# Dynamic Programming

## 1 Introduction

Dynamic programming is a method used in algorithms to solve complex problems by breaking them down into simpler subproblems. It's a technique for solving problems efficiently by storing the results of expensive function calls and reusing them when the same inputs occur again, thus reducing the computation time.

### 1.1 Properties

There are two main properties of dynamic programming:

1. **Optimal Substructure**: This means that the optimal solution to the problem can be constructed from the optimal solutions of its subproblems. In other words, a problem has an optimal substructure if an optimal solution can be created by combining optimal solutions to its subproblems.

2. **Overlapping Subproblems**: This occurs when the problem can be broken down into subproblems which are reused several times. In contrast to divide-and-conquer algorithms that solve each subproblem fresh, dynamic programming saves the result of these subproblems to avoid recomputing their solutions.

### 1.2 Implementation

Dynamic programming can be implemented in two ways:

1. **Top-Down Approach (Memoization)**: This approach involves starting from the top and breaking the problem down into subproblems. As solutions to subproblems are computed, they are stored in a memory structure (such as a hash table or array). If the same subproblem occurs again, the solution is retrieved from the memory instead of being recomputed. This approach uses recursion and memoization.

2. **Bottom-Up Approach (Tabulation)**: This method starts from the simplest subproblems and iteratively solves larger subproblems using the solutions to smaller subproblems. It fills up a table (usually an array or a matrix) in a way that every step towards solving the larger problem builds on the solutions of the smaller subproblems. This approach is iterative and often more space-efficient than the top-down approach.

### 1.3 Comparison with Greedy Algorithm

Both algorithms are to find the optimal substructures which are constructed from optimal solutions to subproblems. However, for Greedy algorithm:

1. It does not guarantee optimality.
2. Make decisions based on local subproblem; once a choice is makde, it is not changed.

For Dynamic Programming:

1. It guarantees optimality; equivalent to exhaustive search; efficient because of the reuse of subproblems.
2. Makes decisions based on all the decisions made in the previous stage, and may reconsider the previous stage's algorithmic path to solution.

## 2 Recipe

The general approach of the DP problem is:

1. Characterize structure of problem: identify **subproblems** whose optimal solutions can be used to build an optimal solution to original problem. Conversely, given an optimal solution to original problem, identify subparts of the solution that are optimal solutions for some subproblems.
2. Write the **recurrence** and **initial cases**, know where the solution of problem is.
3. Look at precedence constraints (draw a figure) and write the algorithm (iterative, or recursive with memos).
4. Study the problem complexity (straightforward with iterative algorithm; don't forget the time to compute one subproblem).
5. Construct optimal solution from computed information (back-tracing).

# 3 Weighted Interval Scheduling

## 3.1 Problem Definition

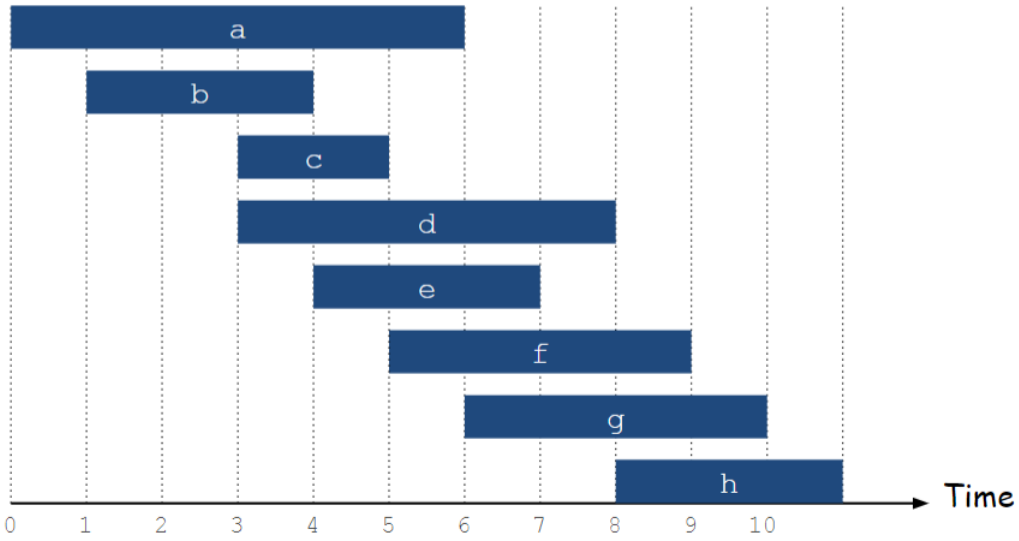


Figure 1: Weighted Interval Scheduling

Assume job $j$ starts at $s_j$, finishes at $f_j$, and has weight $v_j$. Define two jobs compatible if they don't overlap. The goal is to **find maximum weight** subset of mutually compatible jobs.

