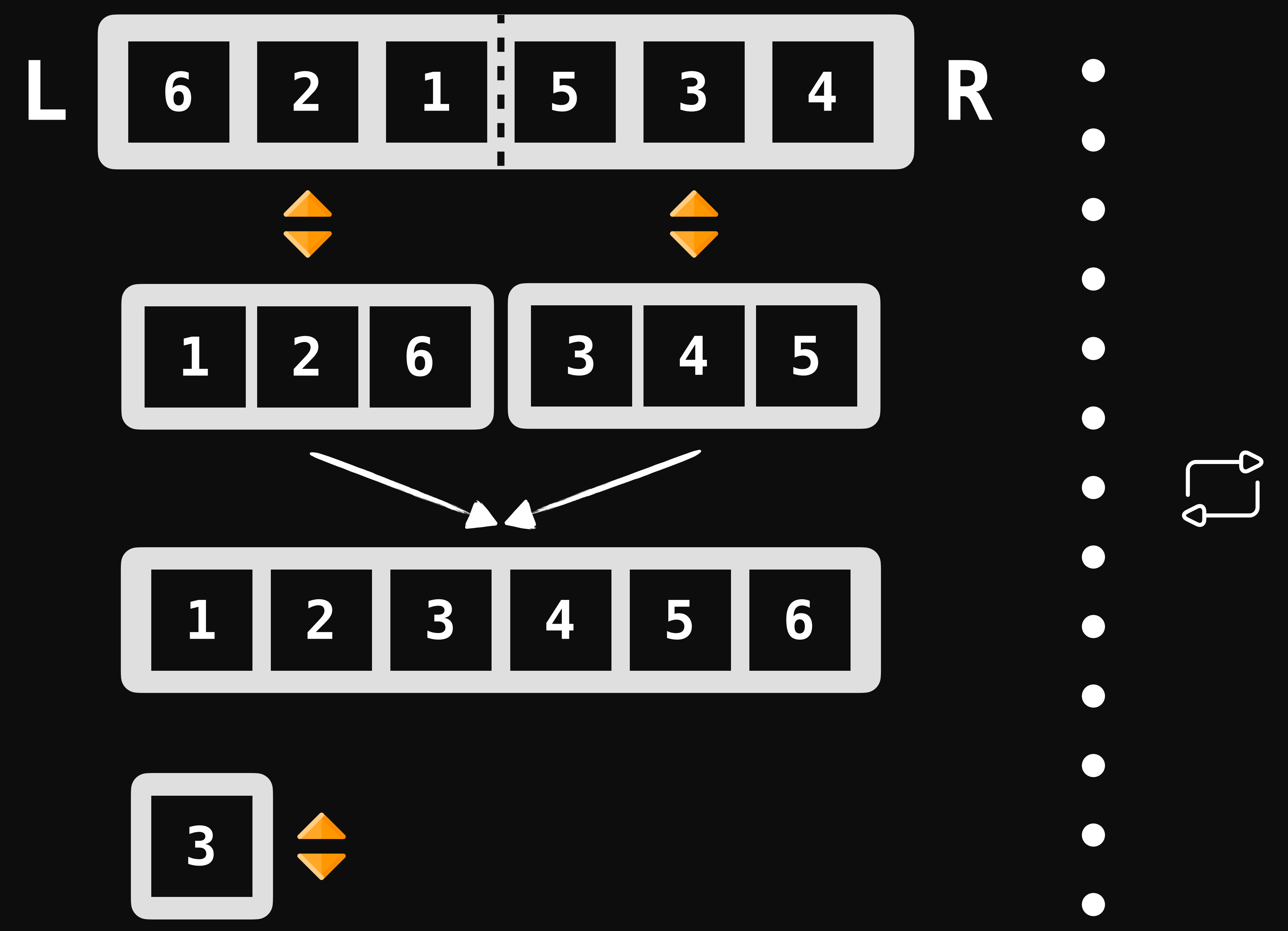
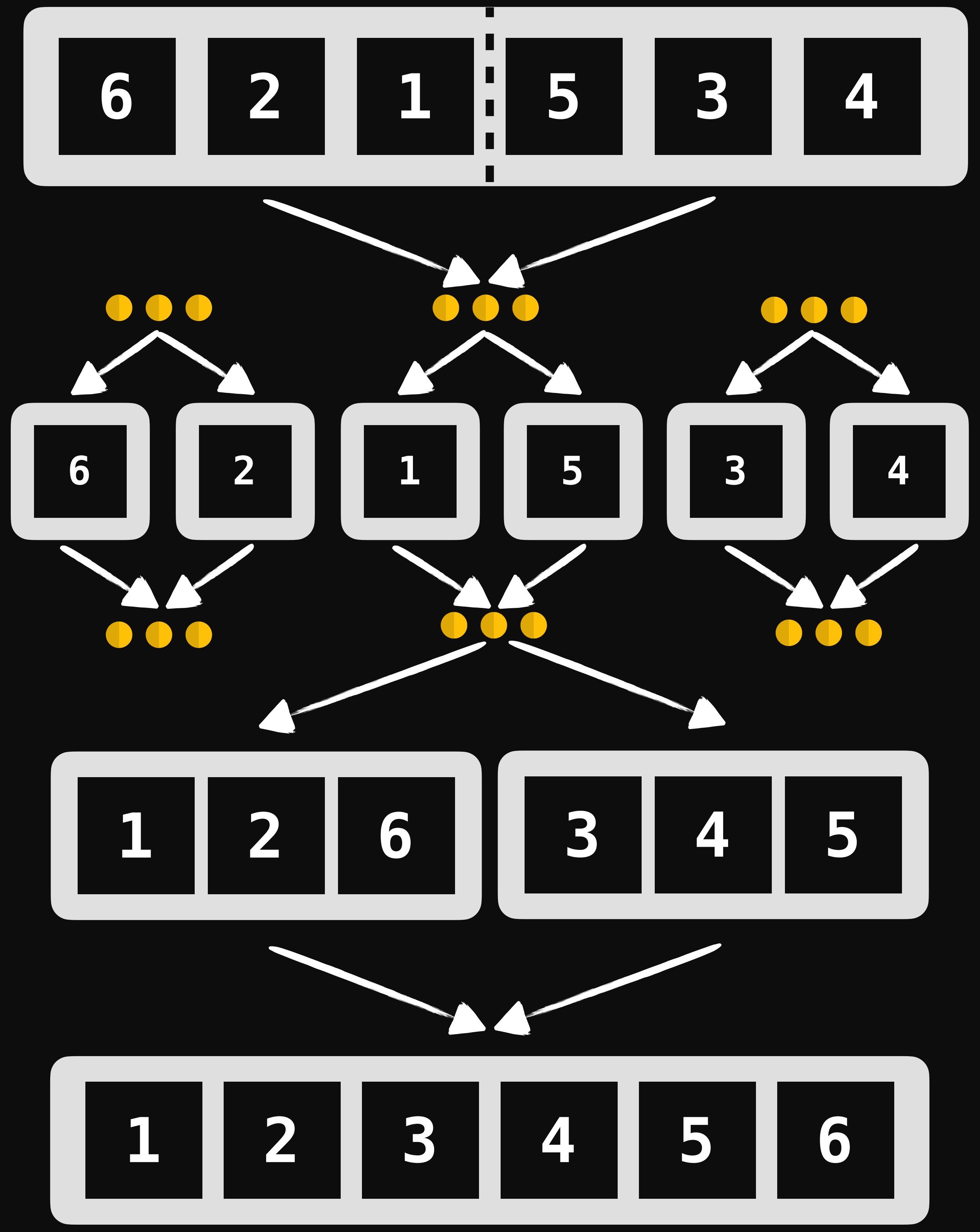
- Find the ***middle*** of the incoming array/slice
- **Sort** the ***LEFT*** side
- **Sort** the ***RIGHT*** side
- **Merge** the ***LEFT*** & ***RIGHT*** side
- Array/Slice of **1 element** is considered **sorted**
- **Repeat** the process using **divide & conquer**

