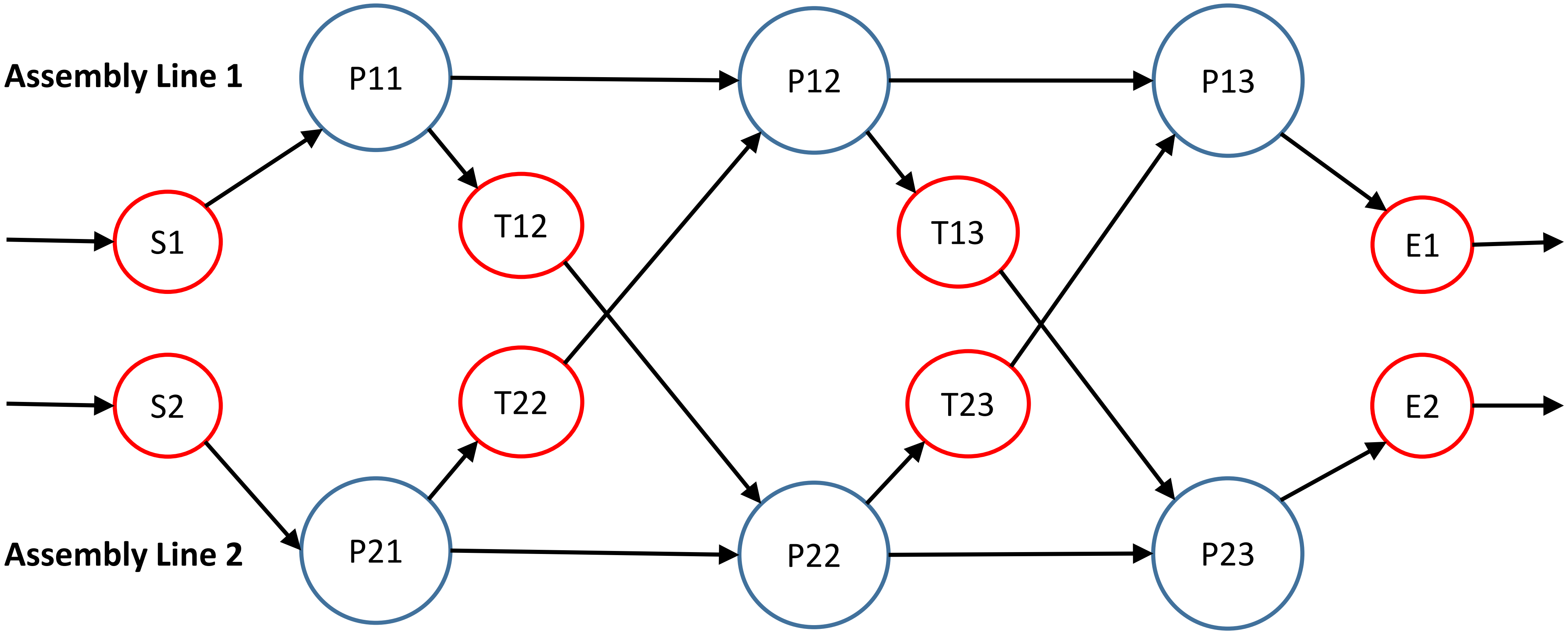
Dynamic Programming

Joy Mukherjee

Assembly Line Scheduling



Production of goods must pass through each of the n stations.

The parallel stations of the two assembly lines perform the same task.

Objective: Minimize the time for building a product.

