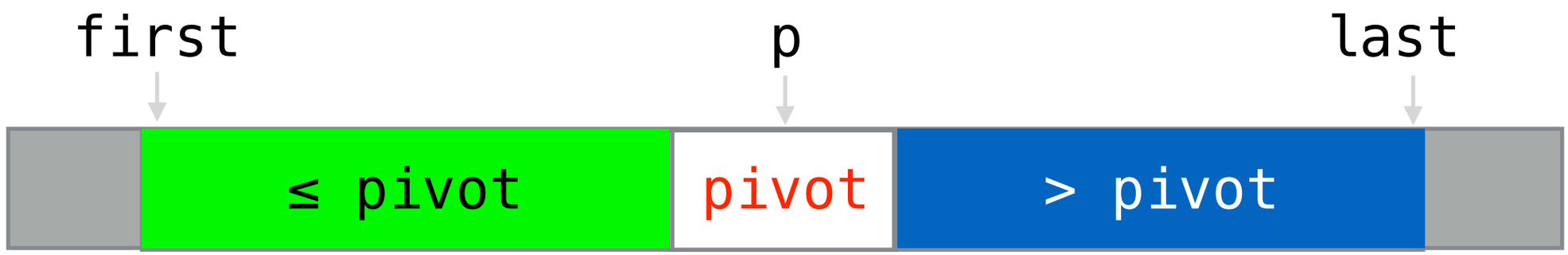
QuickSort

partition

Precondition: Array `arr` and indices `first` and `last` such that

$$0 \le \texttt{first} \le \texttt{last} \le \texttt{arr.length}$$

Postcondition: Rearranges `arr` and returns `p` so that:



partition

```
private static int partition(int[] arr,
                             int first, int last)
```

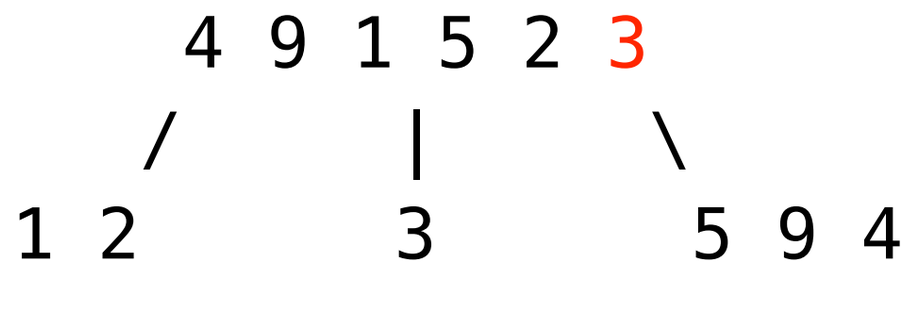
quickSort

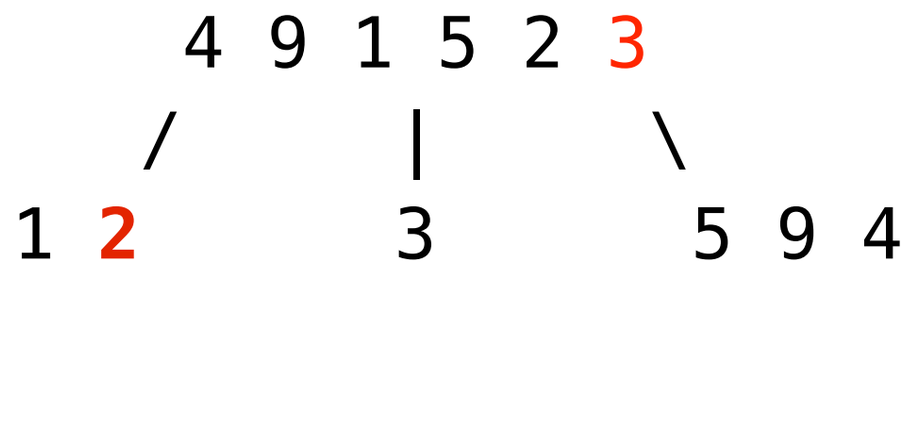
```
public static void quickSort(int[] arr)
{
  quickSortRec(arr, 0, arr.length-1);
}
```

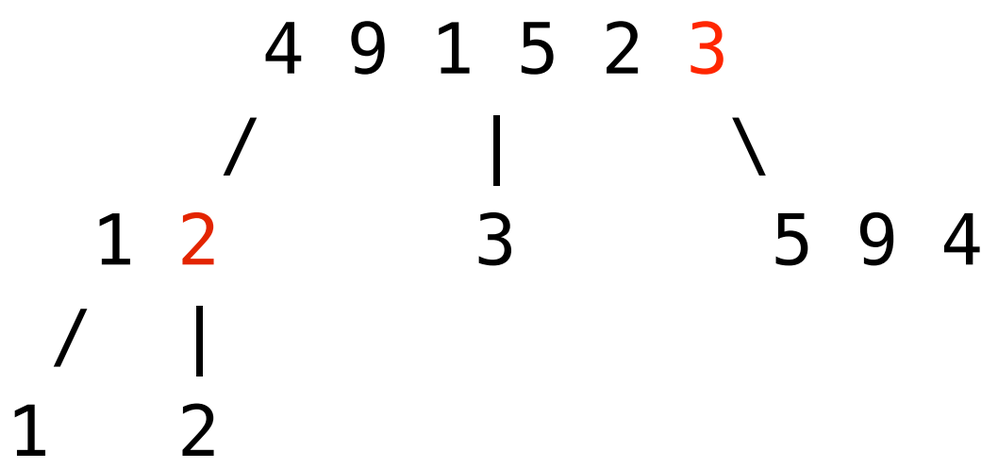
```
quickSortRec(arr, first, last)
  if (first ≥ last) return
  p = partition(arr, first, last)
  quickSortRec(arr, first, p - 1)
  quickSortRec(arr, p + 1, last)
```

