Dynamic Programming

Dynamic programming was invented by Richard Bellman in the early 1950's while he worked for the RAND corporation on "multistage decision processes".

- Origins of the term "Dynamic Programming" (1950):
  - "What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to."

Dynamic Programming Overview

- Dynamic programming solves optimization problems.
  - The overall strategy underlying the algorithm is that the given problem can be solved by first solving smaller subproblems that have the same structure. Two features characterize these subproblems:

1. Unlike the divide-and-conquer algorithm, which relies on subproblems being independent, dynamic programming handles problems that have overlapping subproblems.
2. We can apply a recursive strategy: We find an optimal solution of each subproblem by first finding optimal solutions for its contained subproblems.

Coins Again

- We are given an unlimited number of coins of various denominations: $d_1, d_2, \ldots, d_m$.
  - We have the same objective as before: Pay out a given sum S with the smallest number of coins possible.
  - The denomination "system" presents us with three possibilities:
    - The d_i values are such that a greedy strategy can pay out the sum S in an optimal fashion.
    - The d_i values are such that a greedy strategy cannot pay out the sum S in an optimal fashion although such an optimal payout does exist.
    - The d_i values are such that **no** strategy can pay out the amount S.

      (Trivial example: $m = 1$, $d_i = 2$ cent coin and S is an odd amount).

A Recursive Strategy for the Coin Problem

- Find an algorithm that produces an optimal solution for ***any* denomination system** or an indication that no solution exists.
- Notation:
  - **Let $coins[i]$ be the smallest number of coins required to pay out the sum i.**
    - We assume $coins[i]$ will be "undetermined" if no solution exists.
    - Note that $coins[0] = 0$ (no coins needed).
  - Idea behind the solution:
    - Sequentially try all possibilities for the first coin paid out.
    - If the coin paid out is d_i, we now recursively solve the change problem for $S - d_i$.
    - So, we can solve the problem recursively but with the added constraint the we always ask for the minimum coin number.

A Recursive Strategy for the Coin Problem

– These considerations lead to the following recursion:

$$coins[i] = 1 + \min_{d_j \le i}\{coins[i - d_j]\}.$$

– This recursion will be at the heart of our dynamic programming algorithm.

– There are two approaches that can be used in working with this recursion:

1. We can insist that the recursion defines a **recursive function** that simply calls itself to get the optimal values associated with the smaller problems. (This is **not** dynamic programming).
   OR
2. We can consider *coins* to be an array that stores the optimal values associated with the smaller problems.

– Just to eliminate it from any further discussion, we start by showing that the first approach is extremely bad!

A Recursive Function for the Coin Problem

– We start with: $coins[i] = 1 + \min_{d_j \le i}\{coins[i - d_j]\}.$

- Recalling that recursive calls were used in divide and conquer we go for the following implementation:

D&C for CP Pseudo-code 1:

```
function coins(i):
    if (i=0) then return 0; // base case
    // recursion:
    min:=infinity;    // Use a very large number here…
    for j:=0 to m do
        if (d[j] <= i) then
            smaller_sol := coins(i - d[j]);
            if smaller_sol < min then min := smaller_sol;
    return 1 + min;
```

– The main program calls $coins(S)$.

A Recursive Strategy for the Coin Problem

- Analysis:
  - The execution time is exponential in S.
  - Consider the recursion tree for the calling of *coins*(S) with denominations 1¢ and 2¢.

$$T(S) = T(S\text{-}1) + T(S\text{-}2)$$

  - Execution time:

$$\Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^{S}\right) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^{2^n}\right)$$

n is the bit length of the input. Execution time is hyper-exponential with respect to the input bit length!

Why is this? Observe that many of the recursive calls are redundant.

- For example, *coins*(2) is called several times.

Dynamic Programming Solution

- Recursive calls worked for divide and conquer because the same subproblems did **not** reappear in several areas of the recursion tree.

