FADML Scribe

Dabbikar Likhith

20-24 Jan 2025

The week (20-24 jan 2025) of the lectures discussed on the foundational design algorithms and problem solving paradigms. Several algorithms were discussed such as Divide and Conquer, Dynamic Programming and Greedy Approach. Practical implementations such as Median finding, Closest pair problem, Matrix chain multiplication and Longest common subsequence (LCS) were explained with step by step breakdown and complexity analysis.

Algorithm design principles

Step 1: Initial solution
1. Recursive formulation
2. Correctness
3. Complexity analysis

Step 2: Exploration of structure
1. Decomposition
2. Analysis recursive structure
3. Recomposition

Step 3: Solution refinement
1. Balance/split
2. Analysis of recurrence
3. Identical subproblem

Step 4: Data structuring and complexity
1. Revise memory/information
2. Analysis of space complexity

Step 5: Final solution
1. Traversal of recursive structure
2. Complexity
3. Pruning/ backtracking
4. Implementation

1. Median Finding Algorithm:

1. **Def:** The median of a sequence is the middle element of the n- sorted numbers. If n is odd the median is the $(n+1)/2^{th}$ element. If n is even, the median is the $n/2^{th}$ smallest element.
2. Naive approach : sorting → selection : Sort the elements and then select the middle element. But the sorting algorithm takes **Ω**(nlogn) time which is suboptimal for large datasets.
3. Optimal median finding algorithm is the selection algorithm: (median of medians) The problem of finding the kth element, including the median , can be solved optimally in O(n) time using the median of medians algorithm.

Algorithm Median of medians:

This is a divide and conquer approach that efficiently finds the median or kth smallest element in O(n) time.

Steps:

Base case: If the number of elements n< 44, sort the array directly and return the kth smallest element
divide : partition the array into [n/5] groups of 5 elements each.
Sort groups: Sort each group individually and extract the median of each group.
Find the MM (median of medians): recursively find the median of medians from the set of medians.
Partition the array: A1: elements less than mm. A2: elements equal to mm, A3: elements greater than mm.
Select in the correct subarray: If |A1| >= k, recurse on A1.
If |A1| + |A2| >= k, return mm
Else, recurse on A3 with the updated k = k-|A1|-|A2|

Algorithm: select
Input: An array A[1..n] of n elements and an integer k, 1 ≤ k ≤ n.
Output: The kth smallest element in A.
1. select(A, k)
Procedure select(A, k)
1. n← |A|
2. if n < 44 then sort A and return (A[k])
3. Let q= ⌊n/5⌋. Divide A into q groups of 5 elements each. If 5 does not divide p, then discard the remaining elements.
4. Sort each of the q groups individually and extract its median. Let the set of medians be M.
5. mm← select(M, ⌈q/2⌉) {mm is the median of medians}
6. Partition A into three arrays:
A1 = {a | a < mm}
A2 = {a | a = mm}
A3 = {a | a > mm}
7. case
|A1| ≥ k: return select(A1, k)
|A1| + |A2| ≥ k: return mm
|A1| + |A2| < k: return select(A3, k − |A1| − |A2|)
8. end case

4. **Analysis of the selection algorithm:**
    - Correctness:The algorithm always finds the kth smallest element by the recursive selection.
    - Time complexity: Steps 1-4 solves or takes O(n) (Partitioning, sorting. finding median of medians takes O9n). recursive step solves utmost T(0.7n) elements in each step.
    - The recursion relation: T(n) <= T(n/5) + T(0.75n) + O(n)
    - Solves to O(n) using the recurrence theorem.

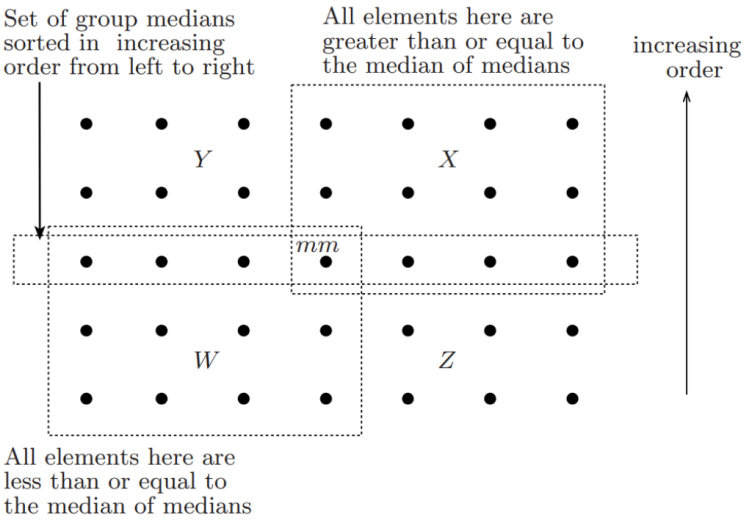


Figure 1: Nature of elements w.r.t Median of Medians .