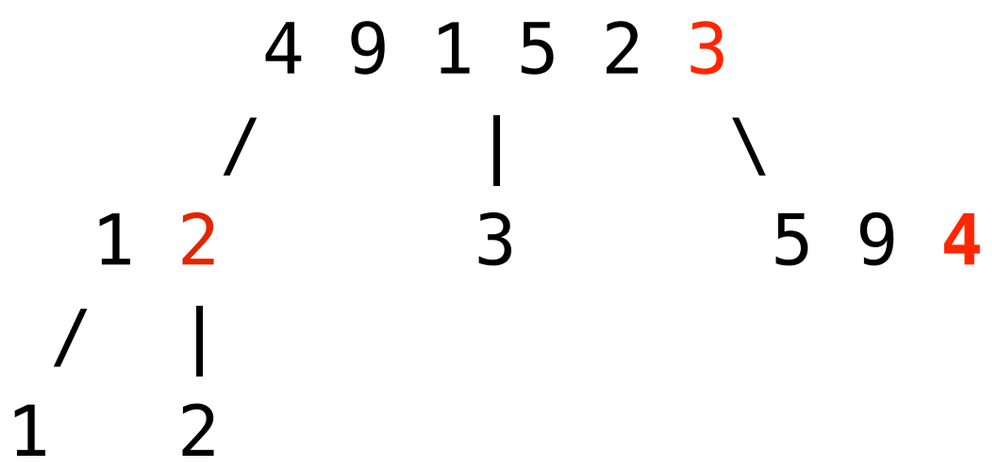
4 9 1 5 2 3

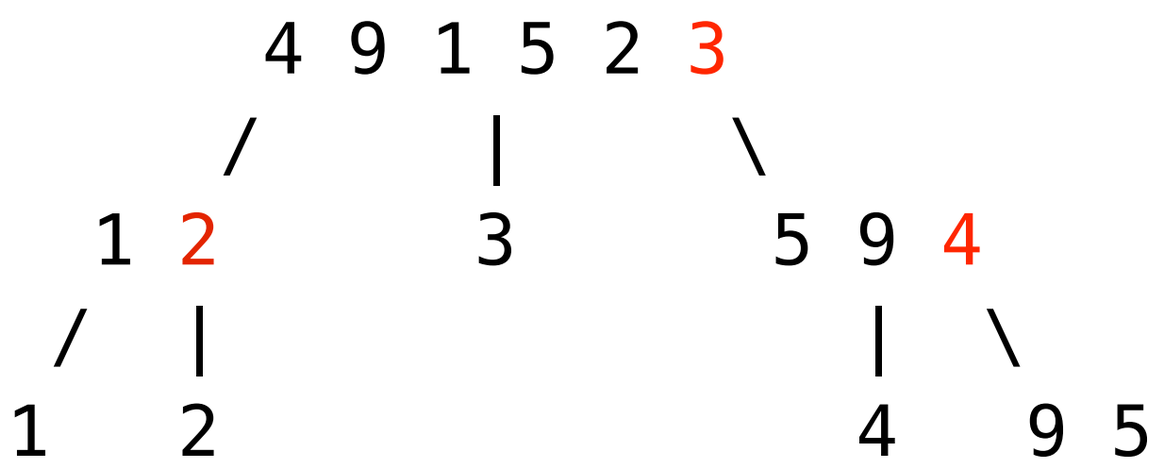
4 9 1 5 2 **3**

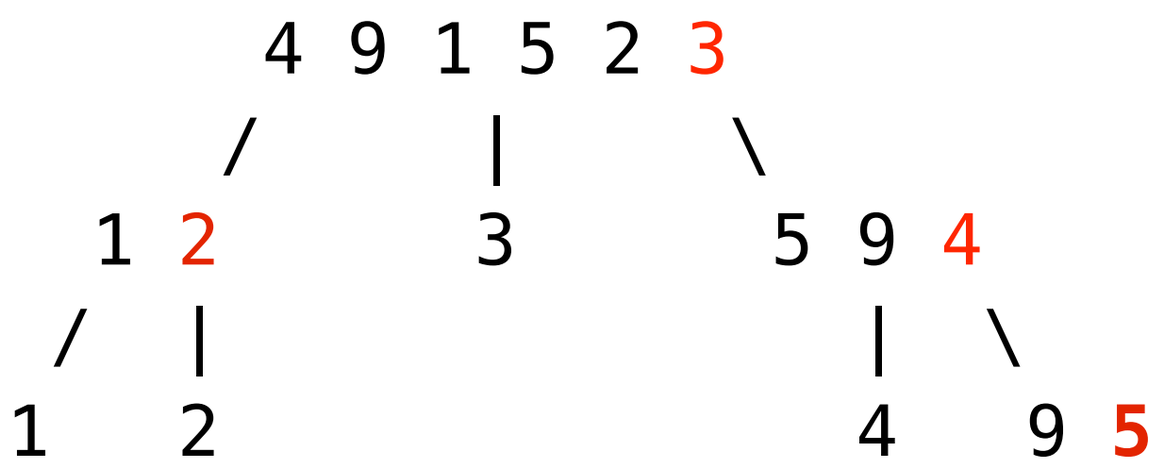

4 9 1 5 2 3
1 2
3
5 9 4

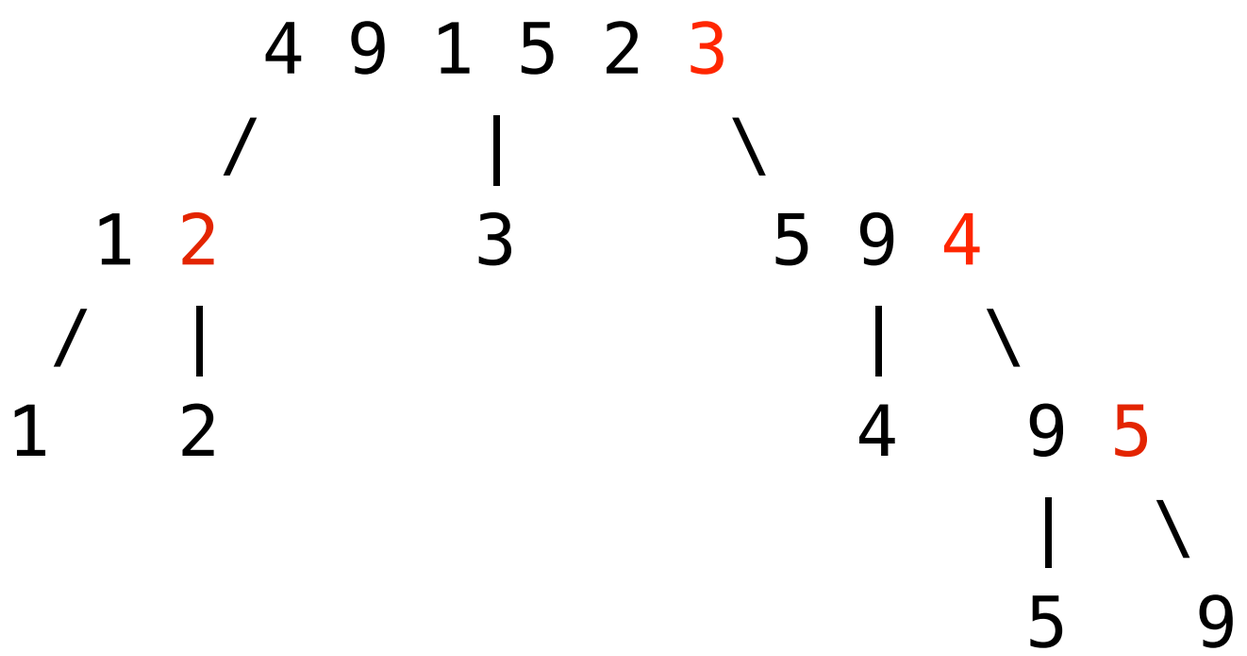

4 9 1 5 2 3
1 2 3 5 9 4


4 9 1 5 2 3
1 2 3 5 9 4
1 2


4 9 1 5 2 3
1 2 3 5 9 4
1 2

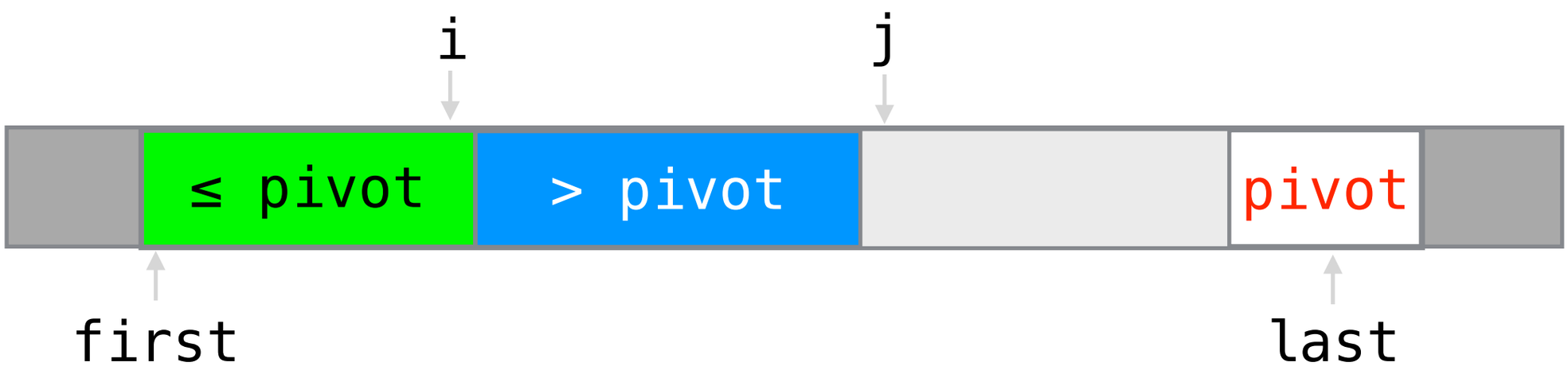

4 9 1 5 2 3
1 2 3 5 9 4
1 2 4 9 5


4 9 1 5 2 3
1 2 3 5 9 4
1 2 4 9 5


4 9 1 5 2 3
1 2 3 5 9 4
1 2 4 9 5
5 9

```
Partition(arr,first,last)
  // Use last element as pivot.
  pivot = arr[last]
  i = first - 1
  for (j = first; j < last; j++)
    if arr[j] ≤ pivot
      i++
      swap arr[i] and arr[j]
// Now put pivot in position i+1.
swap arr[i+1] and arr[last]
return i + 1
```

