Recursion

Pedro Ribeiro

DCC/FCUP

2024/2025

Recursion

Mathematics:

- A formula is said to be **recursive** when it includes references to itself
  Examples:
  - Factorial: $n! = n \times (n-1)!$
  - Fibonacci: $fib(n) = fib(n-1) + fib(n-2)$

Computer Science:

- A function is said to be **recursive** when it includes calls to itself
  - A non-recursive function is **iterative**

- **Recursion** is one of the most used algorithmic techniques
  - Many algorithms can be expressed more easily (and elegantly) with recursion
  - Understanding the recursive formulation will often give you insight on the structure of the problem at hand

- In this lecture we will see several **recursion examples**

Recursion: a first example

- Let's image we have an integer array and that we want to find the **largest number between** positions (indices) *start* and *end*.
- Let's first create a function that return the maximum between two integers: (easy to understand, right?)

```
int max(int a, int b) {
  if (a >= b) return a;
  else return b;
}
```

- The classical **iterative** solution to this problem is to make a cycle between *start* and *end* and store the max until the current position:

```
int maxIt(int v[], int start, int end) {
  int maxSoFar = v[start];
  for (int i=start+1; i<=end; i++)
    maxSoFar = max(maxSoFar, v[i]);
  return maxSoFar;
}
```

Recursion: a first example

- How could we **recursively** define the maximum of an array?

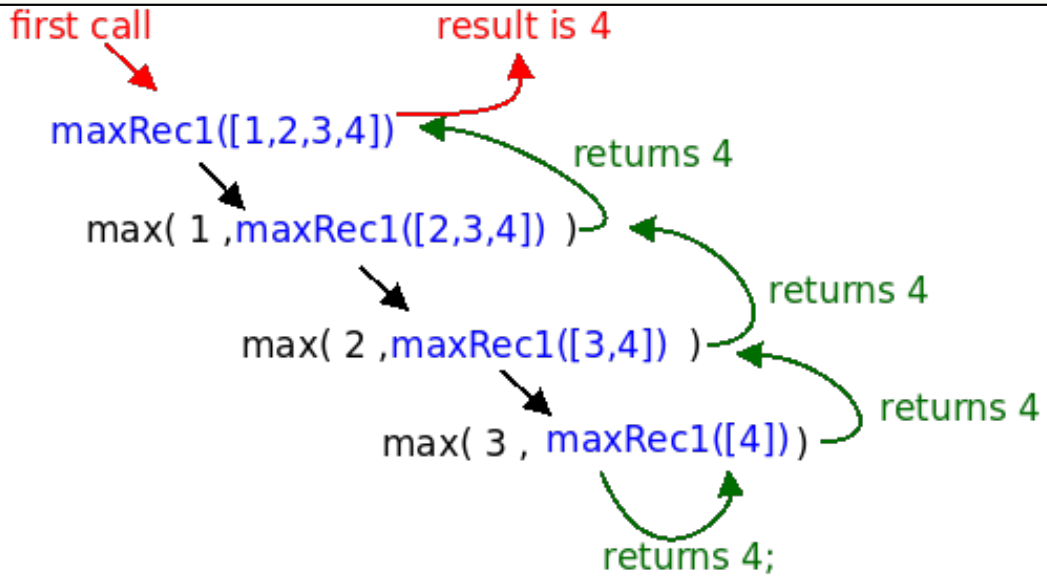
- In any **recursive function** we usually need the following:
  (divide and conquer strategy)
  - **Base Case:** "small case" where we return the result without recursion
  - **Divide** the problem into one or more small cases smaller than the original, closer to the base case
  - **Recursively** call the function for the smaller cases
  - **Combine** the results to obtain the global solution

Note: you can download the code for all the versions of the maximum function given in the slides: `max.c` (source)

Recursion: a first example

- How can we **divide the array** into smaller pieces?
  - One way is to consider the entire array except the first element:
    $max(v)$ = maximum between the 1st element and max(rest of the array v)
- The **base case** is an array of size 1: the maximum is that element

