Learning Objectives

Learners will be able to…

- **Define divide and conquer**
- **Implement divide and conquer algorithms.**
- **Analyze the time and space complexities associated with these divide and conquer algorithms.**

Divide and Conquer

Merge Sort Revisited

In the last assignment, we talked about the merge sort and why it is suited to a recursive implementation. In short, the ability to create subproblems out of a larger problem means recursion is a good choice for the algorithm. Below is the `sort` method from a merge sort. We see how the left side of the original array is sorted into arrays of only one element. From there, the right side of the original array is divided into arrays of only one element.

```
void sort(int arr[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        sort(arr, left, mid);         //divide array into the
left half
        sort(arr, mid + 1, right);    //divide array into the
right half
        merge(arr, left, mid, right); //combine the sorted
arrays
    }
}
```

Computer scientists refer to this problem solving technique as **divide and conquer**. It should noted that the divide and conquer technique oven includes a third step, one that combines all of the small subproblems into a single solution. We saw how a merge sort uses the `merge` method after all of the dividing has been done to generate the final, sorted array.

An algorithm for a merge sort is similar to an algorithm for factorial. Both of them use recursion to divide the bigger problem into smaller subproblems. $5!$ is defined as $5 \times 4!$, and $4!$ is defined as $4 \times 3!$ and so on. Both of the recursive algorithms for a merge sort and factorial rely on divide and conquer.

```
public static int factorial(int n) {
  if (n == 1) {
    return 1;
  } else {
    //calculate the factorial of a smaller number
    return n * factorial(n - 1);
  }
}
```

Because of this, people often have the incorrect belief that the terms recursion and divide and conquer can be used interchangeably. While these two terms are similar, there is an important difference. Divide and

conquer is the problem solving technique, while recursion is the way in which divide and conquer is expressed as code.

When to Use Divide and Conquer

The Divide and Conquer algorithmic design paradigm is based on multi-branched recursion, where a problem is solved by:

1. **Dividing it into smaller sub-problems.**
2. **Conquering the sub-problems recursively**.
3. **Combining these solutions to craft the answer to the original problem**.

This strategy leans on the idea that a complex problem can often be solved more easily by splitting it into smaller, easier, and more manageable parts. The characteristics of divide and conquer are:

1. **Self-similarity**: Each sub-problem is essentially a smaller version of the general problem.
2. **Independence**: The sub-problems are solved independently.
3. **Recursive Nature**: The approach repeatedly breaks down a problem into sub-problems.

Binary Search

Iterative Binary Search

Binary Search is a quintessential example of the divide and conquer approach in action. Given a sorted list and a target value, binary search aims to find the target's position in the list. Instead of searching sequentially, binary search divides the list in half repeatedly until the target value is found or the range is empty.

Here's a simplified outline of how Binary Search works:

1. Compare the target value to the middle element.
2. If they are not equal, the half in which the target cannot lie is eliminated.
3. Continue the search on the remaining half.

This continuous halving is a perfect demonstration of the divide and conquer strategy. It should be noted, however, that you do not have to use recursion to implement a divide and conquer algorithm. The code sample below uses a while loop instead of recursion. **Important**, do not try to run the code below in the IDE. It will cause errors.

```
public static int binarySearch(int[] array, int target) {
    int left = 0;
    int right = array.length - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (array[mid] == target)
            return mid;

        if (array[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}
```

Recursive Binary Search

Let's implement a recursive solution for a binary search. The base case is when the array is empty (`right < left`) or when you have found the search target (`target == array[mid]`). In all other cases, we are going to recurse. However, we want to ignore the half of the list that we know cannot

contain the search target. When you call `binarySearch` with `left` and `mid - 1`, you are ignoring the right half of the array. When you call the function with `mid + 1` and `right`, you are ignoring the left half.

```
    public static int binarySearch(int[] array, int target, int
        left, int right) {
        int mid = left + (right - left) / 2;

        if (right < left)
            return -1;

        if (target == array[mid])
            return mid;
        else if (target < array[mid])
            return binarySearch(array, target, left, mid - 1);
        else
            return binarySearch(array, target, mid + 1, right);
    }
```

This approach is significantly faster than a linear search, especially for larger datasets. For a list with `n` elements, Binary Search can find an element in about $O(logn)$ steps, whereas linear search would take, on average, $O(n/2)$ steps.

