Data Structures and Algorithms

Sorting

Dr. Lina Xu
lina.xu@ucd.ie

School of Computer Science,
University College Dublin

September 25, 2018

Learning outcomes

After this lecture and the related practical students should...

- understand the operation of traditional sorting algorithms
- understand the limitation of comparison based sorting
- be able to define and use recursive functions
- understand the operation of advanced and non comparison based sorting algorithms
- be able to use bitwise operators to sort keys into buckets in radix sort

Table of Contents

Why do we sort data?

The main reason that we spend so much time sorting data is because we usually want to search it

- The complexity of linear search is $O(n)$
- The complexity of binary search is $O(\log n)$

Binary search is the best possible solution for searching data that is stored in a linear data structure.

- However, this improvement comes at a cost, because the data must be sorted

What are we sorting?

- For each of these sorting algorithms we will be talking about sorting numbers
- Simply sorting numbers is something that would not be very useful
- Any type of data can be sorted but we must decide how it should be sorted
  - Sort people by age
  - Sort items for sale by price
  - Sort shoes by size
- The piece of information that we use to sort the data is called the key
- In all of our examples we will only show the key, but we should not forget that each key has more data associated with it

Complexity of Sorting Data

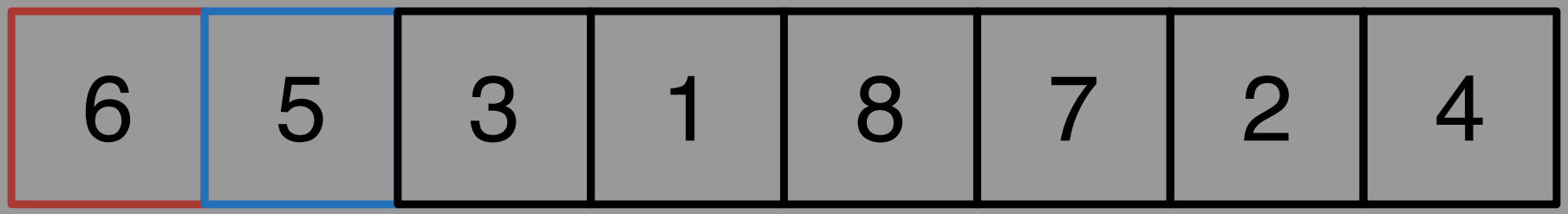
- What is the complexity of sorting data?
  1. $O(1)$
  2. $O(\log n)$
  3. $O(n)$
  4. $O(n * \log n)$
  5. $O(n^2)$

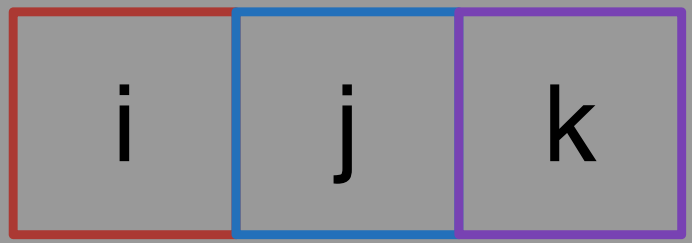
Table of Contents

Selection Sort

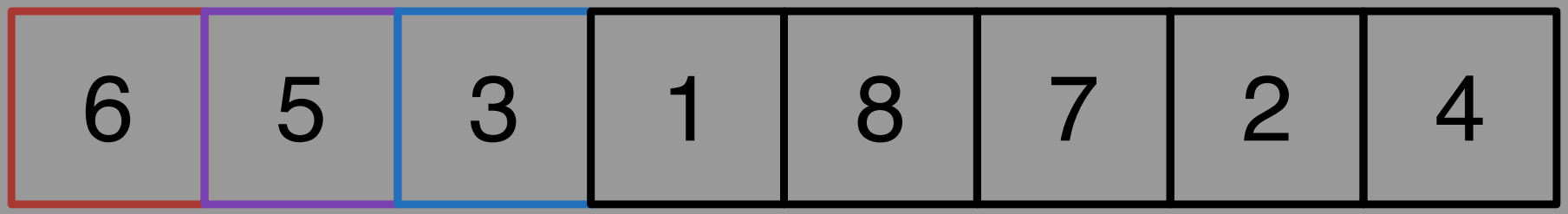
```
1  void selectionSort(int f[], int n) {
2    int i, j;
3    for (i = 0; i < n; i = i + 1) {
4      int k = i;
5      for (j = i + 1; j < n; j = j + 1) {
6        if (f[j] < f[k]) {
7          k = j;
8        }
9      }
10     int temp = f[i];
11     f[i] = f[k];
12     f[k] = temp;
13   }
14 }
```

