Fundamentals of Algorithm Design and Machine Learning

Name : Chimala Rohith Chandra
Roll no : 24BM6JP15
Professor Name : Aritra Hazra
Date : 23-01-2025
Lecture slot : 08:00 hrs to 08:55 hrs

==========

- **Dynamic Programming:**

Dynamic programming, like the divide-and-conquer paradigm, solves problems by combining the solutions to subproblems. As we saw in previous classes, divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming applies when the subproblems overlap or are repetitive, that is, when subproblems share sub-subproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub subproblems. A dynamic-programming algorithm solves each sub sub problem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each sub subproblem.

The following are some problems in which dynamic programming is used to solve it in an optimal way.

1. **Matrix chain multiplication:**

Problem:
Given the matrix multiplication chain of 'j' matrices, $\mathbf{M_1M_2M_3.....M_j}$. Find out the most efficient way to group the matrices that minimizes the total number of scalar multiplications.

Solution:
For a matrix multiplication chain of n matrices: $\mathbf{M_1M_2M_3.....M_j}$. Where a matrix Mp is of dimensions $m_{p-1} \times m_p$ for p = 1, 2, 3, ... n. Let C[i,j] denote the computation cost of evaluating the chain $M_iM_{i+1}M_{i+2}$ ..... M_j most efficiently.

$$C[i,j] = \begin{cases} 0 & \text{if } i > j \quad \text{(base)} \\ 0 & \text{if } i = j \quad \text{(base)} \\ \min\limits_{i \le k < j} \{ (C[i,k] + C[k+1,j] + n_{i-1} \times n_k \times n_j) \} & \text{if } i < j \end{cases}$$

To demonstrate how this recursion works, let us consider the example of a matrix multiplication chain of four matrices $\mathbf{M_1M_2M_3M_4}$. Our objective is to evaluate C[1,4].
Consider,

- $C[1,1] = C[2,2] = C[3,3] = C[4,4] = 0.$
- $C[1,2] = C[1,1] + C[2,2] + n_0 \times n_1 \times n_2 = n_0 \times n_1 \times n_2$
- $C[2,3] = C[2,2] + C[3,3] + n_1 \times n_2 \times n_3 = n_1 \times n_2 \times n_3$
- $C[3,4] = C[3,3] + C[4,4] + n_2 \times n_3 \times n_4 = n_2 \times n_3 \times n_4$
- $C[1,3] = \min \begin{cases} C[1,1] + C[2,3] + n_0 \times n_1 \times n_3 \\ C[1,3] + C[3,3] + n_0 \times n_2 \times n_3 \end{cases} = \min \begin{cases} n_1 \times n_2 \times n_3 + n_0 \times n_1 \times n_3 \\ n_0 \times n_1 \times n_2 + n_0 \times n_2 \times n_3 \end{cases}$
- $C[2,4] = \min \begin{cases} C[2,2] + C[3,4] + n_1 \times n_2 \times n_4 \\ C[2,3] + C[4,4] + n_1 \times n_3 \times n_4 \end{cases} = \min \begin{cases} n_2 \times n_3 \times n_4 + n_1 \times n_2 \times n_4 \\ n_1 \times n_2 \times n_3 + n_1 \times n_3 \times n_4 \end{cases}$
- $C[1,4] = min \begin{cases} C[1,1] + C[2,4] + n_0 \times n_1 \times n_4 \\ C[1,2] + C[3,4] + n_0 \times n_2 \times n_4 \\ C[1,3] + C[4,4] + n_0 \times n_3 \times n_4 \end{cases}$

$$= min \begin{cases} C[2,4] + n_0 \times n_1 \times n_4 \\ n_0 \times n_1 \times n_2 + n_2 \times n_3 \times n_4 + n_0 \times n_2 \times n_4 \\ C[1,3] + n_0 \times n_3 \times n_4 \end{cases}$$

The memoization table for this is as follows:

		i →			
j ↓		**1**	**2**	**3**	**4**
	1	0			
	2	C[1,2]	0		
	3	C[1,3]	C[2,3]	0	
	4	C[1,4]	C[2,4]	C[3,4]	0

Notice that for computing C[1,3] we need values of only C[1,1], C[2,3], C[1,3] and C[3,3]. Similarly, for computing C[2,4], we need values of only C[2,2], C[3,4], C[2,3] and C[4,4], and so on and so forth. We can generalise from this observation that for computing any C[i,j] we need values from the j^{th} row and i^{th} column. Hence, we can fill the table iteratively, starting from the diagonal and approaching the left-down corner.

Time complexity:
Since there are $\mathbf{O(n^2)}$ values to be filled i.e subproblems, and for each subproblem, **O(n)** possible values of k. Hence, the time complexity of the algorithm is $\mathbf{O(n^3)}$.

