Dynamic Programming

Overview

- Powerful design technique
- Invented by Richard Bellman
- Intended for optimization problems
  - e.g 0-1 knapsack
- Can be thought of as an
  - Exhaustive search done intelligently

The idea

- Divide the problem into subproblems
  - Can be overlapping
- Solve subproblems
- Reuse the solutions to the subproblems
  - Store intermediate results
- Solve original problem

Nth Fibonacci Number

- Find n^{th} Fibonacci Number
- Usual Recursive Solution
  - recfib(n)

```
{
        if ( n==0 )
                return 0;
        if ( n<=2 )
                return 1;
        return (recfib(n-1) + recfib(n-2) );
}
```

- T(n) = T(n-1)+T(n-2) + 2// does unnecessary extra work
  - Exponential cost

Nth Fibonacci Number

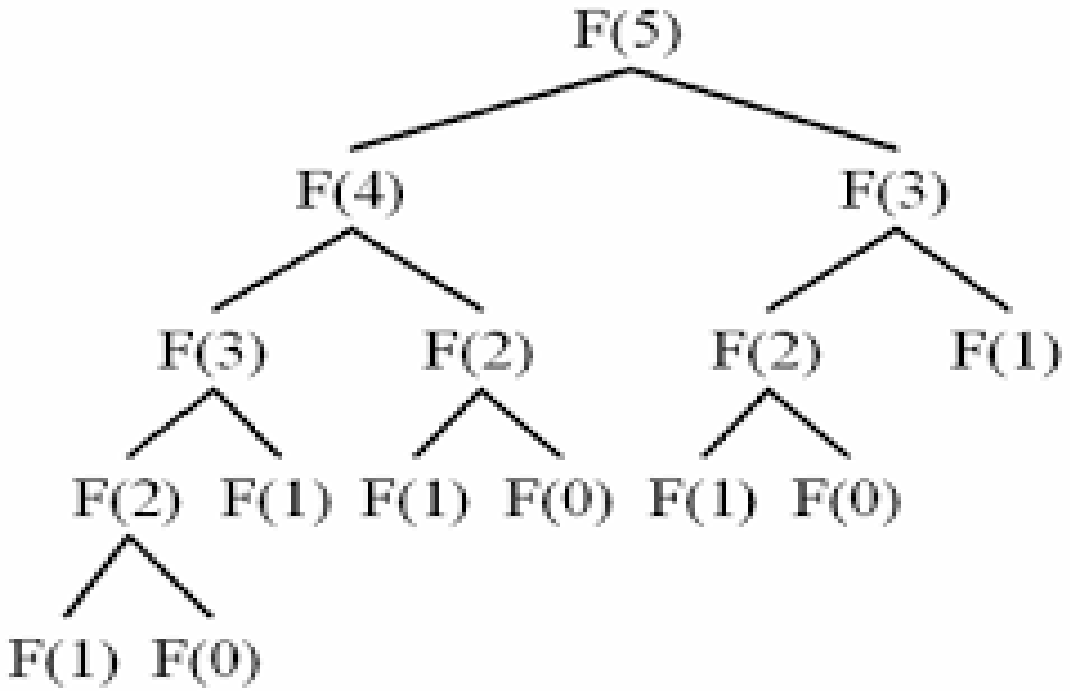
- Find n^{th} Fibonacci Number
- Usual Recursive Solution
  - recfib(n)

```
{
        if ( n==0 )
                return 0;
        if ( n<=2 )
                return 1;
        return (recfib(n-1) + recfib(n-2) );
}
```

- T(n) = T(n-1)+T(n-2) + 2// does unnecessary extra work
  - Exponential cost

Flaw in Recursion

- Same subproblem is being recomputed repeatedly
- Can be avoided by storing results of subproblems and reusing it



Src: http://www.ics.uci.edu/~eppstein/161/960109.html

Memoized Solution

- Memoization: Store intermediate results so as to reuse
- Memoized Algorithm

```
memo = {};
mfib(n):
    if n in memo: return memo[n]
    if n<=2: f=1
    else: f=mfib(n-1)+mfib(n-2)
    memo[n]=f
    return f
```

Analysis

- The recursion call invoked atmost n times
  - Non recursive work per call $\Theta(1)$
  - Cost $\Theta(n)$

Bottom-up Approach

- Use an iterative method

```
dynamic_fib(n){
    int f[n+1];
    f[1]=f[2]=1;
    for (int i=3; i<=n;i++)
        f[i]=f[i-1]+f[i-2];
    return f[n];
    }
```

