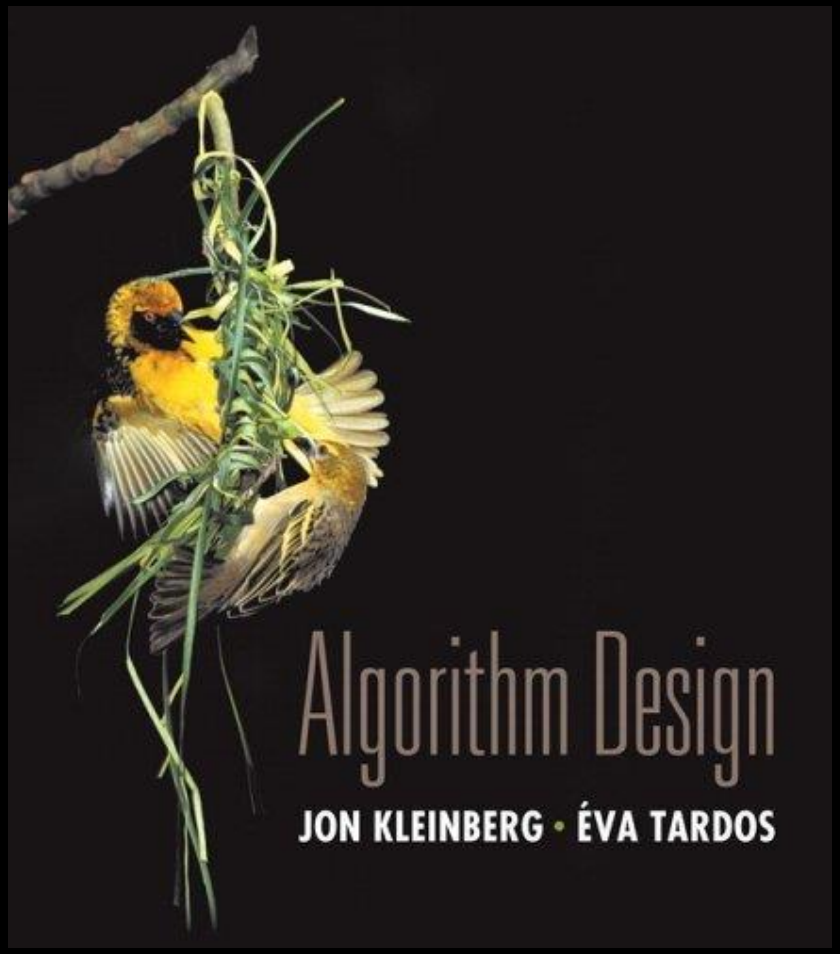

Chapter 6

Dynamic Programming

6.1 Weighted Interval Scheduling

Weighted Interval Scheduling: DP 2 - Bottom-Up

Dynamic programming to fill up the memoization table M:

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute lastCompat(1), lastCompat(2), …, lastCompat(n)
M = [] * n

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      #Option of having item j in the optimal solution
      of subproblem OPT(j)
      If vj + M[lastCompat(j)] > M[j-1]
          M[j] = vj + M[lastCompat(j)]
      #Option of not having item j in the optimal
      solution of subproblem OPT(j)
      Else
          M[j] = M[j-1]
}
```

Weighted Interval Scheduling: DP 2 - Bottom-Up

Obtaining the optimal solution for each subproblem, not just value:

- Array Sol stores optimal solution relative to OPT(i) in Sol(i)

```
Iterative-Compute-Opt {
   M[0] = 0
   Sol[0] = {}
   for j = 1 to n
      #Option of having item j in the optimal solution
      of subproblem OPT(j)
      If v_j + M[lastCompat(j)] > M[j-1]
          M[j] = v_j + M[lastCompat(j)]
          Sol[j] = Sol[lastCompat(j)] ∪ {j}
      #Option of not having item j in the optimal
      solution of subproblem OPT(j)
      Else
          M[j] = M[j-1]
          Sol[j] = Sol[j-1]

