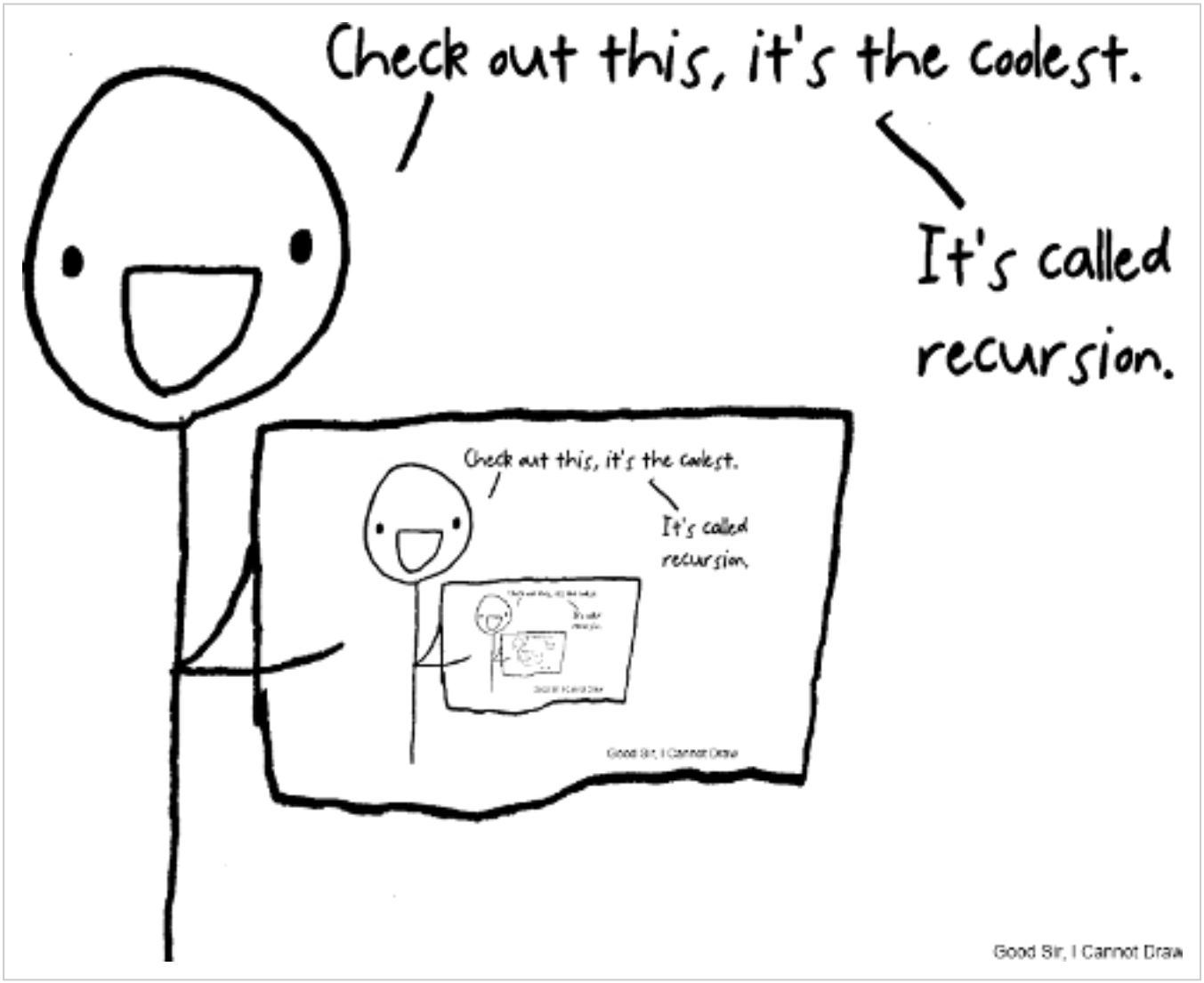
Recurrence

This chapter introduces recursive algorithms and how they can be analyzed.

The word *recurrence* literally means **the fact of occurring again**.



Merge Sort

Merge sort is a typical text-book example of a recursive algorithm. The idea is very simple: We divide the array into two equal parts, sort them recursively, and then combine the two sorted arrays. The base case for recursion occurs when the size of the array reaches a single element. An array consisting of a single element is already sorted.