Cost is O(n) (both time and space)

Requirements

- Simple Subproblem : Must be able to divide problems into simple subproblems each similar to the original
  - Usually there are polynomial number of subproblems
- Subproblem Optimality: optimal solution to global problem must be composition of optimal subproblem solutions using simple combining operation
- Subproblem Overlap: Optimal solutions to unrelated subproblems can contain subproblems in common

5 Easy Steps

- Define subproblems
- Guess part of the solution
- Relate subproblems
- Recurse and Memoize or build DP table
- Solve final problem

Matrix Chain Multiplication

- Given n two dimensional matrices whose product we wish to compute
  - $A = A_0.A_1.A_2....A_{n-1}$
  - A_i is d_i X d_{i+1} matrix, for i = 0, 1, 2 ..n-1
  - Use standard matrix multiplication algorithm
- Matrix multiplication associative
  - B.(C.D) = (B.C).D
  - We can paranthesize A anyway we wish

Impact of Parenthesization

- Parenthesization impacts the number of scalar multiplications done
- E.g B is a 2 x 10 matrix, C is a 10 x 50 matrix, D a 50 x 20 matrix
  - B.(C.D) requires 2.10.20 + 10.50.20 = 10400
  - (B.C).D requires 2.10.50 + 2.50.20 = 3000
- Paranthezation matters!!!!
- What is the optimal paranthization that minimizes total number of scalar multiplications?

Brute Force

- Enumerate all parenthization and choose best
  - Calculate multiplications done by each
- Complexity
  - Number of all parenthizations is set of all binary trees that have n external nodes
  - Exponential in n
  - nth Catalan Number, = $\Omega(4^n/n^{3/2})$

DP Strategy

- Identify Subproblem
  - Compute best parenthization for some sub-expression
    - $A_i.A_{i+1}.....A_j$
  - $N_{i,j}$, the minimum number of multiplications needed to compute subexpression
  - Original problem characterized as finding $N_{0,n-1}$
  - Number of subproblems $\Theta(n^2)$

DP Strategy

- Guess: Outermost or last multiplication to be performed
  - $(A_i...A_k).(A_{k+1}....A_j)$
- Relate subproblems
  - Try all possibilities for k to get best solution
    - Number of choices = O(j-i+1) = O(n)
  - Irrespective of value of k
    - $(A_i...A_k)$ and $(A_{k+1}....A_j)$ must be solved optimally
- Recursion