Binary Search Analysis

Time Complexity

The primary reason binary search is efficient lies in its mechanism: With each comparison, it halves the search space. Let's break this down:

- **First iteration**: We have n elements.
- **Second iteration**: We have $n/2$ elements.
- **Third iteration**: We have $n/4$ elements.

... and so on. We can see this process when we ran the following:

```
if (target < array[mid])
```

To determine how many steps it would take to get down to 1 element (since $$. This tells us that the binary search has a time complexity of $O(logn)$.

This logarithmic time complexity ensures that even for large datasets, the number of steps to find an element (or determine its absence) grows very slowly.

Space Complexity

Binary search requires only a constant amount of additional memory to perform, which includes storage for a single variable: `mid`. Therefore, regardless of how large our input array is, the space we need (apart from the array itself) remains the same. This translates to $O(1)$.

That means that a binary search has a constant space complexity, even when you search large arrays.

Best, Worst, and Average Cases

- **Best Case**: The best-case scenario for binary search occurs when the target element is right in the middle of the sorted list, meaning we have found the element in just one comparison. Best case time complexity is $O(1)$.
- **Worst Case**: The worst-case scenario happens when the target element is either one of the endpoints of the list or is not in the list at all. In such cases, the search would make the maximum number of comparisons, i.e., $O(logn)$ comparisons. Thus, the worst-case time complexity is $ O(log n)$.

- **Average Case**: On average, binary search will have to make multiple comparisons before locating the desired index or concluding the absence of an element. The average case, too, will be bound by $O(logn)$.

info

In comparison to linear search, which operates in linear time, binary search can vastly outperform it, especially as the dataset grows. However, it's essential to note that binary search **requires** a sorted list, and the time taken to sort a list (if it is not already sorted) should be considered when assessing the overall efficiency for specific applications.

Max Value

Finding the Max Value

Let's practice the divide and conquer technique once more. This time, we are going to find the element with the largest value in an array of integers. Create the method `maxValue` that takes an array of integers, and integer representing the left-most element in the array, and another integer representing the right-most element in an array.

```
    public static int maxValue(int[] array, int left, int right)
        {

    }
```

Our base case is when our array has a length of 2. We determine this by subtracting `right` from `left`. If the difference is `1` or less, than the array has two elements. If the element at index `left` is less than the element at index `right`, then return the element at index `left`. Otherwise, return the element at index `right`.

```
    public static int maxValue(int[] array, int left, int right)
        {

        if (right - left <= 1) {
            return array[left] > array[right] ? array[left] :
        array[right];
        }
    }
```

For the recursive case, we need to create a few variables. Calculate the mid-point of the array and assign it to `mid`. Create `leftMax` and assign it the value of recursively calling `maxValue` with the left-half of the array (the elements from index `left` to index `mid`). Then create `rightMax` and assign it the value of recursively calling `maxValue` with the right-half of the array (the elements from `mid + 1` to index `right`). Finally return the greater value between `leftMax` and `rightMax`.

```
    public static int maxValue(int[] array, int left, int right)
        {

        if (right - left <= 1) {
            return array[left] > array[right] ? array[left] :
        array[right];
        } else {
            int mid = left + (right - left) / 2;
            int leftMax =  maxValue(array, left, mid);
            int rightMax = maxValue(array, mid +1, right);
            return leftMax > rightMax ? leftMax : rightMax;
        }
    }
```

This algorithm displays the divide and conquer technique because is repeatedly divides the original array into smaller and smaller arrays. Once these arrays reach a small enough size, then the max value is calculated for each one. These values are then compared to find the largest value in the array.

challenge

Try This Variation

Create the `minValue` method that takes an array of integers and returns the element with the smallest value. Use recursion and the divide and conquer technique in your algorithm.