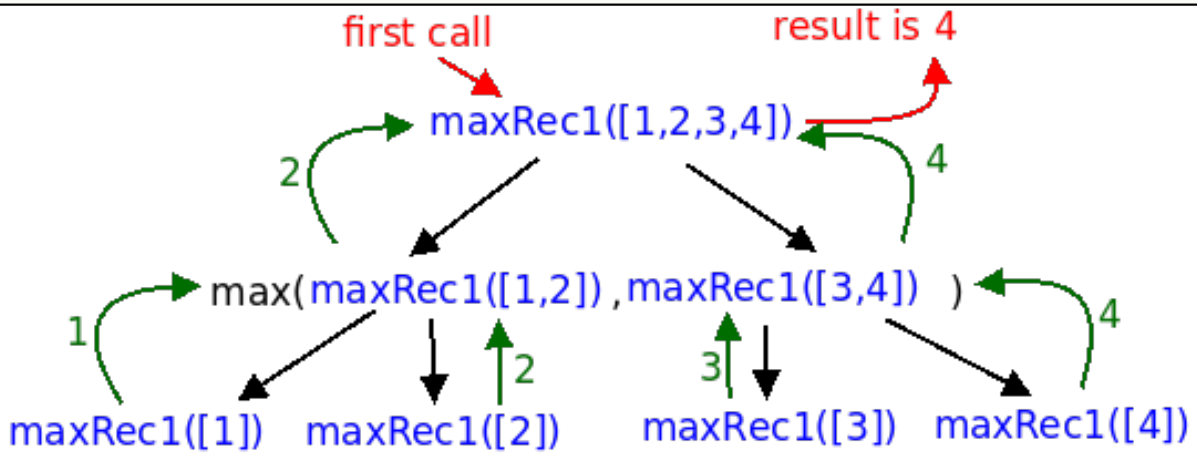
```
int maxRec1(int v[], int start, int end) {
  if (start == end) return v[start];          // base case
  int maxOther = maxRec1(v, start+1, end); // recursive call
  return max(v[start], maxOther);             // combine result
}
```



Recursion: a first example

- How can we **divide the array** into smaller pieces?
  - Another way is to divide the array into two halves:
    $max(v)$ = maximum between max(left half) and max(right half)
- The **base case** is still an array of size 1

```
int maxRec2(int v[], int start, int end) {
  if (start == end) return v[start];     // base case
  int middle = (start + end) / 2;
  int max1 = maxRec2(v, start, middle); // recursive call
  int max2 = maxRec2(v, middle+1, end); // recursive call
  return max(max1, max2);                // combine result
}
```



Recursion: MergeSort

- Let's now see the same pattern, but for **sorting** elements:

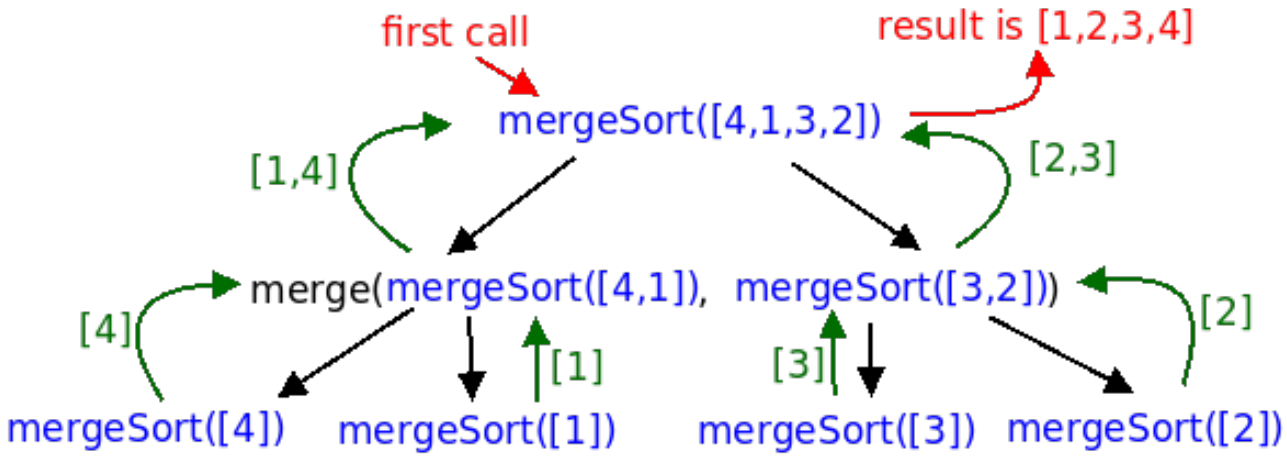
MergeSort

Base Case: if the array is of size 1, than it is already sorted

Divide: divide the initial array into two halves

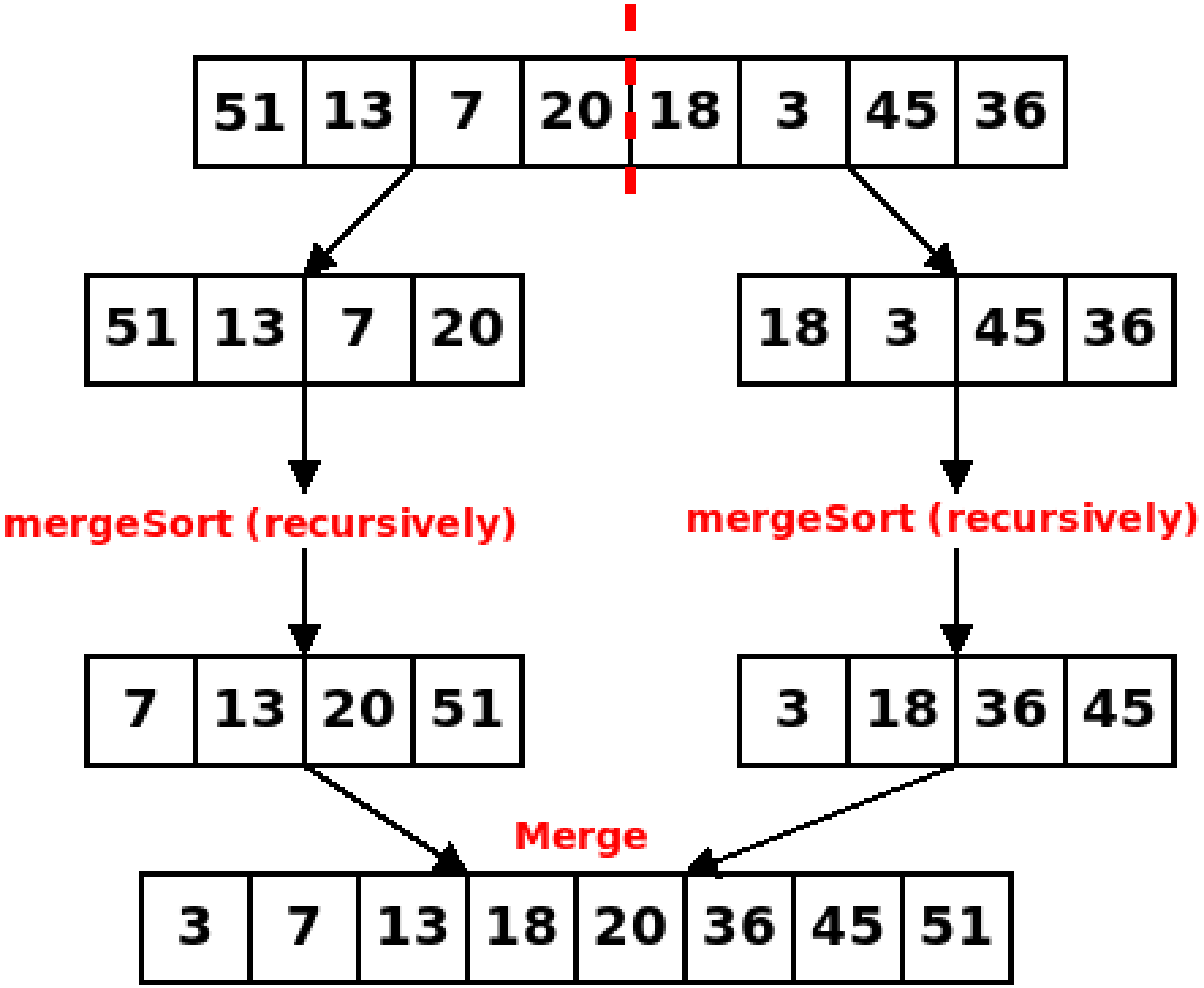
Recursion: sort recursively both halves