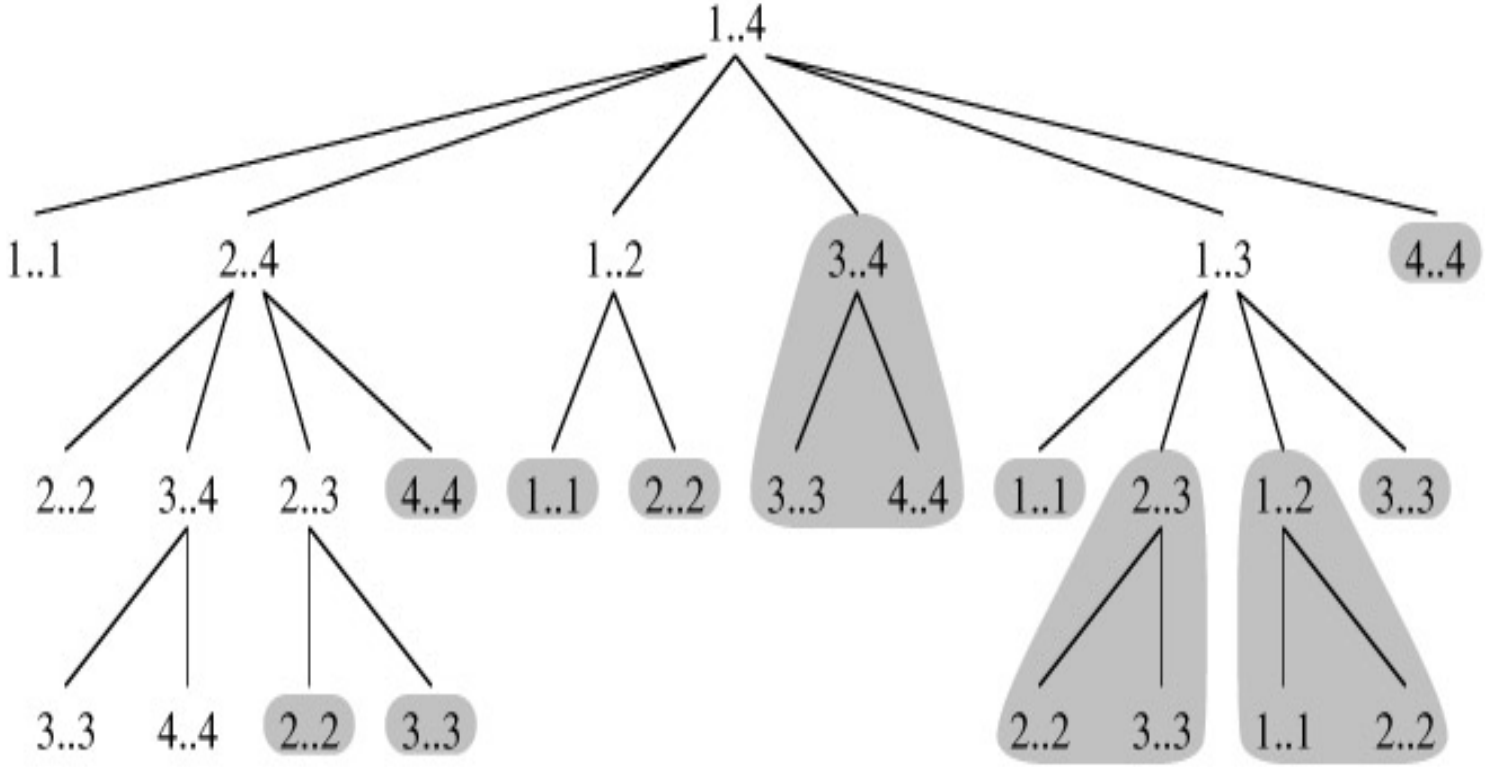
$$N_{i,j} = min_{i \le k < j} N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}$$

Recursive Algorithm

```
RecursiveMCP(int d[], int i, int j){
        if (i == j) return 0;
        int min = Integer.MAX_VALUE;
        for (int k=i; k<j; k++)        {
            int count = RecursiveMCP(d, i, k) +
                        RecursiveMCP(d, k+1, j) +
                        d[i]*d[k+1]*d[j+1];
            if (count < min)
                        min = count;
        }
        return min;
    }
```

DP Strategy

- Recursion results in unnecessary extra computations
  - Memoization can help



Src: http://www.geeksforgeeks.org/dynamic-programming-set-8-matrix-chain-multiplication/

DP Strategy

- Memoize
  - Create a 2d matrix m to store intermediate results
  - N[i,j] represents the cost of multiplying the maxtrices $A_i..A_j$
- Solve final problem
  - Entry N[0,n-1] gives the total cost of multiplying the matrices
  - Optimal parenthesization can be determined by backtracking through table

Recursion with Memoization

- MemoizedMCP(int d[], int i, int j){

```
if N[i,j] != null return N[i,j];
if (i == j) return N[i,i] = 0;
N[i,j] = infinity;
for (int k=i; k<j; k++)        {
    int count = MemoizedMCP(d, i, k) +
                MemoizedMCP(d, k+1, j) +
                d[i]*d[k+1]*d[j+1];
    if (count < N[i,j])
                 N[i,j] = count;        }
return N[i,j];     }
```