Space complexity:
And total space complexity is $\mathbf{O(n^2)}$.

2. Longest Common Subsequence (LCS) Problem:

Problem Statement:

Given two sequences, X of length m and Y of length n, find the longest subsequence common to both X and Y. Find the maximum length of a common subsequence between the two sequences.

Solution:

We need to find the LCS of $L_1 = < L_1[1], L_1[2], L_1[3],....L_1[m] >$ and $L_2 = < L_2[1], L_2[2], L_2[3],....L_2[n] >$. A naïve approach as an initial solution would be to search for every element of L_1 iteratively in L_2. This would yield us time complexity of **$O(n^2)$**.

Refined solution:

Let L[i, j] be the length of LCS of $L_1[i]$ and $L_2[j]$ where i = 0,1,2,3,....m and j = 0,1,2,3,....n. The base case is if either of i and j is 0, i.e. one of the sequences has length 0, then LCS would also be 0. Therefore, we have,

$$L[i,j] = \begin{cases} 0 \text{ if } i = 0 \text{ or } j = 0 \\ 1 + L[i-1, j-1] \text{ if } i,j > 0 \text{ and } L_1[i] = L_2[j] \\ max \begin{cases} L[i-1,j] \\ L[i,j-1] \end{cases} \text{when } L_1[i] \neq L_2[j] \end{cases}$$

The problem can be either solved recursively where each L[i, j] iteratively calls L[i, j-1] and L[i-1, j] until a base case is reached, then the solutions to subproblems are recombined from bottom to top and the results are stored in the memoization table.

Consider the example where X = <Y, E, L, L, O, W> and Y = <H, E, L, L, O>. As we can see that LCS is <E, L, L, O>. Consider the memoization table for L[i, j] . We aim to evaluate L[6,5] and find out the LCS. We can do this iteratively by starting with L[0,0] = L[0,:] = L[:,0] = 0. Since x1 ≠ y1, L[1,1] would be the maximum of L[0,1] and L[1,0] which is 0. Similarly, we have L[:, 1] = L[1,:] = 0 since 'H' from sequence Y is not in sequence X and 'Y' from sequence X is not in sequence Y. L[2,1] = 0 since <Y, E> doesn't have any common sequence with <H>. Similarly, we can fill the rest of the table either row-wise or column-wise.

As we can see from the table, L[6,5] = 4; hence, the length of LCS of X = <Y, E, L, L,O, W> and Y = <H, E, L, L, O> is 4.

To find LCS, we need to find the indices i and j where the increment occurred. This can be done by moving backwards from L[6,5] towards L[0,0] by following the direction where the increment occurred (as shown by the red arrow). Hence, we can see that the LCS is <E, L, L, O>.

Time & space complexity : $O(n^2)$

				Y					
	i		j →						
				0	1	2	3	4	5
					H	E	L	L	O
X	0		0	0	0	0	0	0	
	1	Y	0	0	0	0	0	0	
	2	E	0	0	1	1	1	1	
	3	L	0	0	1	2	2	2	
	4	L	0	0	1	2	3	3	
	5	O	0	0	1	2	3	4	
	6	W	0	0	1	2	3	4	

3. Edit distances problem:

Problem Statement:

Given two strings 'L_1' and 'L_2'. L_1 is to be converted to L_2 using the following 3 operations:
Deletion of a character.
Substitution of a character with another one.
Insertion of a character.
Each operation has a cost corresponding to it and the aim is to carry out the conversion of string in minimum possible cost.
Taking an example of conversion of string 'HELLO'(source) to 'YELLOW'(target). Our aim is to find the minimum cost path for the same. Let the cost for each operation be as following:
Insertion: +1
Deletion: +1
Substitution: +2
Table is initialized with dimensions (n+1) * (m+1), where n is the length of the source and m is the length of the target. First row represents conversion of an empty string to target string and the first column represents conversion of source string to an empty target string. First column and first row will be updated as following:
For the first row, since we are converting an empty string to target string, each character will be inserted, hence cost will be the cost of insertion of each character.
For first column since we are converting source to an empty string, each character will have to be deleted, hence the cost will be the deletion cost of each character

Solution:

If $L_1[i]$ matches $L_2[j]$, no transformation is to be performed and cost is carried over from the previous diagonal cell.
No operation (Nop) : **D[i,j] = D[i-1,j-1]**, if **$L_1[i] = L_2[j]$**
If $L_1[i]$ is different from $L_1[j]$, there are three possible options and minimum cost is computed :
if $L_1[i] \neq L_2[j]$
Deletion from the source: **D[i-1, j] + 1**
Insertion into the target: **D[i, j-1] + 1**
Substitution: **D[i-1, j-1] + 2**

