Contents

Section outline

Chain matrix multiplication

Example (Chain matrix multiplication)

- Let A_1 be a 10×100 matrix
- Let A_2 be a 100×5 matrix
- Let A_3 be a 5×50 matrix
- Need to compute $A_1 \times A_2 \times A_3$

Chain matrix multiplication

Example (Chain matrix multiplication)

- Let A_1 be a 10×100 matrix
- Let A_2 be a 100×5 matrix
- Let A_3 be a 5×50 matrix
- Need to compute $A_1 \times A_2 \times A_3$
- Cost of $(A_1 \times A_2) \times A_3$: $(10 \times 100 \times 5) + (10 \times 5 \times 50) = 7,500$
- Cost of $A_1 \times (A_2 \times A_3)$: $(100 \times 5 \times 50) + (10 \times 100 \times 50) = 75,000$

Chain matrix multiplication

Example (Chain matrix multiplication)

- Let A_1 be a 10×100 matrix
- Let A_2 be a 100×5 matrix
- Let A_3 be a 5×50 matrix
- Need to compute $A_1 \times A_2 \times A_3$
- Cost of $(A_1 \times A_2) \times A_3$: $(10 \times 100 \times 5) + (10 \times 5 \times 50) = 7,500$
- Cost of $A_1 \times (A_2 \times A_3)$: $(100 \times 5 \times 50) + (10 \times 100 \times 50) = 75,000$

Definition

Chain matrix multiplication Given n matrices, $A_1, \ldots, A_\imath, \ldots, A_n$, where for $1 \leq \imath \leq n$, $A_\imath$ is a $p_{\imath-1} \times p_\imath$, matrix, parenthesise the product $A_1 \times \ldots \times A_\imath \times \ldots \times A_n$ so as to minimize the total cost of multiplication, assuming that the cost of multiplying a $p\imath{-}1 \times p_\imath$ matrix by a $p_\imath \times p_{\imath+1}$ matrix using the naive algorithm is $p_{\imath-1} \times p_\imath \times p_{\imath+1}$

Formulation for chain matrix multiplication

- If there is just a single matrix, there is nothing to decide
- For n ($n \geq 2$), we need to divide the problem into two parts suitably, in one of the $n - 1$ possible ways
- The sub-problems are solved optimally to get the requisite solution

Formulation for chain matrix multiplication

- If there is just a single matrix, there is nothing to decide
- For n ($n \geq 2$), we need to divide the problem into two parts suitably, in one of the $n - 1$ possible ways
- The sub-problems are solved optimally to get the requisite solution
- However, the quality of the solution is dependent on the point of division

Formulation for chain matrix multiplication

- If there is just a single matrix, there is nothing to decide
- For n ($n \geq 2$), we need to divide the problem into two parts suitably, in one of the $n-1$ possible ways
- The sub-problems are solved optimally to get the requisite solution
- However, the quality of the solution is dependent on the point of division
- So, we need to consider all possible ways to divide the problem into two parts and retain the best choice
- The total number of possible solutions to be handled is huge:

$$N_n = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} N_k N_{n-k} & n > 1 \end{cases}$$

Formulation for chain matrix multiplication

- If there is just a single matrix, there is nothing to decide
- For n ($n \geq 2$), we need to divide the problem into two parts suitably, in one of the $n-1$ possible ways
- The sub-problems are solved optimally to get the requisite solution
- However, the quality of the solution is dependent on the point of division
- So, we need to consider all possible ways to divide the problem into two parts and retain the best choice
- The total number of possible solutions to be handled is huge:

$$N_n = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} N_k N_{n-k} & n > 1 \end{cases} \equiv C_{n-1} = \begin{cases} 1 & n = 0 \\ \sum_{k=0}^{n-2} C_k C_{n-k} & n \geq 1 \end{cases}$$

- $(n-1)^{\text{th}}$ Catalan number, $C_{n-1} \in \Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$

Naive solution for chain matrix multiplication

```
chnMatMulSim(int p[x-1 ..  y]) { // dimensions
  int n=y-x+1;
  if (n==1) return 0; // single matrix
  for L = 2 to n { // lengths of subchains
    for i = x to n-L+1 { // starts of subchains
      j = i+L-1; // ends of subchains
      c = ∞; // init to find min cost
      for k = i to j - 1 { // check all splits
        q = chnMatMulSim(p[i-1,k] +
          chnMatMulSim[k,j] + p[i-1]×p[k]×p[j];
        if (q < c) c = q; // check for lower cost
      }
    }
  }
  return c;
}
```

Analysis of naive method

$$
\begin{aligned}
T(n) &= \begin{cases} 0 & n = 1 \\ \sum_{k=1}^{n-1} \left(T(k) + T(n-k) + c\right) & n > 1 \end{cases} \\
&= 2\sum_{k=1}^{n-1} T(k) + (n-1)c & n > 1 \\
T(n+1) &= 2\sum_{k=1}^{n} T(k) + nc & n > 1 \\
&= \underbrace{2\sum_{k=1}^{n-1} T(k) + (n-1)c}_{T(n)} + 2T(n) + c & n > 1 \\
&= 3T(n) + c & n > 1 \\
T(n) &= 3T(n-1) + c & n > 1 \\
&= \tfrac{1}{2}\left(3^{n-1} - 1\right)c
\end{aligned}
$$

Thus, the naive method works in exponential time.