- Complexity $O(n^3)$

Bottom-up Approach

Initialize $N_{i,i}$ to 0 for i ← 0 to n-1

for b ← 1 to n-1

  for i ← 0 to n-b-1

    Set j ← i+b

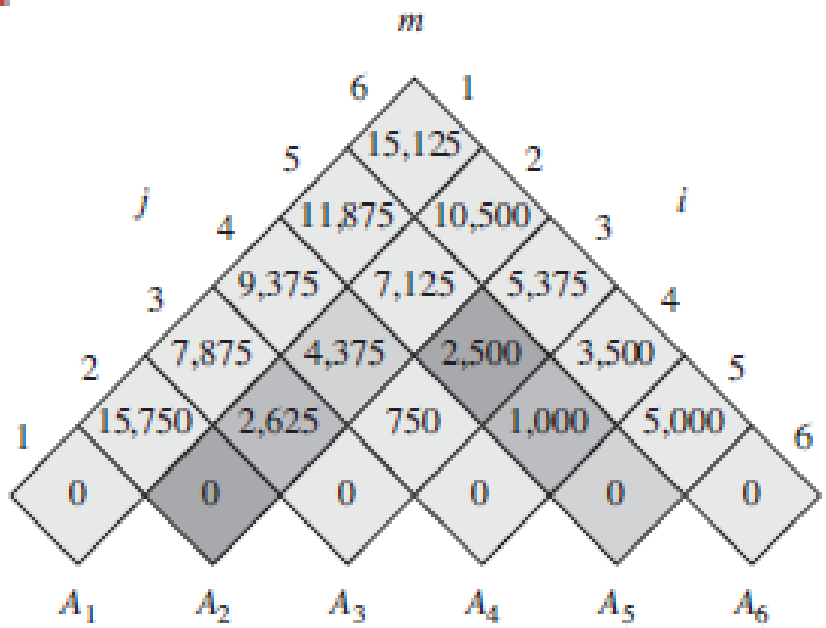
    Set $N_{i,j}$ to +infinity

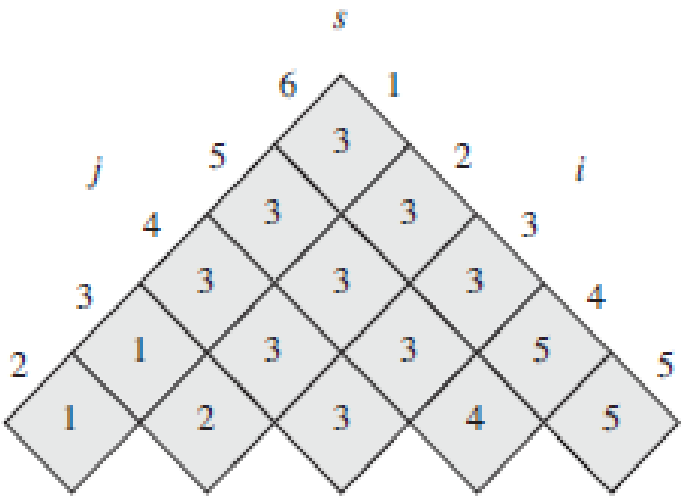
    for k ← i to j-1 do

$$N_{i,j} = \min\left\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\right\}$$

Example: CLR pg 276

- N=6 and A1:30x35, A2:35x15, A3:15x5, A4:5x10, A5:10x20, A6:20x25





$$N_{2,5} = min \begin{cases} N_{2,2} + N_{3,5} + d_1 d_2 d_5 = 0 + 2500 + 35 . 15 . 20 = 13{,}000 \\ N_{2,3} + N_{4,5} + d_1 d_3 d_5 = 2625 + 1000 + 35 . 5x . 20 = 7125 \\ N_{2,4} + N_{5,5} + d_1 d_4 d_5 = 4375 + 0 + 35 . 10 . 20 = 11{,}375 \end{cases}$$

Bottom-up solution outline

- First compute $N_{i,i} = 0$ for i = 1 , 2 , ... , n
- Then compute $N_{i,i+1}$ for i = 1 , 2 , ... , n-1 and so on
  - The $N_{i,j}$ computed depends on $N_{i,k}$ and $N_{k+1,j}$ already computed
- Using this layout, the minimum cost $N_{i,j}$ for multiplying a subchain $A_iA_{i+1}. . . A_j$ of matrices
  - Use intersection of lines running northeast from A_i and northwest from A_j
- Each entry $s_{i,j}$ records a value of k such that an optimal parenthesization of $A_iA_{i+1}. . . A_j$ splits the product between A_k and A_{k+1}

Generalization

- Given a linear sequence of objects, an associative binary operation on those objects, and a way to compute the cost of performing that operation on any two given objects (as well as all partial results), compute the minimum cost way to group the objects to apply the operation over the sequence
- Subproblems in Strings
  - Prefix A[:i]
  - Postfix A[i:]
  - Substring A[i:j]

Problem

- Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is

〈 5, 10, 3, 12, 5, 50 ,6 〉

Longest Common Subsequence

- Given a string X of size n, a subsequence of X is any string
  - sequence of characters that are not necessarily contiguous but taken in order from X
    - e.g AAAG is subsequence of CGATAATTGAGA
  - of form $X[i_1]X[i_2]...X[i_k]$, $i_j<i_{j+1}$ for $j=1,...,k$
- Longest Common Subsequence (LCS) problem
  - Given two character strings, X of size n and Y of size m, over some alphabet find
  - A longest sequence S that is a subsequence of both X and Y

