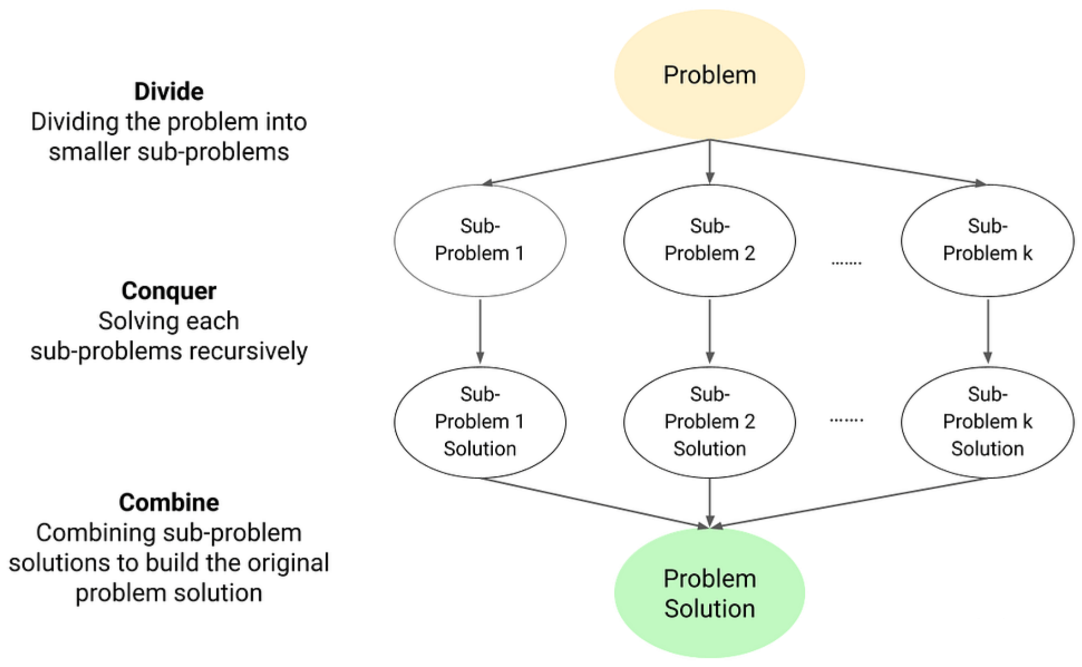
Divide and Conquer Algorithm

As the name itself signifies, the divide and conquer algorithmic approach involves breaking down a given problem into smaller sub-problems, post which these sub-problems are individually solved before being merged again into the output solution.



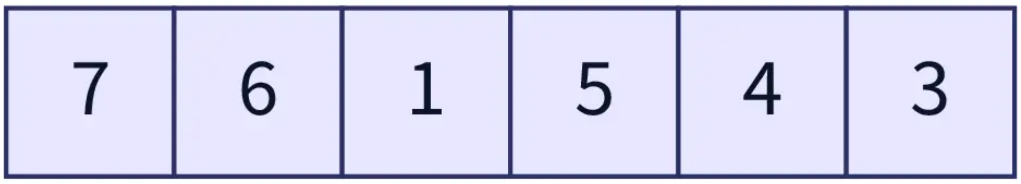
The divide and conquer algorithms work on the principle of atomicity. The notion here is that it is easier to study (and hence, solve) the smallest unit of a problem than the whole problem altogether. Following this principle, in the divide and conquer algorithm, we keep dividing the given problem into smaller problems to the point that these smaller sub-problems can no longer be further divided.

