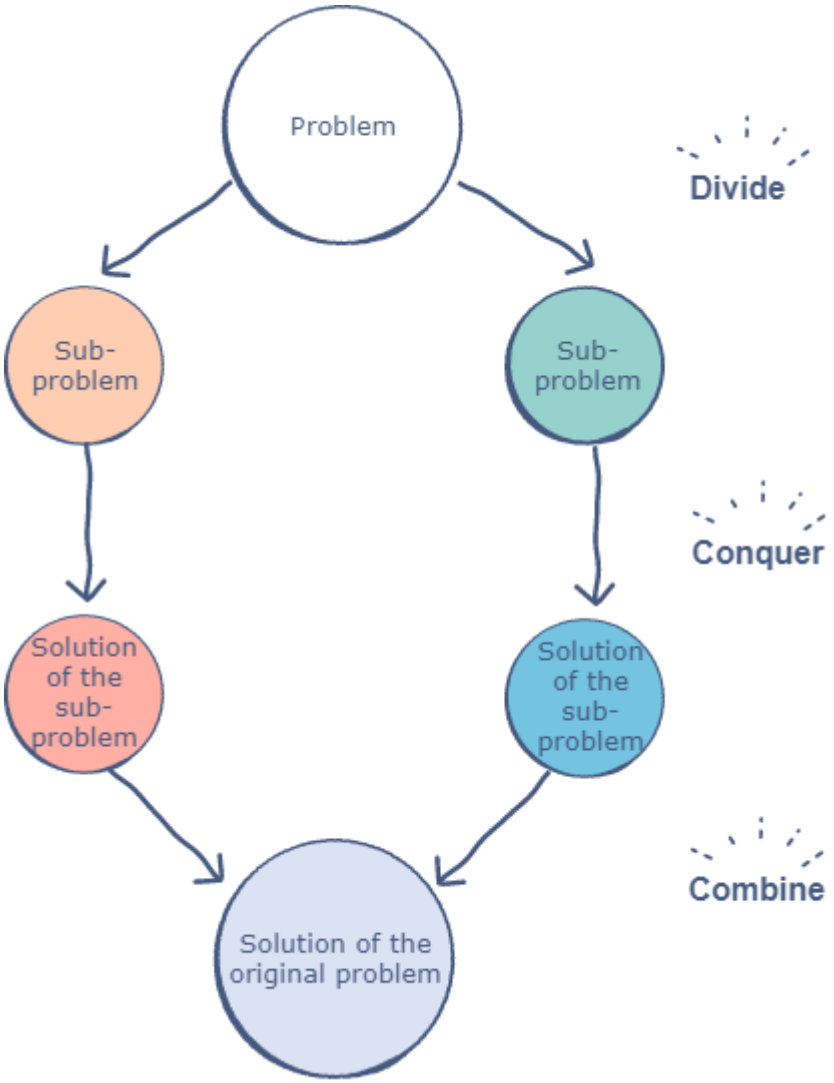
Divide and Conquer

Divide and conquer is an important algorithm design technique based on recursion. The divide and conquer strategy solves the problem by:

- **Divide:** breaking the problem into smaller subproblems that are themselves smaller instances of the same type of problem.
- **Conquer:** Conquer the subproblems by solving them recursively.
- **Combine:** Combine the solutions to the subproblems into the solution for the original given problem.

Divide and Conquer Visualization

Assume that n is the size of the original problem. As described above we can see that the problem is divided into subproblems with each of size n/b (for some constant b). We solve the subproblem recursively and combine the solutions to get the solution for the original problem.




```
DivideAndConquer( P ){
        if(small (P))
                //   P is very small so that the solution is obvious
                return solution( n );

        Divide the problem P into k subproblems P1, P2, P3, ..., Pk
        return (
```

```
            Combine (
                DivideAndConquer( P1 ),
                DivideAndConquer( P2 ),
                ...
                DivideAndConquer( Pk )
            )
        )
}
```

Example: Given array nums of size n, return the majority element. The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

If we know the majority element in the left and right halves of an array, we can determine which is the global majority element in linear time.

Approach: If we know the majority element in the left and right halves, we can find the global majority element. Recurse on left and right halves of the array. The array on which the work is being performed can be denoted by lo and hi. If the left and right halves have the same majority element, then for that slice of the array, the majority element is the one obtained from left and right halves. And if they don't, we count the occurrences of the majority elements of the left and right halves to obtain which half's answer is globally correct. The overall answer for the array is thus the majority element between indices 0 and n.

```
majorityElement(int[] nums) {
    return majorityElementRec(nums, 0, nums.length-1)
 }

countTheMajorityElementInRange(nums, num,  lo, hi) {
    count = 0
    i = lo
    while i < hi
       if (nums[i] == num)
          count += 1
```

```
        i += 1

    return count
}

majorityElementRec(nums, lo, hi) {
    //    base case; the only element in an array of size 1 is the majority element
    if (lo == hi) {
       return nums[lo]
    }

    //   recurse on left and right halves of this slice.
    mid = (hi-lo)/2 + lo
    leftAns= majorityElementRec(nums, lo, mid)
    rightAns = majorityElementRec(nums, mid+1, hi)

    //   if the two halves agree on the majority element, return it.
    if leftAns equals rightAns
         return leftAns


    //   otherwise, count each element and return the "winner".
    leftCount = countTheMajorityElementInRange(nums, leftAns, lo, hi)
    rightCount =countTheMajorityElementInRange(nums, rightAns, lo, hi)

    if(leftCount > rightCount)
        return leftCount
     return rightCount
}
```

Time Complexity: O(nlogn).

Advantages of Divide and Conquer

- **Solving difficult problems**: Divide and conquer is a powerful method for solving difficult problems. As an example, consider the Tower of Hanoi problem. This requires breaking the problems into subproblems, solving the trivial cases and combining the subproblems to solve the original problem. Dividing the problem into subproblems so that the subproblem can be combined again is a major difficulty in designing a new algorithm. For many such problems divide and conquer provides a simple solution
- **Parallelism: Divide and conquer allows us to solve the problems independently, this allows for execution in multi-processor machines, especially shared memory systems where the communication of data between processes does not need to be planned in advance because different subproblems can be executed on different processors.**
- **Memory Access**: Divide and conquer algorithm naturally tend to make efficient use of memory caches. This is because once a subproblem is small, all its subproblems can be solved within the cache without accessing the slower main memory.

Applications of Divide and Conquer

- Binary Search
- Merge Sort and Quicksort
- Median Finding
- Min and Max finding
- Matrix Multiplication
- Closest Pair problem