The idea of string similarity comes in Machine Translation, Information Extraction, Speech Recognition.

In machine translation, we might want to know how well a machine translation system does.

Suppose the machine translation system represented some sentence translated from Chinese.

We can check how many words were changed or inserted.

Here we are checking for complete words and not single characters.

Markov’s Chain.

Formula for finding number of non-adjacent subsets of given numbers.

Application of minimum edit distance.

HashMap.equals() method compares 2 hashmaps by key-value pairs.

It means both hashmap instances must have exactly same key-value pairs and both must be of same size.

The order of key-value pairs can be different and does not play a role in comparison.

For optimization:

If we are returning a boolean on some count

Instead of doing an if-else on the value of the count, return the boolean which has the comparison.

**DFS vs BFS**:

Heavily depends on the structure of the search tree and the number and location of solutions.

* If we know a solution is not far from the root of the tree, a breadth first search (BFS) might be better.
* If the tree is very deep and the solutions are rare, depth first search (DFS) might take an extremely long time, but BFS could be better.
* If the tree is very wide, a BFS might need too much memory, so it might be completely impractical.
* If solutions are frequent but located deep in the tree, BFS could be impractical.
* If the search tree is very deep we will need to restrict the search depth for depth first search (DFS),

BFS has an interesting property. It first finds all the vertices that are one edge away from the starting point, then all the vertices that are 2 edges away, and so on. This is useful if we are trying to find the shortest path from the starting vertex to a given vertex.

We start a BFS, and when we find the specified vertex, we know that path we have traced so far is the shortest path to the node. If there were a shorter path, the BFS would have found it already.

BFS can be used for finding the neighbour nodes in peer to peer networks like BitTorrent, GPS systems to find nearby locations, social networking sites to find people in the specified distance.

Breadth First Search is generally the best approach when the depth of the tree can vary, and you only need to search part of the tree for a solution.

For example, finding the shortest path from a starting value to a final value is a good place to use BFS.

Depth First Search is commonly used when you need to search the entire tree.

It's easier to implement (using recursion) than BFS, and requires less state:

While BFS requires you store the entire 'frontier', DFS only requires you store the list of parent nodes of the current element.

**IMP NOTE**:

In the collections we always use object types and not primitive types.

If we have List<Integer[ ]>

Arrays are of fixed size. So, to iterate over arrays we should use for-each loop.

Elements have to added in the array at the time of creation.

Adding elements into a list can be done anytime.

When thinking of using for-loops, think about why not while loop?

Nice way to convert 2d array into List<List<>>

Integer[][] array = { {1, 3, 4, 10}, {2, 5, 9, 11} , {6, 8, 12, 15} , {7, 13, 14, 16 } } ;

List<List<Integer>> list = **new** ArrayList<>();

**for**(Integer[] item: array) {

list.add(Arrays.*asList*(item)) ;

}

To get number of rows and columns from a 2d list,

**int** numOfColumns = list.get(0).size() ;

**int** numOfRows = list.size();

We can use Arrays.asList() like this:

List<Integer> array = **new** ArrayList<>(Arrays.*asList*(10, 15, 8, 12, 94, 81, 5, 2, 11));

Instead of creating an array and then passing it in the asList() function.

When a data structure is associated in a recursion, it is usually returned or passed in between the recursive calls.

**Note**:

In a DP, we don’t need to always use an array as a cache. We can use a map, etc.

To build solutions in Dynamic Programming, we can use chaining.

To add an element at the head of an ArrayList, we can use

list.add(0, yourObject);

to replace an element at a given index of an Array List, we use list.set (index, value)

Multi-dimensional array or list is like another list inside a list index.

See the lcs problem.

With SonarLint,

**Cognitive Complexity** is a measure of how difficult a unit of code is to intuitively understand. Unlike [Cyclomatic Complexity](https://docs.codeclimate.com/docs/cyclomatic-complexity), which determines how difficult your code will be to test, Cognitive Complexity tells you how difficult your code will be to read and understand.

Now, Code Climate can help you identify which methods are overly difficult to understand and prevent introducing them into your code.