Assembly Line Scheduling

Si = Starting Delay for Assembly Line i

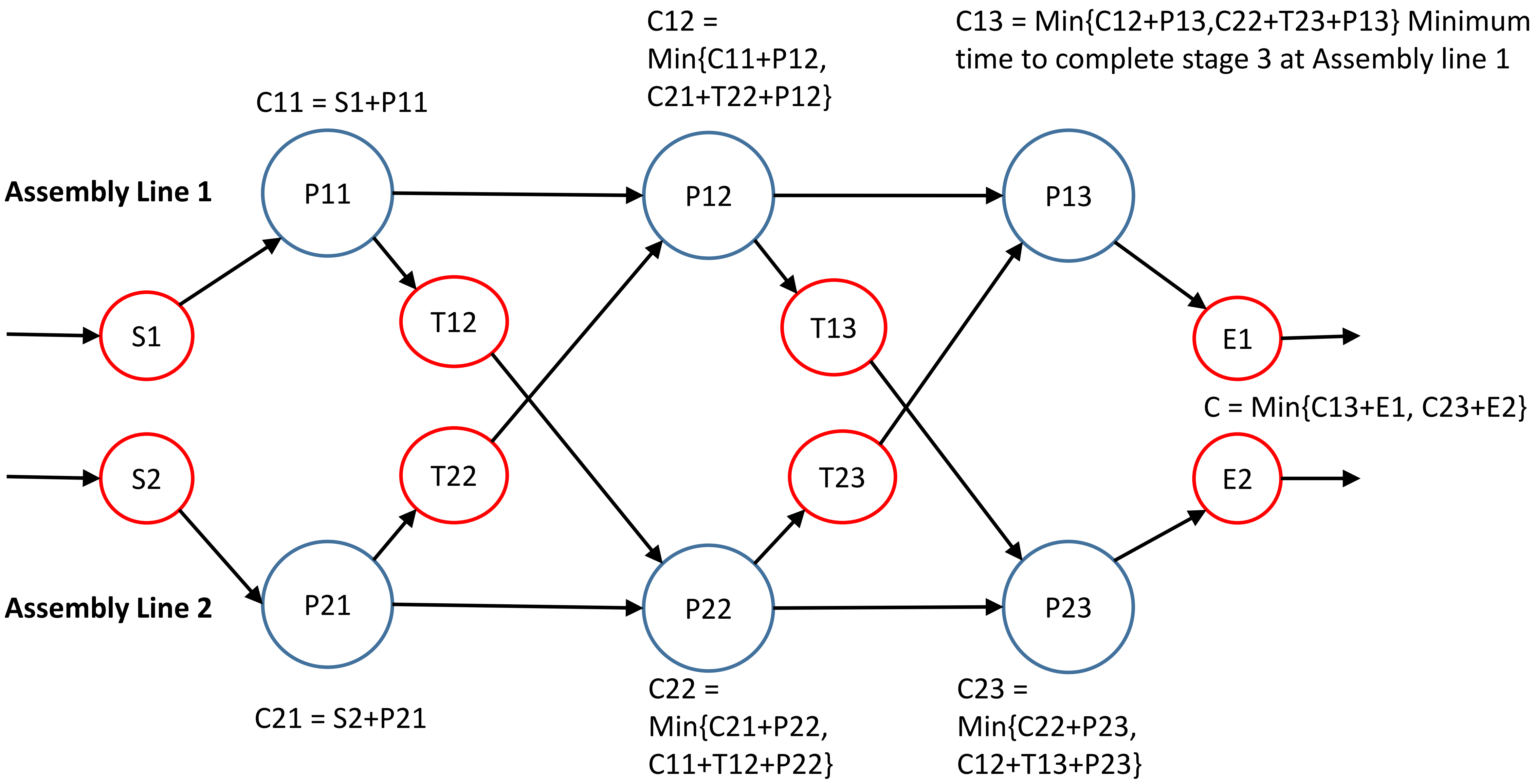
Ei = Ending Delay for Assembly Line i

Pij = Processing Time at Assembly Line i for stage j

T1j = Switching Time from stage j-1 of Assembly Line 1 to stage j of Assembly Line 2

T2j = Switching Time from stage j-1 of Assembly Line 2 to stage j of Assembly Line 1

Algorithm: Assembly Line Scheduling



Minimum Coin Change Problem

Given infinite supply of each of C = {C1,C2, ..,Cm} valued coins, find the minimum number of coins required to make the change of value R.

C = {1, 4, 5}

R = 8

Greedy Algorithm gives answer	**4**	**(One Rs 5 coin, Three Rs 1 coins)**
Dynamic Programming gives answer	**2**	**(Two Rs 4 coins)**