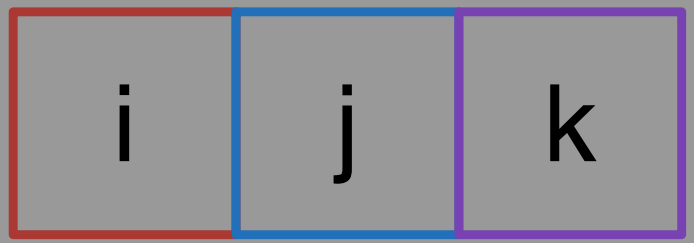
Selection Sort Example



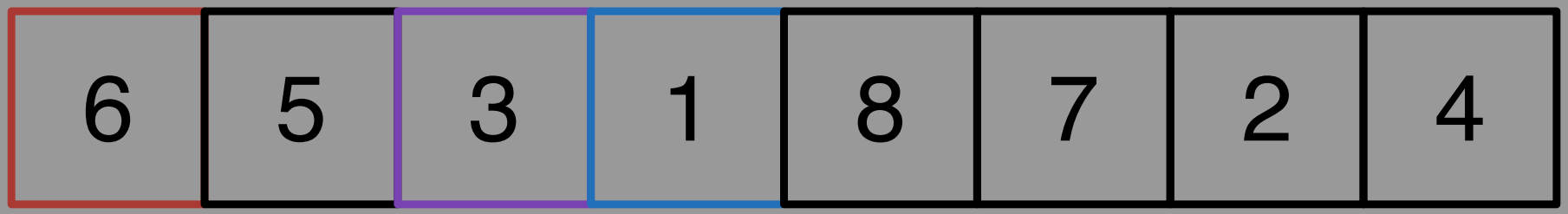


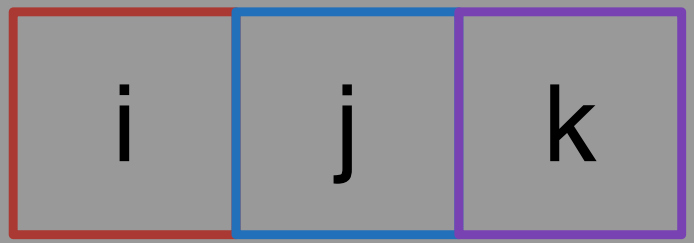
Selection Sort Example



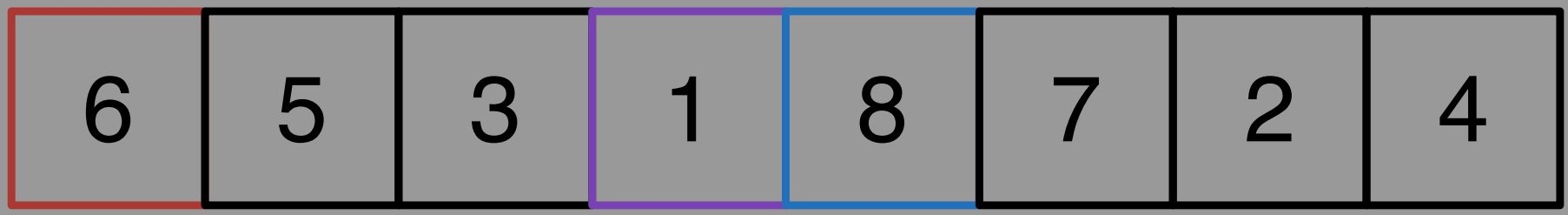


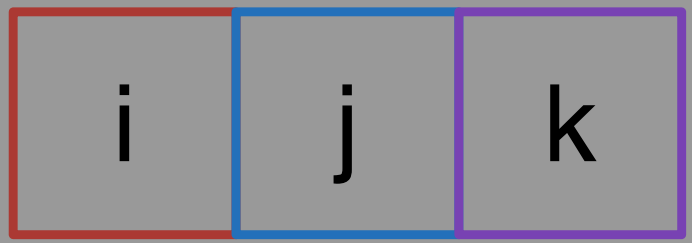
Selection Sort Example



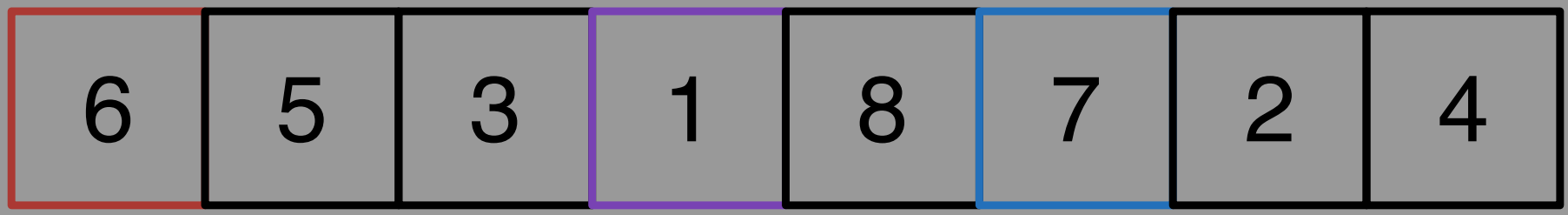


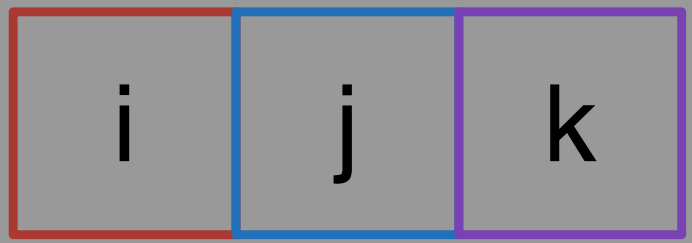
Selection Sort Example



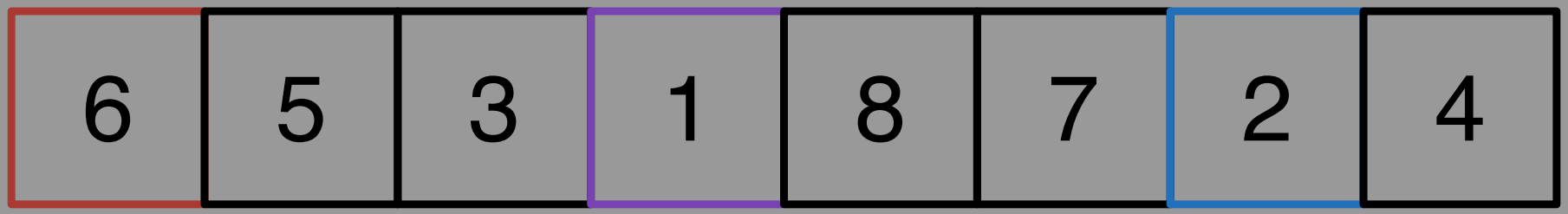


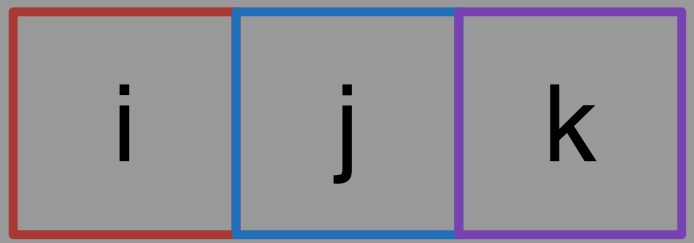
Selection Sort Example



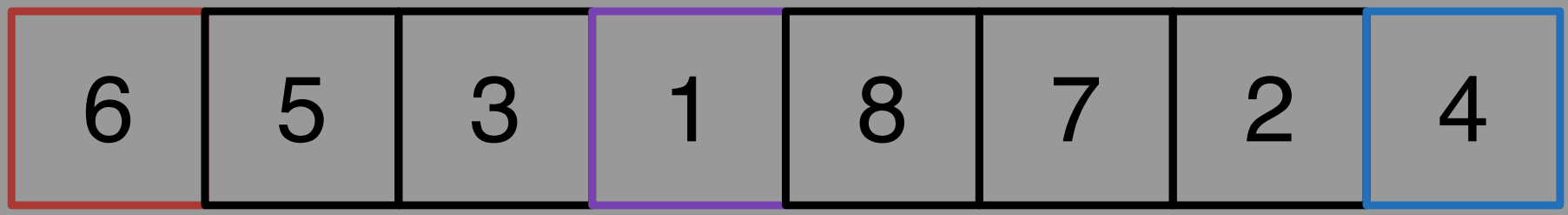


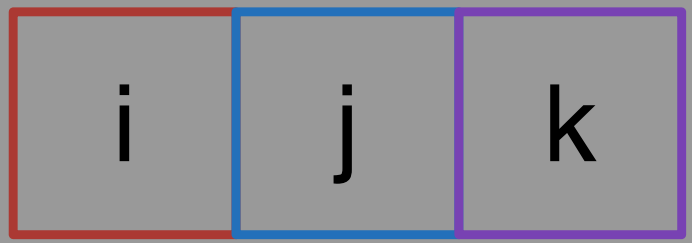
Selection Sort Example



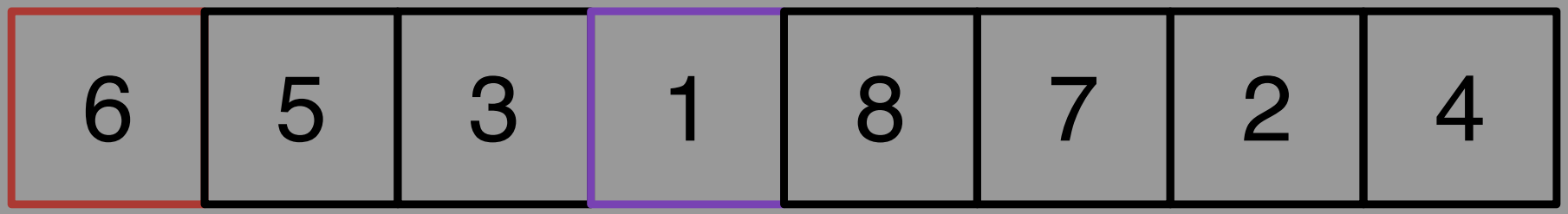


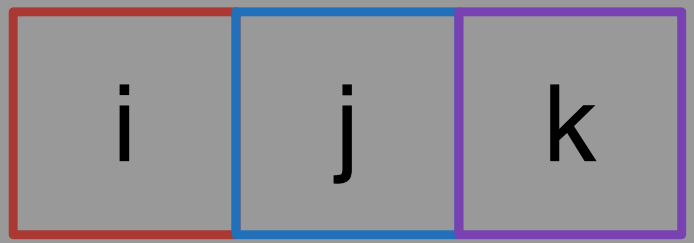
Selection Sort Example



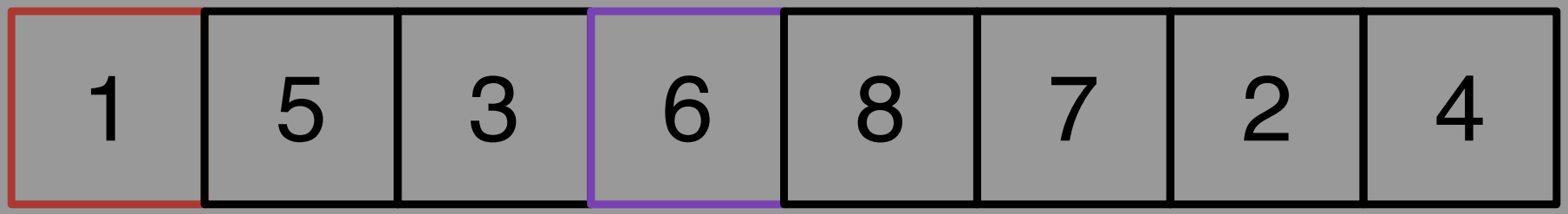


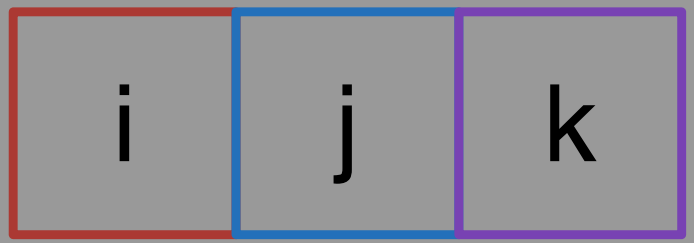
Selection Sort Example



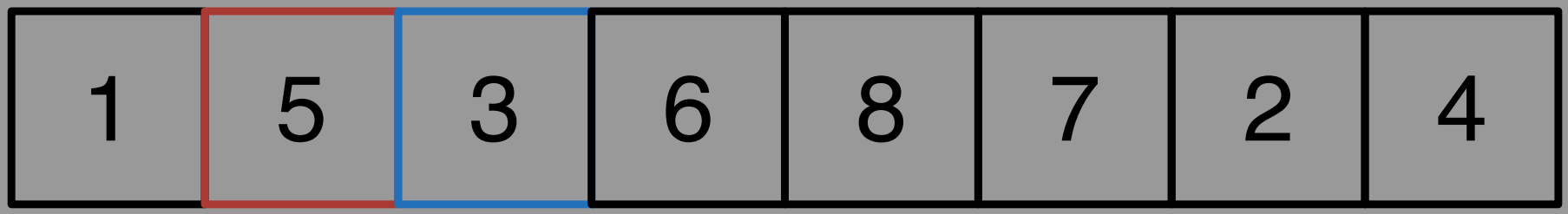


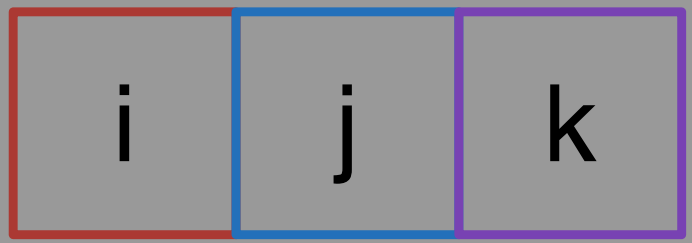
Selection Sort Example



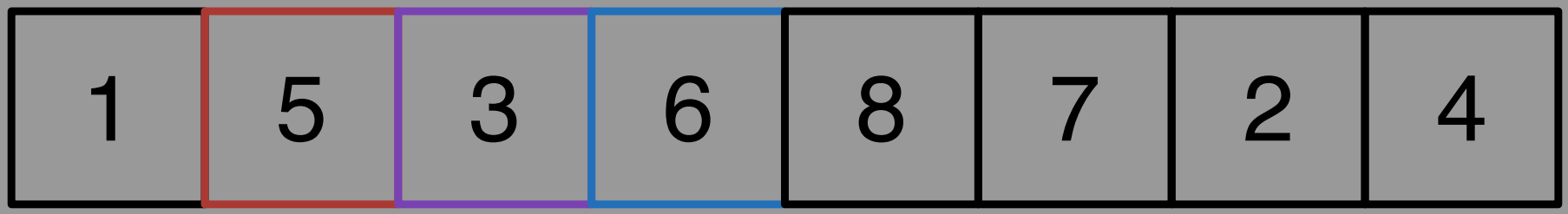


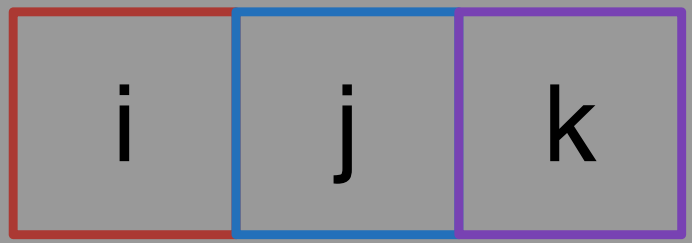
Selection Sort Example



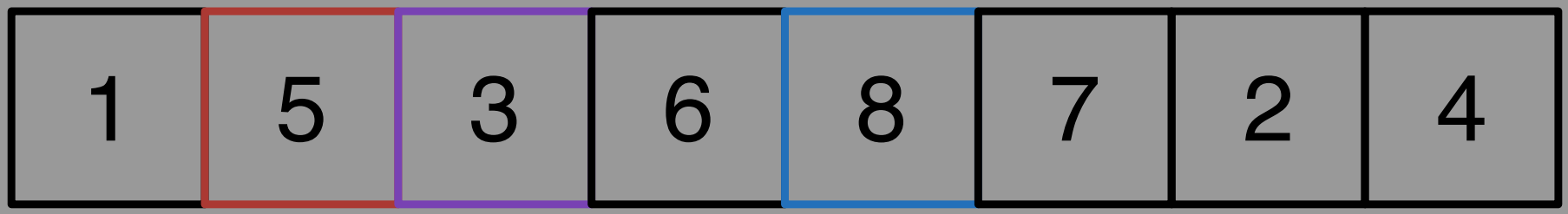


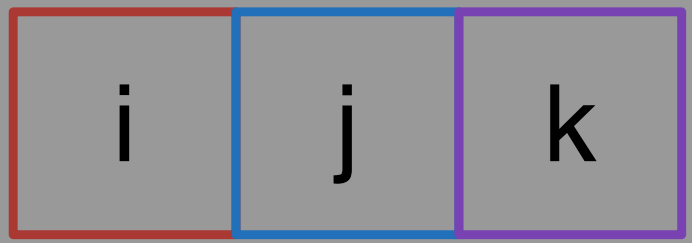
Selection Sort Example



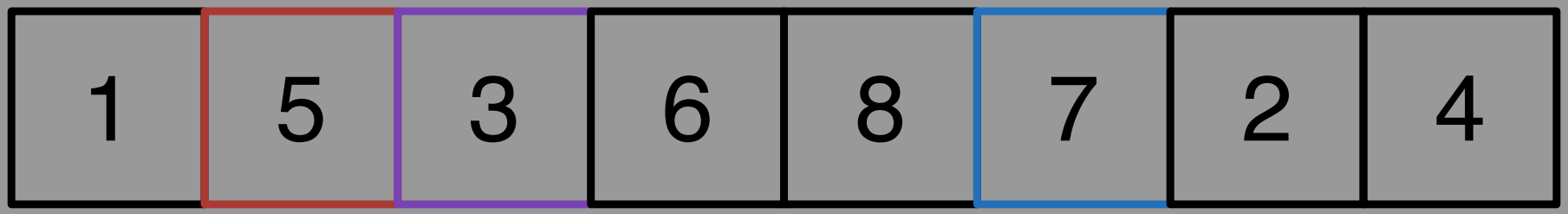


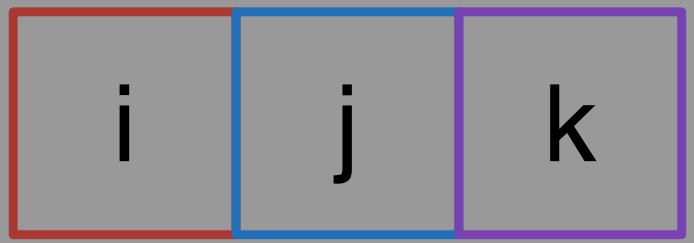
Selection Sort Example



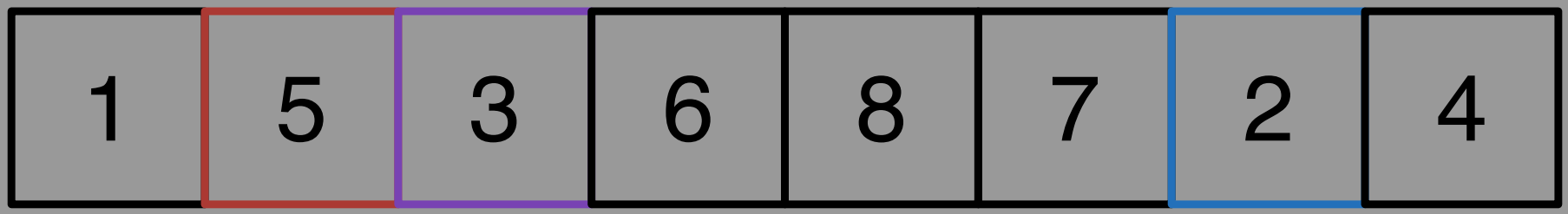


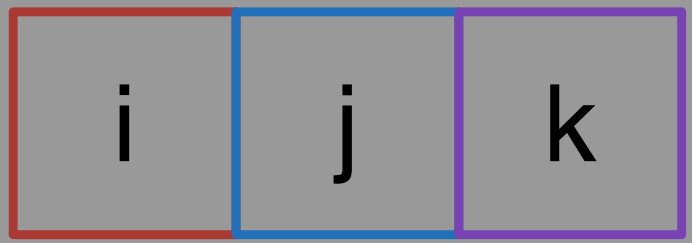
Selection Sort Example



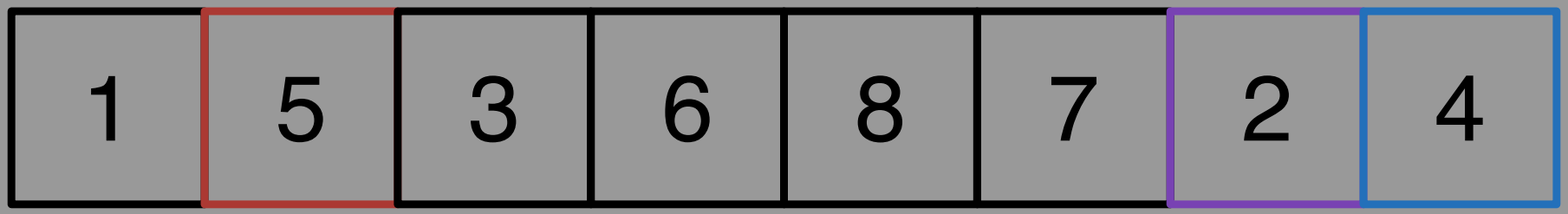


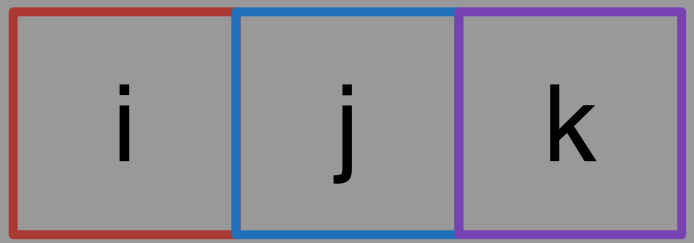
Selection Sort Example



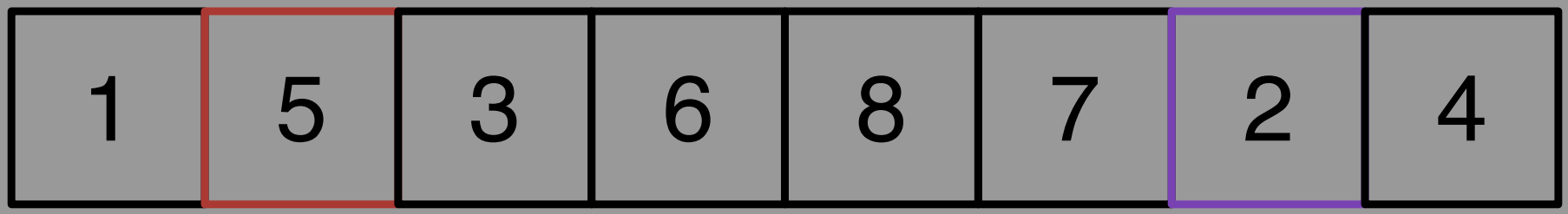


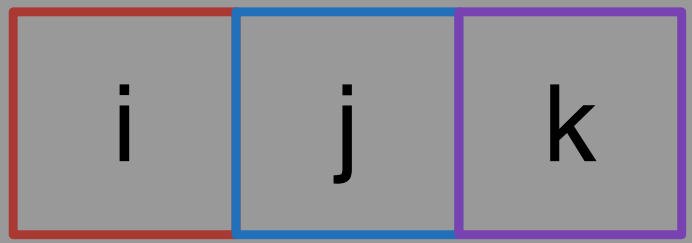
Selection Sort Example



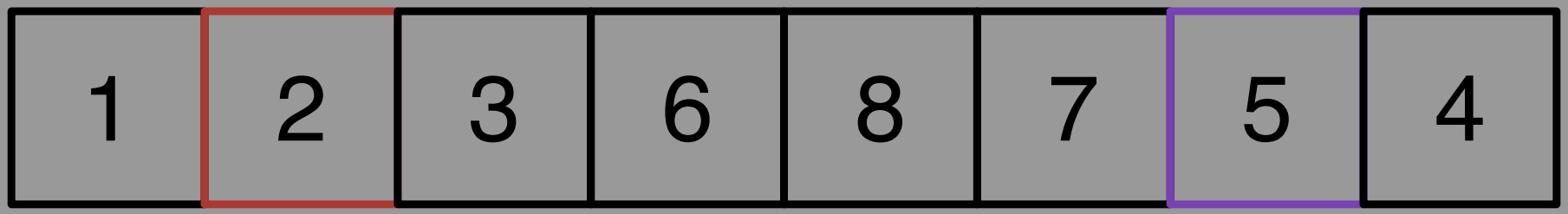


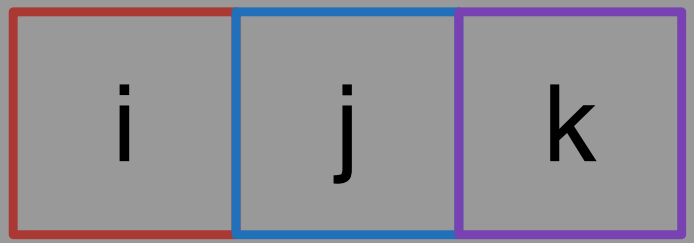
Selection Sort Example



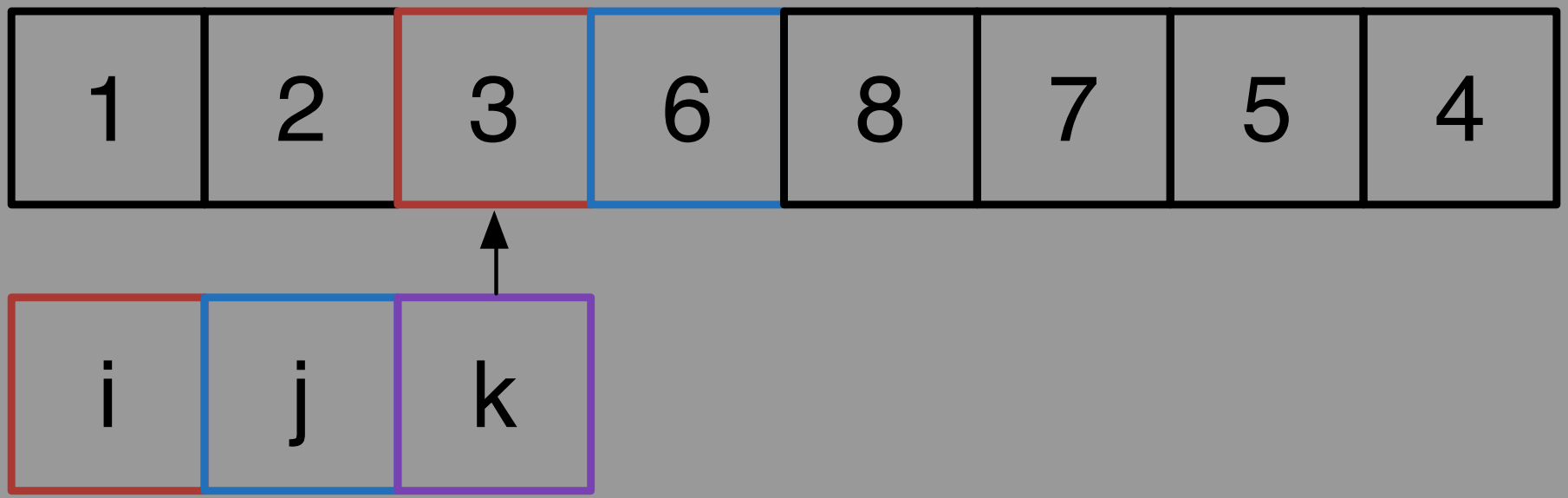


Selection Sort Example

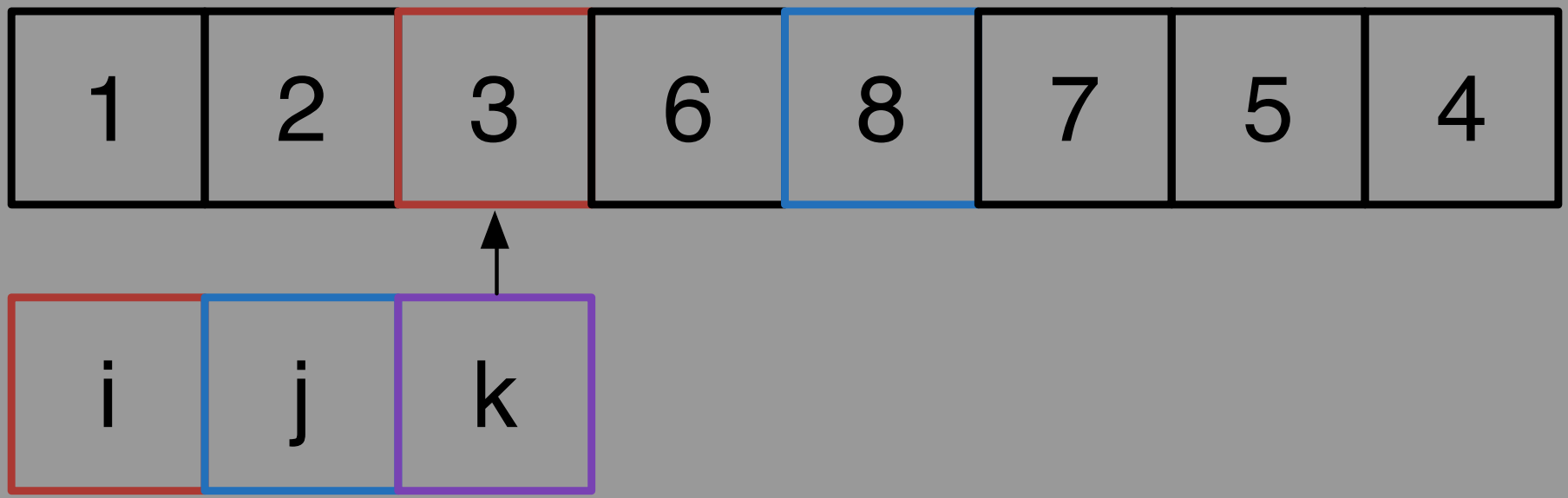




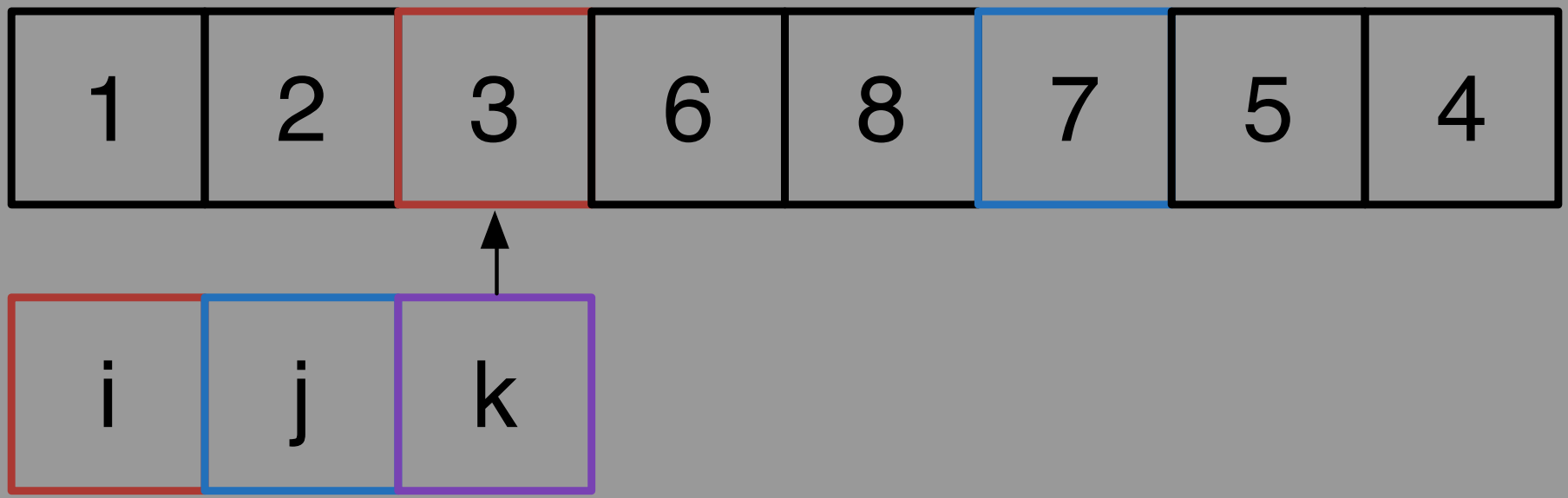
Selection Sort Example



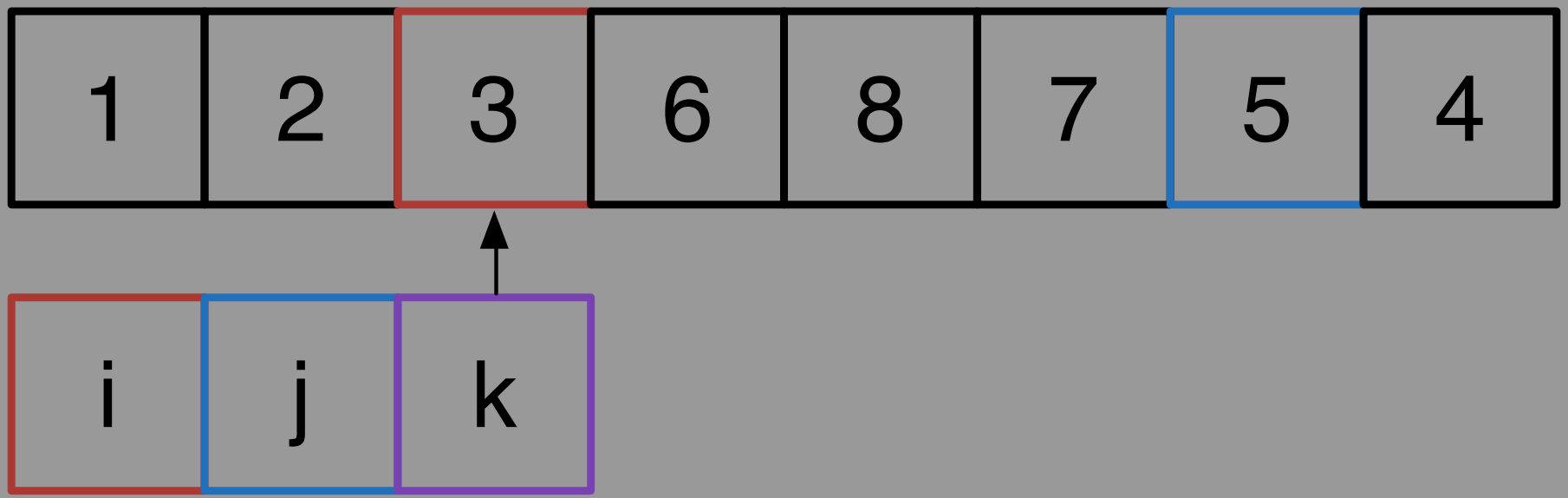
Selection Sort Example



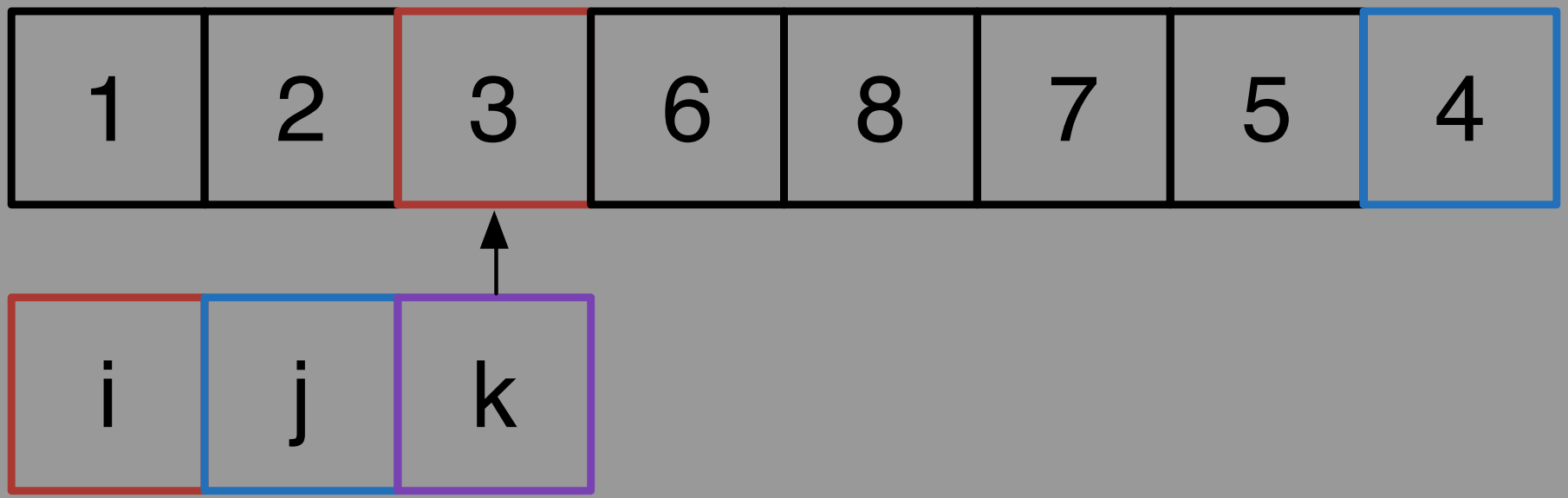
Selection Sort Example



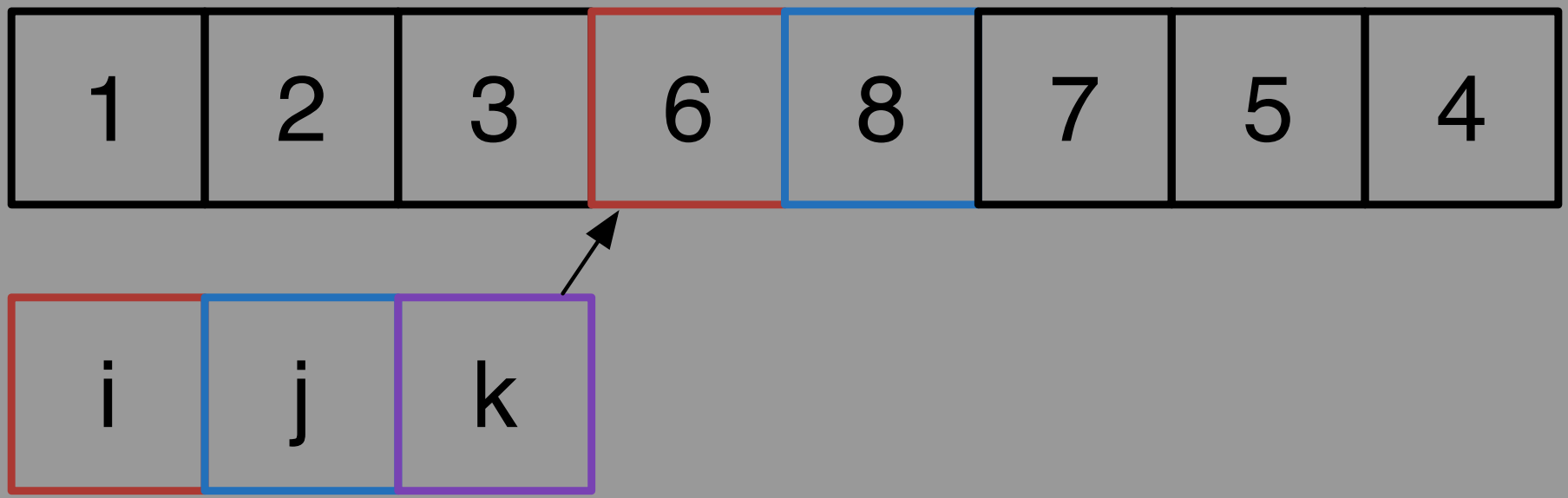
Selection Sort Example



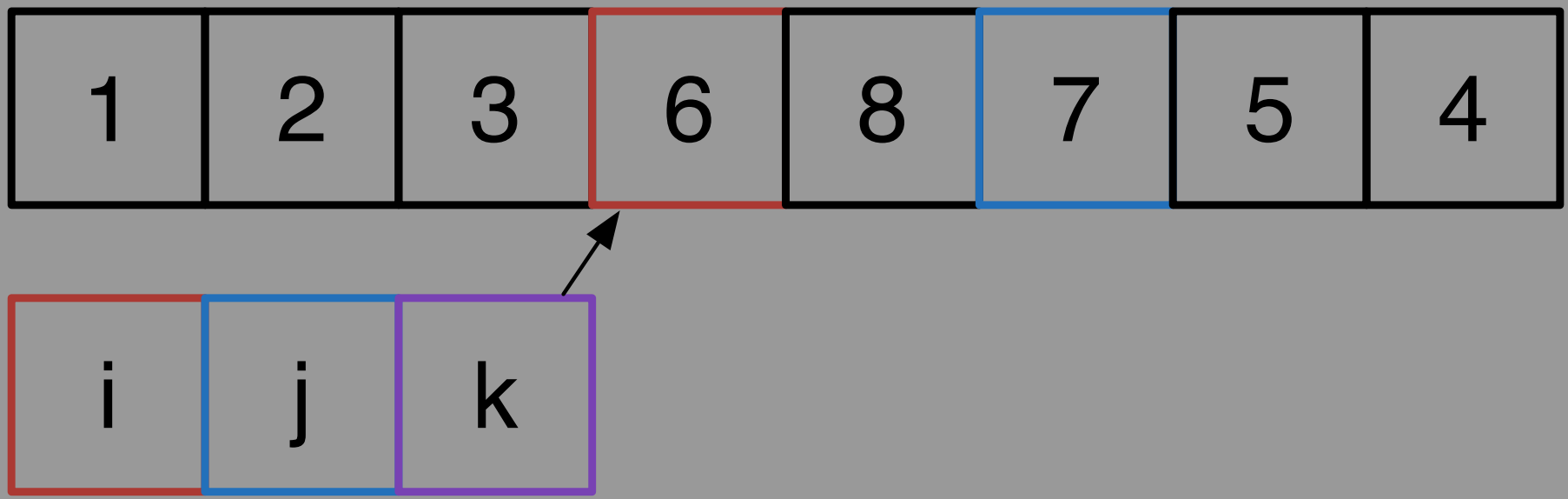
Selection Sort Example



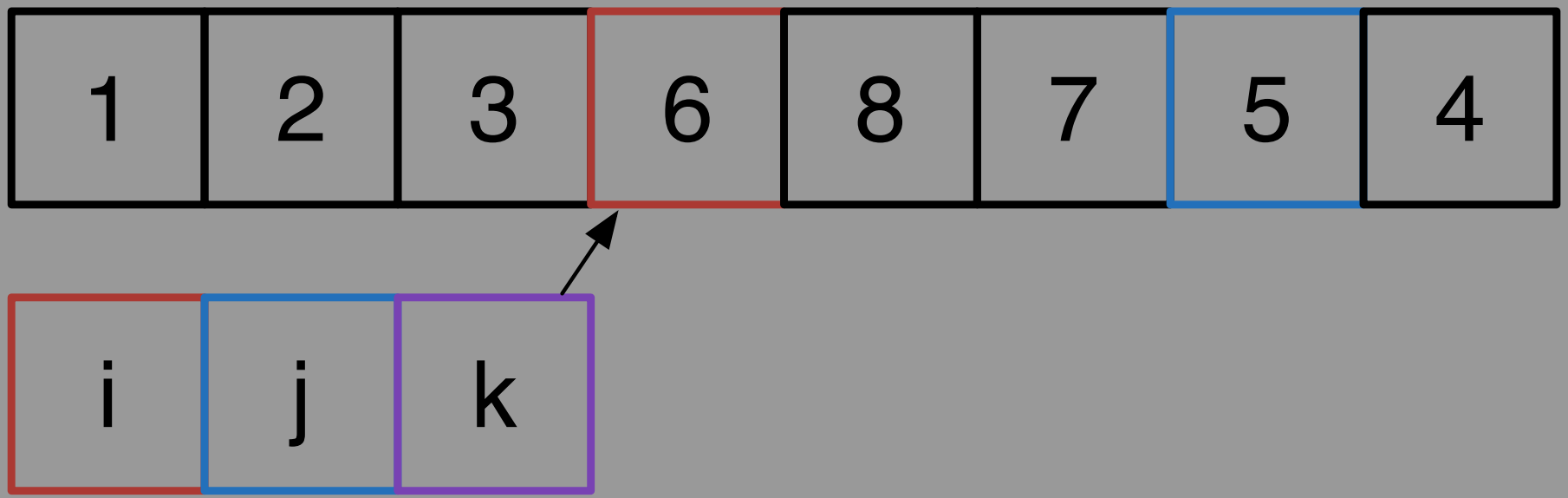
Selection Sort Example



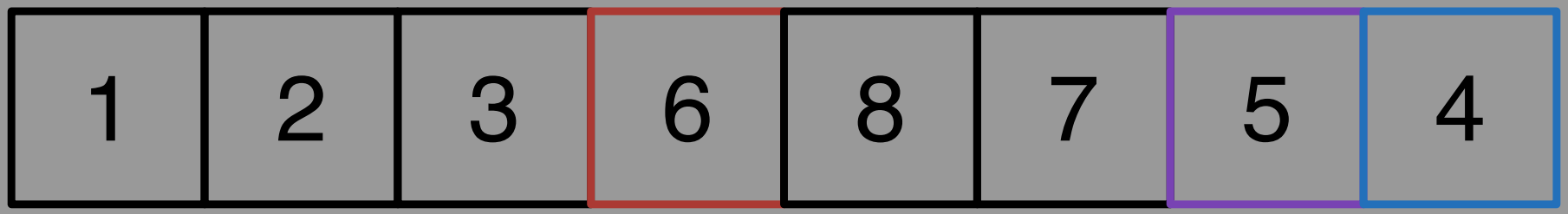
Selection Sort Example

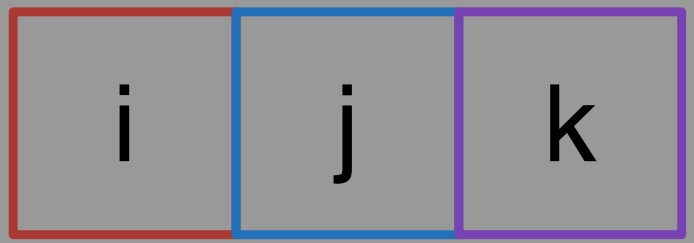


Selection Sort Example

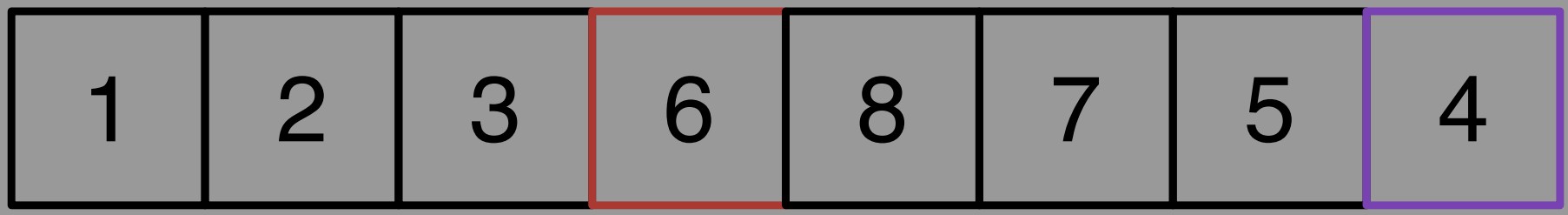


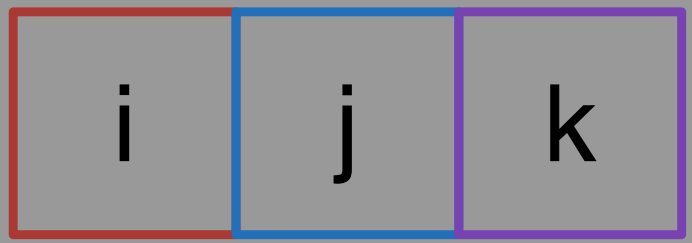
Selection Sort Example



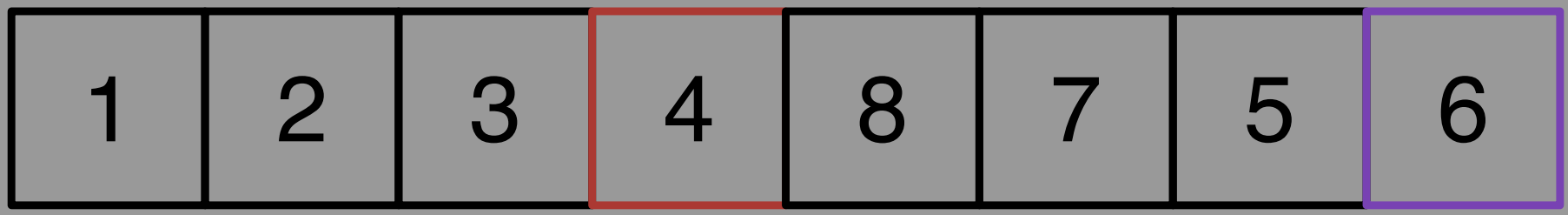


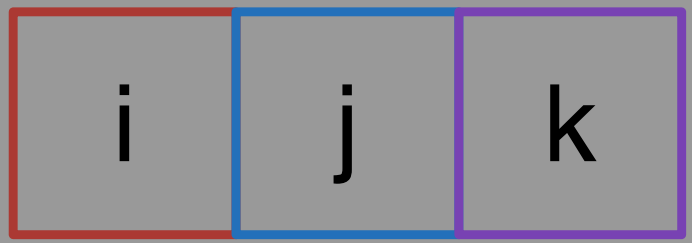
Selection Sort Example



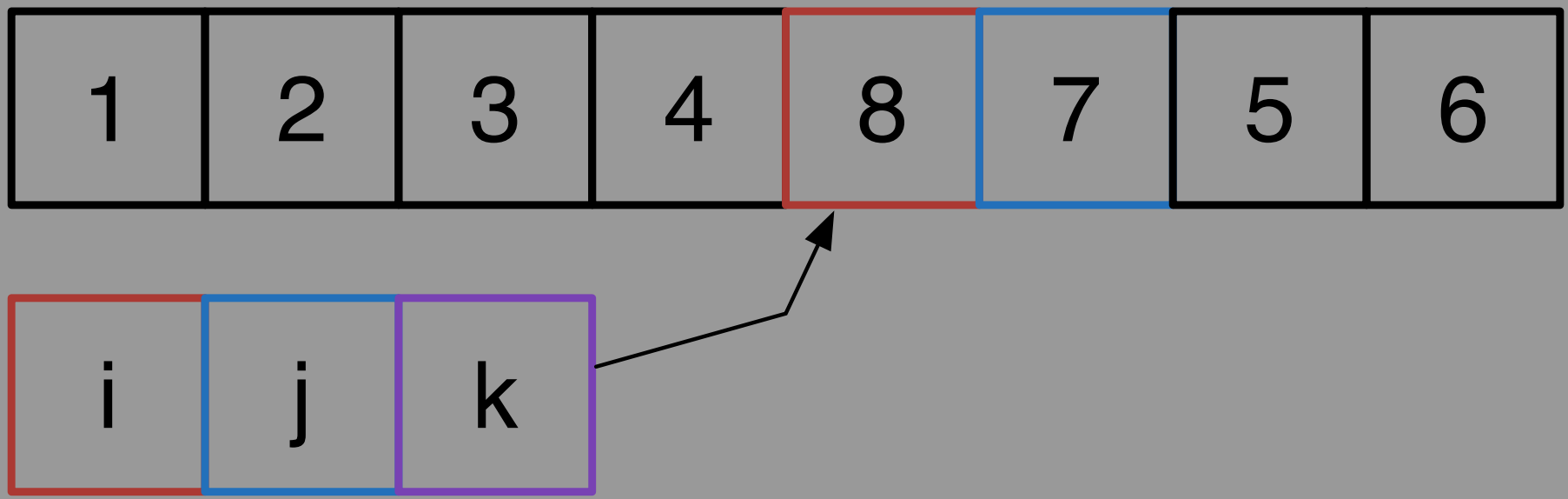


Selection Sort Example

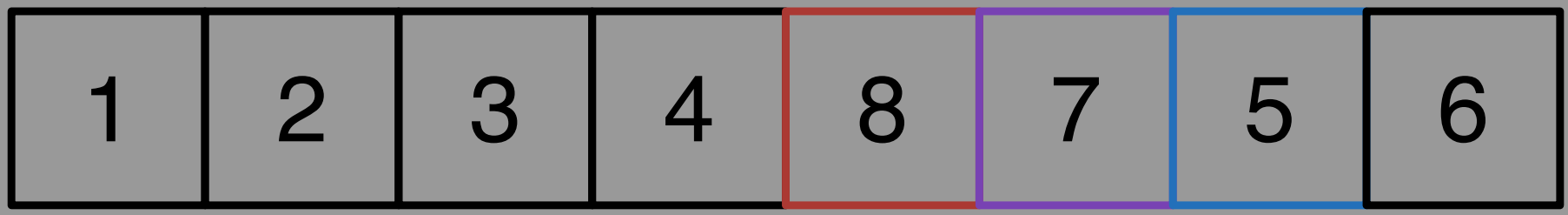


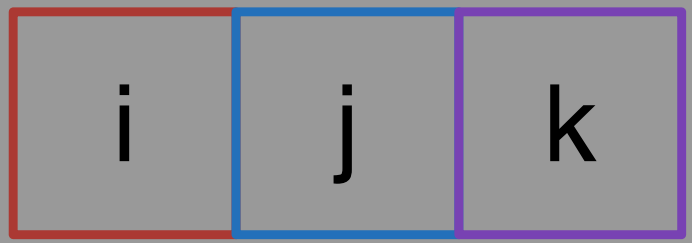


Selection Sort Example

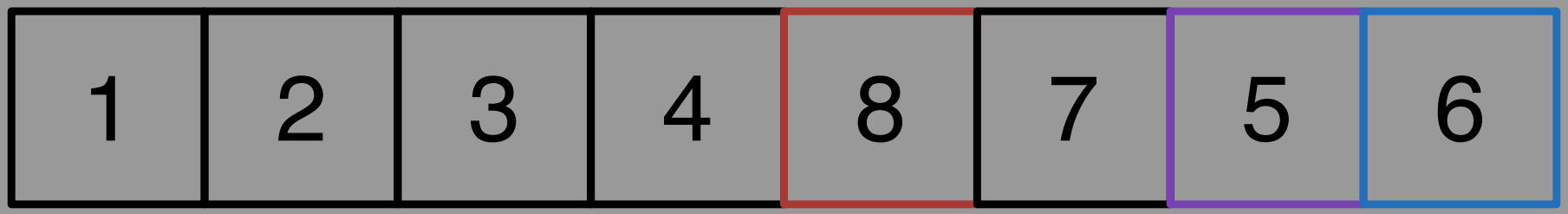


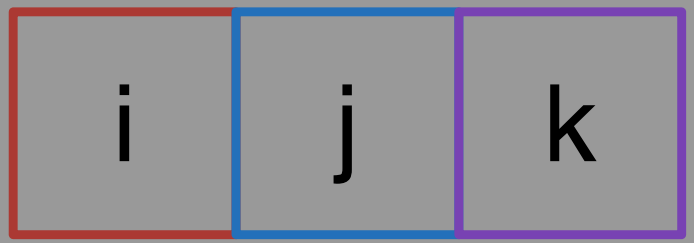
Selection Sort Example



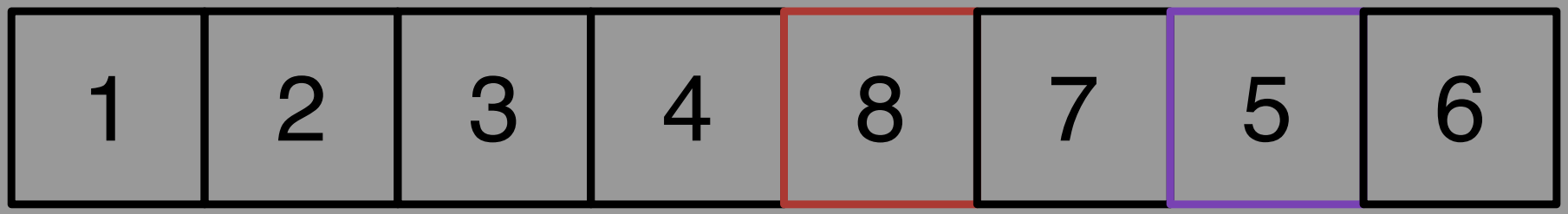


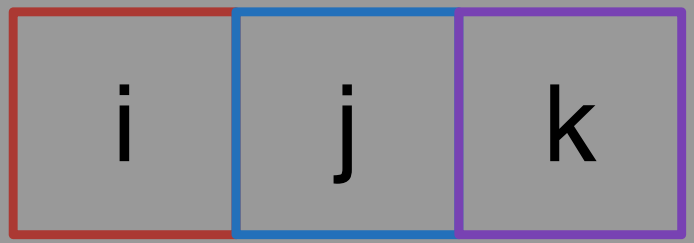
Selection Sort Example



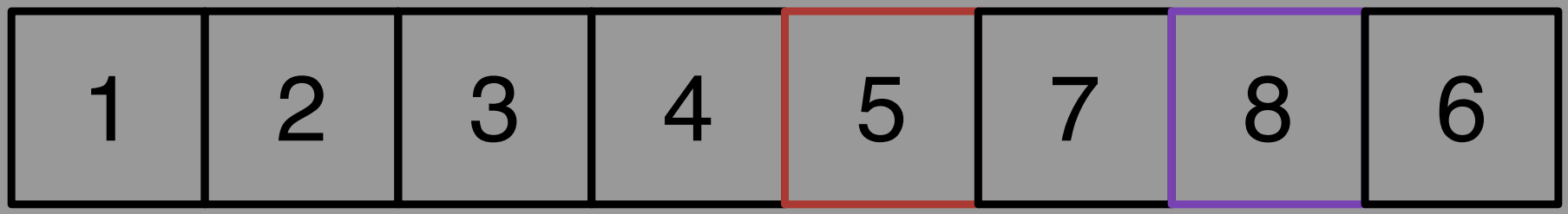


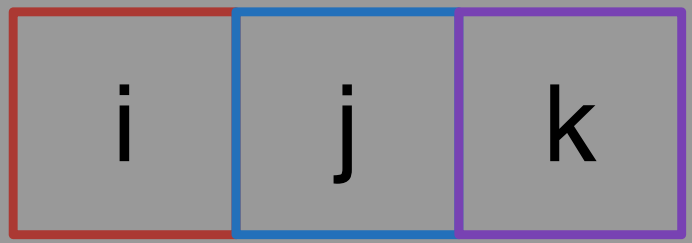
Selection Sort Example



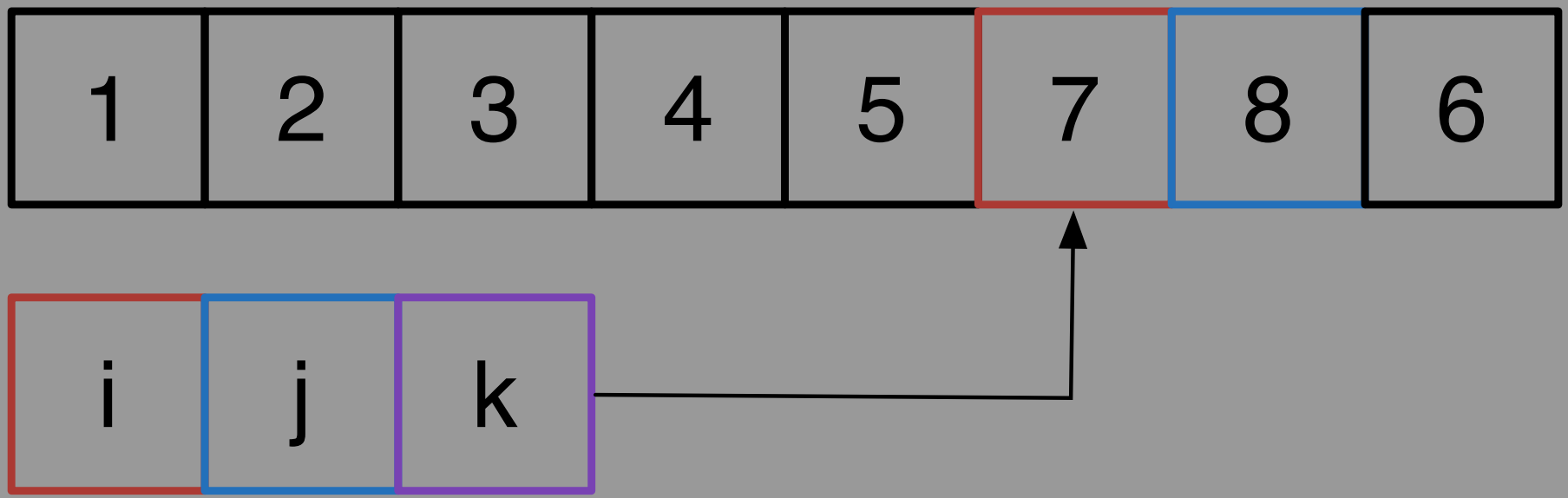


Selection Sort Example

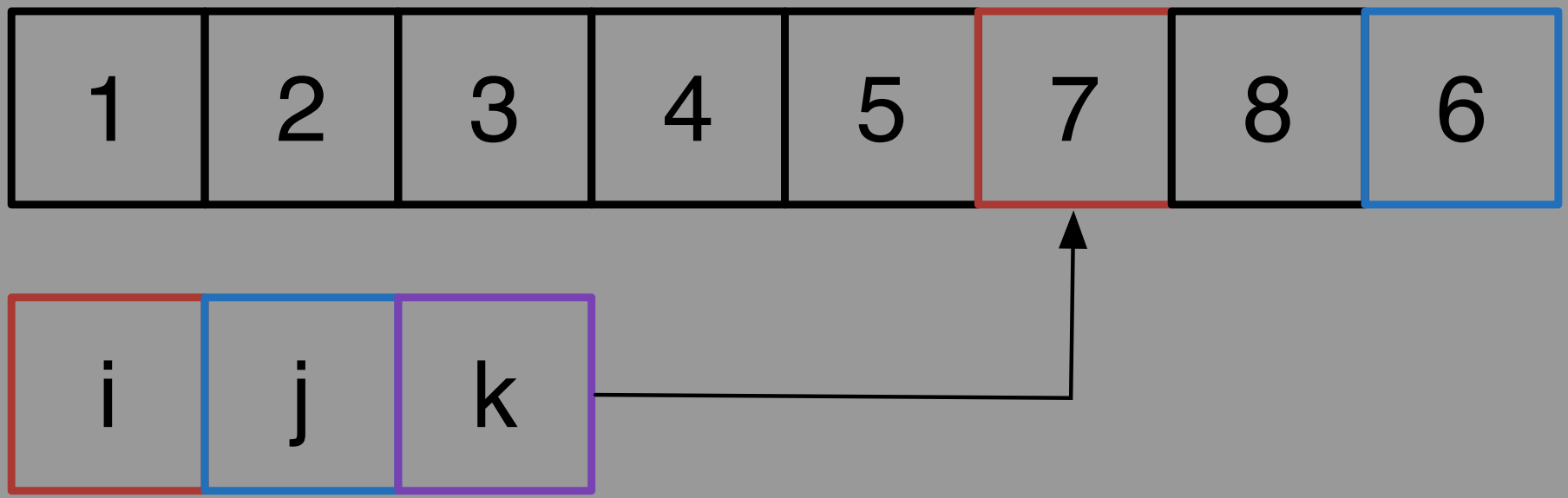




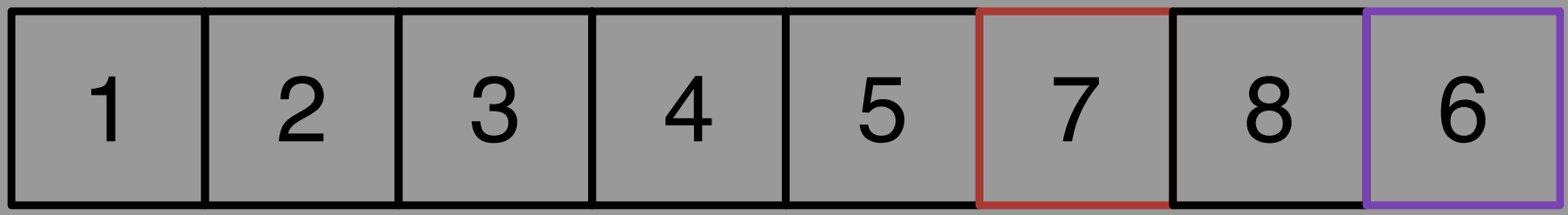
Selection Sort Example

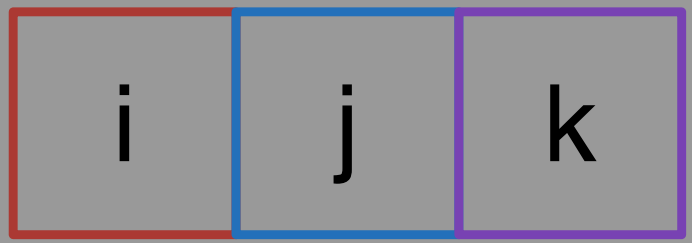


Selection Sort Example

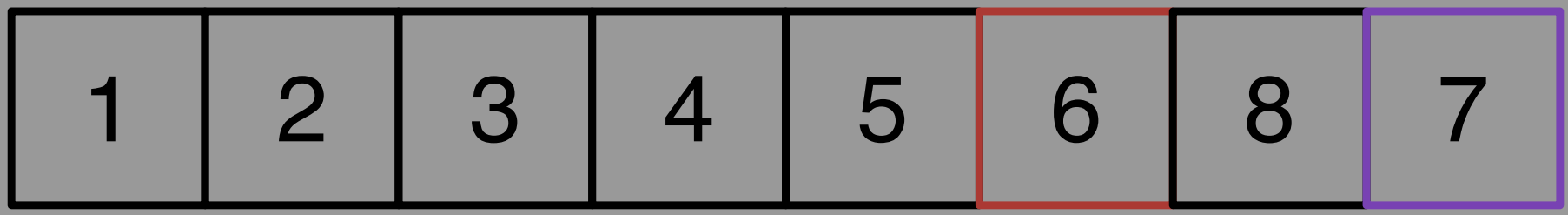


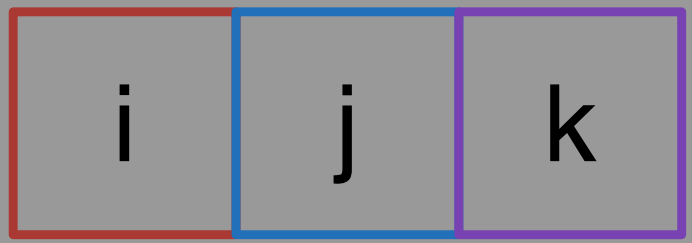
Selection Sort Example





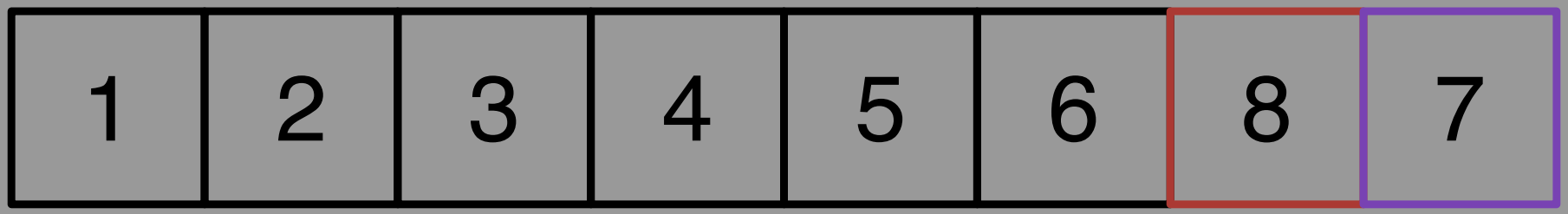
Selection Sort Example

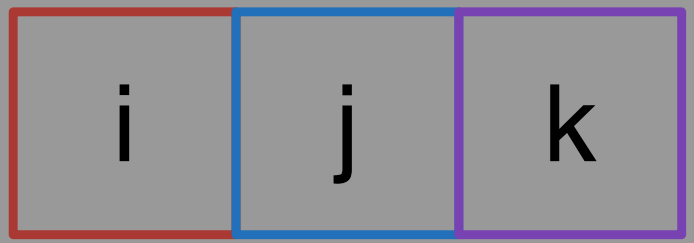




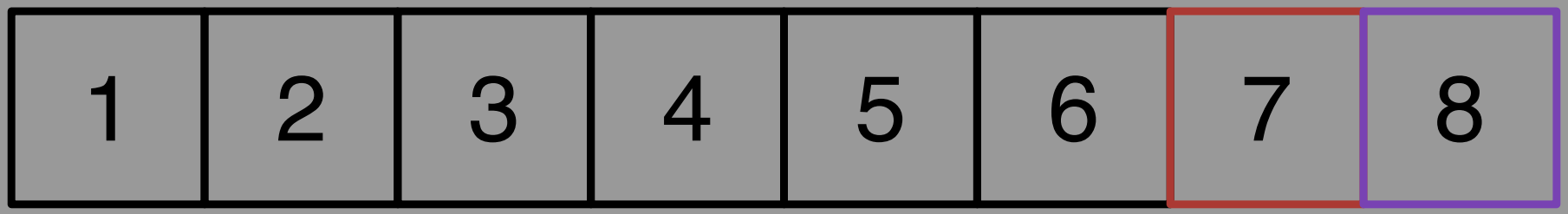
Selection Sort Example

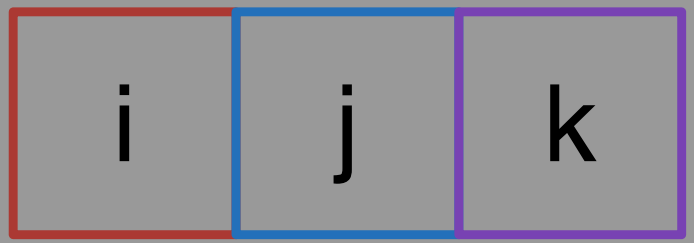
Selection Sort Example





Selection Sort Example





Complexity of Selection Sort

Selection sort is not a very efficient algorithm

- The worst case complexity is $O(n^2)$
- The best case complexity is $O(n^2)$

This is usually the case when we have nested loops that iterate over the length of an array

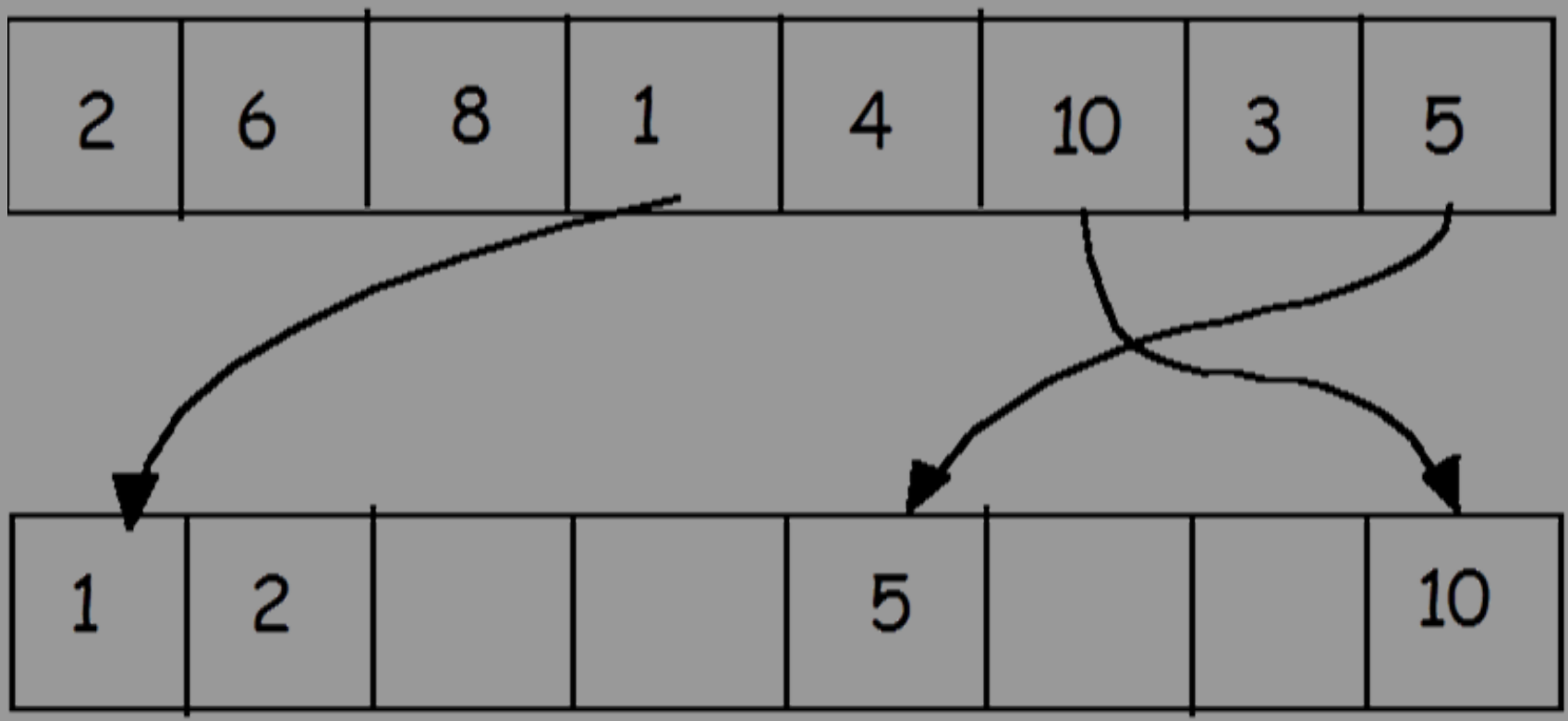
- $O(n^2)$ is not a very good solution
- The running time will be very long for large values of n

Table of Contents

Rank Sort

Lets look at the rank sort algorithm

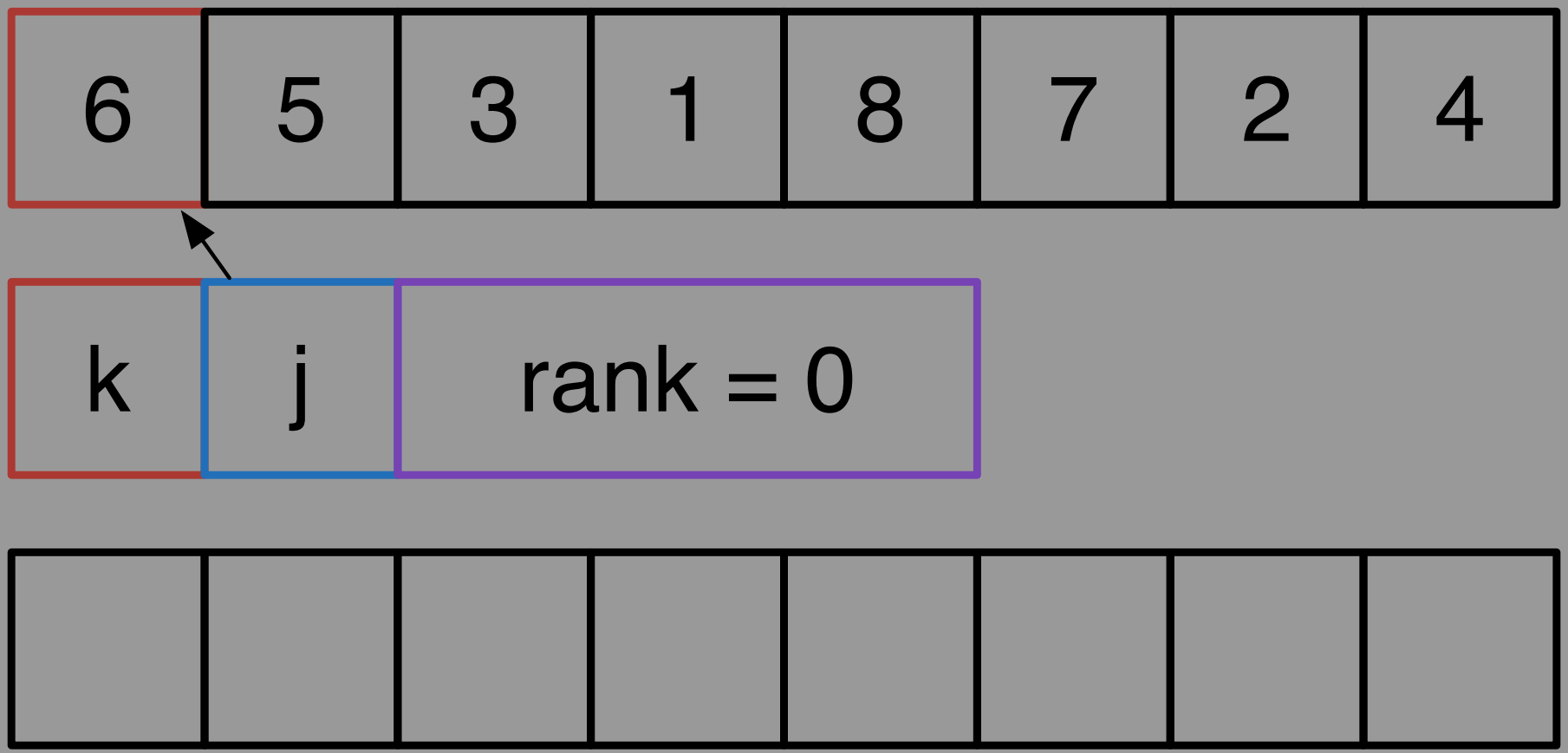
- The idea is that for every element in the array we count the number of elements smaller than it
- We than place the element into a new array in the correct position



Rank Sort

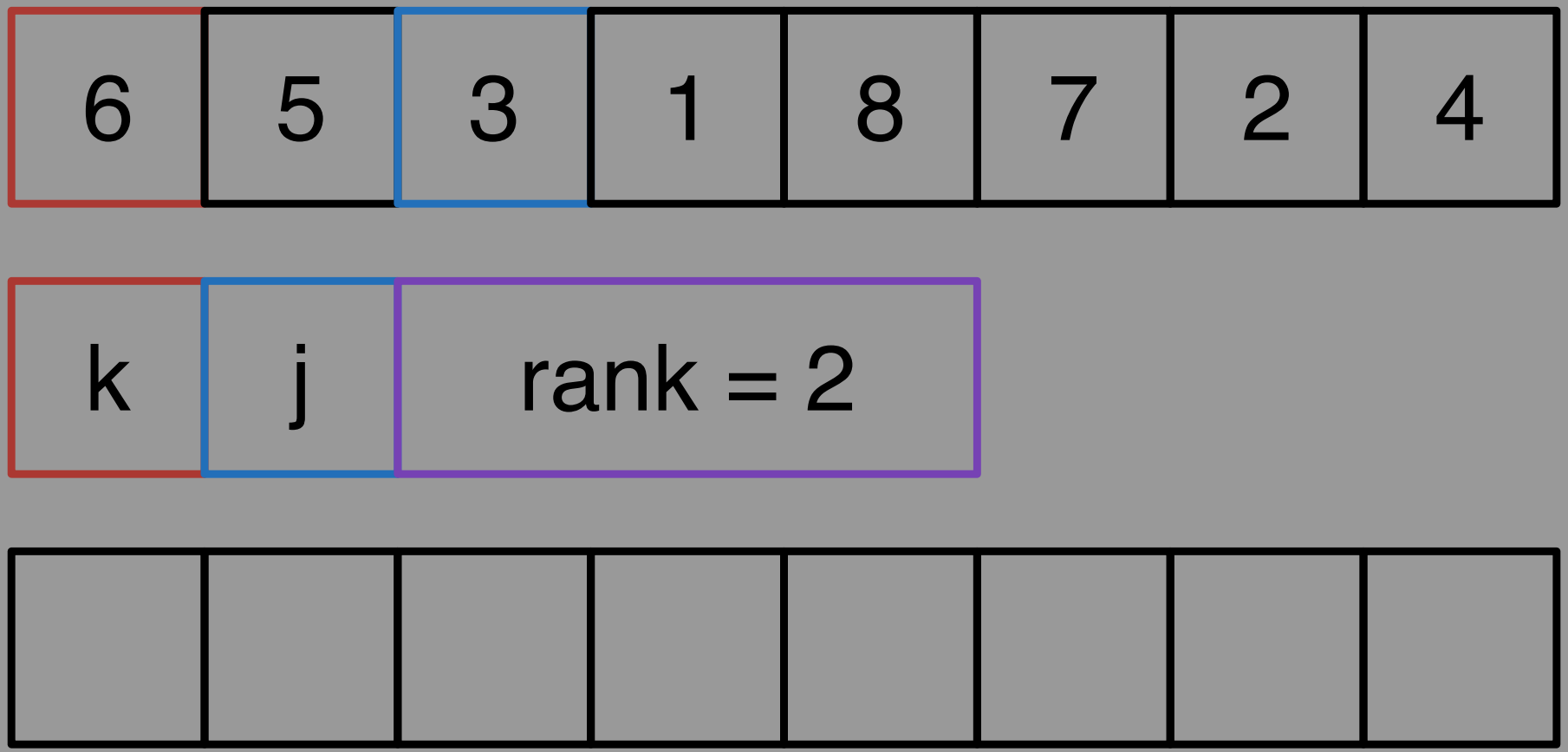
```
1  int* rankSort(int b[], int n) {
2     int* c = malloc(sizeof(int)*n);
3     int k, j;
4     for (k = 0; k < n; k++) {
5        int rank = 0;
6        for (j = 0; j < n; j++ ) {
7           if (b[j] < b[k])  rank++;
8        }
9        c[rank] = b[k];
10    }
11    return c;
12 }
```