Combine: join both sorted halves (*merge*) into a final global sorted array



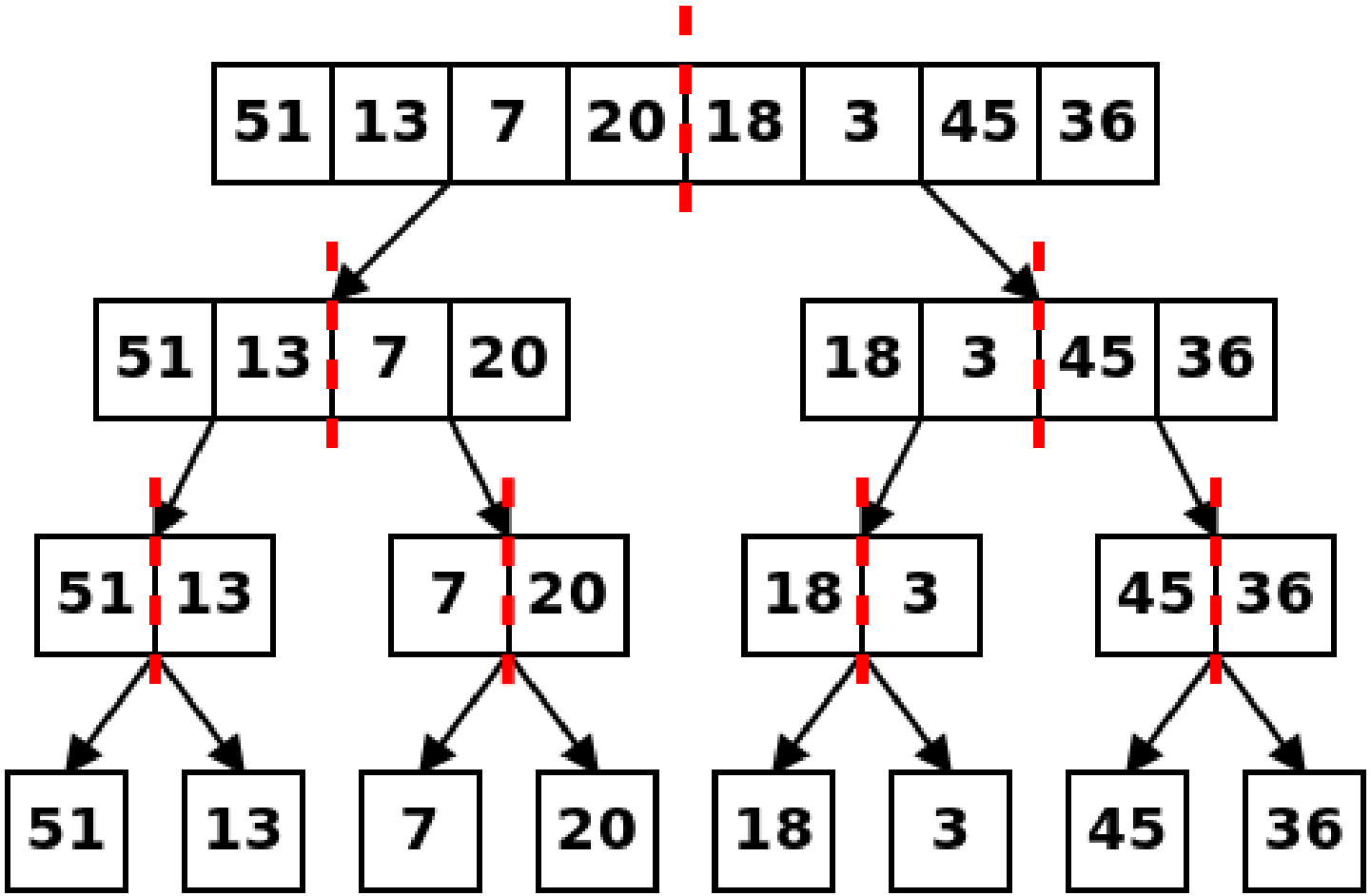
Recursion: MergeSort

What happens from the point of view of the 1st call to the recursive function:



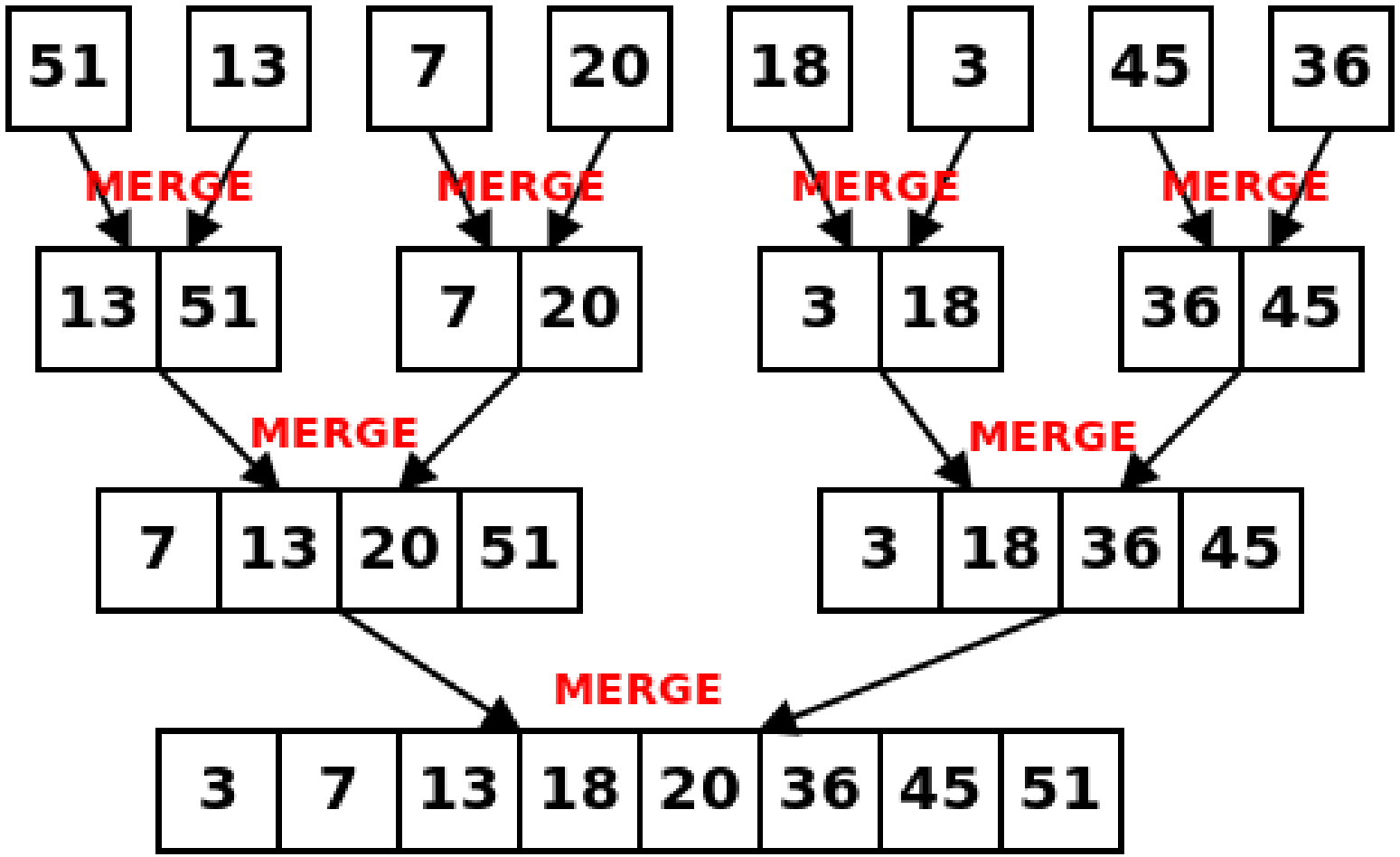
Recursion: MergeSort

Division (and recursion):