DP formulation for chain matrix multiplication

- Key observation:

DP formulation for chain matrix multiplication

- Key observation: Occurrence of common sub-problems
- To solve for $\prod_{i=1}^{5} A_i$, need to consider $A_1 \times A_2$ while solving for $\prod_{i=1}^{4} A_i$ and also $\prod_{i=1}^{3} A_i$
- May other sub-problems are also repeated
- Considerable savings possible by reusing solutions to earlier identified sub-problems (memoization), thereby avoiding solving those again and again

DP formulation for chain matrix multiplication

- Key observation: Occurrence of common sub-problems
- To solve for $\prod_{\imath=1}^{5} A_\imath$, need to consider $A_1 \times A_2$ while solving for $\prod_{\imath=1}^{4} A_\imath$ and also $\prod_{\imath=1}^{3} A_\imath$
- May other sub-problems are also repeated
- Considerable savings possible by reusing solutions to earlier identified sub-problems (memoization), thereby avoiding solving those again and again
- Problems can be solved bottom-up to obtain required solution

Bottom-up chain matrix multiplication

Example (Efficient computation of $A_1 \times A_2 \times \ldots \times A_7$)

Bottom-up chain matrix multiplication

Example (Efficient computation of $A_1 \times A_2 \times \ldots \times A_7$)

A_1 A_2 A_3 A_4 A_5 A_6 A_7

Bottom-up chain matrix multiplication

Example (Efficient computation of $A_1 \times A_2 \times \ldots \times A_7$)

$(A_1 \times A_2)$ $(A_2 \times A_3)$ $(A_3 \times A_4)$ $(A_4 \times A_5)$ $(A_5 \times A_6)$ $(A_6 \times A_7)$

A_1 A_2 A_3 A_4 A_5 A_6 A_7

Bottom-up chain matrix multiplication

Example (Efficient computation of $A_1 \times A_2 \times \ldots \times A_7$)

$$\left(\prod_{i=1}^{3} A_i\right) \quad \left(\prod_{i=2}^{4} A_i\right) \quad \left(\prod_{i=3}^{5} A_i\right) \quad \left(\prod_{i=4}^{6} A_i\right) \quad \left(\prod_{i=5}^{7} A_i\right)$$

$$(A_1 \times A_2) \; (A_2 \times A_3) \; (A_3 \times A_4) \; (A_4 \times A_5) \; (A_5 \times A_6) \; (A_6 \times A_7)$$

$$A_1 \quad A_2 \quad A_3 \quad A_4 \quad A_5 \quad A_6 \quad A_7$$

Bottom-up chain matrix multiplication

Example (Efficient computation of $A_1 \times A_2 \times \ldots \times A_7$)

$$\left(\prod_{i=1}^{4} A_i\right) \quad \left(\prod_{i=2}^{5} A_i\right) \quad \left(\prod_{i=3}^{6} A_i\right) \quad \left(\prod_{i=4}^{7} A_i\right)$$

$$\left(\prod_{i=1}^{3} A_i\right) \quad \left(\prod_{i=2}^{4} A_i\right) \quad \left(\prod_{i=3}^{5} A_i\right) \quad \left(\prod_{i=4}^{6} A_i\right) \quad \left(\prod_{i=5}^{7} A_i\right)$$

$$(A_1 \times A_2) \ (A_2 \times A_3) \ (A_3 \times A_4) \ (A_4 \times A_5) \ (A_5 \times A_6) \ (A_6 \times A_7)$$

$$A_1 \quad A_2 \quad A_3 \quad A_4 \quad A_5 \quad A_6 \quad A_7$$

Bottom-up chain matrix multiplication

Example (Efficient computation of $A_1 \times A_2 \times \ldots \times A_7$)

$$\left(\prod_{i=1}^{5} A_i\right) \quad \left(\prod_{i=2}^{6} A_i\right) \quad \left(\prod_{i=3}^{7} A_i\right)$$

$$\left(\prod_{i=1}^{4} A_i\right) \quad \left(\prod_{i=2}^{5} A_i\right) \quad \left(\prod_{i=3}^{6} A_i\right) \quad \left(\prod_{i=4}^{7} A_i\right)$$