Partitioning



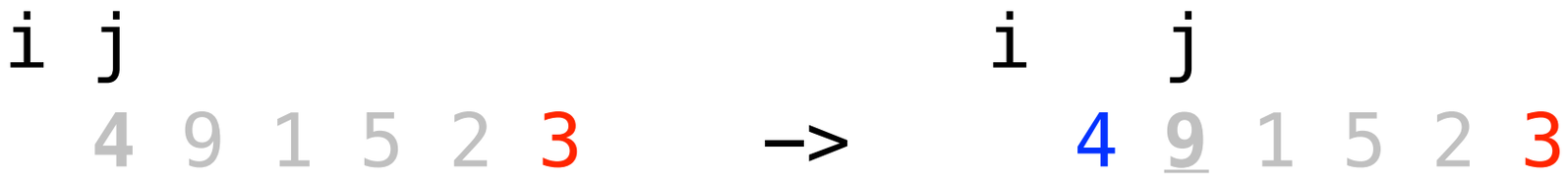
4 9 1 5 2 3

i j

4 9 1 5 2 3

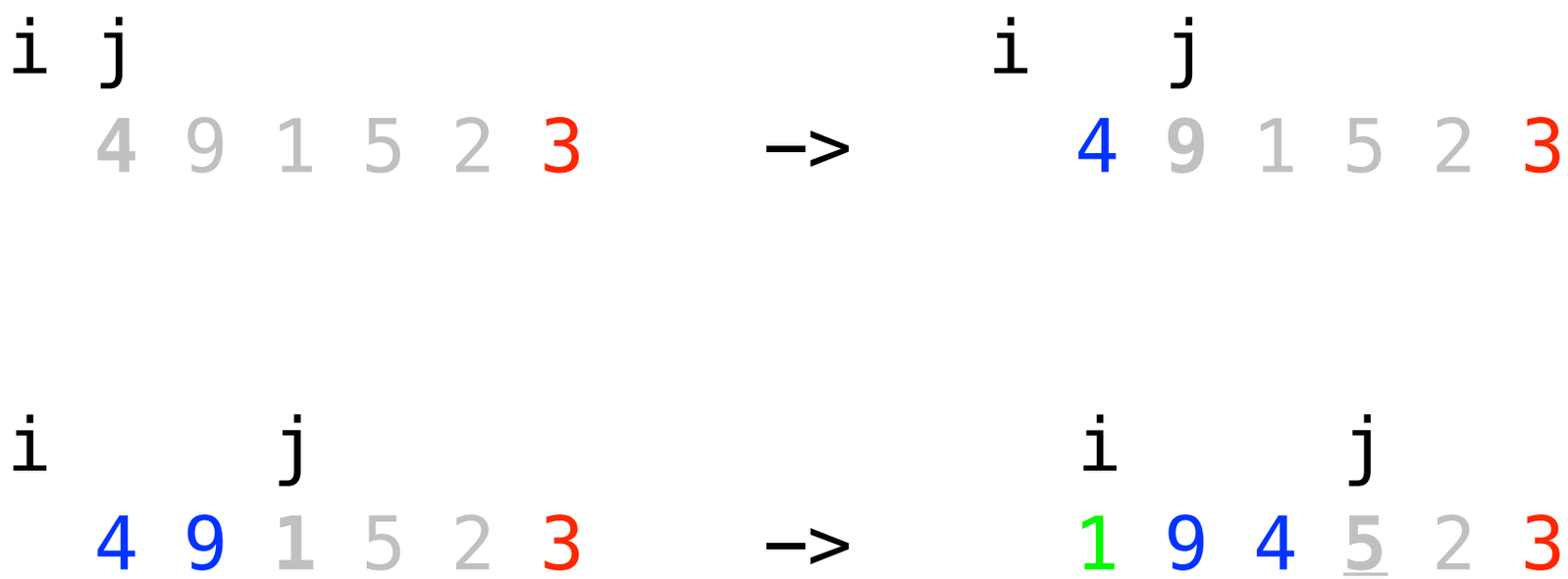
i j

4 9 1 5 2 3


i j
4 9 1 5 2 3
->
i j
4 9 1 5 2 3

```
i j                          i   j
  4 9 1 5 2 3      ->          4 9 1 5 2 3
```

```
i     j
  4 9 1 5 2 3
```


i j
4 9 1 5 2 3
->
i j
4 9 1 5 2 3
i j
4 9 1 5 2 3
->
i j
1 9 4 5 2 3

```
i j                     i   j
  4 9 1 5 2 3    ->       4 9 1 5 2 3

i     j                   i     j
  4 9 1 5 2 3    ->       1 9 4 5 2 3
```

```
i j                         i   j
  4 9 1 5 2 3      ->        4 9 1 5 2 3

i     j                     i     j
  4 9 1 5 2 3      ->        1 9 4 5 2 3

i       j                   i       j
1 9 4 5 2 3        ->      1 2 4 5 9 3
```

```
i j                         i   j
  4 9 1 5 2 3      ->        4 9 1 5 2 3

i     j                     i     j
  4 9 1 5 2 3      ->        1 9 4 5 2 3

i       j                   i       j
1 9 4 5 2 3        ->      1 2 4 5 9 3

  i       j
1 2 3 5 9 4
```

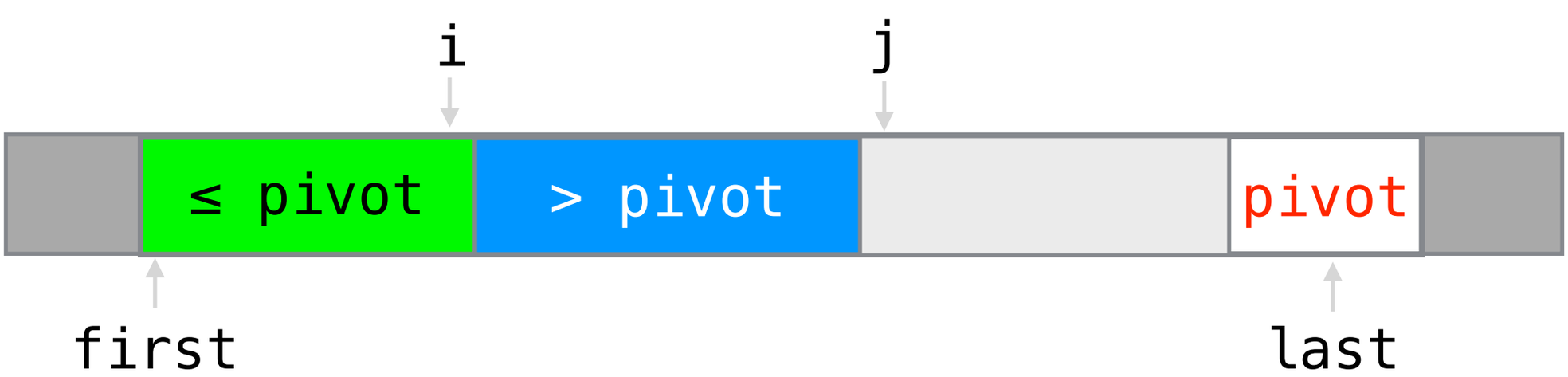
Loop Invariant

After each iteration of the **`for`** loop,

- if `first ≤ k ≤ i`, then `arr[k] ≤ pivot`,
- if `i+1 ≤ k ≤ j-1`, then `arr[k] > pivot`, and
- if `k == last`, then `arr[k] = pivot`.