Rank Sort Example

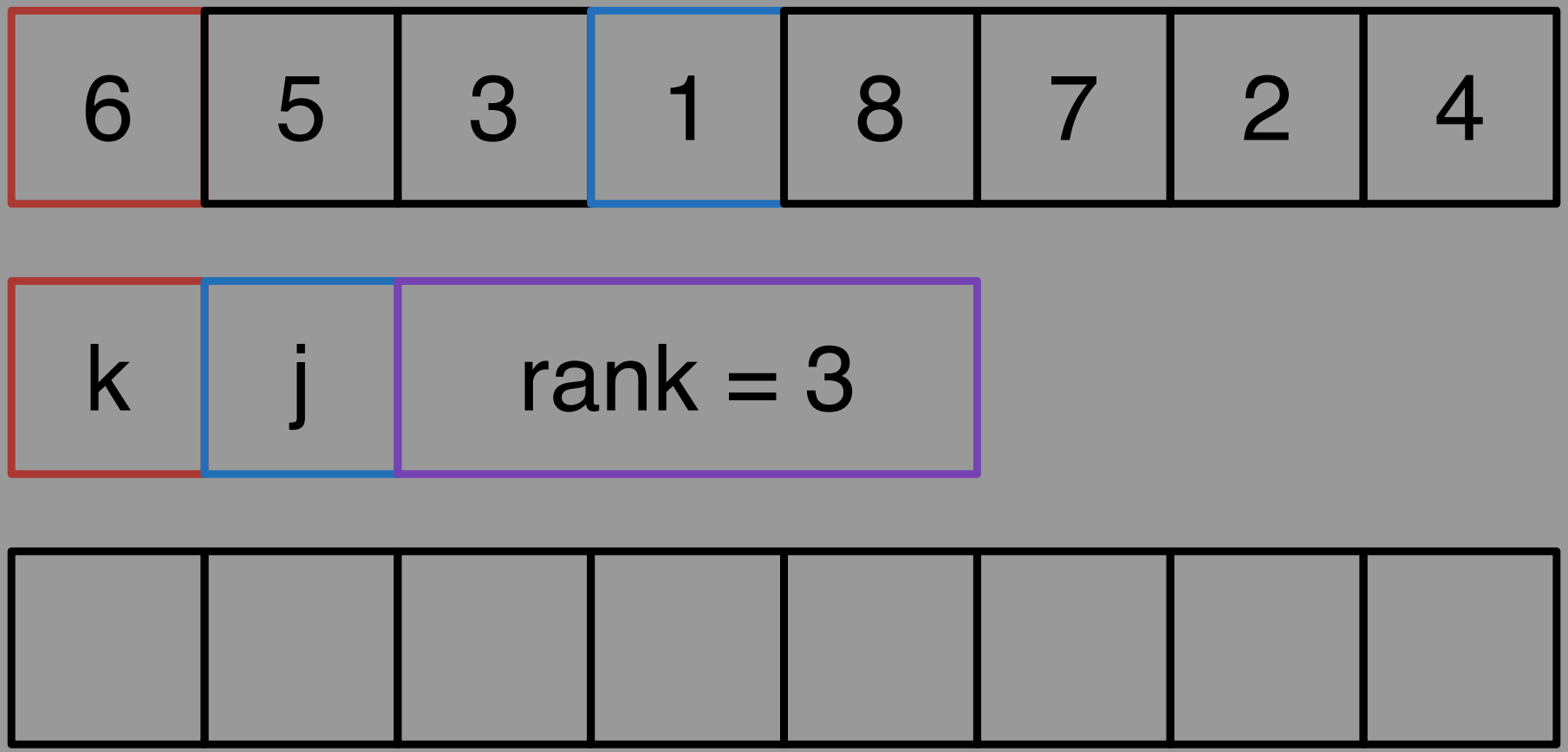


Rank Sort Example

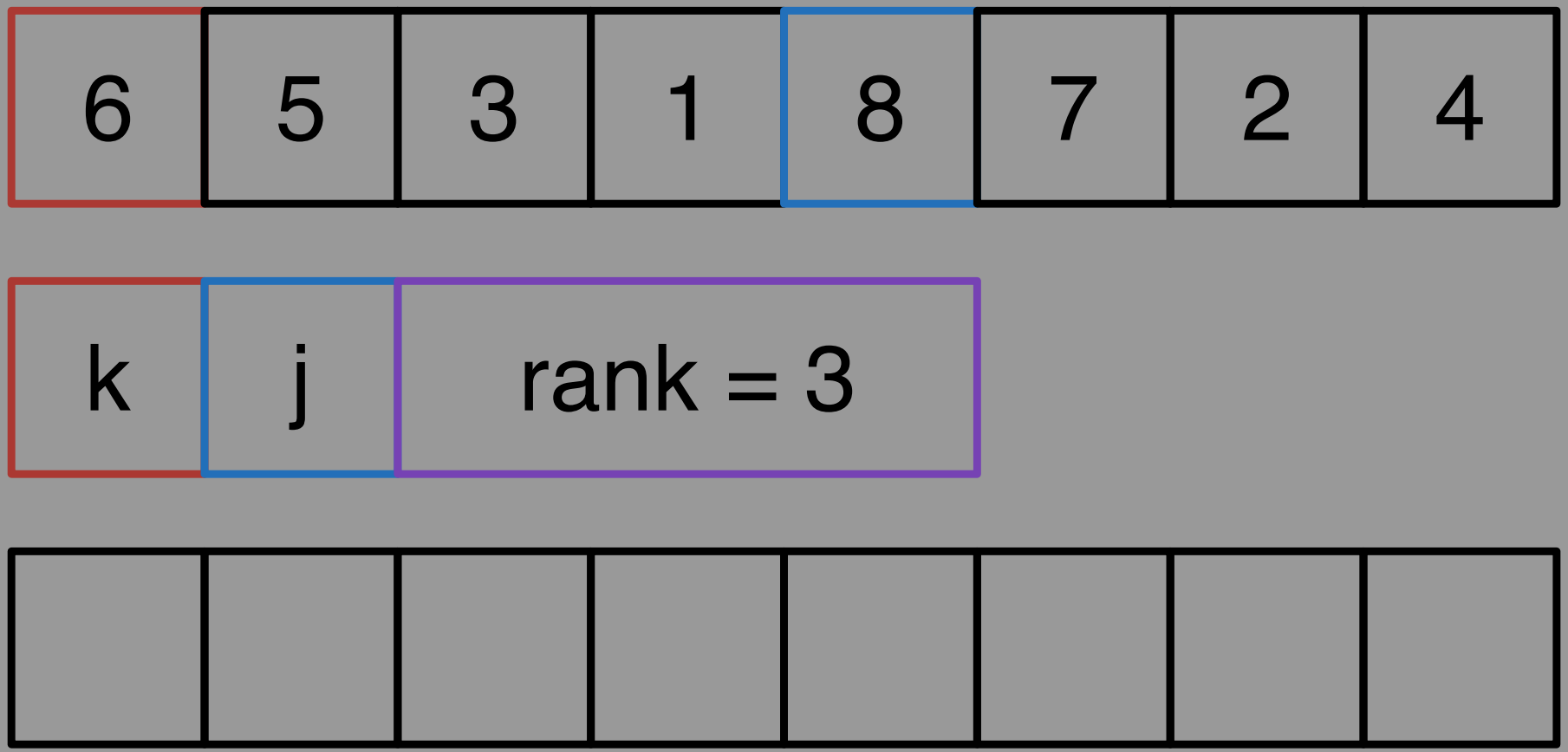
Rank Sort Example



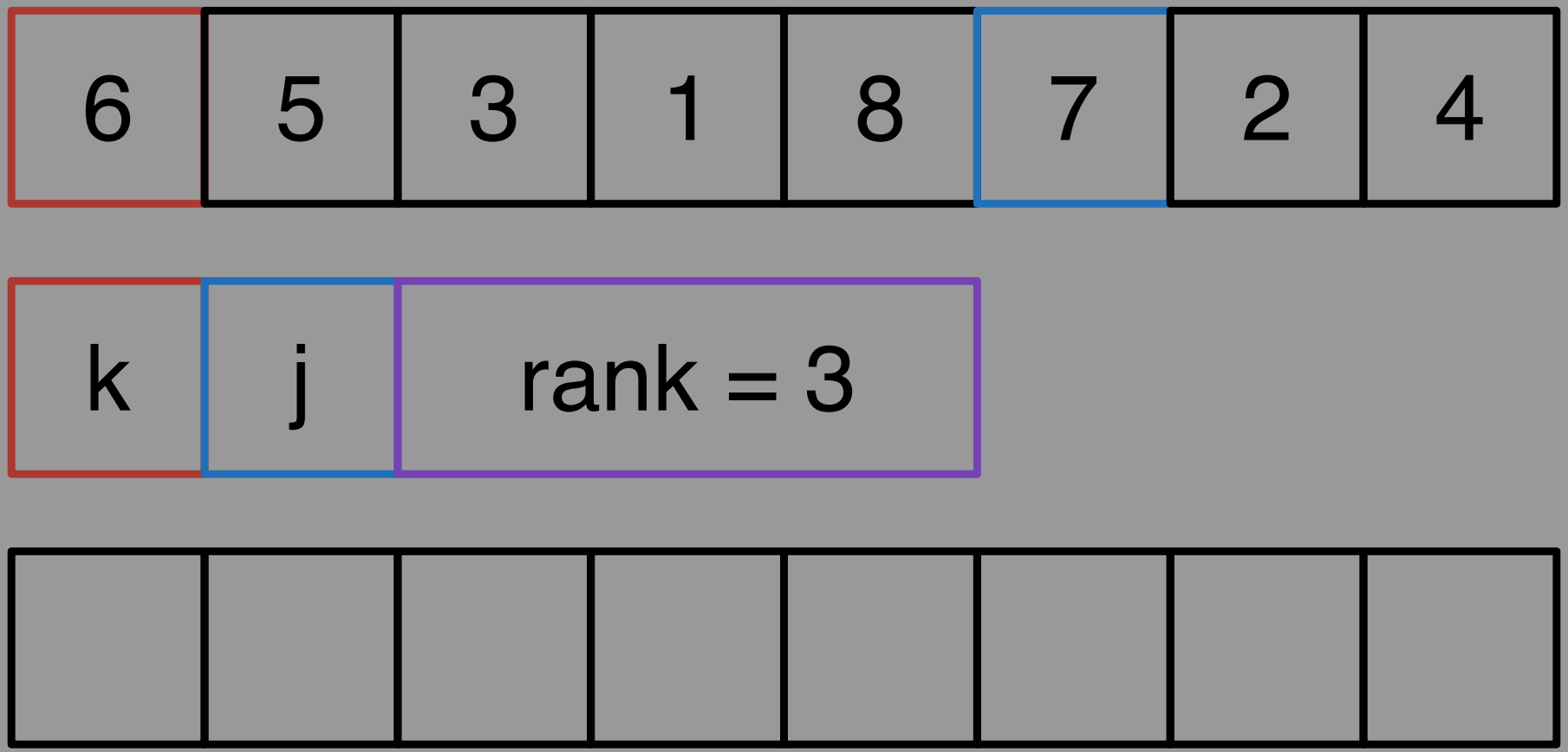
Rank Sort Example



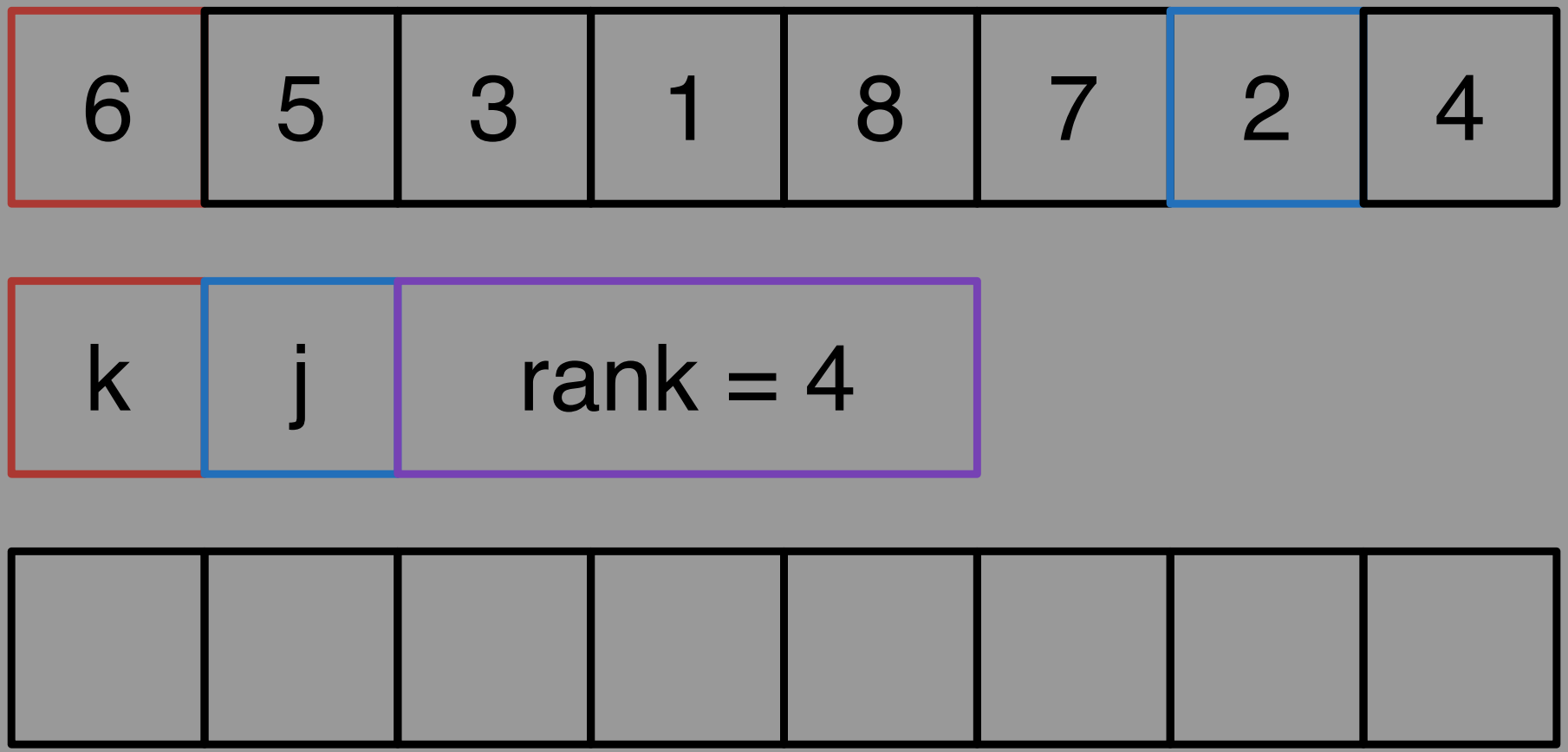
Rank Sort Example



Rank Sort Example



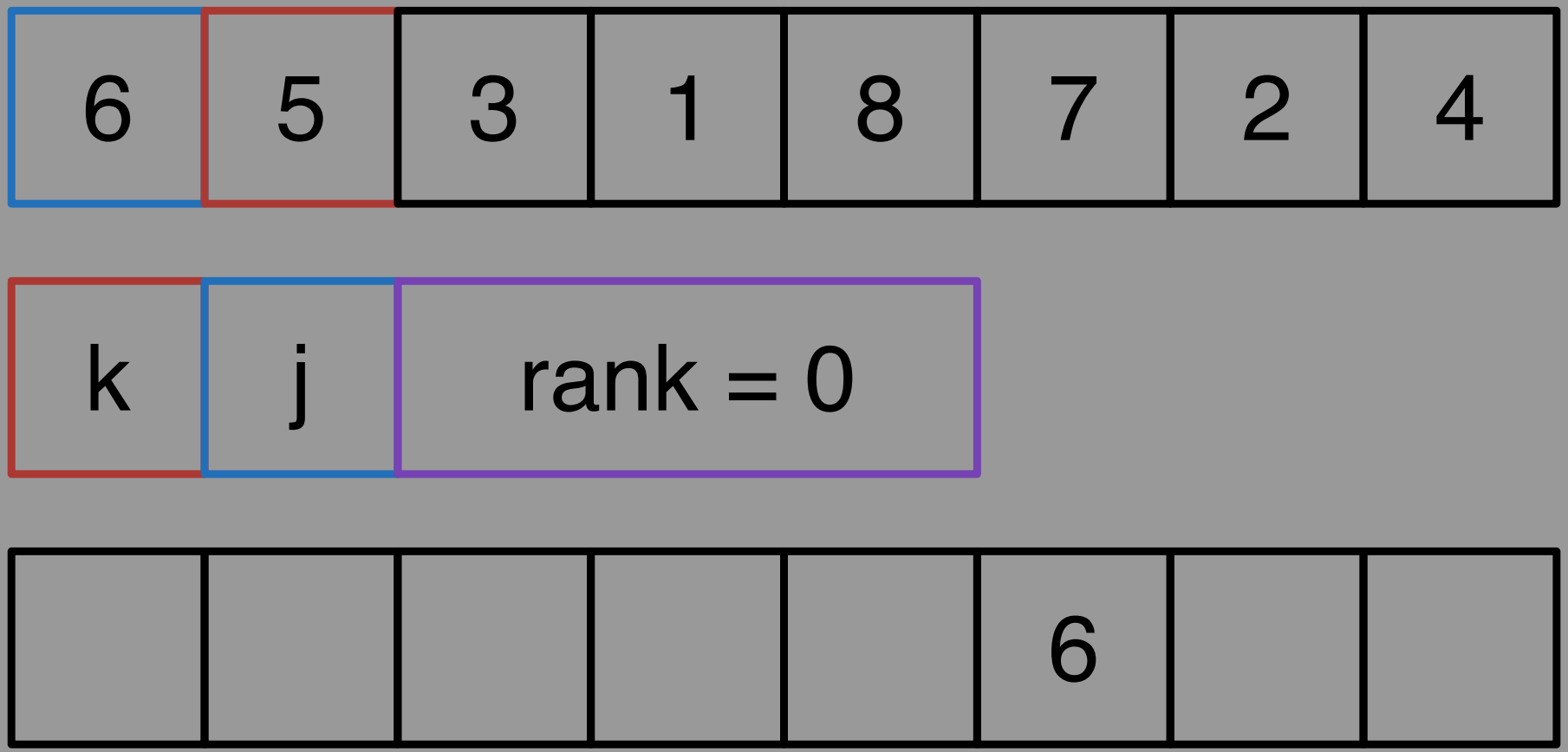
Rank Sort Example



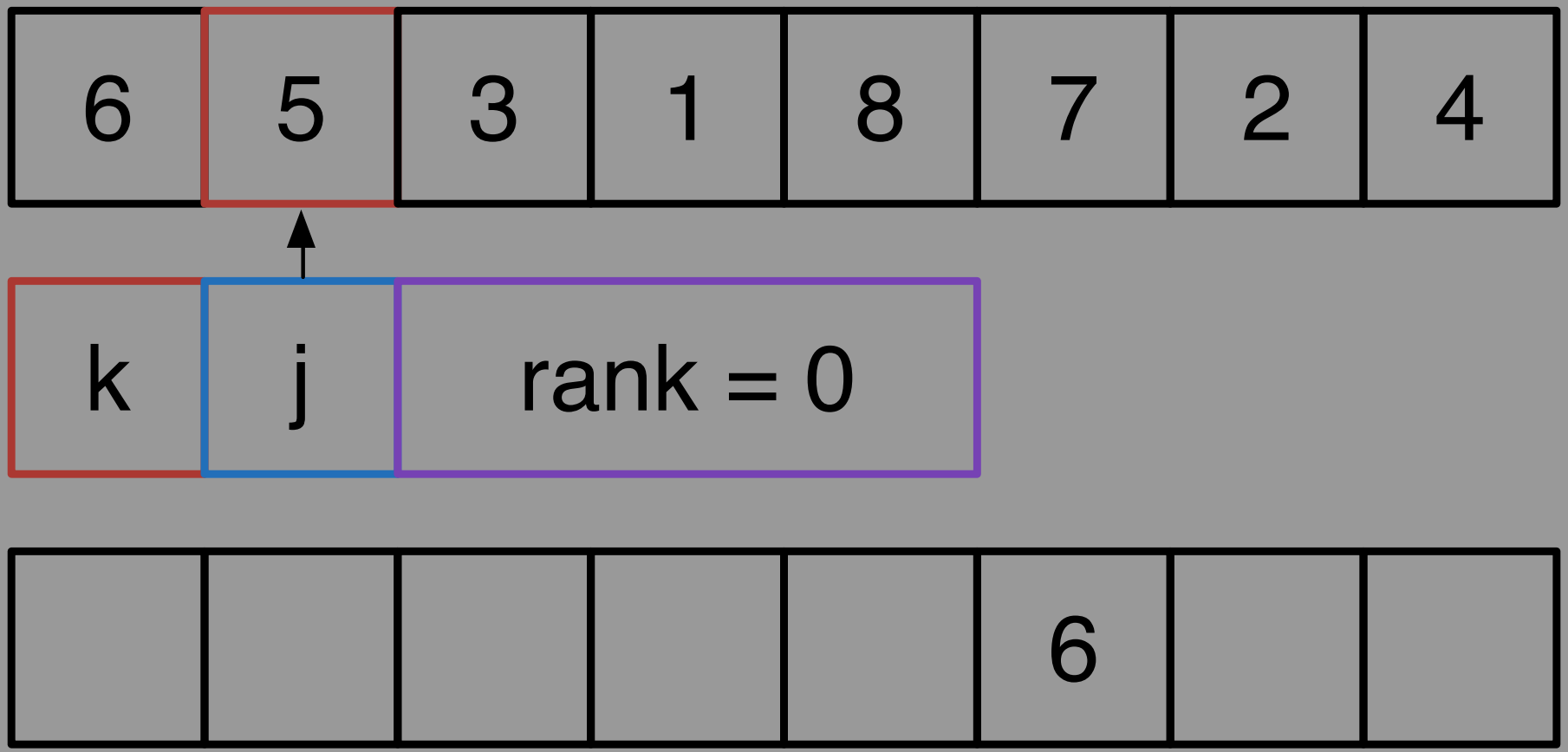
Rank Sort Example

Rank Sort Example

Rank Sort Example

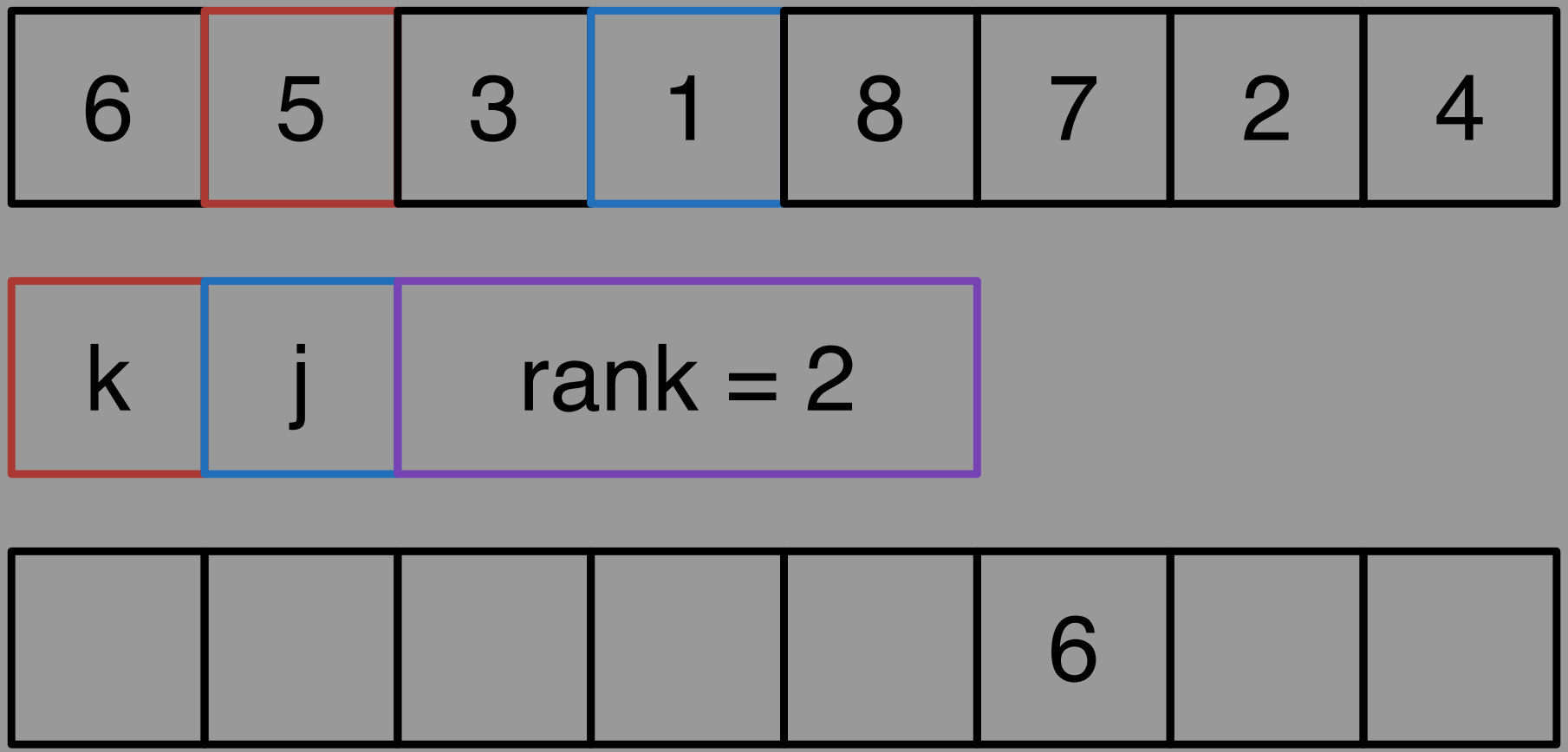


Rank Sort Example

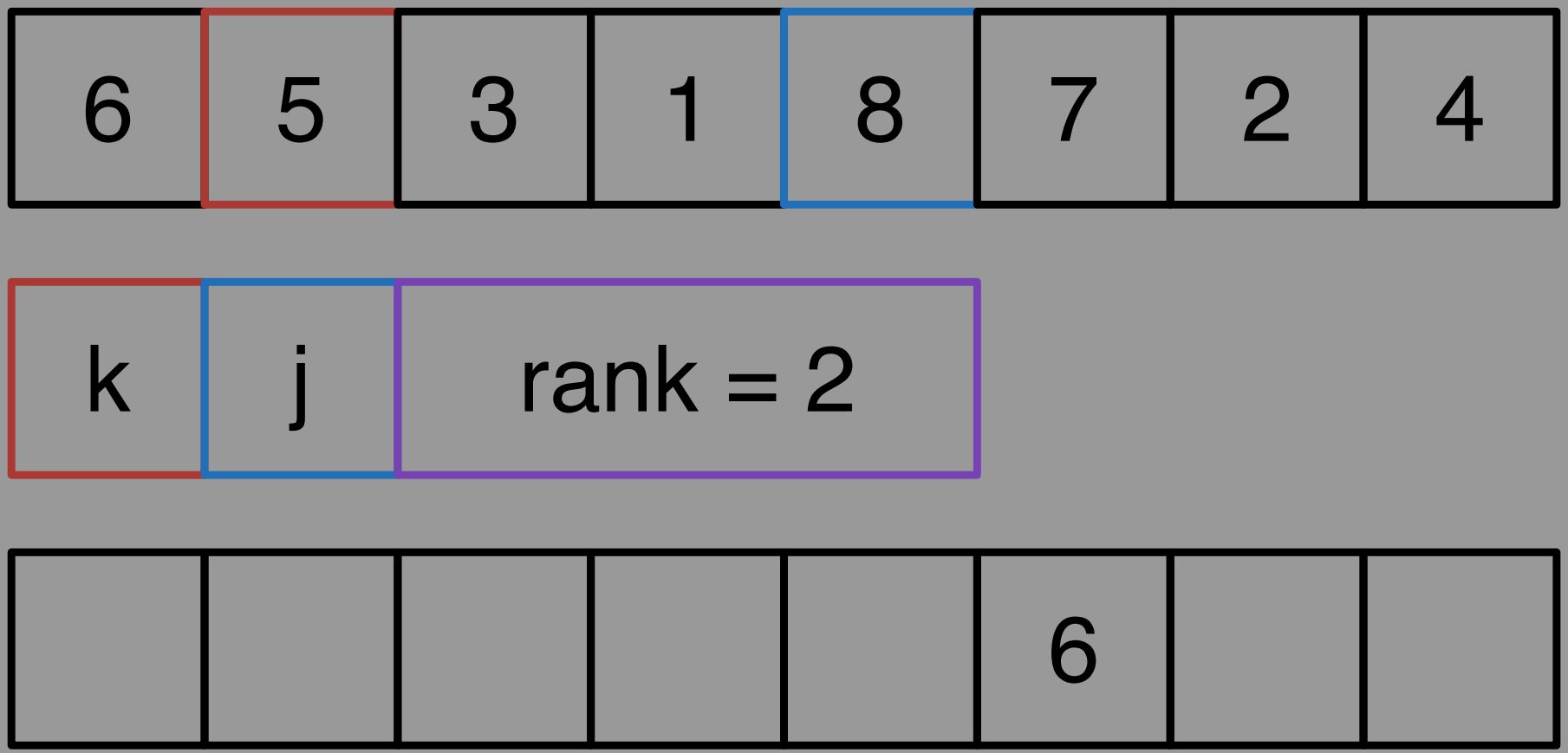


Rank Sort Example

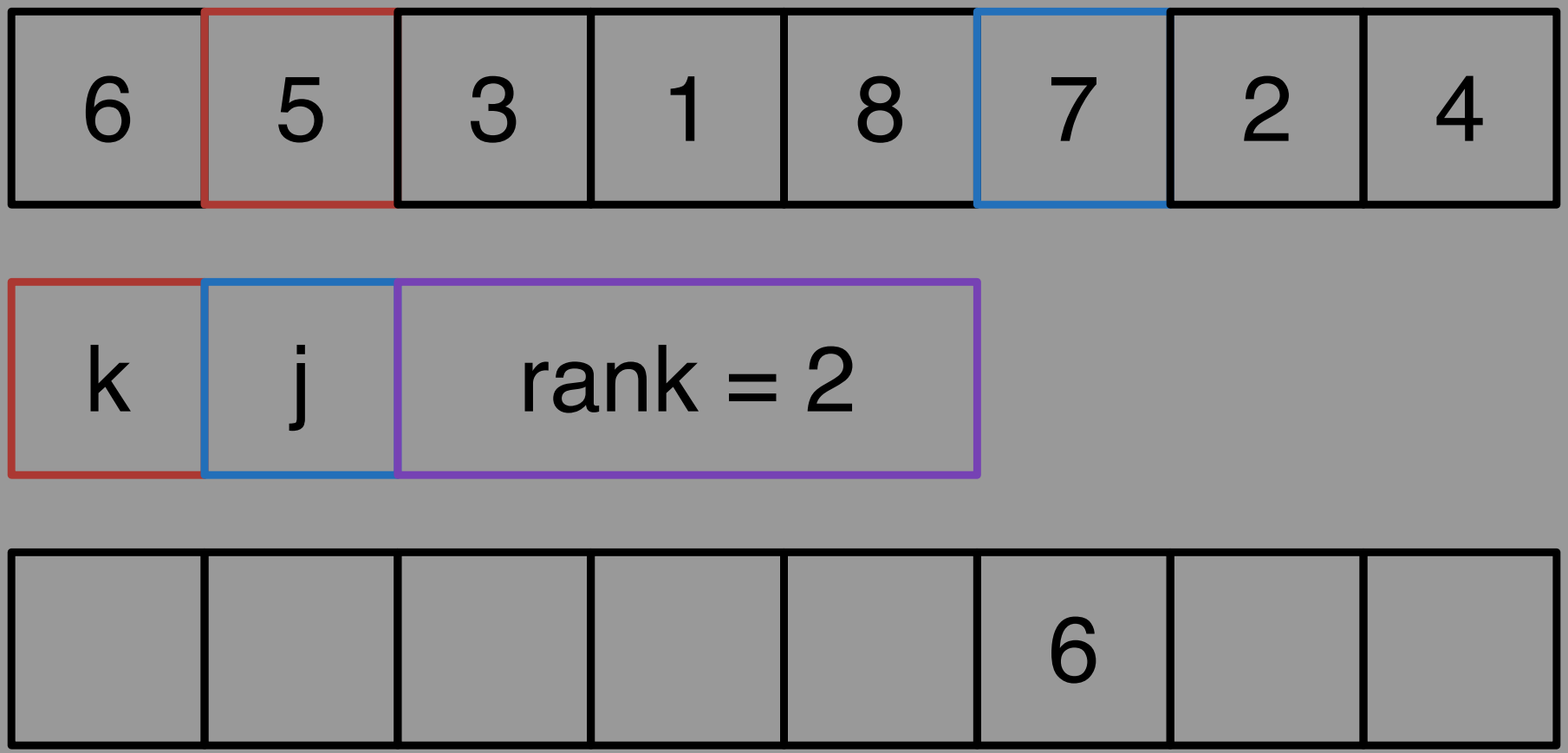
Rank Sort Example



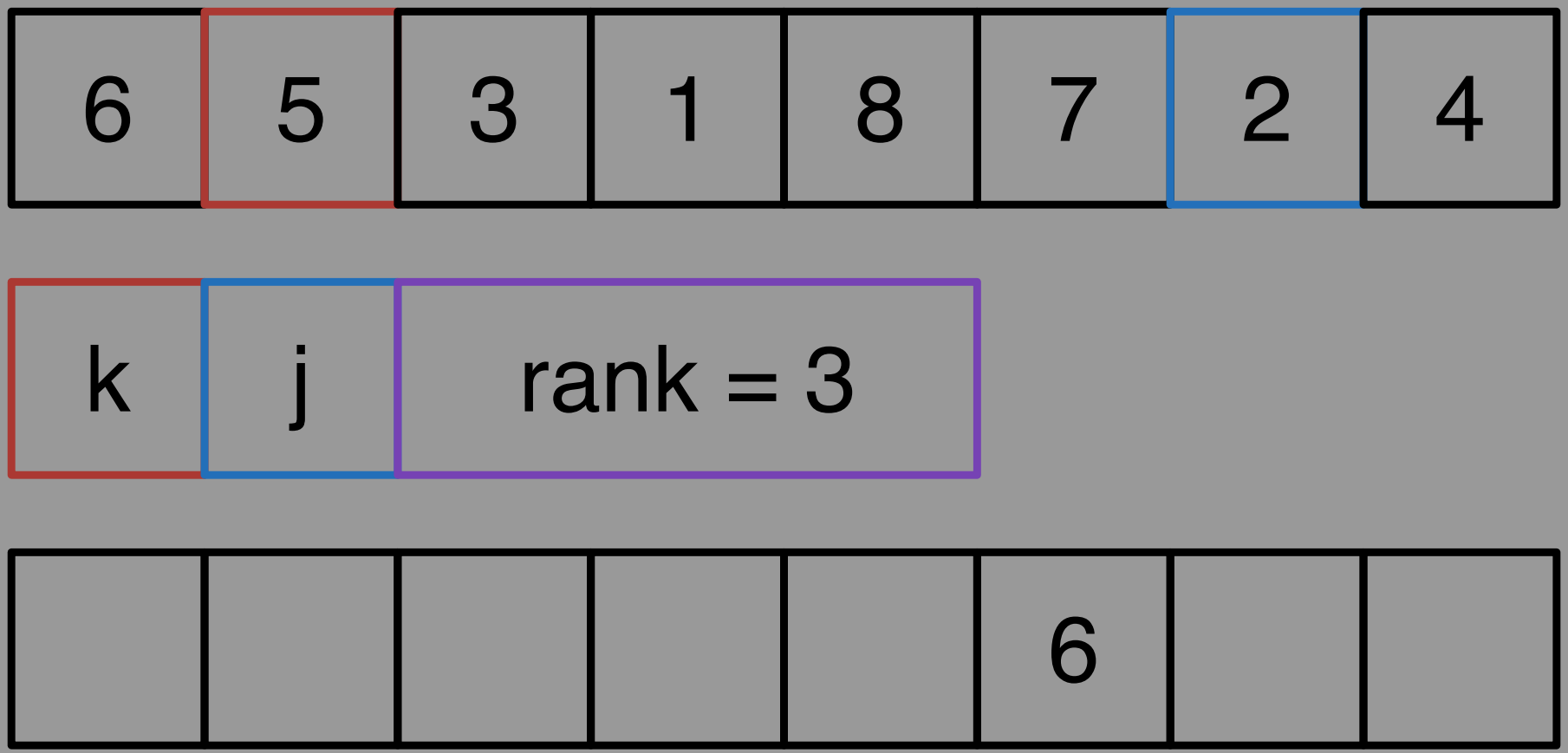
Rank Sort Example



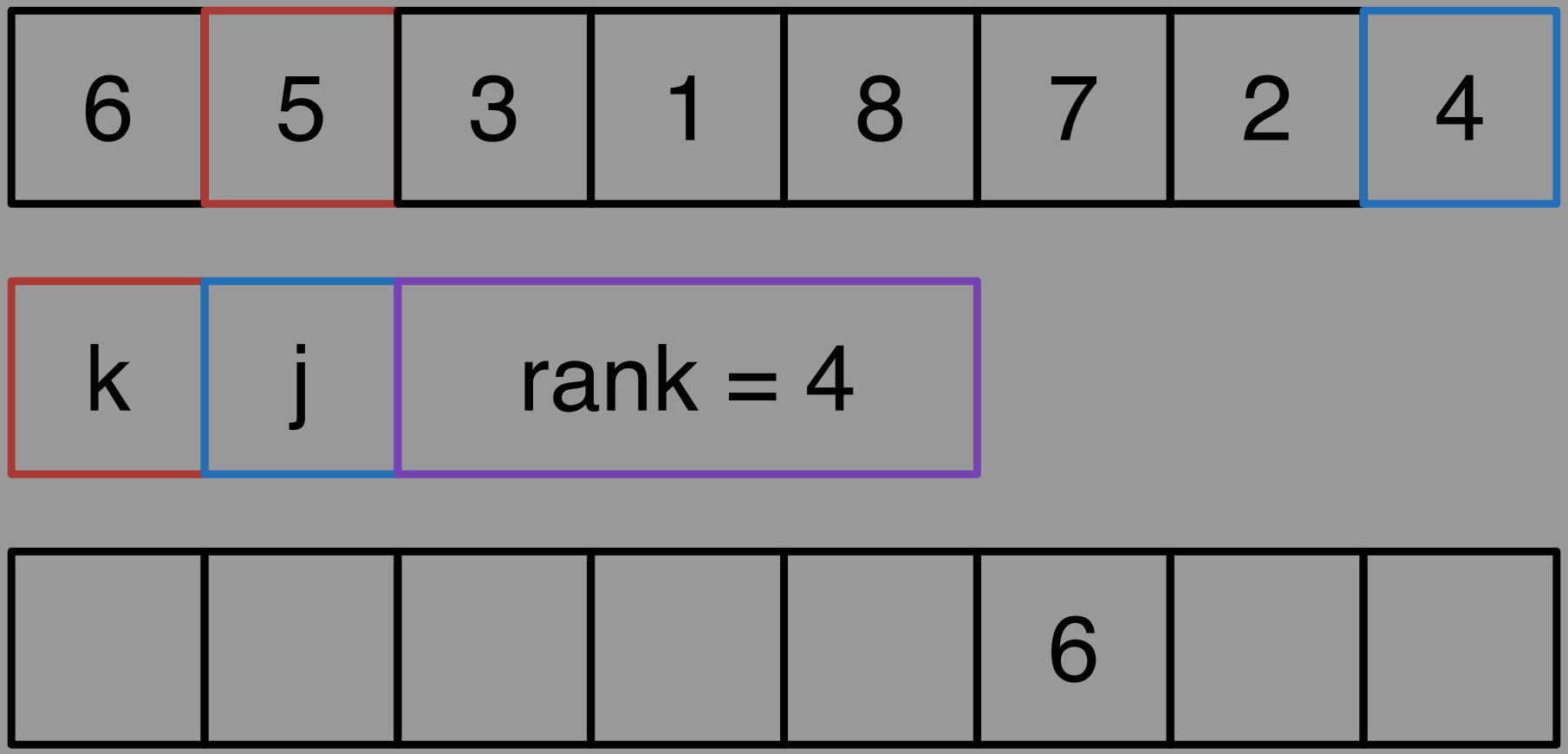
Rank Sort Example



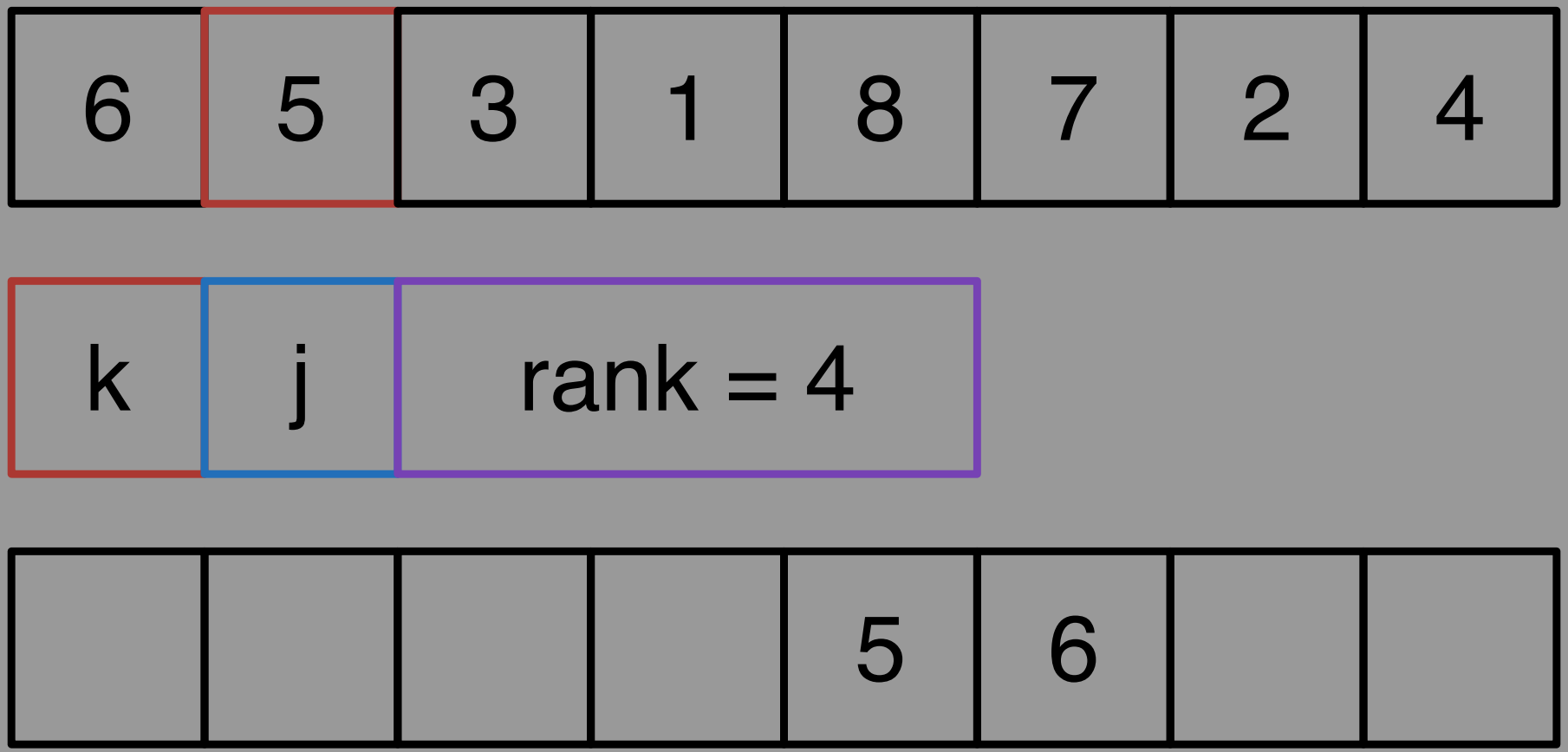
Rank Sort Example



Rank Sort Example



Rank Sort Example



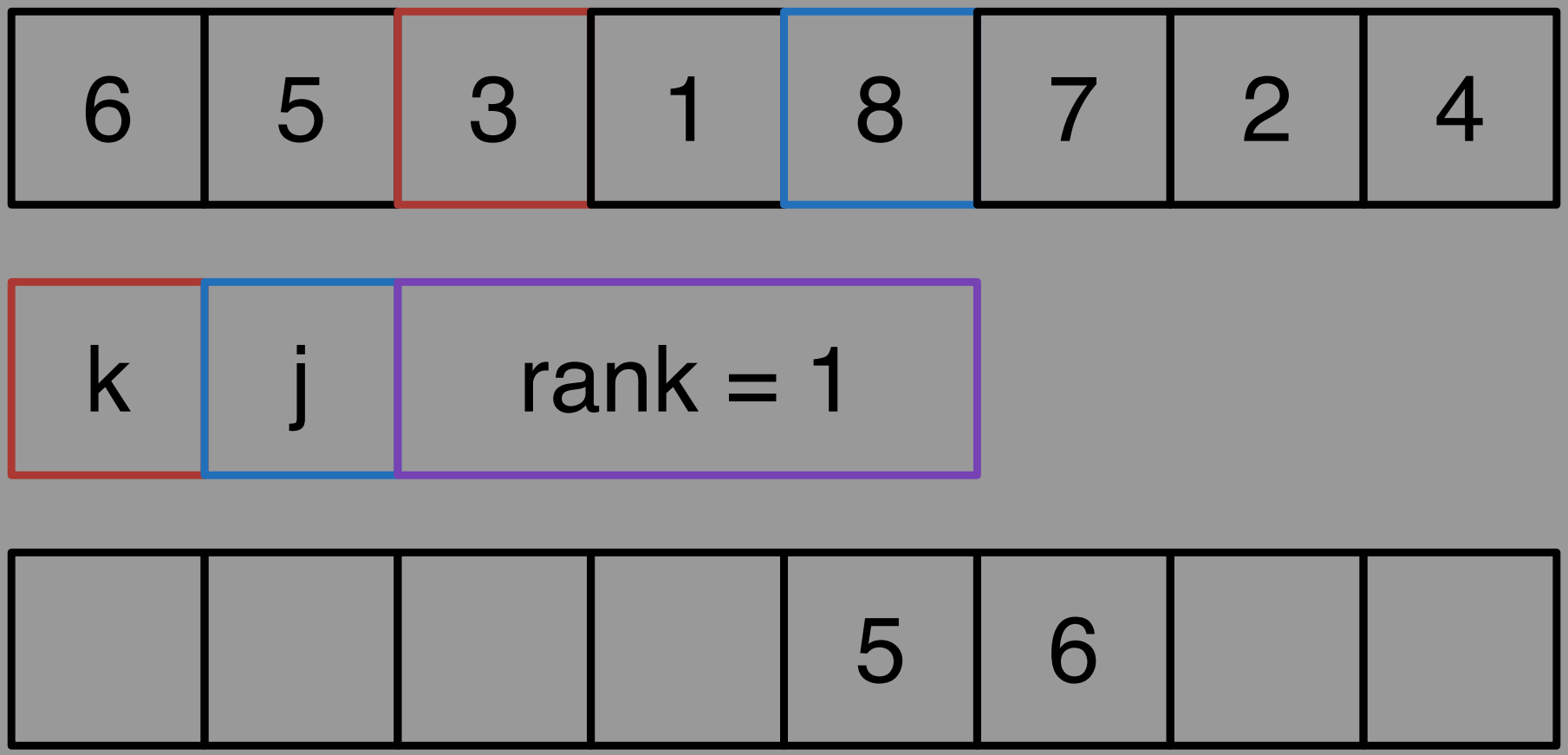
Rank Sort Example

Rank Sort Example

Rank Sort Example

Rank Sort Example

Rank Sort Example



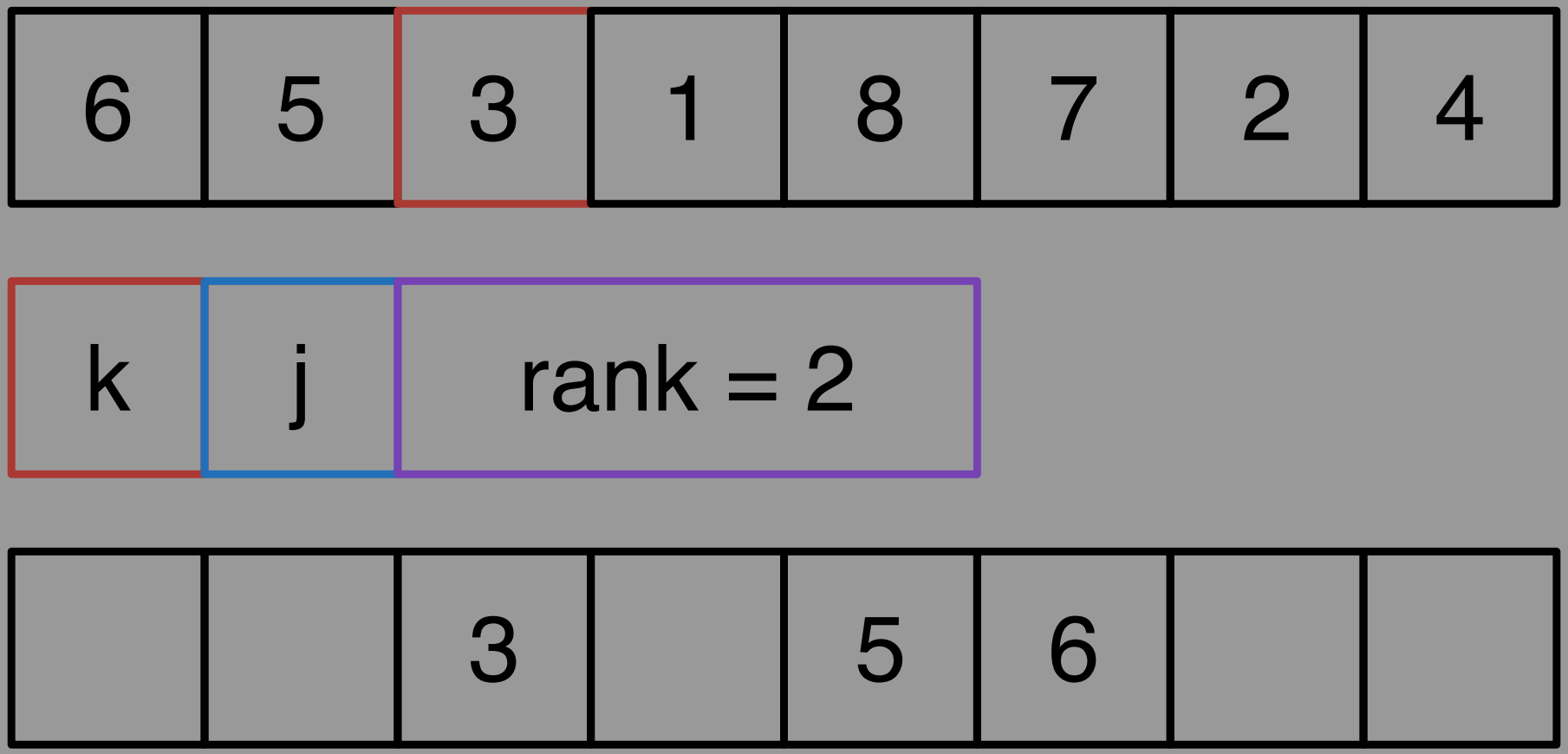
Rank Sort Example

Rank Sort Example

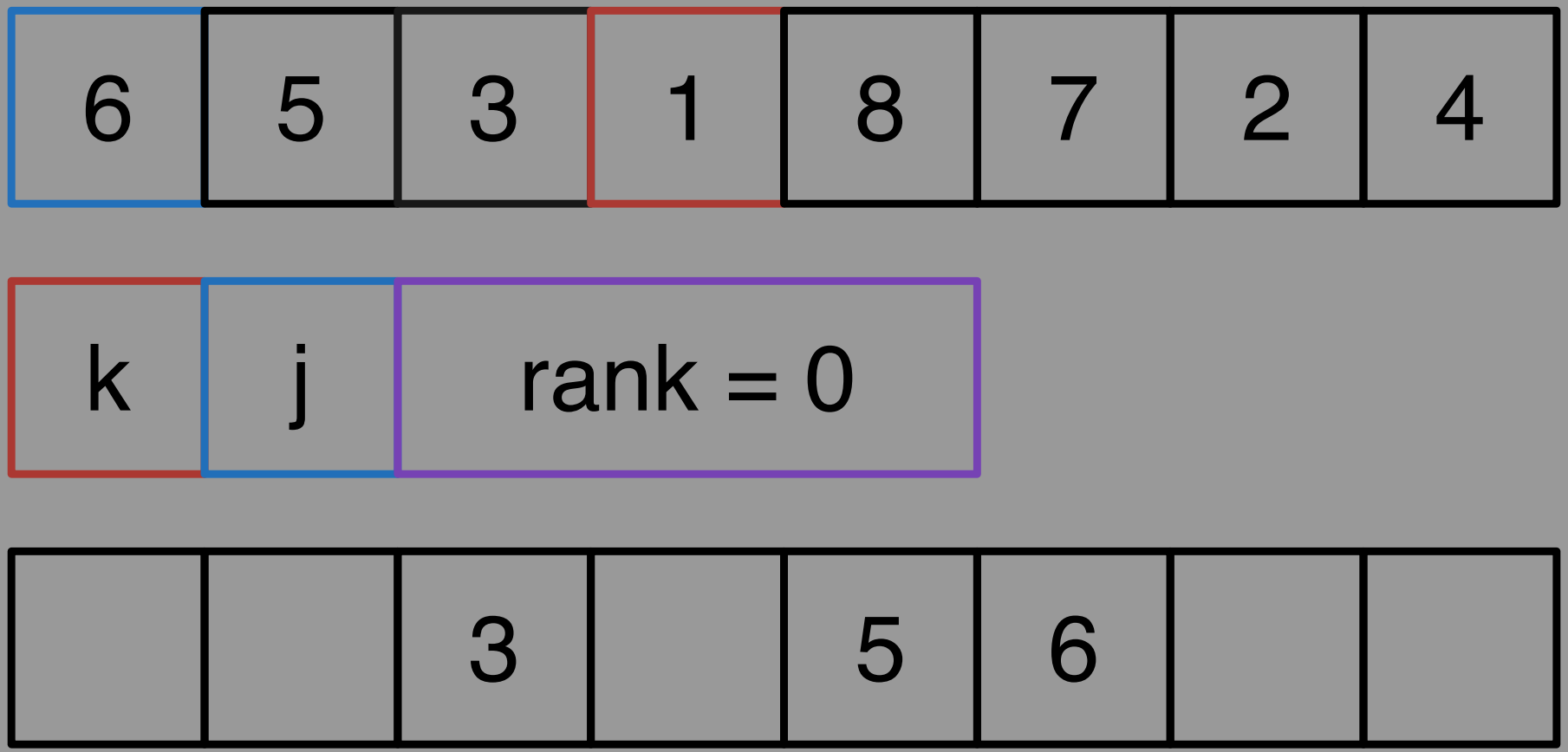
Rank Sort Example

Rank Sort Example

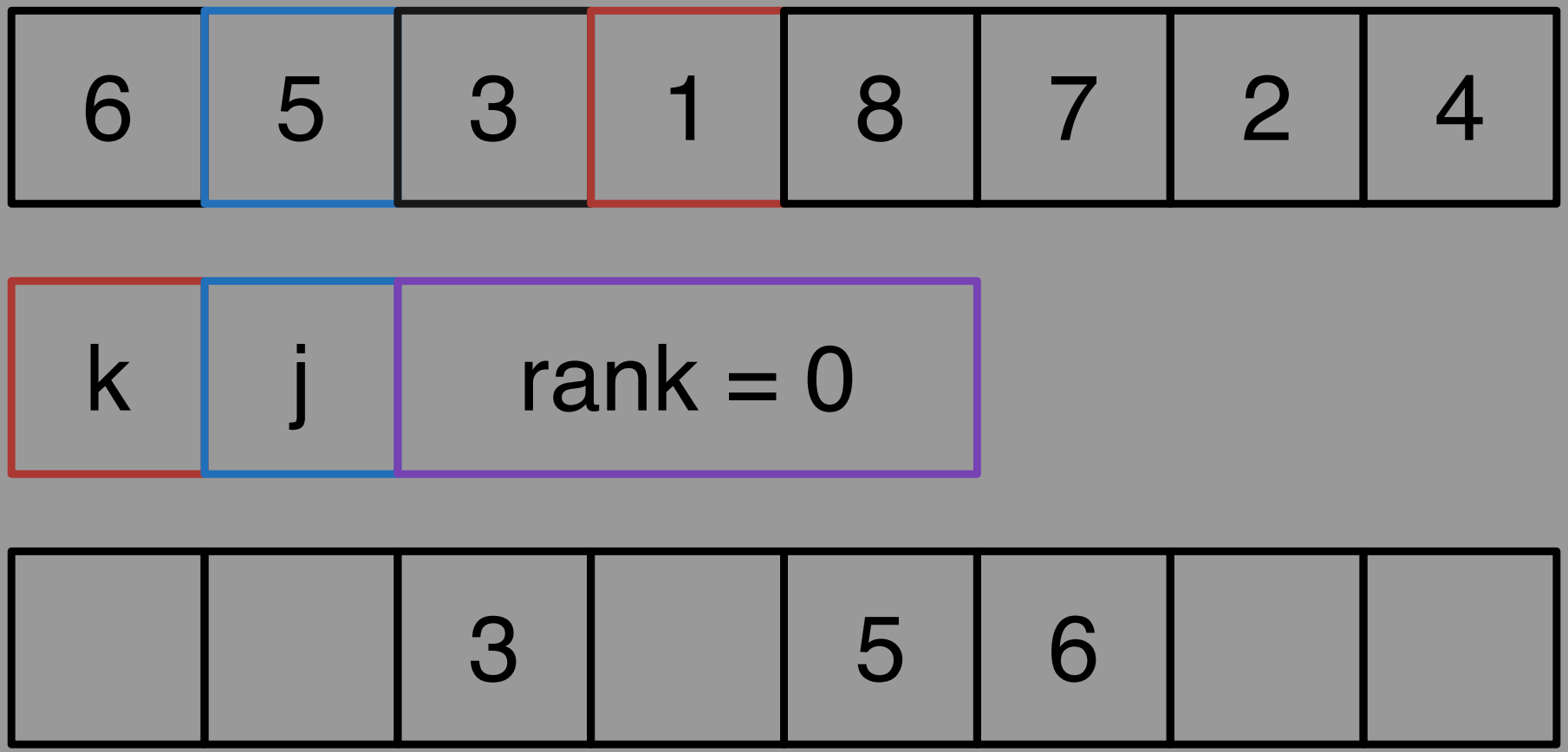
Rank Sort Example



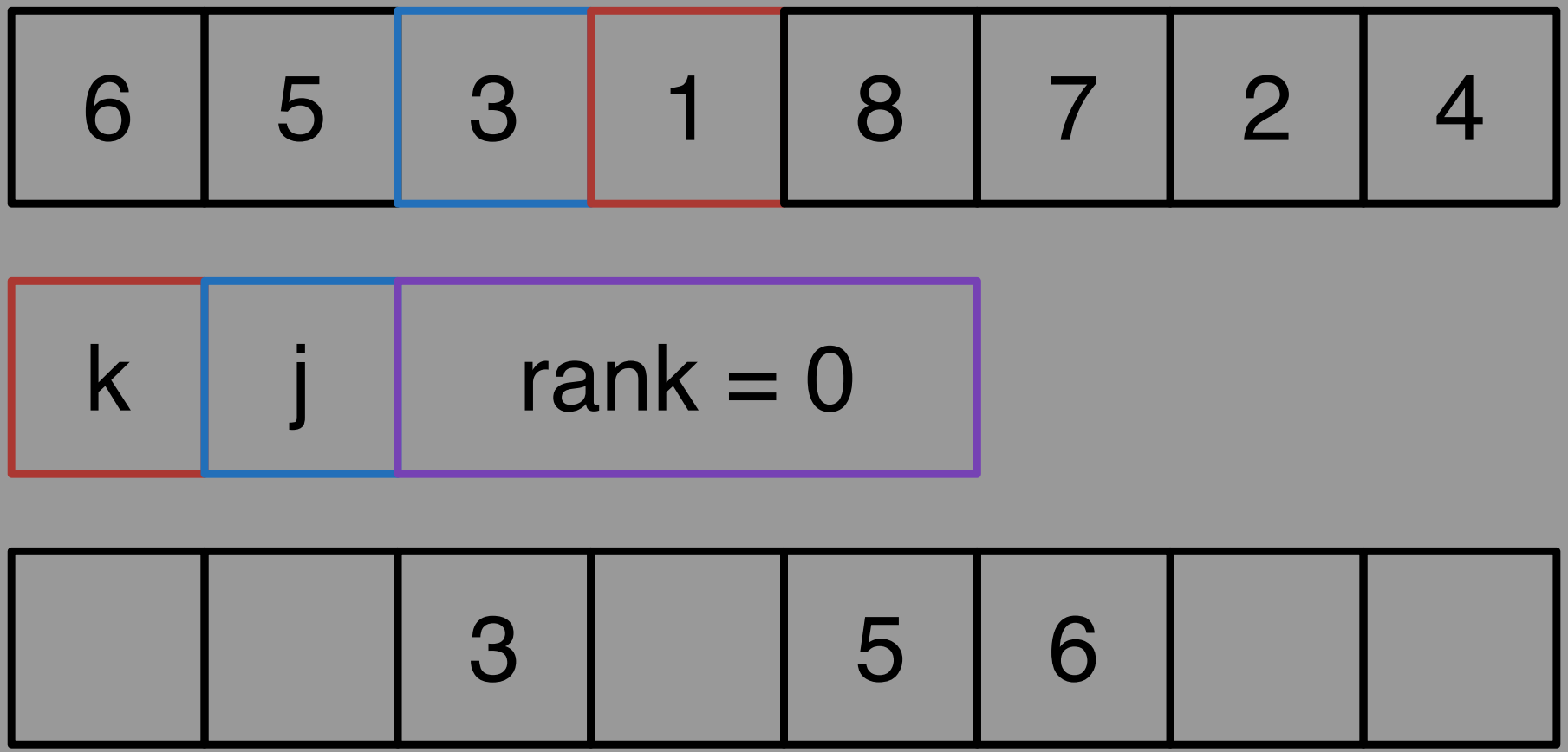
Rank Sort Example



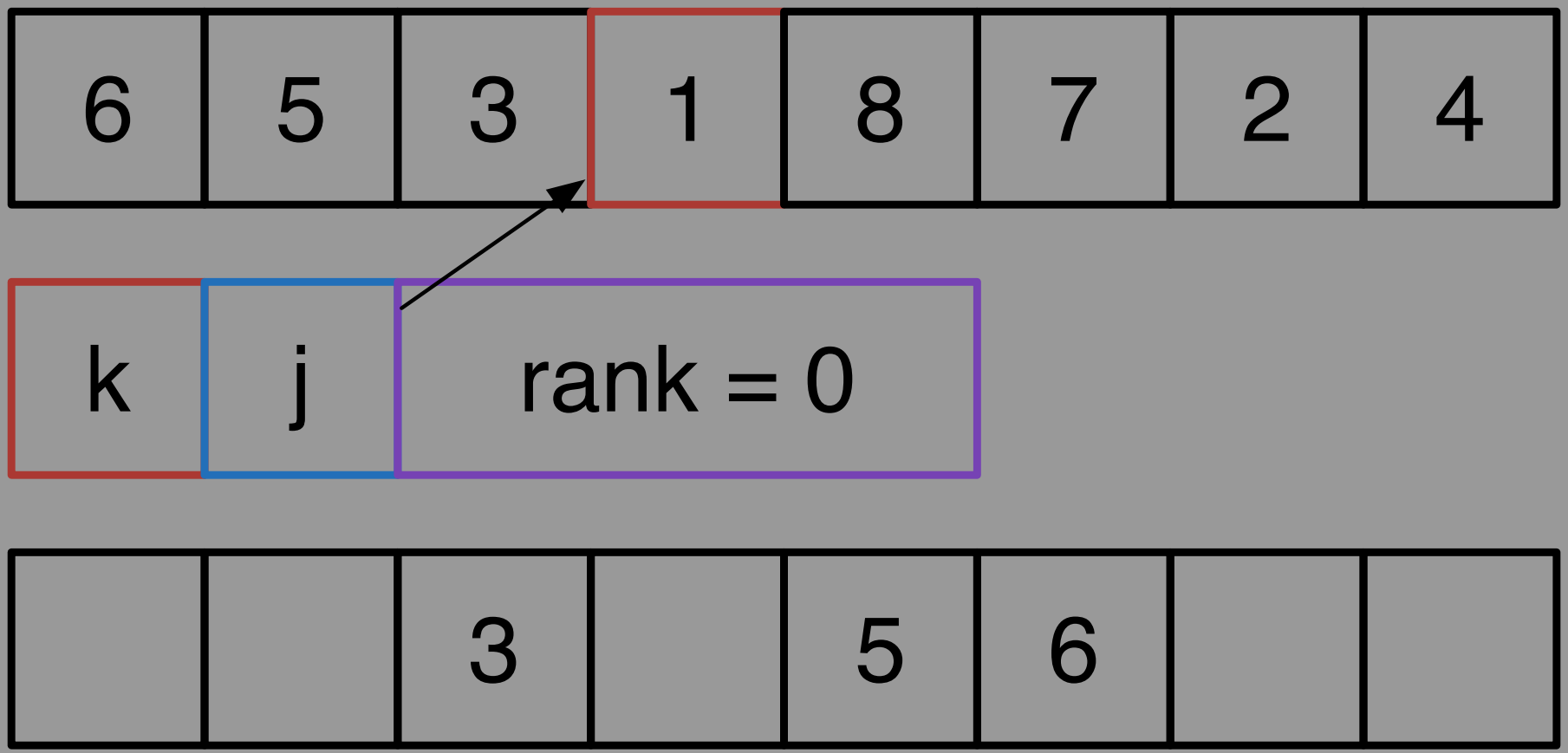
Rank Sort Example



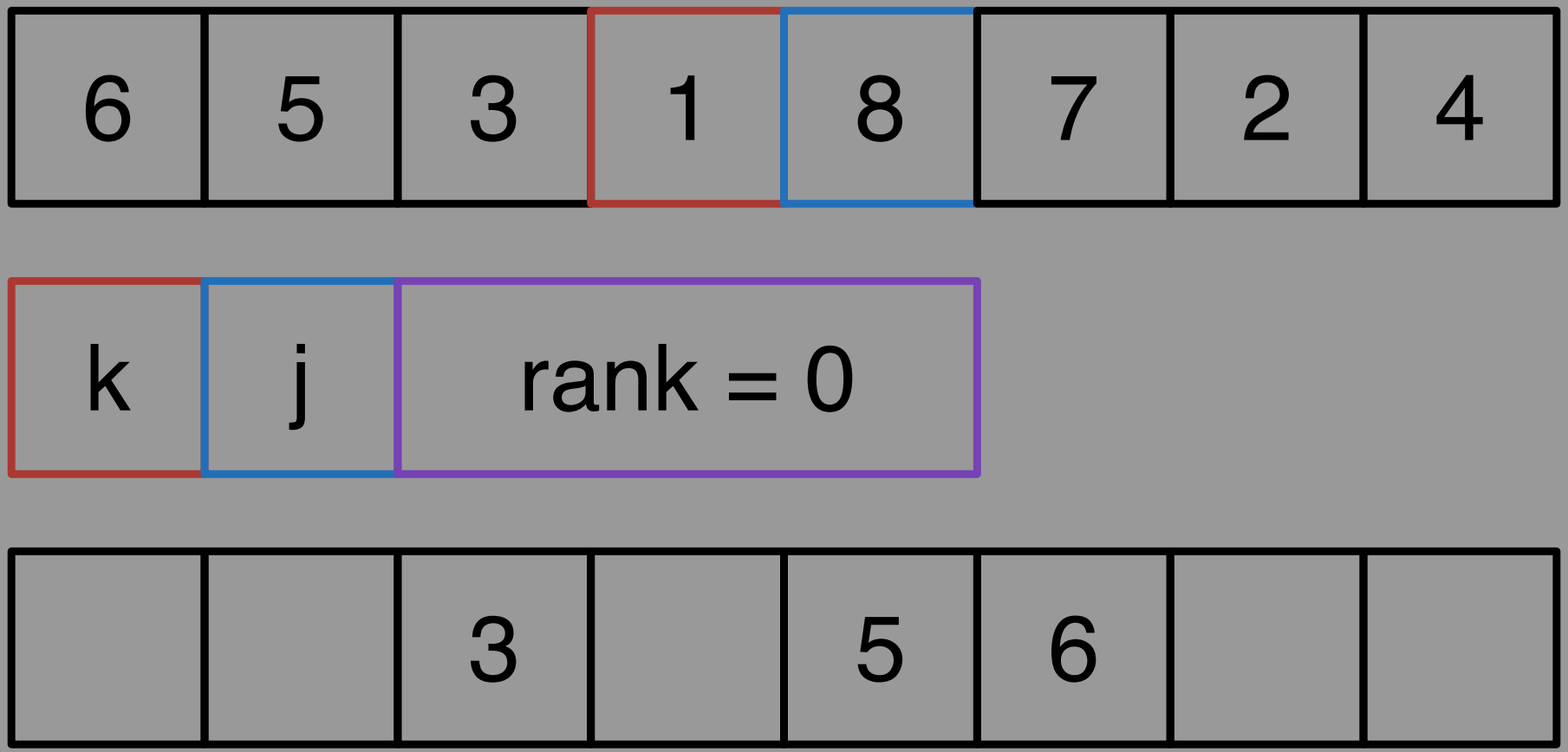
Rank Sort Example



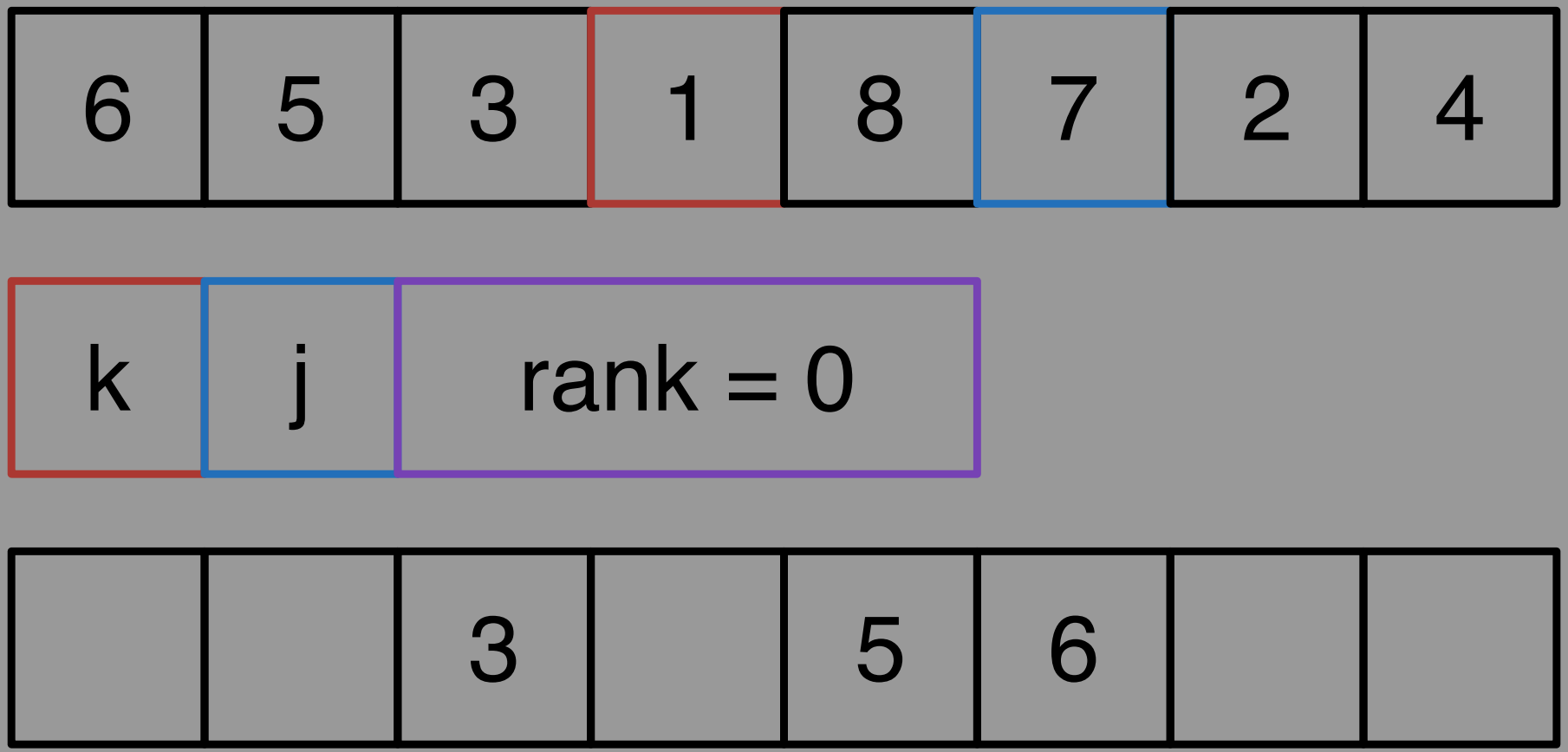
