


☐

I'm not robot


reCAPTCHA

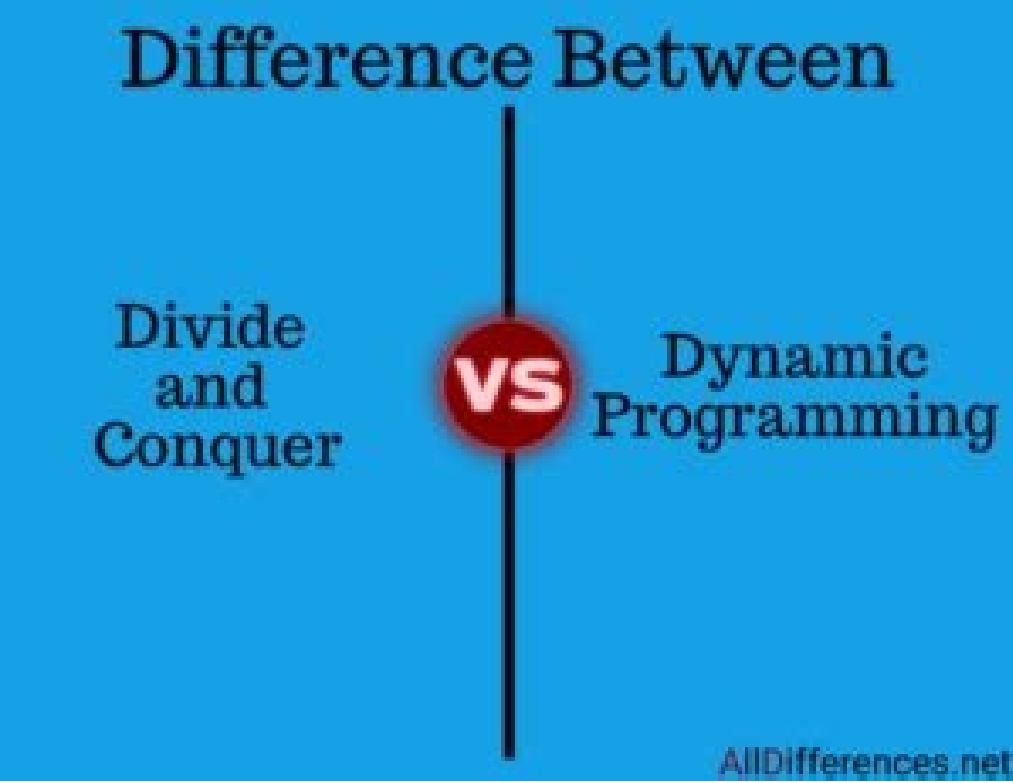
Continue

Difference between divide and conquer and dynamic programming pdf

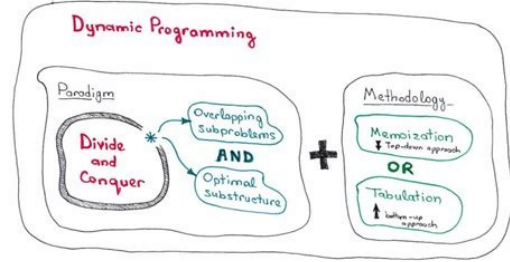
ReadDiscussCoursesPracticeImprove Article Save Article Like Article Greedy algorithm, divide and conquer algorithm, and dynamic programming algorithm are three common algorithmic paradigms used to solve problems. Here's a comparison among these algorithms:Approach:Greedy algorithm: Makes locally optimal choices at each step with the hope of finding a global optimum.Divide and conquer algorithm: Breaks down a problem into smaller subproblems, solves each subproblem recursively, and then combines the solutions to thesubproblems to solve the original problem.Dynamic programming algorithm: Solves subproblems recursively and stores their solutions to avoid repeated calculations.Goal:Greedy algorithm: Finds the best solution among a set of possible solutions.Divide and conquer algorithm: Solves a problem by dividing it into smaller subproblems, solving each subproblem independently, and then combining thesolutions to the subproblems to solve the original problem.Dynamic programming algorithm: Solves a problem by breaking it down into smaller subproblems and solving each subproblem recursively.Time complexity:Greedy algorithm: O(nlogn) or O(n) depending on the problem.Divide and conquer algorithm: O(nlogn) or O(n^2) depending on the problem.Dynamic programming algorithm: O(n^2) or O(n^3) depending on the problem.Space complexity:Greedy algorithm: O(1) or O(n) depending on the problem.Divide and conquer algorithm: O(nlogn) or O(n^2) depending on the problem.Dynamic programming algorithm: May or may not provide the optimal solution.Divide and conquer algorithm: May or may not provide the optimal solution.Dynamic programming algorithm: Guarantees the optimal solution.Examples:Greedy algorithm: Huffman coding, Kruskal's algorithm, Dijkstra's algorithm, etc.Divide and conquer algorithm: Merge sort, Quick sort, binary search, etc.Dynamic programming algorithm: Fibonacci series, Longest common subsequence, Knapsack problem, etc.In summary, the main differences among these algorithms are their approach, goal, time and space complexity, and their ability to provide the optimal solution. Greedy algorithm and divide and conquer algorithm are generally faster and simpler, but may not always provide the optimal solution, while dynamic programming algorithm guarantees the optimal solution but is slower and more complex. Greedy Algorithm:Greedy algorithm is defined as a method for solving optimization problems by taking decisions that result in the most evident and immediate benefit irrespective of the final outcome. It is a simple, intuitive algorithm that is used in optimization problems.Divide and conquer is an algorithmic paradigm in which the problem is solved using the Divide, Conquer, and Combine strategy.