We can measure the running time of a function.

**long** startTime = System.*nanoTime*();

functionCall();

**long** stopTime = System.*nanoTime*();

System.***out***.println("Time : " + (stopTime - startTime));

Some of the recursive problems can be converted into DP.

There are some constraints. The most obvious one is that recursive calls must overlap. I.e. during the execution of an algorithm, the recursive function must be called multiple times with the same parameters.

This lets you truncate the recursion tree by memoization. So you can always use memoization to reduce the number of calls.

However, this reduction of calls comes with a price. You need to store the results somewhere. The next obvious constraint is that you need to have enough memory.

This comes with a not-so obvious constraint. Memory access always requires some time.

You first need to find where the result is stored and then maybe even copy it to some location. So in some cases, it might be faster to let the recursion calculate the result instead of loading it from somewhere. But this is very implementation-specific and can even depend on the operating system and hardware setup.

**VIMP**:

* If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor
  + No overhead for recursion and less overhead for maintaining table
  + There are some problems for which the regular pattern of table accesses in the dynamic-programming algorithm can be exploited to reduce the time or space requirements even further
* If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required

Memoization vs Tabulation:

It depends on the problem.

*Memoization* usually requires more code and is less straightforward, but has computational advantages in some problems, mainly those which you do *not* need to compute all the values for the whole matrix to reach the answer.

*Tabulation* is more straightforward, but may compute unnecessary values. If you do need to compute all the values, this method is usually faster, though, because of the smaller overhead.

In Knapsack problem, memoization (Top Down) does better than bottom up approach.

Well I believe theoretically you should be able to solve a DP problem with either approach. However, there are instances when bottom up approach can become too expensive.

Consider a knapsack problem with the knapsack\_size = 200,000 and the num\_items = 2000. To fill in a two dimensional DP table with just ints is not going to be possible.

You'll exhaust the main memory of an ordinary computer. We do not require to fill in all the entries in a table to achieve the desired final computation. A recursive top-down approach is far superior in a case like this.

**List To Array:**

List<Integer> sourceList = Arrays.asList(0, 1, 2, 3, 4, 5);

Integer[] targetArray =

sourceList.toArray( new Integer[sourceList.size()] );

// We have to do this because array is of fixed size, so we first have to initialize that much amount of space and then copy the list elements to this allocated array.

Integer[] sourceArray = { 0, 1, 2, 3, 4, 5 };

List<Integer> targetList = Arrays.asList(sourceArray)

Better to use List<Integer[]> in place of List<int[]>

**Using Map and List for a tree**:

We create a Trie using a map because keys serve as the indices of the children nodes and the values serve as the actual child nodes.

For a node we not saving its name in the object, but storing the name of the children as keys in the map.

In a node we are also storing the word that can be formed by reaching that particular node but we do so only for the node which has ‘\*’ as one of its children values. For other nodes, we store empty word.

We use a map because key values of a map are unique and we want the unique structure.

In a trie node, we are not storing the name of the node but storing the name of the children nodes as key values in the hash map.

In a map, we can check whether a key or a value exists in it separately but for an ArrayList, we have to check for a particular node value separately (cannot using contains() method).

If in case we did not want to store the words ‘this’ and ‘that’ under the same branch, then we can use an ArrayList (Remember B trees?)

The basic idea behind using Map and ArrayList to create a tree is because a node can have any number of children rather than having exactly 2 children as in a binary tree.

If we use an array list to create a tree, then the keys values here are the indices as a list is always ordered.

And when add a child of a node in the array list then we can do so according to the index or according to some other logic (Like defining the size of the list already and then adding a child to some particular index according to logic).

**IMP NOTE**:

In a tree, if number of vertices is n, then number of edges is n-1.

**Note**: Cannot convert int[] to List<Integer>

**IMP**:

If we are deleting an item from a list, then do not forget to add the check for non-empty list.

**Substring from nth occurrence of a character in a string**:

<https://stackoverflow.com/questions/3976616/how-to-find-nth-occurrence-of-character-in-a-string>