$$\left(\prod_{i=1}^{3} A_i\right) \quad \left(\prod_{i=2}^{4} A_i\right) \quad \left(\prod_{i=3}^{5} A_i\right) \quad \left(\prod_{i=4}^{6} A_i\right) \quad \left(\prod_{i=5}^{7} A_i\right)$$

$$(A_1 \times A_2) \ (A_2 \times A_3) \ (A_3 \times A_4) \ (A_4 \times A_5) \ (A_5 \times A_6) \ (A_6 \times A_7)$$

$$A_1 \quad A_2 \quad A_3 \quad A_4 \quad A_5 \quad A_6 \quad A_7$$

Bottom-up chain matrix multiplication

Example (Efficient computation of $A_1 \times A_2 \times \ldots \times A_7$)

$$\left(\prod_{i=1}^{6} A_i\right) \quad \left(\prod_{i=2}^{7} A_i\right)$$

$$\left(\prod_{i=1}^{5} A_i\right) \quad \left(\prod_{i=2}^{6} A_i\right) \quad \left(\prod_{i=3}^{7} A_i\right)$$

$$\left(\prod_{i=1}^{4} A_i\right) \quad \left(\prod_{i=2}^{5} A_i\right) \quad \left(\prod_{i=3}^{6} A_i\right) \quad \left(\prod_{i=4}^{7} A_i\right)$$

$$\left(\prod_{i=1}^{3} A_i\right) \quad \left(\prod_{i=2}^{4} A_i\right) \quad \left(\prod_{i=3}^{5} A_i\right) \quad \left(\prod_{i=4}^{6} A_i\right) \quad \left(\prod_{i=5}^{7} A_i\right)$$

$$(A_1 \times A_2) \; (A_2 \times A_3) \; (A_3 \times A_4) \; (A_4 \times A_5) \; (A_5 \times A_6) \; (A_6 \times A_7)$$

$$A_1 \quad A_2 \quad A_3 \quad A_4 \quad A_5 \quad A_6 \quad A_7$$

Bottom-up chain matrix multiplication

Example (Efficient computation of $A_1 \times A_2 \times \ldots \times A_7$)

$$\left(\prod_{i=1}^{7} A_i\right)$$

$$\left(\prod_{i=1}^{6} A_i\right) \quad \left(\prod_{i=2}^{7} A_i\right)$$

$$\left(\prod_{i=1}^{5} A_i\right) \quad \left(\prod_{i=2}^{6} A_i\right) \quad \left(\prod_{i=3}^{7} A_i\right)$$

$$\left(\prod_{i=1}^{4} A_i\right) \quad \left(\prod_{i=2}^{5} A_i\right) \quad \left(\prod_{i=3}^{6} A_i\right) \quad \left(\prod_{i=4}^{7} A_i\right)$$

$$\left(\prod_{i=1}^{3} A_i\right) \quad \left(\prod_{i=2}^{4} A_i\right) \quad \left(\prod_{i=3}^{5} A_i\right) \quad \left(\prod_{i=4}^{6} A_i\right) \quad \left(\prod_{i=5}^{7} A_i\right)$$

$$(A_1 \times A_2) \; (A_2 \times A_3) \; (A_3 \times A_4) \; (A_4 \times A_5) \; (A_5 \times A_6) \; (A_6 \times A_7)$$

$$A_1 \quad A_2 \quad A_3 \quad A_4 \quad A_5 \quad A_6 \quad A_7$$

Bottom-up chain matrix multiplication (contd.)

Example (Upper triangular DP matrix for $A_1 \times A_2 \times \ldots \times A_7$)

Bottom-up chain matrix multiplication (contd.)

Example (Upper triangular DP matrix for $A_1 \times A_2 \times \ldots \times A_7$)

A_1

A_2

A_3

A_4

A_5

A_6

A_7

Bottom-up chain matrix multiplication (contd.)

Example (Upper triangular DP matrix for $A_1 \times A_2 \times \ldots \times A_7$)

A_1	$(A_1 \times A_2)$					
	A_2	$(A_2 \times A_3)$				
		A_3	$(A_3 \times A_4)$			
			A_4	$(A_4 \times A_5)$		
				A_5	$(A_5 \times A_6)$	
					A_6	$(A_6 \times A_7)$
						A_7

Bottom-up chain matrix multiplication (contd.)

Example (Upper triangular DP matrix for $A_1 \times A_2 \times \ldots \times A_7$)

A_1	$(A_1 \times A_2)$	$\left(\prod_{i=1}^{3} A_i\right)$				
	A_2	$(A_2 \times A_3)$	$\left(\prod_{i=2}^{4} A_i\right)$			
		A_3	$(A_3 \times A_4)$	$\left(\prod_{i=3}^{5} A_i\right)$		
			A_4	$(A_4 \times A_5)$	$\left(\prod_{i=4}^{6} A_i\right)$	
				A_5	$(A_5 \times A_6)$	$\left(\prod_{i=5}^{7} A_i\right)$
					A_6	$(A_6 \times A_7)$
						A_7

Bottom-up chain matrix multiplication (contd.)