- Here is a better way to use the recursion formula:
- Consider pseudo-code 2:

NOTE: coins[ ] is now an array.

```
coins[0] := 0;
for i := 1 to S do  // Do minimal payout for all values ≤ S.
    min := infinity;
    for j := 1 to m do
        if d[j] <= i and coins[i - d[j]] < min then
            min := coins[i - d[j]];
    coins[i] := 1 + min;
return coins[S];
```

Dynamic Programing Solution

- Execution time for Pseudo-code 2:
  - Note: Unlike pseudo-code 1 it is *not* a recursive function.
    - It has just two nested loops with constant time operations at the innermost part, so: $\Theta(mS)$. (Much, much better).
    - But it is still exponential in the bit length of the input….
      - So greedy will be better **if it is compatible with the denominational system.**

Recovery of the Solution

The previous pseudo-code only gives the coin count.

- There is no report on which coins make up the optimal solution.
- We need an array (choice[ ]) that will track the best solution.
- Remember only the last choice since the previous choices can be recovered from the sub-problems.
- This is Dynamic Programming!

```
coins[0] := 0;
for i := 1 to S do
   min := infinity;
   for j := 1 to m do
      if d[j] <= i and
              coins[i - d[j]] < min then
         min := coins[i - d[j]];
         minchoice := j;      // Note!
   coins[i] := 1 + min;
   choice[i] := minchoice; // Note!

// Now recover the solution
if coins[S] = infinity then
   write 'No solution!';
else
   while S > 0 do
      write d[choice[S]];
      S := S - d[choice[S]];
```

Dynamic Programming vs. Greedy (1)

- Note the difference in strategies:
  - As an algorithm, dynamic programming is the "clever cousin" of greedy.
    - When given a sum i, greedy immediately assumes that it should deflate the sum by the value of the largest denomination to get the smaller problem.
  - Dynamic programming is more thorough:
    - It tries a variety of denominations.
    - The success of each attempt is evaluated by knowing the coin count associated with the solution of the smaller problem (and this has already been computed and stored in an array!).
      - Terminology: We can also refer to the optimal coin count for a subproblem as the **optimal score** for that subproblem.

Dynamic Programming vs. Greedy (2)

- Note the difference in strategies:
  - Greedy builds the final solution as it makes its greedy choices.
  - Dynamic programming does not immediately derive the final solution.
    - It concentrates on the optimal scores associated with all solutions.
    - After all optimal scores are computed, it is in a position to derive the final solution using a trace-back strategy.

Some Terminology

- Notice the two components of a solution:
  - ***Optimal scores*** were computed.
    - In dynamic programming we want to minimize or maximize some score value specified by the problem.
    - When we use the term "optimal score" we are referring to this maximum or minimum (depending on the problem).
      - In the coins problem the optimal score to be computed was the minimum number of coins required for the payout.
      - Note that optimal scores are computed for a whole range of smaller problems that grew in the size (as determined by the input parameter) until we got the optimal score for the largest problem.
  - After all optimal scores are computed the ***configuration*** to achieve that optimal score was determined using a trace-back strategy.
    - In the coins problem a configuration is the set of coins used in a payout.

Dynamic Programming

- In dynamic programming the optimal scores are computed for various subproblems and are stored in a score array.
  - For example: in the coin changing problem we used a 1D array $coins[1..S]$ where $coins[i]$ recorded the number of coins necessary to pay out sum $i \leq S$.
    - A special value of infinity recorded the failure of the algorithm.
    - Note that there was also the notion of **where** the answer for the full problem would be found (in $coins[1..S]$ it was in $coins[S]$).
- In general, we need a strategy that allows us to **determine the optimal score for any problem by using the optimal scores of smaller problems**.
- These optimal scores for small problems will have been calculated earlier.
  - Recall coin changing: $coins[i] = 1 + \min_{j \ni d_j \leq i} \{coins[i - d_j]\}$

A Recipe for Designing a D. P. Algorithm (1)

1. Describe the subproblems.
   - Provide a clear description as to what makes up a subproblem.
   - How is a score evaluated for a subproblem?
   - What is the recursion defining the score values?
2. Determine how an array will be used to evaluate the scores:
   - What are the dimensions of the array?
   - How does the score in a cell of the array depend on the scores of other cells in the array?
     - Define the order of computation.
   - Where are the base case (initial) scores set up?
     - What are their values?
   - Where do we find the optimal score for the given problem?