Rank Sort Example



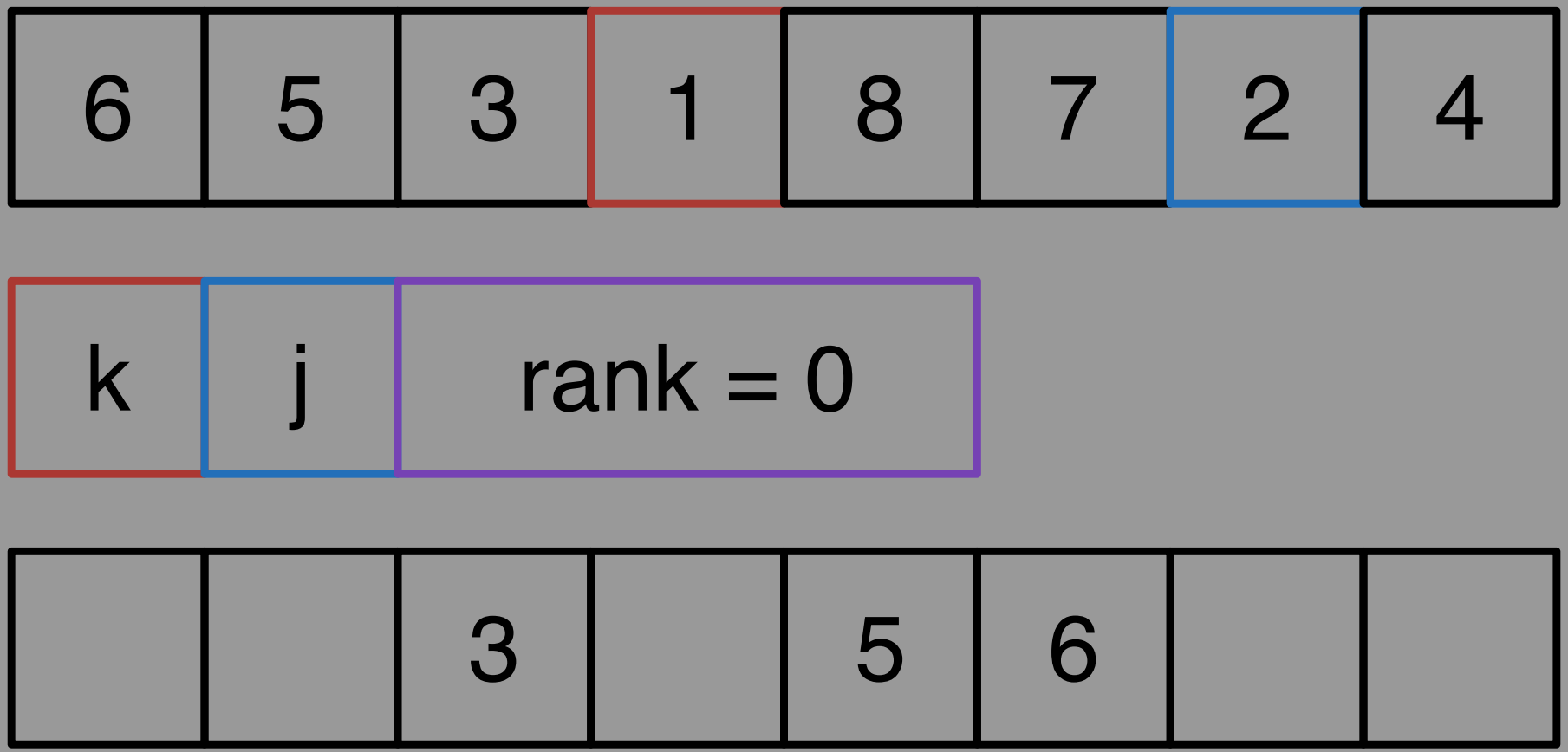
Rank Sort Example



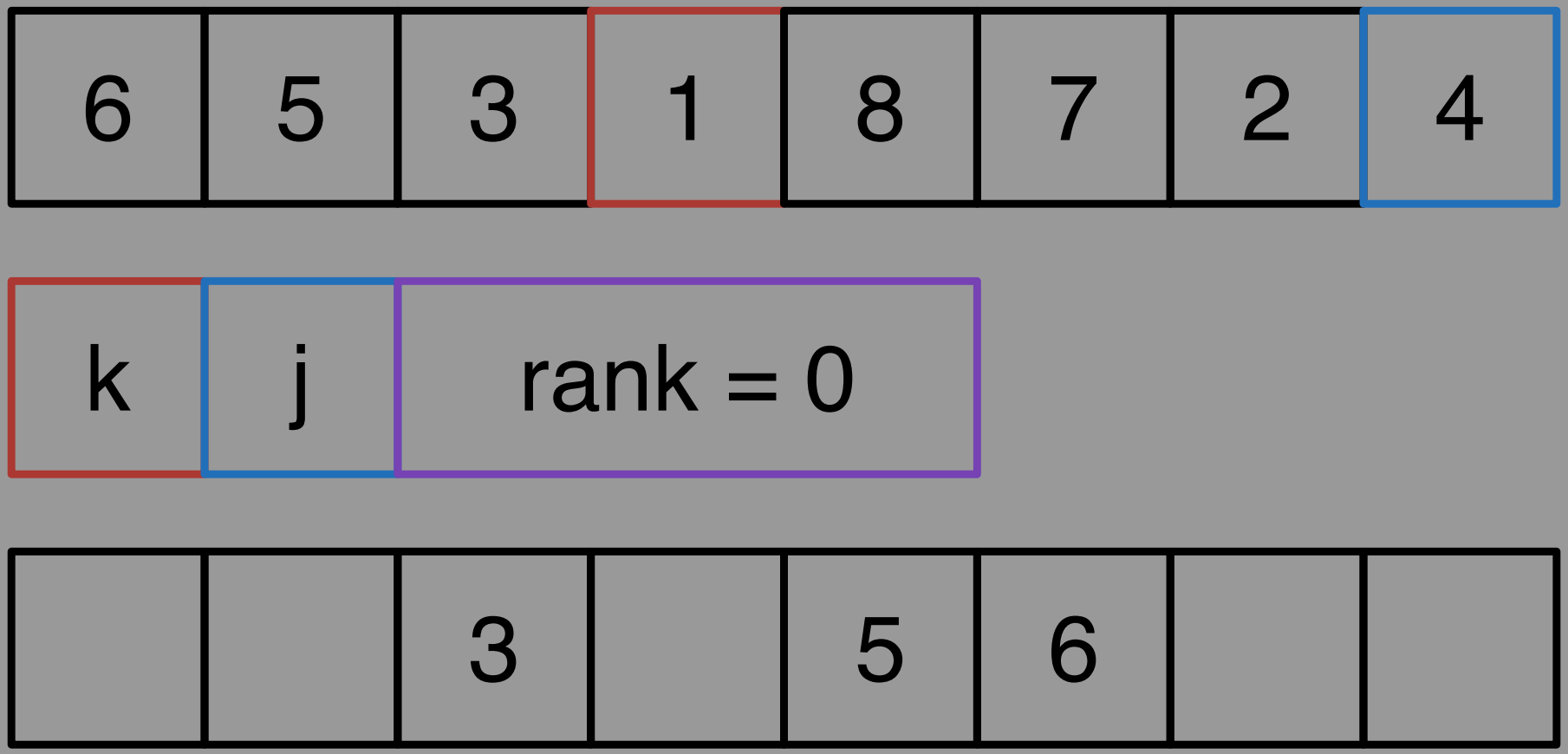
Rank Sort Example



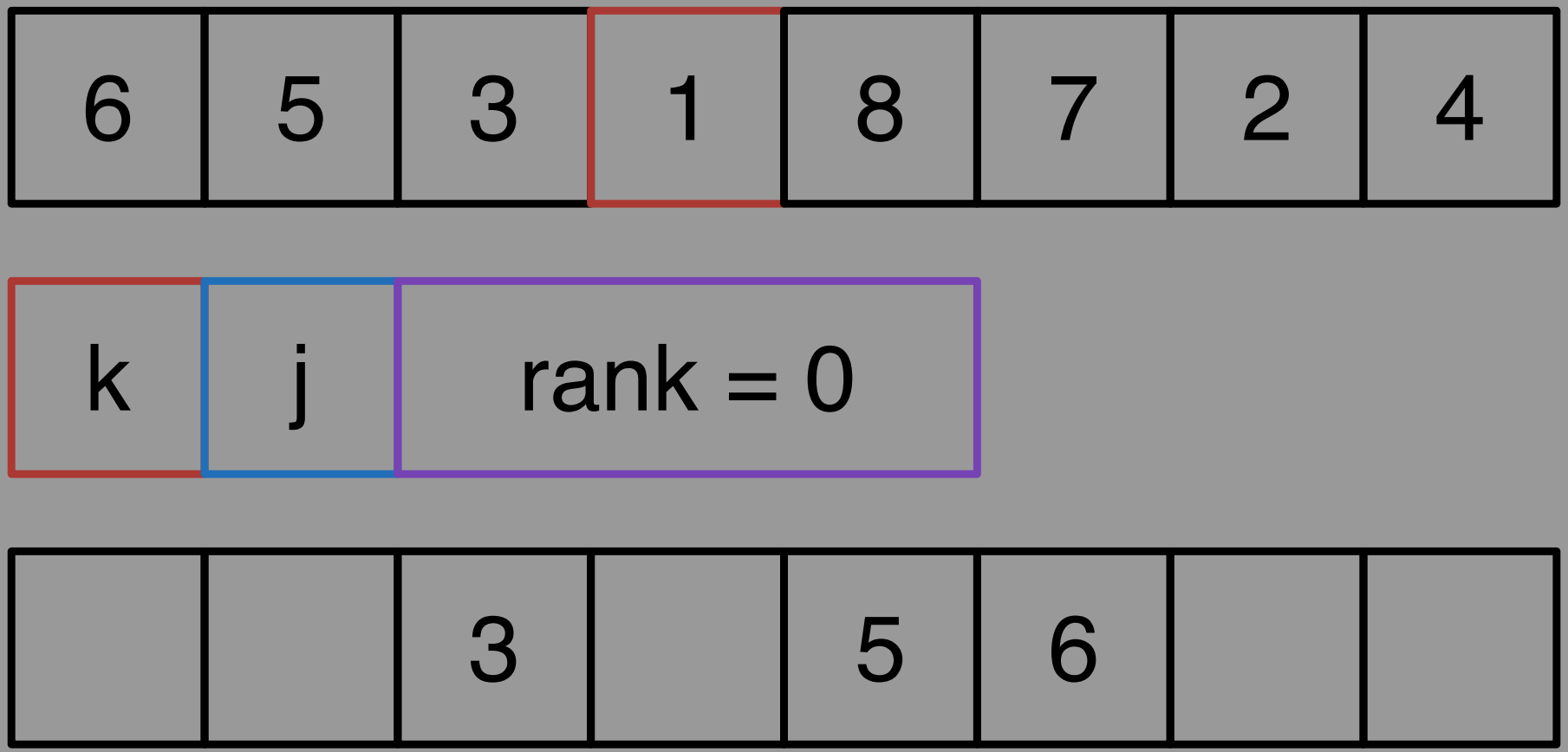
Rank Sort Example



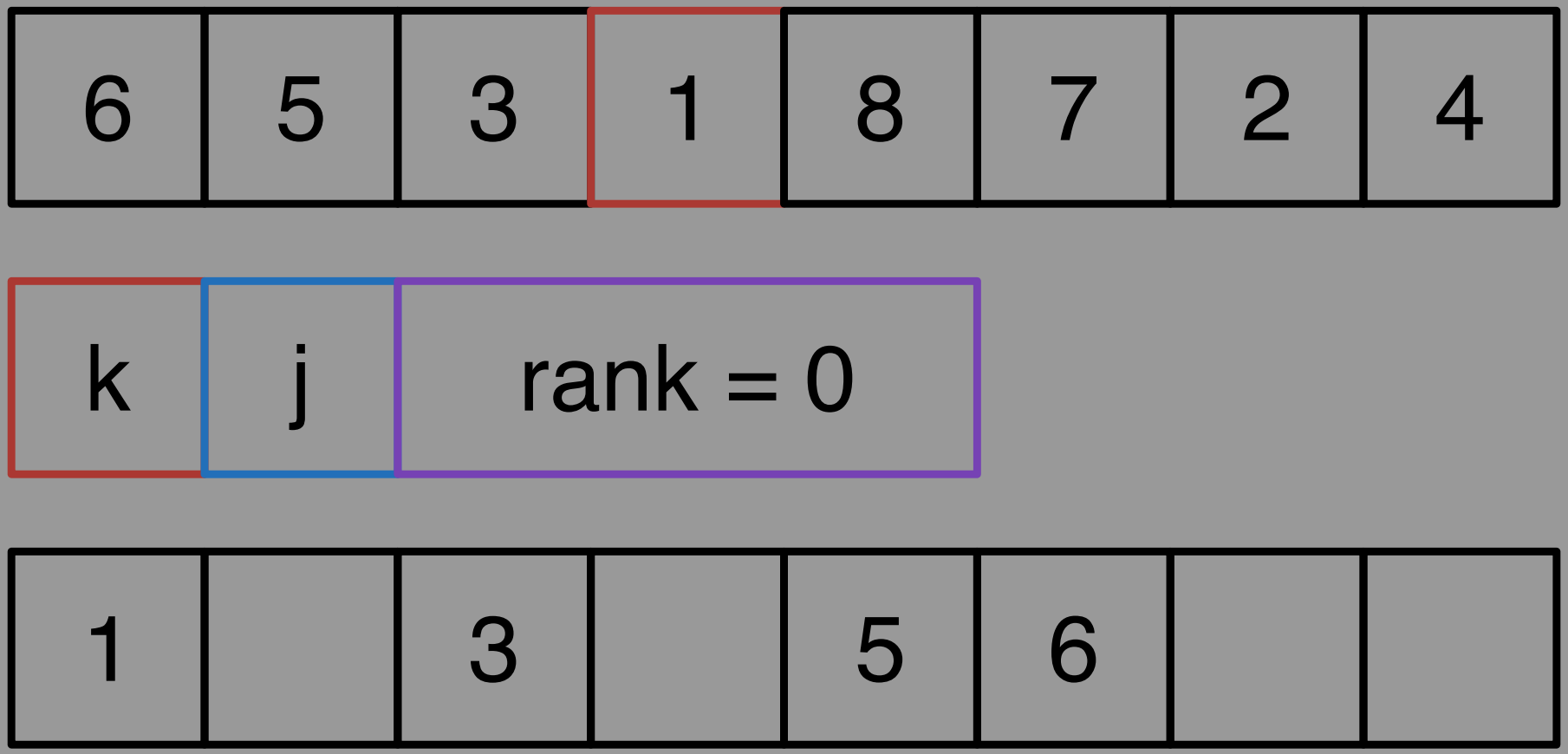
Rank Sort Example



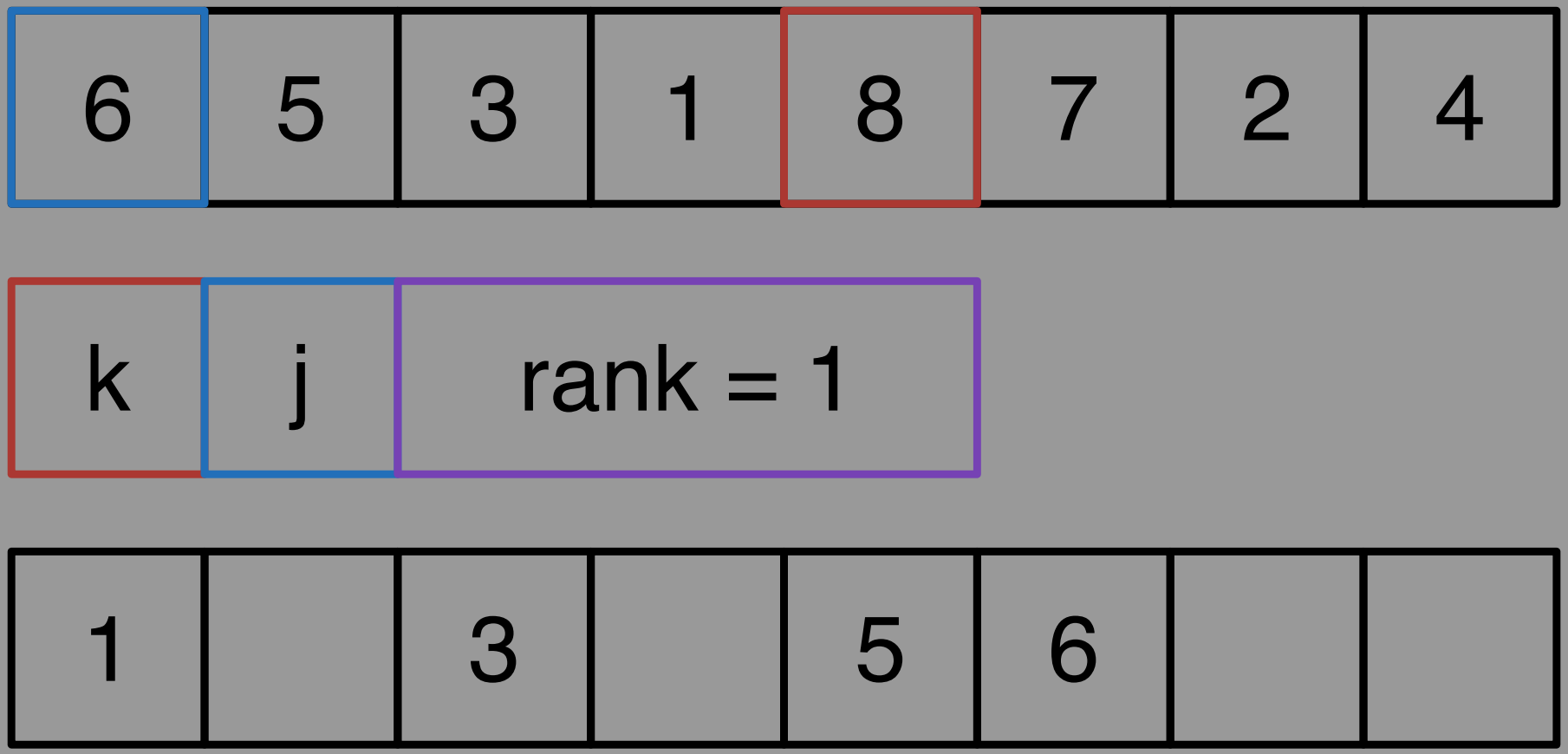
Rank Sort Example



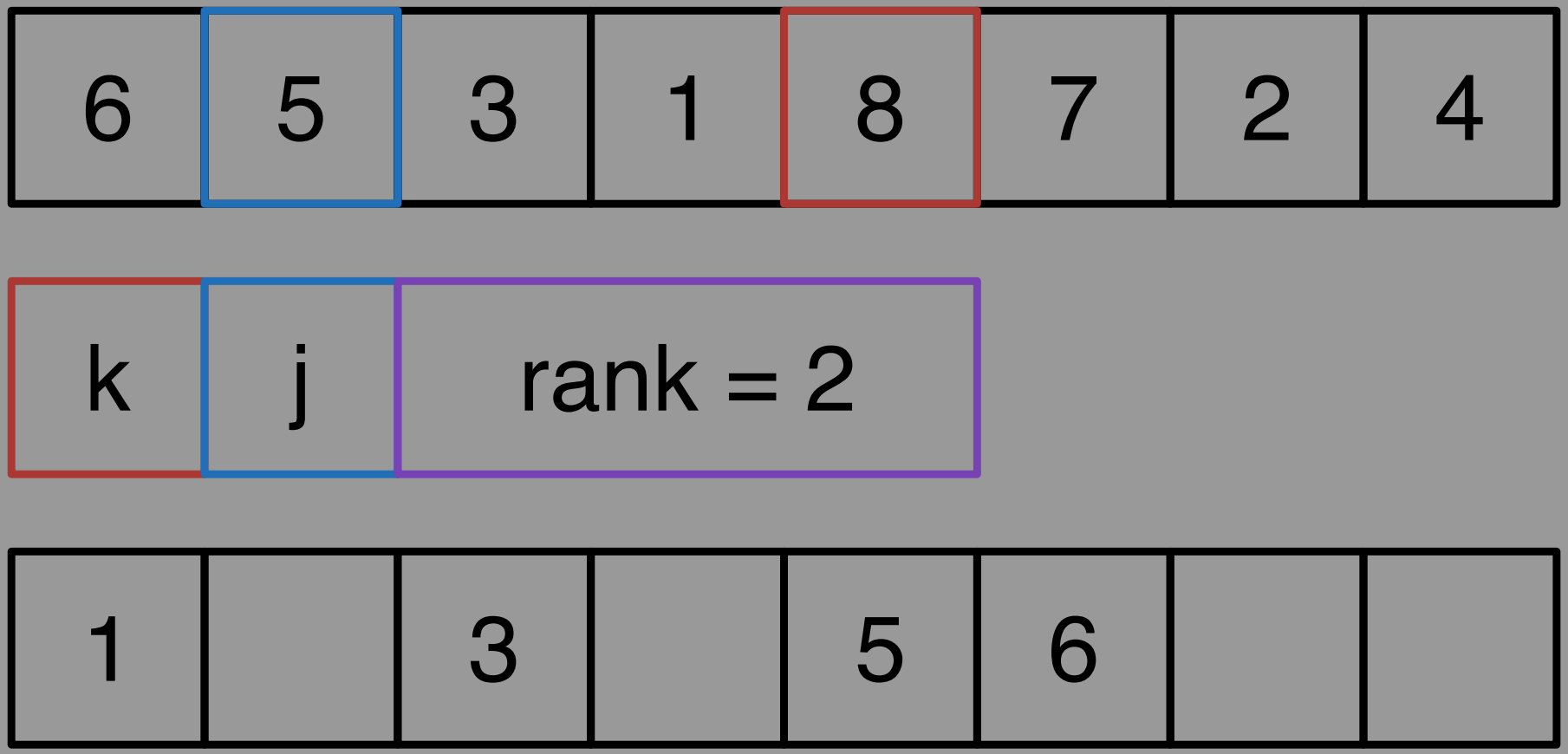
Rank Sort Example



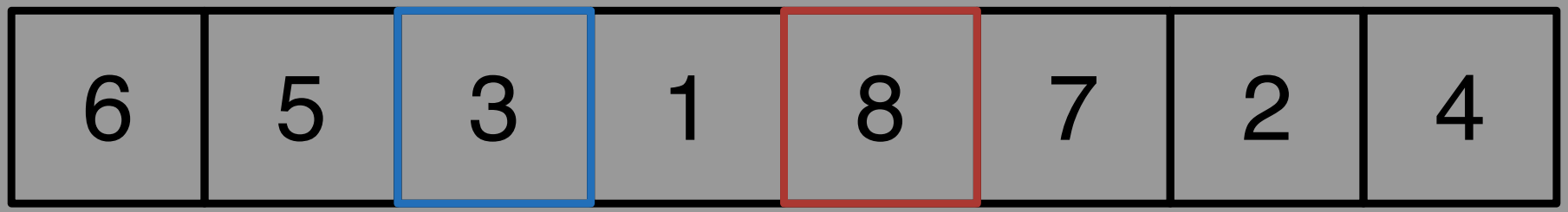
Rank Sort Example

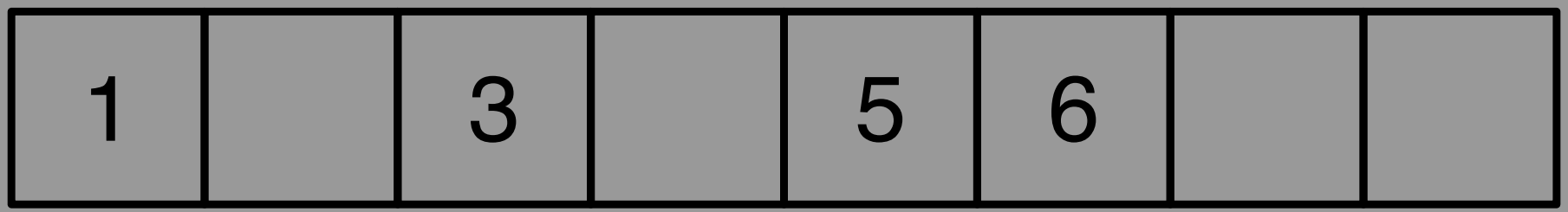


Rank Sort Example

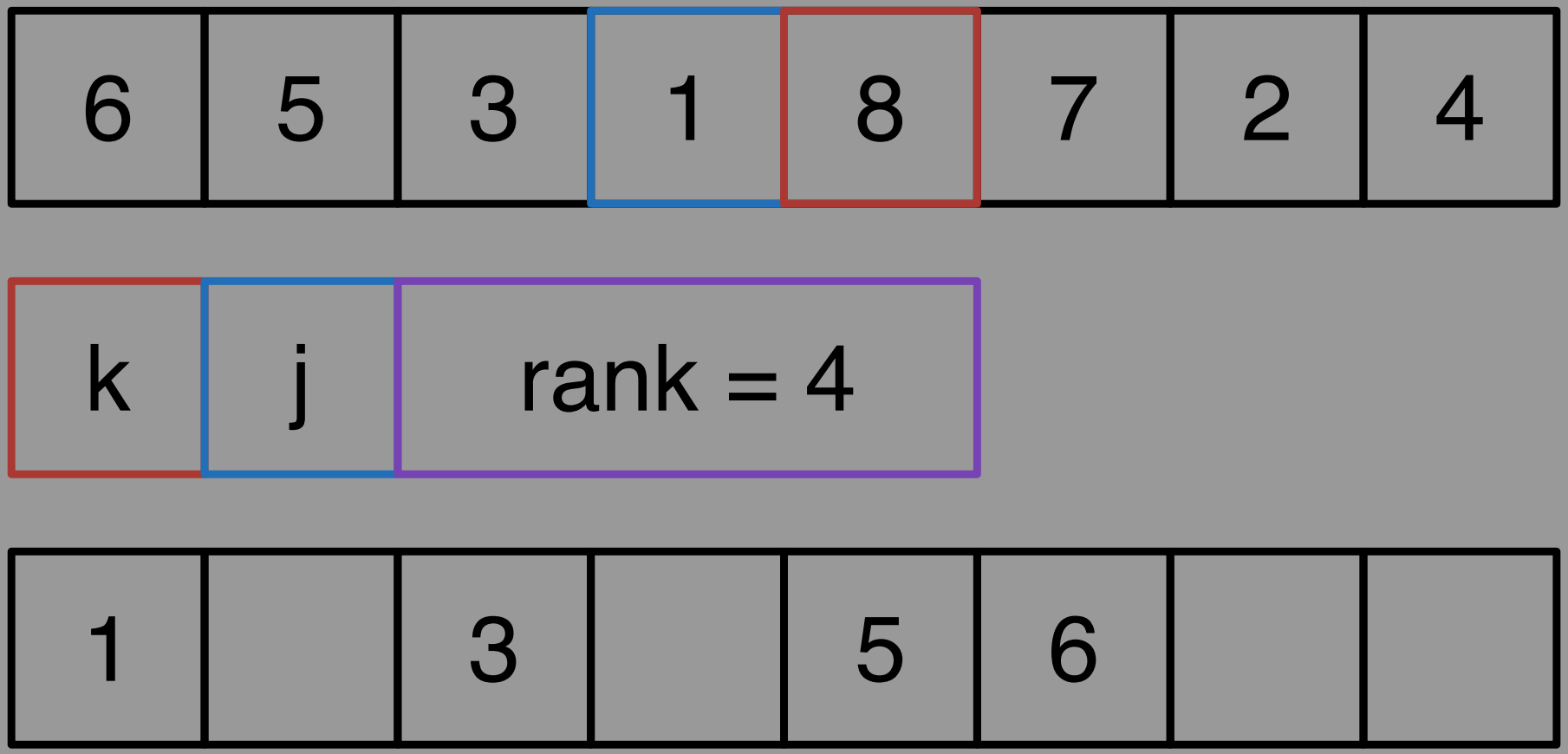


Rank Sort Example

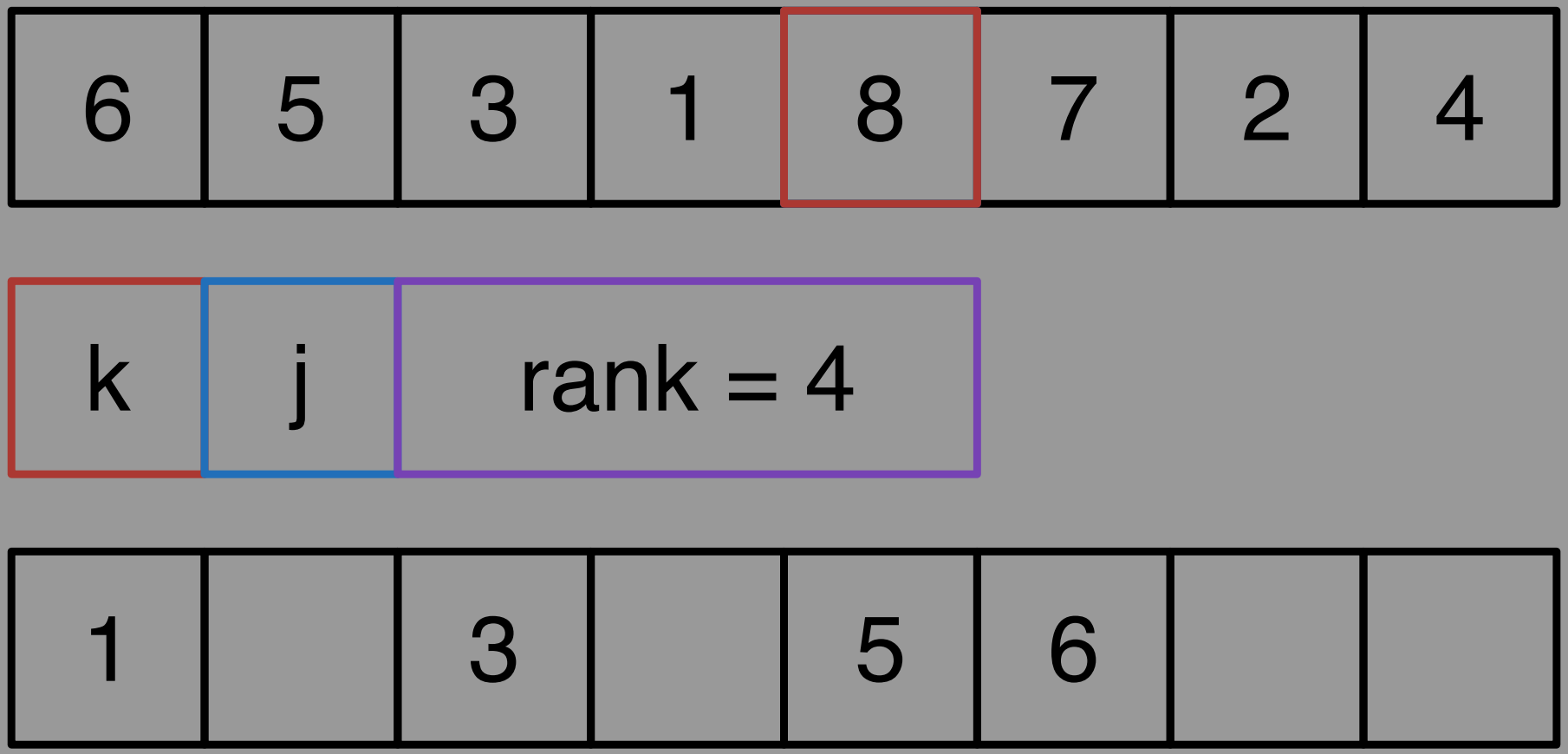




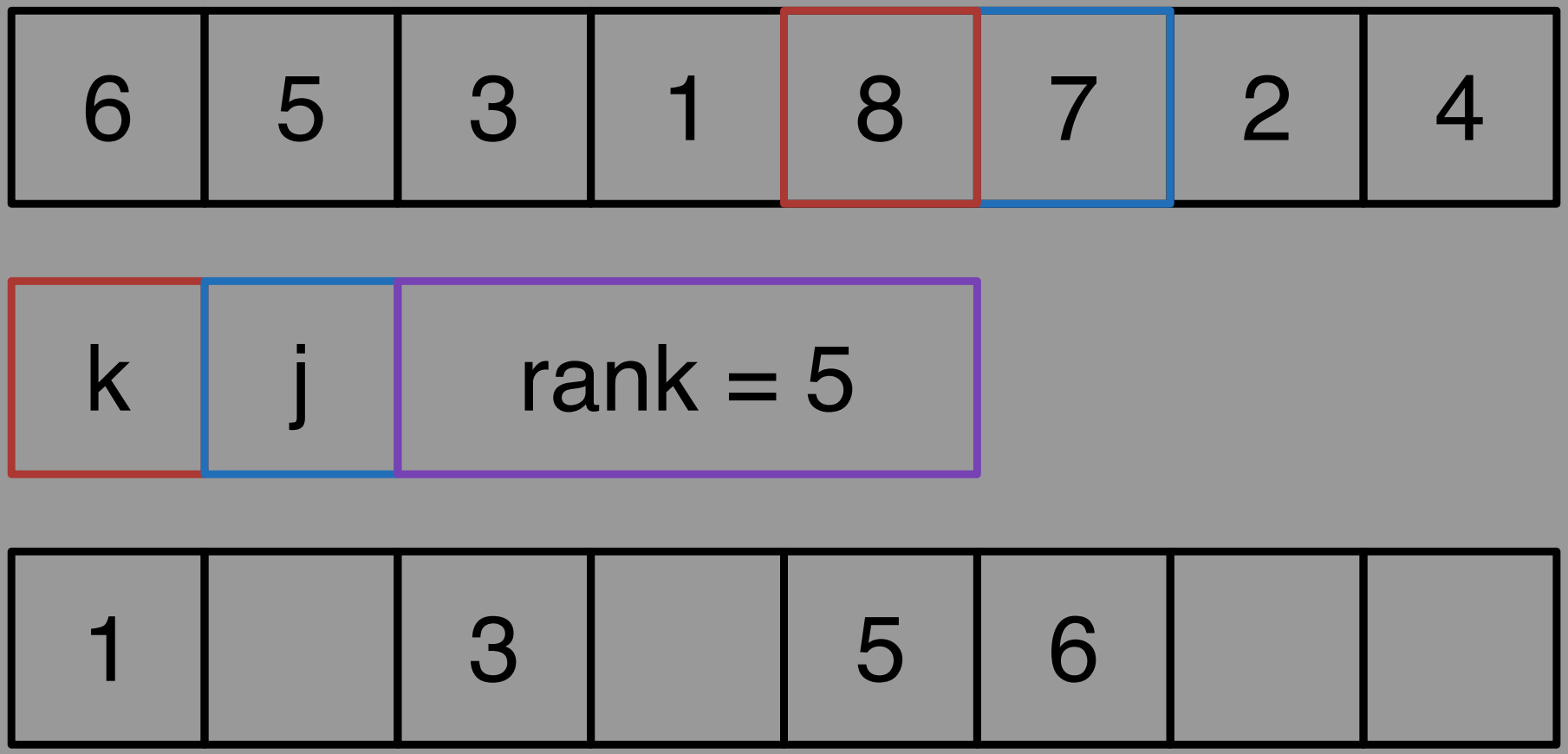
Rank Sort Example



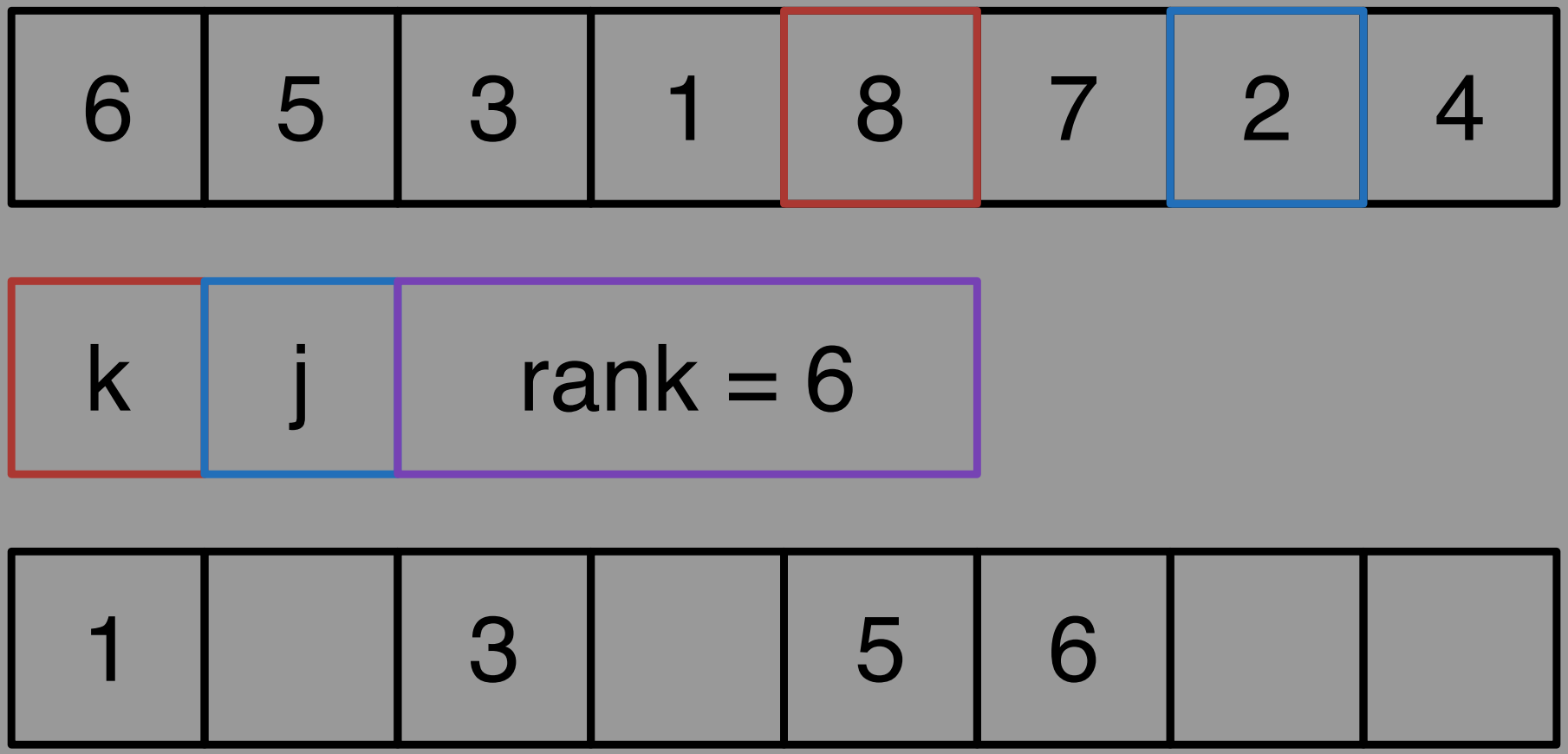
Rank Sort Example



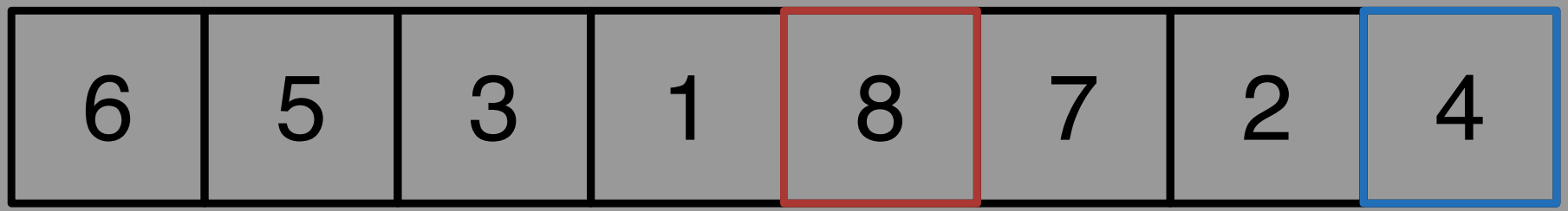
Rank Sort Example

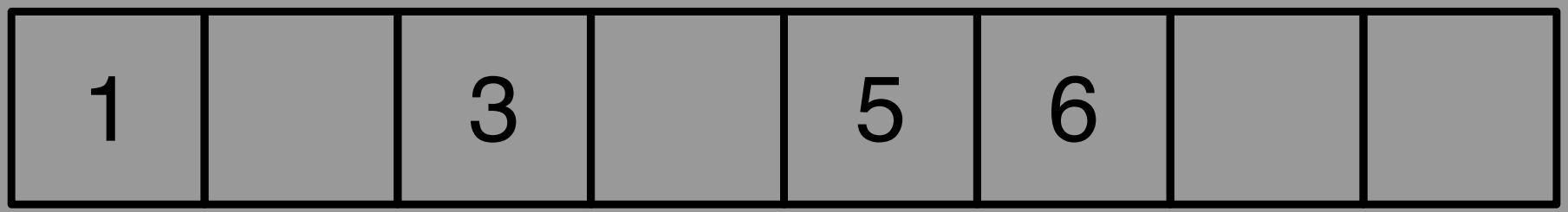


Rank Sort Example

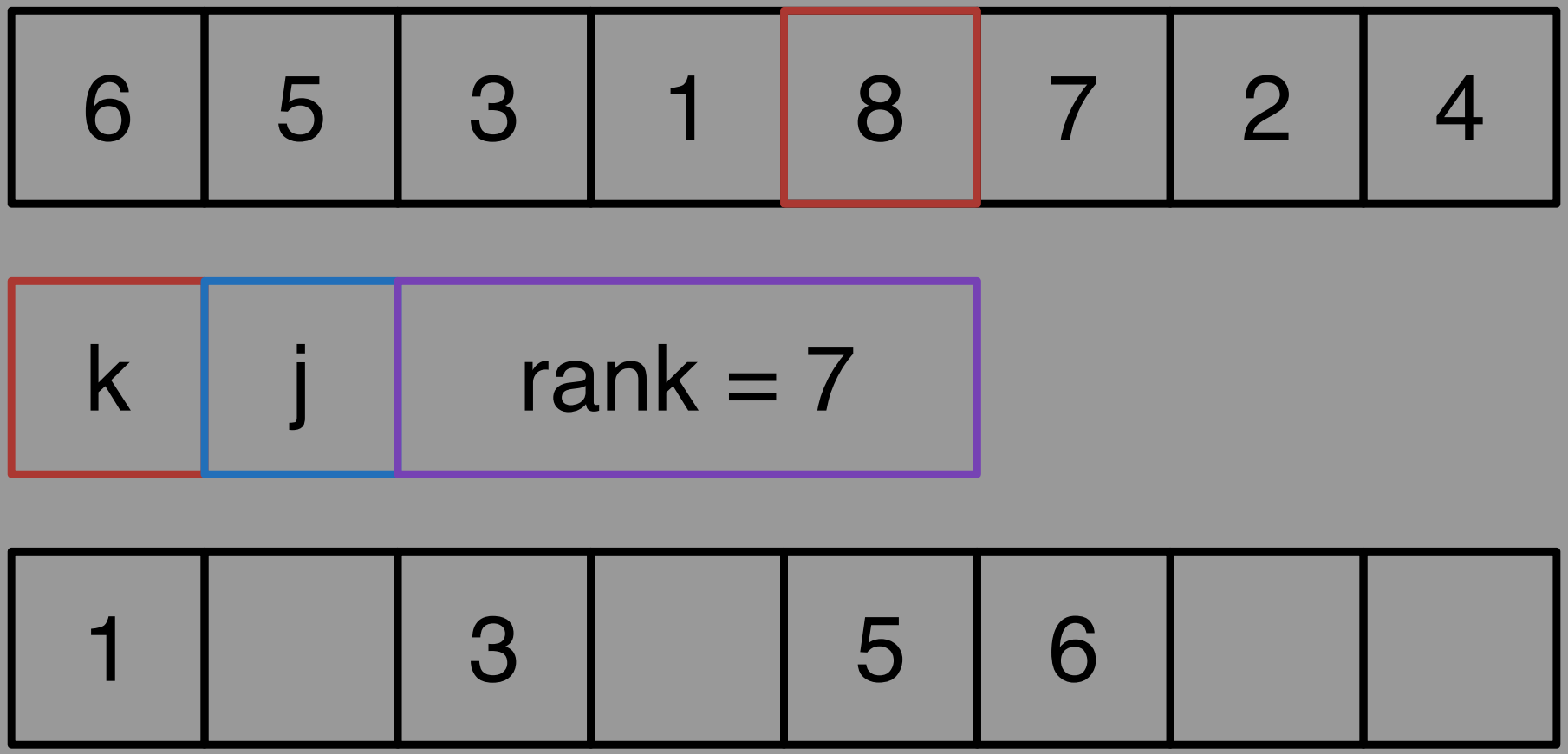


Rank Sort Example

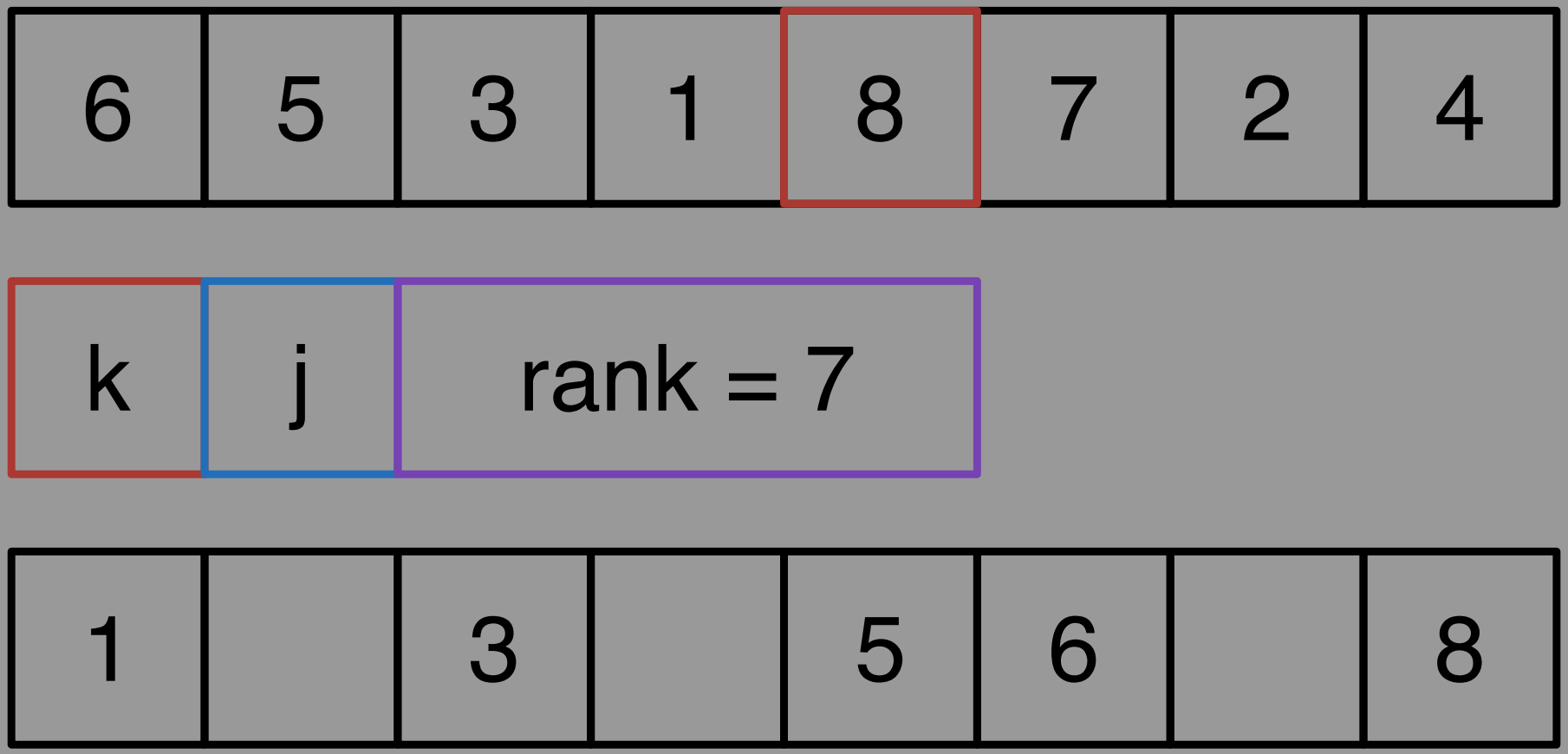




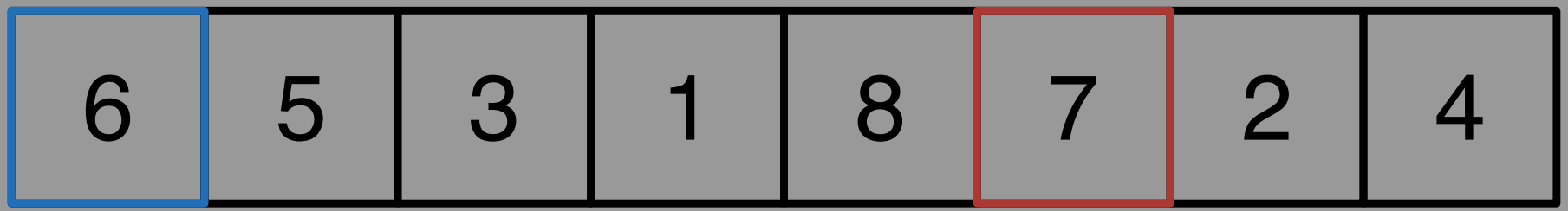
Rank Sort Example

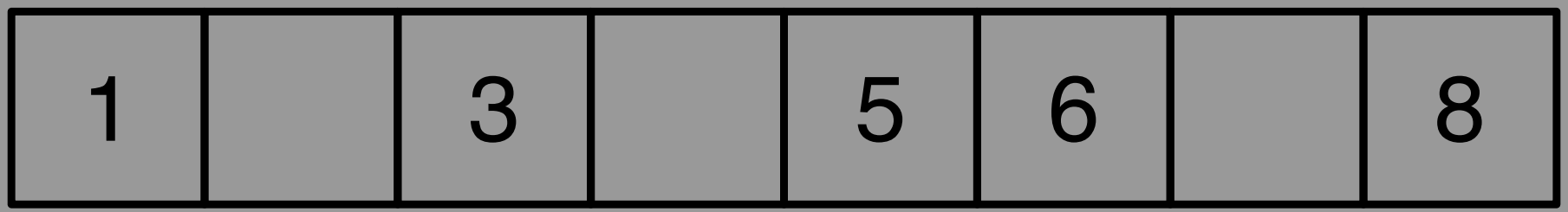


Rank Sort Example

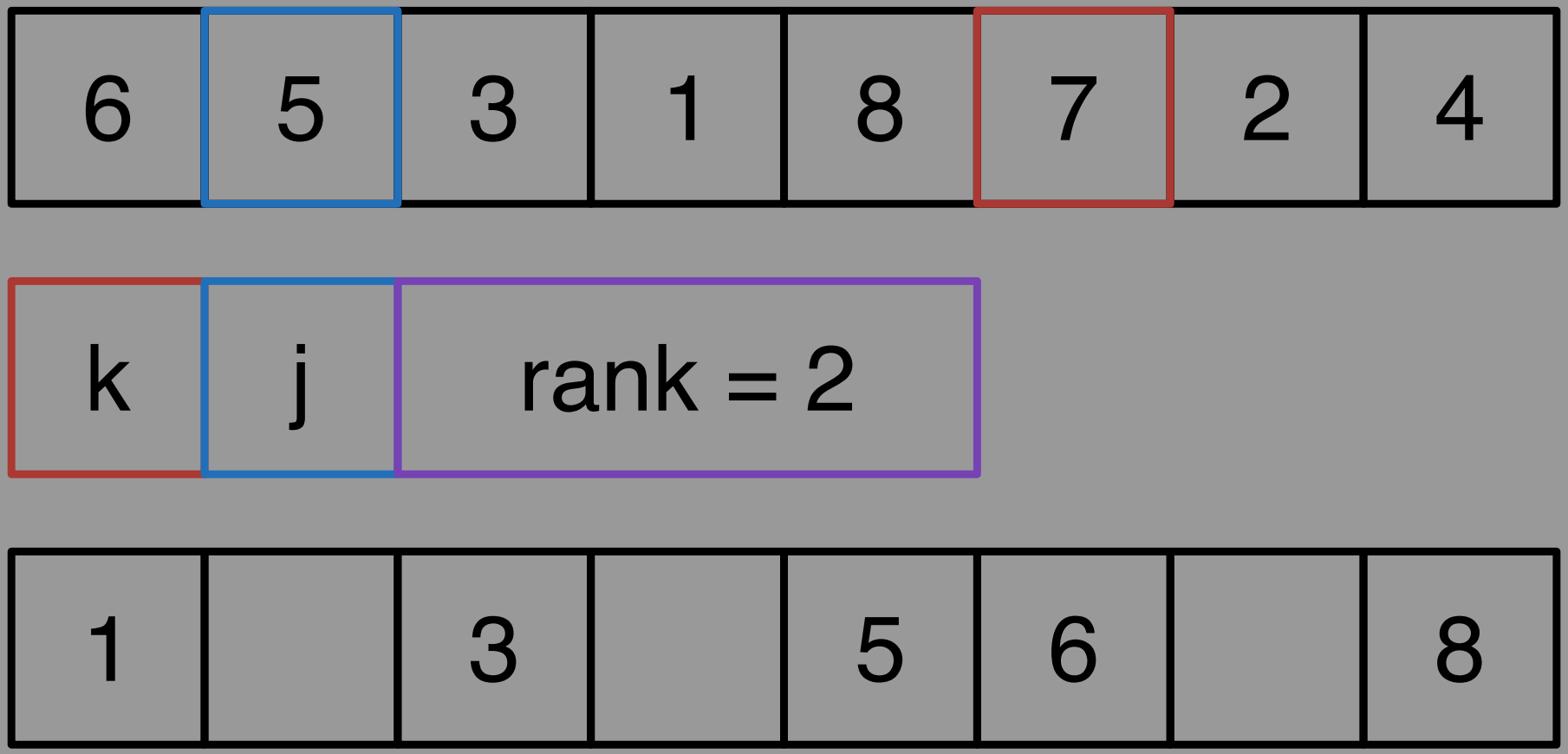


Rank Sort Example

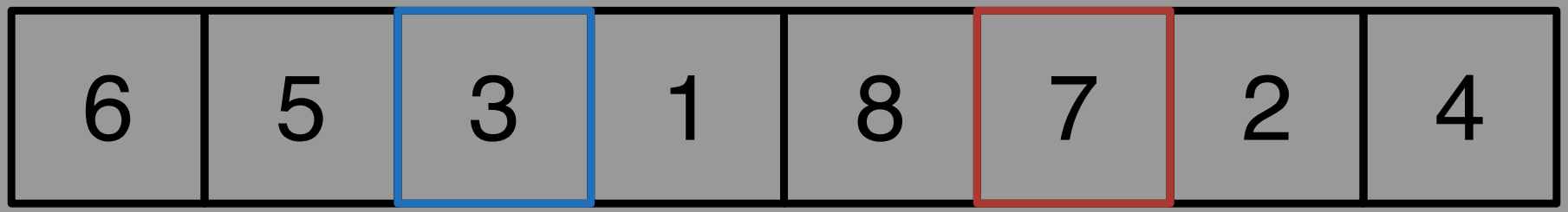


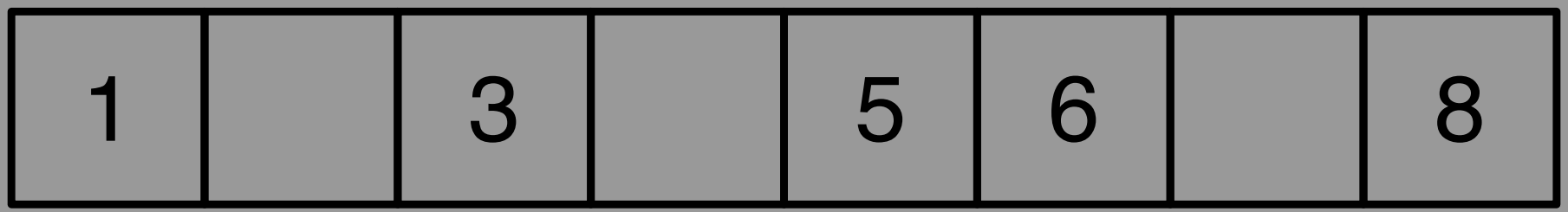


Rank Sort Example

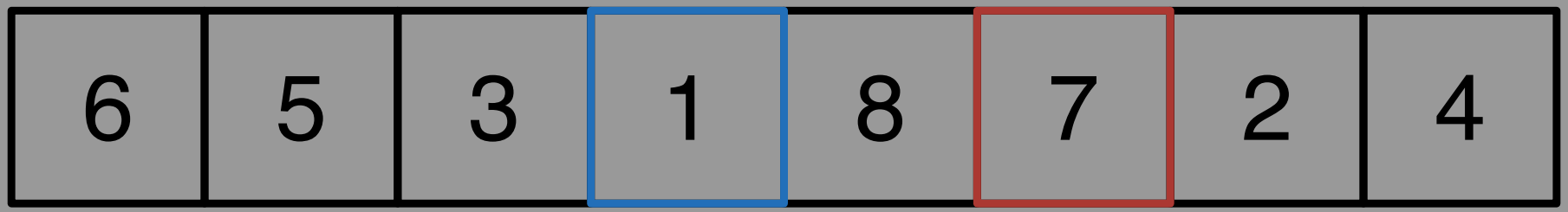


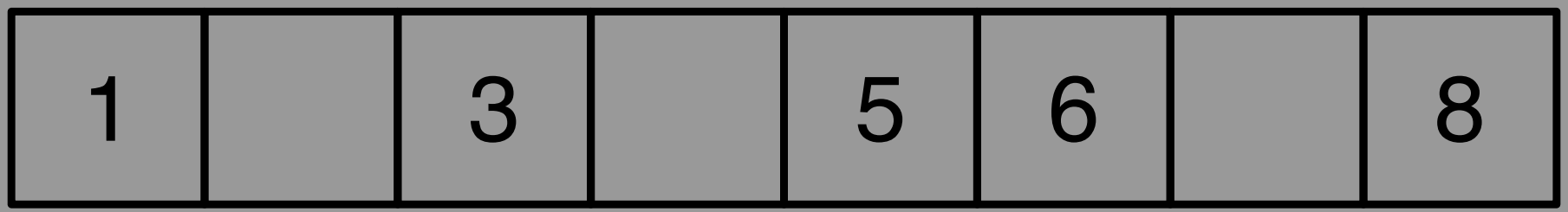
Rank Sort Example



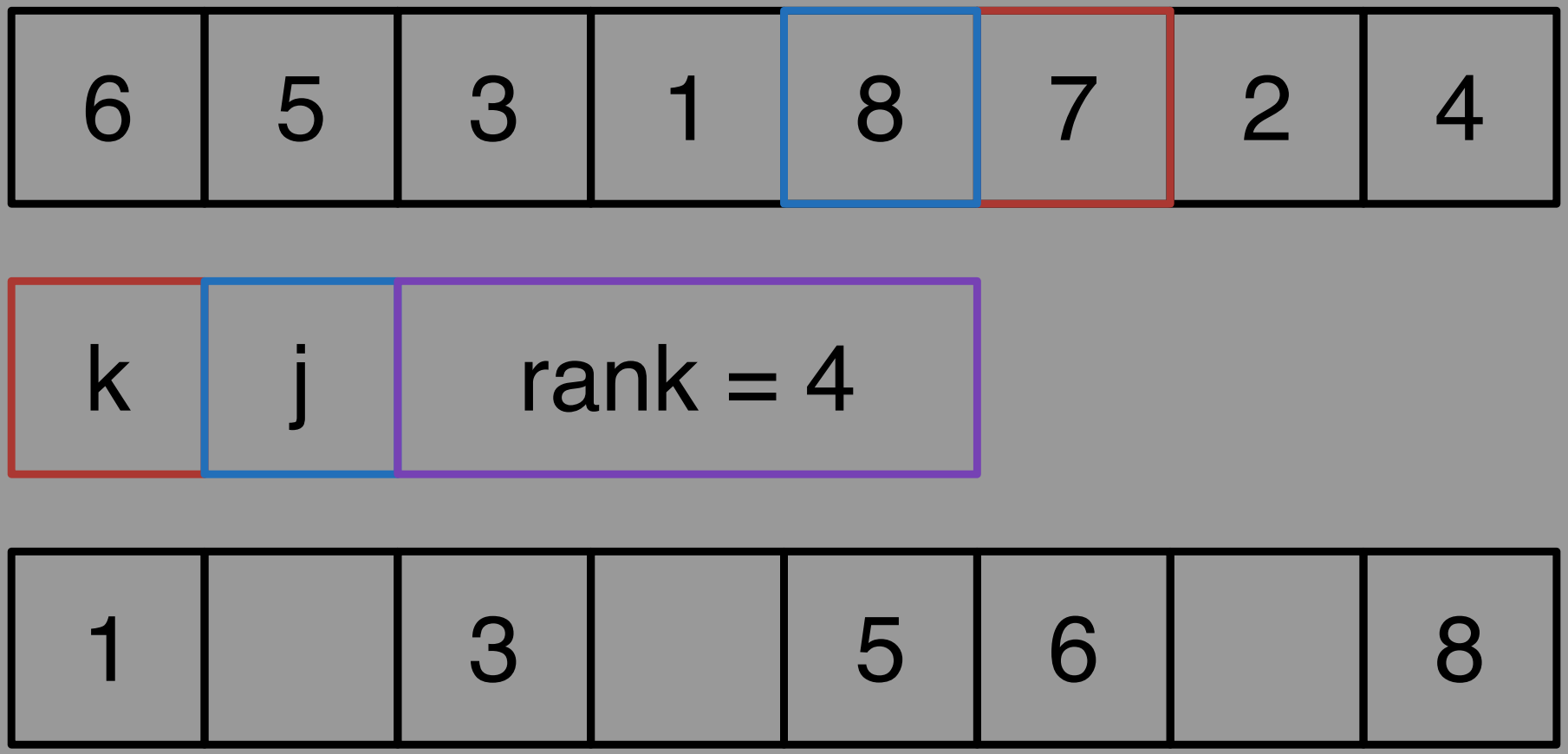


