DYNAMIC PROGRAMMING CONCEPTS AND APPLICATION REVIEW PAPER

Deepansha Singh
West Windsor Plainsboro High School South, New Jersey

Abstract :

Understanding and applying dynamic programming algorithmic concepts is crucial towards solving real world problems. This review paper delves into important concepts, various problems, and an application of dynamic programming. From DNA sequence alignment to efficient scheduling, this optimization method is used in various fields today.

Contents

1	**Introduction**	**3**
2	**Problem #1 : Weighted Interval Scheduling**	**4-6**
3	**Problem #2 : Knapsack Problem**	**7-8**
5	**Application : DNA Sequence Alignment**	**9-12**
6	**Conclusion**	**13**

Dynamic Programming Introduction :

Dynamic programming is an optimization technique which breaks down the problem into subproblems. Unlike other algorithmic methods, some subproblems in dynamic programming may overlap. Additionally, it has a faster run time than brute force and various other algorithmic techniques, due to the subproblems which are constructed from the original problem. The polynomial run time is usually $O(n^2)$ or $O(n^3)$. Additionally, a divide and conquer method is used to tackle the problem. After identifying the input, output, constraints, and cases, a recurrence relation is made. Using this recurrence relation, an efficient algorithm is created to solve the particular problem by breaking it into subproblems.

Problem#1 - Weighted Interval Scheduling

Objective : Find largest set of non-overlapping/compatible intervals, where the sum of the weights is maximized

Weighted interval scheduling problem consists of a particular resource. Let us take a scenario where everyone wants to book a room, where each interval consists of a start time, finish time, and weight specification. The goal is to find a schedule, where the sum of the weights is maximized and all constraints are met. Note that this problem can be solved with a greedy algorithm, if all edges are of weight one. However, in many real world scenarios, the weight is not 1. Compatible intervales mean two consecutive intervals are nonoverlapping, and the terms "compatible" and "non-overlapping" can be used interchangeably in this dynamic programming problem.

Input :

- ➢ n requests/intervals where each request is labeled from $1, 2, ..., n$
  - Each request :
    - Start time specification, s_i, where $i \in 1, 2, ..., n$
    - Finish time specification, f_i, where $i \in 1, 2, ..., n$
      - Assume finish times are distinct and are ordered in non-decreasing order
    - Weight specification, w_i, where $i \in 1, 2, ..., n$

Let us define a solution $OPT[j]$, which finds and outputs the maximum weight of the non-overlapping/compatible intervals from intervals $1, 2, ..., j$. We want OPT[n]. Let us define 2 cases :

Case I : j^{th} interval is not an optimal solution

$$OPT[j] = OPT[j-1]$$

Case II : j^{th} interval is an optimal solution

Before we make a recurrence relation, we must define the following function :

$prev(j)$ = interval with largest finish time, f_i (index), s.t. previous interval and current interval don't overlap

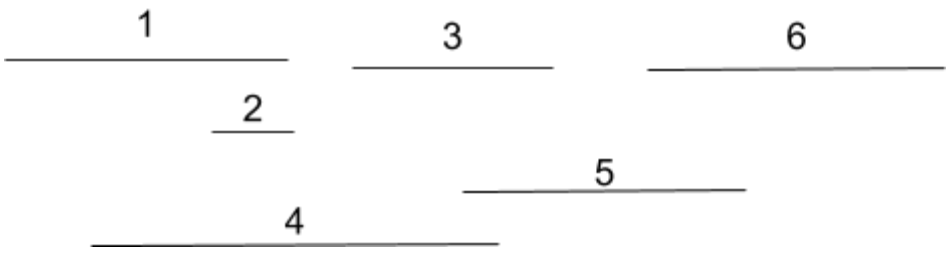


Figure 1

When viewing Figure 1, it is clear that $prev(6) = 4$, $prev(5) = 2$, and $prev(4) = 0$.