Brute Force

- Enumerate all possible common subsequences of X
  - Take the longest one that is also a subsequence of Y
  - Potentially 2^n different subsequences of X, each of which requires O(m) time to determine if it is a subsequence of Y
- Exponential algorithm
  - $O(2^n m)$
  - Use dynamic programming method

Dynamic Programming Strategy

- Identifying subproblem
  - Computing the length of longest common subsequence of prefix of X and prefix of Y
    - i.e, X[0,...i] and Y[0,...j] as L[i,j]
    - $\Theta(n^2)$ subproblems
- Idea
  - Either X[i] or Y[j] or neither are part of LCS
- Guess
  - Choose X[i] or Y[j] or neither as part of the LCS

DP Strategy

- Relate Subproblems
  - Case 1: X[i] = Y[j] = c
    - LCS of X[0,...i] and Y[0,...j] ends with c
    - Let some LCS be $X[i_1]X[i_2]...X[i_k] = Y[j_1]Y[j_2]...Y[j_k]$
      - If $X[i_k]$ or $Y[j_k]$ =c, we get same sequence by setting $i_k=i$, $j_k=j$
      - If $X[i_k] \neq c$, we can get longer LCS by adding c to the end
    - Hence L[i,j] = L[i-1,j-1]+1 if X[i] = Y[j]

DP Strategy

- Relate Subproblems
  - Case 2: X[i] ≠ Y[j]
    - Cannot have common subsequence that includes both X[i] and Y[j]
      - LCS can end with X[i] or Y[j] or neither but not both
    - Hence L[i,j] = max{L[i-1,j],L[i,j-1]} if X[i] ≠ Y[j]

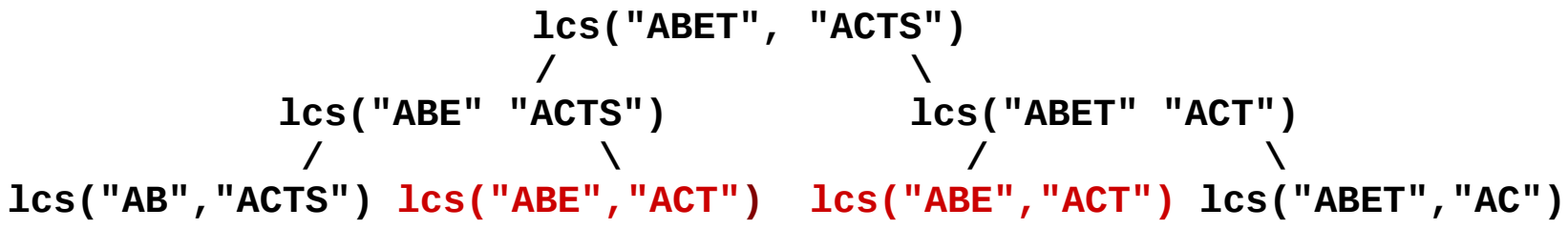
Recursive LCS

- LCS(X,Y,i,j)

  if X[i] =='\0' or Y[j] =='\0': return 0;

  else if X[i] == Y[j] return 1 + lcs(X, Y, i-1, j-1);

  else return max(lcs(X, Y, i, j-1), lcs(X, Y, i-1, j));



Recursion with Memoization

- Memoize
  - Store cost of longest subsequence between X[0:i] and Y[0:j] to L[i,j]
- LCS(X,Y, i, j){

```
if (L[i,j] < 0) {
    if (X[i] == '\0' || Y[j] == '\0') L[i,j] = 0;
    else if (A[i] == B[j]) L[i,j] = 1+ LCS(i-1, j-1);
    else L[i,j] = max(LCS(X,Y,i-1, j), LCS(X,Y,i, j-1));}
return L[i,j];
}
```

https://www.ics.uci.edu/~eppstein/161/960229.htm
l

Analysis

- Call to subproblem – constant
  - Called only when an entry to the table is made
  - O(m+1)(n+1) entries
- Cost atmost 2(m+1)(n+1)+1 ie O(mn)

Iterative LCS

- Build Table bottom-up
- IterativeLCS(X,Y,m,n)

```
for i ← -1 to n-1 do L[i,-1] ← 0
for j ← 0 to m-1 do L[-1,j] ← 0
for i ← 0 to n-1 do
    for j ← 0 to m-1 do
        if X[i]=Y[j] then
            L[i,j] ← L[i-1,j-1]+1
        Else
            L[i,j] ← max{L[i-1,j],L[i,j-1]}
```