We'll determine the time complexity of merge sort by unrolling the recursive

calls made by the algorithm to itself, and examining the cost at each level of recursion. The total cost is the sum of individual costs at each level.

The running time for a recursive solution is expressed as a *recurrence equation* (an equation or inequality that describes a function in terms of its own value on smaller inputs). The running time for a recursive algorithm is the solution to the recurrence equation. The recurrence equation for recursive algorithms usually takes on the following form:

Running Time = Cost to divide into n subproblems + n * Cost to solve each of the n problems + Cost to merge all n problems

In the case of merge sort, we divide the given array into two arrays of equal size, i.e. we divide the original problem into sub-problems to be solved recursively.

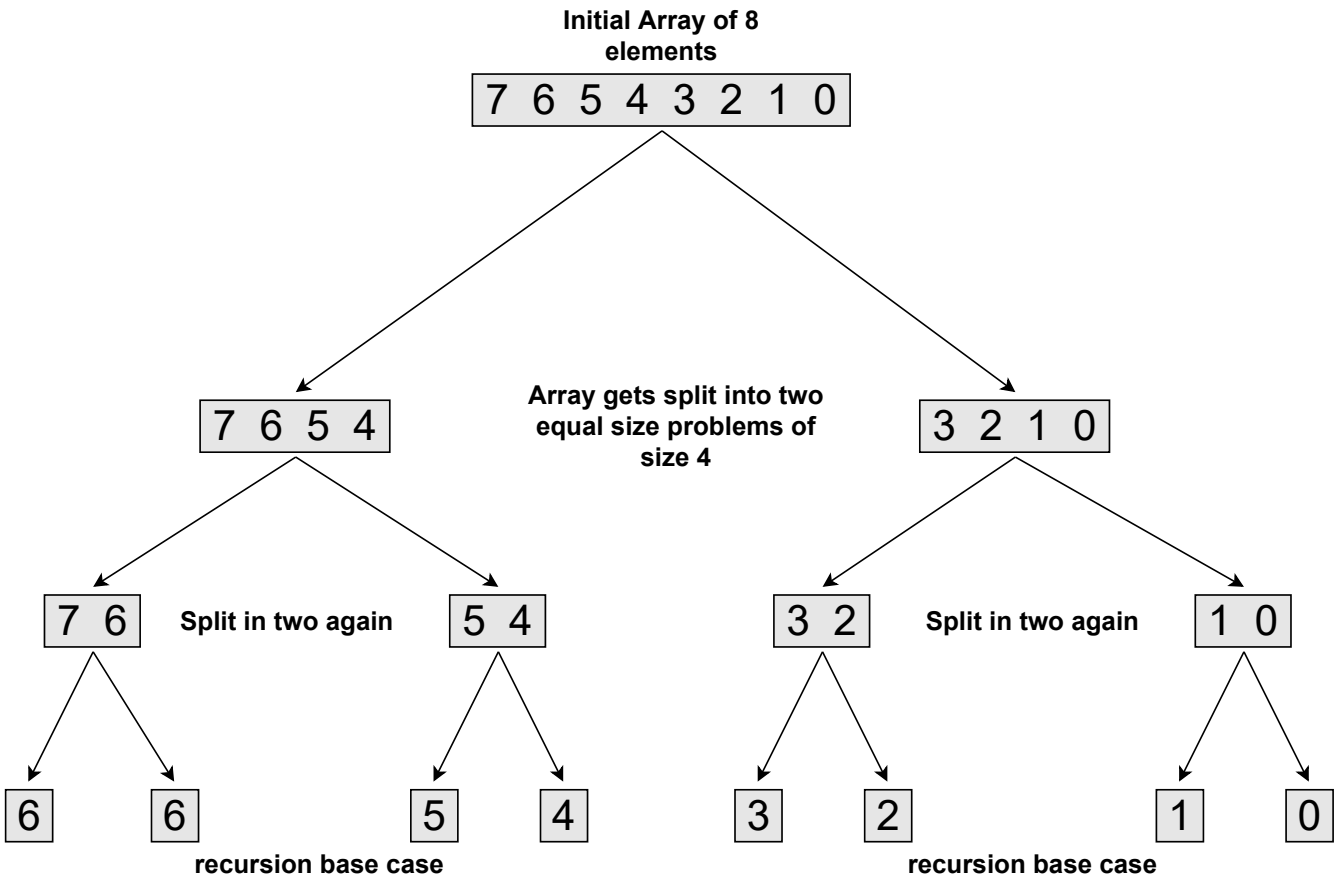
Following is the recurrence equation for merge sort.

