**D&C Template**

In the previous article of merge sort, we have introduced the general steps involved in all the divide-and-conquer algorithms. In this article, we will present you a ***pseudocode*** ***template*** that could help you to structure your code when implementing the algorithm in the divide-and-conquer paradigm. Furthermore, we will demonstrate with some concrete examples on how to apply the template.

*Template*

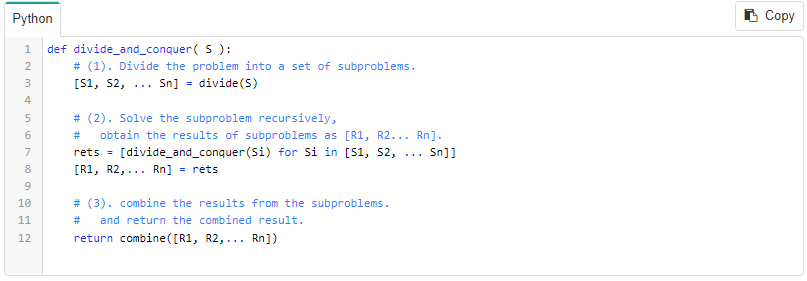
There are in general three steps that one can follow in order to solve the problem in a divide-and-conquer manner.

1.**Divide.** Divide the problem *S* into a set of subproblems: {*S*1​,*S*2​,...*Sn*​} where *n* ≥ 2, *i.e.* there are usually more than one subproblem.

2. **Conquer.** Solve each subproblem recursively.

3.**Combine.**Combine the results of each subproblem.

We can summarize the above steps in the following pseudocode template.



As one can see from the above template, the essential part of the divide and conquer is to figure out the recurrence relationship between the subproblems and the original problem, which subsequently defines the functions of divide() and combine().

In the next sections, we will show you how to apply the above template to implement the algorithms for some concrete examples.

*Validate Binary Search Tree*

Sometimes, tree related problems can be solved using divide-and-conquer algorithms.

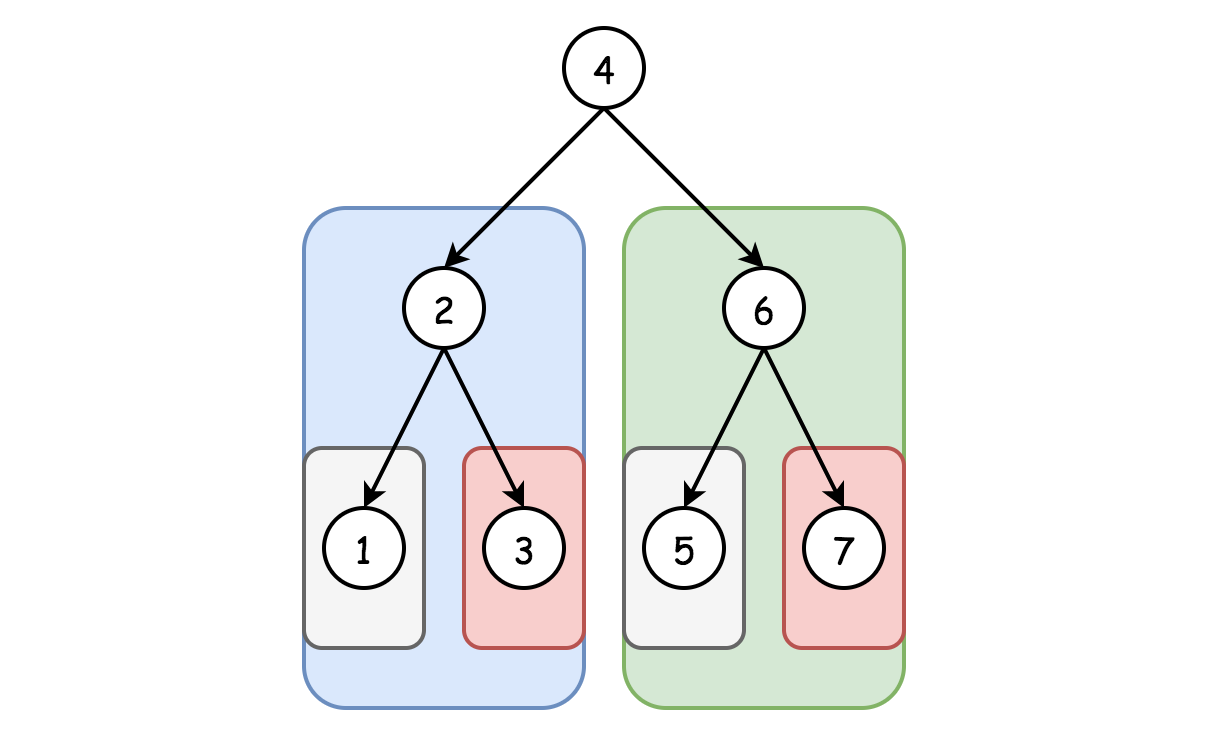
For example, look at the following problem statement:

Given a binary tree, validate if the given tree is a binary search tree (BST). The BST must meet all of the following properties:

1. All values on the left subtree of a node should be less than the value of the node.
2. All values on the right subtree of a node should be greater than the value of the node.
3. Both the left and right subtrees must also be binary search trees.

Read **point no. 3** above very carefully. The definition of BST is recursive in nature, making this a natural divide and conquer problem.

Below is an example of a BST shown in the following figure.



*Fig 2. Binary Search Tree*

**1.** In the first step, we divide the tree into two subtrees -- its left child and right child.  (**Divide**)

**2.** Then in the next step, we ***recursively*** validate each subtree is indeed a binary search tree.  (**Conquer**)

**3.** Upon the results of the subproblems from Step 2, we return true if and only if both subtrees are both valid BST.  (**Combine**)

The recursion in **Step 2.** would reach the base case where the subtree is either empty or contains a single node, which is a valid BST itself.

Notice some important details are still missing from **Step 2.** above, which are left as an exercise for the reader. For example, how do you verify these two properties which are also required for a BST?

1. All values on the left subtree of a node should be less than the value of the node.
2. All values on the right subtree of a node should be greater than the value of the node.

Following this article you can [try solving this exercise](https://leetcode.com/explore/learn/card/recursion-ii/470/divide-and-conquer/2874/) by yourself. We also provide a [step-by-step solution](https://leetcode.com/problems/validate-binary-search-tree/solution/) if you are still stuck following the exercise.

*Search a 2D Matrix II*

Write an efficient algorithm that searches for an integer value in an [*m*×*n*] matrix. This matrix has the following properties:

* Integers in each row are sorted in ascending from left to right.
* Integers in each column are sorted in ascending from top to bottom.

There are several ways to solve the above problem. Here we give an overall idea to solve it in the divide-and-conquer manner.

As one might notice, given the matrix, if we divide it into some sub-matrices by cutting it either by row and/or column, the resulting matrices would still keep the above two properties of the original matrix. Given the above insight, here is how we can apply the template to solve the problem.

***1***. We divide the matrix into 4 sub-matrices by choosing a pivot point based on a row and a column.  (**Divide**)

***2***. Then we ***recursively*** look into each sub-matrix to search for the desired target.  (**Conquer**)

***3***. If we find the target in either of the sub-matrices, we stop the search and return the result immediately.  (**Combine**)

The base cases in the above recursion would be either the input matrix is empty or it contains only a single element. As a simple strategy, one can choose the ***middle point*** both on the row and column as the pivot points to divide the matrix.

Do we really need to look into each of the divided 4 sub-matrices? Notice that the smallest and the largest element of the input matrix is located in the top left and bottom right corner respectively, which also applies to each of the divided sub-matrices. In fact, we need to only look into 3 of the sub-matrices.