Example

0	ϕ	A	B	C	A	B
ϕ	0	0	0	0	0	0
A	0	1	1	1	1	1
A	0	1	1	1	2	2
B	0	1	2	2	2	3
A	0	1	2	2	3	3
C	0	1	2	3	3	3
A	0	1	2	3	4	4

LCS = “ABCA”

https://s3.amazonaws.com/hr-challenge-images/8677/1433916387-bae1920043-LCS.png

Exercise

- Show the longest common subsequence table L for the following strings

X = MULTIDIMENSIONAL

Y = MANGROVEFOREST

The 0-1 Knapsack Problem








Determine which items to take so that it fits the sack, and total utility is maximized (an item must be taken completely or dropped)

The optimization problem

- Given set S of n items, where each item i has
  - Benefit b_i >= 0 and Weight w_i
  - We can take an item completely or drop it
- We have knapsack that can carry weight atmost W
- Goal – Maximize total benefit s.t

$$\sum_{i \in S} w_i \leq W$$

- Objective function to maximize

$$\sum_{i \in S} b_i$$

The DP Solution

- Subproblem Definition:
  - Let S_k be a subset of items from the original set S
  - Define B[k,w] as maximum total benefit of subset S_k, whose weight is w, considering item k
- Guess:
  - Should item k be included in the solution?

The DP Solution

- Relating sub-problems
  - Best subset of S_k that has total weight w is either best subset S_{k-1} with weight w, or best subset S_{k-1} with weight $w-w_k$ plus item k
  - Item k can be part of the knapsack or not

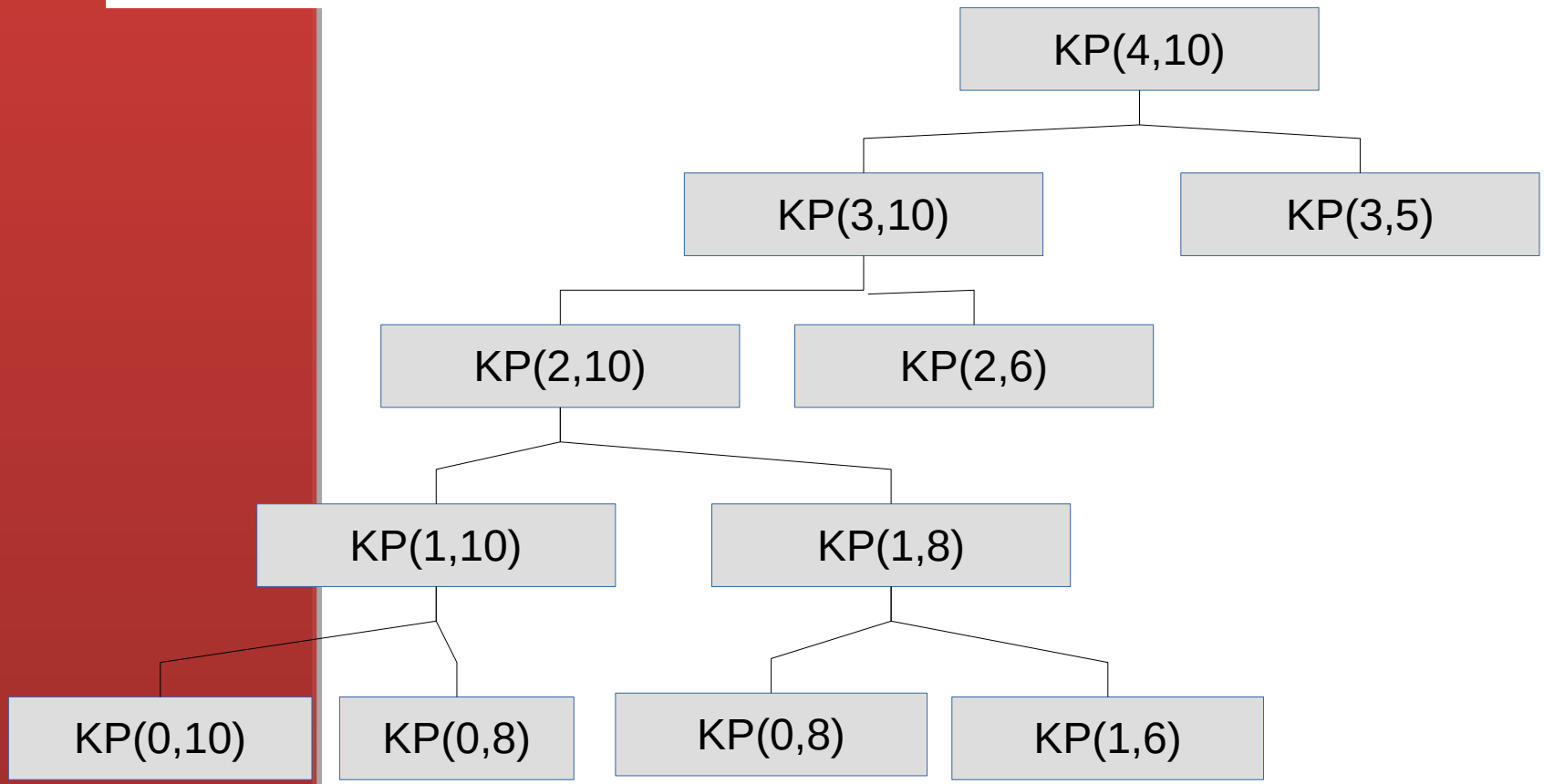
$$B[k,w] = \begin{cases} B[k-1,w] \; if \; w_k > w \\ max\left[B[k-1,w], B[k-1,w-w_k] + b_k\right] \; else \end{cases}$$