   Return Sol[n]
}
```

Weighted Interval Scheduling: DP 2 - Bottom-Up

Very important: When the code enters this "If" it does not mean that item j is in the optimal solution for the full problem...

... it only mean that item j belongs to the optimal solution for subproblem OPT(j)

```
        #Option of having item j in the optimal solution
        # subproblem OPT(j)
        If v_j + M[lastCompat(j)] > M[j-1]
            M[j] = v_j + M[lastCompat(j)]
            Sol[j] = Sol[lastCompat(j)] ∪ {j}
        #Option of not having item j in the optimal
        solution of subproblem OPT(j)
        Else
            M[j] = M[j-1]
            Sol[j] = Sol[j-1]

    Return Sol[n]
}
```

Weighted Interval Scheduling: DP 2 - Bottom-Up

Very important 2: Never use Sol in the recurrence relatio, 99% chance of getting it wrong

First write recurrence relation/algo just to fill up memoization table M, then add Sol

```
        of subproblem OPT(j)
        If v_j + M[lastCompat(j)] > M[j-1]
            M[j] = v_j + M[lastCompat(j)]
            Sol[j] = Sol[lastCompat(j)] ∪ {j}
        #Option of not having item j in the optimal
        solution of subproblem OPT(j)
        Else
            M[j] = M[j-1]
            Sol[j] = Sol[j-1]

    Return Sol[n]
}
```

Maior subsequência crescente

Maior subsequência crescente: encontrando o tamanho

```
Input: n, a1,…,an ,

M(1)←1
For j=2 to n
  M(j)←1
  For i=1 to j-1 // faz OPT(i) <- max_i { 1+OPT(i) | i<j e a(j)>a(i) }
       If A(i) < A(j) and 1+M(i)>M(j) then
           M(j) ← 1+M(i)
       End If
  End For
End For

max ← 0, maxi ← 0
For i=1 to n          // encontra indice do maior OPT(i)
 If M(j) > max, set max ← M(j) and maxi ← i
End For

Return M(maxi)
```

Maior subsequência crescente: encontrando o tamanho

```
Input: n, a_1,…,a_n ,

M(1)←1
Sol(1)={a_1}
For j=2 to n
  M(j)←1
  Sol(j) ← a_j
  For i=1 to j-1 // faz OPT(i) <- max_i { 1+OPT(i) | i<j e a(j)>a(i) }
        If A(i) < A(j) and 1+M(i)>M(j) then
            M(j) ← 1+M(i)
            Sol(j) ← concatenation of Sol(i) plus a_j
        End If
  End For
End For

max ← 0, maxi ← 0
For i=1 to n          // encontra indice do maior OPT(i)
  If M(j) > max, set max ← M(j) and maxi ← i
End For

Return Sol(maxi)
```

6.4 Knapsack Problem

Knapsack Problem: Bottom-Up

```
Input: n, w_1,…,w_N, v_1,…,v_N

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (w_i > w)
         M[i, w] = M[i-1, w]
      else
         if v_i + M[i-1, w-w_i] > M[i-1, w]
             M[i, w] = v_i + M[i-1, w-w_i]
         else
             M[i, w] = M[i-1, w]

return M[n, W]
```

Knapsack Problem: Bottom-Up

```
Input: n, w_1,…,w_N, v_1,…,v_N

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (w_i > w)
         M[i, w] = M[i-1, w]
         Sol[i, w] = Sol[i-1, w]
      else
         if v_i + M[i-1, w-w_i] > M[i-1, w]
             M[i, w] = v_i + M[i-1, w-w_i]
             Sol[i, w] = Sol[i-1, w-w_i] ∪ {i}
         else
             M[i, w] = M[i-1, w]
            Sol[i, w] = Sol[i-1, w]

return Sol[n, W]
```