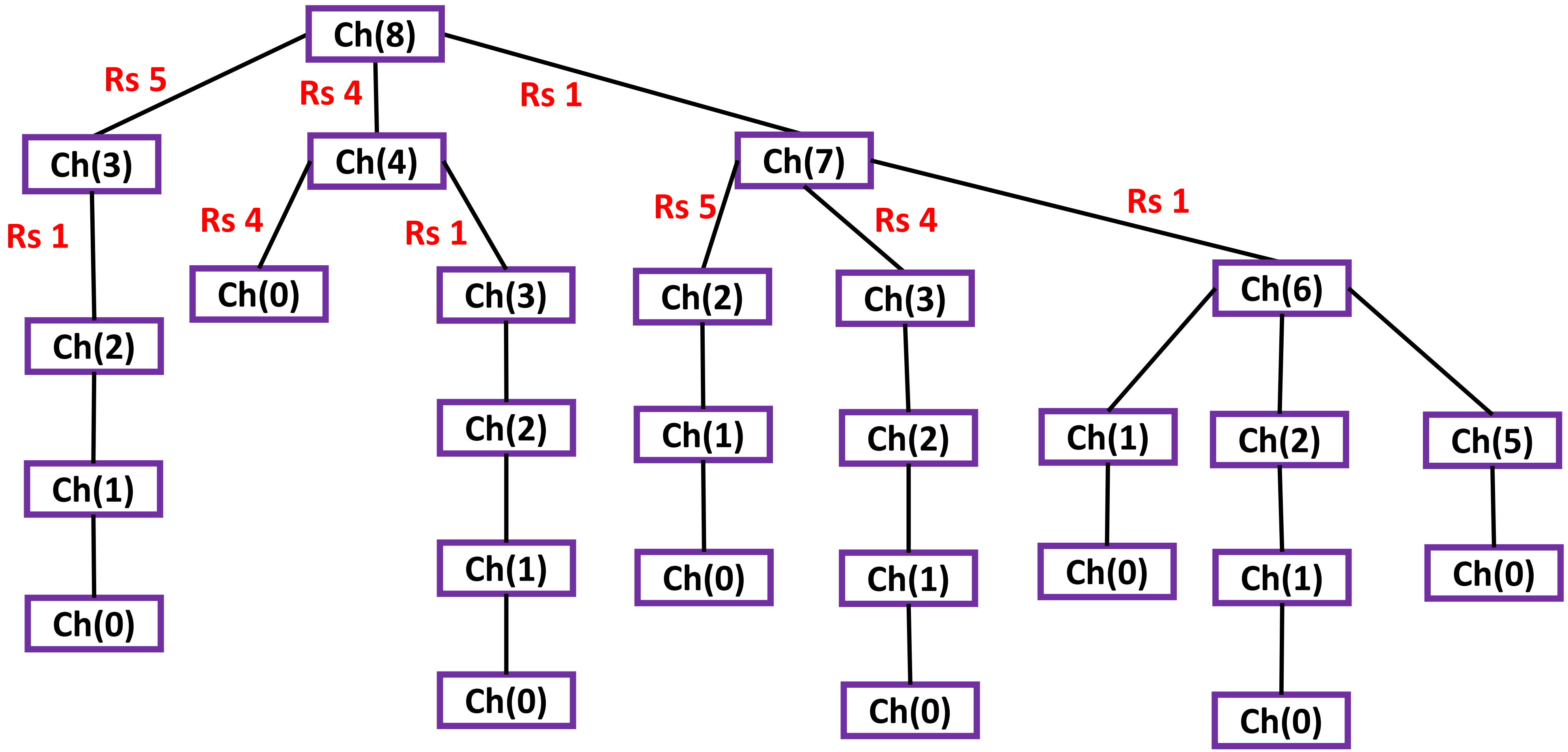
R = 18

Greedy Algorithm gives answer	**6**	**(Three Rs 5 coins, Three Rs 1 coins)**
Dynamic Programming gives answer	**4**	**(Two Rs 5 coins, Two Rs 4 coins)**

Minimum Coin Change Problem

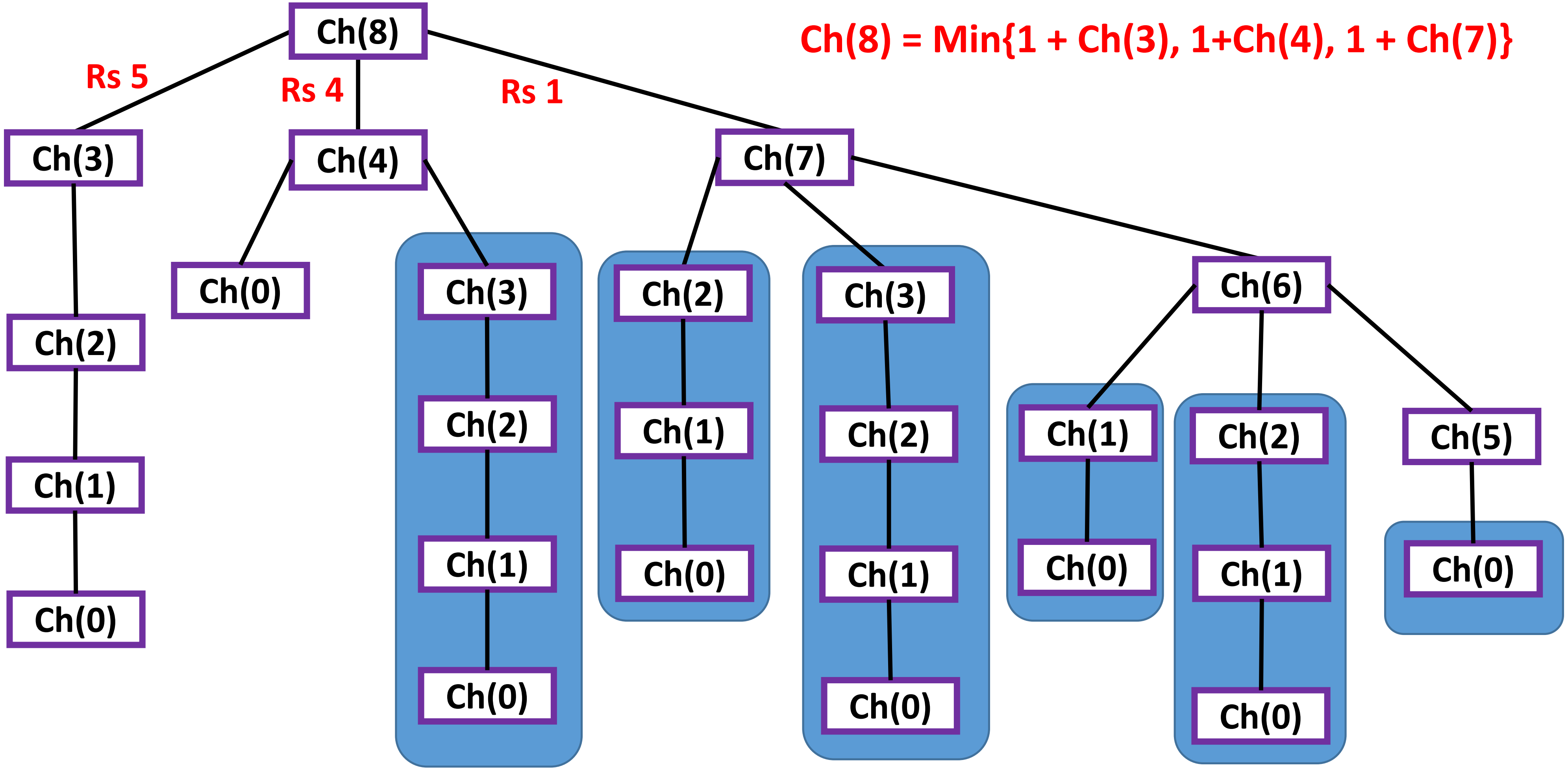
- $Ch(R)$ = Minimum no of coins to make a change for Rs. R
- $= \text{minimum}(1 + Ch(R-5),\quad 1 + Ch(R-4),\quad 1 + Ch(R-1))$

