** **C** **B** D **A** B

Y = **B** D **C** A **B** **A**

Brute force algorithm would compare each subsequence of X with the symbols in Y

LCS Algorithm

- if |X| = m, |Y| = n, then
  then there are 2^m subsequences of x;
  to be compared with Y (n comparisons)
- running time is $O(n\ 2^m)$
- Can we do better?

LCS

- Given: two strings: X[1…m] Y[1…n]

Find L (m, n) = length of the LCS of

X[1…m] and Y[1…n]

General case

L (j, k) = length of the LCS of X[1…j] and Y[1…k] (0 ≤ j ≤ m, 0 ≤ k ≤ n)

How many cases?

LCS

L (j, k) = length of the LCS of X[1…j] and Y[1…k] (0 ≤ j ≤ m, 0 ≤ k ≤ n)

1) j = 0 or k = 0
2) J > 0 and k > 0
   2.1) X[j] = Y[k]
   2.2) X[j] $\neq$ Y[k]

LCS

L (j, k) = length of the LCS of X[1…j] and Y[1…k] (0 ≤ j ≤ m, 0 ≤ k ≤ n)

1) j = 0 or k = 0 Then L(j,k) = 0
2) J > 0 and k > 0
   2.1) X[j] = Y[k] then
   L(j,k) = L(j-1,k-1) + 1
   2.2) X[j] $\neq$ Y[k] then has to choose max of
   → take j not take k
   → take k not take j

LCS

L (j, k) = length of the LCS of X[1…j] and Y[1…k] (0 ≤ j ≤ m, 0 ≤ k ≤ n)

1) j = 0 or k = 0 Then L(j,k) = 0

2) J > 0 and k > 0

   2.1) X[j] = Y[k] then

   L(j,k) = L(j-1,k-1) + 1

   2.2) X[j] $\neq$ Y[k] then has to choose max of

   → take j not take k L(j,k) = L(j,k-1)

   → take k not take j L(j,k) = L(j-1,k)

   max(L(j,k-1), L(j-1,k))

LCS

```
allocate array L[0...m][0...n];
for j = 0 to m
   for k = 0 to n
        if (j==0 || k==0)
                L[j][k] = 0;
        else if (X[j]==Y[k])
                L[j][k] = L[j–1][k–1] + 1;
        else
                L[j][k] = max {L[j–1][k], L[j][k–1]};
```

Problem

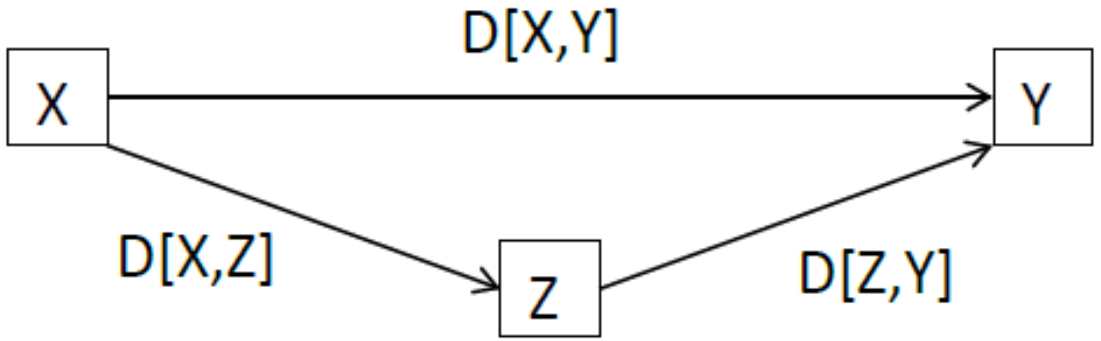
- **10405 - Longest Common Subsequence**
- **231 - Testing the CATCHER**
- **531 – Compromise**
- **481 - What Goes Up**
- **10066 - The Twin Towers**
- **10100 - Longest Match**

All pairs shortest path

- Given a weighted (undirected or directed) graph, find a minimum-distance path from each start vertex to each destination vertex
- **Restriction: here we allow edges with negative weights, but not any cycle with negative total weight**
- why negative-weight cycle is forbidden:
  - Loop over the negative-weight to -INF