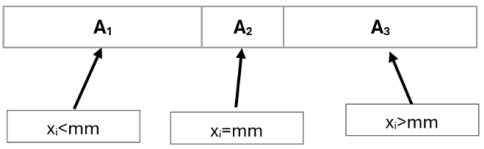


Figure 2: Splitting of array into three parts based on mm

5. Key takeaways:
    - sorting based approach takes **Ω**(nlogn).
    - mm algorithm→ finds the median in O(n), making it optimal.
    - It ensures worst case linear time, unlike quickselect which has an $O(n^2)$ worst case.
    - The threshold (44 elements) is used to keep the constant factors manageable.
6. Alternative approach:
    - A randomized algorithm can also achieve O(n) expected time.
    - Instead of selecting a deterministic median of medians, it randomly selects a pivot.
    - This is similar to QuickSelect, but lacks worst-case guarantees.
7. Conclusion :
    - Naïve Sorting + Selection → Ω(n log n)
    - Median of Medians Algorithm → O(n) worst-case
    - Randomized Selection (QuickSelect) → O(n) expected, $O(n^2)$ worst-case
    - Optimal approach: Median of Medians ensures O(n) worst-case performance.

Why groups of 5?

When dividing the input into groups, choosing an odd-sized group (like 5) proves to be more efficient than an even-sized group.

Choosing groups of 5 in the Median of Medians algorithm ensures efficient trimming and sorting. Even-sized groups require extra computations to determine the median, leading to inefficiencies. For instance, when n = 3, sorting n/3 elements and keeping 2n/3 for recursion results in no effective trimming. The sorting cost, given by O(n log k), increases as k grows, making larger groups less efficient. Empirical data shows that for n = 1000, the sorting cost is 2321.92 for k = 5 but rises significantly for larger values. Thus, choosing 5 as the group size ensures optimal balance between sorting efficiency and element elimination, achieving an O(n) worst-case time complexity.

The amount of data trimmed every-time is atleast
3 x [[n/5]/2]= (3/2) x [n/5] ≤ 3/2 x [(n-4)/5]

Length of the new data
n-3/2 x [(n-4)/5] = 0.7n + 1.2

T(n) = O(n) + T(n/5) + T(0.7n +1.2), Here c1·n = n/5 , c2·n = 0.7n + 1.2
= cn(1/(1-⅕-¾) = 20cn

The recurrence relation is given by
T(n) = T(c1n)+T(c2n) + bn
≤ Kc1n+Kc2n + bn ≤ Kn

The solution to this recurrence relation is
K ≥ b/(1-c1-c2) (when c1 = .75 and c2 = .2 then → k ≥ 20b

2. Closest pair algorithm:

Def: Given a set of n points in a plane, the goal is to find the pair of points (p1, p2) that have the smallest euclidean distance, defined as

$$d(p1,p2) = \sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$$

Naive approach computes the distances between all the pairs, taking $O(n^2)$ time. Instead divide and conquer approach can solve problem efficiently in Θ(nlogn) time which is optimal solution.

Divide and conquer approach:

1. Sort the points by the increasing x coordinates
2. Divide the set into two equal halves S_1 (left) and S_2 (right) using the vertical dividing line L
3. Recursively compute the smallest distances δ□ in S□ and δ_r in S_r.
4. Merge Step (Key Optimization): Compute δ', the smallest separation between a point in S□ and a point in S_r.
5. Final Result: The overall closest pair distance $\delta = \min(\delta$□, δ_r, δ').

Efficient merge step:

1. A naive $O(n^2)$ pairwise comparison would be inefficient. Instead, only points within distance δ from L need to be checked.

2. These points lie within a δ × 2δ strip, forming at most 8 critical points (4 from each side).
3. Sorting these points by y-coordinate allows checking only the 7 closest neighbors, reducing comparisons significantly.

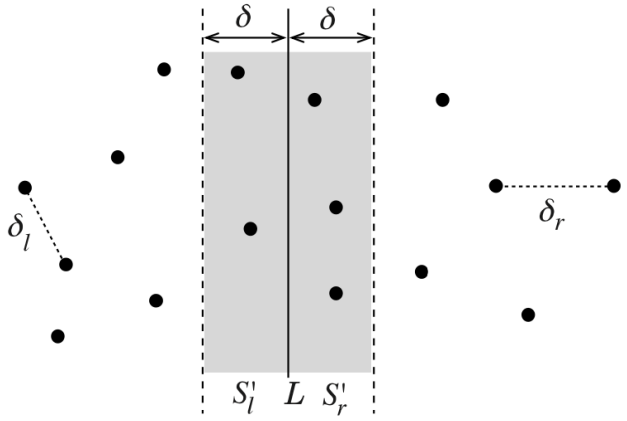
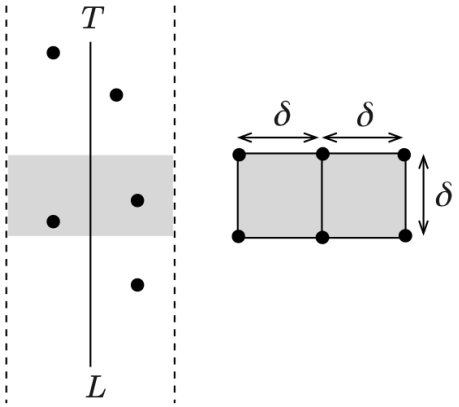


