**Chapter 15:**

Dynamic Programming solves problems by combining the solution to subproblems.

* Define *Programming* as a tabular method.
* Subproblems overlap (share subsubproblems)
  + Difference from *Divide-and-Conquer* algorithms.
* Subsubproblems are solved once and saved in a table, avoiding repetition.

Dynamic Programming is applied to *optimization problems*.

* Problems with many possible values, many can be optimal.

4 steps for developing a Dynamic Programming algorithm:

* Characterize structure of an optimal solution
* Recursively define value of an optimal solution
* Compute value of an optimal solution (in bottom-up fashion)
* Construct an optimal solution from Step 3.

Dynamic programming is a time-memory trade-off. Exponential run time can be reduced to polynomial – this happens when distinct subproblems are polynomial in number and can be solved in polynomial time.

Implementations:

* Top-down with memoization.
  + Check first to see if subproblem is done from a table or array.
* Bottom-up.
  + Subproblems depend on smaller subproblems.
  + Sort the subproblems by size and start with the smallest. Thus, all of the prerequisites of each step are solved.

**Rod Cutting**:

Optimize the price of selling steel rods by cutting the up in different ways or not before selling.

* 2^(n-1) ways to cut up a rod of length n.
* Store all possible cut options in table for each length, and keep track of the max. Add all the maxes together to show maximum revenue.

This problem exhibits *Optimal Substructure*.

* *Optimal Substructure* is when optimal solutions to a problem are built from optimal solutions to related subproblems than can be solved independently.

Recursively (top-down implementation without memoization):

* The related subproblem is dividing the remainder of a rod after the first cut.
* One recursive line computes the maximum revenue:
  + Q = max(-infinity, p[i] + recurse(p,n-i) for i = 1 to n
* This method creates a huge stack frame, and the run time double for each increment of n by 1.
* The run time is exponential in n.

Clearly, dynamic programming would reduce the runtime of this algorithm because the recursion does many divisions repeatedly.

Top-down with memoization:

* First check the table, r, to see if the revenue at this step has been calculated.
* Now the recursive line is:
  + Q = max(-infinity, p[i]+memo\_recursion(p,n-I,r)) for i=1 to n
* Then return the last entry of the table after using the pervious entries to calculate it.

Bottom-up with memoization:

* Create a blank array with 0 at the beginning.
* The recursive line is:
  + Q = max(-infinity, p[i] + r[j-i]) for j=1 to n, for i = 1 to j
* Return the last entry.

**Matrix-chain Multiplication**:

Given chain {A1,A2,…,An} of n matrices to be multiplied, compute the product.

The subroutine is multiplying pairs of matrices once they are parenthesized.

* Associativity means all parenthesizations have the same product.
* Fully Parenthesized if single matrix or product of of Fully Parenthesized matrix products in parentheses.
* Example A1A2A3A4
  + (A1(A2(A3A4)))

(A1((A2A3)A4))

((A1A2)(A3A4))

((A1(A2A3))A4)

(((A1A2)A3)A4)

Multiplying two matrices (AB = C): (A=p x q, B=q x r, C=p x r),

if cols(A) != rows(B), return incompatible multiplication

else C = rows(A) x cols(B) new matrix

for i = 1 to rows(A)

for j = 1 to cols(B)

Cij = 0

for k = 1 to cols(A)

Cij = Cij + aik\*bjk (this happened pqr times) (line 8)

return C

Parenthesization is important example:

* ABC, A = 10x100, B = 100x5, C=5x50
  + ((AB)C) line 8 occurs 7500 times
  + (A(BC)) line 8 occurs 75000 times (10x more)

**The Matrix-Chain Multiplication Problem:**

* Given n matrices, An, in a chain where Ai has dimensions pi-1xpi, fully parenthesize the the product in a way that minimizes number of scalar multiplications (line 8).
  + Determine an order for multiplication before doing it.
  + The cost-benefit of this is typically negative (as in the example above it is not worth changing).
* Counting the number of parenthesizations:
  + Cost-benefit analysis for the notion above.
  + Number of parenthesizations P(n) of n matrices is 1 if there is only 1 matrix, or the sum from k=1 to n-1 of P(k)P(n-k) if there is more than 1 matrix.
* Applying 4 dynamic programming steps:
  + To characterize the structure of an optimal solution, consider i<j, and starting point k greater than or equal to i. The first product is Ai…k x Ak+1…j = Ai…j. This is proven to be an optimal parenthesization.
  + Next, optimize subproblems by finding minimum cost, m[i,j], of parenthesizing Ai…j. Recursively for 1<=i<=j<=n, m[i,j] = cost of computing subproducts is Ai…k and Ak+1…j and multiplying them together. Computing Ai…kAk+1…j takes Pi-1PkPj scalar multiplications. Thus, m[i,j] = 0 if i=j, and min{m[i,k] + m[k+1,j] + Pi-1PkPj} if i<j for i<=k<j. s[i,j] is the value of k where this equation is optimized.
  + Recurrence recursion is still exponential in this case unlike the rod cutting problem. Computing with optimal costs is don’t my modifying the recursive equation for m[i,j] to be tabular bottom-up. Matrix-Chain-Order algorithm does this. Fill in the table m in a manner that corresponds to solving the parenthesization problem for chains of increasing length. For chain AiAi+1…Aj, the subproblem size is the length j-i-1 of the chain.