Optimal Substructure



Overlapping Subproblems

Ch(8) = Min{1 + Ch(3), 1+Ch(4), 1 + Ch(7)}



Minimum Coin Change Problem

Given infinite supply of each of C = {1, 4, 5} valued coins, find the minimum number of coins required to make the change of value R.

$$\text{Change}(R) = 0 \qquad \text{if } R = 0$$

$$= 1 + \text{Minimum}\,\{\text{Change}(R-1), \text{Change}(R-4), \text{Change}(R-5)\}$$

$$= \text{Minimum}\,\{1 + \text{Change}(R-1), 1 + \text{Change}(R-4), 1 + \text{Change}(R-5)\}$$

Minimum Coin Change Problem

Given infinite supply of each of C = {C1,C2, ..,Cm} valued coins, find the minimum number of coins required to make the change of value R.

$$\text{Change}(R) = 0 \qquad \textbf{if } R = 0$$

$$= 1 + \text{Minimum}_{\{1 \le i \le m\}} \{\text{Change}(R - C[i])\} \quad C[i] \le R \quad \textbf{otherwise}$$

Algorithm: Minimum Coin Change Problem

```
int Change(int C[], int m, int R)
{
    // T[i] = minimum number of coins required to make the change of value i
    int i, j, table[R+1];
    table[0] = 0;
    for (i = 1; i <= R; i++)
        table[i] = ∞;

    for (i = 1; i <= R; i++)
        for (j = 0; j < m; j++)
            if (C[j] <= i  &&  table[ i - C[j] ] + 1 < table[i])
                table[i] = table[ i - C[j] ] + 1;

    return table[R];
}
```

Rod Cutting Puzzle

Input: a rod of length n inches and prices of all pieces of size ≤ n.

Objective: Maximize the value obtained by cutting up the rod and selling the pieces.

Example: n = 5 and C[] = {2, 7, 5, 12, 15}

Ways to Cut the Rod	Total Value Obtained
5	15
1 + 4	2 + 12 = 14
2 + 3	7 + 5 = 12
1 + 1 + 3	2 + 2 + 5 = 9
1 + 2 + 2	**2 + 7 + 7 = 16**
1 + 1 + 1 + 2	2 + 2 + 2 + 7 = 13
1 + 1 + 1 + 1 + 1	2 + 2 + 2 + 2 + 2 = 10