Rank Sort Example

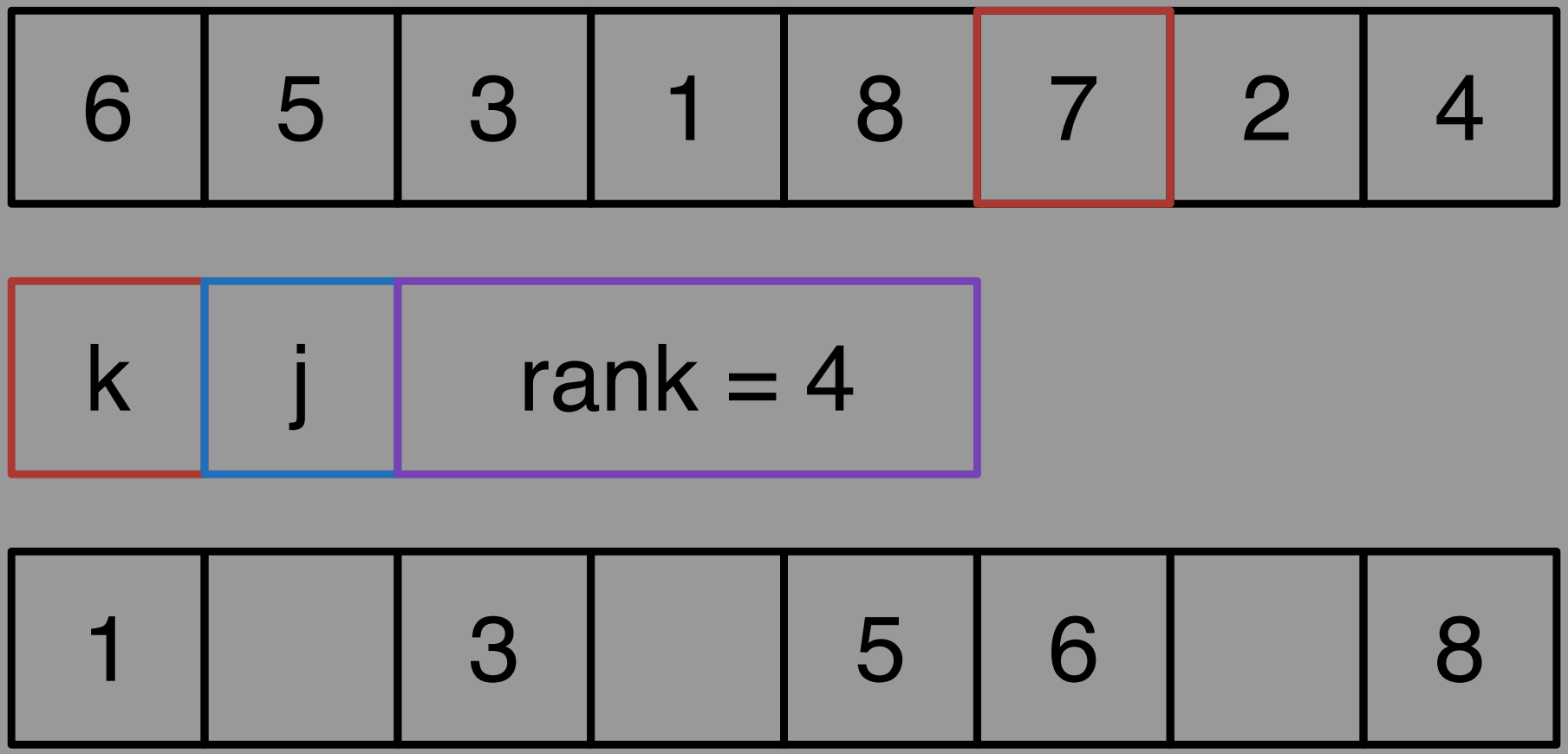




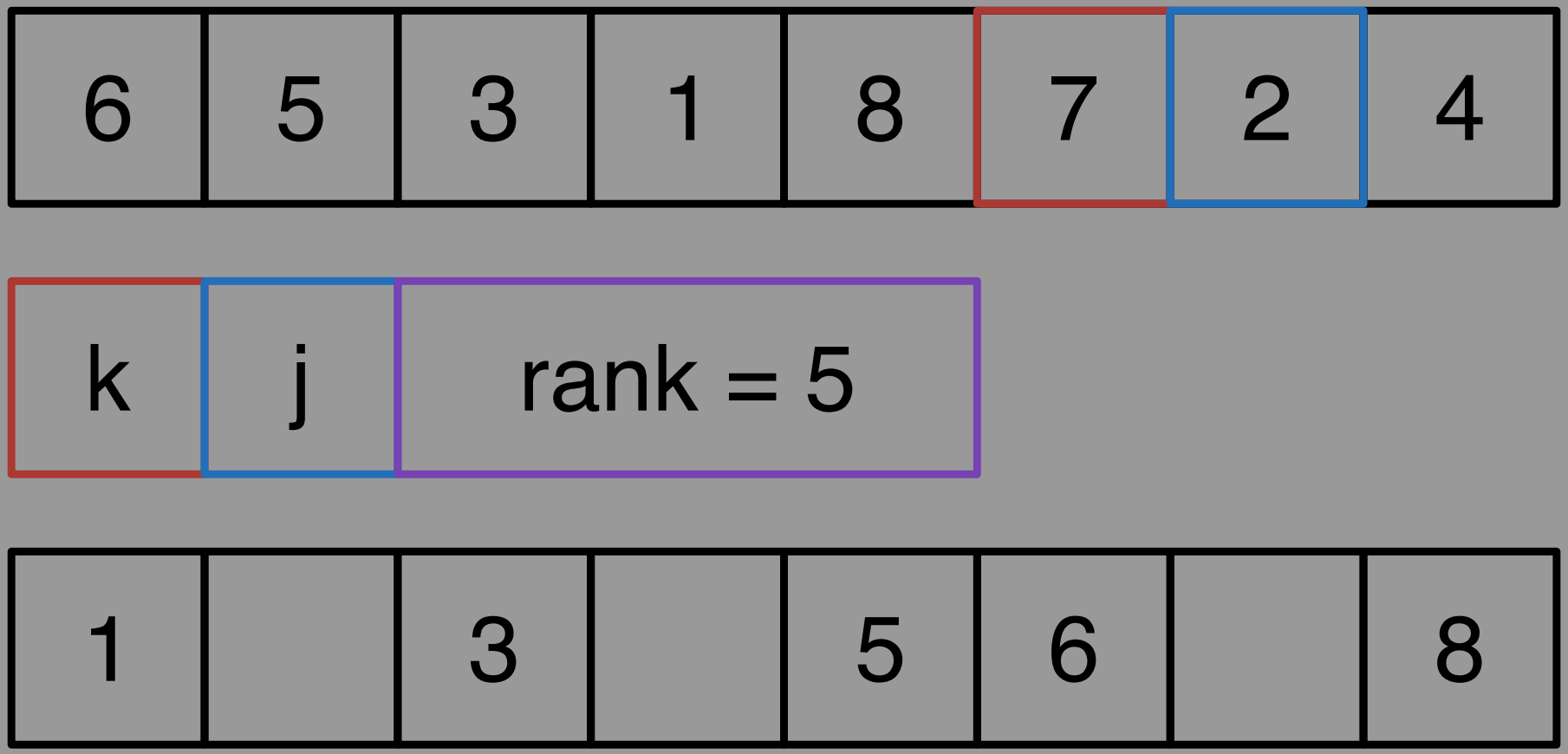
Rank Sort Example



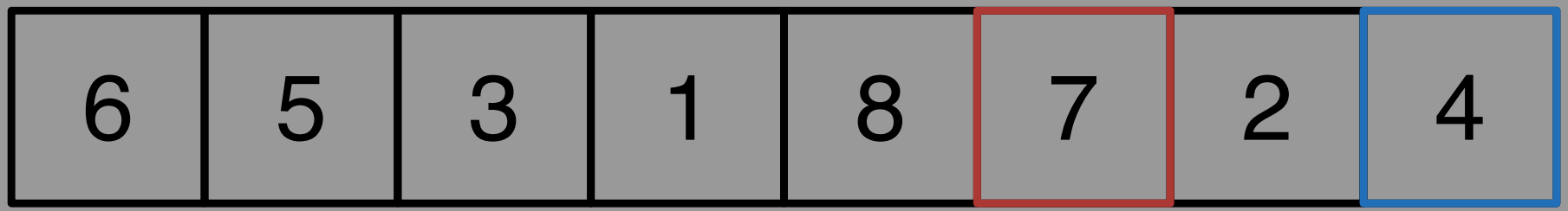
Rank Sort Example

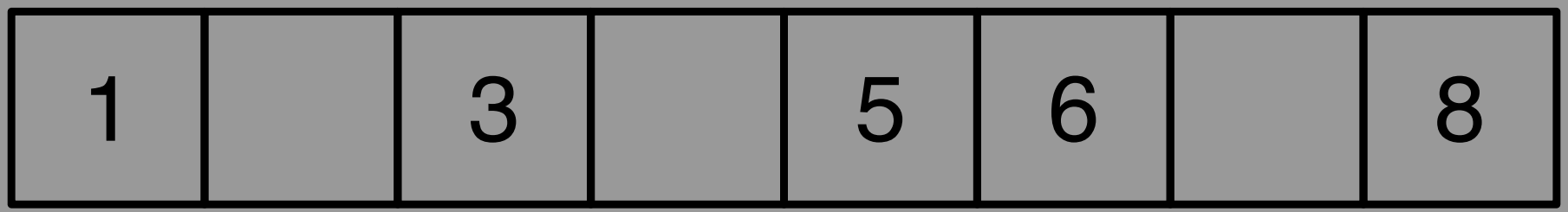


Rank Sort Example

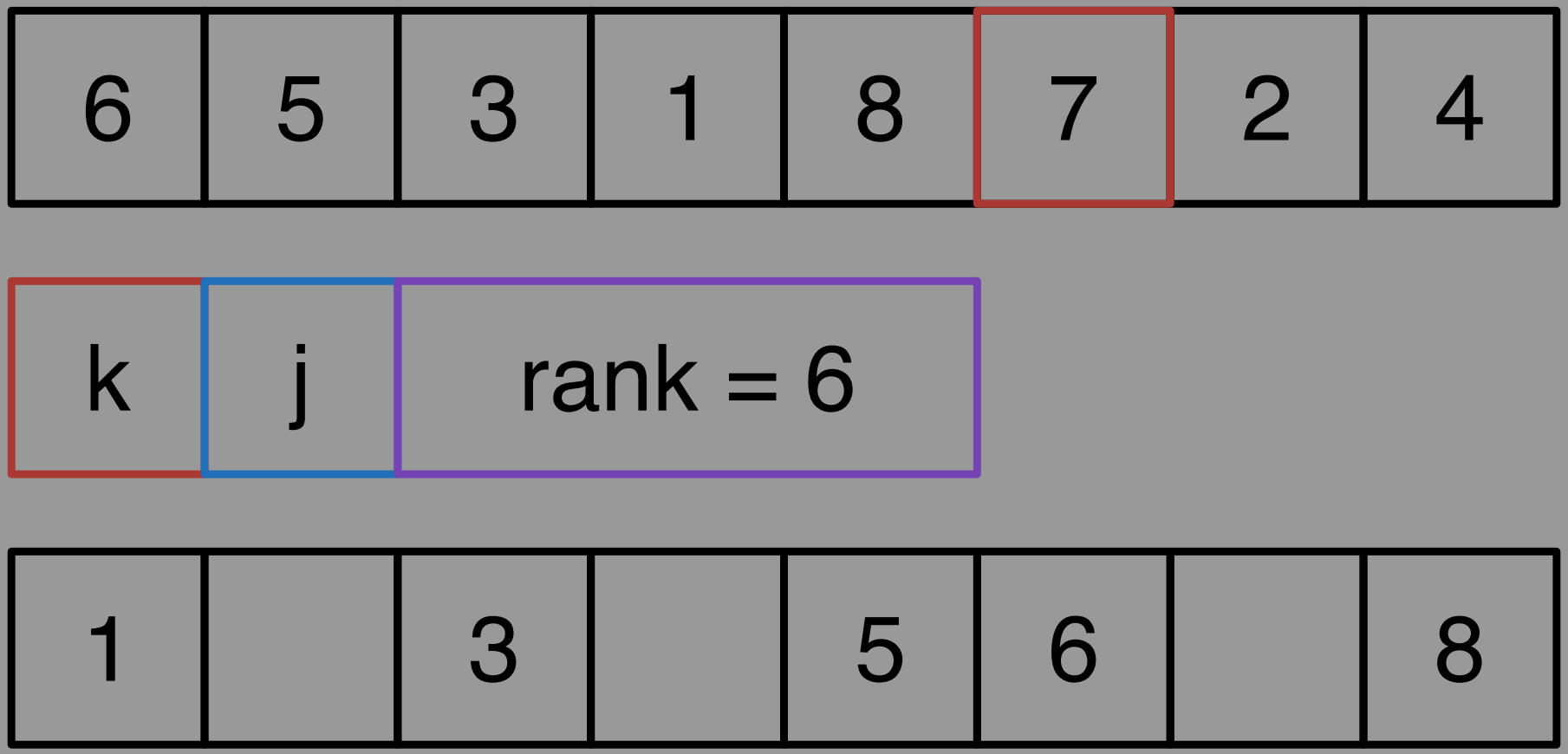


Rank Sort Example

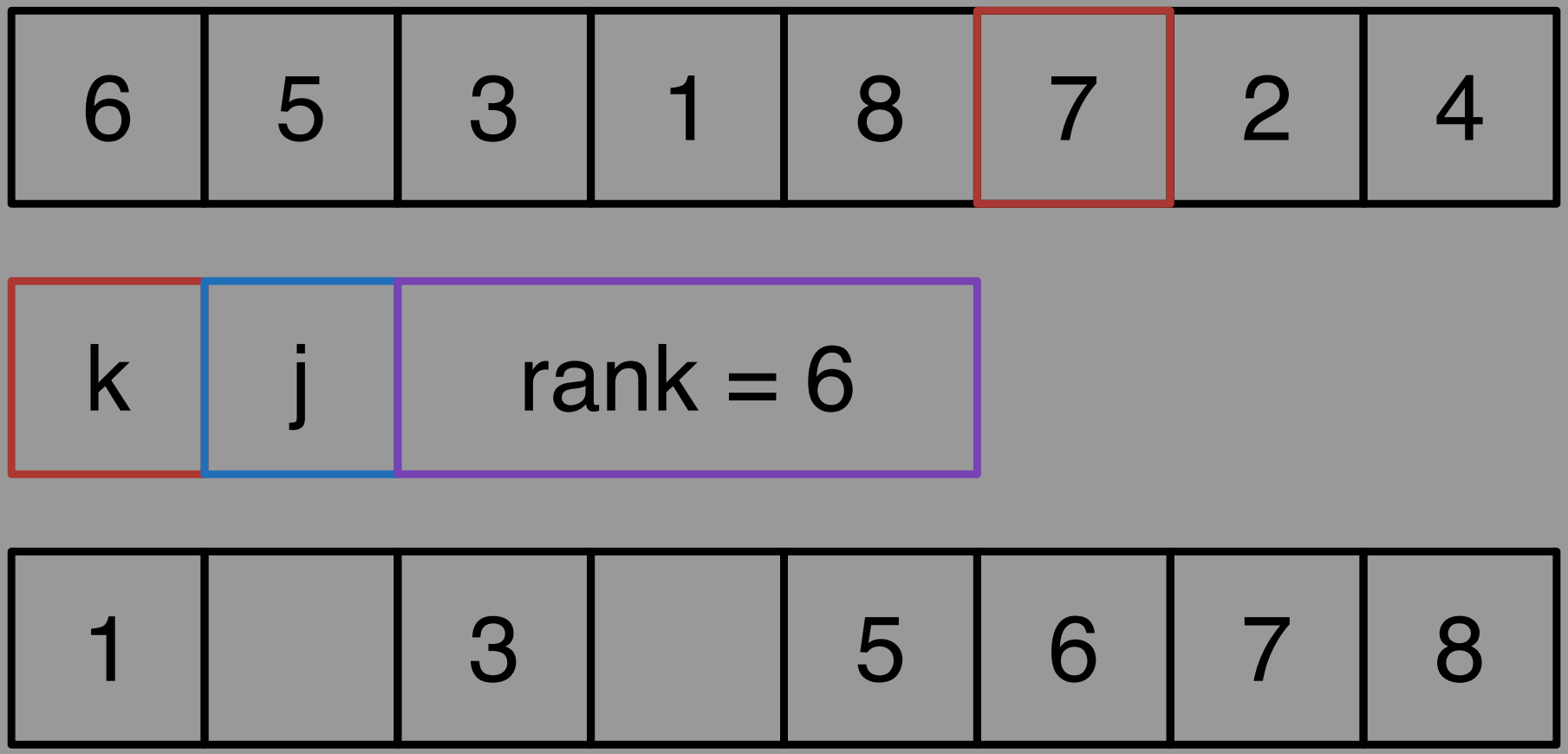




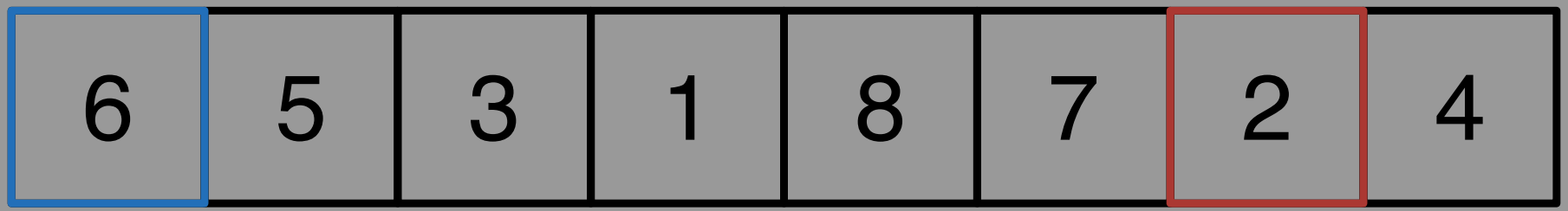
Rank Sort Example

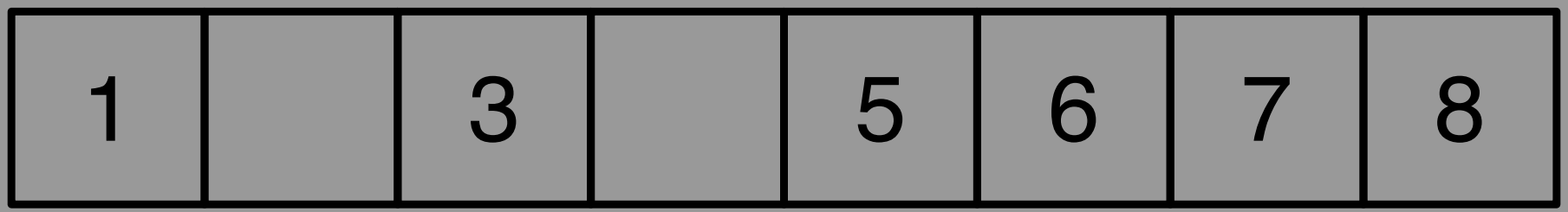


Rank Sort Example

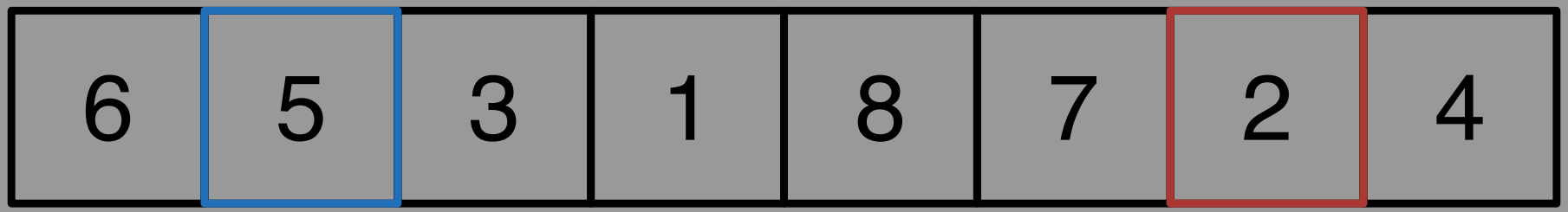


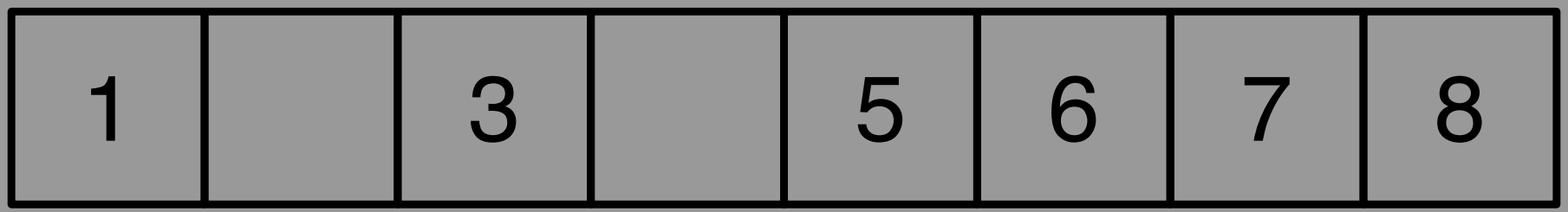
Rank Sort Example



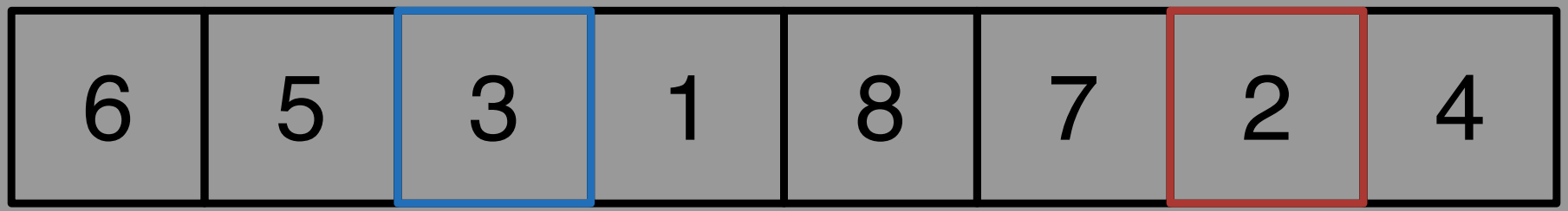


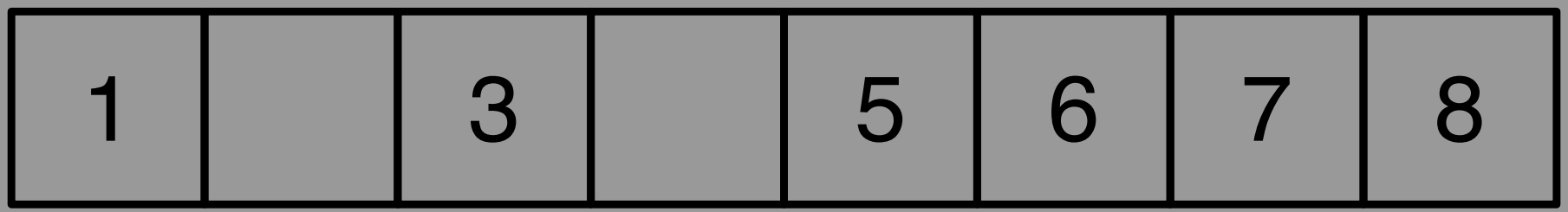
Rank Sort Example



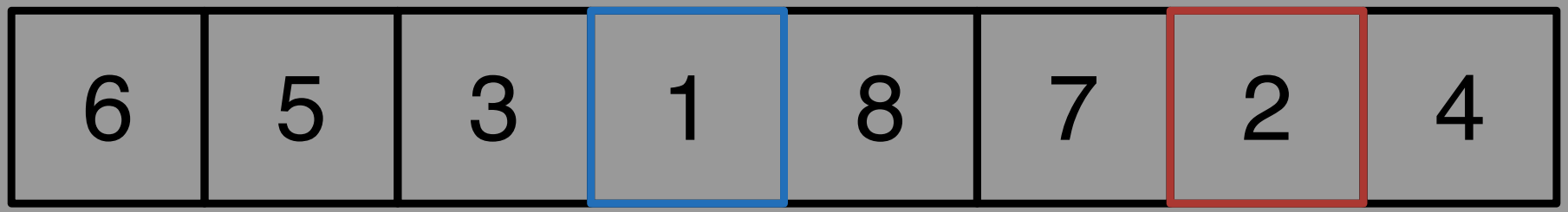


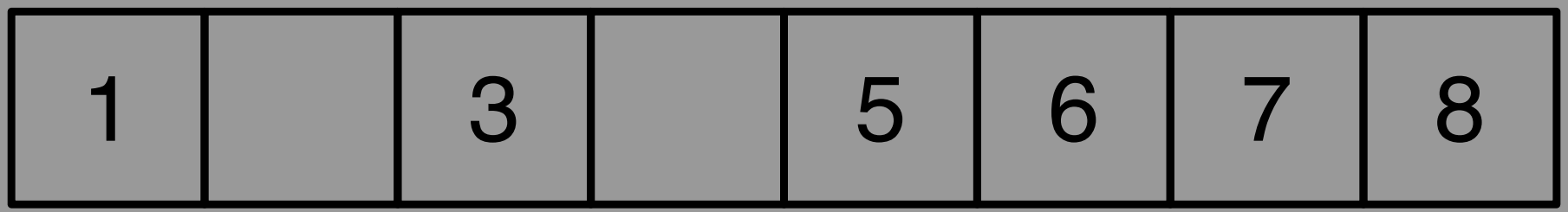
Rank Sort Example



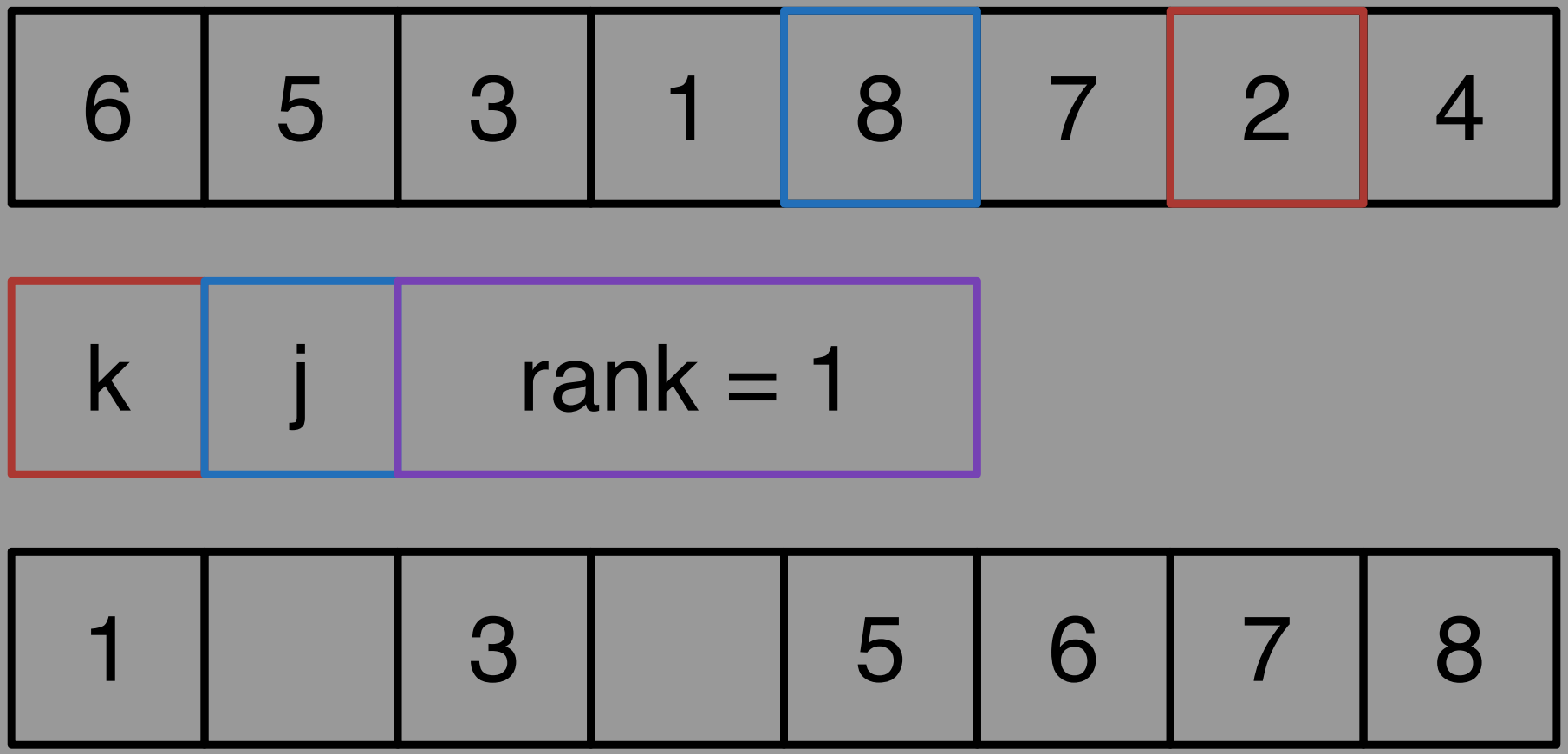


Rank Sort Example

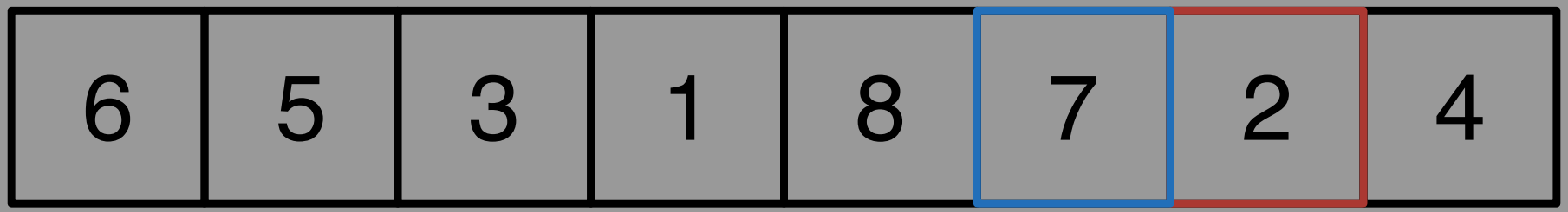


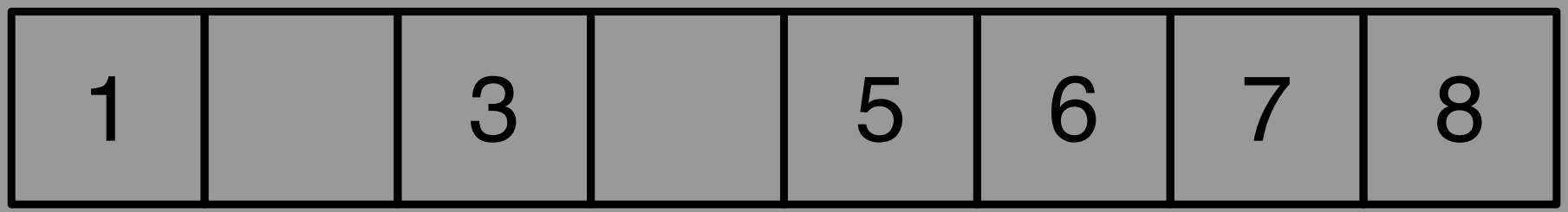


Rank Sort Example

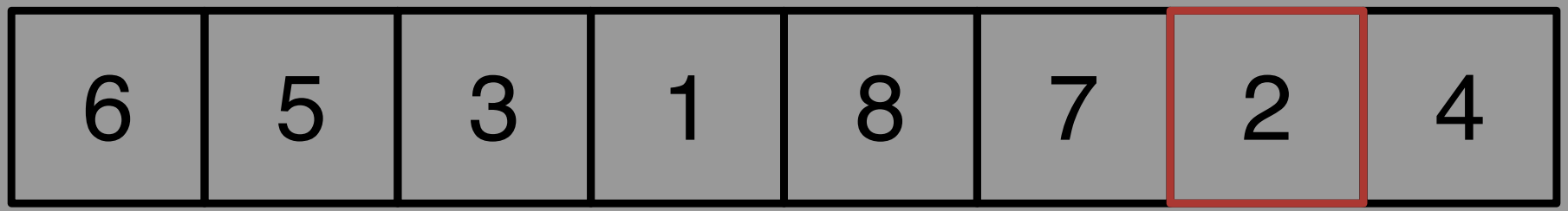


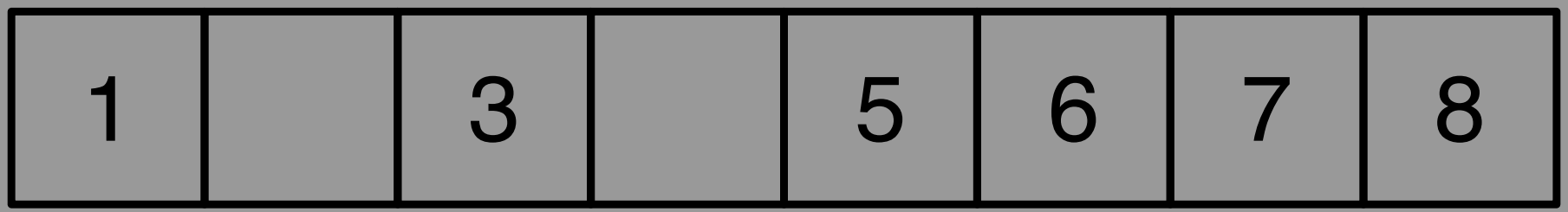
Rank Sort Example



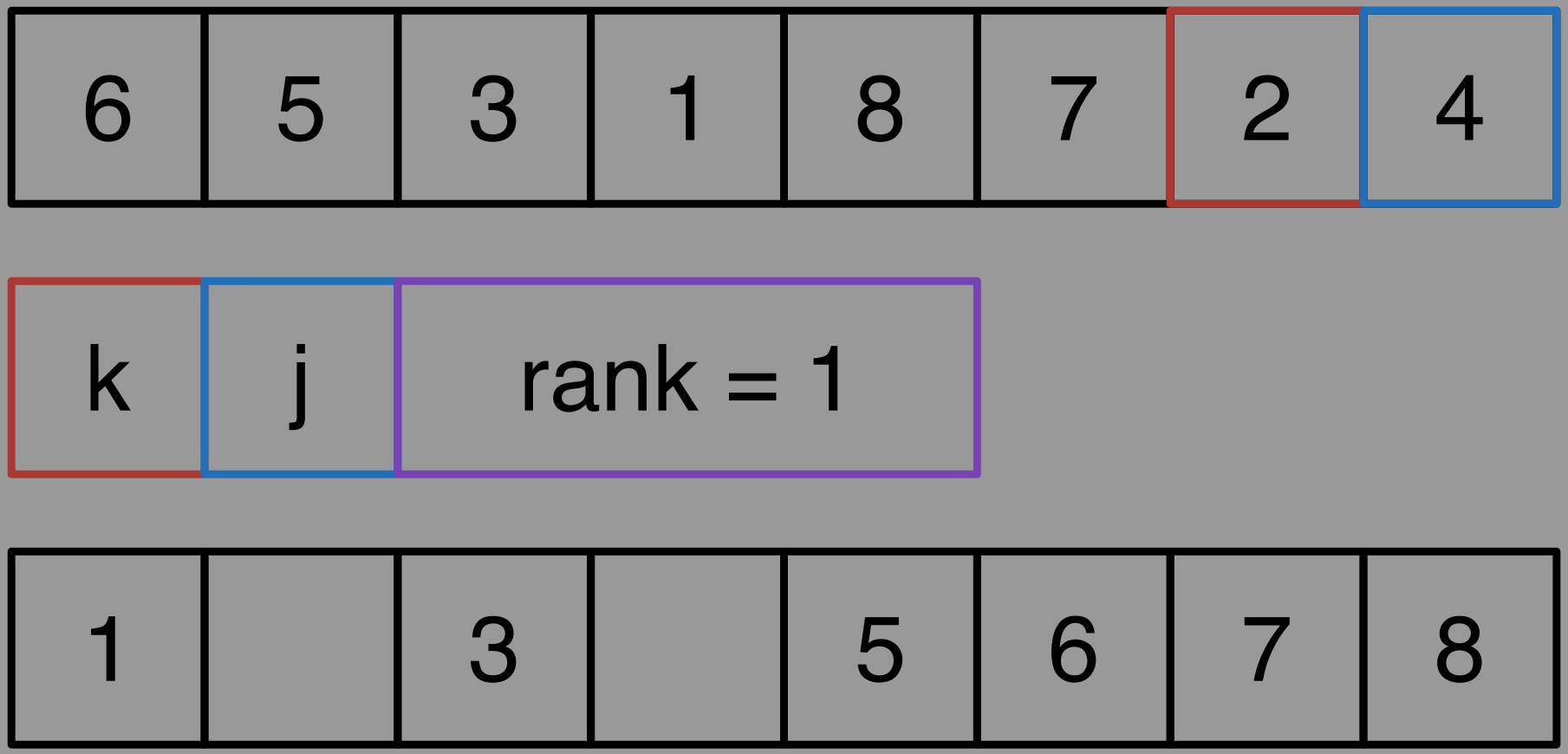


Rank Sort Example

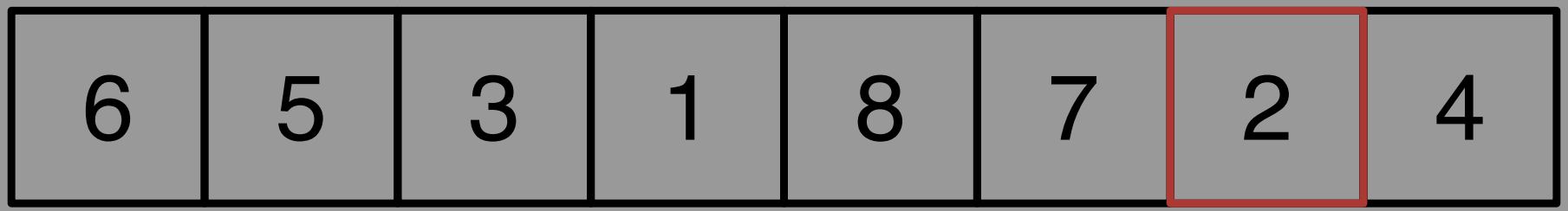


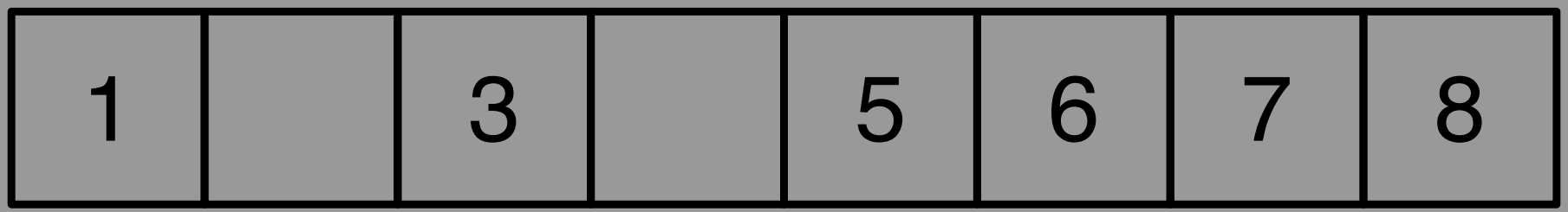


Rank Sort Example

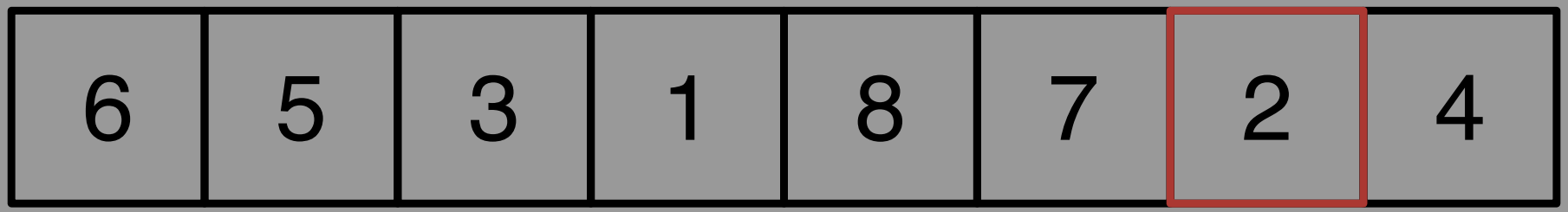


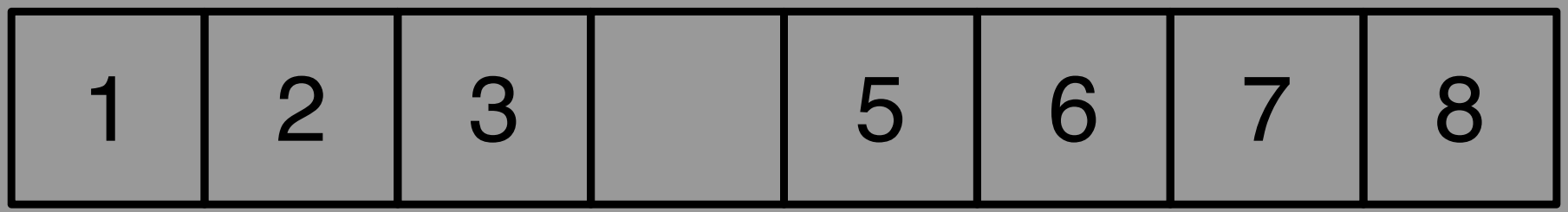
Rank Sort Example



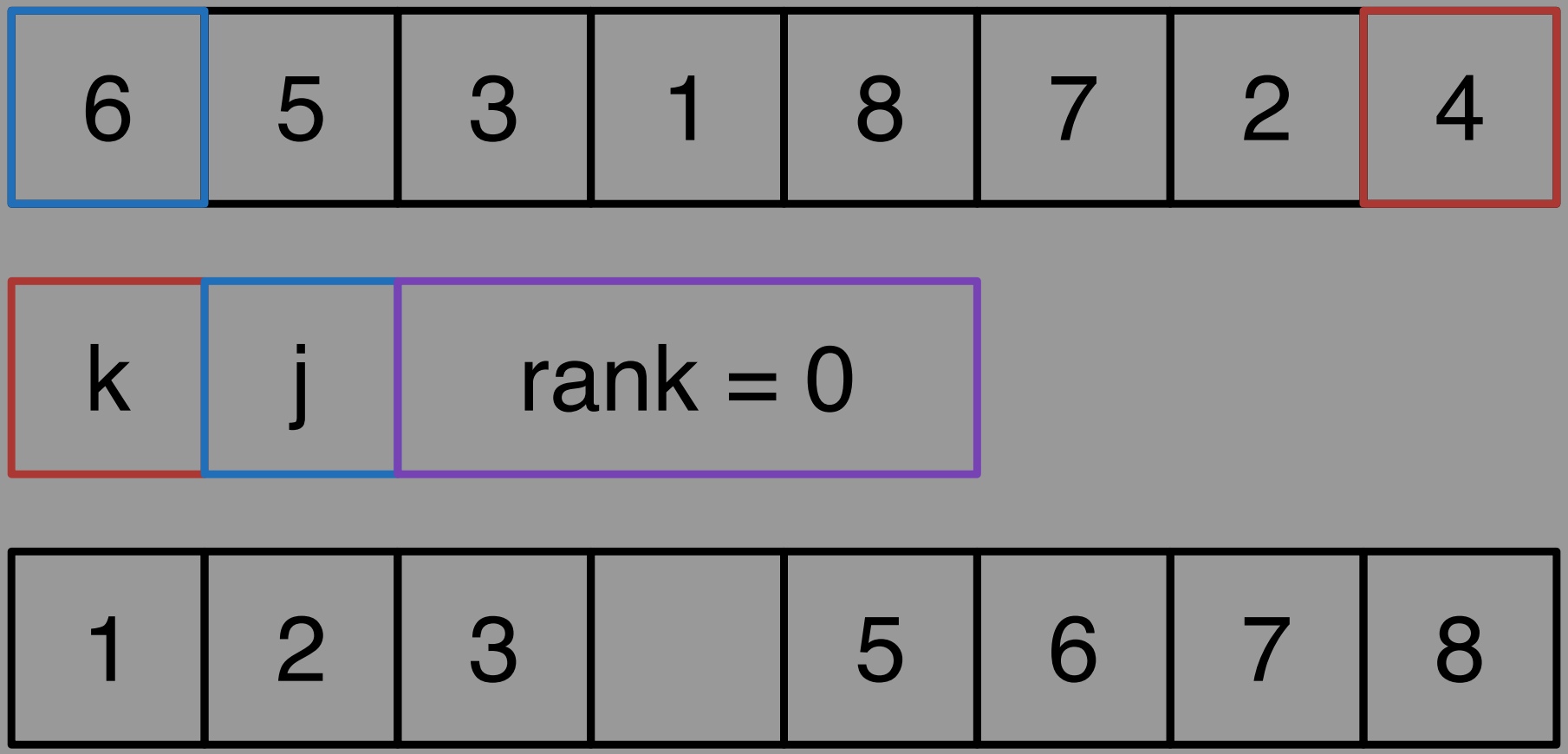


Rank Sort Example

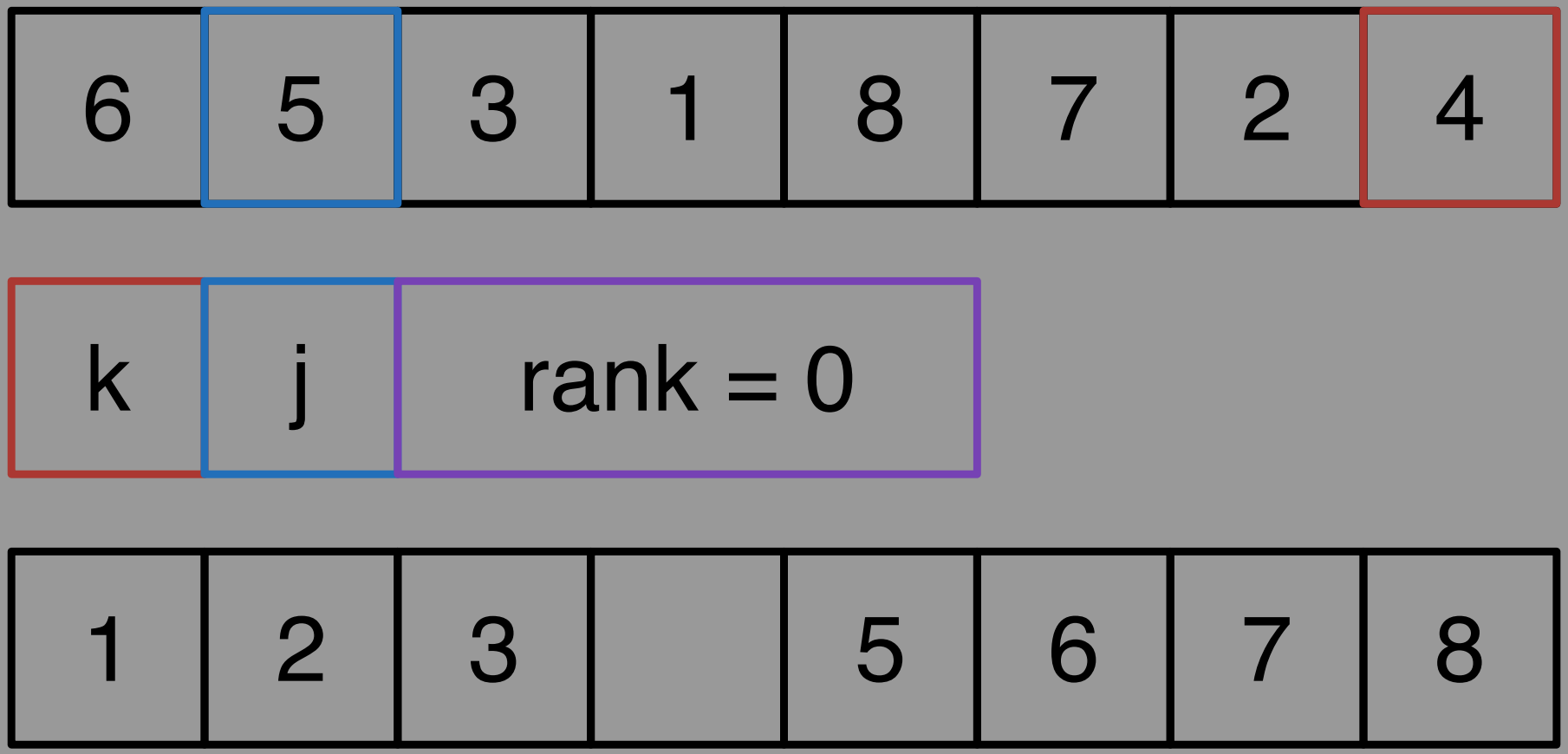