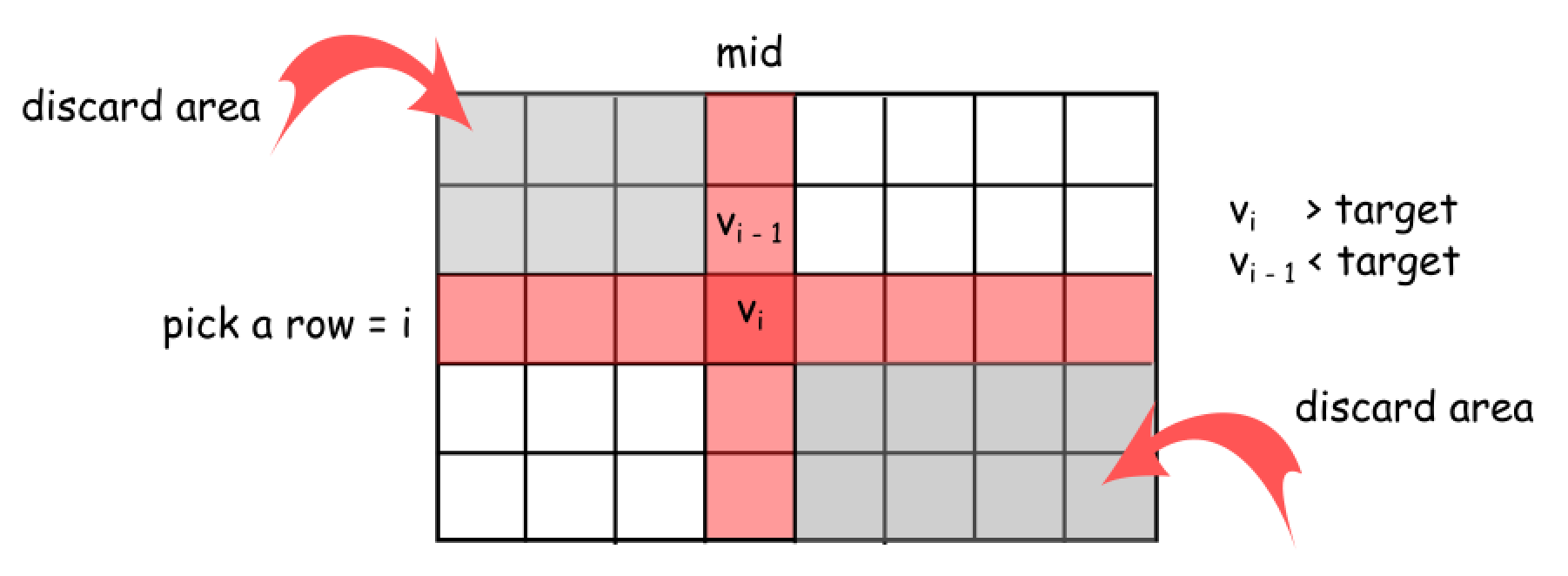
1. If our target is equal to the pivot, we have found our target and immediately return the result.
2. If our target is less than the pivot, we can discard the bottom-right sub-matrix. All elements in that region must be greater or equal than the pivot.
3. If our target is greater than the pivot, we can discard the top-left sub-matrix. All elements in that region must be less than or equal than the pivot.

We have just finished the divide-and-conquer approach, now try to [code the solution](https://leetcode.com/explore/learn/card/recursion-ii/470/divide-and-conquer/2872/) yourself! As a follow up exercise, can you derive the time complexity? At the end of this chapter, we provide [Master Theorem](https://leetcode.com/explore/learn/card/recursion-ii/470/divide-and-conquer/2871/) as an alternative way to derive time complexity like this problem.

The above divide-and-conquer algorithm can still be further improved, which we will provide insights below.

***As an improvement to the above divide-and-conquer algorithm, we could devise a better strategy by choosing the pivot points wisely.***

We illustrate a strategy in the following figure, to reduce the search zones into 2 sub-matrices, instead of 3 sub-matrices.



*Fig 1. Search 2D Matrix II*

First, we choose the middle point on the column which divides the matrix into two sub-matrices. We then fix on this middle column to further determine an optimal row to divide the matrix. We scan the elements along the chosen middle column, to locate the boundary where the value of the element just goes beyond the target value, *i.e.*  {*Vi*−1​<target<*Vi*​}. From this point, we divide the original matrix into 4 sub-matrices. And we just need to zoom into the bottom left and top right sub-matrices to look for the target value, while ignoring the top left and bottom right sub-matrices.

We ignore the top left sub-matrix that ends with the element *Vi*−1​, because all the elements within this sub-matrix would be less than the target value. Similarly, we ignore the bottom right sub-matrices that starts with the element *Vi*​, because we know that all the elements within this sub-matrix would be greater than the target value.