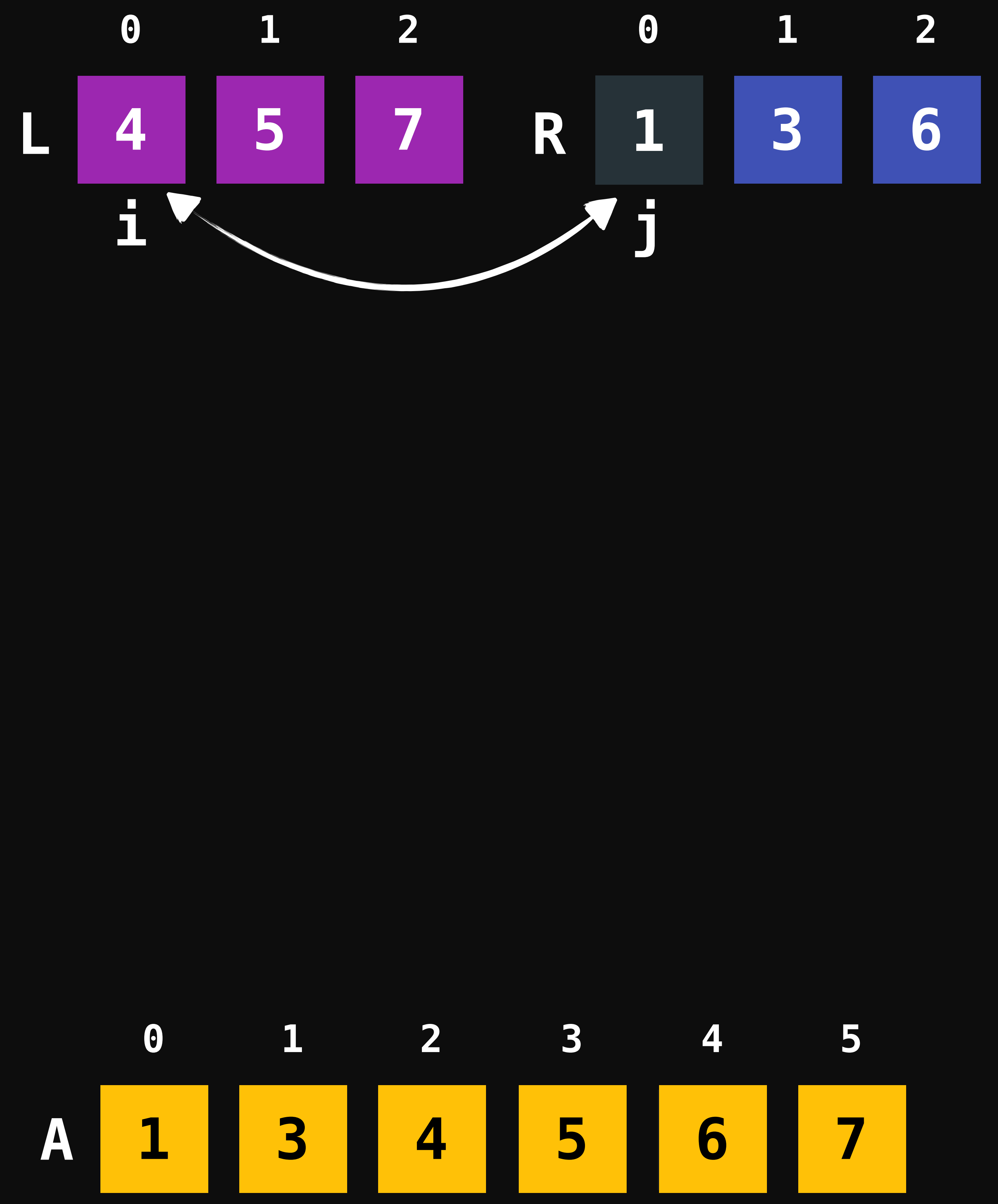


- **Divide** a problem into **smaller problems** of the **same type**
- **Repeat** the process till the problem **cannot be divided**
- **Collect** & **merge** indivisible problems (**solutions**) (**Conquer**)



THE ALGORITHM

- We need a **merge** & a **sort** function
- **Split** the slice in **2** sides
- Call **sort** recursively on the **Left** side
- Call **sort** recursively on the **Right** side
- Call **merge** recursively for **Left** & **Right** side
- **Return** the **sorted merged** slice



