Dynamic Programming

- Dynamic Programming is similar to Divide & Conquer
  - Both are algorithm design methods
  - Both solve problems by combining solutions to subproblems
- Dynamic Programming is different from Divide & Conquer
  - Dynamic Programming solves optimization problems
    - Goal is to find the best of many possible answers
  - Dynamic Programming is applied when subproblems are not independent
    - "Not independent" means the same subproblem can show up many times
    - Each subproblem is computed just once, then the answer is saved

A Simple Example

```
procedure F(n) returns integer {
    if (n = 0) or (n = 1) return 1;
    return F(n-1) + F(n-2);
}
```

- What does it compute?
- What's wrong with this program?

An Improvement

```
procedure F(n) returns integer {
    R: array[0..n] of integer;
    R[0] = 1; R[1] = 1;
    for i = 2 to n do
        R[i] = R[i-1] + R[i-2];
    return R[n];
}
```

- Instead of recomputing F(5) over and over, we compute it once and remember it
- This is a simple example of Dynamic Programming
- Analysis
  - Time = O(n)
  - Space = O(n)
- Can improve again to use just O(1) space
  - How?

Another Example: Matrix-Chain Multiplication

- Given a sequence of matrices to be multiplied, find the best order to do the multiplications
- Example: (4X2)(2X3)(3X1)
- Matrix properties
  - Matrix multiply is associative: (AB)C = A(BC)
  - But it's not commutative: AB is not necessarily equal to BA (usually just one of these orders even makes sense)
  - It takes mnp multiplies to multiply an (mXn) matrix times an (nXp) matrix

Matrix-Chain Multiplication

- Restatement: Given matrices $M_1, M_2, \ldots, M_n$ where M_i has r_i rows and r_{i+1} columns, find the way to compute the product of the matrices that uses the smallest number of scalar multiplies

A recursive solution:

```
best (i,j):
// Least cost for M[i]...M[j] //
if (i = j) return 0;
min = infinity;
for k = i to j-1 do
    cost = best(i,k) + best(k+1,j)
      + r[i]*r[k+1]*r[j+1];
    if (cost < min) min = cost;
return min;
end best.
```


Dynamic Programming Solution

- Can't afford to keep computing best(i,j) over and over for the same values
- So we store it in an array: C[i,j] is the least cost for multiplying M[i]...M[j]
- Just be careful that each position in C[i,j] is filled before it's used

```
LeastCostMatrixMult() {
for i = 1 to n do C[i,i] = 0;
for size = 1 to n-1 do
   for i = 1 to n-size do
      min = infinity; j = i+size;
      for k = i to j-1 do
         cost = C[i,k] + C[k+1,j]
              + r[i]*r[k+1]*r[j+1];
         if (cost < min) min = cost;
      C[i,j] = min;
return C[1,n].
```

Analysis for Matrix-Chain Multiplication

- Time: $O(n^3)$
- Space: $O(n^2)$
- This algorithm only computed the least cost. How do we remember which multiplies led to the least cost?
- To remember choices:
  - Whenever we assigned a cost to C[i,j], we had a particular k we were using
  - So we use another array Mem[i,j] = k whenever C[i,j] is assigned
  - The sequence of multiplies can be reconstructed starting with the *last* multiply which is held in Mem[1,n]

What to Look For

- Dynamic Programming can be used when:
  - The goal is to find an optimum answer among many possible answers
  - There is a simple recursive algorithm that solves the problem, but it takes too much time because recursive calls are repeated
- Example applications
  - Matrix-chain multiplication
  - Longest common subsequence
  - All-pairs shortest paths in a graph
  - Minimum weight triangulation of a convex polygon
  - Approximate string matching (finding the best match when deletion and substitution are allowed)

Application: Protein Alignment

- Goal: Determine how two protein sequences are related
- Proteins are strings of amino acids
  - There are 20 amino acids
  - DNA is basically a machine for making proteins
  - An amino acid within a protein is often called a *residue*
- Mutations can
  - Replace one amino acid by another
  - Delete an amino acid
  - Add an amino acid
- What is the optimal alignment of two protein sequences?
  - There can be substitutions
  - There can be gaps

Example Alignment

- Consider
  RHYPGDFSPA
  - part of human hemoglobin theta chain

  ARFPADFTAE
  - human zeta chain
- Which is alignment is better?

  RHYPGDFSPA
  ARFPADFTAE

  or

  -RHYPGDFSPA-
  ARF-PADFT-AE

  where "-" denotes a gap caused by deletion or insertion

How to Score Alignments

- "Better" depends on how we want to assign scoring values
- We need scoring values for
  - Identical residues
  - Gaps
  - Residues in same group
    - Polar
    - Hydrophobic
    - Charged
  - Residues in different groups
- We simplify things for our examples
  - Allow all 26 letters, not just the 20 that are used to represent amino acids
  - We assume 2 groups
    - Vowels
    - Consonants
  - Scoring values
    - 4 for identical
    - 1 for same group
    - 0 for different groups
    - -1 for gap

Scoring Example

RHYPGDFSPA
ARFPADFTAE
0114044101 = 16 points

-RHYPGDFSPA-
ARF-PADFT-AE
-41-40441-4- = 18 points

The "-" above represents -1

- Are there other alignments that give the same score or a higher score?
- Can we resolve these questions without having to look at all possible alignments?