Example (Upper triangular DP matrix for $A_1 \times A_2 \times \ldots \times A_7$)

A_1	$(A_1 \times A_2)$	$\left(\prod_{i=1}^{3} A_i\right)$	$\left(\prod_{i=1}^{4} A_i\right)$			
	A_2	$(A_2 \times A_3)$	$\left(\prod_{i=2}^{4} A_i\right)$	$\left(\prod_{i=2}^{5} A_i\right)$		
		A_3	$(A_3 \times A_4)$	$\left(\prod_{i=3}^{5} A_i\right)$	$\left(\prod_{i=3}^{6} A_i\right)$	
			A_4	$(A_4 \times A_5)$	$\left(\prod_{i=4}^{6} A_i\right)$	$\left(\prod_{i=4}^{7} A_i\right)$
				A_5	$(A_5 \times A_6)$	$\left(\prod_{i=5}^{7} A_i\right)$
					A_6	$(A_6 \times A_7)$
						A_7

Bottom-up chain matrix multiplication (contd.)

Example (Upper triangular DP matrix for $A_1 \times A_2 \times \ldots \times A_7$)

A_1	$(A_1 \times A_2)$	$\left(\prod_{i=1}^{3} A_i\right)$	$\left(\prod_{i=1}^{4} A_i\right)$	$\left(\prod_{i=1}^{5} A_i\right)$		
	A_2	$(A_2 \times A_3)$	$\left(\prod_{i=2}^{4} A_i\right)$	$\left(\prod_{i=2}^{5} A_i\right)$	$\left(\prod_{i=2}^{6} A_i\right)$	
		A_3	$(A_3 \times A_4)$	$\left(\prod_{i=3}^{5} A_i\right)$	$\left(\prod_{i=3}^{6} A_i\right)$	$\left(\prod_{i=3}^{7} A_i\right)$
			A_4	$(A_4 \times A_5)$	$\left(\prod_{i=4}^{6} A_i\right)$	$\left(\prod_{i=4}^{7} A_i\right)$
				A_5	$(A_5 \times A_6)$	$\left(\prod_{i=5}^{7} A_i\right)$
					A_6	$(A_6 \times A_7)$
						A_7

Bottom-up chain matrix multiplication (contd.)

Example (Upper triangular DP matrix for $A_1 \times A_2 \times \ldots \times A_7$)

$$
\begin{array}{ccccccc}
A_1 & (A_1 \times A_2) & \left(\prod_{i=1}^{3} A_i\right) & \left(\prod_{i=1}^{4} A_i\right) & \left(\prod_{i=1}^{5} A_i\right) & \left(\prod_{i=1}^{6} A_i\right) & \\
 & A_2 & (A_2 \times A_3) & \left(\prod_{i=2}^{4} A_i\right) & \left(\prod_{i=2}^{5} A_i\right) & \left(\prod_{i=2}^{6} A_i\right) & \left(\prod_{i=2}^{7} A_i\right) \\
 & & A_3 & (A_3 \times A_4) & \left(\prod_{i=3}^{5} A_i\right) & \left(\prod_{i=3}^{6} A_i\right) & \left(\prod_{i=3}^{7} A_i\right) \\
 & & & A_4 & (A_4 \times A_5) & \left(\prod_{i=4}^{6} A_i\right) & \left(\prod_{i=4}^{7} A_i\right) \\
 & & & & A_5 & (A_5 \times A_6) & \left(\prod_{i=5}^{7} A_i\right) \\
 & & & & & A_6 & (A_6 \times A_7) \\
 & & & & & & A_7
\end{array}
$$

Bottom-up chain matrix multiplication (contd.)

Example (Upper triangular DP matrix for $A_1 \times A_2 \times \ldots \times A_7$)

$$
\begin{array}{ccccccc}
A_1 & (A_1 \times A_2) & \left(\prod_{i=1}^{3} A_i\right) & \left(\prod_{i=1}^{4} A_i\right) & \left(\prod_{i=1}^{5} A_i\right) & \left(\prod_{i=1}^{6} A_i\right) & \left(\prod_{i=1}^{7} A_i\right) \\
 & A_2 & (A_2 \times A_3) & \left(\prod_{i=2}^{4} A_i\right) & \left(\prod_{i=2}^{5} A_i\right) & \left(\prod_{i=2}^{6} A_i\right) & \left(\prod_{i=2}^{7} A_i\right) \\
 & & A_3 & (A_3 \times A_4) & \left(\prod_{i=3}^{5} A_i\right) & \left(\prod_{i=3}^{6} A_i\right) & \left(\prod_{i=3}^{7} A_i\right) \\
 & & & A_4 & (A_4 \times A_5) & \left(\prod_{i=4}^{6} A_i\right) & \left(\prod_{i=4}^{7} A_i\right) \\
 & & & & A_5 & (A_5 \times A_6) & \left(\prod_{i=5}^{7} A_i\right) \\
 & & & & & A_6 & (A_6 \times A_7) \\
 & & & & & & A_7
\end{array}
$$