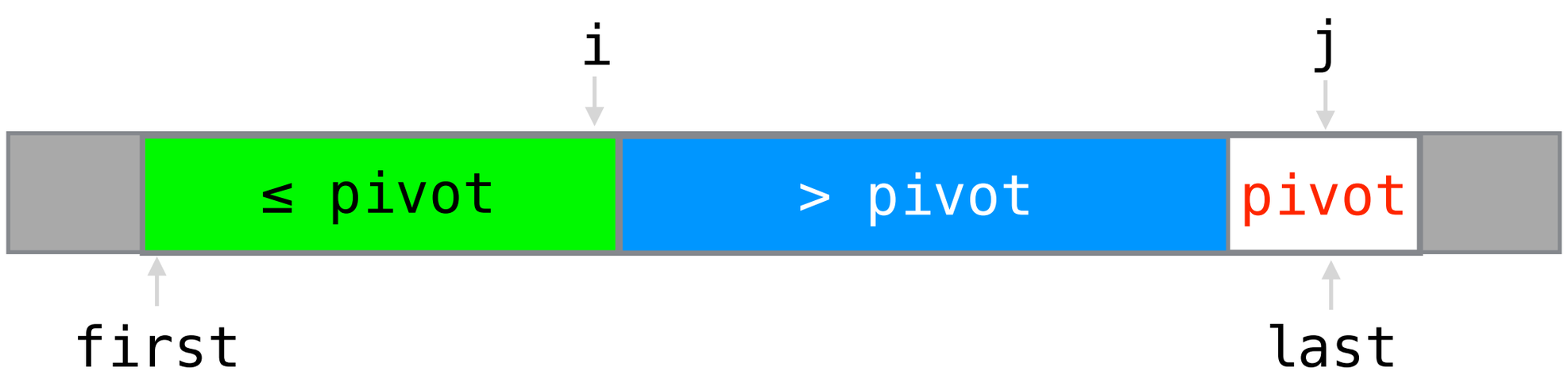
Loop Invariant

After each iteration of the **`for`** loop,



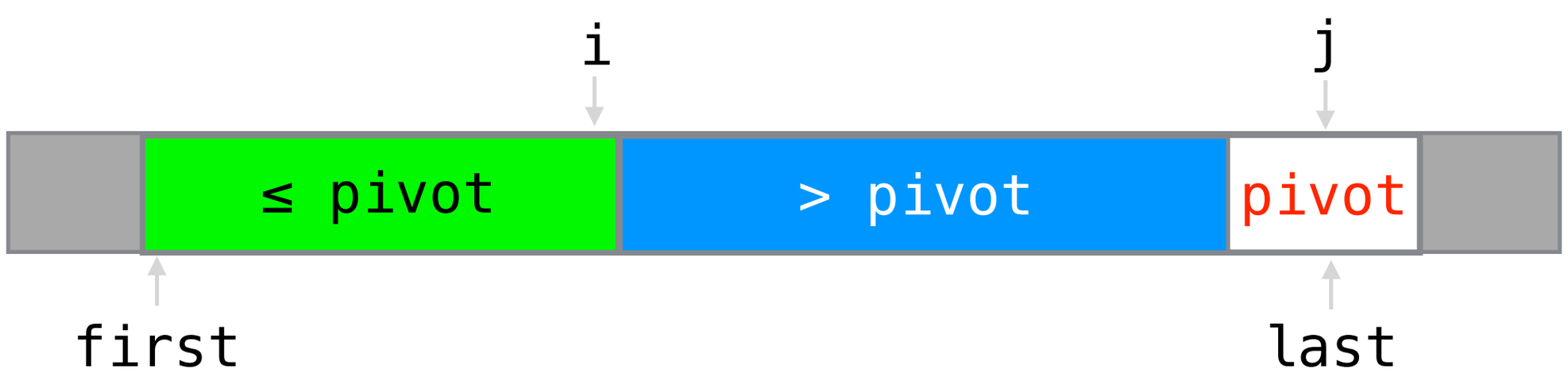
Loop Invariant: Termination

After the **last** iteration of the **`for`** loop,



Loop Invariant: Termination

After the **last** iteration of the **`for`** loop,