## 3.2 Failure of Greedy Algorithm

Recall that the greedy algorithm when all weights are 1:

1. Consider jobs in ascending order of finish time
2. Add job to subset if it is compatible with previously chosen jobs

But this algorithm will easily fail if arbitrary weights are allowed:

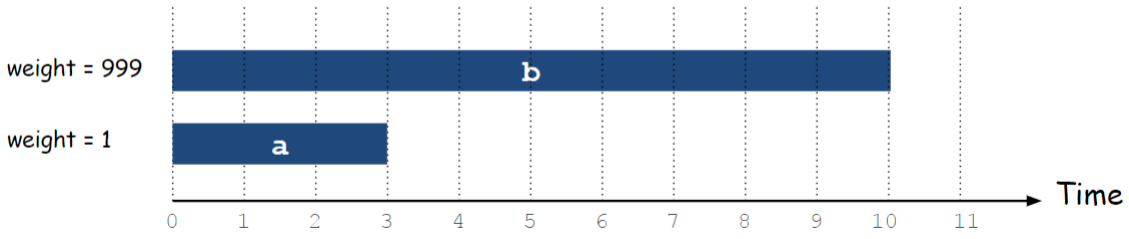


Figure 2: Failure of Greedy Algorithm

## 3.3 Binary Choice

Now consider an optimal solution $O$ for jobs $\{1, 2, ..., n\}$. No matter what $O$ is, for job $n$, **we have two cases: either $O$ contains the last job $n$, or $O$ does not contain the last job $n$.** Define $p(j)$ as the largest index $i < j$ such that job $i$ is compatible with $j$.

### 3.3.1 $O$ Contains Job $n$

If $O$ contains job $n$, then for the remaining part of the solution $O - \{n\}$:

1. $O-\{n\}$ could not contain any job that is incompatible with $n$ (job $p(n)+1, p(n)+2, \cdots, n-1$), or in other words, it only contains jobs in $\{1, 2, \cdots, p(n)\}$
2. Since we assume $O$ is feasible, so $O-\{n\}$ is also a feasible solution for the problem of scheduling $\{1, 2, \cdots, p(n)\}$
3. Also, $O - \{n\}$ **must be an optimal solution for scheduling** $\{1, 2, \cdots, p(n)\}$. Otherwise, we can take the optimal solution for $\{1, 2, \cdots, p(n)\}$ and safely add job $n$ to it, obtain an overall solution $O'$ better than the given optimal solution $O$.

### 3.3.2 $O$ Does not Contain Job $n$

If $O$ does not contain job $n$:

1. Then $O$ is a feasible solution for scheduling jobs $\{1, 2, \cdots, n-1\}$
2. If $O$ is not the optimal solution for $\{1, 2, \cdots, n-1\}$, then we can replace it with the optimal solution and obtain a better solution also for scheduling $1, 2, \cdots, n$
3. Therefore, $O$ must contain the optimal solution for scheduling $1, 2, \cdots, n-1$

### 3.3.3 Final Expression

Define $OPT(j)$ = **value of optimal solution to the problem consisting of jobs** $\{1, 2, \cdots, j\}$, with two cases:

- $OPT(j)$ selects job $j$: must include optimal solution to problem consisting of remaining compatible jobs $\{1, 2, \cdots, p(j)\}$ with value $OPTp(j)$ and collect the profit $v_j$ from including $j$, so the expression is:

$$OPT(j) = v(j) + OPT(p(j)) \tag{1}$$

- $OPT(j)$ does not select job $j$: must include optimal solution to problem consisting of remaining compatible jobs $\{1, 2, \cdots, j\}$, so the expression is:

$$OPT(j) = OPT(j-1) \tag{2}$$

So the final **recurrence relation** is:

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \text{otherwise} \end{cases} \tag{3}$$