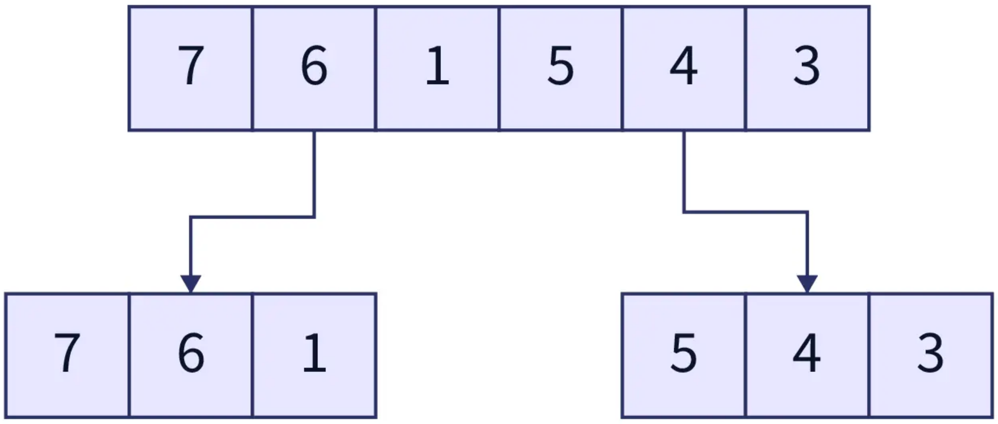
- **Divide :**This step constitutes dividing the problem at hand into smaller, atomic problems. An inherent rule here is that these atoms must be representative of the original problem, which simply implies that these sub-problems have to be a part of the original problem. The divide component of the divide and conquer algorithm follows a recursive approach for breaking down the problem until none of the sub-problems are further divisible. The output of the algorithm, after this step, are atoms of the original problem that can now be worked upon for deriving a solution.
- **Conquer :**

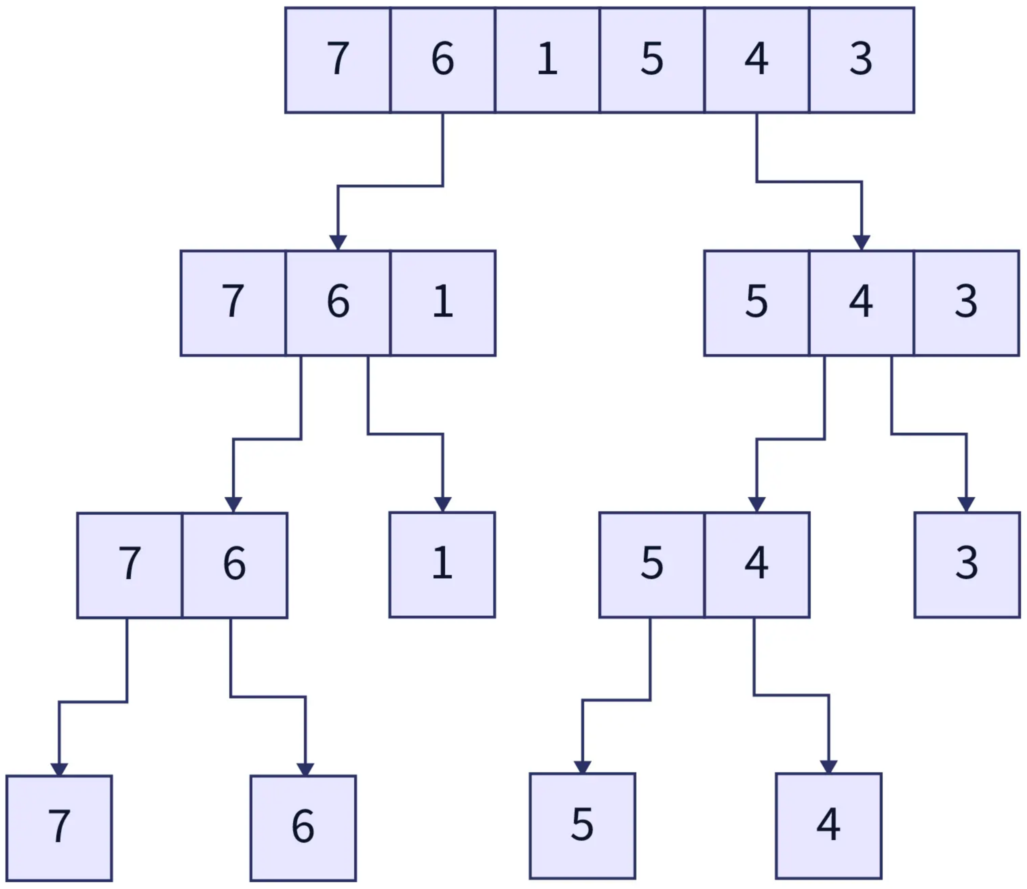
  This is the intermediary step within the divide and conquer problem-solving approach, where in theory, all the individual atomic sub-problems are solved and their solutions are obtained. However in practice, generally the original problem has already been broken down in the last stage (i.e., the dividing stage) to a level that the sub-problems are considered “solved” on their own.