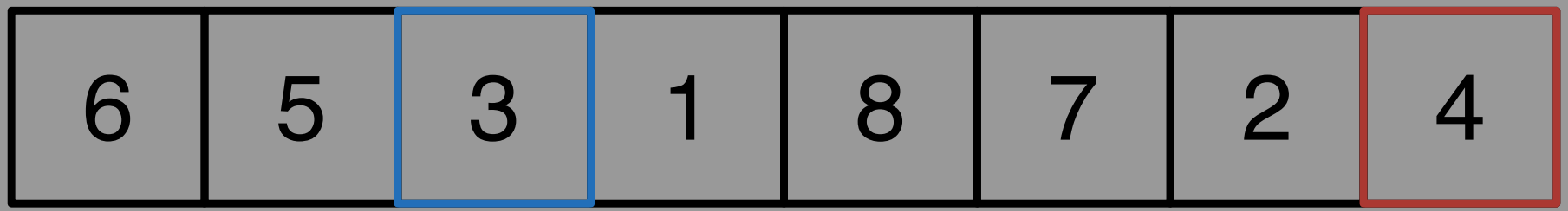
Rank Sort Example

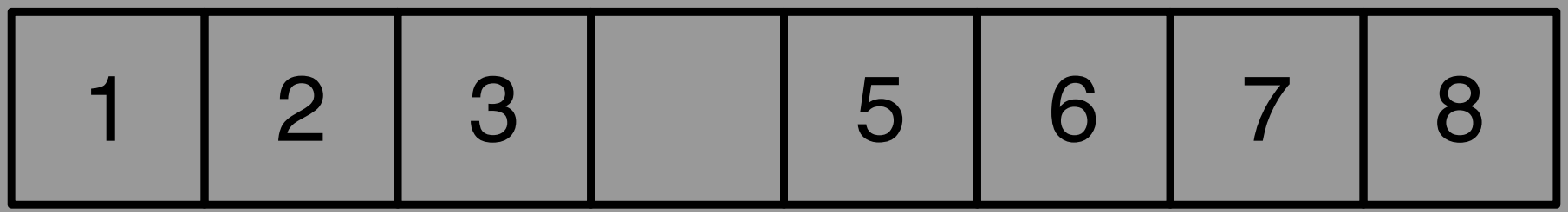


Rank Sort Example

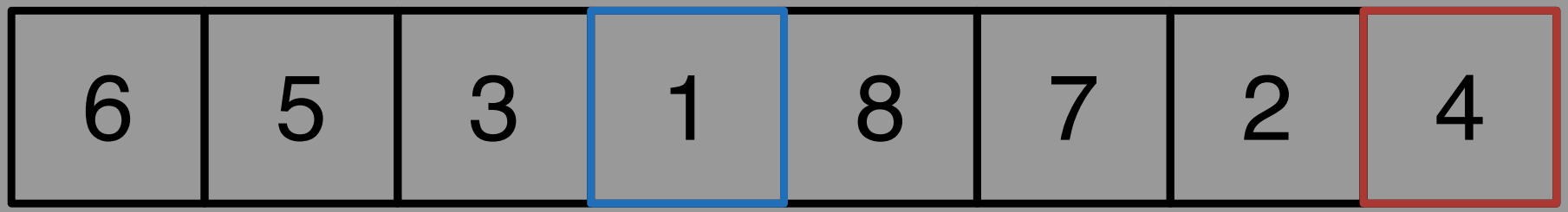


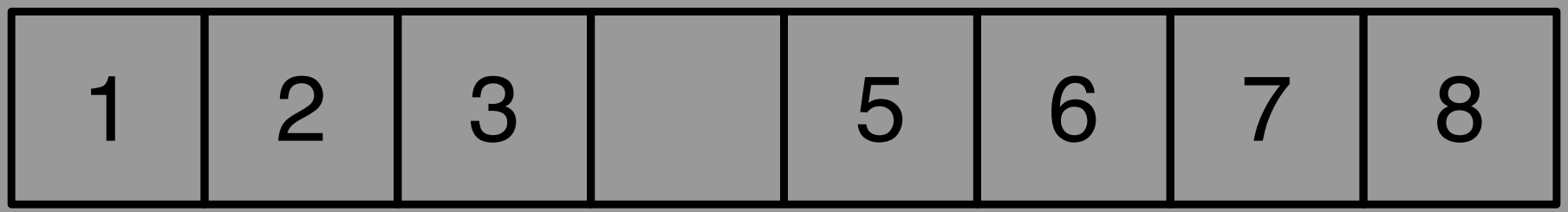
Rank Sort Example



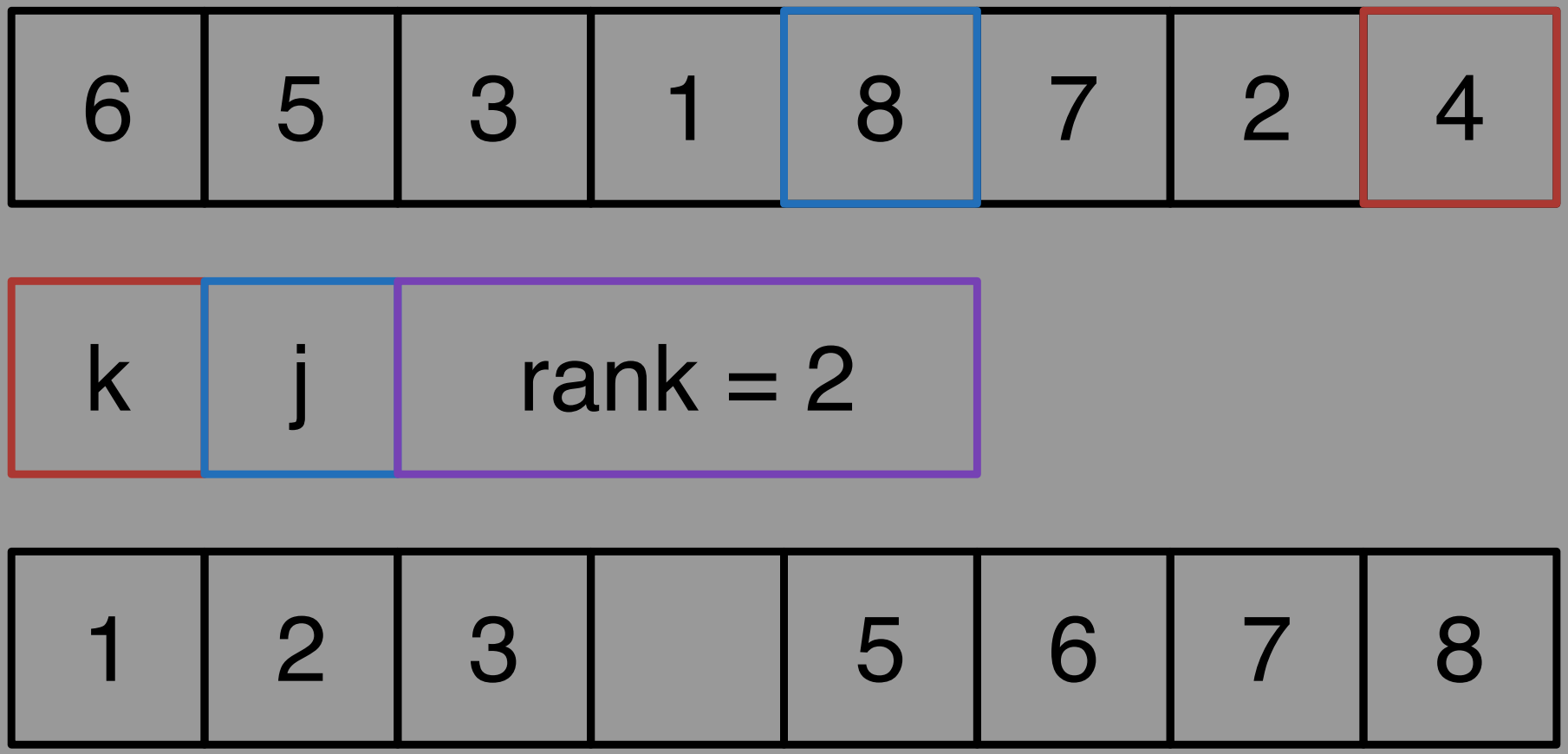


Rank Sort Example

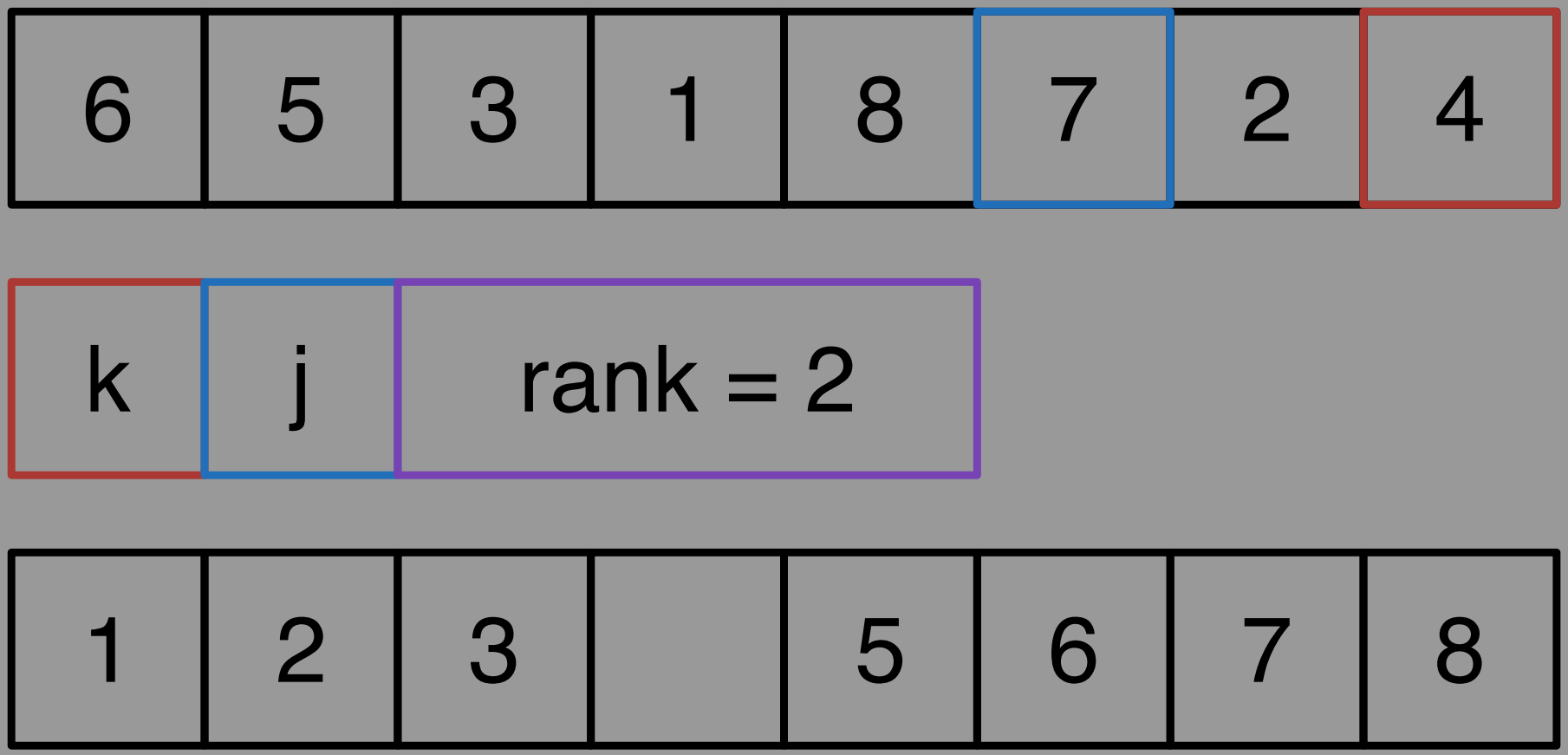




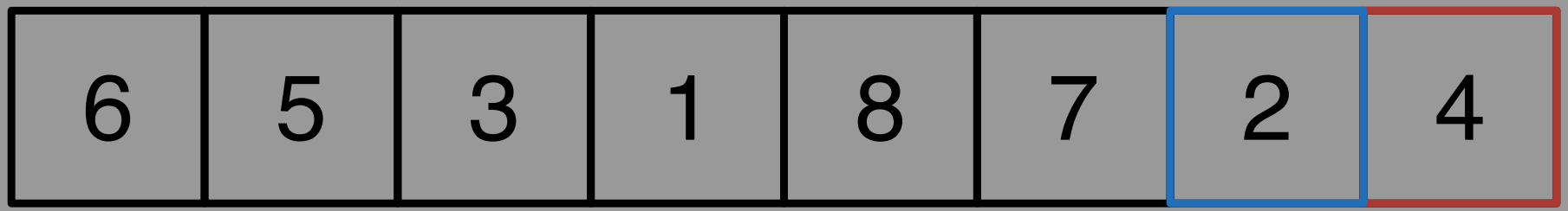
Rank Sort Example

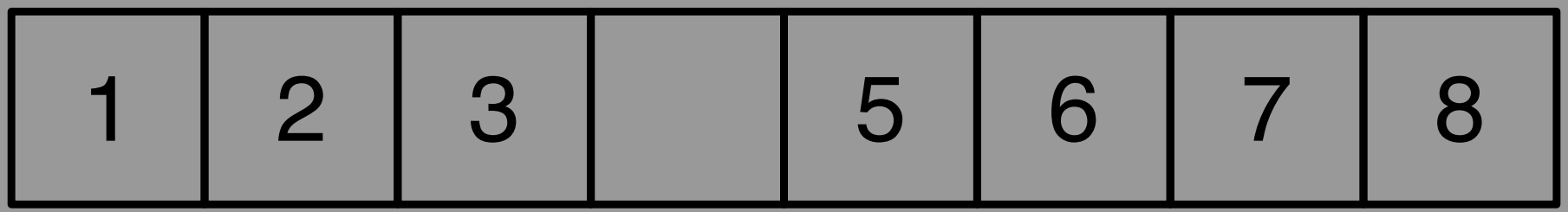


Rank Sort Example

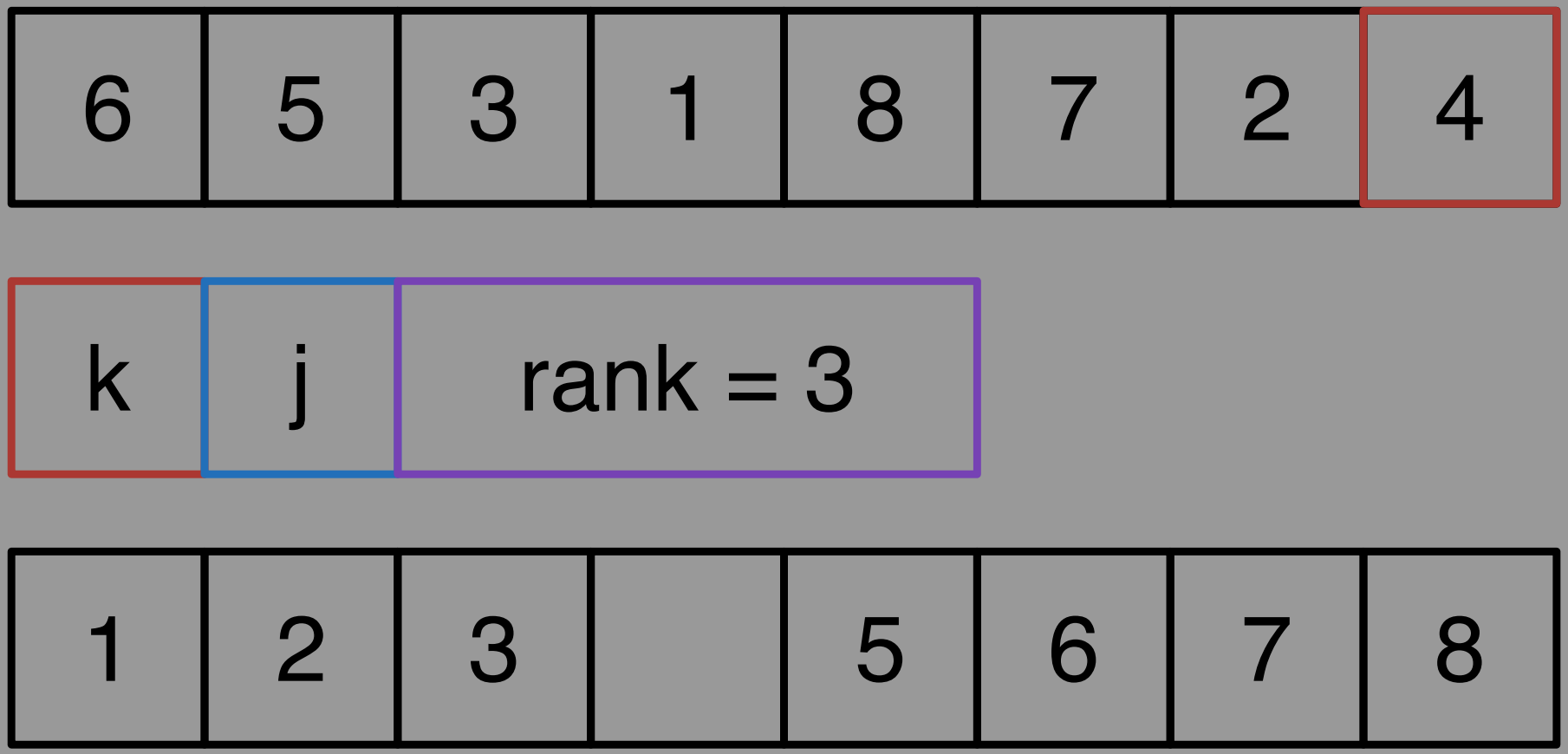


Rank Sort Example

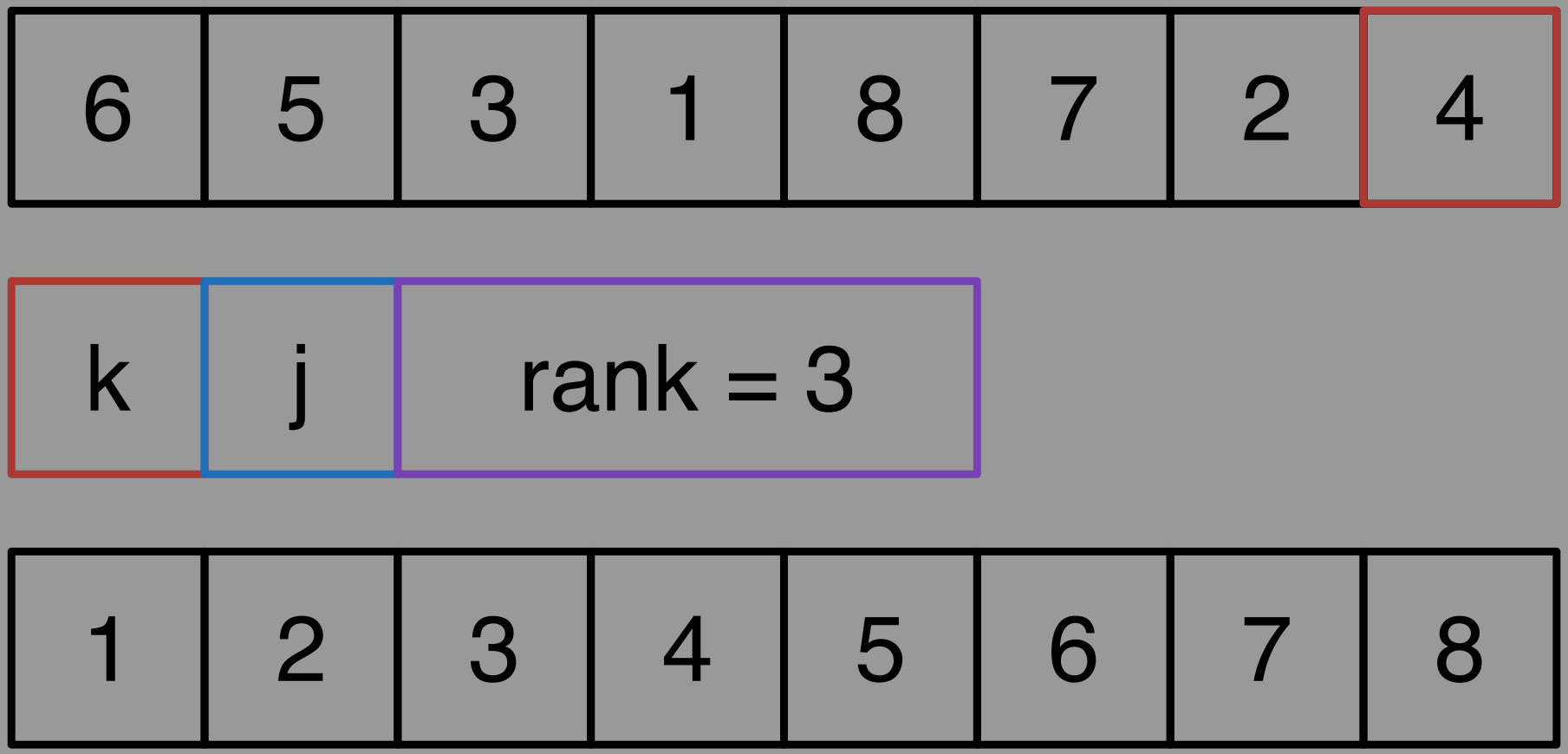




Rank Sort Example



Rank Sort Example



Complexity of Rank Sort

What is the complexity of rank sort?

- $O(n^2)$

When compared with selection sort, can we tell which is better

- No, they are the same

Table of Contents

Insertion Sort

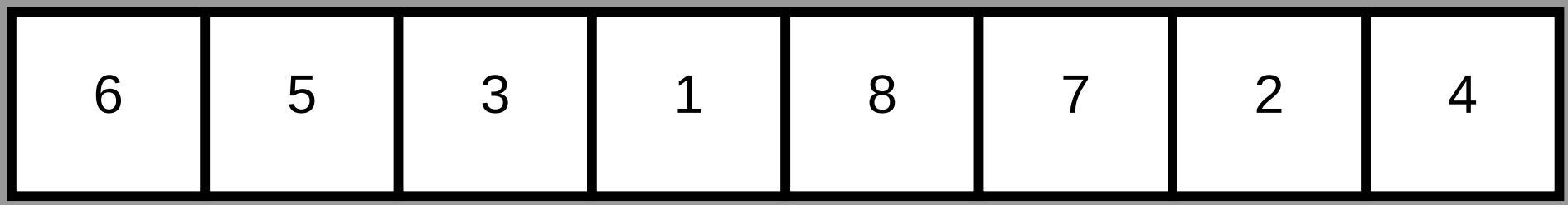
Insertion sort iterates through the array and keeps part of the array in sorted order until it reaches the end

- On every step of the outer loop we add a new number to the sorted part of the array in the correct place

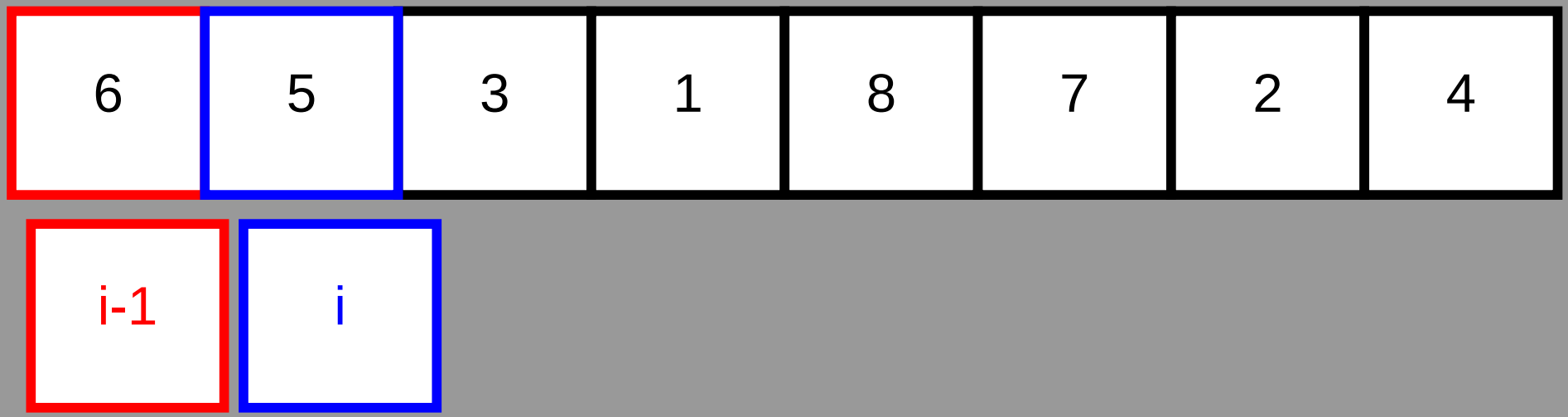
Insertion Sort

```
1  void inSort(int k[], int n){
2     int j = 0;
3     while(j < n){
4        int i = j;
5        while(i > 0 && k[i] < k[i-1]){
6           int temp = k[i];
7           k[i] = k[i-1];
8           k[i-1] = temp;
9           i = i - 1;
10       }
11       j = j + 1;
12    }
13 }
```