A Recipe for Designing a D. P. Algorithm (2)

3. Recovery of the solution
   - For every cell, keep track of the cell that was used to generate the optimal score.
     - Which cell is responsible for generating the optimal score?
     - Usually: which cell won in the min or max "competition" specified by the recursion?
   - Use a trace-back strategy to determine the full configuration that gives the optimal score for the full problem.

Longest Common Subsequence (1)

- Motivation:
  - How similar are humans and chimpanzees?
    - One possible way of dealing with this question is to look at the similarities between their DNA sequences.
    - DNA sequences are the strings over a short alphabet: (**`A, C, G, T`**).
    - Our next problem presents one way of discovering how similar two DNA sequences are.

Longest Common Subsequence (2)

- First some notation:
  - A *subsequence* of a string X is a string that can be obtained by deleting some of the characters from X.
    - Note that characters in the subsequence have the same order as they did in the given sequence X.
    - For example:
      X = "`supercalifragilisticexpialidocious`"
      T = "`perfidious`" is a subsequence of X.

Longest Common Subsequence

- Finding the longest common subsequence:
  - We are given two strings:
    $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$.
  - The problem is to find the longest string Z that is a subsequence of both X and Y.
- Problem instance: Given two sequences:

  $X =$ **A L G O R I T H M**
  $Y =$ **L O G A R I T H M**

  - What is the LCS? Characters in an LCS are underlined.
  - Another example:

X = "**supercalifragilisticexpialidocious**" and
Y = "**supercalafajalistickespeealadojus**".
The LCS of X and Y is Z = "**supercalfalisticepaldous**".

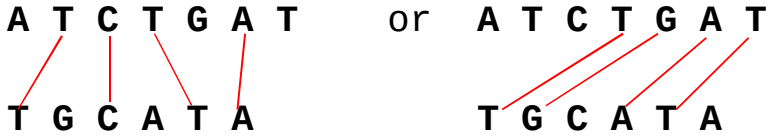
http://en.wikipedia.org/wiki/Supercalifragilisticexpialidocious.

There May be Several LCS's

- Is an LCS unique? No. Consider:
  X = **ATCTGAT** and Y = **TGCATA**
  We can have either **TCTA** or **TGAT** as an LCS.
- Visualizing the LCS:

  A**T**-**C**-**T**G**A**T or ATC**TG**-**AT**-
  -**T**G**C**A**T**-**A**- ---**TG**C**AT**A

  – Or match up with lines: (lines never cross!)



Prefixes

- We compute the LCS using dynamic programming:
  - Our strategy is to solve the LCS of strings X and Y, by first solving various subproblems.
  - A subproblem for us will be to first find an LCS for various **prefixes** of X and Y.
    - The optimal value for such a subproblem is the length of the LCS of the prefixes.
    - The configuration for this subproblem is the LCS itself.

The i^{th} prefix of a string X denoted by $X[i]$ will be the substring of X made up from the first i characters of X.

- Since prefixes are essentially defined by their last position, we will be interested in the following clever observations about the last character of an LCS:

Optimal Substructure of an LCS

- Let $X[i] = (x_1, \ldots, x_i)$ and $Y[j] = (y_1, \ldots, y_j)$ be prefixes with LCS $Z[k] = (z_1, \ldots, z_k)$.
- Then:

1. If $x_i = y_j$, then $z_k = x_i = y_j$ and $Z[k\text{-}1]$ is an LCS of $X[i\text{-}1]$ and $Y[j\text{-}1]$.

   - For $x_i \neq y_j$ we have the following two cases:

2. $z_k \neq x_i$ implies that $Z[k]$ is an LCS of $X[i\text{-}1]$ and $Y[j]$.

3. $z_k \neq y_j$ implies that $Z[k]$ is an LCS of $X[i]$ and $Y[j\text{-}1]$.

Optimal Substructure of an LCS

- Discussion for point (1):
  - If $z_k \neq x_i$, then we could simply append x_i to $Z[k]$ to obtain an LCS of $X[i]$ and $Y[j]$ with longer length than k, contradicting the assumption that $Z[k]$ was the *longest* common subsequence. So: $z_k = x_i = y_j$.
  - Then it is clear that $Z[k\text{-}1]$ is an LCS of $X[i\text{-}1]$ and $Y[j\text{-}1]$ because if there was an LCS W of $X[i\text{-}1]$ and $Y[j\text{-}1]$ that had k or more characters we would simply append x_i to it to get an LCS of $X[i]$ and $Y[j]$ longer than $Z[k]$ (contradiction).