▼ Solution

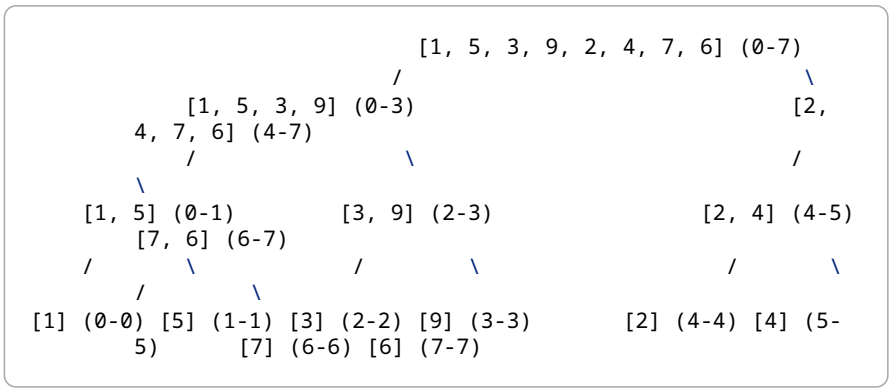
```
    public static int minValue(int[] array, int left, int
        right) {

        if (right - left <= 1) {
            return array[left] < array[right] ? array[left]
        : array[right];
        } else {
            int mid = left + (right - left) / 2;
            int leftMin =  minValue(array, left, mid);
            int rightMin = minValue(array, mid +1, right);
            return leftMin < rightMin ? leftMin : rightMin;
        }
    }
```

Max Value Analysis

In short, the time complexity for all scenarios (worst, best, and average) is $O(n)$, while the space complexity for all these scenarios is $O(logn)$. A more in-depth analysis is provided below. We are going to be referencing a recursion tree. In our code for finding the maximum element in an array, the recursion tree provides a good way to understand the execution flow and complexity. Let's consider an example array of length $n = 8$, with elements `[1, 5, 3, 9, 2, 4, 7, 6]`.

The root of the recursion tree represents the initial call, which covers the entire array from index `0` to `7`. The root then has two children: one representing the left half of the array (from index `0` to `3`) and the other representing the right half of the array (from index `4` to `7`).

Here is how the recursion tree would look like in this example:



Time Complexity:

- The **worst-case** scenario for this algorithm would involve traversing the entire array to find the maximum value. In this case, the function will be called (n) times, where (n) is the size of the array. Each call takes constant time $O(1)$ to execute the comparisons and returns. Therefore, the worst-case time complexity would be $O(n)$.
- The **best-case** time complexity is also $O(n)$. Even if the maximum element is the very first or last element of the array, the algorithm must still traverse the entire array to guarantee that it is indeed the maximum. Just like in the worst-case, each function call takes $O(1)$ time and there are n such function calls.
- The **average-case** scenario would also necessitate going through all the elements of the array to find the maximum element, resulting in a time complexity of $O(n)$.

Space Complexity:

- The **worst-case** space complexity of this algorithm is $O(logn)$. This occurs when the recursion tree is deepest, which would happen when you need to go through all the elements in the array. Each recursive call takes up space on the call stack. The maximum depth of the recursion tree would be $(logn)$, resulting in a worst-case space complexity of $O(logn)$.
- The **best-case** space complexity would still be $O(logn)$. This is because, even if the maximum element is found in the first or last position, you would still make the same number of recursive calls and the depth of the recursive call stack would still be $(logn)$.
- Just like the worst and best cases, the **average-case** space complexity is $O(logn)$. This is because the algorithm must traverse the array in a divide-and-conquer manner, regardless of the actual values in the array. Each level of recursion adds a new layer to the call stack, resulting in a maximum depth of $(logn)$.

Formative Assessment 1

Formative Assessment 2