**Backtracking comparison to Dynamic Programming**

Dynamic Programming and Backtracking have multiple similarities and differences and are often confused when first learning about them. Often, the confusion comes simply from the name collision where there is a part of DP that involves 'backtracking' to recreate an optimal solution sequence at the end of the DP. For example, when finding the minimum path through a grid in DP we may want to minimize the number of jumps taken, but to reconstruct that final path we often need to go backwards from our final solution to retrace all of the steps that got us to the optimal point in order to reconstruct the minimal path. This is not the Backtracking algorithm.

In backtracking, we are exhaustively searching through a solution space by applying local transformations and collecting solutions we find as we go. It is named 'backtracking' because we will need to back out of each local transformation we make to our current solution in order to continue searching the rest of the space. Imagine searching a labrynth and you have no information so your only choice is to search all paths out. Of course you leave a trail of crumbs behind you so that once you hit a dead end you can backtrack along the crumbs to return to the first position where you could have made a different decision, i.e., all forks in the road. You can continue to do this until you have explored all possible paths through the labrynth.

It is similar to DP in that we are exhaustively searching a solution space guaranteeing that we will touch all solutions that matter. And it is also similar to DP in that the difference between a highly performant algorithm and not is often how you structure your search. In the case of backtracking, the best solutions often model the problem in some way that allows them to quickly prune state prefixes that cannot lead to solutions. 评价一个好的backtracking的解法重点在于能多快的剪枝

Unlike DP, backtracking is typically not looking for one optimal solution, but is instead looking for all that satisfy some criteria. So the structure of the algorithm is much less about being efficient at throwing previous information that you don't need because it doesn't lead you to the optimal solution and more about learning what parts of the space you don't need to explore as you are collecting all solutions that qualify.

**比如对于L216, L903为何适用于Backtracking思想，是因为我们需要求出所有的满足要求的permutation的总个数，而不是唯一的一个最优解，Unlike DP, backtracking is typically not looking for one optimal solution, but is instead looking for all that satisfy some criteria. 那么是否可以在找寻单一一个满足要求的permutation的过程中使用呢？依然以L903为例，在寻找单一一个满足要求的permutation的过程中为了有效节约时间，对于已经遇到过的子状态(overlapped sub stats)不用再搜索一遍，直接用memoization处理**

**Refer to**

[L216.Combination Sum III](note://WEB322b73c0787cd12b4eed700838bed14d)

[L903.Valid Permutations for DI Sequence (Ref.L629,L216)](note://WEB5384ba088f568c44d4f27c5a9048997b)

**DFS vs Backtracking vs DP**

**Explain 1: What is difference between backtracking and depth first search?**

**Refer to**

<https://leetcode.com/discuss/general-discussion/136503/what-is-difference-between-backtracking-and-depth-first-search>

A1:

In Backtracking, we need to restore previous state of visited node, by making visited=false, after exploring current path whereas in DFS the state of the node remains same after a path is explored so that it will not be explored again. Or we can say that pure DFS is a variant of backtracking in which state of visited nodes are not restored, and this variant is only useful for problems related to searching (reachability, etc) and not for problems involving pattern finding, for which we need to use the usual backtracking tree pruning algorithm.DFS is quicker than a generic backtracking algorithm, due to the reason stated above.

A2:

Difference 1:DFS handles an explicit tree.While Backtracking handles an implicit tree

Difference 2:**Depth First Search is a special type of backtracking algorithmic design paradigm where the process of backtracking takes place in the leaf nodes.** **In case of backtracking,the algorithm also rejects the useless branch of the state-space tree.This is why DFS maintains the entire tree structure while Backtracking maintains a pruned tree**

**DFS是一种特殊的backtracking，只在叶子结点执行backtracking，而backtracking会在条件不满足时优化掉部分树的分支，这就是为何DFS遍历了完整的树状结构而backtracking会对树状结构进行剪枝操作**

**Explain 2: What's the difference between backtracking and depth first search?**

**Refer to**

<https://stackoverflow.com/questions/1294720/whats-the-difference-between-backtracking-and-depth-first-search>

Backtracking is a more general purpose algorithm.

Depth-First search is a specific form of backtracking related to searching tree structures. From Wikipedia:

One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

It uses backtracking as part of its means of working with a tree, but is limited to a tree structure.

Backtracking, though, can be used on any type of structure where portions of the domain can be eliminated - whether or not it is a logical tree. The Wiki example uses a chessboard and a specific problem - you can look at a specific move, and eliminate it, then backtrack to the next possible move, eliminate it, etc.