Greedy Algorithm Design	
Comparison: Dynamic Programming <ul style="list-style-type: none">At each step, the choice is determined based on solutions of subproblems.Sub-problems are solved first.Bottom-up approachCan be slower, more complex	Greedy Algorithms <ul style="list-style-type: none">At each step, we quickly make a choice that currently looks best.A local optimal (greedy) choice.Greedy choice can be made first before solving further sub-problems.Top-down approachUsually faster, simpler

A typical Divide and Conquer algorithm solve a problem using the following three steps:Divide: This involves dividing the problem into smaller sub-problems.Conquer: Solve sub-problems by calling recursively until solved.Combine: Combine the sub-problems to get the final solution of the whole problem.Dynamic Programming:Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has sometimes repeated calls for the same input states, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.Greedy Algorithm vs Divide and Conquer Algorithm vs Dynamic Algorithm.S.NoGreedy AlgorithmDivide and conquerDynamic Programming1Follows Top-down approachFollows Top-down approachFollows bottom-up approach2Used to solve optimization problemUsed to solve decision problemUsed to solve optimization problem3The optimal solution is generated without revisiting previously generated solutions; thus, it avoids the re-computationSolution of subproblem is computed recursively more than once.The solution of subproblems is computed once and stored in a table for later use.4It may or may not generate an optimal solution. It is used to obtain a solution to the given problem, it does not aim for the optimal solution! always generates optimal solution.5Iterative in nature.Recursive in nature.Recursive in nature.6 efficient and fast than divide and conquer. For instance, single source shortest path finding using Bellman Ford Algo takes O(VE) time.7Extra memory is not required.some memory is required.more memory is required to store subproblems for later use.8Examples: Fractional Knapsack problem, Activity selection problem, Job sequencing problem.Examples: Merge sort, Quick sort, Strassen's matrix multiplication.Examples: 0/1 Knapsack, All pair shortest path, Matrix-chain multiplication.Last Updated : 17 Mar, 2023Like Article Save Article In this article I'm trying to explain the difference/similarities between dynamic programming and divide and conquer approaches based on two examples: binary search and minimum edit distance (Levenshtein distance).The ProblemWhen I started to learn algorithms it was hard for me to understand the main idea of dynamic programming (DP) and how it is different from divide-and-conquer (DC) approach. When it gets to comparing those two paradigms usually Fibonacci function comes to the rescue as great example. But when we're trying to solve the same problem using both DP and DC approaches to explain each of them, it feels for me like we may lose valuable detail that might help to catch the difference faster. And these detail tells us that each technique serves best for different types of problems.I'm still in the process of understanding DP and DC difference and I can't say that I've fully grasped the concepts so far. But I hope this article will shed some extra light and help you to do another step of learning such valuable algorithm paradigms as dynamic programming and divide-and-conquer.Dynamic Programming and Divide-and-Conquer SimilaritiesAs I see it for now I can say that dynamic programming is an extension of divide and conquer paradigm.I would not treat them as something completely different.



Because they both work by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.So why do we still have different paradigm names then and why I called dynamic programming an extension. It is because dynamic programming approach may be applied to the problem only if the problem has certain restrictions or prerequisites. And after that dynamic programming extends divide and conquer approach with memoization or tabulation technique.Let's go step by step...Dynamic Programming Prerequisites/RestrictionsAs we've just discovered there are two key attributes that divide and conquer problem must have in order for dynamic programming to be applicable:Once these two conditions are met we can say that this divide and conquer problem may be solved using dynamic programming approach.Dynamic Programming Extension for Divide and ConquerDynamic programming approach extends divide and conquer approach with two techniques (memoization and tabulation) that both have a purpose of storing and re-using sub-problems solutions that may drastically improve performance.