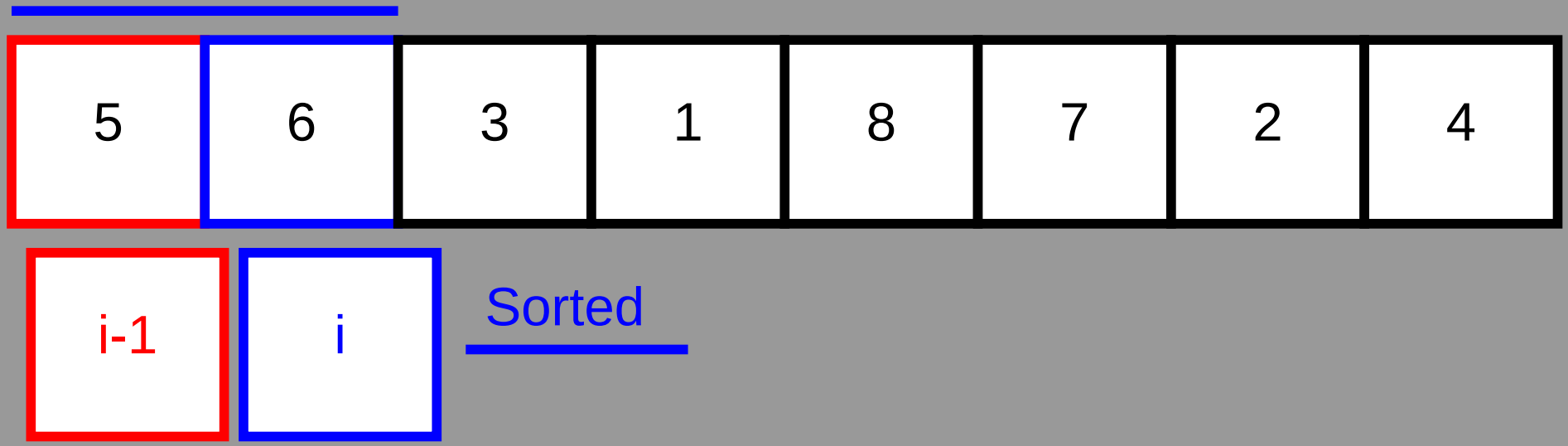
Insertion Sort Example



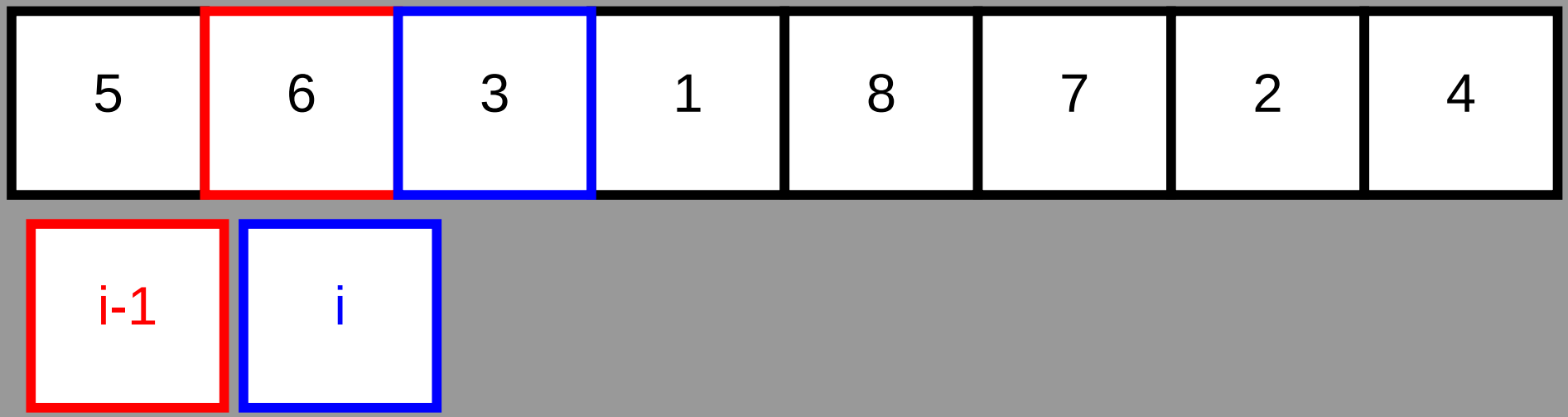
Insertion Sort Example



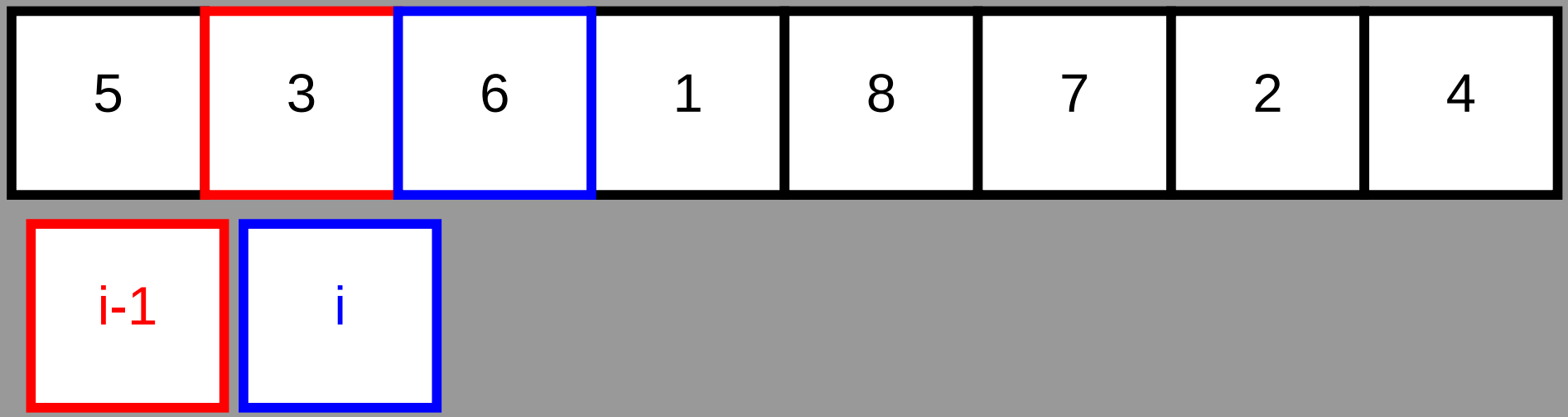
Insertion Sort Example



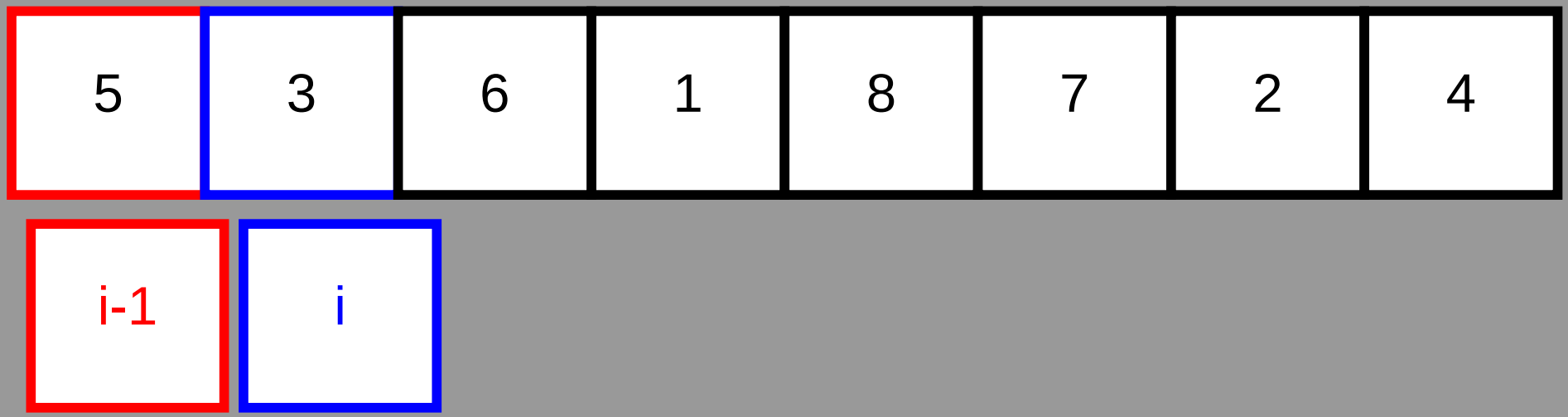
Insertion Sort Example



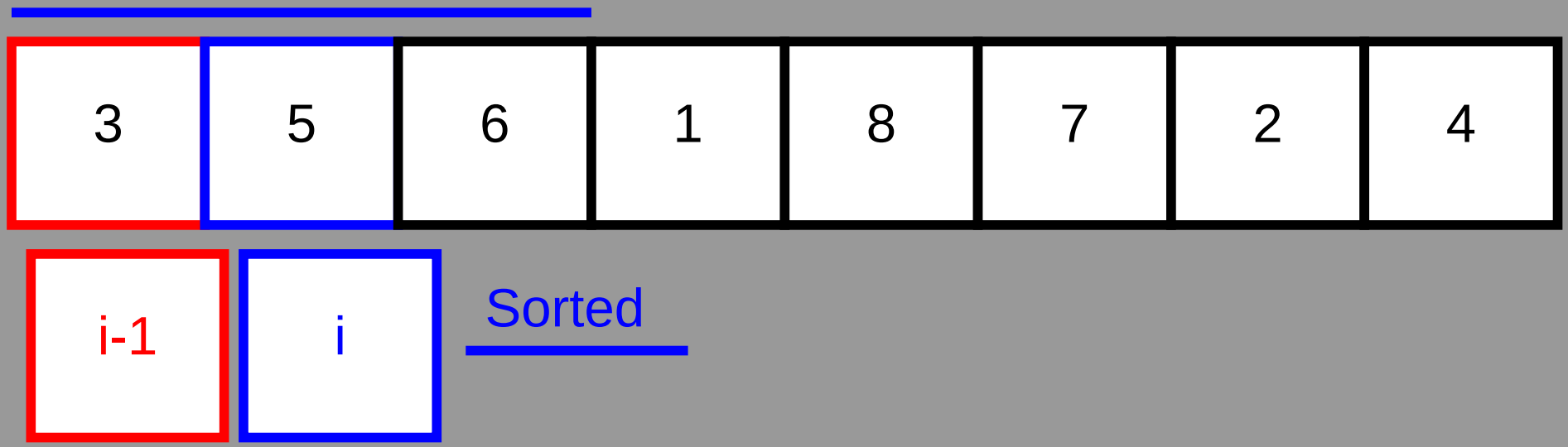
Insertion Sort Example



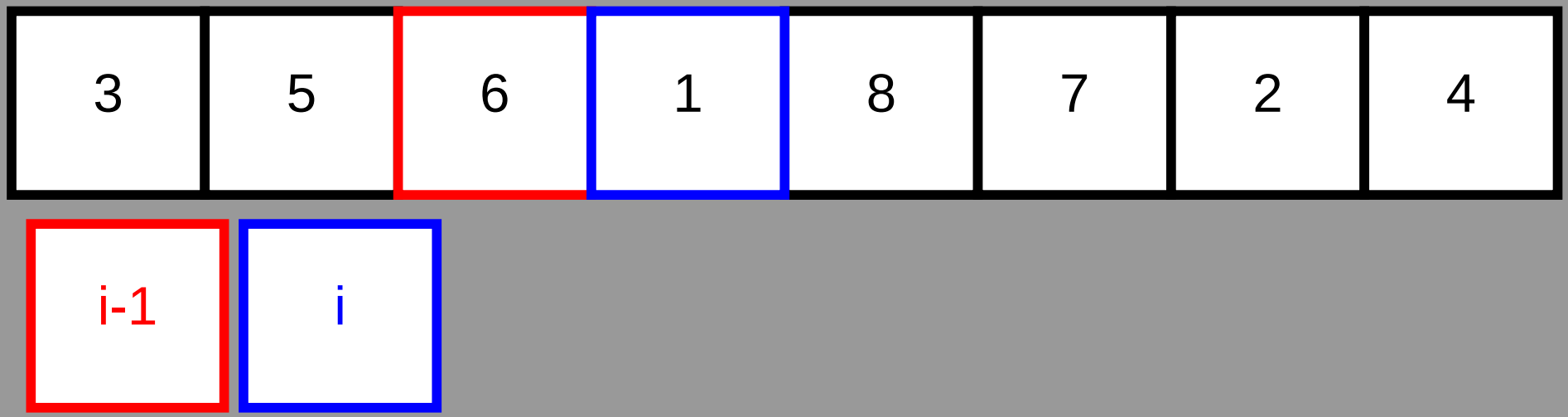
Insertion Sort Example



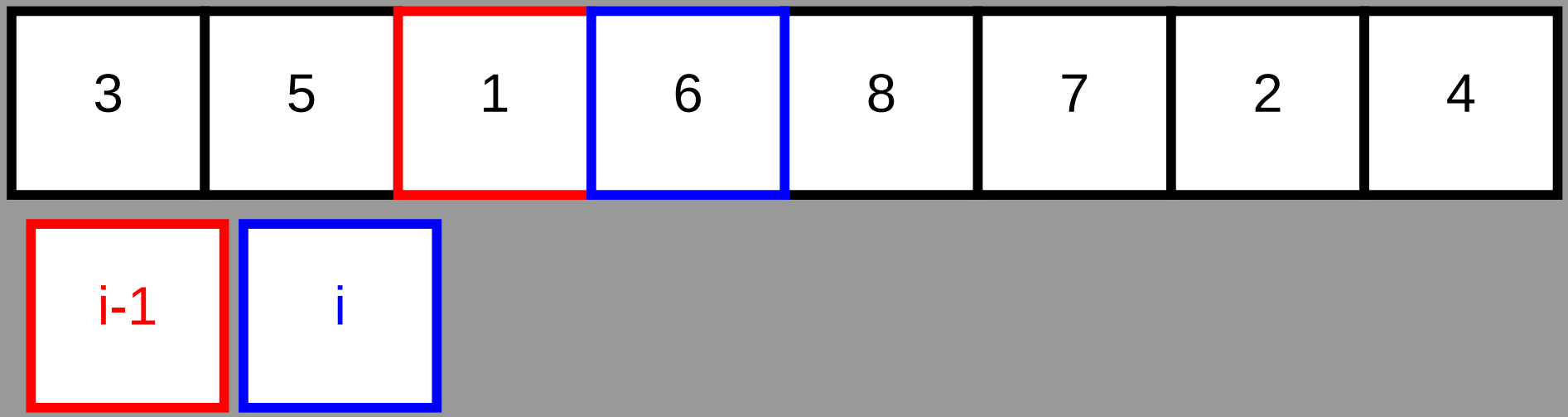
Insertion Sort Example



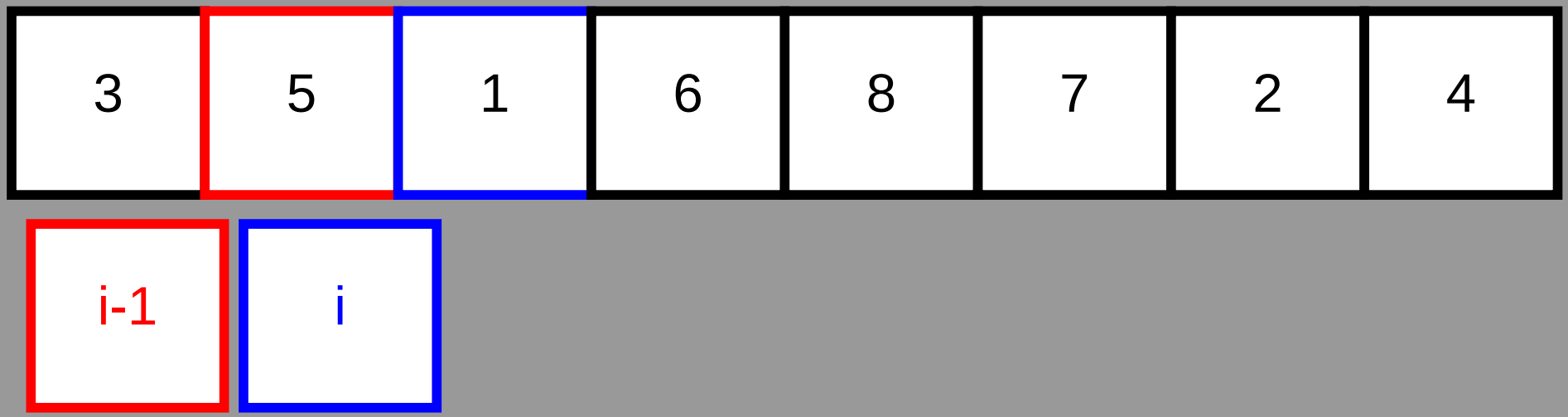
Insertion Sort Example



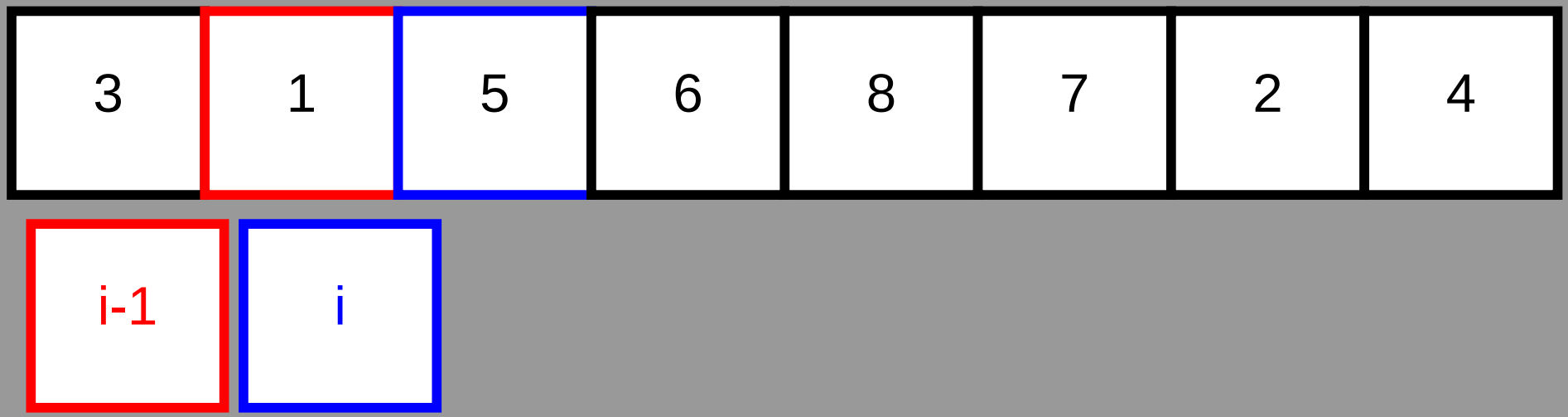
Insertion Sort Example



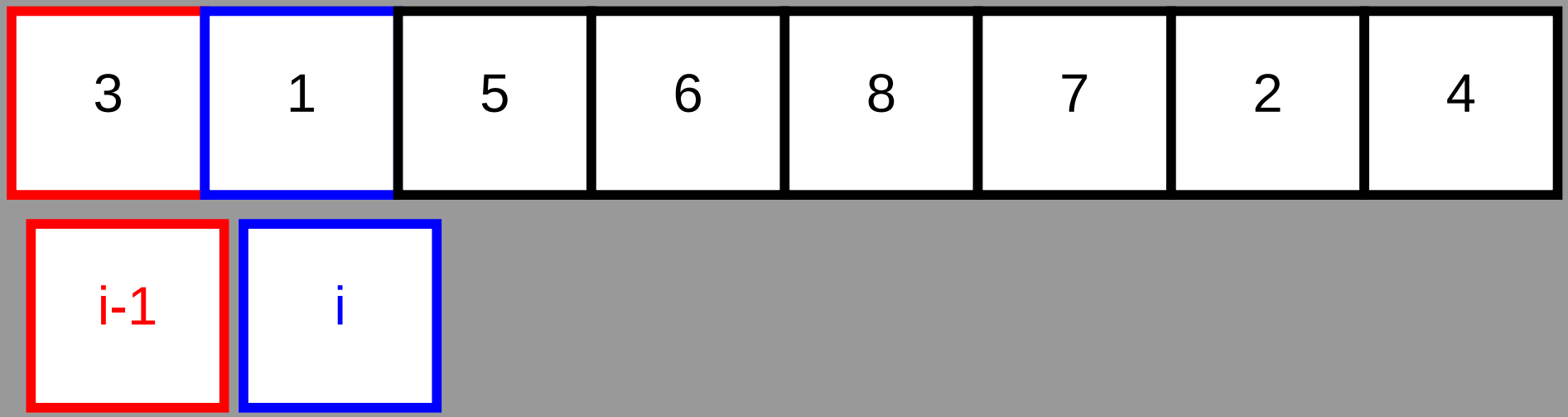
Insertion Sort Example



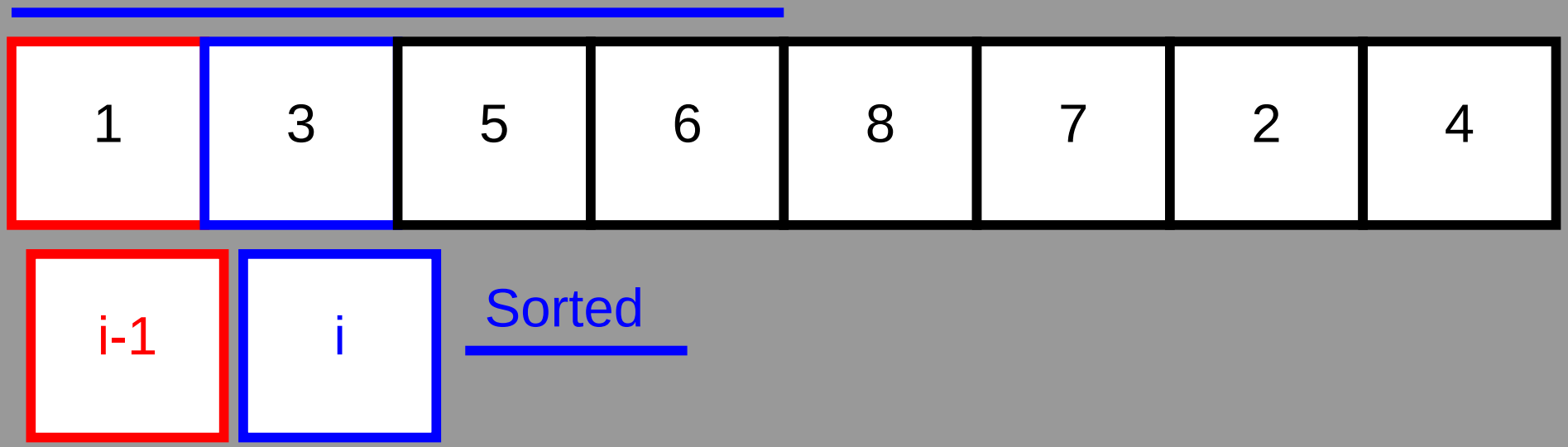
Insertion Sort Example



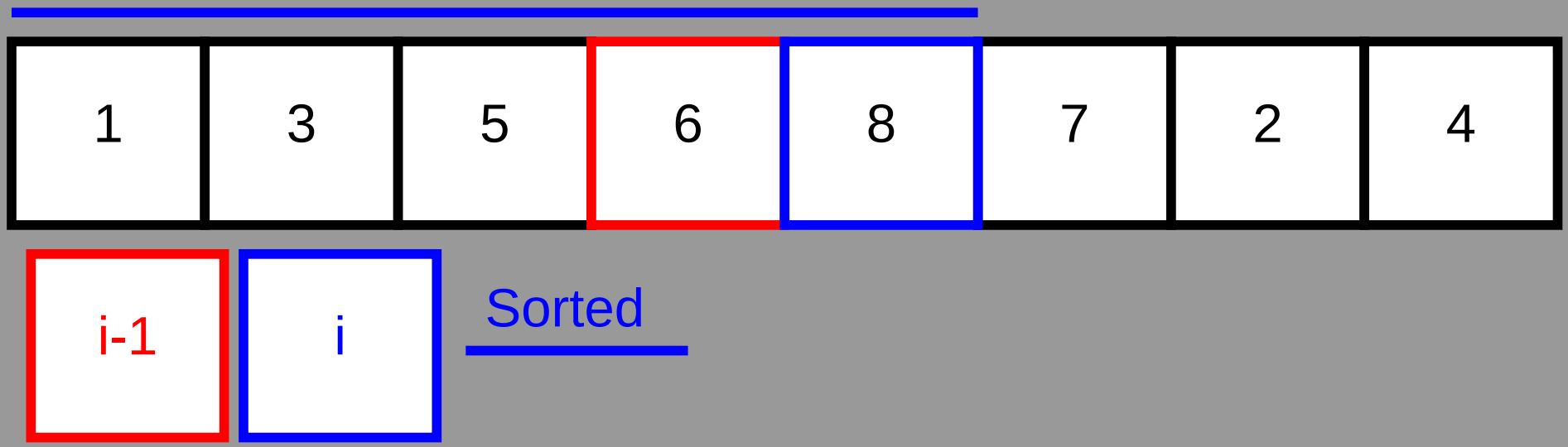
Insertion Sort Example



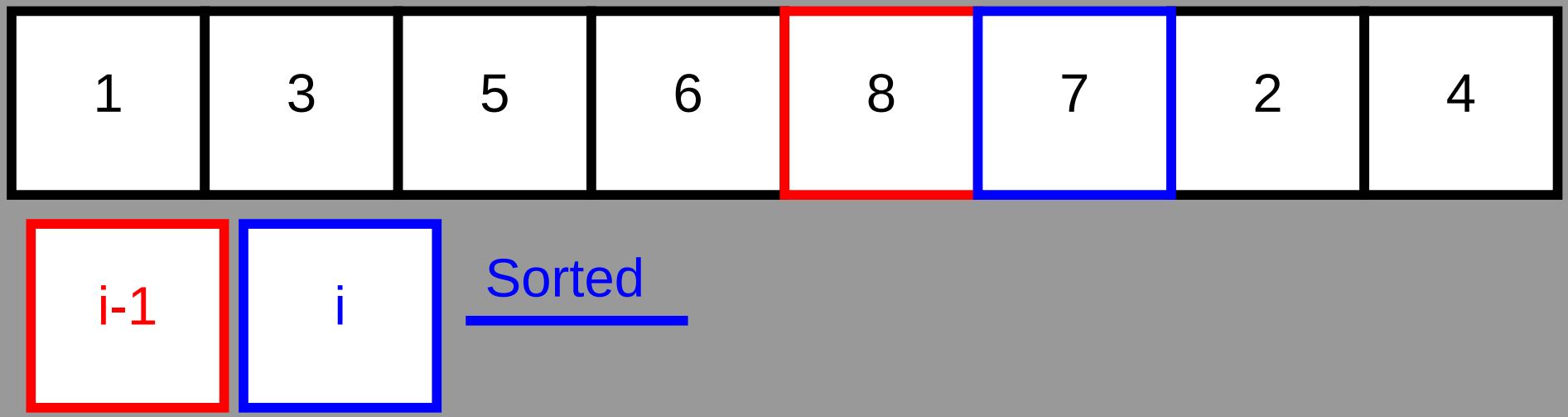
Insertion Sort Example



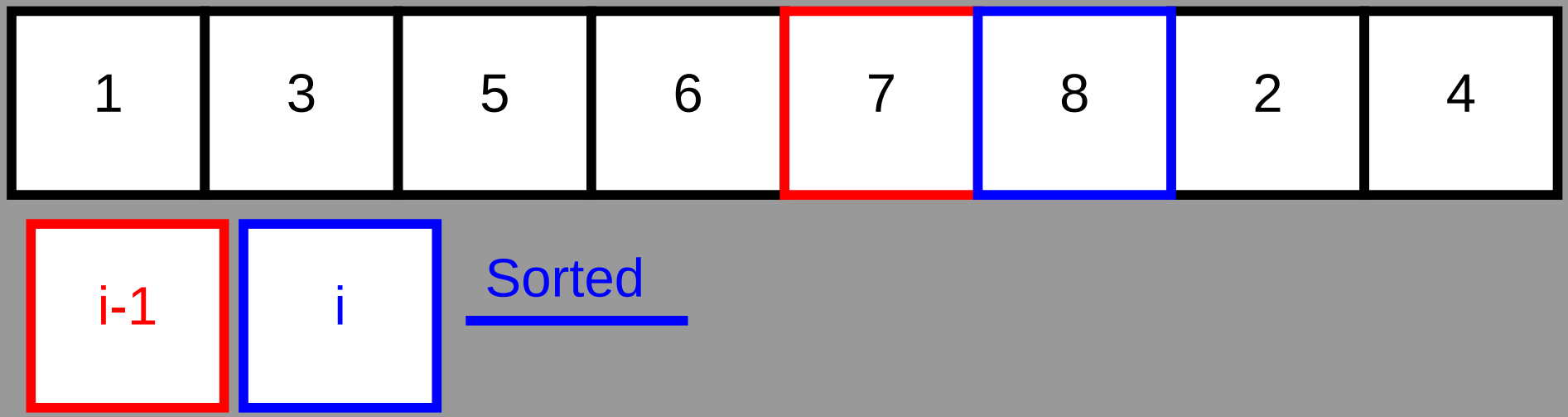
Insertion Sort Example



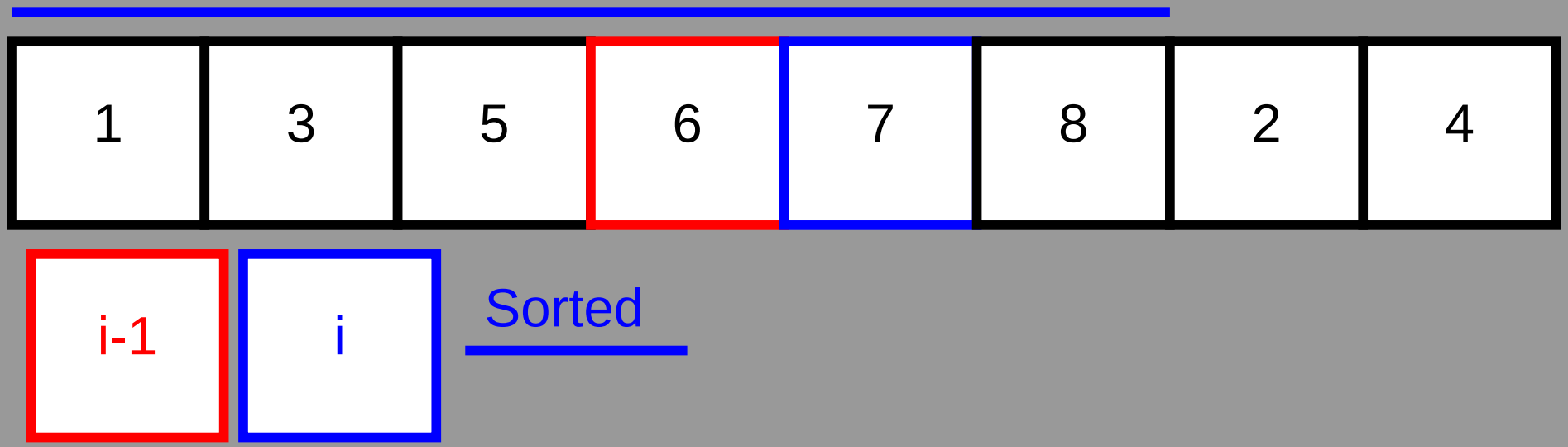
Insertion Sort Example



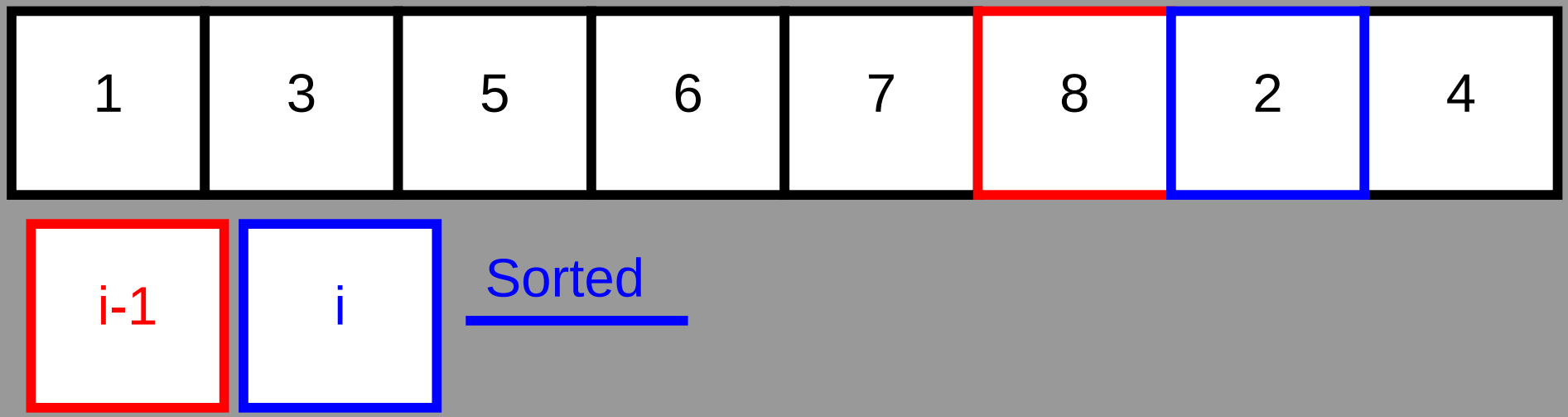
Insertion Sort Example



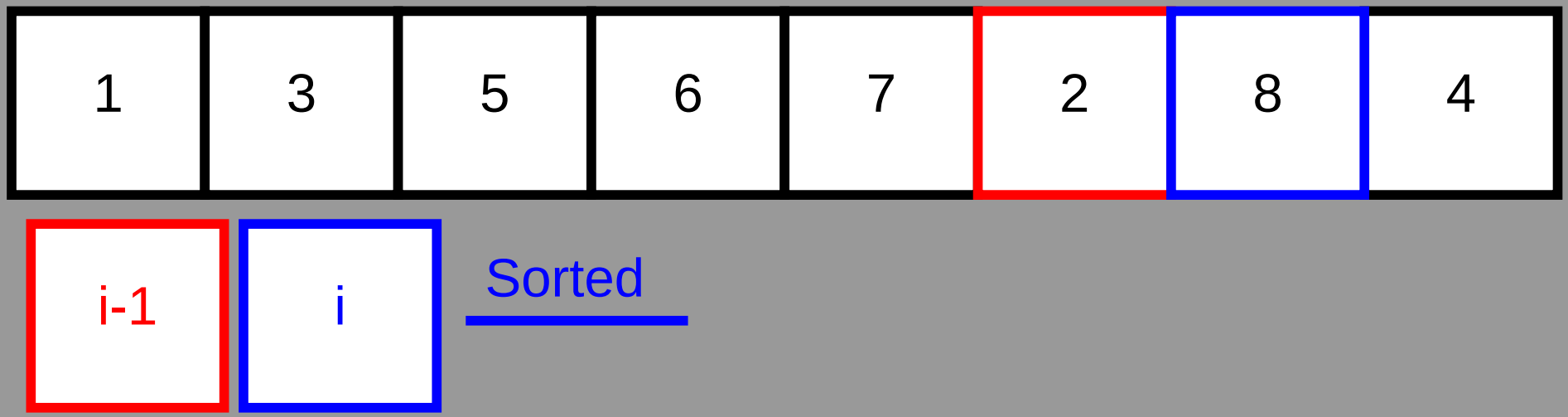
Insertion Sort Example



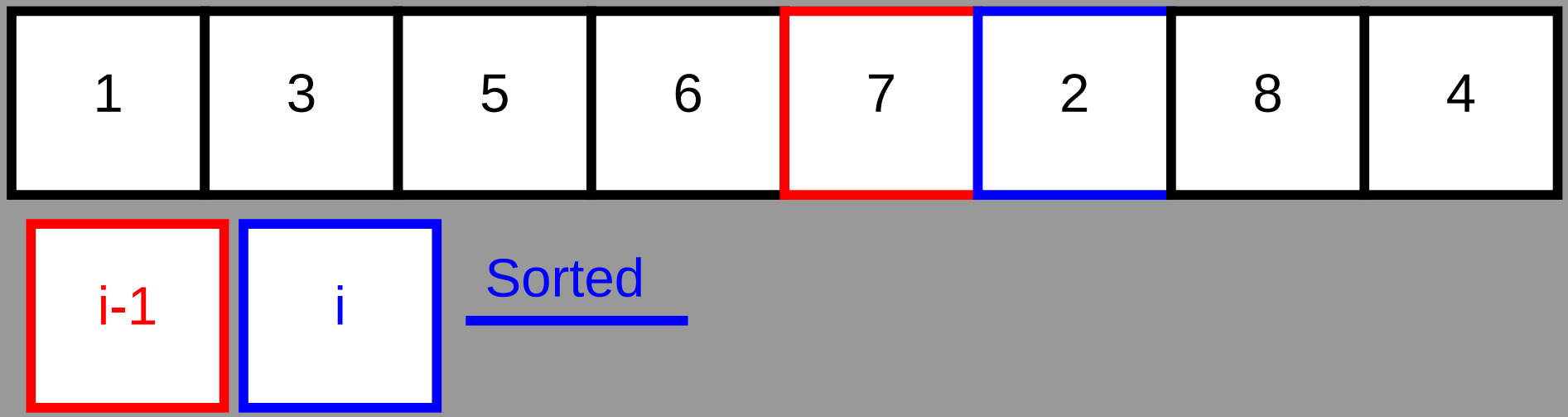
Insertion Sort Example



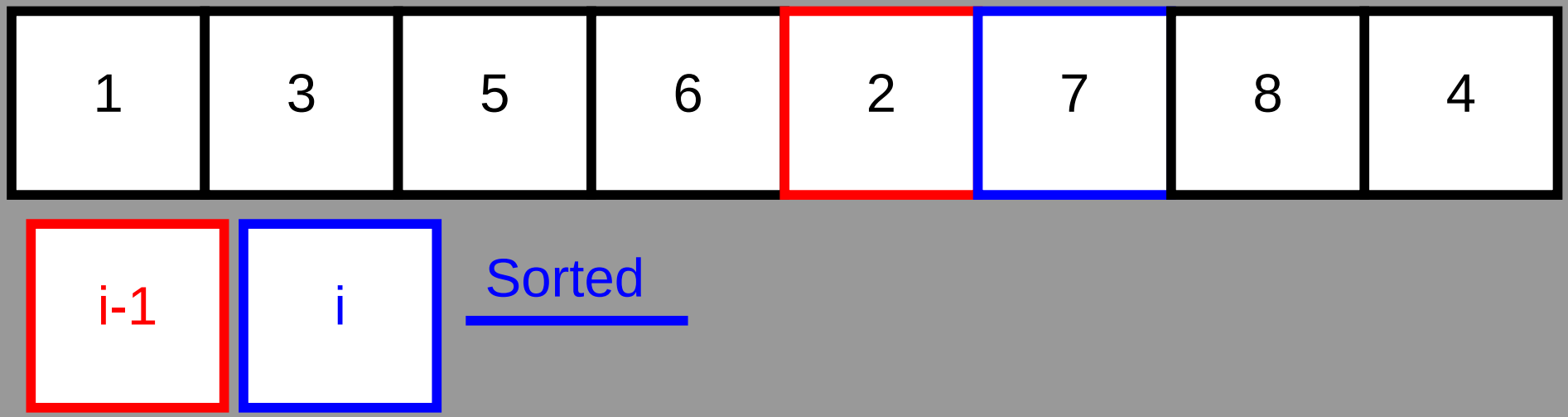
Insertion Sort Example



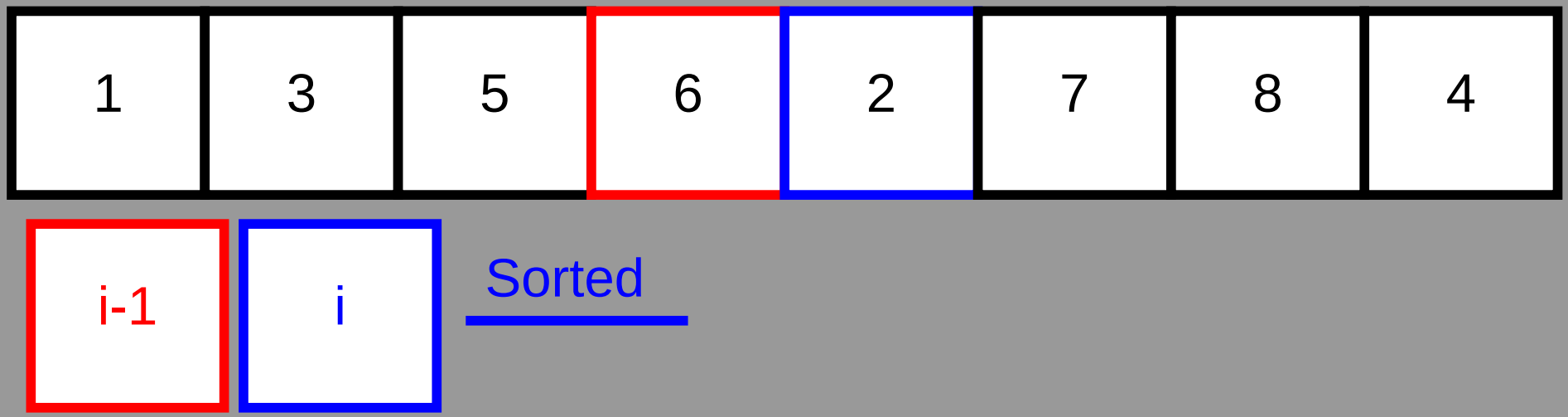
Insertion Sort Example



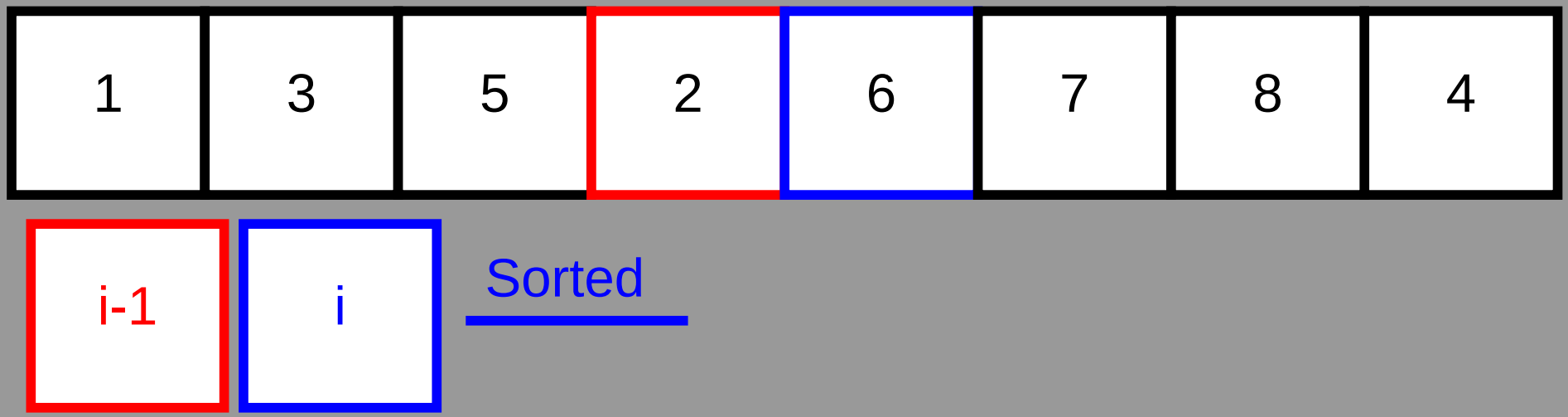
Insertion Sort Example



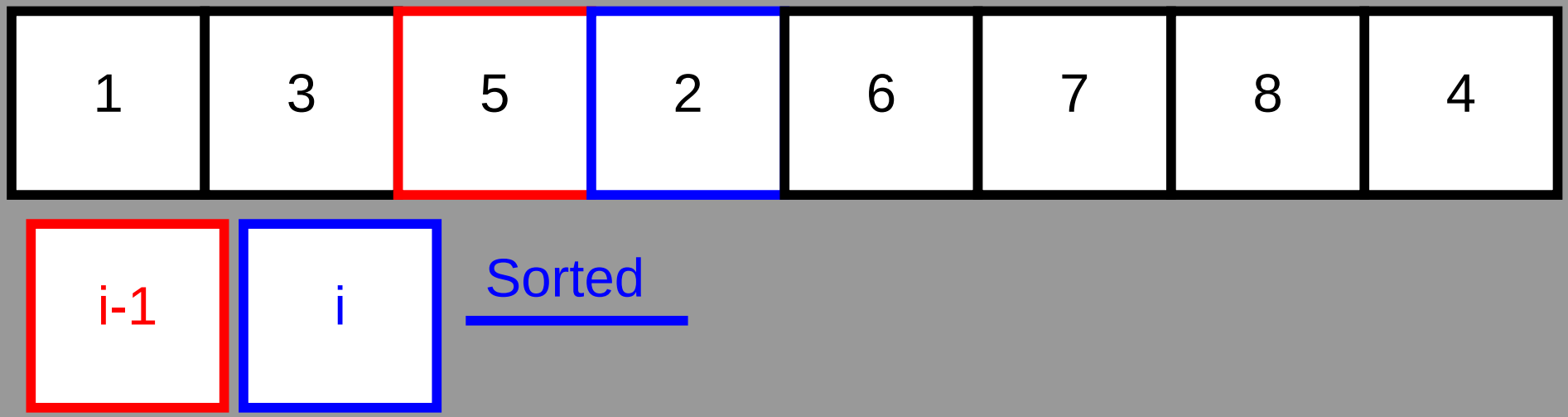
Insertion Sort Example



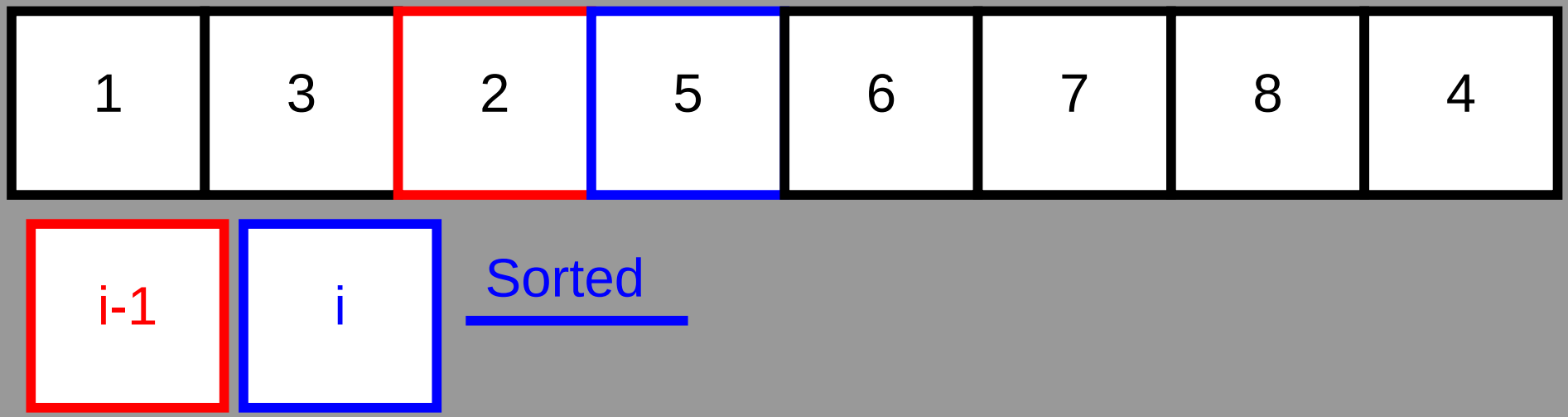
Insertion Sort Example



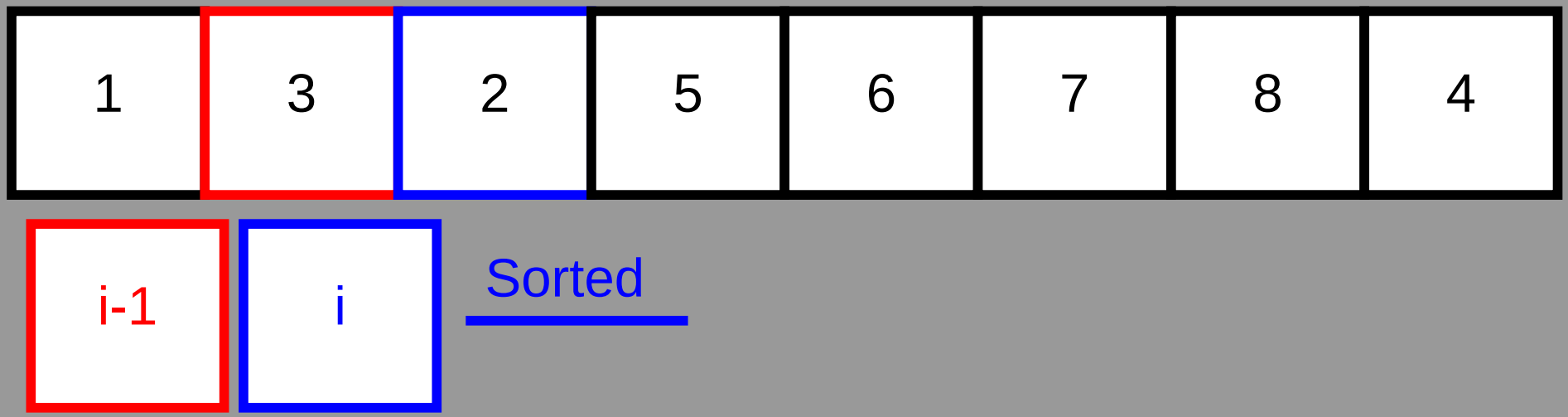
Insertion Sort Example



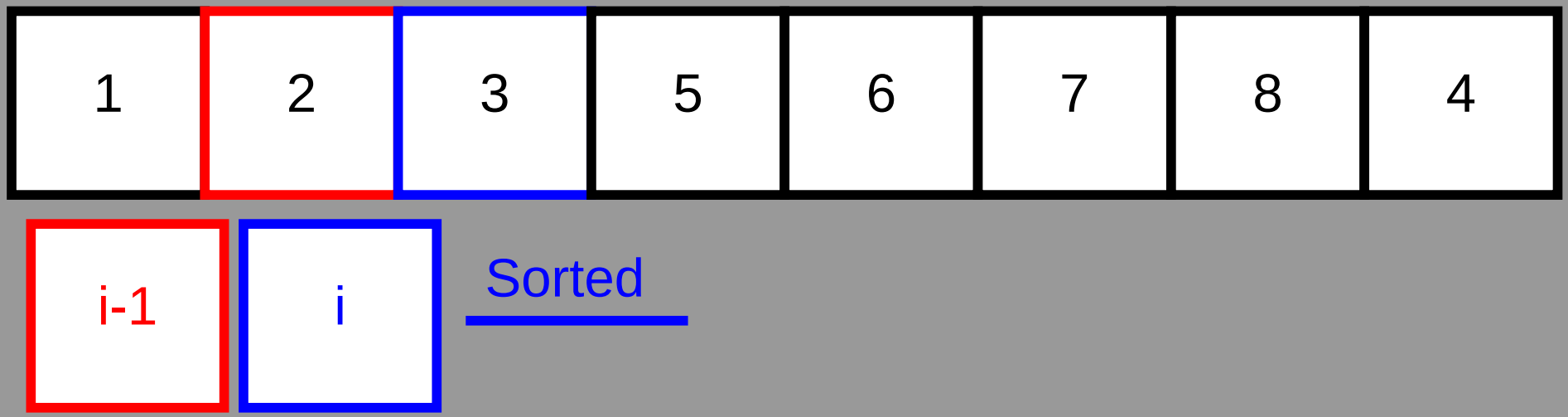
Insertion Sort Example



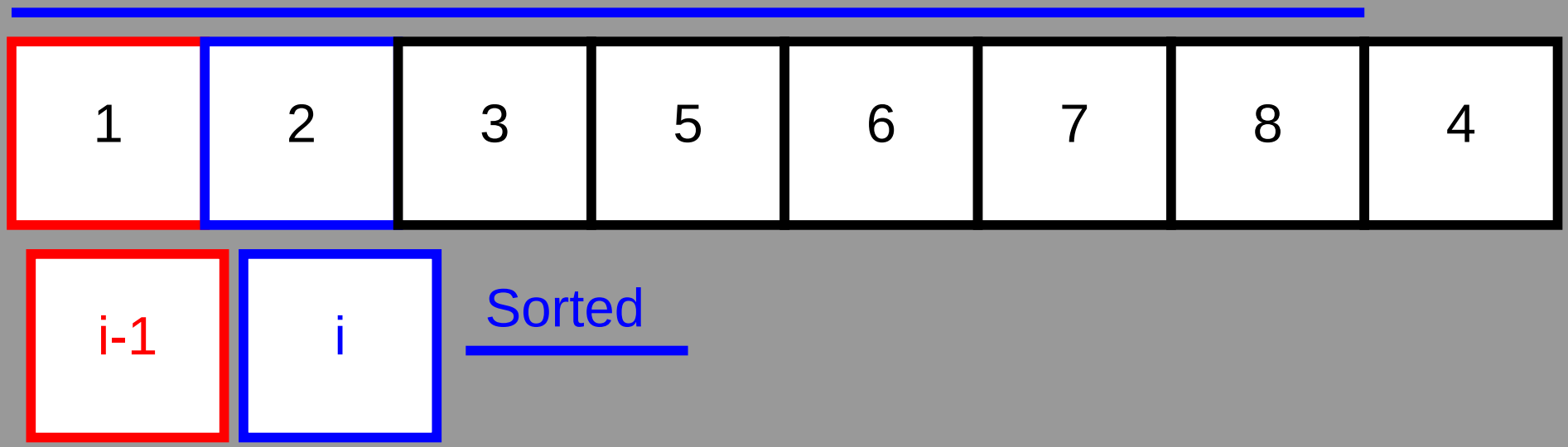
Insertion Sort Example



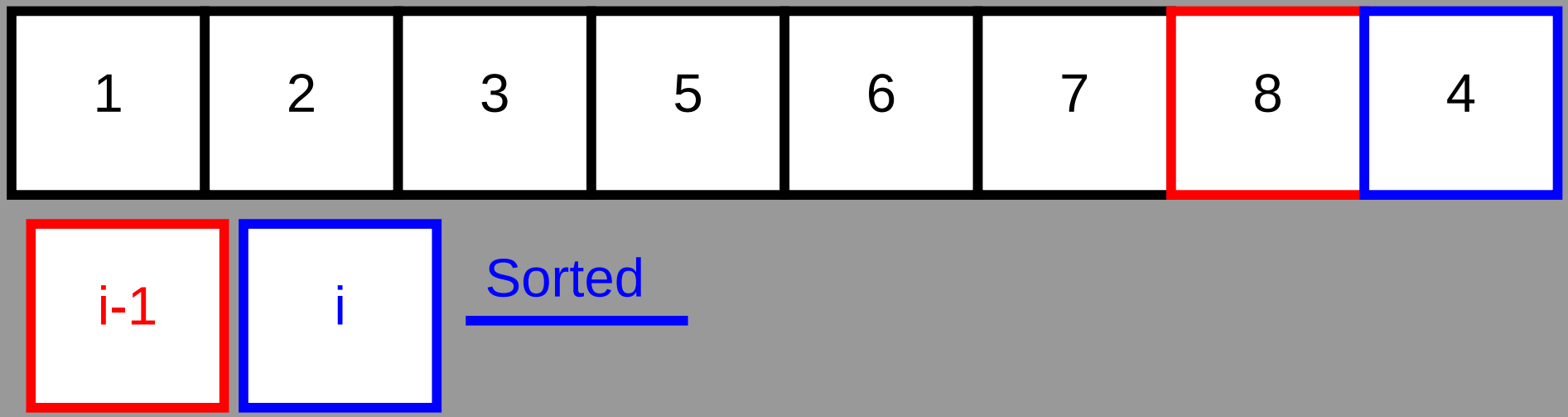
Insertion Sort Example



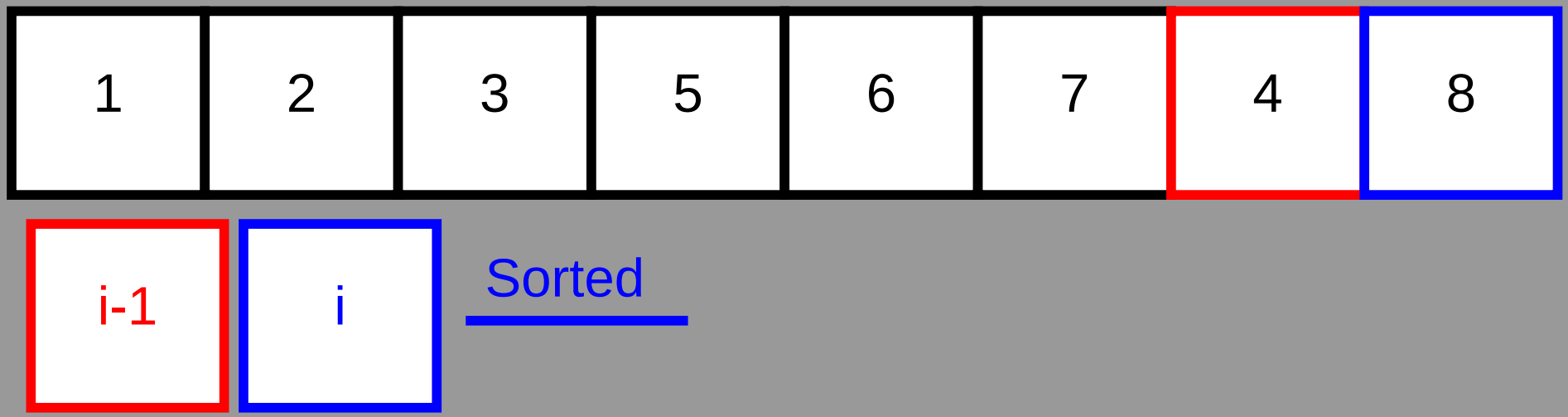
Insertion Sort Example



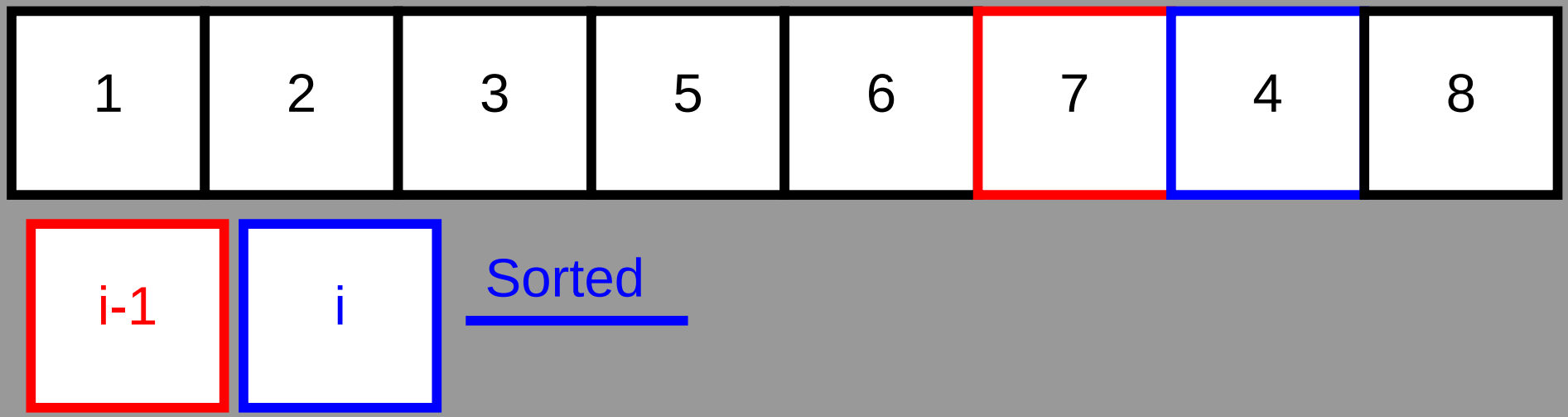
Insertion Sort Example



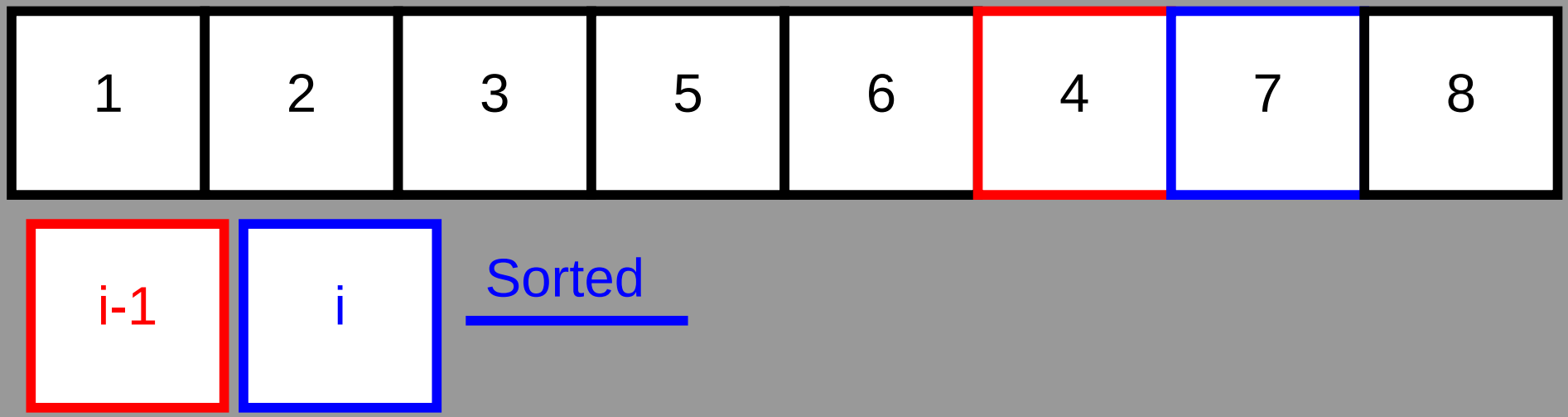
Insertion Sort Example



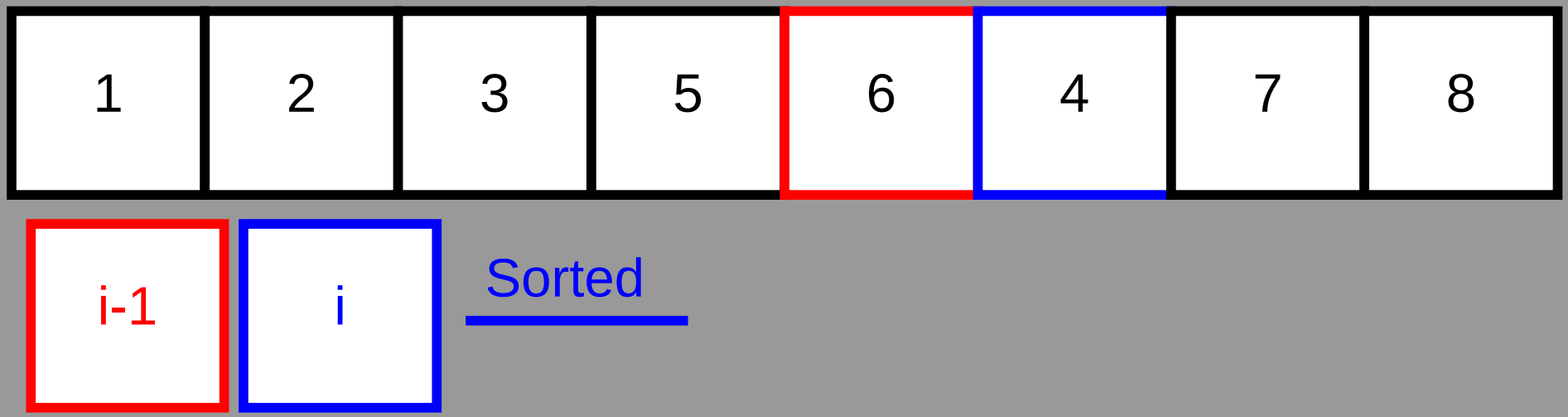
Insertion Sort Example



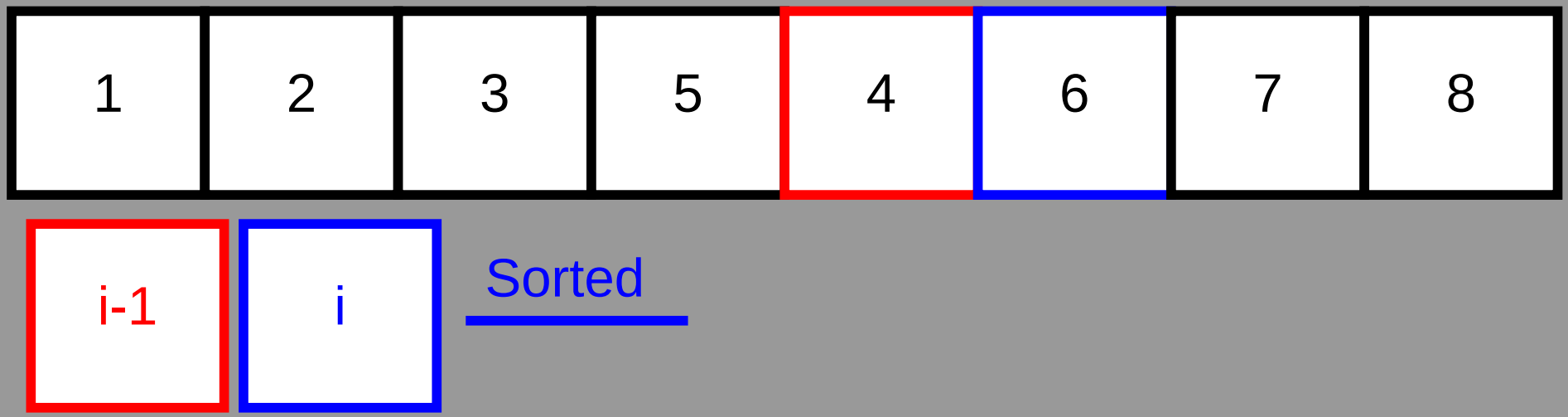
Insertion Sort Example



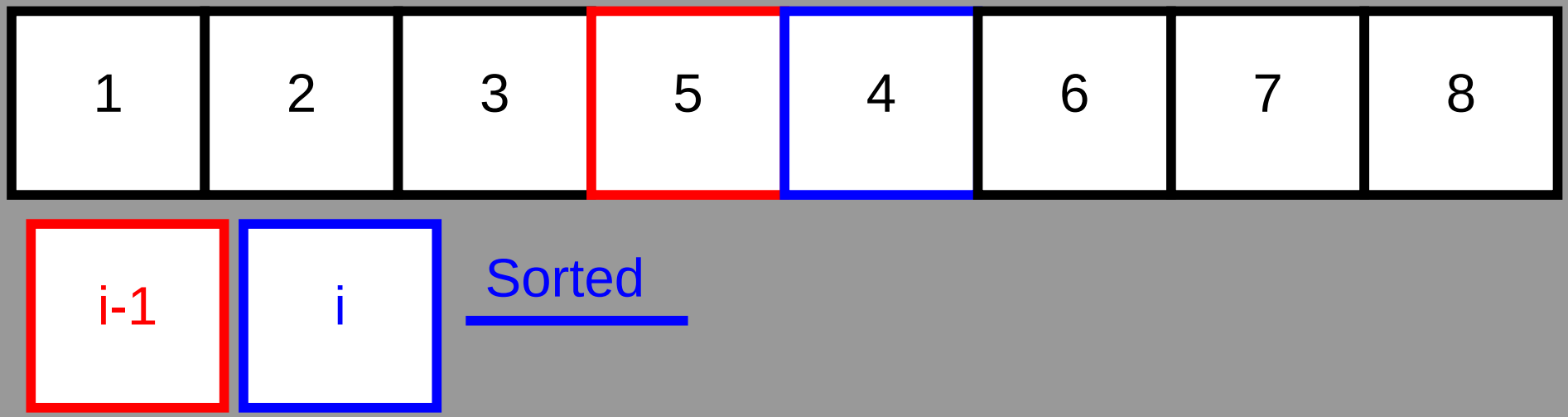
Insertion Sort Example



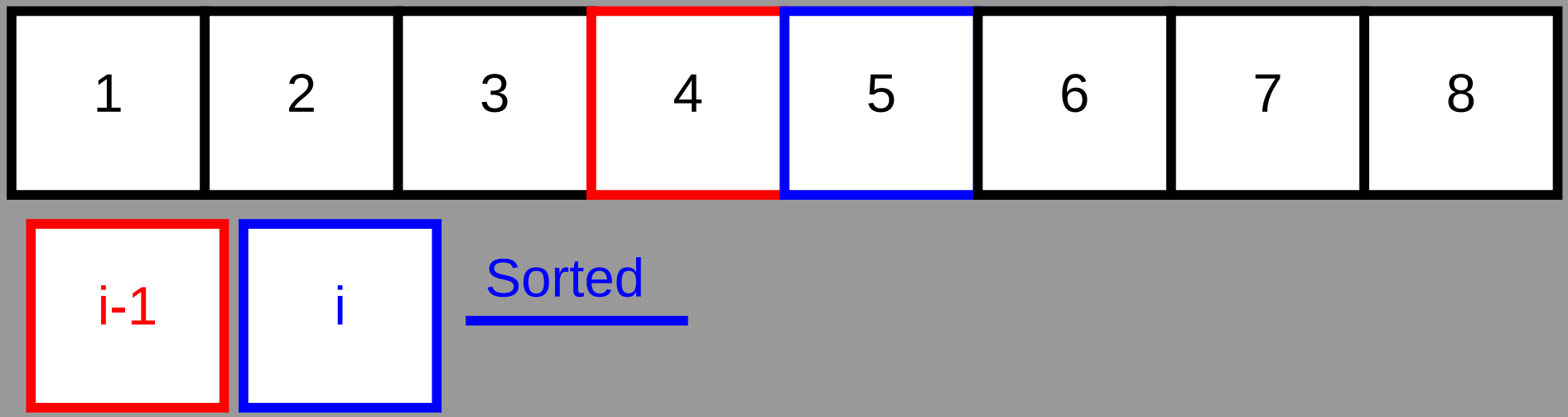
Insertion Sort Example



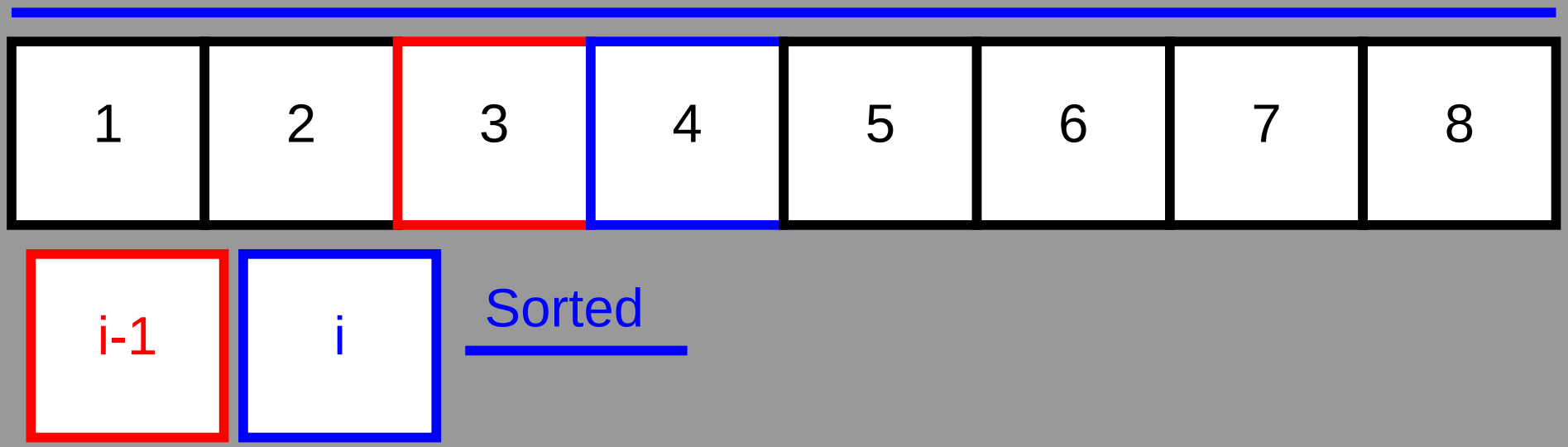
Insertion Sort Example



Insertion Sort Example



Insertion Sort Example



Complexity of Insertion Sort

- What is the worst case complexity of insertion sort?
  - $O(n^2)$
- When does this happen?
  - When the data is sorted in reverse order
- What is the best case complexity of insertion sort?
  - $O(n)$
- When does this happen?
  - When the data is already sorted

Table of Contents

Faster Sorting

All of the algorithms we have looked at so far have been $O(n^2)$, is it possible to write an algorithm that is more efficient?

- Sorting is mostly based on a single operation
  - Comparing two numbers
- If we are sorting an array of n elements, there are $n!$ possible permutations of the values
- The minimum cost of sorting is the same as the number comparisons required to choose one permutation out of $n!$

Faster Sorting

If we have to choose one permutation out of $n!$ possible permutations, how many operations do we have to perform?