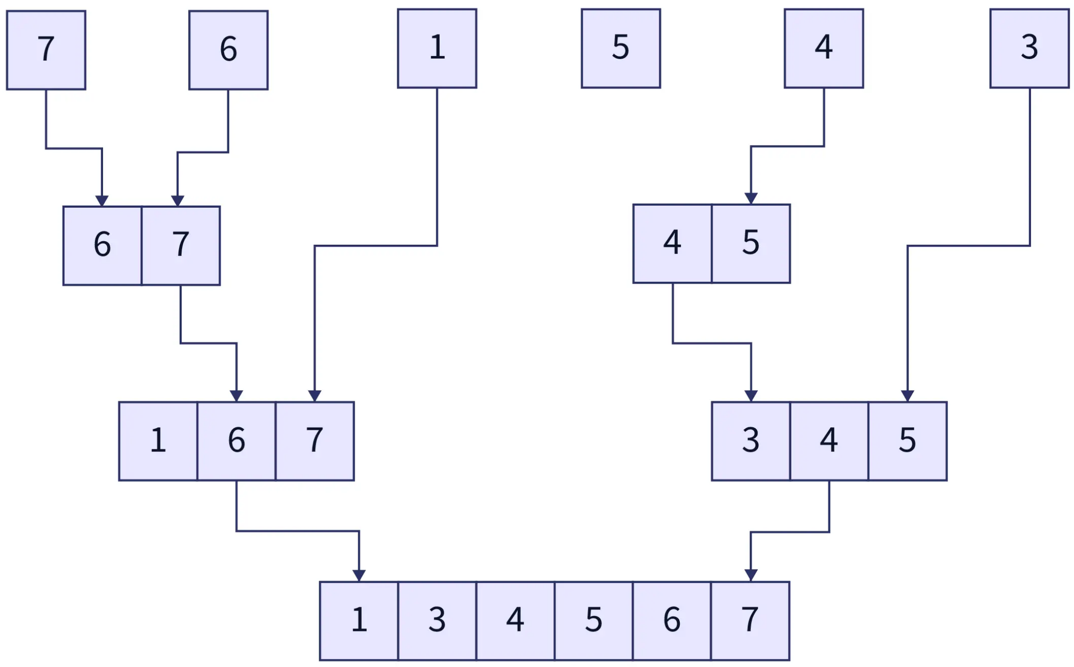
- **Merge :**

  Merge constitutes the final stage of the divide and conquers algorithm, where the individual solutions obtained from the previous stage of the algorithm are recursively merged till the solution of the original problem is formulated. The output derived after this stage is the required solution for the problem.