**Explain 3: What's the difference between backtracking and depth first search?**

**Refer to**

<https://stackoverflow.com/questions/1294720/whats-the-difference-between-backtracking-and-depth-first-search>

The way I look at DFS vs. Backtracking is that backtracking is much more powerful. DFS helps me answer whether a node exists in a tree, while backtracking can help me answer all paths between 2 nodes.

Note the difference: DFS visits a node and marks it as visited since we are primarily searching, so seeing things once is sufficient. Backtracking visits a node multiple times as it's a course correction, hence the name backtracking.

Most backtracking problems involve:

def dfs(node, visited):

  visited.add(node)

  for child in node.children:

    dfs(child, visited)

  visited.remove(node) # this is the key difference that enables course correction and makes your dfs a backtracking recursion.

**Explain 4: How do I decide between Dynamic Programming vs Backtracking?**

**Refer to**

<https://www.reddit.com/r/leetcode/comments/ntuycc/how_do_i_decide_between_dynamic_programming_vs/>

At the end of the day at the heart, everything is DFS.

DFS + not visiting an invalid node = Backtracking.

DFS + not visiting node twice = Dynamic Programming. [let's ignore tabular for now]

2. You are concerned with what the actual solutions are rather than say the most optimum value of some parameter. (if it were the latter it’s most likely DP or greedy).

Imagine all the subset possible for your array. [1,1,2,5,3,4]

Backtracking will ask you how many ways you can add these numbers to sum of 5. ie [1,4], [1,4], [2,3], [5]

DP problem will ask you like the optimal answer. Like what's the least number of digits you can use to have sum of 5. ie [5] so answer is 1 digit.

Do you see the difference? One ask you basically " what the actual solutions" is while other is asking you "say the most optimum value of some parameter".

DP problems in these kind of question will ask you a some min/max/best/least out of your subset where as backtracking will ask all the way possible ways.

**Explain 5: Is it possible to convert all backtracking algorithms in to dynamic programming approach?**

**Refer to**

<https://stackoverflow.com/questions/34926329/is-it-possible-to-convert-all-backtracking-algorithms-in-to-dynamic-programming>

Q:

Is it possible to convert all backtracking algorithms in to dynamic programming approach? I have tried a few backtracking algorithms and successfully converted them to dynamic programming by applying the concept of memoization. Is it possible to convert every backtracking algorithm to dynamic programming? If dynamic programming is so much efficient than backtracking, why we still use backtracking?

A:

1.We use dynamic programming for problems that have overlapping sub-problems. Thats the criteria for a problem to be solve using dynamic programming. –

2.I would solve a sodoku puzzle with backtracking, though I do not see how we could apply dynamic programming there. So I don't think the set of problems that can be solved by backtracking is a subset of the set of problems that can be solved by some sort of dynamic programming. –

3.Backtracking algorithms often make use of pruning, you usually can't prune in DP because you don't know what you'll need yet. (I'm excluding memoization from DP here because it's not DP, it's memoization) –

A potential way to transform backtracking into DFS / DP may go with divide and conquer -> why divide and conquer ?  because backtracking usually used when try to find all potential solutions, which means more like 0/1 knapsack, and 0/1 knapsack on DFS perspective is very similar to divide and conquer (pick or not pick each element), right ?

Backtracking (focus on all solution) can or cannot transform into DP (focus on optimal solution) -> more like Backtracking cannot directly transform into DP, the better choice is find a pure / Native DFS solution, then transform it into DP

**Refer to**

<https://www.javatpoint.com/dynamic-programming-vs-backtracking>

**Dynamic programming vs Backtracking**

Before understanding the differences between dynamic programming and backtracking, we should know about dynamic programming and backtracking separately.

**What is Backtracking?**

Backtracking is one of the problem-solving techniques. Using this technique, we can solve our problem. This strategy uses a Brute force approach, and the brute force approach says that for the given problem, we should try out all the possible solutions and pick the desired solution from all the possible solutions. In contrast, dynamic programming is used to solve the optimization problem, but backtracking is not used to solve the optimization problems. When multiple solutions to a given problem exist, then backtracking uses all the solutions to solve the problem.