	0	Y	E	L	L	O	W
0	0	1	2	3	4	5	6
H	1	2	3	4	5	6	7
E	2	3	2	3	4	5	6
L	3	4	3	2	3	4	5
L	4	5	4	3	2	3	4
O	5	6	5	4	3	2	3

From this we need to find the minimum cost to reach **D[i,j]**
Final solution is given by the bottom-right cell. Which in this case depicts that the minimum edit distance between HELLO and YELLOW, which is D[5,6] = 3

Time complexity : O(n*m)
Space complexity : O(n*m)

4. 0-1 knapsack problem:

Problem Statement:

How to take the most valuable items that can be carried in a knapsack capable of carrying at most C items. One can choose to take any subset of n items in the store. The i^{th} item is of volume v_i and wealth value w_i, where v_i and w_i are integers. How many items should be taken? Constraint here is that we can either put an item set completely in the bag or cannot put it at all, that is, the volume and value from i^{th} element will either increase by v_i and w_i respectively or none at all.

Solution:

A 2D table can be created in which all values can be stored progressively. Time complexity for which can be obtained recursively as $O(n*W)$where n is the number of items in knapsack and W is its capacity. As there are only n*W values to be calculated.

Let **K[I, n]** denote the maximum value of the items taken from a set of items of value denoted by i, by a knapsack of capacity n.

K[I, n] is computed based on the selection of i th item of value w_i

If the item is taken then **K[I, n] = w_i + K[I-{w_i}, n-v_i]**

If the item is not taken **K[I, n] = 0 + K[I-{w_i}, n-v_i]**

And these values are stored in a table. And then select the maximum value from the table

Taking an example where: Wealth value, w = {1, 4, 5, 7},volume , v = {1, 3, 4, 5},The knapsack capacity is 7

Build a table where each entry K[i, n] will represent the maximum value that can be obtained using the first i items for a knapsack of capacity c. We start by creating a table with (n+1) rows and (c+1) rows.

Initially, the first row is filled with zero as no item gives zero value. Then, iterate through the items and the capacities, filling the table with maximum possible values.

Results can be obtained from the bottom right corner of the table where the value is 9 and knapsack is filled at its capacity 7.

Items \ Capacity	0	1	2	3	4	5	6	7
0 item	0	0	0	0	0	0	0	0
1 item (V=1, W=1)	0	1	1	1	1	1	1	1
2 item (V=3, W=4)	0	1	1	4	5	5	5	5
3 item (V=4, W=5)	0	1	1	4	5	6	6	9
4 item (V=5, W=7)	0	1	1	4	5	7	8	9

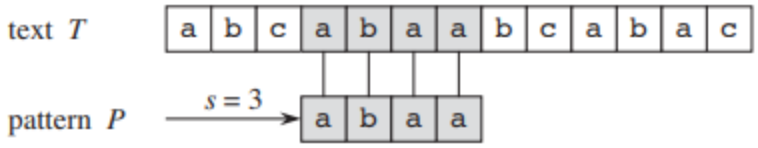
Time complexity : O(n*c)

Space complexity : O(n*c)

5. String matching :

We assume that the text is an array **T [1….n]** of length n and that the pattern is an array **P[1..m]** of length m(where **m < n**).

Let say that pattern **P** occurs with shift **s** in text **T** (pattern **P** occurs beginning at position **s + 1** in text T) if **0 ≤ s ≤ n-m** and **T [s+1…s+ m]= P [1…m]** (that is, if **T[s+ j]= P[j]** , for **1 ≤ j ≤ m**). If P occurs with shift s in T , then we call s a valid shift; otherwise, we call s an invalid shift. The string-matching problem is the problem of finding all valid shifts with which a given pattern P occurs in a given text T .



We can also find the maximum overlap of the pattern with respect to the pattern starting point present in the text

Among their many other applications, string-matching algorithms search for particular patterns in DNA sequences. Internet search engines also use them to find Web pages relevant to queries.

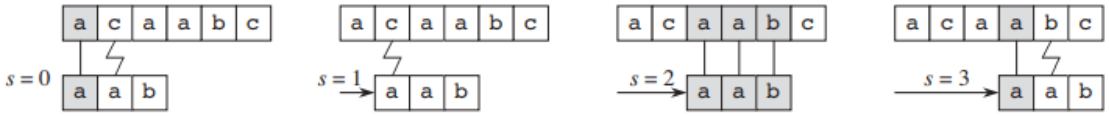
Initial solution:

The naive algorithm finds all valid shifts using a loop that checks the condition P [1…m] = T[s+1…s+m] for each of the n -m+1 possible values of s.