7 6 1 5 4 3
7 6 1
5 4 3
7 6
1
5 4
3
7
6
5
4



- **Divide operation :** The divide operation is a constant time operation (O(1)) regardless of the size of the subarray since we are just finding the
- middle index of the subarray (mid=(start+end)/2) and thereafter dividing the subarray into two parts based on this middle index.
- During the entire divide operation, we keep on dividing the initial array into 2 subarrays at each level, till we get subarrays of size 1. In total, there are going to be approximate log2n levels of division operations.
- **Conquer operation :** As we discussed earlier, upon breaking down the initial array to sub-arrays of size 1, these sub-arrays get sorted by default. Hence, the conquer operation has a constant time complexity of O(1).
- **Merge operation :** The merging operation at each level, where we sort and merge 2 subarrays at a time recursively, is a linear time operation, i.e., O(n). As we discussed earlier, since there are log2n levels in the division process, the merge step, on average, will take around n.log2n steps.

Application of Divide & Conquer Algorithms

- **Binary Search :**

  Binary search is a search-based algorithm that follows the divide and conquers approach and can be used for searching a target element within a sorted array. While searching for an element within an array, binary search divides the array into two halves at each step, discarding one of the halves based on the value of the middle element. The algorithm can search for the target element with an O(logn) time complexity, as compared to the O(n) complexity in linear search.

- **Merge Sort :**

  Earlier, we saw the working of merge sort, which is a divide and conquer-based sorting algorithm, with the worst-case time complexity of O(nlog2n).

- **Quick Sort :**

  Quick sort, as we saw earlier, is another divide-and-conquer-based sorting algorithm that is often regarded as one of the fastest sorting algorithms, especially for large array sizes.

- **Strassen's Matrix Multiplication :**

  While dealing with matrix multiplication for large matrix sizes, Strassen's matrix multiplication algorithm tends to be significantly faster as compared to the standard matrix multiplication algorithm.

- **Karatsuba Algorithm :**

  The Karatsuba algorithm is a highly efficient algorithm for the multiplication of n-bit numbers. It is a divide and conquers algorithm that performs the multiplication operation with a time complexity of O($n^{1.59}$) as compared to the O(n^2) of the brute force approach.

Find the number of rotations in a circularly sorted array

Given a circularly sorted integer array, find the total number of times the array is rotated. Assume there are no duplicates in the array, and the rotation is in the anti-clockwise direction.

```
Input:  nums = [8, 9, 10,0,1, 2, 5, 6]
Output: The array is rotated 3 times


Input:  nums = [2, 5, 6, 8, 9, 10]
Output: The array is rotated 0 times
```

```
package Divide_Conquer;

public class Circularly_Sorted_Array {
  // Function to find the total number of times the array is rotated
  public static int findRotationCount(int[] nums) {
    // search space is nums[left…right]
    int left = 0;
    int right = nums.length - 1;

    // loop till the search space is exhausted
    while (left <= right) {
      // if the search space is already sorted, we have
      // found the minimum element (at index `left`)
      if (nums[left] <= nums[right]) {
        return left;
      }

      int mid = (left + right) / 2;
      // find the next and previous element of the `mid` element
      // (in a circular manner)
      int next = (mid + 1) % nums.length;
      int prev = (mid - 1 + nums.length) % nums.length;

      // if the `mid` element is less than both its next and previous
      // neighbor, it is the array's minimum element

      if (nums[mid] <= nums[next] && nums[mid] <= nums[prev]) {
        return mid;
      }

      // if nums[mid…right] is sorted, and `mid` is not the minimum element,
      // then the pivot element cannot be present in nums[mid…right],
      // discard nums[mid…right] and search in the left half

      else if (nums[mid] <= nums[right]) {
        right = mid - 1;
      }

      // if nums[left…mid] is sorted, then the pivot element cannot be present
      // in it; discard nums[left…mid] and search in the right half

      else if (nums[mid] >= nums[left]) {
        left = mid + 1;
      }
    }

    // invalid input
    return -1;
  }

  public static void main(String[] args) {
    int[] nums = { 8, 9, 10,2, 5, 6 };

    System.out.println("Array is rotated " + findRotationCount(nums) + " times");
  }
}
```