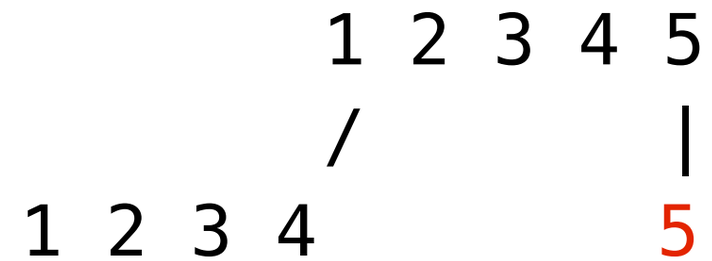


Postcondition is achieved by swapping `arr[i+1]` and `arr[last]`

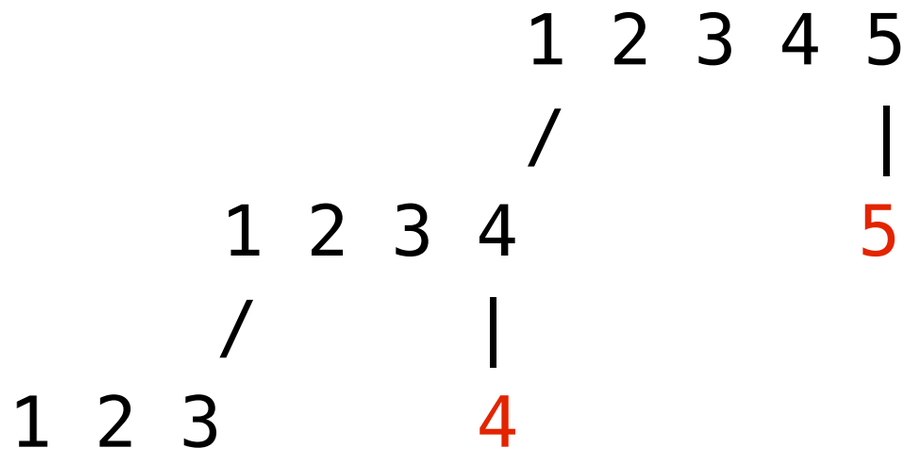
Quicksort: Worst Case

1 2 3 4 5

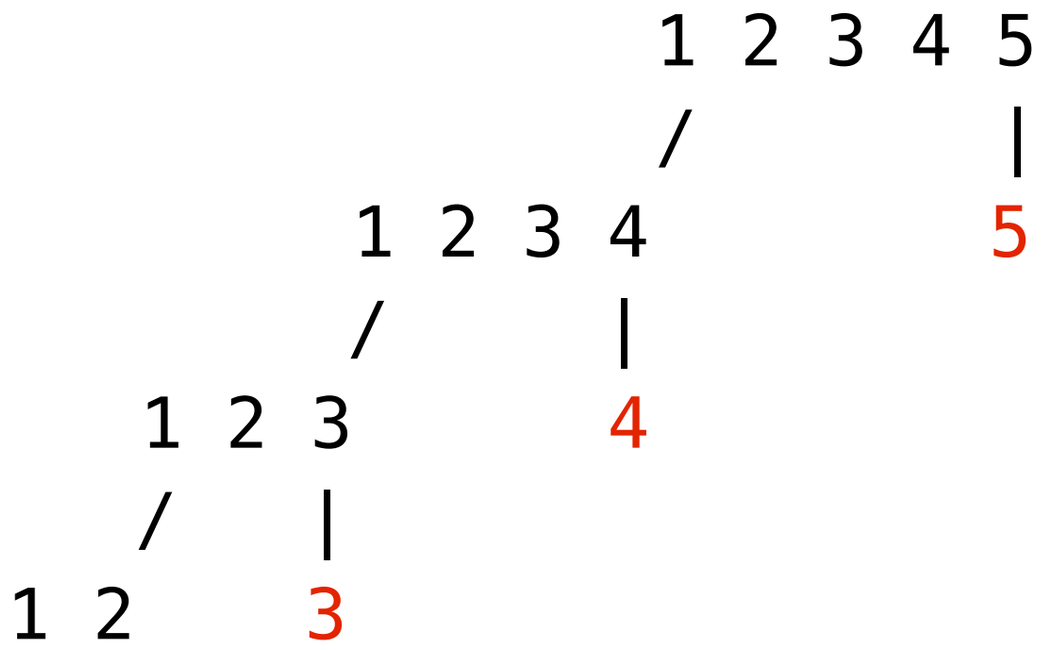
Quicksort: Worst Case



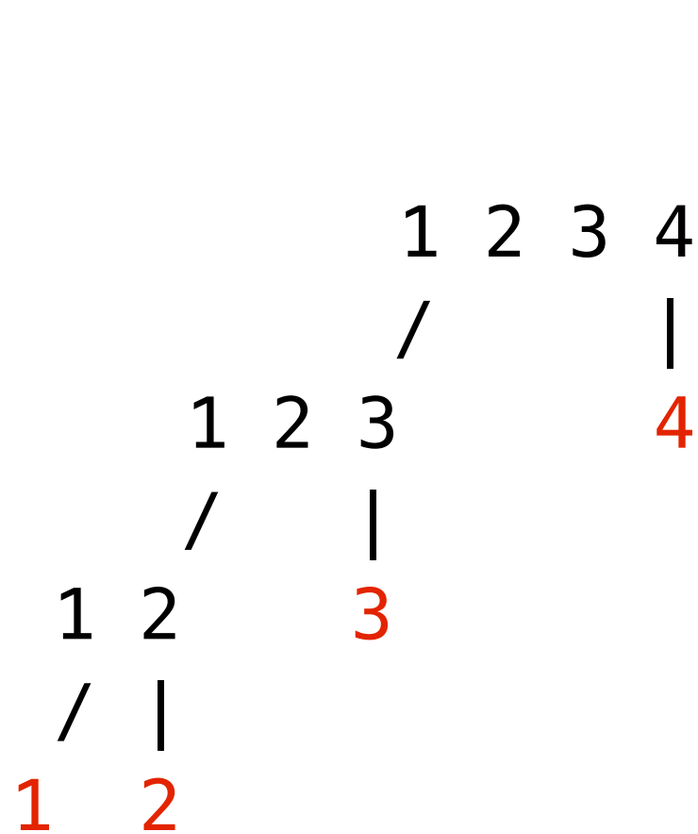
Quicksort: Worst Case