Recurrence relation :

$$OPT(j) = \begin{cases} 0 & \text{, if } j = 0 \\ max(OPT[j-1], OPT[prev] + p_j) & \end{cases}$$

Below, is the algorithm which correctly implements the recurrence relation shown above. This run time is $O(n)$, because the loop traverses from the first element to the last, where $j \in 1, 2, ..., n$. Unlike other algorithmic techniques which would take a longer run time (exponential), dynamic programming implements the recurrence relation efficiently, which yields a shorter run time.

$Compute - OPT$ {

$OPT[0] = 0$

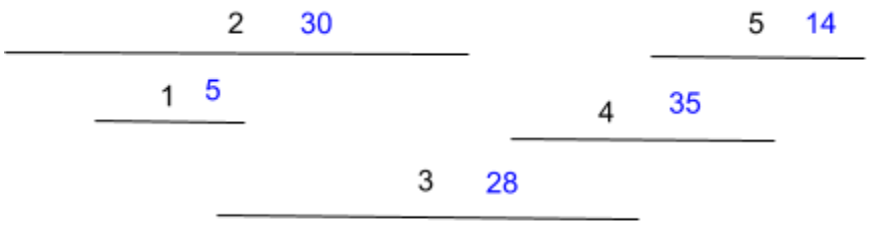
For $j = 1$ to n

$OPT[j] = max(v_j + OPT[p(j)], OPT[j-1])$

}

Return $OPT[n]$

Example of Weighted Interval Schdreeduling:



0	1	2	3	4	5
0	5	30	30	65	65

`

Figure 2

Problem #2 - Knapsack Problem

Objective : Pack the items in the knapsack such that the weight constraints are met and total profit is maximized

Knapsack problem consists of a knapsack, where items need to be placed such that the capacity constraint is considered, and profit is maximized. Each item has its own profit, and can only be added to the knapsack, i.f.f. it's addition to the bag doesn't exceed the capacity.

Input :

- Knapsack
- Capacity c
- n items numbered from $1, 2, ..., n$
- Item i, where $i \in 1, 2, ..., n$
  - Weight constraint w, where $w_i > 0$ & $i \in 1, 2, ..., n$
  - Profit p_i. where $p_i > 0$ & $i \in 1, 2, ..., n$

Let us define the solution $OPT(i)$ = max profit obtained when considering items $1, 2, .., i$, where all constraints are met. Below is a WRONG approach to solving problem. Note that this solution is wrong, because more subproblems are needed to solve the problem.Thus, with dynamic programming, a more optimal technique, we can devise a better method to solve the Knapsack problem.

$$OPT(i) = OPT(i-1) \leftarrow i \notin OPT$$
$$p_i + OPT(i-1) \leftarrow i \in OPT$$

Below is the CORRECT Solution:

Let us define a solution, $T[i, c]$ = max profit obtained by considering items $1, 2, 3, i$, where the knapsack has a capacity of c. This capacity takes the weight variable for each item into account. This solution will consider all subcases, due to the dynamic programming algorithmic approach taken.

Recurrence relation:

$$T[i,c] = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w >= c \\ maxT[i-1,c], T[i-1,c-w_i] + p_i & \text{otherwise} \end{cases}$$

$ComputeT[i,c]$

For $c \leftarrow 0$ to w do

$T[0,C] \leftarrow 0$ For $i \leftarrow 0$ to n do

$T[i,0] \leftarrow 0$

For $i1$ to n do

For $c \leftarrow 1$ to w do

If ($w_i <= c$ then)

$T[i,c] \leftarrow max(T[i-1,c],$

$p_i + T[i-1,c-w_i]$

Else

$T[i,c] \leftarrow T[(i-1]),c]$

Return $T[n,w]$

Runtime of Algorithm :

$O(nw)$

Application : DNA Sequence Alignment :

a. Longest Common Subsequence (LCS)

Objective : Give 2 DNA sequences, find the longest common subsequence between them.

This application of DNA sequence alignment takes in two Strings, and determines the longest common subsequence, where the literals don't have to be in consecutive order. In this application, bases of DNA sequences need to be compared for various biological processes to occur.