Illustration of combine step



Further Illustration of the combine step

Time Complexity:

$$T(n) = \begin{cases} c, & \text{if } n \leq 3, \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \Theta(n), & \text{if } n > 3. \end{cases}$$

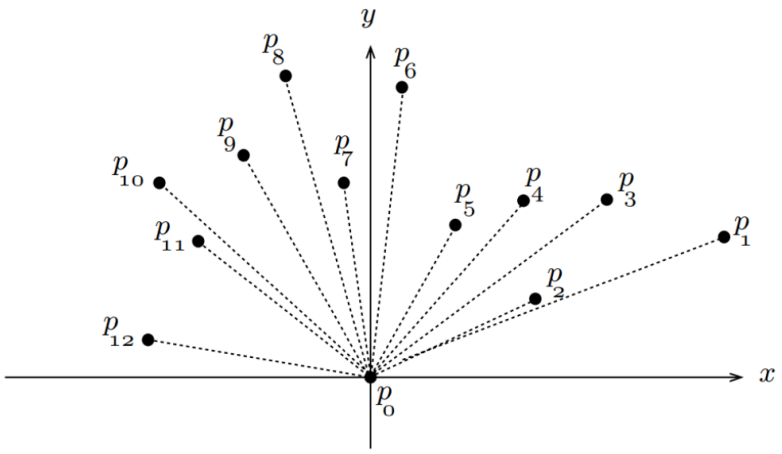
Θ(n) is attained by the mergesort algorithm, where sorting is every recursion is replaced by merging.

Pseudo code:

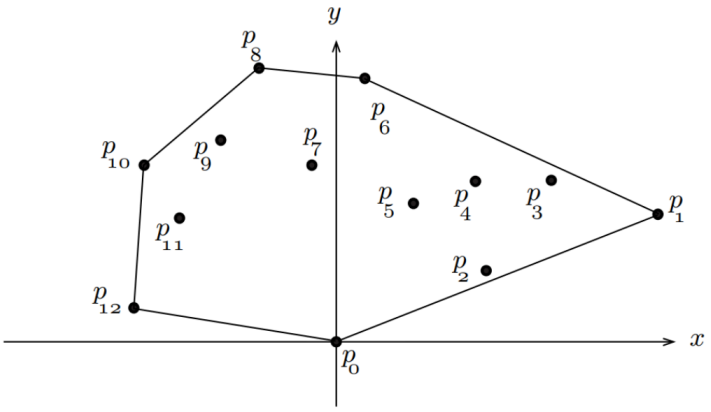
Input: A set S of n points in the plane.
Output: The minimum separation realized by two points in S.
1. Sort the points in S in nondecreasing order of their x-coordinates.
2. (δ, Y)← cp(1, n)
3. return δ

Procedure cp(low , high)
1. if high− low + 1 ≤ 3 then
2. compute δ by a straightforward method.
3. Let Y contain the points in nondecreasing order of y-coordinates.
4. else
5. mid ← ⌊(low + high)/2⌋
6. x0 ← x(S[mid])
7. (δl , Yl)← cp(low, mid)
8. (δr , Yr)← cp(mid + 1, high)
9. δ← min{δl, δr }
10. Y ← Merge Yl with Yr in nondecreasing order of y-coordinates.
11. k← 0
12. for i← 1 to |Y | {Extract T from Y }
13. if |x(Y [i])− x0| ≤ δ then
14. k← k + 1
15. T[k]← Y [i]
16. end if
17. end for {k is the size of T}
18. δ′
← 2δ {Initialize δ′ to any number greater than δ}
19. for i← 1 to k− 1 {Compute δ′ }
20. for j ← i + 1 to min{i + 7, k}
21. if d(T[i], T[j]) < δ′ then δ′
← d(T[i], T[j])
22. end for
23. end for
24. δ← min{δ, δ'}
25. end if
26. return (δ,Y)

3. Convex Hull Algorithm:



A set of point S for which the convex hull is calculated



Final convex hull

The Graham scan algorithm is given by

1. Select the y - coordinate with the lowest y- coordinate. Call it p_0.
2. Sort the points by the angle relative to the bottom most point and the horizontal.
3. The coordinates are transformed such that p_0.
4. Iterate in the sorted order, by placing each point on a stack but only if it makes a counter clockwise turn relative to the previous 2 points on the stack.
5. Pop the previous point off the stack if it is making a clockwise turn.