Optimal Substructure of an LCS

- Discussion for point (2):
  - If $z_k \neq x_i$, then $Z[k]$ must be a common subsequence of $X[i\text{-}1]$ and $Y[j]$.
  - Furthermore, $Z[k]$ must be the LCS of $X[i\text{-}1]$ and $Y[j]$ since if there was a longer common subsequence of $X[i\text{-}1]$ and $Y[j]$ (say W with length greater than $Z[k]$), then since W is also a subsequence of both $X[i]$ and $Y[j]$ we would have a common subsequence of $X[i]$ and $Y[j]$ longer than $Z[k]$ (contradiction).
- Discussion for point (3):
  - symmetric to that of point (2)

A Recurrence for LCS

– Let score $C[i, j]$ be the **length** of the LCS of X_i and Y_j.

– Note the base case: $C[i, 0] = C[0, j] = 0$ for all i, j.

– Main recurrence:

$$C[i, j] = \begin{cases} C[i-1, j-1]+1 & \text{if } X[i] = Y[j] \\ \max\{C[i-1, j], C[i, j-1]\} & \text{if } X[i] \neq Y[j] \end{cases}$$

– Order of evaluation:

- To compute $C[i, j]$, we need to know the values of: $C[i - 1, j - 1]$, $C[i - 1, j]$, and $C[i, j - 1]$.
- Computing the cells from the top-most corner by rows will ensure that these scores will be ready before computing $C[i, j]$.

– Our answer for the LCS will be found in: $C[m, n]$.

LCS D.P. Pseudocode

```
// base cases
for i := 0 to m do C[i,0] := 0;
for j := 0 to n do C[0,j] := 0;

// filling the matrix
for i := 1 to m do
  for j := 1 to n do
    if X[i] = Y[j] then C[i,j] := C[i-1, j–1] + 1;
    else C[i,j] := max(C[i-1,j], C[i,j-1]);
return C[m,n];
```

– Analysis:

- $\Theta(1)$ per entry, $\Theta(mn)$ entries, $\Theta(mn)$ total.

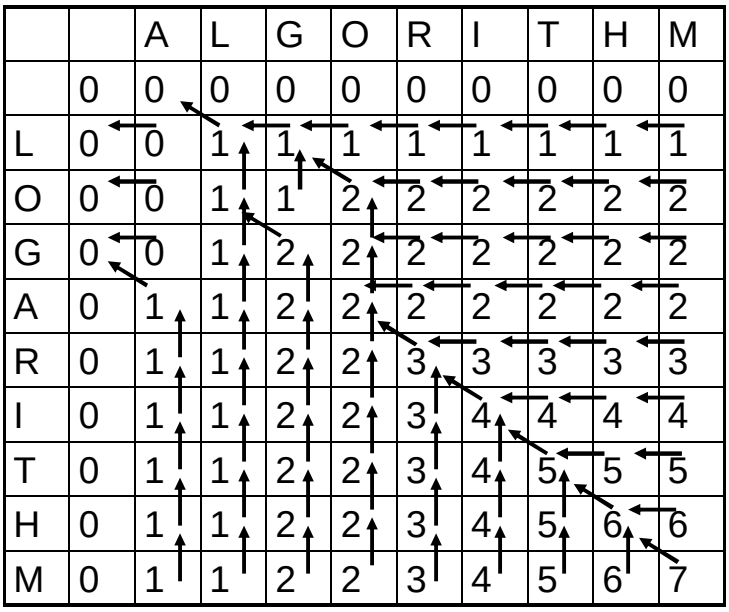
		A	L	G	O	R	I	T	H	M
	0	0	0	0	0	0	0	0	0	0
L	0	0	1	1	1	1	1	1	1	1
O	0	0	1	1	2	2	2	2	2	2
G	0	0	1	2	2	2	2	2	2	2
A	0	1	1	2	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3	3
I	0	1	1	2	2	3	4	4	4	4
T	0	1	1	2	2	3	4	5	5	5
H	0	1	1	2	2	3	4	5	6	6
M	0	1	1	2	2	3	4	5	6	7