Recursion: MergeSort

Combine:



Recursion: MergeSort

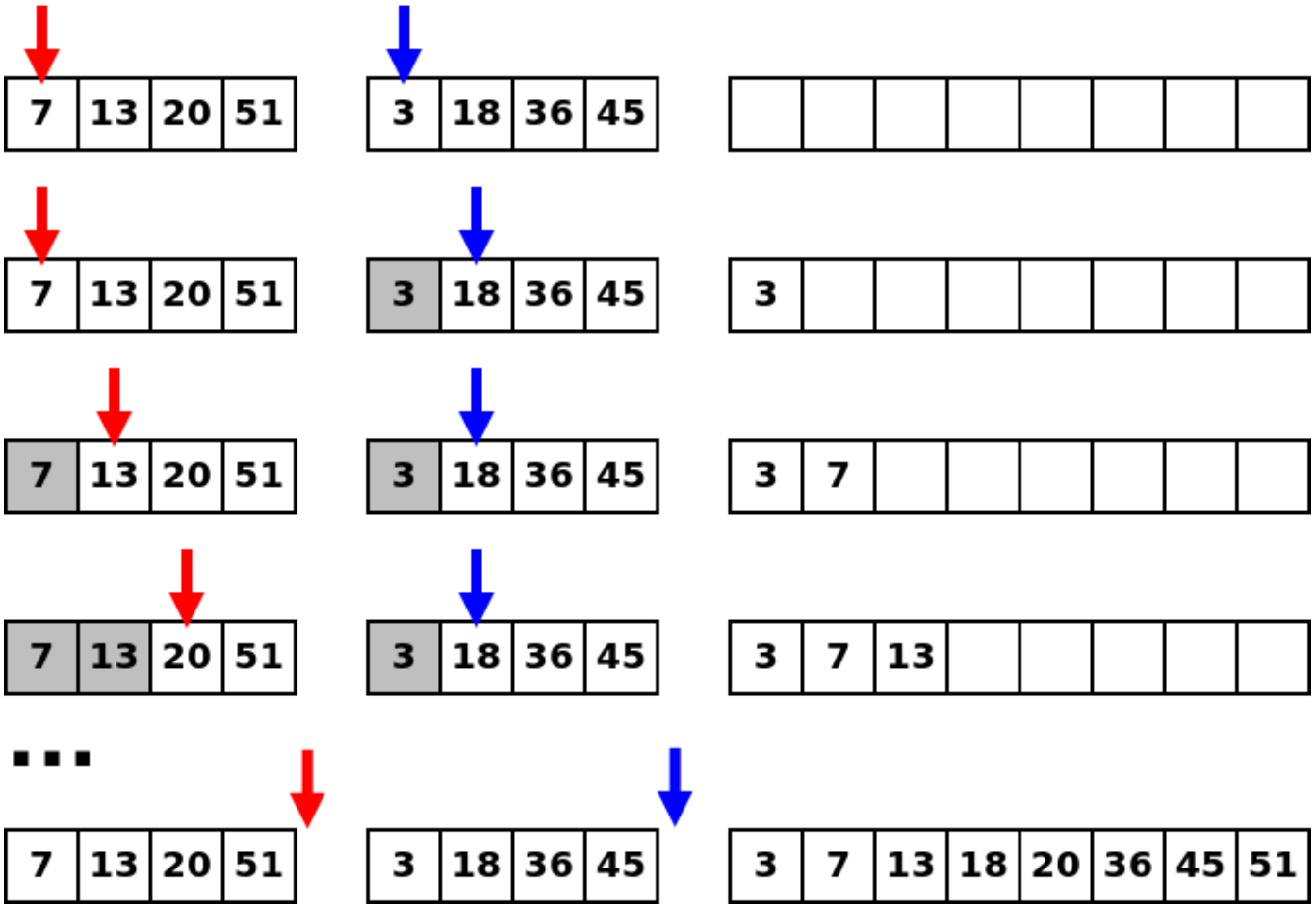
- Let's see this in actual C code
- We will assume we are sorting between positions *start* and *end* (to sort everything we just need to call with arguments 0 and $size - 1$)
- The main function is very similar to our previous ones:

```
void mergeSort(int v[], int start, int end) {
  if (start == end) return;          // base case
  int middle = (start + end) / 2;
  mergeSort(v, start, middle);       // recursive call
  mergeSort(v, middle+1, end);       // recursive call
  merge(v, start, middle, end);      // combine results
}
```

- The *"hard"* part is merging two sorted halves...