Find the first or last occurrence of a given number in a sorted array

Given a sorted integer array, find the index of a given number’s first or last occurrence. If the element is not present in the array, report that as well.

```
Input:

nums = [2, 5, 5, 5, 6, 6, 8, 9, 9, 9]
target = 5

Output:

The first occurrence of element 5 is located at index 1
The last occurrence of element 5 is located at index 3


Input:

nums = [2, 5, 5, 5, 6, 6, 8, 9, 9, 9]
target = 4

Output:

Element not found in the array
```

```
package Divide_Conquer;

public class Occurrence {
  // Function to find the first occurrence of a given number
  // in a sorted integer array
  public static int findFirstOccurrence(int[] nums, int target) {
    // search space is nums[left…right]
    int left = 0;
    int right = nums.length - 1;

    // initialize the result by -1
    int result = -1;

    // loop till the search space is exhausted
    while (left <= right) {
      // find the mid-value in the search space and compares it with the target
      int mid = (left + right) / 2;

      // if the target is located, update the result and
      // search towards the left (lower indices)
      if (target == nums[mid]) {
        result = mid;
        right = mid - 1;
      }

      // if the target is less than the middle element, discard the right half
      else if (target < nums[mid]) {
        right = mid - 1;
      }

      // if the target is more than the middle element, discard the left half
      else {
        left = mid + 1;
      }
    }

    // return the leftmost index, or -1 if the element is not found
    return result;
  }

  // Function to find the last occurrence of a given number
  // in a sorted integer array
  public static int findLastOccurrence(int[] nums, int target) {
    // search space is nums[left…right]
    int left = 0;
    int right = nums.length - 1;

    // initialize the result by -1
    int result = -1;

    // loop till the search space is exhausted
    while (left <= right) {
      // find the mid-value in the search space and compares it with the target
      int mid = (left + right) / 2;

      // if the target is located, update the result and
      // search towards the right (higher indices)
      if (target == nums[mid]) {
        result = mid;
        left = mid + 1;
      }

      // if the target is less than the middle element, discard the right half
      else if (target < nums[mid]) {
        right = mid - 1;
      }

      // if the target is more than the middle element, discard the left half
      else {
        left = mid + 1;
      }
    }

    // return the leftmost index, or -1 if the element is not found
    return result;
  }

  public static void main(String[] args) {
    int[] nums = { 2, 5, 5, 5, 6, 6, 8, 9, 9, 9 };
    int target = 5;

    int index = findFirstOccurrence(nums, target);

    if (index != -1) {
      System.out.println("The first occurrence of element " + target + " is located at index " + index);
    } else {
      System.out.println("Element not found in the array");
    }

    index = findLastOccurrence(nums, target);

    if (index != -1) {
      System.out.println("The last occurrence of element " + target + " is located at index " + index);
    } else {
      System.out.println("Element not found in the array");
    }
  }
}
```