DP solution for chain matrix multiplication

```
chainMatMul(int p[0 .. n], int n) { // dimensions
  int s[1 .. n − 1, 2 .. n]; // split positions
  for i = 1 to n m[i, i] = 0; // single matrix
  for L = 2 to n { // lengths of subchains
    for i = 1 to n−L+1 { // starts of subchains
      j = i+L−1; // ends of subchains
      m[i, j] = ∞; // init to find min cost
      for k = i to j − 1 { // check all splits
        q = m[i,k] + m[k+1,j] + p[i−1]×p[k]×p[j];
        if (q < m[i, j]) { // check for lower cost
          m[i, j] = q;// found, so update
          s[i, j] = k; // record split for min cost
        }
      }
    }
  }
  return m[1, n] and s;
}
```

Optimal chain matrix multiplication

```
matMuOptl(i, j) {
  if (i == j) return A[i]; // base case
  else {
    k = s[i, j]; // split position for A_i × ... × A_j
    X = matMulOpt(i, k); // X = A_i × ... × A_k
    Y = matMulOpt(k + 1, j); // Y = A_{k+1} × ... × A_j
    return X×Y;
  }
}
```

Optimal chain matrix multiplication

```
matMuOptl(i, j) {
  if (i == j) return A[i]; // base case
  else {
    k = s[i, j]; // split position for A_i × ... × A_j
    X = matMulOpt(i, k); // X = A_i × ... × A_k
    Y = matMulOpt(k + 1, j); // Y = A_{k+1} × ... × A_j
    return X×Y;
  }
}
```

- Space requirement with DP: $O(n^2)$
- Time requirement with DP: $O(n^3)$ – DP solution scheme encounters $\frac{n}{2}$ sub-problems each of size $\frac{n}{2}$ and requiring evaluation of $\frac{n}{2} - 1$ parenthesisations
- Time requirement with naive (non DP) approach: $\Theta\left(3^{n-1}\right)$
- Better algorithms are also available

Section outline

2 **Dynamic Programming**
- Algorithm design paradigms
- DP examples
- Fibonacci number computation

Algorithm design paradigms

Divide-and-conquer (DC) Break up a problem into two or more sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form a solution to original problem

Dynamic programming (DP) Break up a problem into a series of sub-problems and use the solutions to build solutions to larger and larger sub-problems

Algorithm design paradigms

Divide-and-conquer (DC) Break up a problem into two or more sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form a solution to original problem

Dynamic programming (DP) Break up a problem into a series of sub-problems and use the solutions to build solutions to larger and larger sub-problems

- Difference between DC and DP can be confusing
- DP makes repeated use of the solutions of sub-problems
- Sub-problems identified for DP are usually overlapping, leading to identification of common smaller sub-problems later
- DP often exhibits *optimal substructure* where an optimal solution is constructed from the optimal solution of its sub-problems

DP examples

- Fibonacci number computation
- Chain Matrix Multiplication
- Coin changing
- Minimum edit distance
- Longest common subsequence
- All shortest paths (Floyd-Warshall)
- Box stacking
- Bridge building

Fibonacci number computation

- $F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-2} + F_{n-1} & n > 1 \end{cases}$
- Direct coding of this recurrence requires exponential execution time
- But, $F_{n-1} = F_{n-3} + F_{n-2}$
- If solutions of sub-problems computed earlier are reused, then the computation takes $O(n)$ additions
- Otherwise, sub-problems are solved repeatedly, wasting time
- Naive algorithm takes exponential time
- Further optimisations yield $O(\lg n)$ algorithm

$$\begin{aligned} F_{2k} &= F_k\,[2F_{k+1} - F_k] \\ F_{2k+1} &= F_{k+1}^2 + F_k^2 \end{aligned}$$

Section outline

Coin changing problem

Definition

Coin change problem

- Coin denominations can be modeled by a set of n distinct positive integer values, arranged in increasing order as $w_1 = 1$ through w_n
- Given a positive integral amount W, find non-negative integers $\{x_1, x_2, \ldots, x_n\}$ such that
  - $\sum_{i=1}^{n} x_i w_i = W$
  - $\sum_{i=1}^{n} x_i$ is minimised
- Each x_j represents the number of coins of denomination w_j used

Coin change problem

- Sometimes the greedy method of picking coins of largest denomiation works
- Coin systems for which the greedy method works are called *canonical coin systems*
- The greedy algorithm does not work arbitrary coin systems