Quicksort: Worst Case



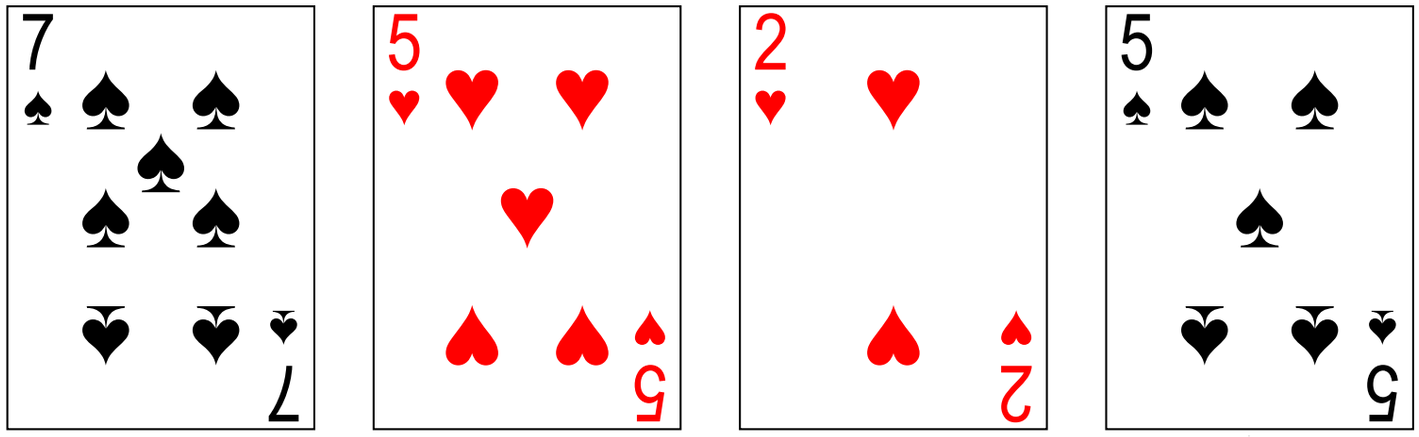
Quicksort: Worst Case



Stability

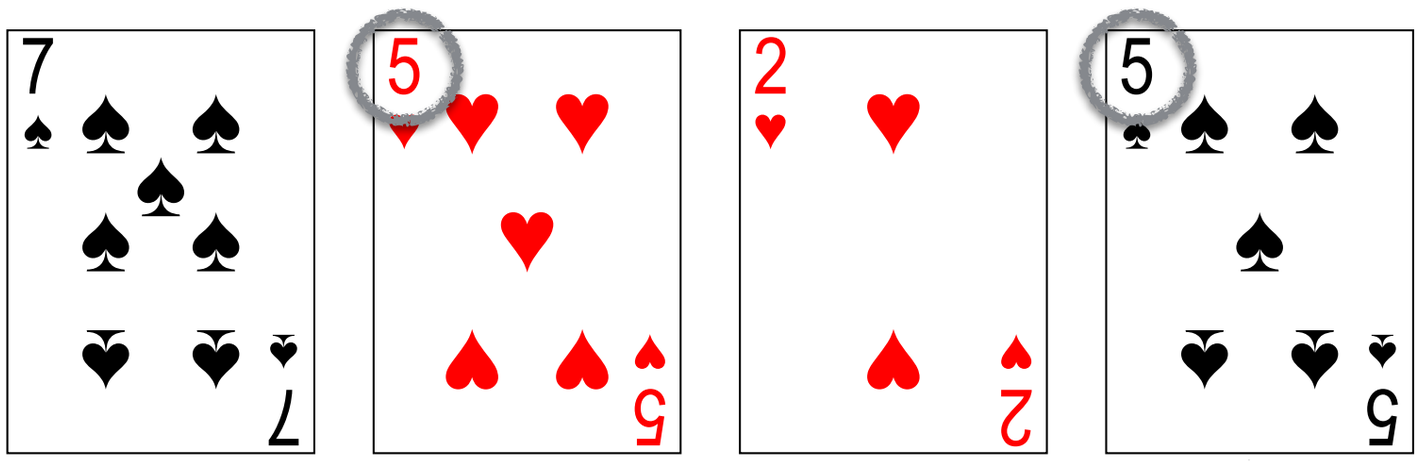
A sorting algorithm is **stable** if whenever there are two records R and S with the same key, and R is before S in the input, then R is before S in the output.