Running Time = Cost to divide into 2 unsorted arrays + 2 * Cost to sort half the original array + Cost to merge 2 sorted arrays

> $$T(n) = Cost\ to\ divide\ into\ 2\ unsorted\ arrays + 2 * T(\frac{n}{2}) + Cost\ to\ merge\ 2\ sorted\ arrays when n > 1$$
>
> $$T(n) = O(1)\ \ when\ n = 1$$

Remember the *solution* to the recurrence equation will be the *running time* of the algorithm on an input of size n.

Merge Sort Recursion Tree

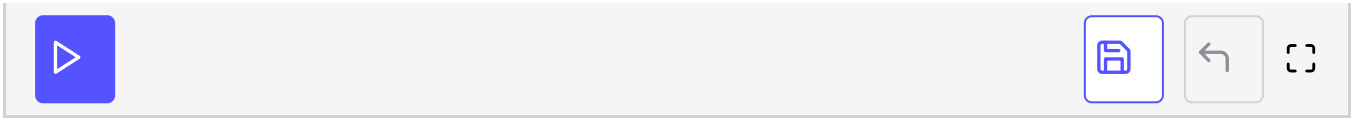


Working of Merge Sort

Merge Sort Implementation

```
class Demonstration {

  private static int[] scratch = new int[10];

  public static void main( String args[] ) {
    int[] input = new int[]{ 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
    printArray(input,"Before: ");
    mergeSort(0, input.length-1, input);
    printArray(input,"After:  ");

  }

  private static void mergeSort(int start, int end, int[] input) {

    if (start == end) {
      return;
    }

    int mid = (start + end) / 2;

    // sort first half
    mergeSort(start, mid, input);

    // sort second half
    mergeSort(mid + 1, end, input);

    // merge the two sorted arrays
```

```
    int i = start;
    int j = mid + 1;
    int k;


    for (k = start; k <= end; k++) {
      scratch[k] = input[k];
    }

    k = start;
    while (k <= end) {

      if (i <= mid && j <= end) {
        input[k] = Math.min(scratch[i], scratch[j]);

        if (input[k] == scratch[i]) {
          i++;
        } else {
          j++;
        }
      } else if (i <= mid && j > end) {
        input[k] = scratch[i];
        i++;
      } else {
        input[k] = scratch[j];
        j++;
      }
      k++;
    }
  }

  private static void printArray(int[] input, String msg) {
    System.out.println();
    System.out.print(msg + " ");
    for (int i = 0; i < input.length; i++)
      System.out.print(" " + input[i] + " ");
    System.out.println();
  }


}
```



Cost to divide

Line-19 is the cost to divide the original problem into two sub-problems, with each one of them as half the size of the original array. The mathematical operation takes place in constant time, and we can say that the cost to divide the problem into 2 subproblems is constant. Let's say it is denoted by d.

Cost to Merge

Line-27 to 55 is the cost to merge two already-sorted arrays. Note the use of a scratch array here. Merge Sort can be implemented in-place but doing so isn't trivial. For now, we'll stick to using an additional array. You'll notice that the

trivial. For now, we'll stick to using an additional array. You'll notice that the merge part has two loops:

1- **for loop line 32 to 34**
2- **while loop line 37 to 55**

Note that both the loops iterate for ***end-start+1***, which is the size of the sub-problem the algorithm is currently working on. The cost of merging two arrays varies with the input size, as follows:

Size of each array = n/2, Cost to merge = $2 * n/2 = n$
Size of each array = n/4, Cost to merge = $2 * n/4 = n/2$
Size of each array = n/8, Cost to merge = $2 * n/8 = n/4$

•

•

Size = 1, Cost = 1 - base case for recursion

Cost to solve 2 subproblems of half the size

The cost to solve each of the two sub-problems (of half the size of the original array) isn't so straightforward, as each of them is expressed as a recursive call to merge sort.

Let's start with the base case: when the problem size becomes a single element, the 1-element array is already sorted and the recursion bottoms out. The cost of solving the base case is constant time, for simplicity, let's assume it's one.