Time Complexity:

The Graham Scan algorithm for finding the Convex Hull of n points sorts the points by their polar angle relative to the lowest point, which takes O(n log n) time. The main loop then processes each point once, pushing and popping from the stack at most O(n) times, leading to an O(n) operation. Since sorting dominates the complexity, the overall time complexity of the algorithm is O(n log n). This makes it efficient for computing the convex hull compared to brute-force approaches like Jarvis March ($O(n^2)$).

Pseudo code:

1 Input: A set S of n points in the plane.
2 Output: CH(S), the convex hull of S stored in a stack St.
1. Let p_0 be the rightmost point with minimum y-coordinate.
2. T[0] ←p_0
3. Let T[1..n−1] be the points in S\{p_0} sorted in increasing polar angle about p_0. If two points p_i and p_j form the same angle with p_0, then the one that is closer to p0 precedes the other in the ordering.
4. push(S_t, T[n−1]); push(S_t, T[0])
5. k←1
6. while k<n−1
7. Let S_t= (T[n−1],...,T[i],T[j]), where T[j] is on top of the stack.
8. if T[i],T[j],T[k] is a left turn then
9. push(S_t, T[k])
10. k←k+ 1
11. else pop(St)
12. end i
13. end while

4. Reductions:

1. Purpose of reduction: Used to establish lower bounds for algorithmic problems by comparing their complexity.
2. Transformation between problems: A problem A with a known lower bound Ω(f(n)) is transformed into another problem B to prove that B also has at least Ω(f(n)) complexity.
3. Reduction process:
    - Step 1: convert an input of A into a valid input for B.
    - Step 2: Solve B.
    - Step 3: Convert the output of B into a valid solution for A.
4. Linear time reduction:
    - A linear reduction is written as $A \propto \square B$, meaning A is reducible to B in O(n) time
5. Significance:
    - Helps prove that if B could be solved faster than Ω(f(n)), then A would also be solved faster, contradicting its known lower bound.

Dynamic Programming:

Def: Dynamic Programming (DP) is an algorithm design technique used for solving combinatorial optimization problems by breaking them into overlapping subproblems and solving them bottom-up while storing intermediate results. Unlike Divide and Conquer, DP avoids redundant recursive calls by memoization (top-down) or tabulation (bottom-up). It is particularly useful for improving the efficiency of brute-force solutions to NP-hard problems, such as solving the Traveling Salesman Problem (TSP) in $O(n^2 2^n)$ time, compared to Θ(n!) with naive enumeration.

5. Matrix Chain multiplication:

Given a sequence of matrices, determine the most efficient way to parenthesize them to minimize the number of scalar multiplications.
Our goal is to find the optimal multiplication order while keeping the matrix sequence unchanged.

Computation Cost:

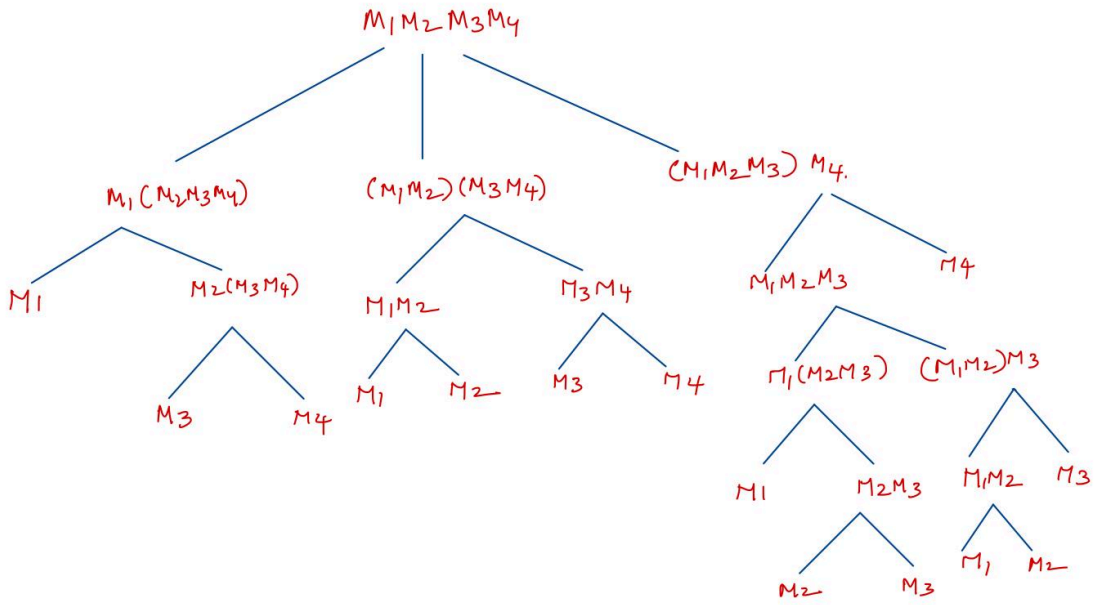
For a matrix $M_1 \times M_2 \times M_3 \times M_4$, different parenthesizations yield different computation costs:

Parenthesization	Computation Cost
$M_1(M_2(M_3M_4))$	8750
$M_1((M_2M_3)M_4)$	55000
$(M_1M_2)(M_3M_4)$	6500

$(M_1(M_2M_3))M_4$	77500
$((M_1M_2)M_3)M_4$	10000

Using brute-force, the number of ways to parenthesize j matrices grows as $\Omega(2^j)$, making exhaustive search impractical. Instead, DP optimally structures the computation.

Exploration of recurrence structure: The recurrence structure for the matrix multiplication chain M_1, M_2, M_3, M_4



Recurrence Structure:

For a matrix multiplication chain of n matrices: $M1M2M3\ldots\ldots Mn$. Where a matrix Mp is of dimensions $mp-1 \times mp$ for $p=1,2,3,\ldots\ldots n$. Let $C[i,j]$ denote the computation cost of evaluating the chain $MiMi+1Mi+2\ldots\ldots Mj$ most efficiently.
base conditions:

- When $i > j$ we have $C[i,j] = 0$ since the matrix production is not commutative.
- When $i = j$, we have $C[i,j]=0$, since there is only one matrix in the chain
- When $i<j$, we have $C[i,j]= \min i \le k < j\ (C[i,k]+C[k+1,j] + ni-1 \times nk \times nj)$

Therefore we have the following recurrence condition

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

Memoization in matrix chain multiplication can be applied using a recursive (top-down) approach, where $C[i,j]$ calls $C[i,k]$and $C[k+1, j]$, partitioning the chain at k until the base case i = j is reached. The results are stored in a memoization table to prevent redundant computations. However, recursion can lead to high memory usage due to deep recursion stacks. To mitigate this, a bottom-up (iterative) approach is preferred, where subproblems are solved progressively without recursion. This avoids excessive function calls while ensuring optimal computation. For example, to compute *C[1,4]* in a four-matrix chain, the process follows structured evaluations based on previously computed subproblems.

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

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

Example of memoization table is as follows:

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

The algorithm processes $O(n^2)$ subproblems, each requiring O(n) computations, leading to a total time complexity of $O(n^3)$, which is significantly better than $O(2^n)$. The space complexity remains $O(n^2)$ due to storing intermediate results. This ensures an efficient solution for matrix chain multiplication.

Pseudo code for matrix multiplication:

Algorithm 6.2 MATCHAIN

Input: An array $r[1..n+1]$ of positive integers corresponding to the dimensions of a chain of n matrices, where $r[1..n]$ are the number of rows in the n matrices and $r[n+1]$ is the number of columns in M_n.

Output: The least number of scalar multiplications required to multiply the n matrices

```
1.  for i ← 1 to n        {Fill in diagonal d_0}
2.      C[i, i] ← 0
3.  end for
4.  for d ← 1 to n − 1        {Fill in diagonals d_1 to d_{n−1}}
5.      for i ← 1 to n − d        {Fill in entries in diagonal d_i}
6.          j ← i + d
7.          comment: The next three lines compute C[i, j]
8.          C[i, j] ← ∞
9.          for k ← i + 1 to j
10.             C[i, j] ← min{C[i, j], C[i, k − 1] + C[k, j] + r[i]r[k]r[j + 1]}
11.         end for
12.     end for
13. end for
14. return C[1, n]
```

6. Longest Common subsequence problem (LCS)

Def: The Longest Common Subsequence (LCS) problem is a combinatorial optimization problem that seeks to find the longest sequence of characters that appear in the same relative order in two given sequences but not necessarily consecutively. Given two sequences X (of length *m*) and Y (of length *n*), LCS aims to determine the maximum-length subsequence that is common to both. The problem exhibits optimal substructure and overlapping subproblems, making dynamic programming the preferred approach for solving it efficiently in O(mn) time.

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

Our goal is to determine the max. length of a common subsequence between the two sequences using dynamic programming to minimize the computational cost.

Naive Approach:

To find the LCS of sequences X and Y, a naïve approach iterates through each element of X and searches for it in Y. This results in a brute-force method with a time complexity of O(n²). Due to redundant computations, this approach is inefficient for large inputs.