0-1 Knapsack: Recursive

```
int RecKnapsack(int k, int w){
    if(k == 0 || w == 0)
        return 0;
    if(w[k] > w)
        return RecKnapsack(k - 1, w);
    else
        return max(RecKnapsack(k - 1, w), Rec
Knapsack(k - 1, w - w[k]) + b[k]);
}
```

Need for Memoization

Item	0	1	2	3
Weight	2	2	4	5
Benefit	3	7	2	9



Memoized

- int MemKnapsack(int k, int w){

```
if(B[k][w] != -1)
    return B[k][w];
if(w[k] > w)
    B[k][w] = MemKnapsack(k - 1, w);
else
    B[k][w] = max(MemKnapsack(k-1, w),
MemKnapsack(k - 1, w - w[k]) + B[k]);
}
```

- Compexity O(nW)

Bottom-up Approach

- for w = 0 to W

  B[0,w] = 0

  for i = 1 to n

  B[i,0] = 0

  for i = 1 to n

  for w = 0 to W

  if w_i <= w // item i can be part of the solution

  if b_i + B[i-1,w-w_i] > B[i-1,w]

  B[i,w] = b_i + B[i-1,w- wi]

  else

  B[i,w] = B[i-1,w]

  else B[i,w] = B[i-1,w] // w_i > w

Finding the Actual Knapsack

- B[n,W] is the maximal value of items that can be placed in the Knapsack.
- Let i=n and k=W

  if B[i,k] != B[i-1,k] then

    mark the ith item as in the knapsack

    i = i-1, k = k-w_i

  else

    i = i-1 // Assume the ith item is not in the knapsack

Coin Change Problem

- You are given some amount N and a set of denominations $C=\{c_1,c_2,..,c_d\}$, and each denominations has n_i coins.
- The goal is to make change for the amount n using the given denominations.
  - Minimize the total number of coins returned for a particular quantity of change.

$$N=\sum_{k=1}^{d} x_k c_k$$

Dynamic Programming Solution

- Subproblem: C[p] is the minimum number of coins of denominations c_1, c_2, . . . , c_d needed for some value p
- Guess: The solution must start with some coin c_i
  - remaining coins in the optimal solution must themselves be the optimal solution to making change for $p-c_i$
  - $C[p] = 1+C[p-c_i]$;