Input :

- ➢ String $x =< x_1, x_2, ..., x_n >$
- ➢ String $y =< y_1, y_2, ., y_m >$

Output :

- ➢ Longest Common Subsequence (LCS)
  - ○ String $z =< z_1, z_2, ..., z_n >$

Recurrence Relation :

$$L[i,j] = \begin{cases} 0 & \text{if } i = 0 or j = 0 \\ L[i-1, j-1] + 1 & \text{if } x_i = y_j \\ maxL[i-1, j], L[i, j-1] & \text{otherwise.} \end{cases}$$

We can visualize the two string such that they are placed in a matrix of size $m * n$ (length of String x, and length of String y). Both the first row and first column will be occupied by 0s, whereas the other positions in the matrix will have integer values which are computed by the algorithm. $LCS[m, n]$ visual representation (matrix) :

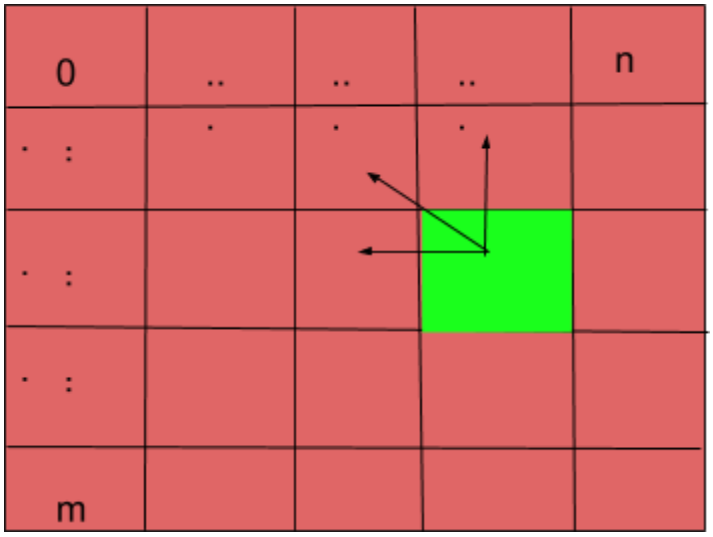


Figure 3

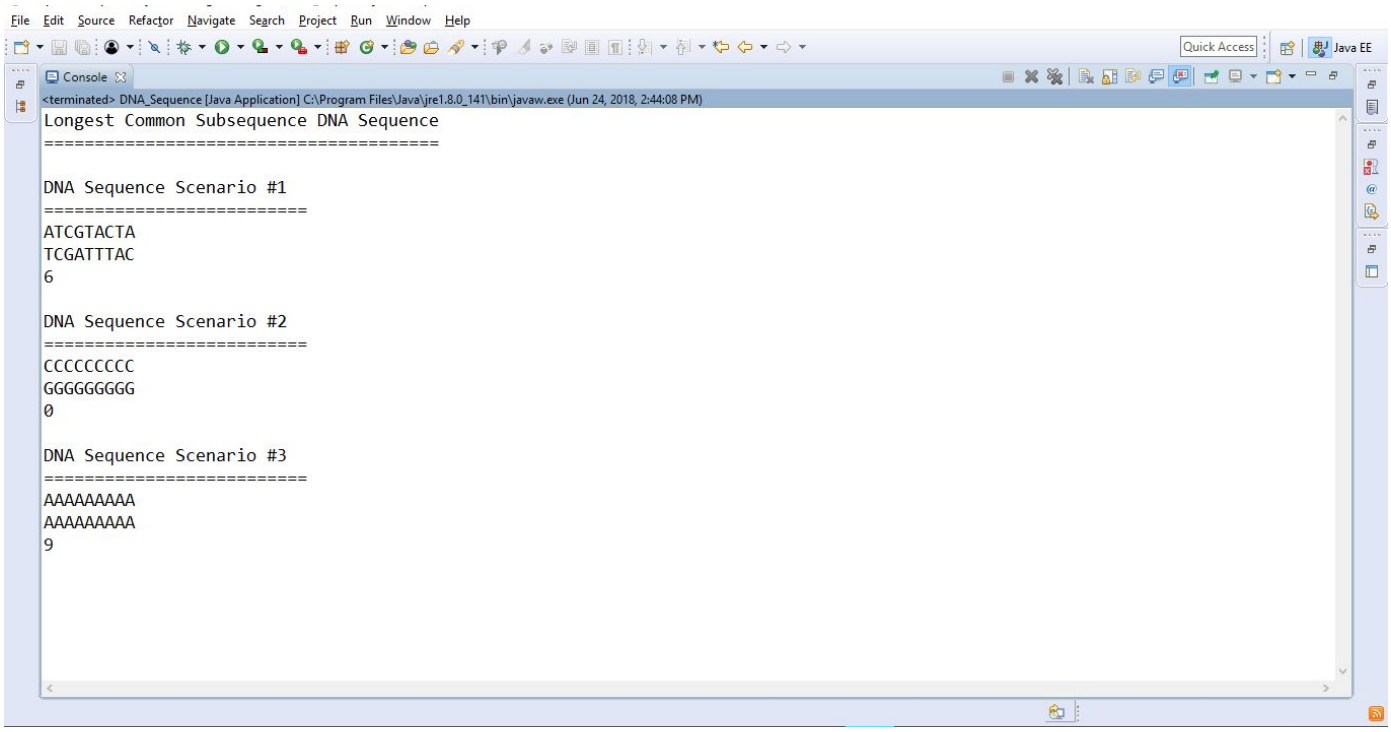
Java Code :

```
/**
 * This dynamic programming problem finds the longest common subsequence (LCS)
 * between 2 DNA Sequences.
 * @author Deepansha Singh
 *
 */
public class DNA_Sequence {

	//note that String x : <x1, x2, x3, ...xn> & String y : <y1, y2, y3, ..., ym>

	public int LongestCommonSubsequence(String x, String y, int n, int m)
	{//n and m represent the lengths of the Strings
		if (n == 0 || m == 0)    //algo traverses through Strings from last index to  first, if
@ 1st index, will return 0
			return 0;
		else if (x.substring(n-1, n).equals(y.substring(m-1, m)))       //if both characters are
=
			return LongestCommonSubsequence(x, y, n-1, m-1) + 1;
		else return Integer.max(LongestCommonSubsequence(x, y, n-1, m),
//similar to pseducode and matrix diagram, find max
```

```
                                                    LongestCommonSubsequence(x, y,
n, m-1));
        }

        public static void main(String[] DEEPANSHA)
        {
                //3 DNA_Sequence Scenarios
                DNA_Sequence one = new DNA_Sequence();          //creating object