For example naive recursive implementation of Fibonacci function has time complexity of O(2^n) where DP solution doing the same with only O(n) time.Memoization (top-down cache filling) refers to the technique of caching and reusing previously computed results. The memoized fib function would thus look like this:memFib(n) { if (mem[n] is undefined) if (n < 2) result = n else result = memFib(n-2) + memFib(n-1) mem[n] = result return mem[n] }Tabulation (bottom-up cache filling) is similar but focuses on filling the entries of the cache. Computing the values in the cache is easiest done iteratively. The tabulation version of fib would look like this: tabFib(n) { mem[0] = 0 mem[1] = 1 for i = 2..n mem[i] = mem[i-2] + mem[i-1] return mem[n] }You may read more about memoization and tabulation comparison here.The main idea you should grasp here is that because our divide and conquer problem has overlapping sub-problems the caching of sub-problem solutions becomes possible and thus memoization/tabulation step up onto the scene.So What the Difference Between DP and DC After AllSince we're now familiar with DP prerequisites and its methodologies we're ready to put all that was mentioned above into one picture.Dynamic programming and divide and conquer paradigms dependencyLet's go and try to solve some problems using DP and DC approaches to make this illustration more clear.Divide and Conquer Example: Binary SearchBinary search algorithm, also known as half-interval search, is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array; if they are unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until the target value is found.



If the search ends with the remaining half being empty, the target is not in the array.ExampleHere is a visualization of the binary search algorithm where 4 is the target value.Binary search algorithm logicLet's draw the same logic but in form of decision tree.Binary search algorithm decision treeYou may clearly see here a divide and conquer principle of solving the problem. We're iteratively breaking the original array into sub-arrays and trying to find required element in there.Can we apply dynamic programming to it? No. It is because there are no overlapping sub-problems. Every time we split the array into completely independent parts. And according to divide and conquer prerequisites/restrictions the sub-problems must be overlapped somehow.Normally every time you draw a decision tree and it is actually a tree (and not a decision graph) it would mean that you don't have overlapping sub-problems and this is not dynamic programming problem.The CodeHere you may find complete source code of binary search function with test cases and explanations. function binarySearch(sortedArray, seekElement) { let startIndex = 0; let endIndex = sortedArray.length - 1; while (startIndex <= endIndex) { const middleIndex = startIndex + Math.floor((endIndex - startIndex) / 2); // If we've found the element just return its position. if (sortedArray[middleIndex] === seekElement)) { return middleIndex; } // Decide which half to choose: left or right one. if (sortedArray[middleIndex] < seekElement)) { // Go to the right half of the array. startIndex = middleIndex + 1; } else { // Go to the left half of the array.

~ simple idea: memoize
memo = {}
fib(n) {
 if n in memo: return memo[n]
 else: if n <= 2: f = 1
 else: f = fib(n-1) + fib(n-2)
 memo[n] = f
 return f
}
=> T(n) = T(n-1) + O(1) = O(n)