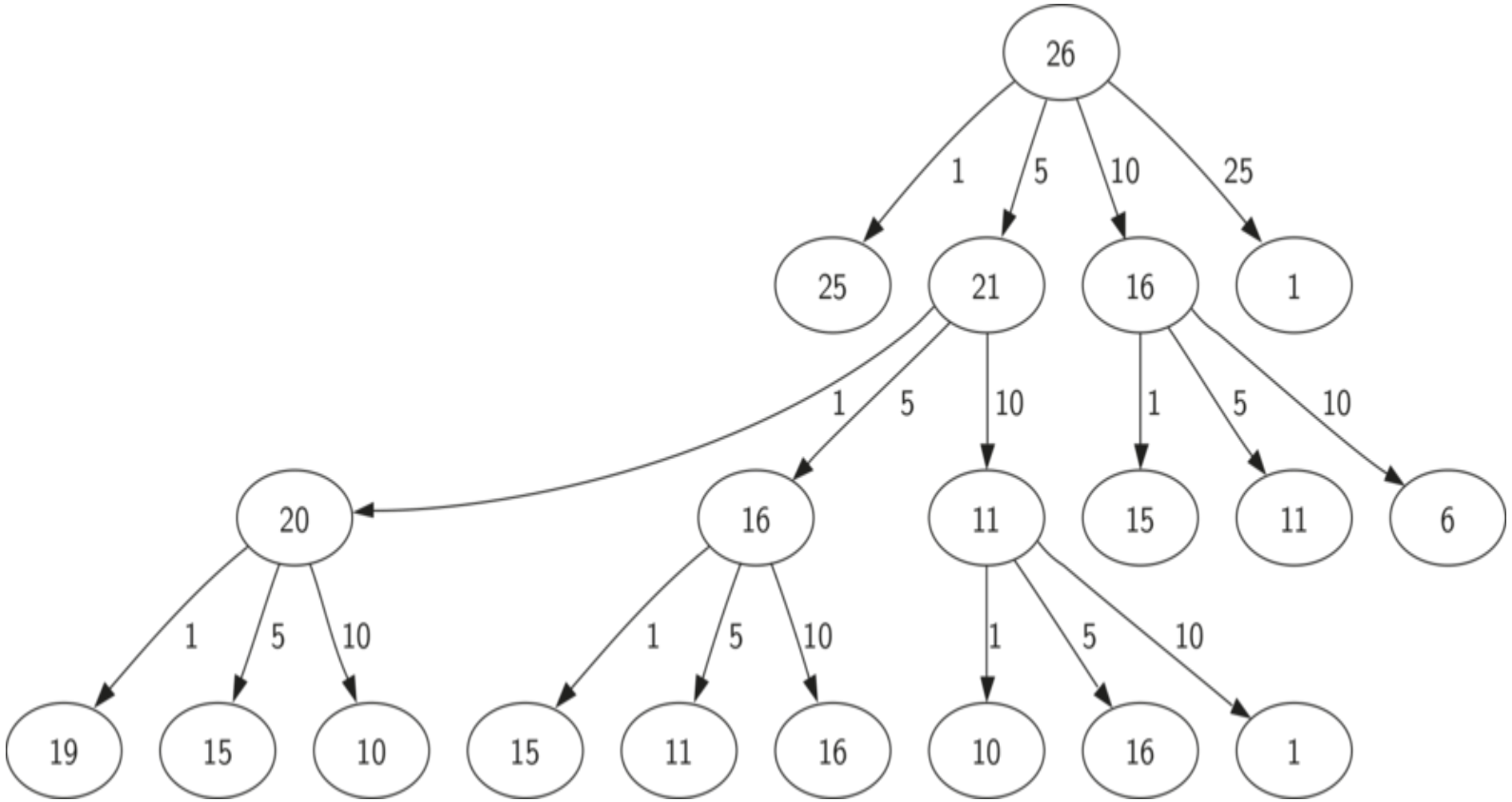
Dynamic Programming Solution

- Relate Subproblems
  - Find the denomination for which C[p] = 1+C[p−c_i] is minimum
- Recursion

$$C[p] = \begin{cases} 0 \; if p = 0 \\ min_{i:c_i \leq p} 1 + c[p - c_i] \; if p > 0 \end{cases}$$

- Find minimum coins for n=1,n=2, etc till final solution reached
  - e.g consider denominations 1,2,5, and choose the denominations in order
  - C[1] =1; C[2] = min(1+C[1], 1); C[5]= min(1+C[4], 1+C[3], 1)

Recursion Tree



Src: http://ice-web.cc.gatech.edu/ce21/1/static/audio/static/pythonds/_images/callTree.png

Bottom-up DP Solution

```
Change(C, d, N)
 C[0] = 0
 for p =  1 to n
        min =   1
        for i  = 1 to d
                if c[i] ⩽ p then
                        if 1 + C[p − c[i]] < min then
                                min =  1 + C[p − c[i]]
                                coin =  i
        C[p] = min
        S[p] =  coin
return C and S
```