                String sequence1 = "ATCGTACTA";           //2 sequences
                String sequence2 = "TCGATTTAC";
                System.out.println("Longest Common Subsequence DNA Sequence");      //title
                System.out.println("=======================================");


                System.out.println("\nDNA Sequence Scenario #1");          //scenario #1
                System.out.println("==========================");
                System.out.println(sequence1 + "\n" + sequence2);
                System.out.println(one.LongestCommonSubsequence(sequence1, sequence2,

        sequence1.length(), sequence2.length()));

                System.out.println("\nDNA Sequence Scenario #2");          //scenario #2
                sequence1 = "CCCCCCCCC";
                sequence2 = "GGGGGGGGG";
                System.out.println("==========================");
                System.out.println(sequence1 + "\n" + sequence2);
                System.out.println(one.LongestCommonSubsequence(sequence1, sequence2,

        sequence1.length(), sequence2.length()));

                System.out.println("\nDNA Sequence Scenario #3");          //scenario #3
                sequence1 = "AAAAAAAAA";
                sequence2 = "AAAAAAAAA";
                System.out.println("==========================");
                System.out.println(sequence1 + "\n" + sequence2);
                System.out.println(one.LongestCommonSubsequence(sequence1, sequence2,

        sequence1.length(), sequence2.length()));
```

```
        }
}
```

Output :



Please note I used Eclipse Oxygen IDE for this Dynamic Programming application problem.

Conclusion :

Dynamic programming is an optimal programming method which breaks the problem into subproblems, and implements a recurrence relation in the algorithm to produce an efficient solution which takes polynomial time, unlike other algorithmic techniques.