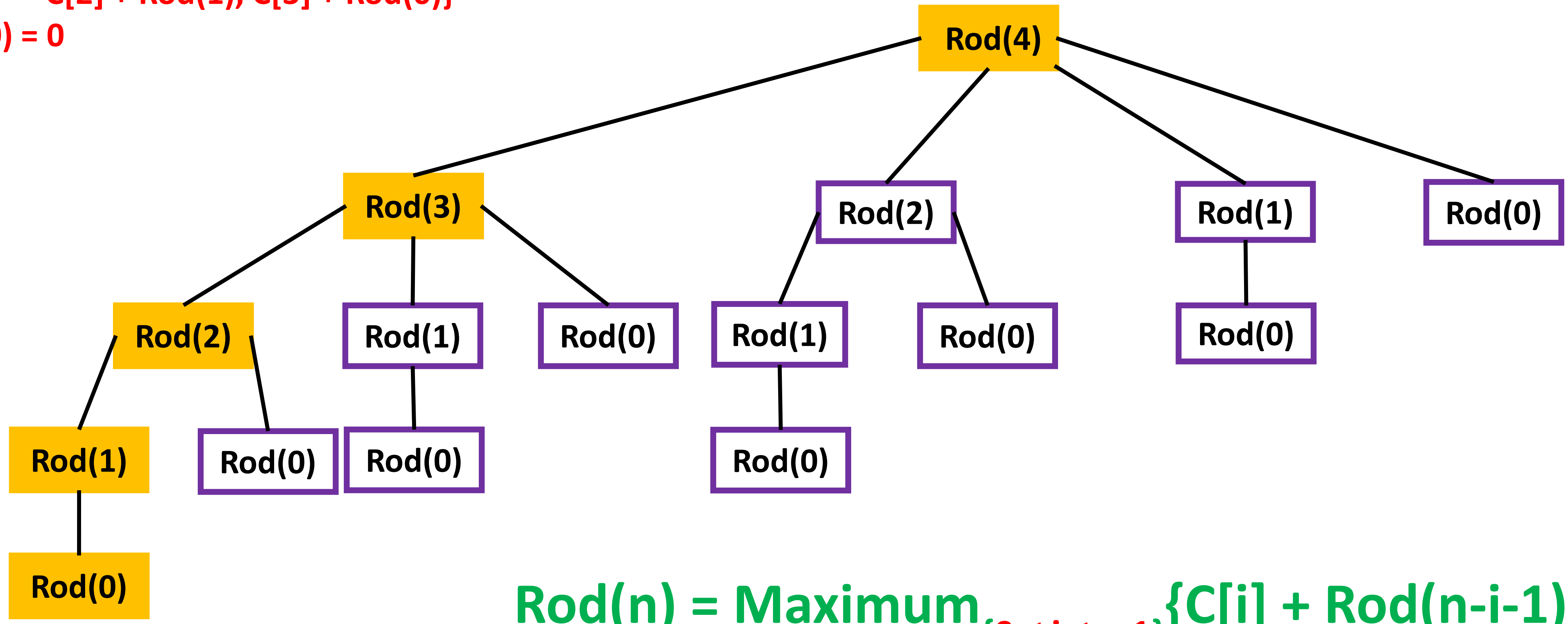
Rod Cutting Puzzle

- C[] = {2, 7, 5, 12, 15}

- Rod(5) = Maximum Profit for cutting a rod of length 5
- = Maximum{C[4] + Rod(0), C[3] + Rod(1), C[2] + Rod(2), C[1] + Rod(3), C[0] + Rod(4)}

Optimal Substructure

Rod(4) = Max{C[0] + Rod(3), C[1] + Rod(2),
C[2] + Rod(1), C[3] + Rod(0)}
Rod(0) = 0



$$Rod(n) = \text{Maximum}_{\{0 \le i \le n-1\}}\{C[i] + Rod(n-i-1)\}$$

$$Rod(0) = 0$$

Execution: Rod Cutting Puzzle

C					
	2	7	5	12	15
	0	1	2	3	4

Array Index starts from 0, i.e., C[i-1] = Cost of i-inch rod

T	0						initialize
T	0	2	0	0	0	0	i = 1 Max{ (C[0] + T[0]) }
T	0	2	7	0	0	0	i = 2 Max{ (C[0] + T[1]), (C[1] + T[0]) } =Max(4, 7) = 7
T	0	2	7	9	0	0	i = 3 Max{ (C[0] + T[2]), (C[1] + T[1]), (C[2] + T[0]) }= =Max(9, 9, 5) = 9
T	0	2	7	9	14	0	i = 4 Max{ (C[0] + T[3]), (C[1] + T[2]) , (C[2] + T[1]) , (C[3] + T[0]) }
T	0	2	7	9	14	16	i = 5 Max{ (C[0] + T[4]), (C[1] + T[3]) , (C[2] + T[2]) , (C[3] + T[1]) , (C[4] + T[0]) }

Algorithm: Rod Cutting Puzzle

```
int cutRod(int C[], int n)
{
        int   i,  j,  Rod[n+1];
        Rod[0] = 0;
        for (i = 1; i <= n; i++)
        {
                Rod[i] = 0;
                for (j = 0; j < i; j++)
                        Rod[ i ] = max(Rod[ i ], C[ j ] + Rod[ i - j - 1 ]);
        }
        return Rod[ n ];
}
```

Subset Sum (SS)

Input: A set A of n non-negative integers, and a value X,

Output: True if there is a subset of A with sum equal to X; False otherwise.

