Dynamic Programming

1 0/1 Knapsack

1.1 Algorithm

n items are given. The profit is stored in an array called v and the weights are stored in an array called w. The current index is represented by i and the weight is represented by W. The memoization matrix is m whose size is $(n+1) \times (W+1)$

Algorithm 1 0/1 Knapsack (Top-down Approach)

procedure PROFIT(i, W)
 if $i < 0$ **then**
  return 0
 if $m_{iW} \neq \text{NIL}$ **then**
  return m_{iW}
 if $w_{i-1} > W$ **then**
  $m_{iW} \leftarrow \text{PROFIT}(i-1, W)$
 else
  $m_{iW} \leftarrow \text{MAX}(\text{PROFIT}(i-1, W-w_{i-1}) + v_{i-1}, \text{PROFIT}(i-1, W))$
 return m_{iW}

Algorithm 2 0/1 Knapsack (Bottom-up approach)

procedure PROFIT(n, W)
 for $i \leftarrow 0, 1, \ldots, n$ **do**
  for $w \leftarrow 0, 1, \ldots, W$ **do**
   if $i = 0$ **or** $w = 0$ **then**
    $m_{iw} \leftarrow 0$
   else if $w_{i-1} \leq W$ **then**
    $m_{iw} \leftarrow \text{MAX}\left(v_{i-1} + m_{(i-1)(W-w_{i-1})}, m_{(i-1)w}\right)$
   else
    $m_{iw} \leftarrow m_{(i-1)w}$
 return m_{nW}

1.2 Time Complexity

From the bottom-up approach, it is clear that the time complexity of the algorithm is $O(n \times W)$. The space complexity is also $O(n \times W)$. However, the space can be optimized to $O(W)$

Algorithm 3 0/1 Knapsack (Optimized Space)

procedure PROFIT(n, W)
  for $i \leftarrow 0, 1, \ldots, n$ **do**
    for $w \leftarrow 0, 1, \ldots, W$ **do**
      if $w_{i-1} \leq W$ **then**
        $m_w \leftarrow \text{MAX}\left(v_{i-1} + m_{W-w_{i-1}}, m_w\right)$
  return m_W

2 Matrix Chain Multiplication

2.1 Algorithm

The dimensions of the matrices are saved in an array d. i is the starting index and j is the ending index. The memoization matrix is DP whose size is $n \times n$, where n is the length of d i.e. 1+ the number of matrices.

Algorithm 4 Matrix Chain Multiplication (Top-Down Approach)

procedure MATMULT(i, j)
  if $i + 1 = j$ **then** ▷ There's only one matrix
    return 0
  if $\text{DP}_{ij} \neq \text{NIL}$ **then**
    return DP_{ij}
  $p \leftarrow \infty$
  for $k \leftarrow (i+1) \ldots j$ **do**
    $c \leftarrow \text{MATMULT}(i, k) + \text{MATMULT}(k, j) + d_i \times d_k \times d_j$
    $p \leftarrow \text{MIN}(c, p)$
  $\text{DP}_{ij} \leftarrow p$
  return p

Here, n is the length of d (the array containing the size of the matrices). The tabulation matrix is DP.

Algorithm 5 Matrix Chain Multiplication (Bottom-Up Approach)

procedure MATMULT(n)
  for $l \leftarrow 2, 3, \ldots, n$ **do** ▷ l is the chain length
    for $i \leftarrow 0, 1, \ldots, (n - l)$ **do**
      $j \leftarrow i + l$
      $\text{DP}_{ij} \leftarrow \infty$
      for $k \leftarrow (i+1) \ldots j$ **do**
        $c \leftarrow \text{DP}_{ik} + \text{DP}_{kj} + d_i \times d_k \times d_j$
        $\text{DP}_{ij} = \text{MIN}(\text{DP}_{ij}, c)$
  return $\text{DP}_{0(n-1)}$

2.2 Complexity analysis

The time complexity of this algorithm is $O(n^3)$ and the space complexity is $O(n^2)$

3 Count Inversions

3.1 Algorithm

3.2 Complexity Analysis

The time complexity is $O(nlogn)$, same as merge sort and the space complexity is $O(n)$

Algorithm 6 Count inversions of an array i.e. number of instances where $A_i > A_j$ for $i < j$ using Merge Sort

```
procedure COUNT(A, l, r)
    c ← 0
    if l < r then
        m ← ⌊(l+r)/2⌋
        c ← c + COUNT(A, l, m)
        c ← c + COUNT(A, m + 1, r)
        c ← c + COUNT_AND_MERGE(A, l, m, r)
    return c
procedure COUNT_AND_MERGE(A, l, m, r)
    n1 ← m − l + 1                          ▷ Length of first subarray
    n2 ← r − m                              ▷ Length of second subarray
    L ← A_{l,...,m}                         ▷ Left Subarray
    R ← A_{m+1,...,r}                       ▷ Right Subarray
    c ← 0                                   ▷ Inversion Count
    i ← l                                   ▷ Index for left subarray
    j ← 0                                   ▷ Index for right subarray
    k ← l                                   ▷ Index for updating the array
    while i < n1 and j < n2 do
        if L_i <= R_j then                  ▷ No inversion
            A_k ← L_i
            i ← i + 1
        else
            A_K ← R_j
            j ← j + 1
            c ← c + (n1 − i)
        k ← k + 1
    while i < n1 do
        A_k ← L_i
        j ← j + 1
        k ← k + 1
    while j < n2 do
        A_k ← R_j
        j ← j + 1
        k ← k + 1
    return c
```