Algorithm:

```
String_match1(T,P){
n = |T|
m = |P|
s=0
for s = 0 to n-m
    If P [1…m]==T[s+1…s+m]
          Print "pattern occurs with shift :" s
}
```

The program slides through the text T to find a pattern P to check each shift is valid or invalid. Here in the below example it is found after 2 shifts.



The naive string-matcher is inefficient because it entirely ignores information gained about the text for one value of s when it considers other values of s. Such information can be quite valuable, however. For example, if P = aaab and we find that s = 0 is valid, then none of the shifts 1, 2, or 3 are valid, since T [4] = b. In the following sections, we examine several ways to make effective use of this sort of information

Time complexity : O(nm)

Solution Refinement: (The Knuth-Morris-Pratt algorithm)

Before taking a leap into the solution lets take a step back to understand a concept - 'border of a string', which helps us to jump to an optimal position.

A **border of a string** is a prefix that is also a suffix of the string but not the whole string. For example, the borders of abcababcab are ab and abcab. Border of abacabab is ab say the border is 2.

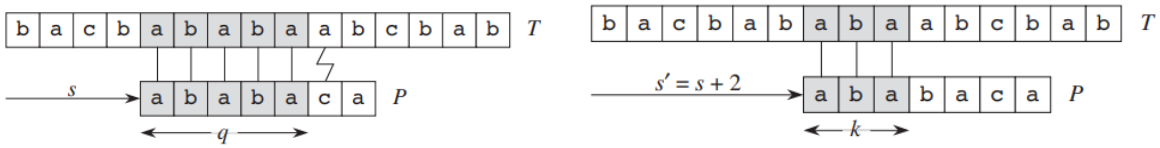
We can use this concept to conveniently shift to the suffix positions of the pattern, when the pattern doesn't match with the text at a particular position during the valid shift checking.

The border prefix function b for a pattern incorporates knowledge about how the pattern matches against shifts of itself. We can take advantage of this information to avoid testing useless shifts in the initial solution of the pattern-matching algorithm.

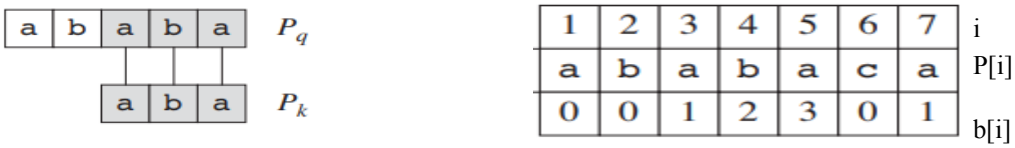
The below figure shows a particular shift s of a template containing the pattern P = ababaca against a text T . For this example, q = 5 of the characters have matched successfully, but the 6th pattern character fails to match the corresponding text character. The information that q characters have matched successfully determines the corresponding text characters. Knowing these q text characters allows us to determine immediately that certain shifts are invalid. In the example of the figure, the shift s + 1 is necessarily invalid, since the first pattern character

(a) would be aligned with a text character that we know does not match the first pattern character, but does match the second pattern character

(b). The shift s' = s + 2 shown in the figure, however, aligns the first three pattern characters with three text characters that must necessarily match.



In general, it is useful to know the answer to the question. Given that pattern characters P [1.. q] match text characters T [s+...s+q], what is the least shift s' > s. such that for some k<q



We can precompute the necessary information by comparing the pattern against itself, as per the above figure demonstrates

Algorithm:

```
border_prefix(P){        //P is the pattern text in array
m = |P|
Let b[1…m] be an array
b[1]=0
j=0
for i = 2 to m{
      while P[j+1] != P[i] and j>0
              j=b[j]
      If b[j+1]== b[i]
              j=j+1
      b[q] = j
}

String_match2(T,P){
n = |T|
m = |P|
b = border_prefix(P)
q=0                                          // number of characters matched
for  k = 1 to n   {                          // scan the text from left to right
    while q>0 and P [q +1] !=  T [k]
            q = b[q]                         // next character does not match
    if P[q+1] == T[i]
            q=q+1                            // next character matches
    if q == m  {                             // check if all of P matched?
       print "Pattern occurs with shift :" k-m
       q = b[q]  }}                          // look for the next match
}
```

Time complexity : O(n+m)
Space complexity : O(m)

==========

References:

- Thomas H. Cormen, Charles E. Lieserson, Ronald L. Rivest and Clifford Stein, Introduction to Algorithms
- Scribe notes of 23BM6JP13_2025-01-21
- Scribe notes of 23BM6JP14_2025-01-21

==========