Example:

Input: n = 4, A[] = {3, 4, 12, 5}, X = 20

Output: True

Input: n = 4, A[] = {3, 4, 12, 5}, X = 18

Output: False

Subset Sum (SS)

- Input: n = 4, A[] = {3, 4, 12, 5}, X = 20
- Output: True

SS(A, 4, 20) = 5 + SS(A, 3, 15) OR SS(A, 3, 20)

= 5 + (12 + SS(A, 2, 3) OR SS(A, 2, 15)) OR (12 + SS(A, 2, 8) OR SS(A, 2, 20)

Optimal Substructure

Input: a set A of n non-negative integers, and a value X,

Output: determine if there is a subset of A with sum equal to X.

SS(A, n, X) = true if X = 0

SS(A, n, X) = false if n = 0 and X > 0

SS(A, n, X) = SS(A, n-1, X) || SS(A, n-1, X – A[n-1])

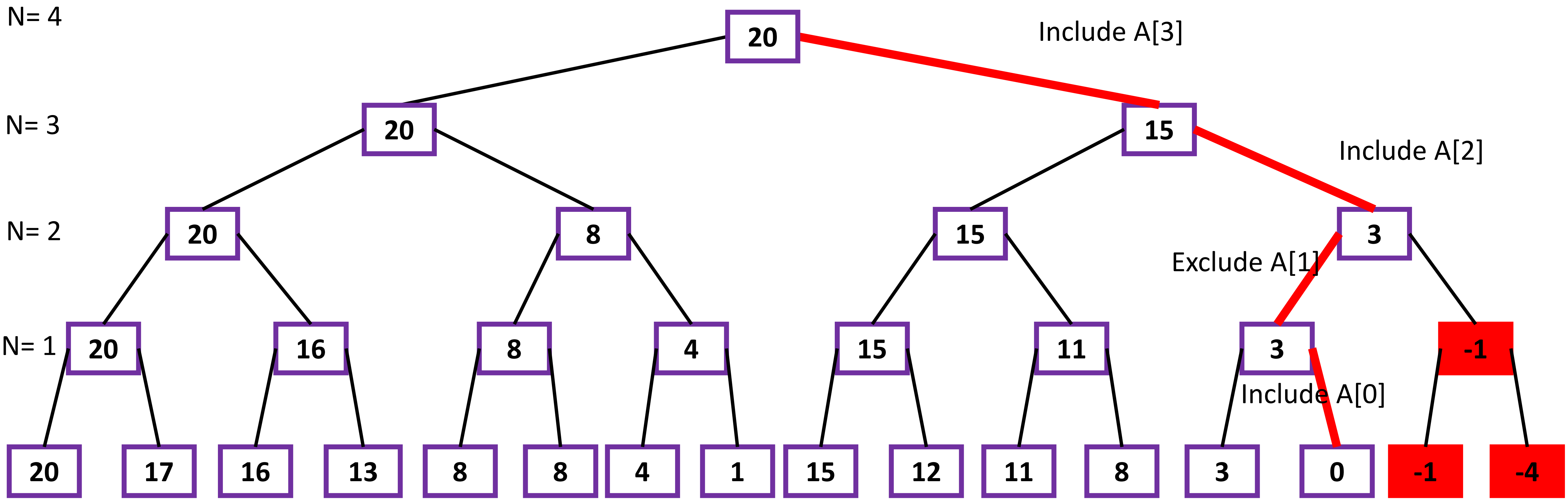
Exclude A[n-1] **Include A[n-1]**

n = 4, A[] = {3, 4, 5, 8}, X = 9

SS(A, 4, 9) = SS(A, 3, 9)

Or SS(A, 3, 1)

Optimal Substructure: SS({3,4,12,5}, 4, 20)



$$SS(A, n, X) = SS(A, n-1, X) \;||\; SS(A, n-1, X - A[n-1])$$

Execution: Subset Sum(SS({1,2,4,6}, 4, 8))

0	X=0	1	2	3	4	5	6	7	8
n=0	1	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0
2	1	1	1	1	0	0	0	0	0
3	1	1	1	1	1	1	1	1	0
4	1	1	1	1	1	1	1	1	1

Subset(4, 8) = Subset(3, 8) || Subset(3, 8-6) = 1

SS(A, 4, 8) = SS(A, 3, 8) || SS(A, 3, 2)

SS(A, 1, 3) = SS(A, 0, 3) || SS(A, 0, 2)