Coin change problem

- Sometimes the greedy method of picking coins of largest denomiation works
- Coin systems for which the greedy method works are called *canonical coin systems*
- The greedy algorithm does not work arbitrary coin systems
- Consider forming coin change of 6 units in the coin system: $\{1, 3, 4\}$
- The greedy method would yield $\{4, 1, 1\}$, however, the optimal change for this system is $\{3, 3\}$

Formulation for coin changing

- Consider a denomination $w_j \leq W$
- If we know how make the change optimally for amount $W - w_j$, then we can make change for W by including a coin of denomination w_j

Formulation for coin changing

- Consider a denomination $w_j \leq W$
- If we know how make the change optimally for amount $W - w_j$, then we can make change for W by including a coin of denomination w_j
- Let $C(W, \overrightarrow{w})$ be the function that returns the change count, given the vector of denominations $\overrightarrow{w}$
- $C(0, \_) = 0$

Formulation for coin changing

- Consider a denomination $w_j \leq W$
- If we know how make the change optimally for amount $W - w_j$, then we can make change for W by including a coin of denomination w_j
- Let $C(W, \overrightarrow{w})$ be the function that returns the change count, given the vector of denominations $\overrightarrow{w}$
- $C(0, \_) = 0$
- To get optimal change for amount W, we consider all possibilities
- $C(W, \overrightarrow{w}) = \min_{w_j \leq W} C(W - w_j, \overrightarrow{w}) + 1$

Formulation for coin changing

- Consider a denomination $w_j \leq W$
- If we know how make the change optimally for amount $W - w_j$, then we can make change for W by including a coin of denomination w_j
- Let $C(W, \overrightarrow{w})$ be the function that returns the change count, given the vector of denominations $\overrightarrow{w}$
- $C(0, \_) = 0$
- To get optimal change for amount W, we consider all possibilities
- $C(W, \overrightarrow{w}) = \min_{w_j \leq W} C(W - w_j, \overrightarrow{w}) + 1$
- This function may not be defined for certain amounts
- If $\overrightarrow{w} = \{2, 3\}$, $C(1, \overrightarrow{w})$ is not defined
- We will not have this problem if $1 \in \overrightarrow{w}$

Formulation for coin changing

- Consider a denomination $w_j \leq W$
- If we know how make the change optimally for amount $W - w_j$, then we can make change for W by including a coin of denomination w_j
- Let $C(W, \overrightarrow{w})$ be the function that returns the change count, given the vector of denominations $\overrightarrow{w}$
- $C(0, \_) = 0$
- To get optimal change for amount W, we consider all possibilities
- $C(W, \overrightarrow{w}) = \min\limits_{w_j \leq W} C(W - w_j, \overrightarrow{w}) + 1$
- This function may not be defined for certain amounts
- If $\overrightarrow{w} = \{2, 3\}$, $C(1, \overrightarrow{w})$ is not defined
- We will not have this problem if $1 \in \overrightarrow{w}$
- A direct coding of this recursive formulation will lead to an exponential time solution
- But, we can efficiently form the solution bottom-up in $O(|w|W)$ time

DP solution for coin change

```
change(w[], k, W) // coins of k denominations in w
    C[0] ← 0 // initialisation for p=0
    for p ← 1 to W // do coin changes bottom-up
      min ← ∞
      for i ← 0 to k-1
        if w[i] ≤ p then // don't overpay
          // decide whether w[i] is a good choice
          if 1 + C[p − w[i]] < min then
            min ← 1 + C[p − w[i]]
            coin ← w[i]
      D[p] ← coin // best coin to pick for W
      C[p] ← min
    return C and S
```

Working of change($\{1, 2, 5\}$, 3, 14)

p:	0														
C[p]:	0														
D[p]:	0														

DP solution for coin change

```
change(w[], k, W) // coins of k denominations in w
    C[0] ← 0 // initialisation for p=0
    for p ← 1 to W // do coin changes bottom-up
      min ← ∞
      for i ← 0 to k-1
        if w[i] ≤ p then // don't overpay
          // decide whether w[i] is a good choice
          if 1 + C[p − w[i]] < min then
            min ← 1 + C[p − w[i]]
            coin ← w[i]
      D[p] ← coin // best coin to pick for W
      C[p] ← min
    return C and S
```

Working of change({1, 2, 5}, 3, 14)

p:	0	1													
C[p]:	0	1													
D[p]:	0	1													

DP solution for coin change

```
change(w[], k, W) // coins of k denominations in w
    C[0] ← 0 // initialisation for p=0
    for p ← 1 to W // do coin changes bottom-up
      min ← ∞
      for i ← 0 to k-1
        if w[i] ≤ p then // don't overpay
          // decide whether w[i] is a good choice
          if 1 + C[p − w[i]] < min then
            min ← 1 + C[p − w[i]]
            coin ← w[i]
      D[p] ← coin // best coin to pick for W
      C[p] ← min
    return C and S
```