Mathrix\_Chain\_Order(p): (note Pi-1, Pk and Pj are dimensions):

1. N = p.length – 1

2. let m[1..n,1..n] and s[1..n-1,2..n] be new tables

3. for i = 1 to n

4. m[i,i] = 0

5. for l = 2 to n

6. for i = 1 to n - l + 1

7. j = i + l – 1

8. m[i,j] = infinity

9. for k = I to j-1

10. q = m[i,k] + m[k+1,j] + Pi-1PkPj

11. if q<m[i,j]

12. m[i,j] = q

13. s[i,j] = k

14. return m and s

See page 376 of textbook for example of output for a chain of length 6. The pyramid is built bottom up with a stencil of the left and right entries below it. It is an upper triangular table. Thus, the top right corner is optimal solution. The run time of this algorithm is O(n^3), a polynomial much better than exponential on n. n^2 space is required to store m and s. This proves its efficiency.

* + Now that the number of scalar multiplications has been determined, we need to find out how to do them with the table s. The final multiplication in computing A1…n is A1…s[1,n] x As[1,n]+1…n. s[1,s[1,n]] is the last multiplication of A1…s[1,n] and s[s[1,n]+1,n] is the last multiplication of As[1,n]+1…n. Thus a recursive algorithm for placing the parentheses is:

Print\_Optimal\_Parens(s,i,j):

1. if i==j

2. print(”Ai”)

3. else print(“(“)

4. Print\_Optimal\_Parens(s,i,s[i,j])

5. Print\_Optimal\_Parens(s,s[i,j]+1,j)

6. print(“)”)

**Eddy Diagrams (Information provided by Dr. Rajopadhye via email)**:  
1. A solid horizontal straight line represents a sequence; we have two sequences  
drawn as two parallel horizontal lines.

2. A solid curved line between two points in the same sequence is an arc; all arcs  
are either above the upper sequence, or below the lower one.

3. A solid line between two points in different sequences is a bond; bonds are  
between the two sequences.

4. A dotted curved line between two points in the same sequence means that  
those two points do not form an arc.

5. Similarly, a dotted line between two points in different sequences means that  
those two points do not form a bond.

6. A dashed curved line between two points in the same sequence denotes either  
2 or 4.

7. A dashed line between two points in different sequences denotes either 3 or 5

8. When there are no additional choices of bonds/arcs in a given region, we label  
it with a color (cyan); no arc or bond crosses a labeled region.

9. A gray region can only have arcs (no bonds touch any point within it.  
10. A point in a sequence may be labeled with an index, and in general, the set  
of such indices are free variables used in the recursions; the index of unlabeled points before (after) labeled points is assumed to be the predecessor (successor) of the label.

The Eddy diagram to compute a certain function is written like a formal (context-free) grammar. The left hand side is labelled with problem specifics [This is to be completed].

**Assignment (given in previous meeting):**

The Roofline Model has a flat portion and an oblique portion. A kernel is represented by a vertical line with an x marking exact performance on the line.

Moving the x left or right represents a decrease or increase in operational intensity, floating point operations per byte, respectively. This is done with memory management optimization within the cache and pertains to cache size and prefetching. If the kernel is under the flat roof for this movement, moving left takes it closer to a memory bound problem, and right takes it further away from a memory bound problem. If the kernel is under the oblique roof for this movement, moving left represents potential to solve memory bound optimizations and move to more complicated memory related roofs, and moving right represents regressing in memory optimization to invite significance in repeating previous optimizations.