Improving the solution so that optimality is increased

If x□ = y□, then LCS length is LCS(X□₋₁, Y□₋₁) + 1; otherwise, it is the maximum of LCS(X□₋₁, Y) and LCS(X, Y□₋₁). This demonstrates overlapping subproblems, as finding LCS(X□₋₁, Y) and LCS(X, Y□₋₁) both rely on LCS(X□₋₁, Y□₋₁). Define L[i, j] as the LCS length

of prefixes $X_1...X_i$ and $Y_1...Y_j$, with the base case L[i, 0] = L[0, j] = 0 when either sequence is empty. This forms the basis for dynamic programming in solving the LCS problem efficiently.

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

If $x_i = y_j$, then $LCS(X_{i-1}, Y_{j-1}) + 1$ gives the LCS length; otherwise, it is the maximum of $LCS(X_{i-1}, Y)$ and $LCS(X, Y_{j-1})$. This shows overlapping subproblems, as both depend on $LCS(X_{i-1}, Y_{j-1})$. We define L[i, j] as the LCS length of prefixes $X_1...X_i$ and $Y_1...Y_j$, with the base case L[i, 0] = L[0, j] = 0 when one sequence is empty. The problem can be solved recursively using memoization, but recursion increases memory usage. Instead, a bottom-up iterative approach is more efficient, reducing redundant calculations and making dynamic programming the preferred method for solving LCS.

Given X = <Y, E, L, L, O, W> and Y = <H, E, L, L, O>, the LCS is <E, L, L, O>. We compute L[6,5] iteratively, starting with L[0,0] = 0 and setting the first row and column to 0 since an empty sequence has no LCS. Since $X_1 \neq Y_1$, we set L[1,1] = 0 and continue filling the table based on matches and previous values. The final value L[6,5] = 4 gives the LCS length. To extract the sequence, we trace back from L[6,5] to L[0,0], following the path where values increased, revealing LCS = <E, L, L, O>

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

Pseudo code:

Algorithm 6.1 LCS
Input: Two strings A and B of lengths n and m, respectively, over an alphabet Σ.
Output: The length of the longest common subsequence of A and B.

```
1.  for i ← 0 to n
2.      L[i, 0] ← 0
3.  end for
4.  for j ← 0 to m
5.      L[0, j] ← 0
6.  end for
7.  for i ← 1 to n
8.      for j ← 1 to m
9.          if a_i = b_j then L[i, j] ← L[i − 1, j − 1] + 1
10.         else L[i, j] ← max{L[i, j − 1], L[i − 1, j]}
11.         end if
12.     end for
13. end for
14. return L[n, m]
```

7. Edit distance problem:

Def: The Edit Distance Problem (or Levenshtein Distance) is a string similarity problem that measures the minimum number of operations required to convert one string into another. The allowed operations are:

1. Insertion (adding a character),
2. Deletion (removing a character), and
3. Substitution (replacing one character with another).

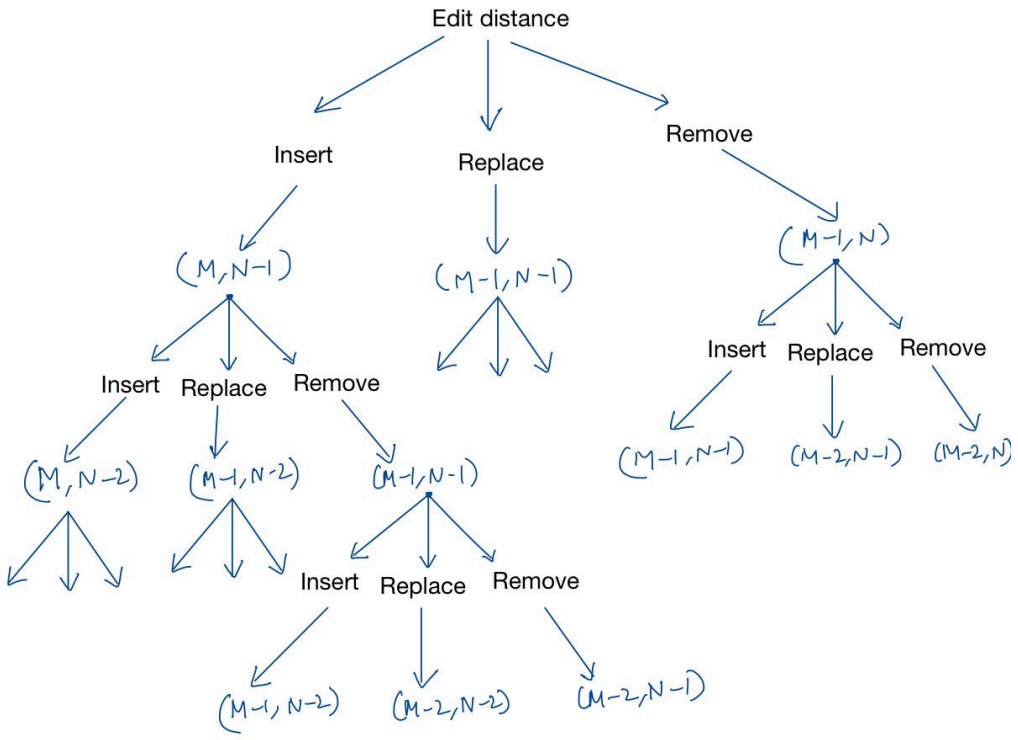
Given two strings X (length *m*) and Y (length *n*), the goal is to compute the minimum number of edits needed to transform X into Y. The problem exhibits optimal substructure and overlapping subproblems, making dynamic programming the preferred approach, with a time complexity of O(mn).