SS[i][j] = Is there a subset of A[0] through A[i-1] that constitutes j

Subset Sum (SS): DP is not a Polynomial Algorithm

- Time Complexity is defined as a function of input size
- N +ve integers, X is another +ve integer
- Size of N = $\log_2 N$ Size of X = $\log_2 X$
- Total size of the input = $N \log_2 X + \log_2 N + \log_2 X = O(N \log_2 X)$
- $O(NX) = O(N\, 2^{\log 2\, X})$ = Exponential Algorithm

Subset Sum (SS)

```
int isSubsetSum(int A[], int n, int X)
{
    int i, j, subset[n+1][X+1];
    // subset[i][j] = 1 if there is a subset of A[0..i-1] with sum equal to j
    for (i = 0; i <= n; i++)    subset[i][0] = 1;        // If sum is 0, then answer is 1
    for (j = 1; j <= X; j++)    subset[0][j] = 0;        // If A is empty and sum is not 0, then answer is 0
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= X; j++) {
            if(j < A[i-1])        subset[i][j]  =  subset[i-1][j];
            else                  subset[i][j]  =  subset[i-1][j]   ||   subset[i - 1][j - A[i-1]];
        }
    }
    return subset[n][X];
}
```

Subset Sum (SS)

Given a set of n integers, and an integer X, write a C/C++/Java program that prints false if there is no subset that constitutes X; otherwise it prints true with the integers in the subset. Note that the integers in the set may be negative, zero, or positive.

0-1 Knapsack Problem

Input: Knapsack of volume V, n items, where item i has volume ci and cost pi

Output: Maximum value that one can put into the knapsack.

Constraints:

1. The total volume of selected items ≤ V
2. Either pick an item or not picked at all (0/1 property)
3. Exactly one copy of each item is available

An Example

Input: V = 6, n = 4

Output: 7 (Pick a0 and a3)

Item	Ci(volume)	Pi(price)
a0	1	2
a1	2	1
a2	3	3
a3	4	5

Constraints:

1. The total volume of selected items ≤ V
2. Either pick an item or not picked at all (0/1 property)

Optimal Substructure

Input: Knapsack of volume V, n items, where item I has volume ci and cost pi

Output: Maximum value that one can put into the knapsack.

$K(i, j)$ = Max cost of items that can be put in a knapsack of volume j out of i items

$K(i, j) = 0 \quad$ if $j = 0$ or $i = 0$

$K(i, j) = K(i-1, j) \quad$ if $C[i-1] > j$ // Volume of i-th item is greater than j

$K(i, j) = \text{maximum}\{K(i-1, j),\ \mathbf{P[i-1] + K(i-1, j-C[i-1]}\})$

Exclude i-th item **Include i-th item**

Execution: Initialization

Input: V = 6, n = 4

Item	ci	pi
a0	1	2
a1	2	1
a2	3	3
a3	4	5

	0	**1**	**2**	**3**	**4**	**5**	**6**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**						
2	**0**						
3	**0**						
4	**0**						

K[i][j] = 0 if i = 0 or j = 0
K[i][j] = Max value collected from first i items in volume j

Execution

Input: V = 6, n = 4

Item	ci	pi
a0	1	2
a1	2	1
a2	3	3
a3	4	5

	0	1	2	3	4	5	6
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**	**2**	**2**	**2**	**2**	**2**	**2**
2	**0**						
3	**0**						
4	**0**						

K[i][j] = max(P[i - 1] + K[i - 1][j - C[i - 1]], K[i - 1][j])
K[1][2] = max(P[0] + K[0][2 - C[0]], K[0][2]) = max(2+0, 0) = 2
K[i][j] = Max value collected from first i items in volume j

Execution

Input: V = 6, n = 4

Item	ci	pi
a0	1	2
a1	2	1
a2	3	3
a3	4	5

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	2	2	2	2	2	2
2	0	2	2	3	3	3	3
3	0						
4	0						

K[i][j] = max(P[i - 1] + K[i - 1][j - C[i - 1]], K[i - 1][j])
K[2][3] = max(P[1] + K[1][3 - C[1]], K[1][3])
= max(1 + 2, 2) = 3

Execution

Input: V = 6, n = 4

Item	ci	pi
a0	1	2
a1	2	1
a2	3	3
a3	4	5

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	2	2	2	2	2	2
2	0	2	2	3	3	3	3
3	0	2	2	3	5	5	6
4	0						

K[i][j] = max(P[i - 1] + K[i - 1][j - C[i - 1]], K[i - 1][j])
K[3][6] = max(P[2] + K[2][6 - C[2]], K[2][6])
= max(3 + 3, 3) = 6

Execution

Input: V = 6, n = 4

Item	ci	pi
a0	1	2
a1	2	1
a2	3	3
a3	4	5

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	2	2	2	2	2	2
2	0	2	2	3	3	3	3
3	0	2	2	3	5	5	6
4	0	2	2	3	5	7	7