Working of `change`($\{1, 2, 5\}$, `3, 14)`

p:	0	1	2												
C[p]:	0	1	1												
D[p]:	0	1	2												

DP solution for coin change

```
change(w[], k, W) // coins of k denominations in w
    C[0] ← 0 // initialisation for p=0
    for p ← 1 to W // do coin changes bottom-up
      min ← ∞
      for i ← 0 to k-1
        if w[i] ≤ p then // don't overpay
          // decide whether w[i] is a good choice
          if 1 + C[p − w[i]] < min then
            min ← 1 + C[p − w[i]]
            coin ← w[i]
      D[p] ← coin // best coin to pick for W
      C[p] ← min
    return C and S
```

Working of change($\{1, 2, 5\}$, 3, 14)

p:	0	1	2	3											
C[p]:	0	1	1	2											
D[p]:	0	1	2	1											

DP solution for coin change

```
change(w[], k, W) // coins of k denominations in w
    C[0] ← 0 // initialisation for p=0
    for p ← 1 to W // do coin changes bottom-up
      min ← ∞
      for i ← 0 to k-1
        if w[i] ≤ p then // don't overpay
          // decide whether w[i] is a good choice
          if 1 + C[p − w[i]] < min then
            min ← 1 + C[p − w[i]]
            coin ← w[i]
      D[p] ← coin // best coin to pick for W
      C[p] ← min
    return C and S
```

Working of change($\{1, 2, 5\}$, 3, 14)

p:	0	1	2	3	4										
C[p]:	0	1	1	2	2										
D[p]:	0	1	2	1	2										

DP solution for coin change

```
change(w[], k, W) // coins of k denominations in w
    C[0] ← 0 // initialisation for p=0
    for p ← 1 to W // do coin changes bottom-up
      min ← ∞
      for i ← 0 to k-1
        if w[i] ≤ p then // don't overpay
          // decide whether w[i] is a good choice
          if 1 + C[p − w[i]] < min then
            min ← 1 + C[p − w[i]]
            coin ← w[i]
      D[p] ← coin // best coin to pick for W
      C[p] ← min
    return C and S
```

Working of change($\{1, 2, 5\}$, 3, 14)

p:	0	1	2	3	4	5									
C[p]:	0	1	1	2	2	1									
D[p]:	0	1	2	1	2	5									

DP solution for coin change

```
change(w[], k, W) // coins of k denominations in w
    C[0] ← 0 // initialisation for p=0
    for p ← 1 to W // do coin changes bottom-up
      min ← ∞
      for i ← 0 to k-1
        if w[i] ≤ p then // don't overpay
          // decide whether w[i] is a good choice
          if 1 + C[p − w[i]] < min then
            min ← 1 + C[p − w[i]]
            coin ← w[i]
      D[p] ← coin // best coin to pick for W
      C[p] ← min
    return C and S
```

Working of change($\{1, 2, 5\}$, 3, 14)

p:	0	1	2	3	4	5	6								
C[p]:	0	1	1	2	2	1	2								
D[p]:	0	1	2	1	2	5	5								

DP solution for coin change

```
change(w[], k, W) // coins of k denominations in w
    C[0] ← 0 // initialisation for p=0
    for p ← 1 to W // do coin changes bottom-up
      min ← ∞
      for i ← 0 to k-1
        if w[i] ≤ p then // don't overpay
          // decide whether w[i] is a good choice
          if 1 + C[p − w[i]] < min then
            min ← 1 + C[p − w[i]]
            coin ← w[i]
      D[p] ← coin // best coin to pick for W
      C[p] ← min
    return C and S
```

Working of `change({1,2,5}, 3, 14)`

p:	0	1	2	3	4	5	6	7							
C[p]:	0	1	1	2	2	1	2	2							
D[p]:	0	1	2	1	2	5	5	2							

DP solution for coin change

```
change(w[], k, W) // coins of k denominations in w
    C[0] ← 0 // initialisation for p=0
    for p ← 1 to W // do coin changes bottom-up
      min ← ∞
      for i ← 0 to k-1
        if w[i] ≤ p then // don't overpay
          // decide whether w[i] is a good choice
          if 1 + C[p − w[i]] < min then
            min ← 1 + C[p − w[i]]
            coin ← w[i]
      D[p] ← coin // best coin to pick for W
      C[p] ← min
    return C and S
```