Moving the x up represents increasing attainable floating point performance through computational optimization. This is done through floating point balance and the implementation of SSE intrinsics relating high level programming code to the compiler and with floating point balance. Under both the flat and the oblique part of the roof, moving upward can encounter both memory and computational ceilings; however, under the flat part, if the x is far from the ridge point it is more likely to reach a computational roof. Under the oblique part, far from the ridge point, it is more likely to experience a memory roof. (Our combined answer in different document, very similar)

From the application of dynamic programming step 3, the matrix chain ordering algorithm is O(n^3). The operational intensity of the matrix chain ordering algorithm is a ratio of the floating point operations to the reads and writes. There are 6 accesses in line 10, so the denominator is 6 accesses. There are 4 floating point operations in line 10 as the index operations are integer operations. Thus, the operational intensity is 4 flops per 6 accesses.

The space complexity of m[i,j] and s[i,j] together is n^2 in the ordering algorithm.

Eddie Diagrams will be drawn by hand.

Solid = connection

Dotted = no connection

Dashed = possible (will change to above eventually)

Now more of the chapter after the matrix chain product introduction.

**Elements of Dynamic Programming**:

An optimization problem must have optimal substructure and overlapping subproblems in order for dynamic programming to apply.

* Memoization helps with top-down recursion implementation.

Optimal substructure is found when there are optimal solutions to subproblems.

* For the MCP, splitting the product Ai..Aj between Ak and Ak+1 contains the optimal solution for parenthesizing Ai..Ak and Ak+1..Aj

Pattern for discovering optimal substructure:

* Show that one choice leads to a set of subproblems such as the first split in parenthesization.
* Assume this choice leads to an optimal solution.
* Characterize the space of resulting subproblems. Start simple and expand as necessary.
* Show subproblem solutions are optimal solutions with cut-and-paste technique. Cut out the non-optimal solutions, and paste in the optimal ones proves by contradiction solutions are optimal with the first assumption.

Optimal substructure variance across problem domains. These also determines runtime of a dynamic programming algorithm:

* Number of subproblems the optimal solution to the original problem uses.
* Number of choices in determining which subproblems to use.
* Rod cutting problem uses 1 subproblem of size n-i and there are n choices for i that might yield the optimal solution. O(n^2).
* Matrix-chain multiplication problem has 2 subproblems and j-i choices. The problems are optimally parenthesizing Ai..Ak and Ak+1..Aj, and there are j-i choices for k that might yield the optimal solution. O(n^3).

Run time can be shown graphically with edges and vertices representing subproblems and choices.

The cost of a problem solution is the cost of the subproblem and the cost of making the choice for it. In MCP, the choice is Ak, and its cost is expressed by Pi-1PkPj, the dimensions.

Rather than making the best choice of subproblems which is the definition of dynamic programming, we can make a greedy choice, the best that looks good right now, and go from there. This called a greedy algorithm.

**Subtleties:**

Consider directed graph G=(V,E) with vertices u,v in V.

* **Unweighted shortest path**: a path from u to v consisting of the fewest edges that do not have weights. Must not contain cycles (simple). This is solved with the breadth-first search algorithm. Exhibits optimal substructure – consider path p and a vertex w that is between or equal to u or v; u->(p)v becomes u->(p1)w->(p2)v; the edges in the p is equal to the sum of edges in p1 and p2; *cut-and-paste* proof – u->w is optimal because if it were not, then p would not be optimal which is a required assumption, the same applies for w->v. Thus the optimal substructure is to consider immediate vertices w.
  + Subproblems do not share resources – proved by the notion that p1+p2 = p and p is the shortest path.
* **Unweighted longest simple path**: a path from u to v consisting of the most edges without weights and no cycles (simple). This does not have the same substructure as unweighted shortest path. Consider 2 independent points between u and v essentially forming a diamond so that there is a cycle. A subproblem choice at w is contradicted by the cut-and-paste proof as once this choice is made, the correct longest path is to back track, not simple, a contradiction. Thus this problem lacks optimal substructure and a solution cannot necessarily be built from solutions to subproblems.
  + This problem is NP-complete meaning it is unlikely to ever be solved in polynomial time.
  + Harder than shortest path because subproblems are *not independent* here. The solution to one subproblem affects the next and possible others. We must not use the end vertex until the last step, and we must use as many vertices as possible without creating a cycle or making the graph not simple.
  + Resources used in one subproblem are unavailable in others.

The matrix-chain multiplication problem exhibits independent subproblems as no matrix can appear in two parenthesizations. Basic conservation of matter shows rod cutting has independent subproblems.

**Overlapping subproblems:**

The space of subproblems should be small - The recursion should do the same thing, not generate new subproblems every time. Number of subproblems should be polynomial in input size. When the same problem is revisited repeatedly in recursion, the subproblems are overlapping.

* Divide-and-conquer is the opposite.
* Overlapping invites memoization
* MCP repeatedly looks up lower rows to build higher rows as mentioned in its triangular table.