Two strings S1 (X) and S2 (Y), the goal is to transform S1 into S2 using the following three operations at minimal cost:

1. Deletion: Remove a character from S1.
2. Substitution: Replace a character in S1 with another.
3. Insertion: Add a character to S1.

Each operation has an associated cost, and the objective is to minimize the total cost required for the conversion.

Recursive Solution for Edit Distance Problem



Time complexity: This recursive solution has an exponential time complexity of $O(3^n)$ due to repeated subproblems, making it inefficient for large strings. To optimize it, memoization (top-down DP) or bottom-up DP can be used.

Pseudocode for the edit distance problem:

```
def EditDistanceRecursive(s1,s2,m,n)
    if m==0:
        return n
    if n==0:
        return m
    if s1[m-1] ==s2[n-1]:
        return EditDistanceRecursive(s1,s2,m-1,n-1)
    insert = EditDistanceRecursive(s1,s2,m,n-1)
    remove = EditDistanceRecursive(s1,s2,m-1,n)
    replace = EditDistanceRecursive(s1,s2,m-1,n-1)
```

```
        return 1+min(insert,remove,replace)

#Example usage

s1 = "horse"

s2 = "ros"

print("Minimum Edit Distance:", EditDistanceRecursive(s1, s2, len(s1), len(s2)))
```

8. Knapsack (0,1) problem:

Problem statement:

The Knapsack Problem involves selecting the most valuable items that can be carried in a knapsack with a maximum capacity of C. Given n items, each item i has a volume v_i and a value w_i, both as integers. The goal is to determine which subset of items to take so that the total value is maximized while ensuring that the total volume does not exceed C. The constraint is that each item must be fully included or excluded—it cannot be taken partially. Thus, the total volume and value will increase by v_i and w_i if an item is chosen or remain unchanged if it is not.

Solution:

A 2D table can be constructed to store values progressively, solving the Knapsack Problem in O(n * W) time, where n is the number of items and W is the knapsack capacity. Since only n * W values need to be computed, this approach ensures efficient computation.

Let K[i, n] represent the maximum value obtained using the first i items for a knapsack of capacity n. The table is computed as follows:

- If item i is included: K[i,n]=wi+K[i−1,n−vi]K[i, n] = w_i + K[i - 1, n - v_i]K[i,n]=wi+K[i−1,n−vi]
- If item i is excluded: K[i,n]=K[i−1,n]K[i, n] = K[i - 1, n]K[i,n]=K[i−1,n]
- The table is then filled iteratively, ensuring each entry stores the maximum possible value.

Example:

Given items with values w = {1, 4, 5, 7} and volumes v = {1, 3, 4, 5}, for a knapsack of capacity 7, we construct a table with (n+1) rows and (W+1) columns.

1. Initialize the first row with zero since no items result in zero value.
2. Iterate through the items and capacities, updating each entry to store the maximum obtainable value.
3. The final result is found in the bottom-right corner of the table, where K[n, W] = 9, indicating the maximum value that can be obtained while filling the knapsack to 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)

Pseudo code of knapsack algorithm:

Algorithm 6.4 KNAPSACK
Input: A set of items $U = \{u_1, u_2, \ldots, u_n\}$ with sizes $s_1, s_2, \ldots, s_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack capacity C.
Output: The maximum value of the function $\sum_{u_i \in S} v_i$ subject to $\sum_{u_i \in S} s_i \leq C$ for some subset of items $S \subseteq U$.

```
1.  for i ← 0 to n
2.      V[i, 0] ← 0
3.  end for
4.  for j ← 0 to C
5.      V[0, j] ← 0
6.  end for
7.  for i ← 1 to n
8.      for j ← 1 to C
9.          V[i, j] ← V[i − 1, j]
10.         if s_i ≤ j then V[i, j] ← max{V[i, j], V[i − 1, j − s_i] + v_i}
11.     end for
12. end for
13. return V[n, C]
```

9. String Matching:

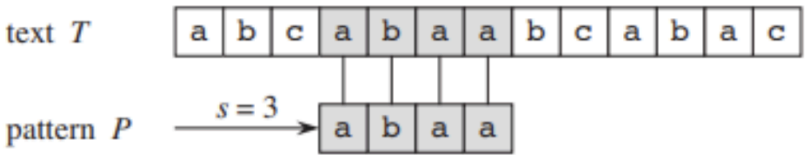
String Matching is the problem of finding one or more occurrences of a pattern (P) of length m within a text (T) of length n, where m ≤ n. The goal is to determine all positions (valid shifts s) in T where P appears as a contiguous substring.