Exponential search

Given a sorted array of `n` integers and a target value, determine if the target exists in the array or not in logarithmic time. If the target exists in the array, return the index
 of it.

```
Input: A[] = [2, 3, 5, 7, 9]
target = 7

Output: Element found at index 3


Input: A[] = [1, 4, 5, 8, 9]
target = 2

Output: Element not found
```

```
package Backtracking;

public class Exponential {
  // Binary search algorithm to return the position of key `x` in
  // subarray A[left…right]
  private static int binarySearch(int[] A, int left, int right, int x) {
    // base condition (search space is exhausted)
    if (left > right) {
      return -1;
    }

    // find the mid-value in the search space and
    // compares it with the key

    int mid = (left + right) / 2;
    // overflow can happen. Use
    // int mid = left + (right - left) / 2;

    // base condition (a key is found)
    if (x == A[mid]) {
      return mid;
    }

    // discard all elements in the right search space,
    // including the middle element
    else if (x < A[mid]) {
      return binarySearch(A, left, mid - 1, x);
    }

    // discard all elements in the left search space,
    // including the middle element
    else {
      return binarySearch(A, mid + 1, right, x);
    }
  }

  // Returns the position of key `x` in a given array `A` of length `n`
  public static int exponentialSearch(int[] A, int x) {
    // base case
    if (A == null || A.length == 0) {
      return -1;
    }

    int bound = 1;

    // find the range in which key `x` would reside
    while (bound < A.length && A[bound] < x) {
      bound *= 2; // calculate the next power of 2
    }

    // call binary search on A[bound/2 … min(bound, n-1)]
    return binarySearch(A, bound / 2, Integer.min(bound, A.length - 1), x);
  }

  // Exponential search algorithm
  public static void main(String[] args) {
    int[] A = { 2, 5, 6, 8, 9, 10 };
    int key = 9;

    int index = exponentialSearch(A, key);

    if (index != -1) {
      System.out.println("Element found at index " + index);
    } else {
      System.out.println("Element not found in the array");
    }
  }
}
```

Check if a number is a perfect square

Given a positive number, check if it is a perfect square without using any built-in library function. A perfect square is a number that is the square of an integer.

```
Input: n = 25
Output: true
Explanation: 25 is a perfect square since it can be written as 5×5.


Input: n = 20
Output: false
Explanation: 20 is not the product of an integer with itself.
```

Using Perfect Square Property

Every perfect square satisfies the property that it is the sum of the odd numbers starting with one.

```
1 = 1
4 = 1 + 3
9 = 1 + 3 + 5
16 = 1 + 3 + 5 + 7
25 = 1 + 3 + 5 + 7 + 9
…
n = 1 + 3 + 5 + … + (2*n-1)
```

```
class Main {
    public static boolean isPerfectSquare(int n)
    {
        // start with the odd number 1
        int odd = 1;

        // loop until the value is 0 or negative
        while (n > 0)
        {
            // subtract the next odd number from `n`
            n = n - odd;

            // take next odd number for the next iteration
            odd = odd + 2;
        }

        // only a perfect square will reach a value of 0
        return n == 0;
    }

    public static void main(String[] args) {
        int n = 25;
        System.out.println(isPerfectSquare(n));
    }
}
```

2. Using Binary Search

We can improve the time complexity of the solution to O(log(n)) by using a binary search algorithm. The idea is to start with the search space `[low, high] = [0, n]` and try to find a `mid` value that satisfies the condition `mid × mid == n`. Based on a comparison result between `n` and `mid × mid`, the procedure will increment `low`, decrement `high`, or return `true` if `n` is the product of `mid` with itself.

```
class Main {
    public static boolean isPerfectSquare(int n)
    {
        // set the search space to [0, n].
        int low = 0, high = n;

        while (low <= high)
        {
            // calculate mid
            int mid = low + ((high - low)/2);

            // if `mid×mid` is equal to `n`, return true
            if (mid * mid == n) {
                return true;
            }
            // if `mid×mid` is less than `n`, update `low` to `mid + 1`
            if (mid * mid < n) {
                low = mid + 1;
            }
            // otherwise, update `high` to `mid - 1`
            else {
                high = mid - 1;
            }
        }

        // we reach this point if the number is not a perfect square
        return false;
    }

    public static void main(String[] args) {
        int n = 25;
        System.out.println(isPerfectSquare(n));
    }
}
```

Using Newton’s Method

Finally, we can use Newton’s method to compute square roots. The mathematical logic behind Newton’s method is outside the scope of this article.

```
class Main {
    public static boolean isPerfectSquare(int n)
    {
        int i = n;
        while (i * i > n) {
            i = (i + n / i) / 2;
        }
        return i * i == n;
    }

    public static void main(String[] args) {
        int n = 25;
        System.out.println(isPerfectSquare(n));
    }
}
```