Representing Alignments in a Table

- We can encode the scoring information in a table

	R	H	Y	P	G	D	F	S	P	A
A	0	0	1	0	0	0	0	0	0	4
R	4	1	1	1	1	1	1	1	1	0
F	1	1	1	1	1	1	4	1	1	0
P	1	1	1	4	1	1	1	1	4	0
A	0	0	1	0	0	0	0	0	0	4
D	1	1	1	1	1	4	1	1	1	0
F	1	1	1	1	1	1	4	1	1	0
T	1	1	1	1	1	1	1	1	1	0
A	0	0	1	0	0	0	0	0	0	4
E	0	0	1	0	0	0	0	0	0	1

- This shows the score for each possible match of residues in our proteins
- An alignment is a path in this table
  - Path is always down and to-the-right
  - Cutting diagonally through a cell represents alignment of the corresponding residues
  - Moving horizontally or vertically represents a gap

Example Paths

Alignment
RHYPGDFSPA
ARFPADFTAE
corresponds to

	R	H	Y	P	G	D	F	S	P	A
A	0	0	1	0	0	0	0	0	0	4
R	4	1	1	1	1	1	1	1	1	0
F	1	1	1	1	1	1	4	1	1	0
P	1	1	1	4	1	1	1	1	4	0
A	0	0	1	0	0	0	0	0	0	4
D	1	1	1	1	1	4	1	1	1	0
F	1	1	1	1	1	1	4	1	1	0
T	1	1	1	1	1	1	1	1	1	0
A	0	0	1	0	0	0	0	0	0	4
E	0	0	1	0	0	0	0	0	0	1

Alignment
-RHYPGDFSPA-
ARF-PADFT-AE
corresponds to

	R	H	Y	P	G	D	F	S	P	A
A	0	0	1	0	0	0	0	0	0	4
R	4	1	1	1	1	1	1	1	1	0
F	1	1	1	1	1	1	4	1	1	0
P	1	1	1	4	1	1	1	1	4	0
A	0	0	1	0	0	0	0	0	0	4
D	1	1	1	1	1	4	1	1	1	0
F	1	1	1	1	1	1	4	1	1	0
T	1	1	1	1	1	1	1	1	1	0
A	0	0	1	0	0	0	0	0	0	4
E	0	0	1	0	0	0	0	0	0	1

Finding the Best Path in a Table

- Finding the best alignment has become finding the best path through a table
- Position (i,j) in the table can be reached from
  - Position (i,j-1)
    - At cost -1
  - Position (i-1,j)
    - At cost -1
  - Position (i-1,j-1)
    - At cost <table entry>

BestPathCost (table T[1..n,1..n]):
// C[i,j] represents best cost
// to reach position (i,j) in T
C[0,0] = 0
for i = 1 to n do C[i,0] = -i;
for j = 1 to n do C[0,j] = -j;
for i = 1 to n do
  for j = 1 to n do
    C[i,j] = max{ C[i,j-1] - 1,
      C[i-1,j] - 1,
      C[i-1,j-1] + T[i,j] };
return C[n,n];

Protein Alignment in Practice

- Analysis (worst-case)
  - Time $O(n^2)$
  - Space $O(n^2)$
- We don't want just the cost
  - We also want to know the alignment (the path)
  - Can walk "backward" through the table to find it
  - Can use M[i,j] to remember choice made for C[i,j]
- It is possible to bring the space down to O(n) with a more complicated algorithm
  - This includes constructing and reporting the path
- Aligning 2 proteins is easy, but finding the best score when aligning multiple proteins is hard (exponential in number of proteins)
  - NP-complete

Optimal (Minimum Weight) Triangulation of a Polygon

- Goal: Divide the given polygon into triangles in such a way as to minimize the lengths of the added lines
- Application: Has been used for shading in graphics
  - Intuition: Min weight triangulation should be good surface approximation since triangles are "small"
  - Aside: Delaunay triangulation may be better choice
- Definitions:
  - Vertices are $v_1,...,v_n$ presented in counterclockwise order
  - D[i,j] holds the distance from v_i to v_j
  - C[i,j] represents min cost for triangulating polygon $v_i,...,v_j$
  - C_k[i,j] represents cost of optimal triangulation of polygon $v_i,...,v_j$ *assuming* that triangle v_i v_k v_j is used

Triangulation Observations

- Note that *mostly* C_k[1,n] = C[1,k]+C[k,n]+D[1,k]+D[k,n]
- But what about when k = 2?
  - D[1,2] appears in the formula, but it's not part of the triangulation
- Trick: Define C[1,2] = -D[1,2]
  In general: C[i,i+1] = -D[i,i+1]
  - Terms cancel to provide correct result
  - With this trick, formula above holds for all k
- Note that
  C[i,j] = $\min_k$ C_k[i,j]
- Agorithmic idea:
  - Determine C[i,j] for all i and j
  - Final result is held in C[1,n]
- What order should we use to fill in C[i,j]?
  - 3 adjacent vertices, then
  - 4 adjacent vertices, then
  - 5 adjacent vertices, then...

Dynamic Programming Algorithm for Minimum Weight Triangulation

```
minWeight ():
  // D[i,j] is distance from v[i] to v[j]
  for i = 1 to n do
    C[i,i+1] = -D[i,i+1];
  for s = 3 to n do   // Problem size
    for i = 1 to n-s+1  //Start vertex
      j = i + s - 1      // End vertex
      C[i,j] = min_{i<k<j} {
          C[i,k]+C[k,j]+D[i,k]+D[k,j]
          }
  return C[1,n].
```

- Analysis
  - Time = $O(n^3)$
  - Space = $O(n^2)$
- To remember the triangulation
  - Set Mem[i,j] to k whenever C[i,j] is set