Formally, a valid match occurs at shift s if:

T[s+1...s+m]=P[1...m]T[s+1 ... s+m] = P[1 ... m]T[s+1...s+m]=P[1...m]

meaning that each character of P aligns exactly with a corresponding substring in T.

String Matching Algorithms, such as Naive Search, Knuth-Morris-Pratt (KMP), Rabin-Karp, and Boyer-Moore, optimize this search process based on different techniques



We can also find the max 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 the DNA sequences. Internet search engines also use them to find the web pages relevant to queries.

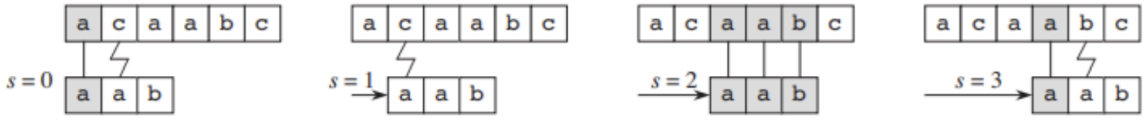
Initial Solution:

The naive string-matching algorithm checks for all possible shifts by iterating through the n - m + 1 positions in the text T. For each shift s, it verifies whether the pattern P[1...m] matches the substring T[s+1...s+m]. This brute-force approach ensures that all valid shifts are identified.

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-matching algorithm is inefficient because it does not use any information from previous checks when moving to the next position. This leads to unnecessary comparisons. For example, if P = "aaab" matches at s = 0, then shifts 1, 2, and 3 cannot be valid if T[4] ≠ 'b'. More efficient algorithms use this information to skip unnecessary checks and improve performance.
Time complexity: O(mn)

Solution refinement: (The Knuth-morris-pratt algorithm):

Before solving the problem, let's understand the concept of the "border of a string", which helps in optimizing pattern matching.

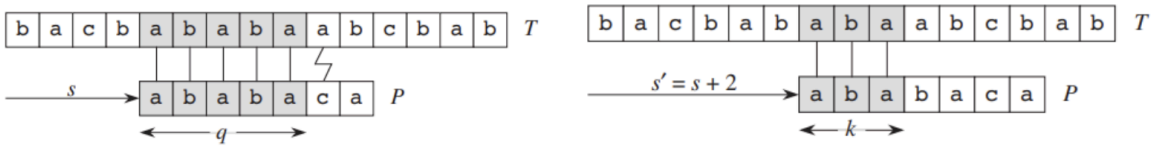
A border of a string is a prefix that is also a suffix, but it cannot be the entire string. For example, in "abcababcab", the borders are "ab" and "abcab". In "abacabab", the border is "ab", so its border length is 2.

This concept helps in skipping unnecessary comparisons when a mismatch occurs in pattern matching. Instead of checking every shift, we can jump directly to the suffix position of the pattern, avoiding invalid shifts.

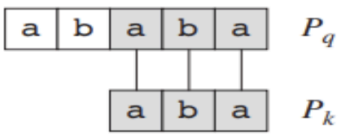
The border prefix function (b) keeps track of how the pattern matches itself at different shifts. This allows us to avoid redundant checks while searching in the text.

For example, consider the pattern P = "ababaca" being matched against a text T. If the first 5 characters match but the 6th character does not, we already know that the next shift (s + 1) is invalid. Instead of shifting by one, we use border information to jump further (s + 2), aligning only the necessary part of the pattern with the text.

This technique is used in efficient pattern-matching algorithms like Knuth-Morris-Pratt (KMP) to reduce unnecessary comparisons.



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



1	2	3	4	5	6	7	i
a	b	a	b	a	c	a	P[i]
0	0	1	2	3	0	1	b[i]

Pseudo Code:

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(m+n)

Space complexity: O(m)

References:

- M.H. Alsuwaiyel, Algorithms Design Techniques and Analysis (Revised Edition)
- Scribe notes of 24BM6JP12_2025-01-20
- Scribe notes of 24BM6JP13_2025-01-21
- Scribe notes of 24BM6JP14_2025-01-21
- Scribe notes of 24BM6JP14_2025-01-23