Recovering the Actual LCS

- Note that this program only computed the length of the LCS. To get the actual LCS:
  - For each cell, keep a backpointer $D[i,j]$ to one of $(i-1, j-1)$, $(i-1, j)$, or $(i, j-1)$ (depending on which one determined $C[i,j]$).
    - We use D for "Direction" and give each direction a name:

`D[i, j]` =

up-left	if	`C[i, j] = 1 + C[i-1, j-1]`
up	if	`C[i, j] = C[i-1, j]`
left	if	`C[i, j] = C[i, j-1]`



		A	L	G	O	R	I	T	H	M
	0	0	0	0	0	0	0	0	0	0
L	0	0	1	1	1	1	1	1	1	1
O	0	0	1	1	2	2	2	2	2	2
G	0	0	1	2	2	2	2	2	2	2
A	0	1	1	2	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3	3
I	0	1	1	2	2	3	4	4	4	4
T	0	1	1	2	2	3	4	5	5	5
H	0	1	1	2	2	3	4	5	6	6
M	0	1	1	2	2	3	4	5	6	7

Using the Backpointers

- We work backwards from $D[m,n]$:
  - When $D[i,j]$ is $(i–1, j–1)$, i.e. up-left, then $X[i] = Y[j]$ and this character is in the LCS.

```
row := m; col := n; lcs := "";
while (row > 0 and col > 0) do
  if (D[row,col] = upleft) then // X[row] = Y[col]
    lcs := lcs.X[row];
    row := row-1; col := col-1;
  else if (D[row,col] = up) then
    row := row-1;
  else if (D[row,col] = left) then
    col := col-1;
reverse lcs;
return lcs;
```

		A	L	G	O	R	I	T	H	M
	0	0	0	0	0	0	0	0	0	0
L	0	0	1	1	1	1	1	1	1	1
O	0	0	1	1	2	2	2	2	2	2
G	0	0	1	2	2	2	2	2	2	2
A	0	1	1	2	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3	3
I	0	1	1	2	2	3	4	4	4	4
T	0	1	1	2	2	3	4	5	5	5
H	0	1	1	2	2	3	4	5	6	6
M	0	1	1	2	2	3	4	5	6	7

The Integer Knapsack Problem

- Problem specification:
  - We are given n objects and a knapsack.
  - Each object i has a positive weight w_i and a positive value v_i.
  - The knapsack can carry a weight of at most W.
  - Fill the knapsack so that the value of objects in the knapsack is maximized.
    - Fractional objects NOT allowed.

Knapsack Problem Solutions

- Brute force:
  - Try all possibilities.
  - An object can be in or out and we sum weights to be sure we are not over W.
  - This takes O($n2^n$) time and is a very bad approach.

Knapsack Problem Solutions

- Greedy:
  - At each step add the object with the highest v_i/w_i ratio (worked for fractional….).
  - Example with $W = 500$
    (So we cannot select more than 500 g.)

Object	weight	value	v_i/w_i
sunglasses	50 g	50	1
swimsuit	300 g	450	1.5
towel	400 g	520	1.3
cooler	250 g	100	0.4

  - Greedy solution: {swimsuit, sunglasses} Value: 500
  - But this is clearly not optimal; we could take the towel and sunglasses, for a total value of 570.

Knapsack Problem Solutions

- Dynamic Programming:
  - We imagine all the objects numbered from 1 to n.
  - Definition of a subproblem:
    - Score specification: Let $V[i, j]$ be the maximum value of the objects, selected from the first i objects, that can fit into a knapsack with upper weight limit j.
    - The final optimal solution will be found in $V[n, W]$.
      - Optimal scores are stored in the $V[i, j]$ array.
      - The configuration for an optimal score will be the set of objects chosen.

Knapsack Problem: Deriving a Recurrence (1)

- Key observation: We either use object i or we do not.
  - Suppose object i is not in the Knapsack.
    - Then there is no difference between $V[i - 1, j]$ and $V[i, j]$.
  - Suppose object i is in the Knapsack.
    - Claim. In this case $V[i, j] = V[i - 1, j - w_i] + v_i$.
    - Proof. Consider an optimal selection extracted from the first $i - 1$ objects with a weight limitation of $j - w_i$.
    - An optimal selection from the first i -1 objects is not more valuable than those chosen from the first $i - 1$ objects as used in $V[i, j]$.