0 1 2
L 4 5 7
i
0 1 2
R 1 3 6
j
0 1 2 3 4 5
A 1 3 4 5 6 7

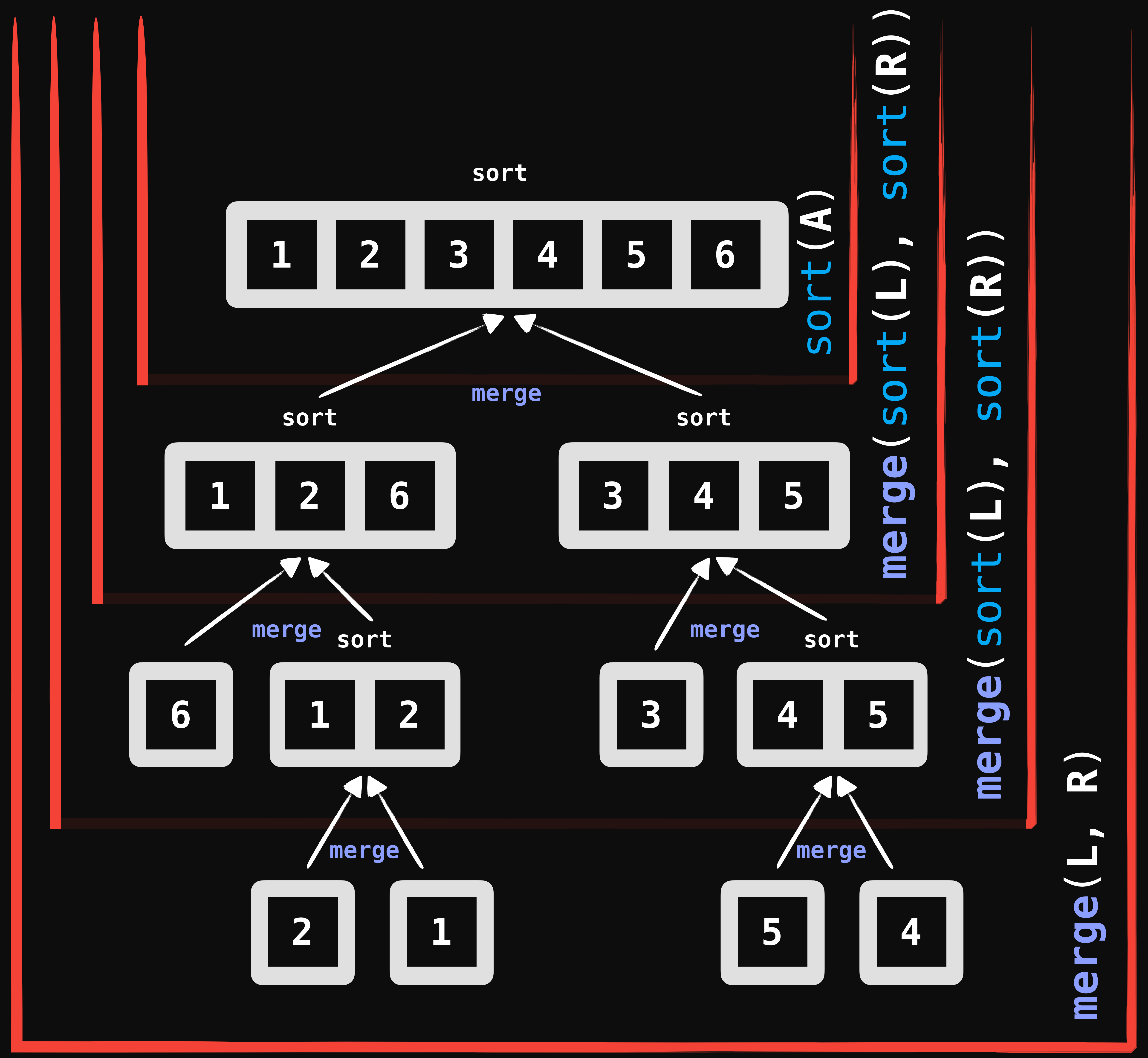
```
A := []int{6,2,1,3,5,4})

sort([]int{6,2,1,3,5,4})
sort([]int{6,2,1})
sort([]int{2,1})

merge([]int{2}, []int{1})
merge([]int{6}, []int{1,2})

sort([]int{3,5,4})
sort([]int{5,4})

merge([]int{5}, []int{4})
merge([]int{3}, []int{4,5})
merge([]int{1,2,6}, []int{3,4,5})
```



```
function merge(L, R) {
  let A[0..L.length+R.length]
  i, j, k = 0, 0, 0

  while i < L.length and j < R.length {
    if L[i] < R[j] {
      A[k] = L[i]; i++
    } else {
      A[k] = R[j]; j++
    }
  }

  while i < L.length {
    A[k] = L[i]; i++; k++
  }
  while j < R.length {
    A[k] = L[j]; j++; k++
  }

  return A
}
```

```
function sort(A) {
  if A.length > 1 {
    n1 = A.length/2
    n2 = A.length-n1
    let L[0..n1] and R[0..n2]

    for i=0 to n1
      L[i] = A[i]
    for j=0 to n2
      R[j] = A[n1+j]

    A = merge(sort(R), sort(L))
  }
  return A
}
```

```
m = A.length/2
L = A[:m]
R = A[m:]
```

IMPROVEMENTS

- Merge sort uses **temporary memory** (L, R slices) (problem for big data sets)
- Merge sort works nicely on **big data sets** & is **stable** & **predictable**
- Insertion sort is **faster** for **small data sets**
- Insertion sort **does not use temporary memory**

INSERTION MERGE SORT

- Use **merge sort** mostly
- **Check data length** before splitting the array/slice
- Use **insertion sort** if the data length is **small** enough

FACTS

- **Merge Sort** is **Efficient** & **Very Fast** for **big data** sets
- **Merge Sort** Time Complexity: O(**n log(n)**)
- **Merge Sort:** Space Complexity: O(**n**)

- **Insertion Sort** is **Efficient** for **small data**
- **Insertion Sort** Best: O(**n**)
- **Insertion Sort** Average/Worst: O(**n*n**)
- **Insertion Sort:** Space Complexity: O(**1**)