## 3.4 Memoization Implementation

In memoization, we store results of each sub-problem in a cache, lookup as needed. The pseudo code is shown below:

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

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), ..., p(n)

for j = 1 to n
  M[j] = empty
M[0] = 0                    global array
M-Compute-Opt(n)


M-Compute-Opt(j) {
  if (M[j] is empty)
    M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
}
```

Figure 3: Memoization Implementation

Now we do the runtime analysis for memoization:

1. The sorting by finish time will take $O(n \log n)$
2. To compute $p()$, we could use binary search for each $j$ to find the satisfied job, which will take $O(\log n)$. Since we do this for each of the $n$ jobs, the total time for this step is $O(n \log n)$
3. For the function M-Compute-Opt(j), each invocation takes $O(1)$ time, either:
   - Return an existing value $M[j]$
   - Fills in one new entry $M[j]$ and makes two recursive calls (constant)

   We need to call M-Compute-Opt(j) $n$ times, so the runtime will be $O(n)$.

Therefore, the runtime for memoization implementation is $O(n \log n)$. But if jobs are pre-sorted by start and finish times, the runtime could reduce to $O(n)$.

## 3.5 Tabulation Implementation

For the bottom-up dynamic programming, the pseudo code is shown below:

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

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

Compute p(1), p(2), ..., p(n)

Iterative-Compute-Opt {
  M[0] = 0
  for j = 1 to n
    M[j] = max(vj + M[p(j)], M[j-1])
}
```

Figure 4: Tabulation Implementation

Now we do the runtime analysis for this implementation, it is nearly the same as memoization:

1. Sorting will take $O(n \log n)$
2. Computing $p()$ will take $O(n \log n)$
3. M-Compute-Opt(j):
   - Initializing $M[0] = 0$ takes constant time, $O(1)$
   - The loop runs from 1 to $n$, each iteration involves a constant amount of work, computing the maximum of two values. Therefore, the time complexity is $O(n)$

Therefore, the runtime for tabulation implementation is $O(n \log n)$. But if jobs are pre-sorted by start and finish times, the runtime could reduce to $O(n)$.

## 3.6 Back-Tracing

The back-tracing function is implemented as below:

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
  if (j = 0)
    output nothing
  else if (vj + M[p(j)] > M[j-1])
    print j
    Find-Solution(p(j))
  else
    Find-Solution(j-1)
}
```

Figure 5: Back Tracing Implementation

The worst case, Find-Solution calls itself every time until $j$ reaches 0, so the time complexity is $O(n)$.

# 4 Longest Common Subsequence (LCS)

## 4.1 Problem Definition

Given two strings/sequences, for example $X = A, B, C, B, D, A, B$; $Y = B, D, C, A, B, A$. Now we want to find the Longest Common Subsequence (a sequence of letters that appears in both X and Y but not necessarily contiguously). Here.

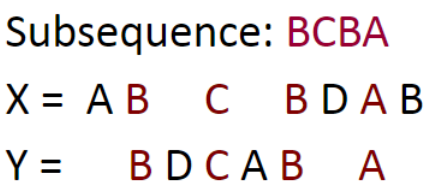


Figure 6: LCS

## 4.2 Algorithm

First define two sequences:

$$X_m = \{x_1, x_2, \cdots, x_m\} \tag{4}$$

$$Y_n = \{y_1, y_2, \cdots, y_n\} \tag{5}$$

And also define $Z$ as the optimal solution, which is just a LCS of $X$ and $Y$:

$$Z_k = \{z_1, z_2, \cdots, z_k\} \tag{6}$$

Now we consider the problem case by case:

1. if $x_m = y_n$: then by the definition of LCS, we have $x_m = y_n = z_k$. This implies that $Z_{k-1}$ must be in $LCS(X_{m-1}, Y_{n-1})$.
2. if $x_m \neq y_n$:
   - Either $x_m \neq z_k$: which means $Z_k$ does not contain $x_m$, and $Z_k$ is in $LCS(X_{m-1}, Y_n)$
   - Or $x_m = z_k$ and $y_n \neq z_k$: which means $Z_k$ does not contain $y_n$, and $Z_k$ is in $LCS(X_m, Y_{n-1})$

The **update process** will be:

1. If $x_m = y_n$, find solution to $LCS(X_{m-1}, Y_{n-1})$ and append $x_m$.
2. If $x_m \neq y_n$, find solution for **each of the two subproblems** $LCS(X_{m-1}, Y_n)$ and $LCS(X_m, Y_{n-1})$, and choose the longer one.

Notice that there is an **overlapping problem** here, $LCS(X_{m-1}, Y_{n-1})$ can appear as a subproblem when solving $LCS(X_{m-1}, Y_n)$ and $LCS(X_m, Y_{n-1})$.

Therefore, the final recurrence relation could be expressed as:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise} \end{cases} \tag{7}$$

## 4.3 Memoization Implementation

The pseudo code is shown below:

```
memo = { }
c(i, j):
    if (i, j) in memo: return memo[i, j]
    else if i=0 OR j=0: return 0
    else if x_i=y_j: f = c(i-1, j-1)+1
    else f = max{c(i, j-1), c(i-1, j)}
    memo[i, j]=f
    return f
return c(m,n)
```

Figure 7: LCS Memoization Implementation

Each subproblem $c[i, j]$ is computed only once, with constant recursive calls per time. There are at most $m \times n$ subproblems, so the time and space complexities are both $O(m \times n)$.

## 4.4 Tabulation Implementation

The pseudo code is shown below:

```
0. m = length(X)  // get the # of symbols in X
1. n  = length(Y) // get the # of symbols in Y
2. allocate matrix c of size (m+1)x(n+1)
3. for i = 1 to m    c[i,0] = 0   // special case: Y_0
4. for j = 1 to n    c[0,j] = 0   // special case: X_0

5. for i = 1 to m              // for all X_i (rows)
6.     for j = 1 to n          // for all Y_j (columns)
7.       if ( Xi == Yj )
8.           c[i,j] = c[i-1,j-1] + 1 // match
9.       else c[i,j] = max( c[i-1,j], c[i,j-1] )
10. return c
```

Figure 8: LCS Tabulation Implementation

Because there are two nested loop, so the time and space complexities are both $O(m \times n)$.

### 4.5 Back-Tracing

The pseudo code is shown below:

```
Run Find-Solution(m,n)

Find-Solution(i, j) {
  if (i = 0 or j = 0)
    return
  else
    if (x_i = y_j)
      print x_i
      Find-Solution(i-1, j-1)
    else
      if (c[i-1,j] > c[i, j-1])
        Find-Solution(i-1, j)
      else
        Find-Solution(i, j-1)
}
```

Figure 9: LCS Back Tracing Implementation

## 5 Knapsack Problem

### 5.1 Problem Definition

Given $n$ items, and define item $i$ weighs $w_i > 0$ and has value $v_i > 0$. Assume knapsack has weight capacity of $W$. The goal is to pack knapsack so as to maximize total value.

### 5.2 Failure of Greedy Algorithm

Notice this problem is different from the homework problem. In the homework problem, the item could be added as a fraction, but here could not. Therefore, the greedy algorithm to repeatedly add item with maximum ratio $\frac{v_i}{w_i}$ will fail.

### 5.3 Algorithm

In this problem, there are two constraints, one is to maximize value, the other is to satisfy the weight capacity requirement. Therefore, **we need two variables for the subproblem**.

Define $OPT(i, w)$ as the max-profit subset of items $\{1, 2, \cdots, i\}$ with weight limit $w$. The goal is to find $OPT(n, W)$. There are two possible cases:

1. $OPT(i, w)$ **does not select item** $i$ **because of the weight limit**: $OPT(i, w)$ will become $OPT(i-1, w)$, selecting the best of $\{1, 2, \cdots, i-1\}$ with weight limit as $w$.

2. $OPT(i, w)$ **selects item** $i$:

- Collect value $v_i$
- New weight limit will become $w - w_i$
- $OPT(i, w - w_i)$ selects best of $\{1, 2, \cdots, i-1\}$ using new weight limit.

Therefore, the final recurrence relation could be expressed as:

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{OPT(i-1, w), v_i + OPT(i-1, w - w_i)\} & \text{otherwise} \end{cases} \tag{8}$$

# 6 Summary

The general forms of the dynamic programming include:

1. Binary choice: Weighted interval scheduling
2. Multi-way choice: RNA secondary structure
3. Dynamic programming over intervals: RNA secondary structure
4. Adding a new variable: Knapsack