All pairs shortest path

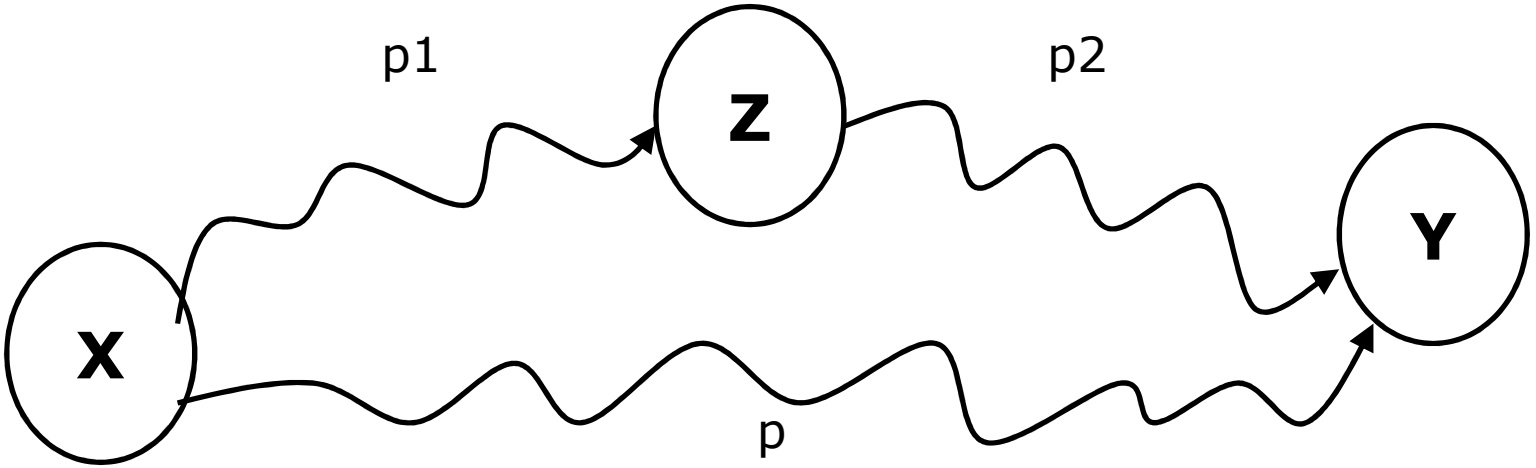
- Intuition for designing an algorithm to solve APSP problem:
- D[X,Y] = distance from X to Y = length of shortest path from X to Y




```
if (D[X,Z] + D[Z,Y] < D[X,Y])
        D[X,Y] = D[X,Z] + D[Z,Y]
```

Floyd-Warshall Algorithm

- Dynamic programming
- $O(V^3)$
- Allow negative weight values
- No negative-weight cycle

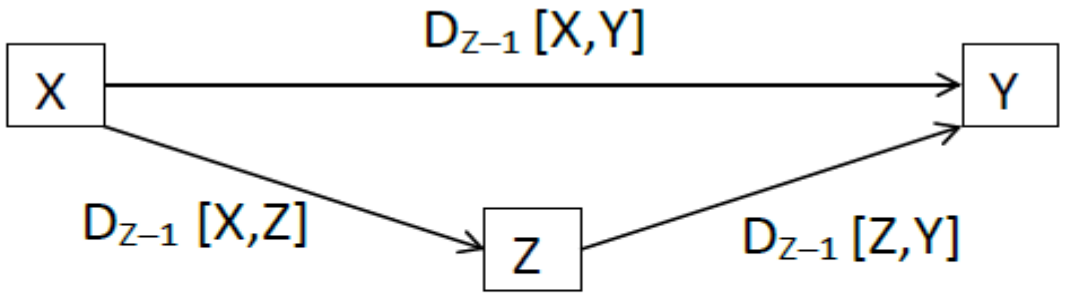


is Z in the path?

Floyd-Warshall Algorithm

Assume vertices are numbered 1…n

- D_Z [X,Y] = length of shortest path from X to Y such that this path is allowed to pass through any or all of the intermediate vertices 1…Z, but it is not allowed to pass through any of Z+1…n



$$D_Z [X,Y] = \min (D_{Z-1} [X,Y],\ D_{Z-1} [X,Z] + D_{Z-1} [Z,Y])$$

Floyd-Warshall Algorithm

```
// initialize D0 = weighted adjacency matrix
for X = 1 to n
        for Y = 1 to n
                if (X==Y)  D0 [X,Y] = 0;
                else if (edge (X,Y) exists)  D0 [X,Y] = weight(X,Y);
                else D0 [X,Y] = ∞;
// compute each DZ matrix from values in the DZ–1 matrix
for Z = 1 to n
        for X = 1 to n
                for Y = 1 to n
                        if (DZ-1 [X,Z] + DZ-1 [Z,Y] < DZ-1 [X,Y])
                                DZ [X,Y] = DZ-1 [X,Z] + DZ-1 [Z,Y];
                        else  DZ [X,Y] = DZ-1 [X,Y];
```

Floyd-Warshall Algorithm

Running time of Floyd's algorithm = $\theta(n^3)$

Memory usage of Floyd's algorithm = $\theta(n^3)$

D_Z [X,Y] can be stored as 3-dimensional array with elements D[X,Y,Z] for $1 \le X \le n$, $1 \le Y \le n$, $0 \le Z \le n$

Problems

- **437 - The Tower of Babylon**
- **429 - Word Transformation**
- **534 - Frogger**
- **544 - Heavy Cargo**
- **567 - Risk**
- **821 - Page Hopping**

Other interesting problems

- **147 – Dollars**
- **357—Let Me Count The Ways**
- **674 –Coin Change**
- **11137 –Ingenuous Cubrency**
- **348 - Optimal Array Multiplication Sequence**
- **443 - Humble Numbers**
- **485 - Pascal's Triangle of Death**
- **487 - Strategic Defense Initiative**
- **507 - Jill Rides Again**