Recursion: MergeSort

How to (**efficiently**) merge two sorted arrays?



Only n comparisons to produce final merged array (*linear execution time*)

Recursion: MergeSort

- The merge part in C code:

```c
void merge(int v[], int start, int middle, int end) {
  int aux[end-start+1]; // new temporary array
  int p1 = start;       // "Position" in the left half array
  int p2 = middle+1;    // "Position" in the right half array
  int cur = 0;          // "Position" in the array aux[]

  while (p1 <= middle && p2 <= end) { // While we can compare
    if (v[p1] <= v[p2]) aux[cur++] = v[p1++]; // choose smaller
    else aux[cur++] = v[p2++];                // and add
  }

  while (p1<=middle) aux[cur++] = v[p1++]; // Add remainder
  while (p2<=end)    aux[cur++] = v[p2++]; // elements

  // Copy array aux[] to v[]
  for (int i=0; i<cur; i++) v[start+i] = aux[i];
}
```

- Fully functional MergeSort code: `mergesort.c` (source)
- Note: MergeSort is very efficient and much faster than algorithms such as BubbleSort, InsertionSort or SelectionSort

Recursion: a few caveats

- The most common errors with recursion are:
  - The **base case has been forgotten**, or not all base cases have been dealt with
  - The recursive case is not applied to smaller cases, and so the **recursion does not converge**

- In both cases the recursion becomes **"infinite"** and the function will run out of memory until it gets a *Stack Overflow* error

  Note: a recursion "internally" uses a stack to store the state of previous calls
  (when it exits a call, it returns to the last call before it)

Recursion: inverting an array

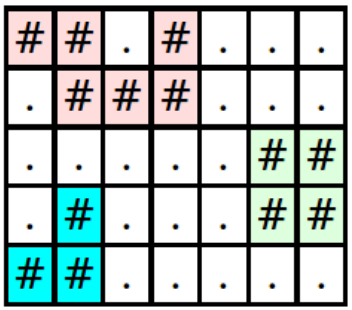
- The most "complicated" part is choosing the recursive division that fits the problem
- Let's look at another case: how to **invert the contents of an array**? Example: $[1, 2, 3, 4, 5] \rightarrow [5, 4, 3, 2, 1]$
  - If we swap the first with the last element... all that's left is **inverting the rest**
  - The **base case** is any array of size less than 2: the inverted array is equal to it
  - Let's assume we're inverting between positions *start* and *end*

Code: `reverse.c` (source)

```
void reverse(int v[], int start, int end) {
  if (start>=end) return; // Base case:  array size < 2
  int tmp = v[start];     // Swap first with last
  v[start] = v[end];
  v[end] = tmp;
  reverse(v, start+1, end-1); // Recursive call for the rest
}
```

Recursion: flood fill

- It can be useful to have more than two recursive calls
- Consider a matrix where two cells are *neighbours* if they are adjacent *vertical* or *horizontally*
- A **connected component** is a set of non-empty neighbouring cells. For example, in the following figure, we have 3 spots:
  - A red component with 6 cells
  - A green component with 4 cells
  - A blue component with 3 cells



- How to calculate the **size of a component**?

