Dynamic Programming

- An Algorithm Design Technique
- A framework to solve Optimization problems
  - Elements of Dynamic Programming
  - Dynamic programming version of a recursive algorithm
  - Developing a Dynamic Programming Algorithm
  - Multiplying a Sequence of Matrices

A framework to solve Optimization problems

- For each current choice:
  - Determine what subproblem(s) would remain if this choice were made.
  - Recursively find the optimal costs of those subproblems.
  - Combine those costs with the cost of the current choice itself to obtain an overall cost for this choice
- Select a current choice that produced the minimum overall cost.

Elements of Dynamic Programming

- Constructing solution to a problem by building it up dynamically from solutions to smaller (or simpler) subproblems
  - sub-instances are combined to obtain sub-instances of increasing size, until finally arriving at the solution of the original instance.
  - make a choice at each step, but the choice may depend on the solutions to sub-problems
- Principle of optimality
  - the optimal solution to any nontrivial instance of a problem is a combination of optimal solutions to some of its sub-instances.
- Memoization (for overlapping sub-problems)
  - avoid calculating the same thing twice,
  - usually by keeping a table of known results that fills up as sub-instances are solved.

Memoization for Dynamic programming version of a recursive algorithm e.g.

- Trade space for speed by storing solutions to subproblems rather than re-computing them.
- As solutions are found for subproblems, they are recorded in a dictionary, say soln.
  - Before any recursive call, say on subproblem Q, check the dictionary soln to see if a solution for Q has been stored.
    - If no solution has been stored, go ahead with recursive call.
    - If a solution has been stored for Q, retrieve the stored solution, and do not make the recursive call.
  - Just before returning the solution, store it in the dictionary soln.

Dynamic programming version of the fib.

```pseudocode
fibDPwrap(n)
Dict soln = create();
return fibDP(soln, n);
fibDP(soln, k)
int fib, f1, f2;
if (k < 2) fib = k;
else if (member(soln, k-1) == false)
f1 = fibDP(soln, k-1);
else
f1 = retrieve(soln, k-1);
if (member(soln, k-2) == false)
f2 = fibDP(soln, k-2);
else
f2 = retrieve(soln, k-2);
fib = f1 + f2;
store(soln, k, fib);
return fib;
```

Development of a dynamic programming algorithm

- Characterize the structure of an optimal solution
  - Breaking a problem into sub-problem
  - Whether principle of optimality apply
- Recursively define the value of an optimal solution
  - Define the value of an optimal solution based on value of solutions to sub-problems
- Compute the value of an optimal solution in a bottom-up fashion
  - Compute in a bottom-up fashion and save the values along the way
  - Later steps use the save values of previous steps
- Construct an optimal solution from computed information
Dynamic programming, e.g.

- Problem: Matrix-chain multiplication
  - a chain of \(<A_1, A_2, \ldots, A_n>\) of n matrices
  - find a way that minimizes the number of scalar multiplications to compute the product \(A_1A_2\ldots A_n\)

- Strategy:
  - Breaking a problem into sub-problem
    - \(A_1A_2\ldots A_kA_{k+1}A_{k+2}\ldots A_n\)
  - Recursively define the value of an optimal solution
    - \(m[i,j] = 0\) if \(i = j\)
    - \(m[i,j] = \min\{i<=k<j\} (m[i,k]+m[k+1,j]+p_{i-1}p_kp_j)\)
    - for \(1 \leq i \leq j \leq n\)

bottom-up approach

- MatrixChainOrder(n)
  - for \(i=1 \text{ to } n\)
    - \(m[i,i] = 0\)
  - for \(1 \leq l \leq n\)
    - for \(i = 1 \text{ to } n-l+1\)
      - \(j = i+l-1\)
      - \(m[i,j] = \infty\)
      - for \(k=i \text{ to } j-1\)
        - \(q = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\)
        - if \(q < m[i,j]\)
          - \(m[i,j] = q\)
          - \(s[i,j] = k\)
  - //At each step, the \(m[i, j]\) cost computed depends only on table entries \(m[k,k]\) and \(m[k+1, j]\) already computed

Construct an optimal solution from computed information

- MatrixChainMult(A, s, i, j)
  - if \(j-i\)
    - \(x = \text{MatrixChainMult}(A, s, i, s[i,j])\)
    - \(y = \text{MatrixChainMult}(A, s, s[i,j]+1, j)\)
    - return matrixMult(x, y)
  - else return \(A_i\)

- Analysis:
  - Time \(\Omega(n^3)\) space \(\Theta(n^2)\)
  - Comparing to Time \(\Omega(4^n/n^{1.5})\) by brute-force exhaustive search.

- >> see Introduction to Algorithms