Knapsack Problem: Deriving a Recurrence (2)

– Repeating:
  - An optimal selection from the first i -1 objects is not more valuable than those chosen from the first i - 1 objects as used in $V[i, j]$.

– This is true because:
  - A more valuable selection from objects 1 to i - 1 could be extended with object i and we would get a total score in excess of $V[i, j]$ in contradiction of the fact that $V[i, j]$ is optimal.

– So the value of $V[i, j]$ must be v_i plus the optimal solution for the first i - 1 objects with a weight limitation of j - w_j.

Knapsack Problem: Deriving a Recurrence (3)

– Considering the above facts we are able to make up the following recurrence for $V[i, j]$:

$$V[i, j] = max\{V[i - 1, j],\ v_i + V[i - 1, j - w_i]\}$$

  - Base case: $V[0, j] = 0$.
  - Order of computation:
    – Use row-order from top-left down to the bottom-right corner.

Knapsack Problem: Pseudocode for Dynamic Programming

```
for j := 0 to W do
  V[0, j] := 0;
for i := 1 to n do
  for j := 1 to W do
    V[i, j] := V[i-1, j];
    if (w[i] <= j) then
       V[i, j] := max(V[i-1, j], V[i-1, j-w[i]] + v[i]);
return V[n, W];
```

Knapsack Problem: Notes on Pseudocode

- We can make the program more memory efficient..
- Note that to compute cell $V[i, j]$, we need only the cells from the previous line and to the left of $V[i\text{-}1, j]$.

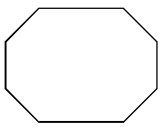
```
for j := 0 to W do
  V[j] := 0;
for i := 1 to n do
  for j := W down to 1 do
    if (w[i] <= j) then
         V[j] := max(V[j], V[j-w[i]] + v[i]);
return V[W];
```

Triangulation in Computational Geometry

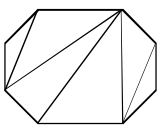
- Motivation:
  - Computational geometry is a branch of computer science that studies algorithms for solving geometric problems.
    - It has applications in computer graphics, robotics, VLSI design, computer-aided design, statistics, numerical computations, and other areas.
    - The first work in the area was done by Emile Lemoine (1902) who studied a problem dealing with the number of operations you need to do in different constructions with compass and straight-edge ruler.

Shortest Length Triangulation (1)

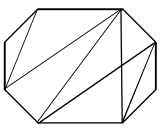
- More terminology:
  - Convex polygon: A polygon is convex, if for any two points on its boundary, all points on the line segment drawn between them are contained in the polygon's interior.
    - A convex polygon is represented by listing its vertices in clockwise order.
  - Triangulation: Every convex polygon can be split into disjoint triangles by a set of non-crossing chords (line segments connecting non-adjacent vertices of the polygon).



Convex polygon



Triangulation



Not allowed

Shortest Length Triangulation (2)

– The length of triangulation is the total length of all chords in the triangulation plus the length of the polygon boundary.
  - Triangulations are important in both graphics and in various applications involving numerical algorithms.
  - A triangulation will decompose a complex polygon into a set of simple shapes (triangles) allowing for simpler methods for dealing with such a polygon.
  - We may want to find a triangulation for which some parameter is optimized (in our example we will be looking for the triangulation with the shortest total length of the chords used in the triangulation).

Shortest Length Triangulation (3)

- Problem specification:
  – Given a convex polygon with vertices listed in clockwise order: $(v_1, v_2, \ldots, v_n)$, find the triangulation that has the shortest length.
- Notation:
  – Let $d(v_i, v_j)$ represent the Euclidean distance between vertices v_i and v_j.
    - In our pseudocode this will be represented by **`D[i,j]`**.
  – Let $T[i, j]$ with $1 \leq i < j \leq n$, represent the length of the shortest length triangulation of the polygon $(v_i, v_{i+1}, \ldots, v_j)$.