Recursion: flood fill

- Recursive definition:
  - Let m[R][C] be the matrix of cells with **R** rows and **C** columns
  - Let $f(y, x)$ be the component of the spot at position (y, x)
  - If m[y][x] is an empty cell, then $f(y, x) = 0$
    Otherwise, $f(y, x) = 1 + f(y+1, x) + f(y-1, x) + f(y, x+1) + f(y, x-1)$

An incorrect implementation:

```
// We are assuming that m[][], R and C are global variables
int f(int y, int x) {
  if (m[y][x] == '.') return 0; // Base case:  empty cell
  int count = 1;                // Non-empty cell
  count += f(y-1, x);           // Adding neighbor cells
  count += f(y+1, x);
  count += f(y, x+1);
  count += f(y, x-1);
  return count;
}
```

- Problems with this implementation? Array boundaries!
  e.g. $f(0, 0)$ will call $f(-1, 0)$, which tries to access a cell outside the limits of the matrix

Recursion: flood fill

- An implementation that is still incorrect:

```
// We are assuming that m[][], R and C are global variables
int f(int y, int x) {
  if (y<0 || y>=R || x<0 || x>=C) return 0; // Out of bounds
  if (m[y][x] == '.') return 0; // Base case:  empty cell
  int count = 1;                // Non-empty cell
  count += f(y-1, x);           // Adding neighbor cells
  count += f(y+1, x);
  count += f(y, x+1);
  count += f(y, x-1);
  return count;
}
```

- Problems with this implementation? Infinite recursion!
  e.g.: f(0,0) calls f(1,0), which calls f(0,0), which calls f(0,1), which calls f(0,0), which calls f(0,1), ...
- We need to make sure we don't go back to a cell we've already visited

Recursion: flood fill

- A correct implementation
  - *visited*[*R*][*C*] is an int array initialized with zeros *(false)*

```
// We're assuming that m[][], R, C and visited[][] are global variables
int f(int y, int x) {
  if (y<0 || y>=R || x<0 || x>=C) return 0; // Out of bounds
  if (visited[y][x]) return 0;  // Cell already visited
  if (m[y][x] == '.') return 0; // Base case:  empty cell
  int count = 1;                // Non-empty cell
  visited[y][x] = 1;            // Mark as visited
  count += f(y-1, x);           // Adding neighbor cells
  count += f(y+1, x);
  count += f(y, x+1);
  count += f(y, x-1);
  return count;
}
```

- You can check a functional implementation
  - Code: `floodfill.c` (source) | Example input: (floodfill_input.txt)

Recursion: generating subsets

- How do you **generate all subsets** of a given set?
  Example: {1,2,3} has 8 subsets:
  - {1,2,3}, {1,2}, {1,3}, {2,3}, {1}, {2}, {3}, {}
- **Recursive definition?** Subsets of {1,2,3} are:
  - {1} ∪ subsets of {2,3}: {1,2,3}, {1,2}, {1,3}, {1}
    e
  - {} ∪ subsets of {2,3}: {2,3}, {2}, {3}, {}
- In other words, the 1st element is either in the set or it isn't, and for each of these cases, we have all the subsets of the 'remainder'

Recursion: generating subsets

- Let the inclusion in the set be represented by an array of True/False:
  Examples: [T,T,T] represents {1,2,3}; [T,T,F] represents {1,2}
- Then all subsets are:
  - Arrays where the 1st position is T + all following subsets
    plus
  - Arrays where the 1st position is F + all following subsets

[T,T,T] = {1,2,3}
[T,T,F] = {1,2}
[T,F,T] = {1,3}
[T,F,F] = {1}
[F,T,T] = {2,3}
[F,T,F] = {2}
[F,F,T] = {3}
[F,F,F] = {}

Recursion: generating subsets

- Implementing: `subsets.c` (source)

```c
// Generate all subsets starting at position "cur"
void goSubsets(int cur, int v[], int n, int used[]) {
  if (cur == n) { // Base case:  we finish the subset
    for (int i=0; i<n; i++) // Write subset
      if (used[i]) printf("%d ", v[i]);
    printf("\n");
  } else { // If we haven't finished, continue generating
    used[cur] = 1; // Subsets that include the current element
    goSubsets(cur+1, v, n, used); // Recursive call
    used[cur] = 0; // Subsets that don't include the current element
    goSubsets(cur+1, v, n, used); // Recursive call
  }
}

// Write all subsets of the array v[] of size n
void subsets(int v[], int n) {
  // array of ints (T/F) to represent the subset
  int used[n];
  goSubsets(0, v, n, used); // call recursive function
}
```