Working of `change(`$\{1,2,5\}$`, 3, 14)`

p:	0	1	2	3	4	5	6	7	8						
C[p]:	0	1	1	2	2	1	2	2	3						
D[p]:	0	1	2	1	2	5	5	2	1						

DP solution for coin change

```
change(w[], k, W) // coins of k denominations in w
    C[0] ← 0 // initialisation for p=0
    for p ← 1 to W // do coin changes bottom-up
      min ← ∞
      for i ← 0 to k-1
        if w[i] ≤ p then // don't overpay
          // decide whether w[i] is a good choice
          if 1 + C[p − w[i]] < min then
            min ← 1 + C[p − w[i]]
            coin ← w[i]
      D[p] ← coin // best coin to pick for W
      C[p] ← min
    return C and S
```

Working of `change(`$\{1, 2, 5\}$`, 3, 14)`

p:	0	1	2	3	4	5	6	7	8	9					
C[p]:	0	1	1	2	2	1	2	2	3	3					
D[p]:	0	1	2	1	2	5	5	2	1	2					

DP solution for coin change

```
change(w[], k, W) // coins of k denominations in w
    C[0] ← 0 // initialisation for p=0
    for p ← 1 to W // do coin changes bottom-up
      min ← ∞
      for i ← 0 to k-1
        if w[i] ≤ p then // don't overpay
          // decide whether w[i] is a good choice
          if 1 + C[p − w[i]] < min then
            min ← 1 + C[p − w[i]]
            coin ← w[i]
      D[p] ← coin // best coin to pick for W
      C[p] ← min
    return C and S
```

Working of change($\{1, 2, 5\}$, 3, 14)

p:	0	1	2	3	4	5	6	7	8	9	10				
C[p]:	0	1	1	2	2	1	2	2	3	3	2				
D[p]:	0	1	2	1	2	5	5	2	1	2	5				

DP solution for coin change

```
change(w[], k, W) // coins of k denominations in w
    C[0] ← 0 // initialisation for p=0
    for p ← 1 to W // do coin changes bottom-up
      min ← ∞
      for i ← 0 to k-1
        if w[i] ≤ p then // don't overpay
          // decide whether w[i] is a good choice
          if 1 + C[p − w[i]] < min then
            min ← 1 + C[p − w[i]]
            coin ← w[i]
      D[p] ← coin // best coin to pick for W
      C[p] ← min
    return C and S
```

Working of change($\{1, 2, 5\}$, 3, 14)

p:	0	1	2	3	4	5	6	7	8	9	10	11			
C[p]:	0	1	1	2	2	1	2	2	3	3	2	3			
D[p]:	0	1	2	1	2	5	5	2	1	2	5	1			

DP solution for coin change

```
change(w[], k, W) // coins of k denominations in w
    C[0] ← 0 // initialisation for p=0
    for p ← 1 to W // do coin changes bottom-up
      min ← ∞
      for i ← 0 to k-1
        if w[i] ≤ p then // don't overpay
          // decide whether w[i] is a good choice
          if 1 + C[p − w[i]] < min then
            min ← 1 + C[p − w[i]]
            coin ← w[i]
      D[p] ← coin // best coin to pick for W
      C[p] ← min
    return C and S
```

Working of change($\{1, 2, 5\}$, 3, 14)

p:	0	1	2	3	4	5	6	7	8	9	10	11	12		
C[p]:	0	1	1	2	2	1	2	2	3	3	2	3	3		
D[p]:	0	1	2	1	2	5	5	2	1	2	5	1	2		

DP solution for coin change

```
change(w[], k, W) // coins of k denominations in w
    C[0] ← 0 // initialisation for p=0
    for p ← 1 to W // do coin changes bottom-up
      min ← ∞
      for i ← 0 to k-1
        if w[i] ≤ p then // don't overpay
          // decide whether w[i] is a good choice
          if 1 + C[p − w[i]] < min then
            min ← 1 + C[p − w[i]]
            coin ← w[i]
      D[p] ← coin // best coin to pick for W
      C[p] ← min
    return C and S
```

Working of **change**($\{1, 2, 5\}$, **3, 14**)

p:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
C[p]:	0	1	1	2	2	1	2	2	3	3	2	3	3	4	
D[p]:	0	1	2	1	2	5	5	2	1	2	5	1	2	1	

DP solution for coin change

```
change(w[], k, W) // coins of k denominations in w
    C[0] ← 0 // initialisation for p=0
    for p ← 1 to W // do coin changes bottom-up
      min ← ∞
      for i ← 0 to k-1
        if w[i] ≤ p then // don't overpay
          // decide whether w[i] is a good choice
          if 1 + C[p − w[i]] < min then
            min ← 1 + C[p − w[i]]
            coin ← w[i]
      D[p] ← coin // best coin to pick for W
      C[p] ← min
    return C and S
```

Time complexity of DP solution: Linear in W, exponential in number of bits of W

Working of `change(`$\{1, 2, 5\}$`, 3, 14)`

p:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
C[p]:	0	1	1	2	2	1	2	2	3	3	2	3	3	4	4
D[p]:	0	1	2	1	2	5	5	2	1	2	5	1	2	1	2