Shortest Length Triangulation (4)

- Triangulation Recurrence:
  - (To be derived in class)

$$T[i, j] = \min_{i<m<j} \{D(i, j) + T[i, m] + T[m, j]\}$$

  - Base case:
    - The base case corresponds to the degenerate polygon with $j = i + 1$.
    - In this case: $T[i, i+1] = D[i, i+1]$.
  - Order of computation:
    - We can order the computation by the difference $j - i$ (going from the smallest polygons to the largest).

Shortest Length Triangulation (5)

Derivation of the recurrence:

Triangulation Pseudocode

```
// base case:  j=i+1
for i := 1 to n-1 do
   T[i, i+1] := D[i, i+1];
for delta := 2 to n-1 do
   // cases where j-i = delta
   for i := 1 to n-delta do
      j := i+delta;
      T[i, j] := infinity;
      // try all possible triangles v_i, v_j, v_m
      for m := i+1 to j-1 do
         cost := D[i, j] + T[i, m] + T[m, j];
         if cost < T[i, j] then
            T[i, j] := cost;
return T[1, n];        // Computation time Θ(n^3)
```

Recovering Chords in the Triangulation

– Note: We always have two subproblems from which any optimal solution is constructed.

- Let $M[i, j]$ be the vertex m used to compute $T[i, j]$.
  - A small modification will be needed for the previous code.
- Then:

```
function give_solution(i, j)
   output edge (i, j);
   if j > i+1 then
      give_solution(i, M[i,j]);
      give_solution(M[i,j], j);
```

Designing Efficient Algorithms – Summary

- Divide and Conquer
  - Divide the given problem into subproblems, solve them recursively and combine partial solutions.
    - efficient
    - sometimes difficult to implement
    - overhead may be large
  - Main challenge: run time analysis
  - Solved problems:
    - Sorting (merge sort, quick sort) $\Theta(n \log n)$
    - Multiplying large numbers $\Theta(n^{1.58\ldots})$
    - Matrix multiplication (Strassen's algorithm) $\Theta(n^{2.81\ldots})$
    - Closest pair problem $\Theta(n \log n)$
    - Linear-time selection $\Theta(n)$.

Designing Efficient Algorithms – Summary

- Greedy Algorithms
  - Take the locally optimal choice in each step
    - easy to design and implement
    - usually efficient.
  - Caution: often a greedy approach cannot be used.
  - Main challenge: prove correctness.
  - Solved problems:
    - Activity selection: $\Theta(n \log n)$
    - Construction of Huffman trees: $\Theta(n \log n)$
    - Coin changing (Canadian system): $\Theta(m)$.

Designing Efficient Algorithms – Summary

- Dynamic Programming
  - Compute scores for subproblems organized in a large matrix
    - easy to implement.
  - Main challenge: come up with the right subproblem.
  - Solved problems:
    - General coin changing $\Theta(mS)$
    - Longest common subsequence $\Theta(mn)$
    - Knapsack problem $\Theta(nW)$
    - Shortest triangulation $\Theta(n^3)$

Homework: Dynamic Programming for the Al Gore Rhythm

You have been hired as a consultant to help Al Gore promote his campaign against global warming. Mr. Gore has decided to tour various cities in the U.S. in order to lecture on this subject and this tour will take place over n consecutive days. For each of these days, Mr. Gore has prescheduled each meeting site and he knows exactly how many people will show up for each site. We will assume that the i^{th} day has a meeting with $u(i)$ attendees. Although Mr. Gore would like to attend all these meetings a recent operation has weakened him to such an extent that he cannot attend meetings on two consecutive days. (When he does not attend a meeting there will be a free showing of the movie "An Inconvenient Truth").

Design an algorithm that will select the days when Mr. Gore will meet the attendees with the objective that, for the entire tour, he will meet the maximum number of these people. Rephrasing the meeting constraint: If he meets attendees on day i then he cannot meet attendees on day $i-1$.

a) Give a recurrence equation for the objective function $S(i)$ represented as an array. $S(i)$ stores the total number of attendees met during the tour from day 1 up to and including day i. Provide a clear description as to why your recurrence is appropriate.
b) Use pseudo-code to specify the order of evaluation, and where the total number of attendees is found.
c) Use pseudo-code to specify how recovery of the solution (days selected) is accomplished.
d) Specify the execution time.