endIndex = middleIndex - 1; } } return -1; }Dynamic Programming Example: Minimum Edit DistanceNormally when it comes to dynamic programming examples the Fibonacci number algorithm is being taken by default. But let's take a little bit more complex algorithm to have some kind of variety that should help us to grasp the concept.Minimum Edit Distance (or Levenshtein Distance) is a string metric for measuring the difference between two sequences. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other.ExampleFor example, the Levenshtein distance between "kitten" and "sitting" is 3, since the following three edits change one into the other, and there is no way to do it with fewer than three edits:ApplicationsThis has a wide range of applications, for instance, spell checkers, correction systems for optical character recognition, fuzzy string searching, and software to assist natural language translation based on translation memory.Mathematical DefinitionMathematically, the Levenshtein distance between two strings a, b (of length |a| and |b| respectively) is given by function lev(|a|, |b|) whereNote that the first element in the minimum corresponds to deletion (from a to b), the second to insertion and the third to match or mismatch, depending on whether the respective symbols are the same.ExplanationOk, let's try to figure out what that formula is talking about. Let's take a simple example of finding minimum edit distance between strings ME and MY. Intuitively you already know that minimum edit distance here is 1 operation and this operation is "replace E with Y". But let's try to formalize it in a form of the algorithm in order to be able to do more complex examples like transforming Saturday into Sunday.To apply the formula to ME>MY transformation we need to know minimum edit distances of ME>M, M>MY and M>M transformations in prior. Then we will need to pick the minimum one and add +1 operation to transform last letters E/Y.So we can already see here a recursive nature of the solution: minimum edit distance of ME>MY transformation is being calculated based on three previously possible transformations. Thus we may say that this is divide and conquer algorithm.To explain this further let's draw the following matrix.Simple example of finding minimum edit distance between ME and MY stringsCell (0, 1) contains red number 1. It means that we need 1 operation to transform M to empty string; delete M. This is why this number is red.Cell (0, 2) contains red number 2. It means that we need 2 operations to transform ME to empty string; delete E, delete M.Cell (1, 0) contains green number 1. It means that we need 1 operation to transform empty string to M; insert M. This is why this number is green.Cell (2, 0) contains green number 2. It means that we need 2 operations to transform empty string to MY; insert Y, insert M.Cell (1, 1) contains number 0. It means that it costs nothing to transform M to M.Cell (1, 2) contains red number 1. It means that we need 1 operation to transform ME to M; delete E.And so on...This looks easy for such small matrix as ours (it is only 3x3). But how we could calculate all those numbers for bigger matrices (let's say 9x7 one, for Saturday>Sunday transformation)?The good news is that according to the formula you only need three adjacent cells (i-1, j), (i-1, j-1), and (i, j-1) to calculate the number for current cell (i, j) . All we need to do is to find the minimum of those three cells and then add +1 in case if we have different letters in i-s row and j-s columnSo once again you may clearly see the recursive nature of the problem.Recursive nature of minimum edit distance problemOk we've just found out that we're dealing with divide and conquer problem here. But can we apply dynamic programming approach to it? Does this problem satisfies our overlapping sub-problems and optimal substructure restrictions? Yes. Let's see it from decision graph.Decision graph for minimum edit distance with overlapping sub-problemsFirst of all this is not a decision tree. It is a decision graph. You may see a number of overlapping subproblems on the picture that are marked with red. Also there is no way to reduce the number of operations and make it less then a minimum of those three adjacent cells from the formula.Also you may notice that each cell number in the matrix is being calculated based on previous ones. Thus the tabulation technique (filling the cache in bottom-up direction) is being applied here. You'll see it in code example below.Applying these principles further we may solve more complicated cases like with Saturday > Sunday transformation.Minimum edit distance to convert Saturday to SundayThe CodeHere you may find complete source code of minimum edit distance function with test cases and explanations. function levenshteinDistance(a, b) { const distanceMatrix = Array(b.length + 1).fill(null).map(() => Array(a.length + 1).fill(null)); for (let i = 0; i <= a.length; i += 1) { distanceMatrix[0][i] = i; } for (let j = 0; j <= b.length; j += 1) { distanceMatrix[j][0] = j; } for (let j = 1; j <= b.length; j += 1) { for (let i = 1; i <= a.length; i += 1) { const indicator = a[i - 1] === b[j - 1] ? 0 : 1; distanceMatrix[j][i] = Math.min(distanceMatrix[j][i - 1] + 1, // deletion distanceMatrix[j - 1][i] + 1, // insertion distanceMatrix[j - 1][i - 1] + indicator, // substitution); } } return distanceMatrix[b.length][a.length]; }ConclusionIn this article we have compared two algorithmic approaches such as dynamic programming and divide-and-conquer. We've found out that dynamic programming is based on divide and conquer principle and may be applied only if the problem has overlapping sub-problems and optimal substructure (like in Levenshtein distance case). Dynamic programming then is using memoization or tabulation technique to store solutions of overlapping sub-problems for later usage.I hope this article hasn't brought you more confusion but rather shed some light on these two important algorithmic concepts! You may find more examples of divide and conquer and dynamic programming problems with explanations, comments and test cases in JavaScript Algorithms and Data Structures repository.Happy coding!Dynamic programming and divide-and-conquer are two commonly used algorithms design techniques that can be used to solve a variety of problems.Dynamic Programming is a technique used for solving problems by breaking them down into smaller overlapping subproblems and storing the results of these subproblems to avoid redundant computation. It is used when the problem exhibits optimal substructure, meaning that the optimal solution can be constructed from optimal solutions of subproblems. It is well-suited for problems that have an inherent sequence, such as making a sequence of decisions. Some common examples of problems solved using dynamic programming include finding the longest common subsequence, the shortest path in a graph, and the Fibonacci sequence.Divide-and-Conquer is a technique used for solving problems by breaking them down into smaller subproblems that can be solved independently and then combining these solutions to obtain the solution to the original problem. It is used when the problem can be divided into smaller subproblems that are similar to the original problem, and the solution to the subproblems can be combined to form a solution to the original problem. Some common examples of problems solved using divide-and-conquer include the quick sort algorithm, binary search, and the merge sort algorithm.In summary, dynamic programming is used when the problem has an optimal substructure and can be solved using a bottom-up approach, while divide-and-conquer is used when the problem can be divided into smaller subproblems that are similar to the original problem and solved using a top-down approach.