Stable

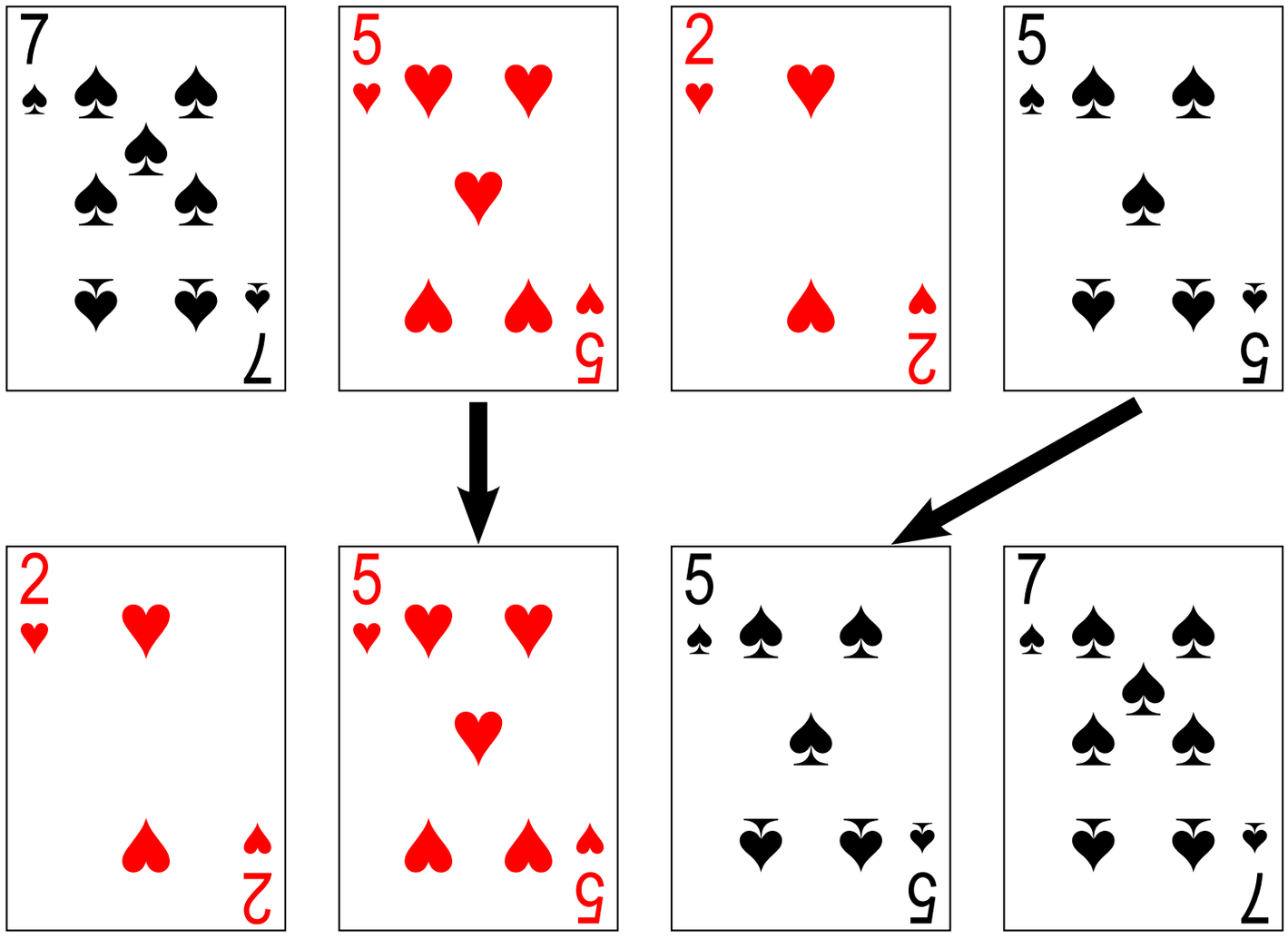


http://en.wikipedia.org/wiki/Sorting_algorithm

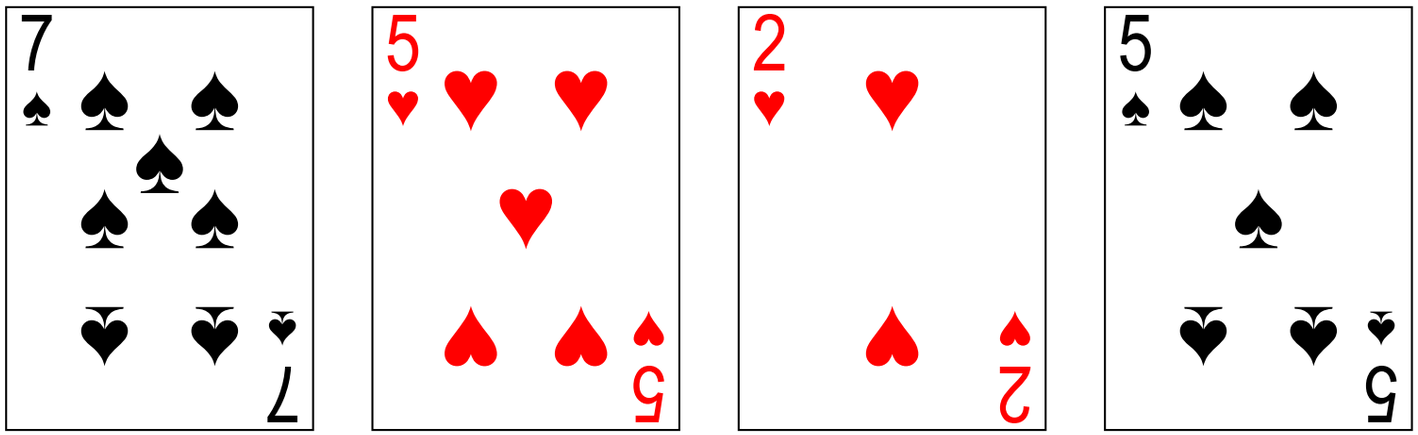
Stable



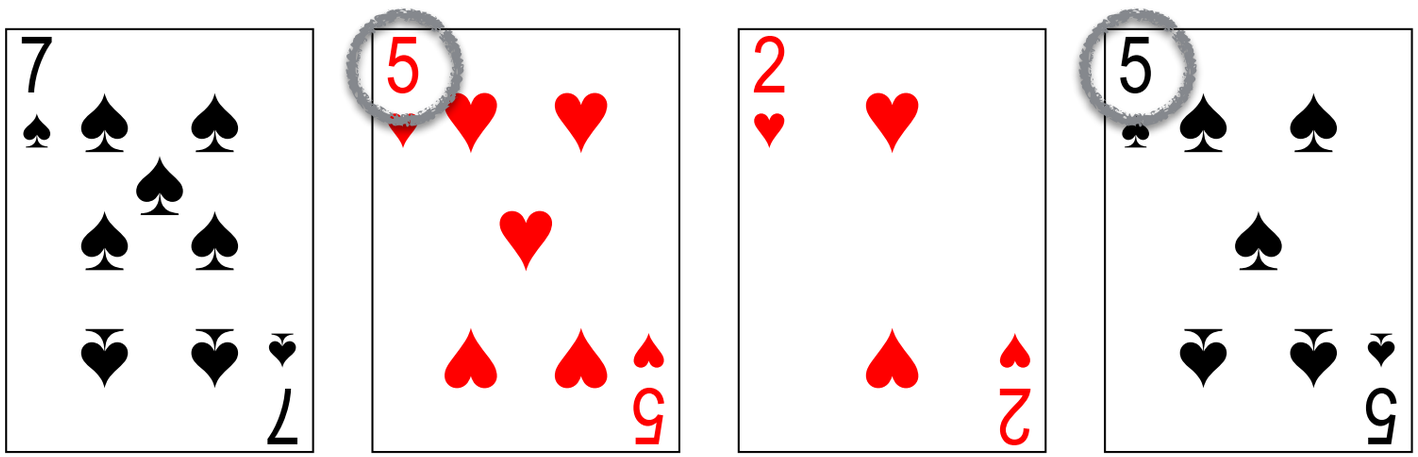
Stable



Unstable



Unstable



Unstable