K[i][j] = max(P[i - 1] + K[i - 1][j - C[i - 1]], K[i - 1][j])

K[4][6] = max(P[3] + K[3][6 - C[3]], K[3][6])

= max(5 + 2, 6) = 7

Algorithm: 0-1 Knapsack Problem

```
void ZeroOneKnapsack(int V, int C[], int P[], int n)
{
    int i, j, K[n + 1][V + 1];
    for (i = 0; i <= n; i++) {
        for (j = 0; j <= V; j++) {
            if (i == 0 || j == 0)
                K[i][j] = 0;
            else if (C[i - 1] <= j)
                K[i][j] = max(P[i - 1] + K[i - 1][j - C[i - 1]], K[i - 1][j])
            else
                K[i][j] = K[i - 1][j];
        }
    }
    printf("%d", K[n][V]);
}
```

Think ☺

Input: Knapsack of volume V, n items, where item i has volume ci and cost pi

Output: Maximum value that one can put into the knapsack.

Constraints:

1. The total volume of selected items ≤ V
2. Either pick an item or not picked at all (0/1 property)
3. Each item is available in plenty

An Example

Input: $V = 60$, $n = 4$

Output: ?

Item	Ci(volume)	Pi(price)	Quantity
a0	7	2	∞
a1	2	8	∞
a2	9	6	∞
a3	4	5	∞

Constraints:

1. The total volume of selected items ≤ V
2. Either pick an item or not picked at all (0/1 property)
3. Each item is available in plenty

Unbounded 0-1 Knapsack Problem

```
int UnboundedKanapsack(int C[], int P[], int n, int V)
{
// T[i] = maximum value that can be put in volume i
   int i, j, T[V+1];
   T[0] = 0;
   for (i = 1; i <= V; i++)
      T[i] = INT_MIN;

   for (i = 1; i <= V; i++)
      for (j = 0; j < n; j++)
         if (C[j] <= i  &&  T[ i - C[j] ] + P[j] > T[i])
            T[i] = T[ i - C[j] ] + P[j];

   return T[V];
}
```

Palindrome Partitioning Problem

- Input: A string s of English lowercase alphabets
- Output: Minimum number of partitions of s such that each partition is a palindrome

- Ex 1: madam 0
- Ex 2: hello 3
- Ex 3: aaaaaxacc 2
- Ex 4: abcde 4
- Ex 5: abaababb 1

Palindrome Partitioning Problem

- Suppose the string is str of length n
- P[n][n]
- P[i][j] = 1 if str[i...j] is a palindrome
- = 0 otherwise
- It will take $O(n^2)$ time to identify all valid palindromic substrings
- C[n]
- C[i] = Minimum partitions needed in s[0..i] to make each partition a palindrome
- It will take $O(n^2)$ time to find minimum palindromic partitions

str[] = “AAAXACC”

P	A(0)	A(1)	A(2)	X(3)	A(4)	C(5)	C(6)
A(0)	1	1	1	0	0	0	0
A(1)		1	1	0	0	0	0
A(2)			1	0	1	0	0
X(3)				1	0	1	0
A(4)					1	0	0
C(5)						1	1
C(6)							1

Palindrome Partitioning Problem

```
int n = strlen(s);
int P[n][n], C[n];
   for(l = 0; l < n; l++) {
      for(i = 0; i < n-l; i++) {
         j = i+l;
         if(s[i] == s[j] && j-i < 2)
            P[i][j] = 1;
         else if(s[i] == s[j] && j-i > 1)
            P[i][j] = P[i+1][j-1];
         else
            P[i][j] = 0;
      }
   }
```

str[] = “AAAXACC”

P	A(0)	A(1)	A(2)	X(3)	A(4)	C(5)	C(6)
A(0)	1	1	1	0	0	0	0
A(1)		1	1	0	0	0	0
A(2)			1	0	1	0	0
X(3)				1	0	1	0
A(4)					1	0	0
C(5)						1	1
C(6)							1

	A(0)	A(1)	A(2)	X(3)	A(4)	C(5)	C(6)
C	0	0	0	1	1	2	2

C[6] = s[6] + C[5] = 1+2 = 3
=s[5,6] + C[4] = 1 + 1 = 2
=s[4,6] + C[3] = Not possible

Palindrome Partitioning Problem

```
C[0] = 0;
 for(i = 1; i < n; i++) {
    C[i] = n-1;
    if(P[0][i] == 1)
       C[i] = 0;
    else {
       for(j = i; j > 0; j--) {
          if(P[j][i] == 1 && C[i] > 1 + C[j-1])
             C[i] = 1 + C[j-1];
       }
    }
 }
 return C[n-1];
```