**Now we will understand that how dynamic programming uses the brute force approach through an example.**

Suppose there are three are three students, two are girls, and one is a boy. We have three chairs, and we have to arrange these students in these chairs. In how many ways we can arrange these students?. As there are 3 students, we can arrange these students in 3! ways, i.e., 6 ways. Now, we have to find out all the possible arrangements and all the arrangements or solutions can be represented in the form of a tree known as a state-space tree.

G1, G2, B1

**State-space tree:**

**First arrangement**

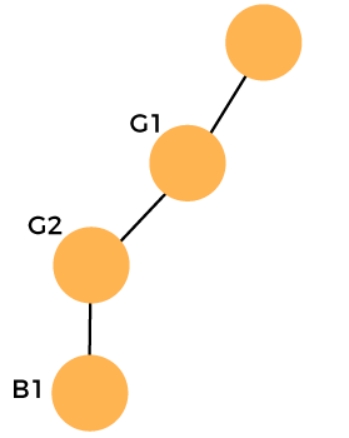
First, consider a node shown as below:



Consider the first girl G1 in the first level, i.e., first chair shown as below:

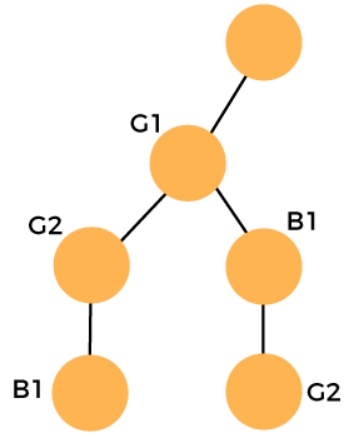
Consider the second girl G2 in the next level, i.e., second chair shown as below:

Consider a boy B1 in the next level, i.e., third chair shown as below:



**Second arrangement**

We will backtrack. First, we remove B1 from the third chair and G2 from the second chair. Add B1 at the second chair and then G2 at the third chair shown as below:

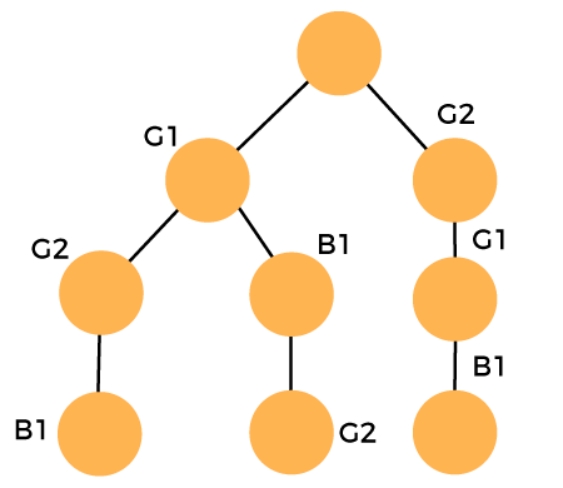


**Third arrangement**

We will backtrack again. First, we remove G2 from the third chair, then we remove B1 from the second chair and then we remove the G1 from the first chair. Consider the G2 in the first chair shown as below:

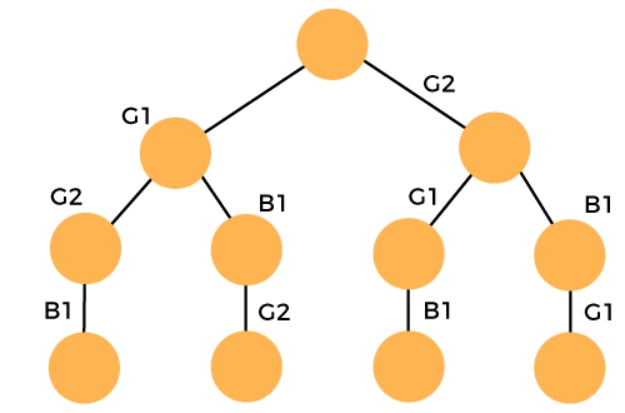
Consider the G1 in the next level, i.e., second chair shown as below:

Consider the B1 in the third level, i.e., third chair shown as below:



**4th arrangement**

We will backtrack again. First, we remove B1 from the third chair then we remove G1 from the second chair. Add B1 at the second chair and then G1 at the third chair shown as below:



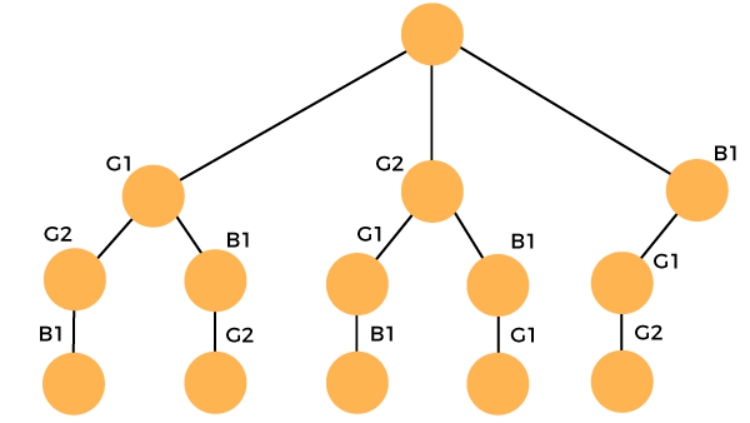
**5th arrangement**

We will backtrack again. First, we remove G1 from the third chair then we remove B1 from the second chair and then we remove G2 from the first chair.

Consider B1 in the first level, i.e., first chair shown as below:

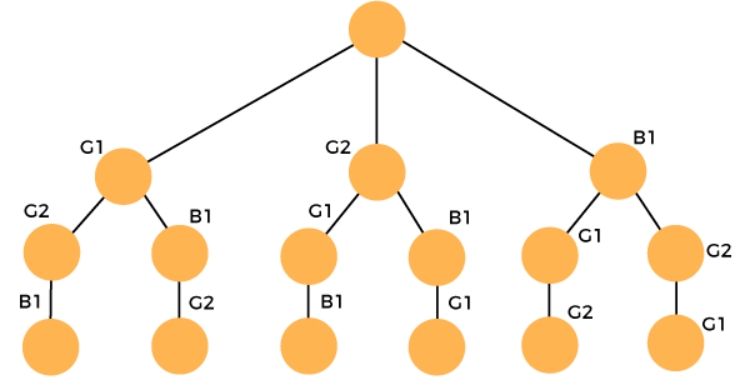
Consider G1 in the next level, i.e., second chair shown as below:

Consider G2 in the next level, i.e., third chair shown as below:



**6th arrangement**

We will backtrack again. First, we remove G2 from the third chair and then we remove G1 from the second chair. Add G2 at the second chair and then add G1 at the third chair shown as below:



**What is Dynamic Programming?**

Dynamic programming is a technique for solving certain type of complex problems efficiently by breaking them down into simpler subproblems and solving each problem exactly once. Dynamic programming stores the result of a subproblem in a table and reuse them when needed to avoid solving the same problems again and again.

**What type of problems can be solved using dynamic programming?**

The following are the two problems that can be solved using dynamic programming:

**o Optimal substructure:** A given problem has optimal substructure property if optimal solution of the given problem can be obtained using the optimal solutions of subproblems. In other words, we can define the solution to the problem by using a recurrence relationship based on its subproblems.

**o Overlapping subproblems:** A given problem has overlapping subproblems property if to solve the problem we have to solve its subproblems multiple times.

Using dynamic programming approach, we avoid solving overlapping subproblems and we solve each problem only once and save the result in a cache. When its needed again we will get the result from the cache instead of solving them again.

**Approaches of Dynamic programming**

There are two approaches used to implement the dynamic programming:

**o Top-down approach:** It is also known as a memoization. It is implemented using both recursion and caching. Whenever the recursive function is called, we check the cache to see whether the problem has already been solved. If it is already solved then we return the result from the cache else we will solve the subproblem, save the result into cache and return the result.

**o Bottom-up approach:** It is also known as tabulation. This technique is used to solve all the smaller subproblems first then move to the larger subproblem that use the result of the smaller subproblems.

**Differences between Dynamic programming and Backtracking**



o Dynamic programming is a technique of dividing the complex problem into simpler sub-problems. This technique is applicable to the problems that exhibit the following properties:

Overlapping subproblems

Optimal sub-structure

Backtracking is a technique that recursively build a solution incrementally and remove all the solutions that does not satisfy the constraints of the problem at any point of time.

o The major difference between the dynamic programming and backtracking is that the dynamic programming completely relies on the principle of optimality which means that the sub sequence of a sequence should be optimal. In contrast to dynamic programming, backtracking does not guarantee the complete optimal solution.

o Dynamic programming is a technique that solves the optimization problem. Optimization problem uses either minimum or maximum result. In contrast to dynamic programming, backtracking uses the brute force approach without considering the optimization problem. If we have multiple solutions then it considers all those solutions.