Cormen Ch 15 - Dynamic Programming

- "programming" in dynamic programming means tabular method (i.e. of or relating to the use of a table) not to writing code
- much like divide-and-conquer, but applies when the subproblems overlap
- typically used for optimization problems
- procedure:
    i. characterize the structure of an optimal solution
    ii. recursively define the value of an optimal solution
    iii. compute the value of an optimal solution -> typically bottom up
    iv. construct an optimal solution from computed information

15.1 - Rod cutting

- given a rod of length n and a table of prices, p sub i for i = 1, 2, ..., n, obtain max revenue obtained by cutting the rod and selling the pieces
- can cut 2^n-1 ways
- the problem has an optimal substructure
- input:
    - p[1...n] -> prices for sizes
    - n -> size of rod
    - returns maximum value possible for rod size
- recursive top-down:

```
cut_rod(p, n):
    if n == 0
        return 0
    q = -inf
    for i == 1 to n
        q = max(q, p[i] + cut_rod(p, n - i))
    return q
```

- inefficient because it solve the same problems over and over again
- recursion tree

```
{
    4: {
        3: {
            2: {
                1: {
                    0: null
```

```
                },
                0: null
            },
            1: {
                0: null
            },
            0: null
        },
        2: {
            1: {
                0: null
            },
            0: null
        },
        1: {
            0: null
        },
        0: null
    }
}
```

- dynamic programming for rod cutting (top-down with memoization)

```
memoized_cut_rod(p, n):
    let r[0...n]
    for i = 0 to n
        r[i] = -inf
    return memoized_cut_rod_aux(p, n, r)

memoized_cut_rod_aux(p, n, r):
    if r[n] >= 0
        return r[n]
    if n == 0
        q = 0
    else q = -inf
        for i = 1 to n
            q = max(q, p[i] + memoized_cut_rod_aux(p, n - 1, r))
    r[n] = q
    return q
```

- bottom-up dynamic cut rod

```
bottom_up_cut_rod(p, n):
    let r[0...n]
    r[0] = 0
    for j = 1 to n
        q = -inf
        for i = 1 to j
            q = max(q, p[i] + r[j - 1])
```

```
        r[j] = q
    return r[n]
```

- subproblems form a graph, the subproblem graph, that embodies the relationship between subproblems, with a directed edge from one subproblem to another if the initial is dependent on the subsequent
- the size of the subproblem graph can help determine the running time
- can extend the dynamic programming example to include both a value and the choice that led to that value (or a set of all permutations of possible choices)

```
extended_bottom_up_cut_rod(p, n):
    let r[0...n] and s[1...n]
    r[0] = 0
    for j = 1 to n
        q = -inf
        for i = 1 to j
            if q < p[i] + r[j - 1]
                q = p[i] + r[j - i]
                s[j] = i
        r[j] = q
    return r and s
```

```
print_cut_rod_solution(p, n):
    (r, s) = extended_bottom_up_cut_rod(p, n)
    while n > 0
        print s[n]
        n = n - s[n]
```

15.2 Matrix-chain multiplication

- fully parenthesized - matrix multiplication string that parenthesizes all pairs so each is of the form (A * B) or A * ()/() * A, i.e. there is only one pair that is of the for (A * B) and the rest are of the second form

```
matrix_multiply(A, B)
    if A.columns != B.rows
        error "incompatible dimensions"
    else let C be a new A.rows x B.columns matrix
        for i = 1 to A.rows
            for j = 1 to B.columns
                c sub ij = 0
                for k = 1 to A.columns
```

```
                c sub ij = s sub ij + a sub ik * b sub kj
    return C
```

- matrix-chain multiplication problem:
  - given a chain of n matrices, fully parenthesize the product in a way that minimizes the number of scalar multiplications
- checking all parenthesizations does not yield an efficient algorithm
  - characterized by recurrence:

```
P(n) = { 1 if n = 1,
         Sum from k = 1 to n − 1 of P(k)*P(n − k) if n >= 2}
```

- similar to Catalan numbers
- dynamic programming:
  i. substructure of optimal parenthesization:
     - the optimal parenthesization of a prefix subchain of matrices is a component of the solution

  ii. recursive solution

```
m[i, j] = { 0 if i = j,
            min as varies i<=k<j of {m[i, k] + m[k + 1, j] + p sub i − 1 * p sub k
```

```
3. computing optimal costs
```

```
matrix_chain_order(p)
    n = p.length − 1
    let m[1...n, 1...n] and s[1...n − 1, 2...n] (tables)
    for i = 1 to n
        m[i, i] = 0
    for l = 2 to n
        for i = 1 to n − l + 1
            j = i + l − 1
            m[i, j] = inf
            for k = i to j − 1
                q = m[i, k] + m[k + 1, j] + p sub i − 1 * p sub k * p sub j
                if q < m[i, j]
                    m[i, j] = q
                    s[i, j[ = k
    return m and s
```

```
4. constructing an optimal solution
```

```
print_optimal_parens(s, i, j)
    if i == j
        print "A" sub i
```

```
    else print "("
        print_optimal_parens(s, i, s[i, j])
        print_optimal_parens(s, s[i, j] + 1, j)
        print ")"
```

15.3 Elements of dynamic programming

- optimal substructure: an optimal solution to a problem contains within it optimal solutions to subproblems
- pattern:
  1. show a solution consists of making a choice which leaves one or more subproblems to be solved
  2. make the assumption that the choice leads to an optimal solution
  3. this assumption allows for determination of the resulting space of of subproblems
  4. show that solutions to subproblems within an optimal solution to the problem must be optimal solutions to subproblems by assuming they aren't and showing a contradiction
- overlapping subproblems*****:
  - recursive algorithm revisits the same problem repeatedly
  - NOTE: this is why dynamic programming uses tables -> store results of repeated subproblems
- memoization:
  - maintain a table of the subproblem solutions and allow control structure to work recursively
  - uses a sentinal value to indicate a solution has not yet been determined for a subproblem