- Every comparison is binary (we are comparing two numbers) and hence reduces the work by half
- This gives is the equation $2^c \geq n!$ where c is the number of comparisons

We can prove using this equation that the number of comparisons required is greater than or equal to $n * \log_2 n$

- This means that it is impossible to sort data with a complexity that is better than $O(n * \log n)$

Faster Sorting

Proof

$$2^c \geq n!$$

$$\equiv \log_2 2^c \geq \log_2 n!$$

$$\equiv c * \log_2 2 \geq \log_2(n * (n-1) * \ldots * 1$$

$$\equiv c \geq \log_2(n) + \log_2(n-1) + \ldots + \log_2(1)$$

$$c \simeq n * \log_2 n^1$$

[1] This uses Stirling's approximation

Table of Contents

Advanced Sorting

There are many algorithms that can sort data in $O(n * \log n)$ complexity

- Today we will look at
  - Quicksort
  - Mergesort
- Before we can start looking a quick sort we must first study a more basic sorting algorithm
  - The Dutch national flag algorithm

Table of Contents

The Dutch National Flag Problem

This problem was designed by a famous Dutch computer scientist named Edsger W. Dijkstra

- It is a sorting problem that can be solved in $O(n)$ complexity

The Problem

- The flag of the Netherlands has three colours: red, white and blue
- We have a line balls of these colours in random order
- Our task is to sort them so that they are grouped together and in the correct order

The Dutch National Flag Problem

Representation

In order to be able to write an algorithm to solve this problem, we need to represent it in a way that the computer can understand

- We represent that balls as an array of numbers
  - 1 represents red
  - 0 represents white
  - 2 represents blue
- Now we can sort this array such that:
  - All ones come before all zeroes
  - All zeroes come before all twos

The Dutch National Flag Problem

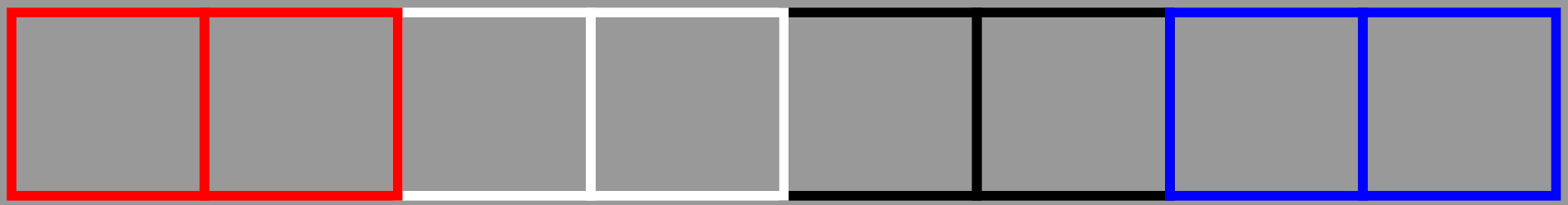
The Solution

The idea behind the solution is quite simple, we keep two partially sorted lists

1. Some numbers at the beginning of the array where the ones and zeroes are in the correct order
2. Some numbers at the end of the array that contain only twos

This leaves the middle of the array containing all the numbers we have not sorted yet

- Every step we pick a single number from the middle and move it to the correct position in one of the sorted lists

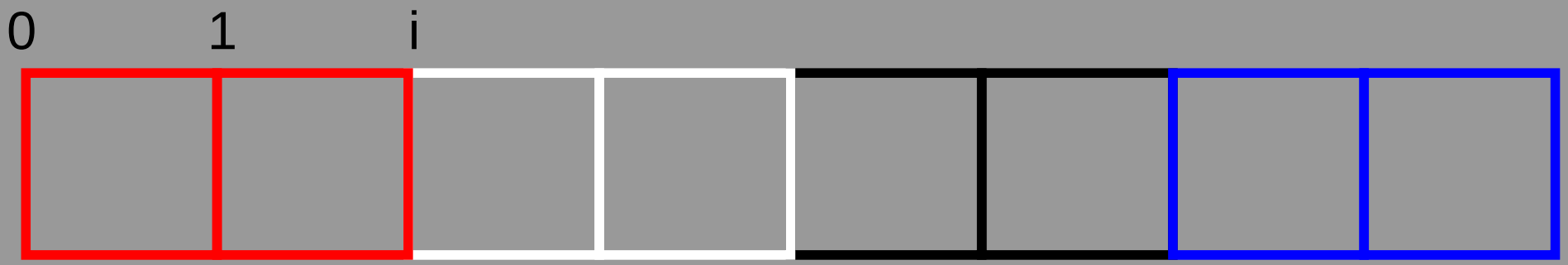


The Solution

Representing the solution

To implement this algorithm we need to represent the end of both partially sorted lists, as well as where the separation between ones and zeroes is in the first list

- Because we already know that the start of the first list is at 0, we only need to remember where the ones end
  - We will use the variable `i` to represent the position after the ones we have already sorted

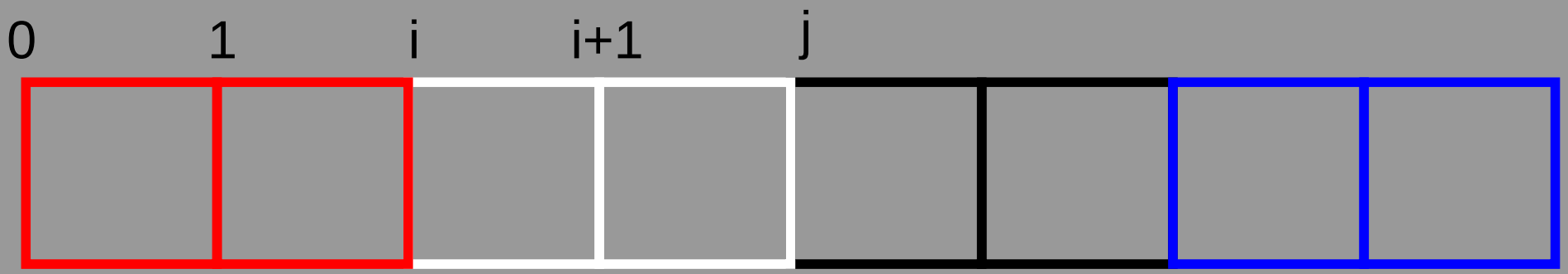


The Solution

Representing the solution

To implement this algorithm we need to represent the end of both partially sorted lists, as well as where the separation between ones and zeroes is in the first list

- Because the zeroes must come after the ones, we already know where the zeroes start (position `i`), again we only need to know where the sorted part ends
  - We will use the variable `j` to represent the position after the zeroes we have already sorted

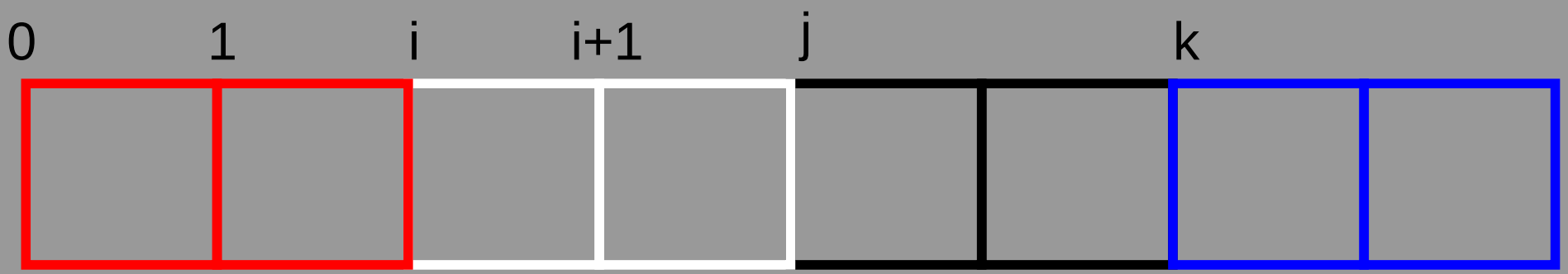


The Solution

Representing the solution

To implement this algorithm we need to represent the end of both partially sorted lists, as well as where the separation between ones and zeroes is in the first list

- The second sorted list works differently
- This is because we do not know where it starts, only where it ends
- This time we need to know where the twos we have already sorted start
  - We will use the variable `k` to represent the position of the first the two we have already sorted



The Algorithm

- In the beginning we have not sorted any numbers
- To represent this we set the values of `i` and `j` to 0 and the value of `k` to `n`
- This basically means that our partially sorted lists are empty

Algoritm

- Every step of the algorithm we look at the number in position `j`
- Based on the value we will put it into the correct place in one of the already sorted lists
- After this we will either increment `j` or decrement `k`
- At the end `j` and `k` will meet in the middle

The algorithm

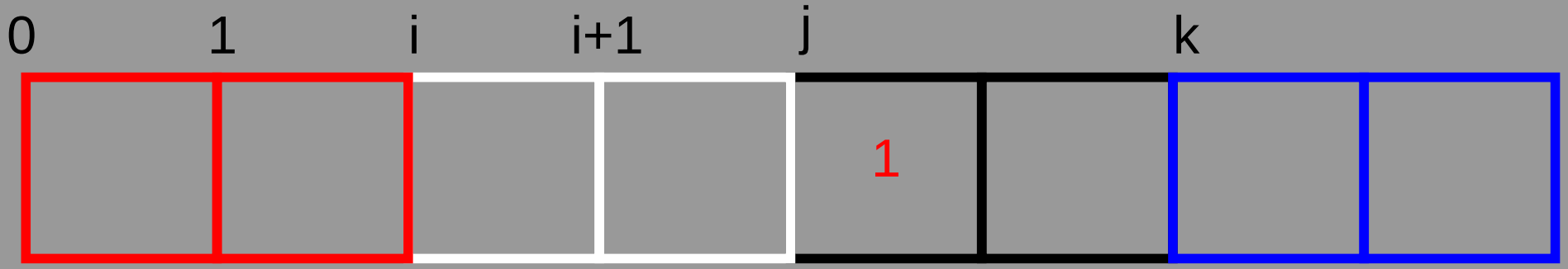
A single step

There are three possible values for the number in position `j`, so we will have three possible outcomes

Possibility 1

If the value at position `j` is one, we swap the value with the value in position `i`

- This means that there is now another one sorted in the correct place so we need to increment `i`
- The value that we moved from position `i` had to be a zero, this means that there is now a zero in position `j` so we need to increment `j`



The algorithm

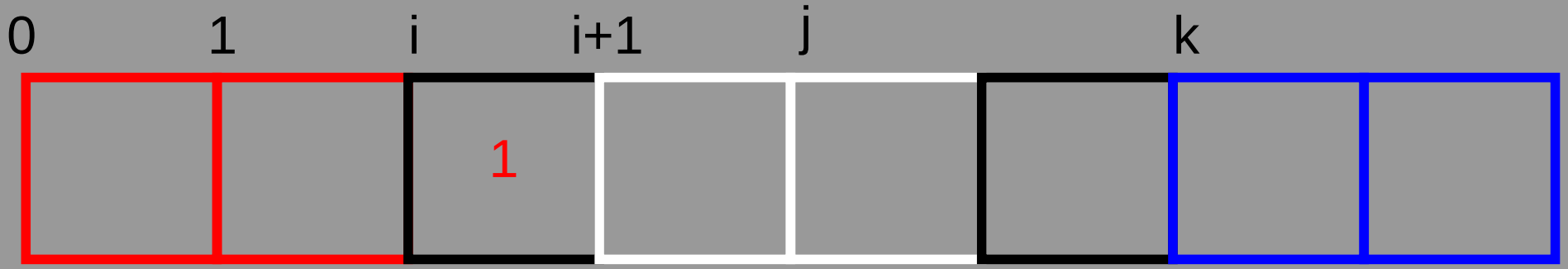
A single step

There are three possible values for the number in position `j`, so we will have three possible outcomes

Possibility 1

If the value at position `j` is one, we swap the value with the value in position `i`

- This means that there is now another one sorted in the correct place so we need to increment `i`
- The value that we moved from position `i` had to be a zero, this means that there is now a zero in position `j` so we need to increment `j`



The algorithm

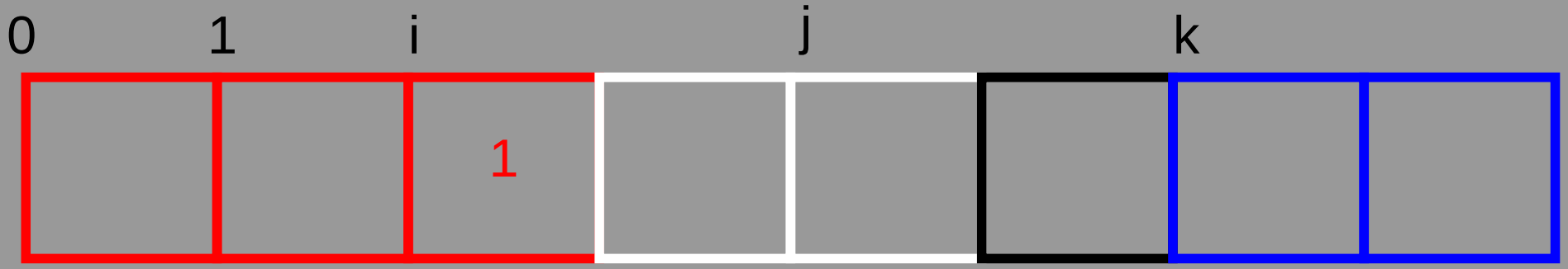
A single step

There are three possible values for the number in position `j`, so we will have three possible outcomes

Possibility 1

If the value at position `j` is one, we swap the value with the value in position `i`

- This means that there is now another one sorted in the correct place so we need to increment `i`
- The value that we moved from position `i` had to be a zero, this means that there is now a zero in position `j` so we need to increment `j`



The algorithm

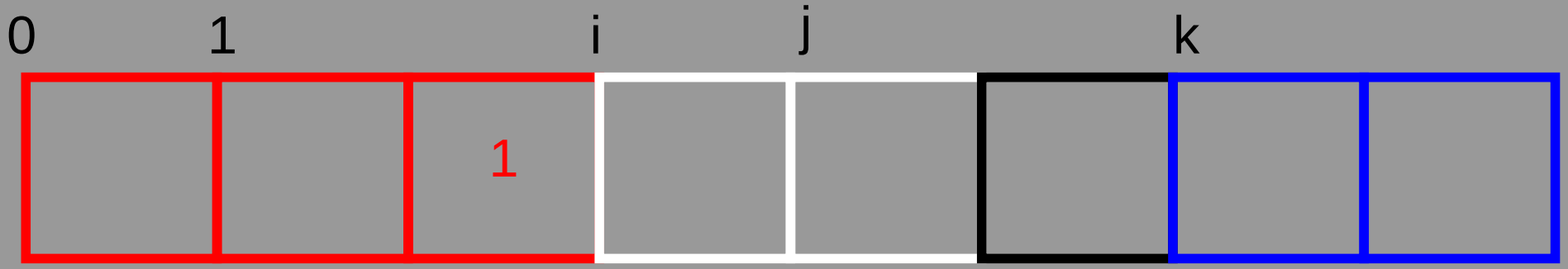
A single step

There are three possible values for the number in position `j`, so we will have three possible outcomes

Possibility 1

If the value at position `j` is one, we swap the value with the value in position `i`

- This means that there is now another one sorted in the correct place so we need to increment `i`
- The value that we moved from position `i` had to be a zero, this means that there is now a zero in position `j` so we need to increment `j`



The algorithm

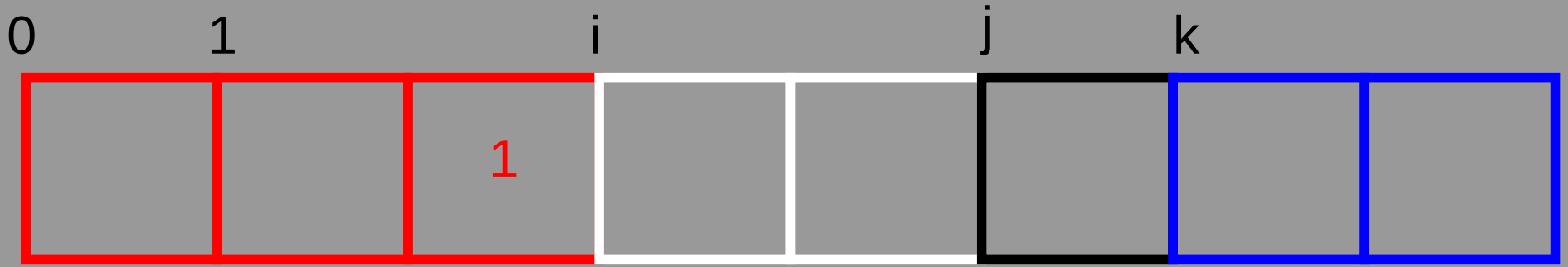
A single step

There are three possible values for the number in position `j`, so we will have three possible outcomes

Possibility 1

If the value at position `j` is one, we swap the value with the value in position `i`

- This means that there is now another one sorted in the correct place so we need to increment `i`
- The value that we moved from position `i` had to be a zero, this means that there is now a zero in position `j` so we need to increment `j`



The algorithm

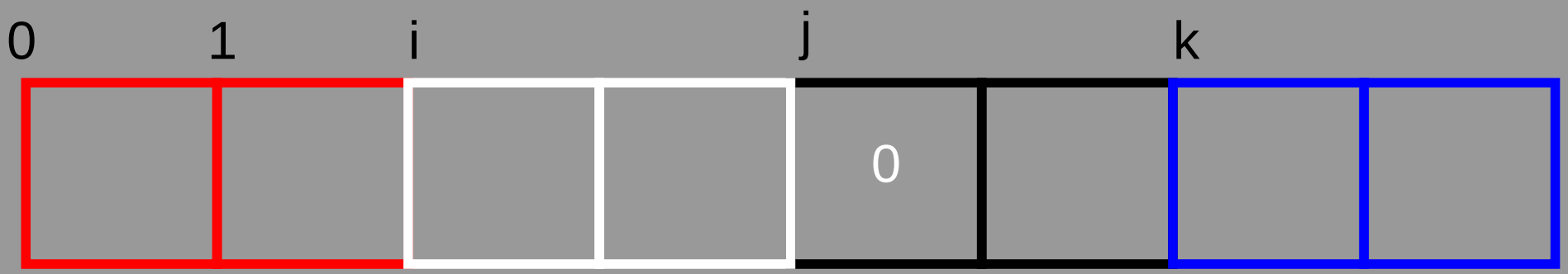
A single step

There are three possible values for the number in position `j`, so we will have three possible outcomes

Possibility 0

If the value at position `j` is zero, it is already in the correct place

- This means that there is now another zero sorted in the correct place so we need to increment `j`



The algorithm

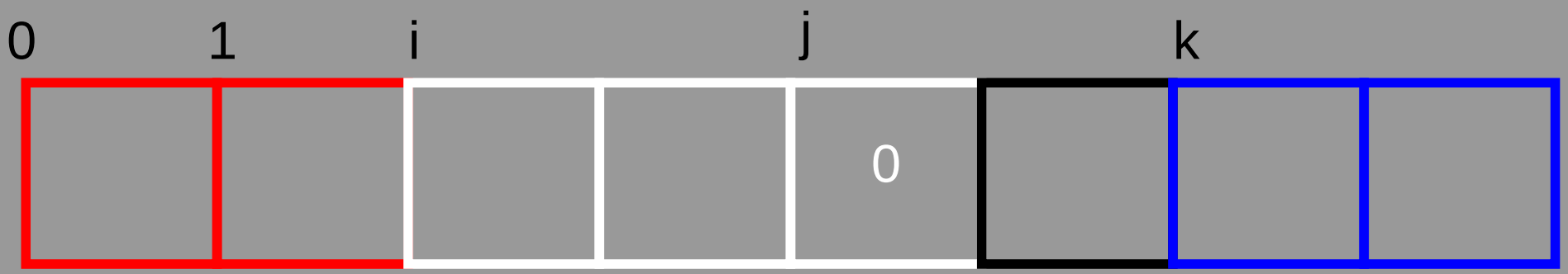
A single step

There are three possible values for the number in position `j`, so we will have three possible outcomes

Possibility 0

If the value at position `j` is zero, it is already in the correct place

- This means that there is now another zero sorted in the correct place so we need to increment `j`



The algorithm

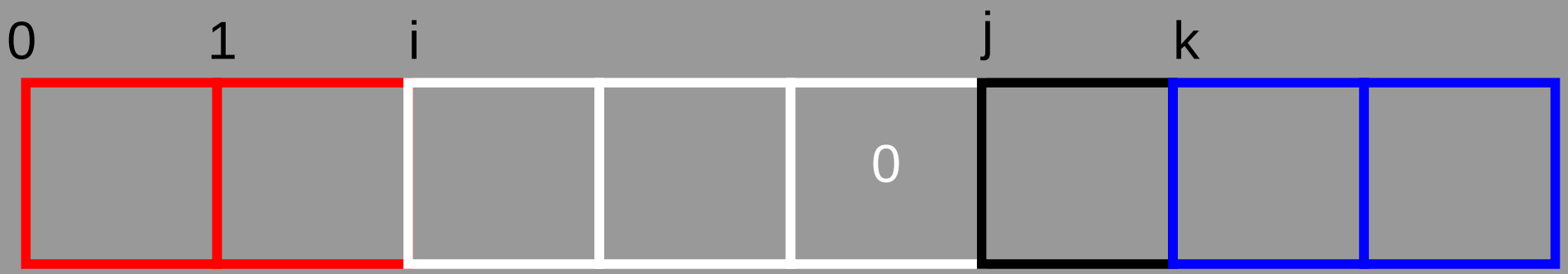
A single step

There are three possible values for the number in position `j`, so we will have three possible outcomes

Possibility 0

If the value at position `j` is zero, it is already in the correct place

- This means that there is now another zero sorted in the correct place so we need to increment `j`



The algorithm

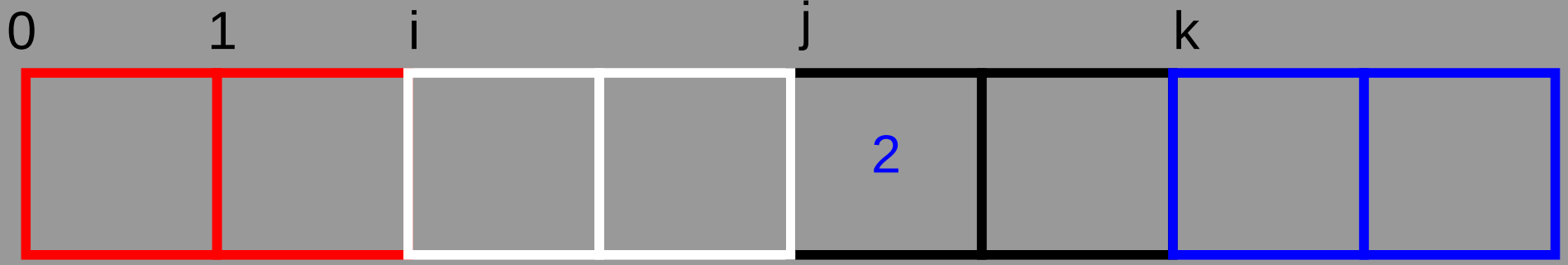
A single step

There are three possible values for the number in position `j`, so we will have three possible outcomes

Possibility 2

If the value at position `j` is two, we swap it with the number in position `(k - 1)` (before the start of the partially sorted twos)

- This means that there is now another two sorted in the correct place so we need to decrement `k`



The algorithm

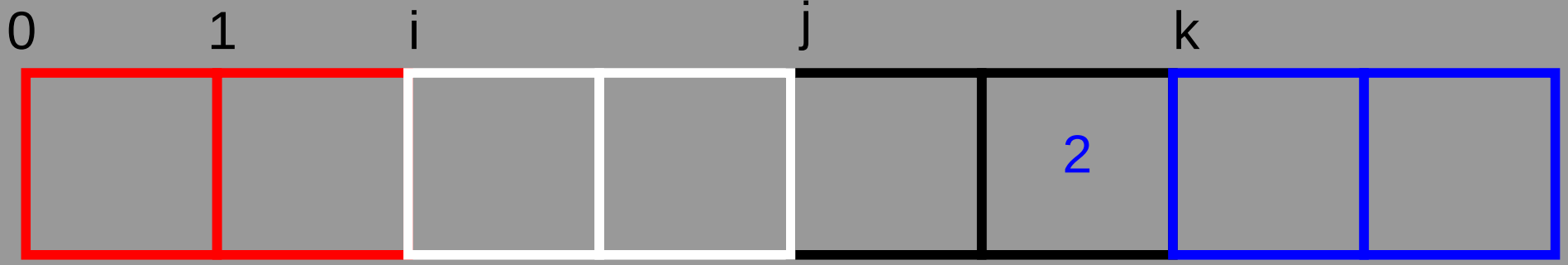
A single step

There are three possible values for the number in position `j`, so we will have three possible outcomes

Possibility 2

If the value at position `j` is two, we swap it with the number in position `(k - 1)` (before the start of the partially sorted twos)

- This means that there is now another two sorted in the correct place so we need to decrement `k`



The algorithm

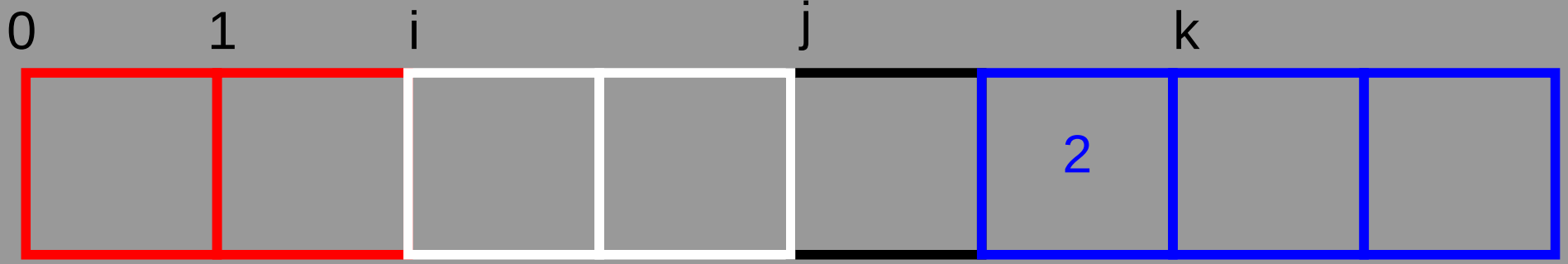
A single step

There are three possible values for the number in position `j`, so we will have three possible outcomes

Possibility 2

If the value at position `j` is two, we swap it with the number in position `(k - 1)` (before the start of the partially sorted twos)

- This means that there is now another two sorted in the correct place so we need to decrement `k`



The algorithm

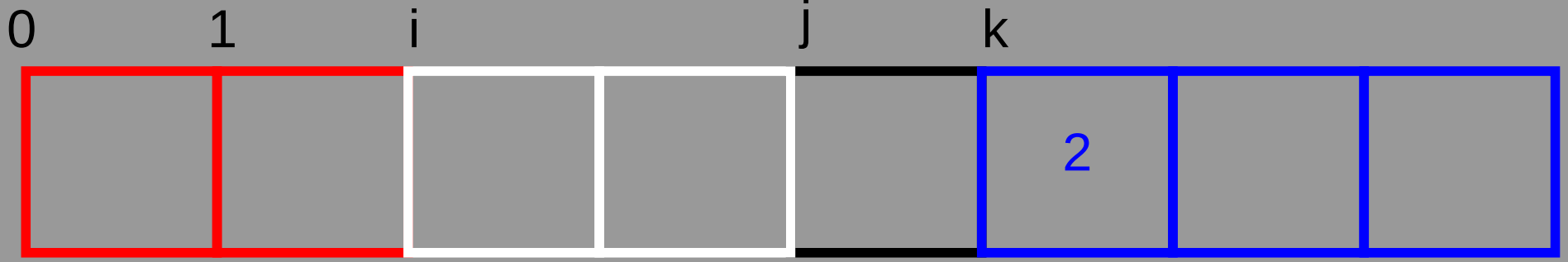
A single step

There are three possible values for the number in position `j`, so we will have three possible outcomes

Possibility 2

If the value at position `j` is two, we swap it with the number in position `(k - 1)` (before the start of the partially sorted twos)

- This means that there is now another two sorted in the correct place so we need to decrement `k`



```
1  void flagSort(int f[], int n){
2    int i = 0;
3    int j = 0;
4    int k = n;
5    while (j!=k) {
6      if(f[j] == 1){
7        swap(f, i, j);
8        i++;
9        j++;
10     } else if(f[j] == 0){
11       j++;
12     } else if (f[j] == 2) {
13       swap(f, j, k-1);
14       k--;
15     }
16   }
17 }
```

Table of Contents

Advanced Sorting

Quicksort

The idea behind quicksort and other $O(n * \log n)$ sorting algorithms is based on idea that sorting $\frac{n}{x}$ elements x times is more efficient that sorting n elements.

- Lets say we have 100 elements
- If we use an $O(n^2)$ algorithm to sort them the cost will be 10000 (100^2)
- If we separate them into two partitions of 50 and sort these the cost will be 5000=$(2 * 50^2)$

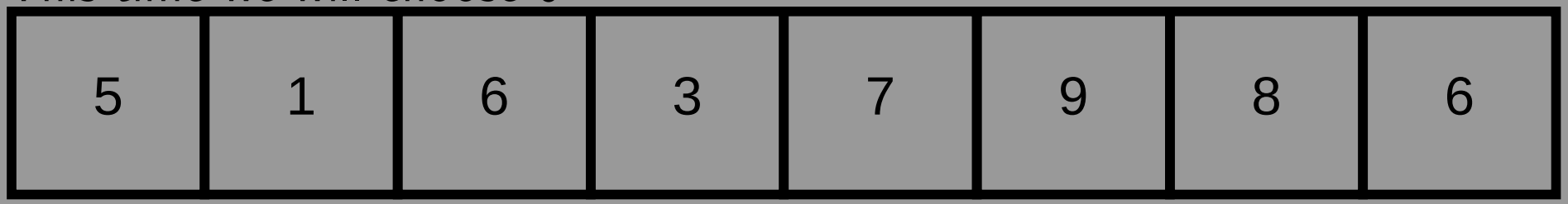
Many advances sorting techniques are based on this idea

Quicksort

- Quicksort uses the idea, that we can sort an array into three partitions in $O(n)$ time (Dutch national flag problem)
- However instead of only sorting zeros, ones and twos, we sort numbers that are less than, equal to and greater than a chosen number
- This chosen number is called a pivot value
  - Every value that is less than the pivot value goes to the left
  - Every value that is equal to the pivot comes after
  - Every value that is greater than the pivot value goes to the end
- It is important to note that the less than and greater than partitions are not finished sorting!!!

Quicksort Example

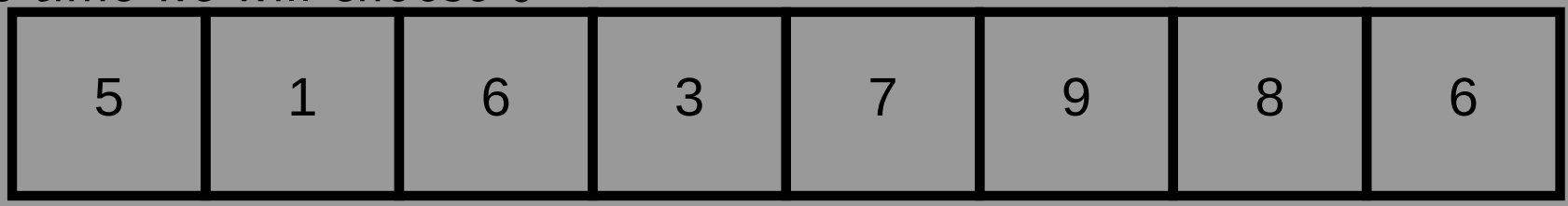
First we must choose a pivot number. This will be explained later.
This time we will choose 6

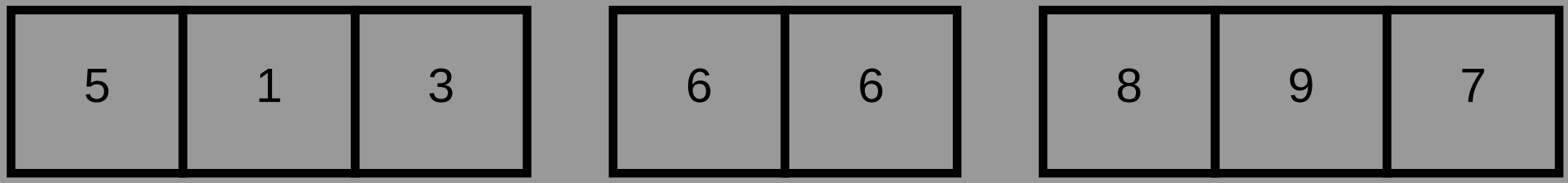


Now we have three partitions, one that is already sorted and two that still need to be sorted. The next step is to use the algorithm again on each of the unsorted partitions

Quicksort Example

First we must choose a pivot number. This will be explained later.
This time we will choose 6

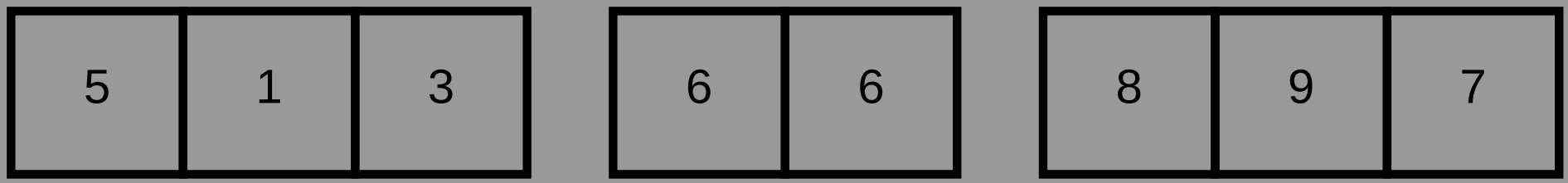




Now we have three partitions, one that is already sorted and two that still need to be sorted. The next step is to use the algorithm again on each of the unsorted partitions

Quicksort Example

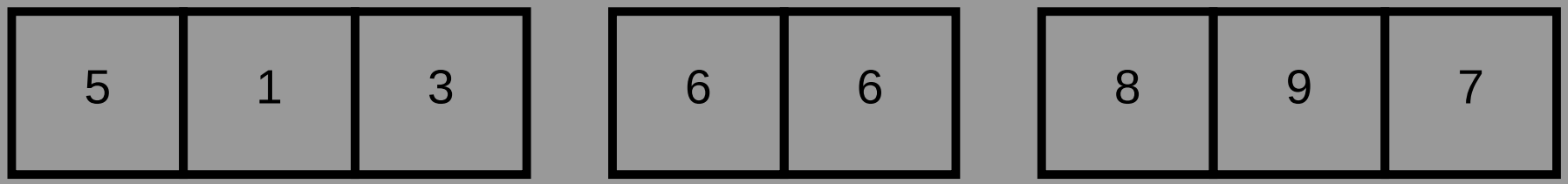
- We have to sort both unsorted partitions
- We will sort the left one first
- Again we have to choose a pivot point, this time we choose 3

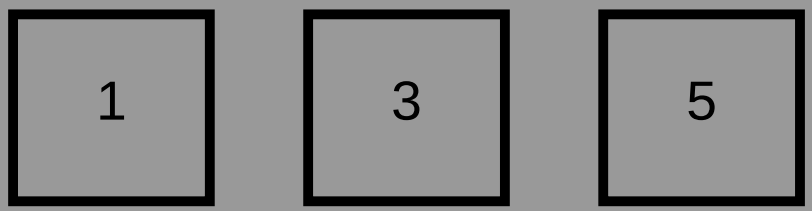


- All we have done is move each of the numbers to the left or right of 3 and they are now sorted

Quicksort Example

- We have to sort both unsorted partitions
- We will sort the left one first
- Again we have to choose a pivot point, this time we choose 3

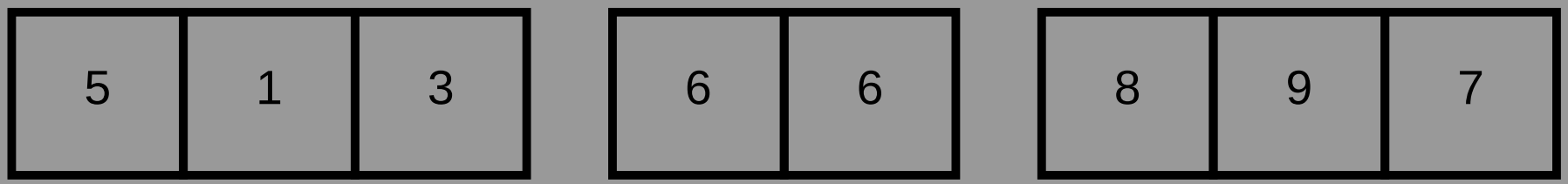


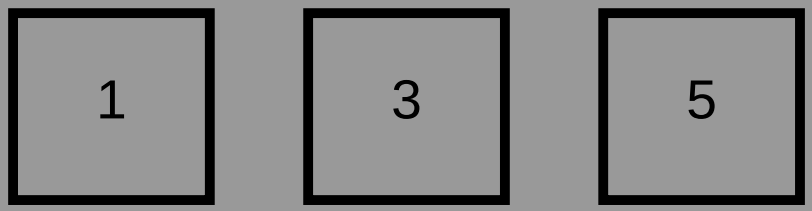


- All we have done is move each of the numbers to the left or right of 3 and they are now sorted

Quicksort Example

- There is only one unsorted partition remaining
- Again we have to choose a pivot point, this time we choose 8

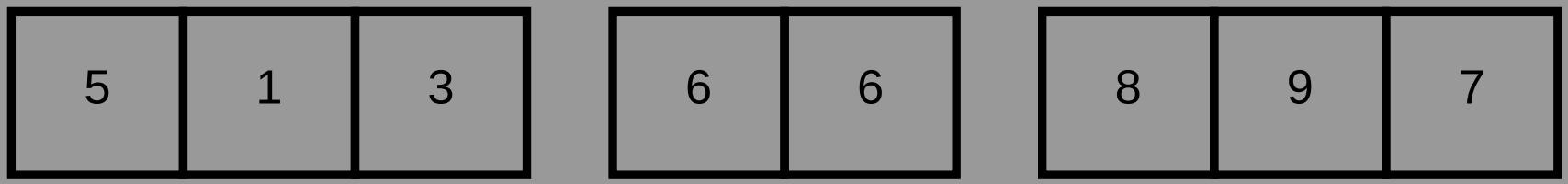




- All we have done is move each of the numbers to the left or right of 8 and they are now sorted
- All of the partitions of the array are now sorted and we are finished

Quicksort Example

- There is only one unsorted partition remaining
- Again we have to choose a pivot point, this time we choose 8



- All we have done is move each of the numbers to the left or right of 8 and they are now sorted
- All of the partitions of the array are now sorted and we are finished

Quicksort Code

```
1  void quickSort(int f[], int lo, int hi){
2    if(hi-lo > 1) {
3      int i = lo;
4      int j = lo;
5      int k = hi;
6      // re-arrange f[lo..hi-1] into three
7      // partitions f[lo..i-1] and f[k..hi-1]
8      //(lo < i < k < hi) and each element in
9      // f[lo..i-1] < each element in f[k..hi-1]
10     quickSort(f,lo,i);
11     quickSort(f,k,hi);
12   }
13 }
```

Pivot Value

- The number selected for the pivot value is very important
- If we choose very well the array will be split very evenly (This is more efficient)
- If we choose poorly we can split the array into two partitions of size 1 and n-1
  - If we keep making this mistake the algorithm is $O(n^2)$
- The best performance would be if we already knew which value would split the partitions evenly
- Finding this out would take more time than we save
- A solution is to guess and pick a value that is in the middle of the partition

Complexity of Quicksort

- The complexity of the quicksort changes with the pivot value chosen
- The more evenly the partitions, the closer to $O(n * \log n)$
- The more uneven the partitions, the closer to $O(n^2)$

Table of Contents

Merging Sorted lists

- We have two arrays that are already sorted
- We want to put all of the values into a single sorted array?
- What is the complexity of this operation?
  - This can be completed in $O(n)$ time

Merging Sorted Lists

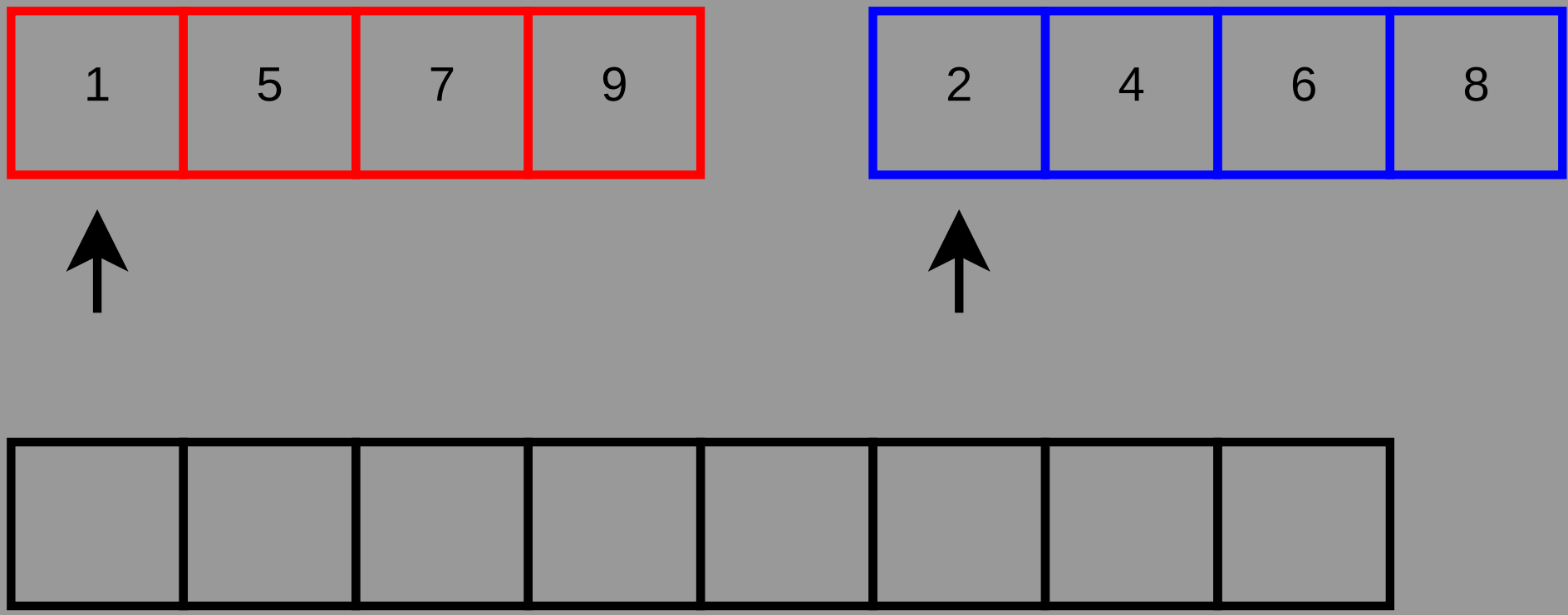
The process of combining two lists is called merging

Efficient Merging

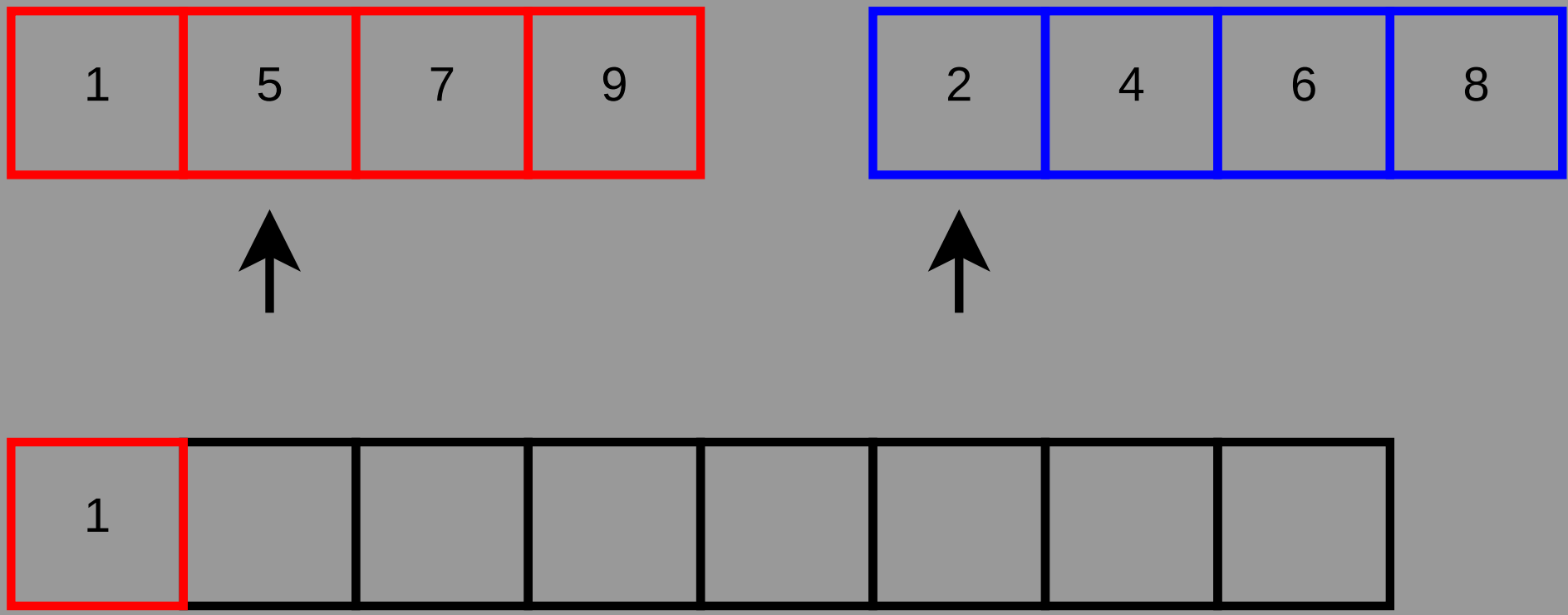
The idea behind merging sorted lists is very simple

- The smallest value in each array is in the first position
- The smallest value in our merged array will be one of these
- We choose the smallest value and place that in the array
- Then we repeat the process but ignore the number we have already copied
- When we are finished both arrays are copied into a new array and already sorted

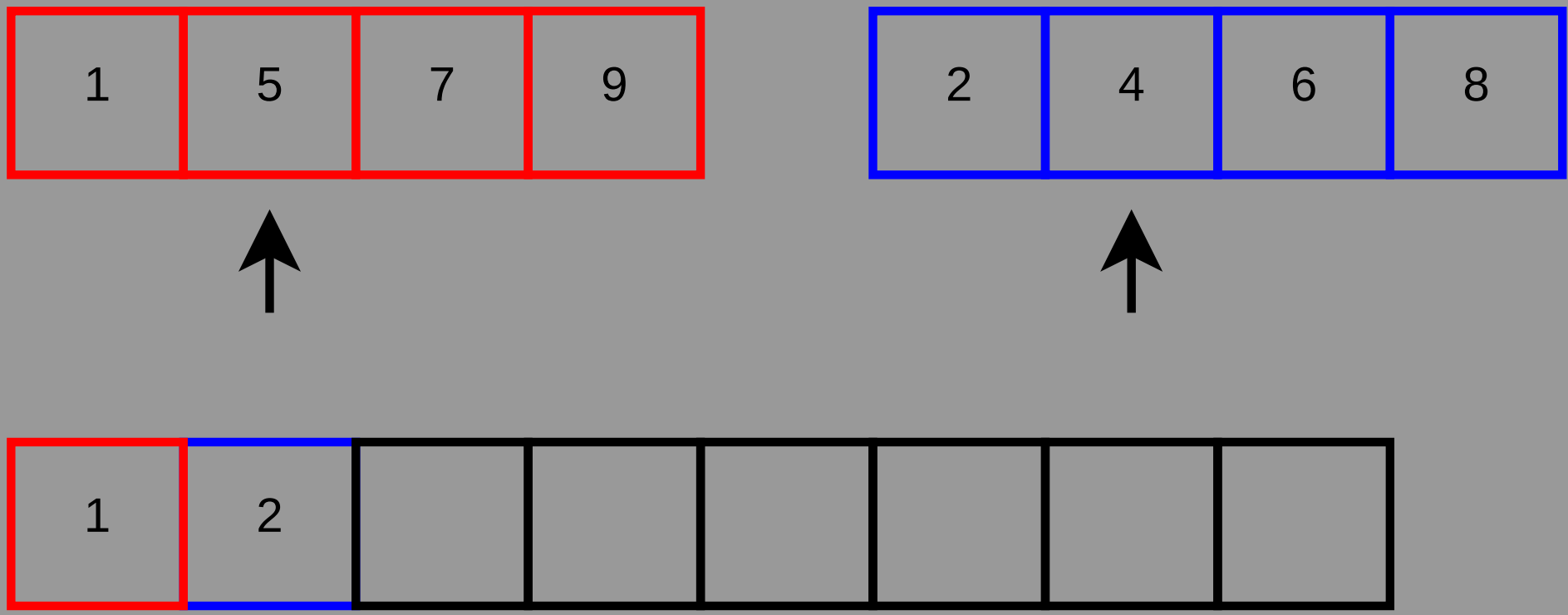
Merging Example



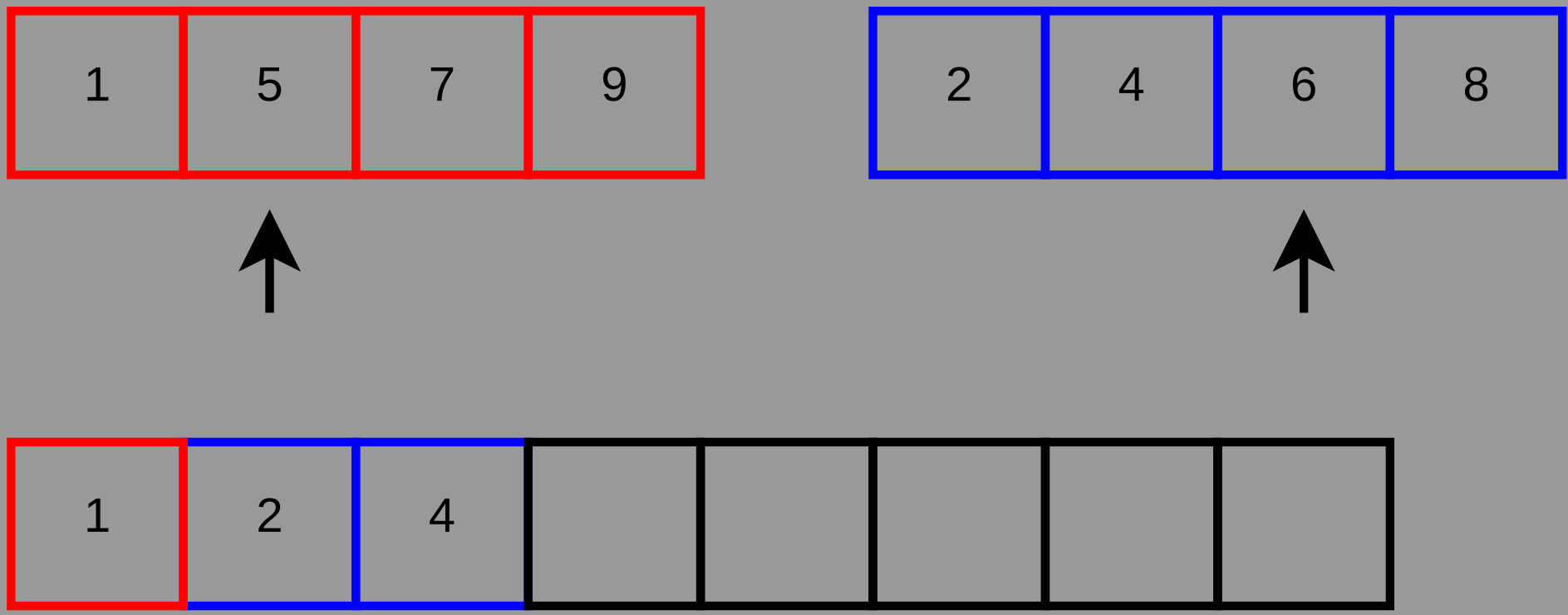
Merging Example



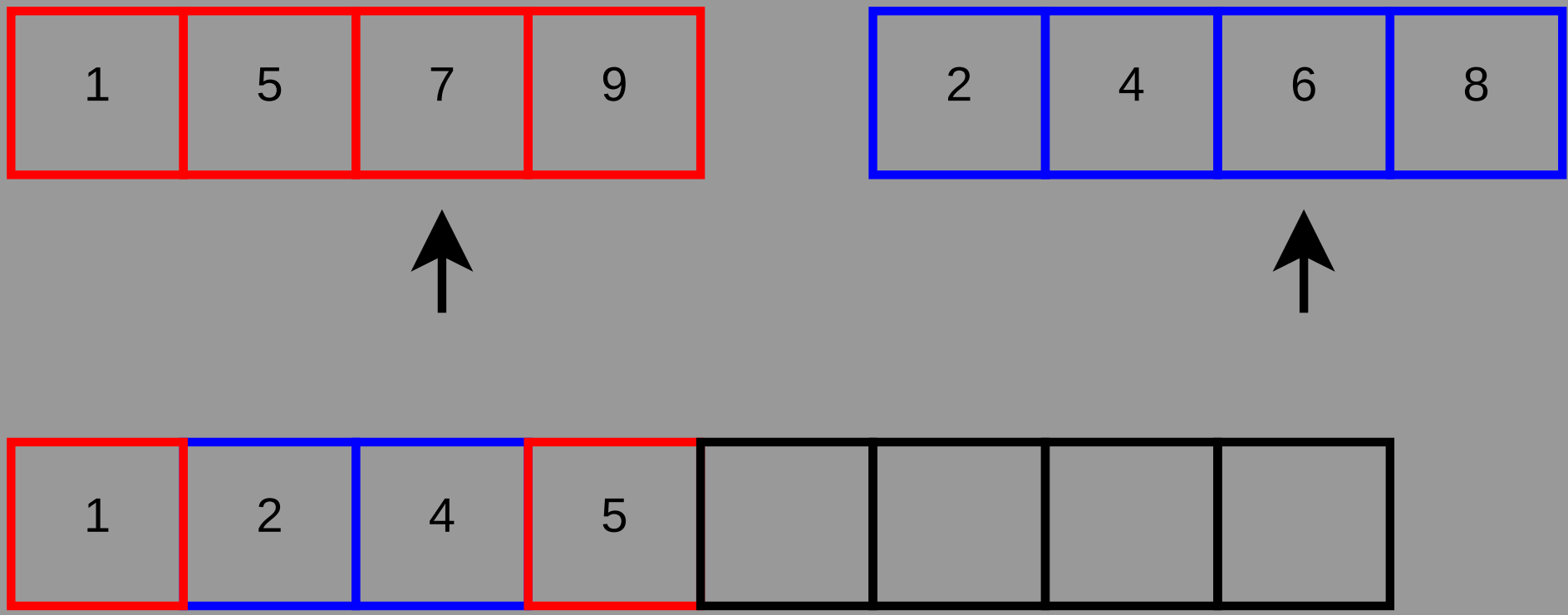
Merging Example



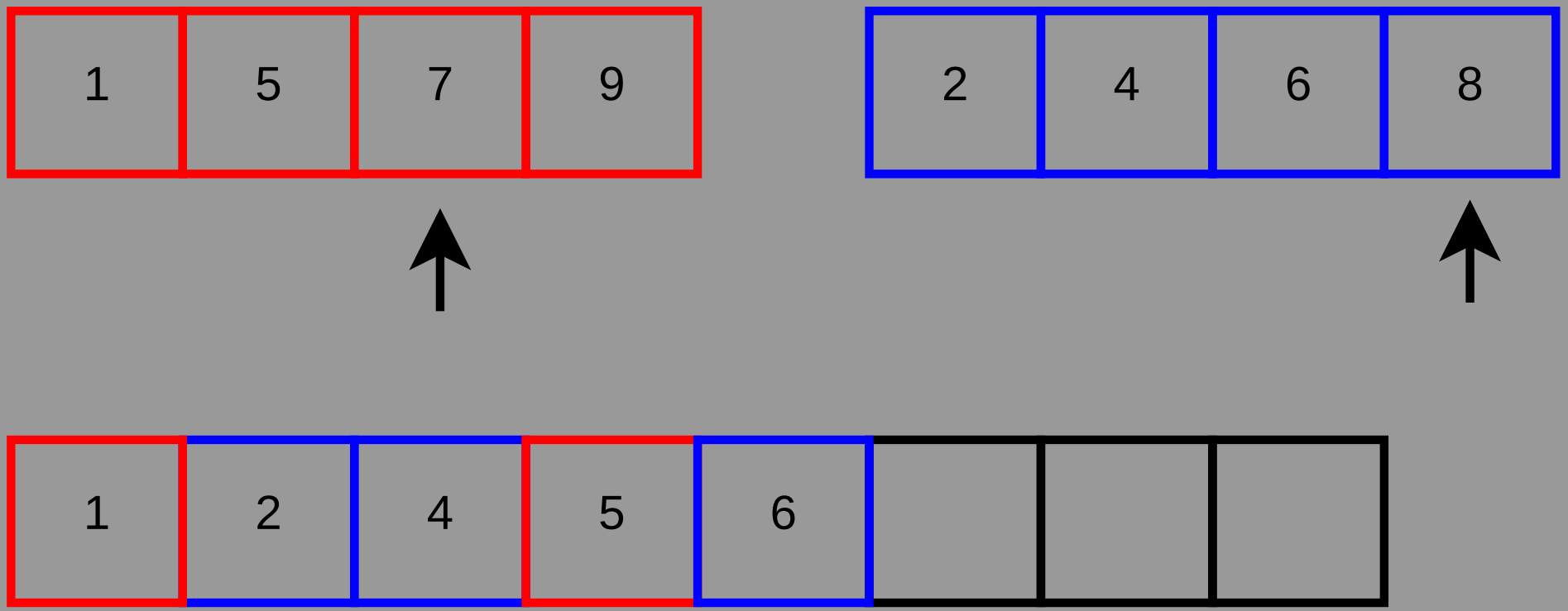
Merging Example



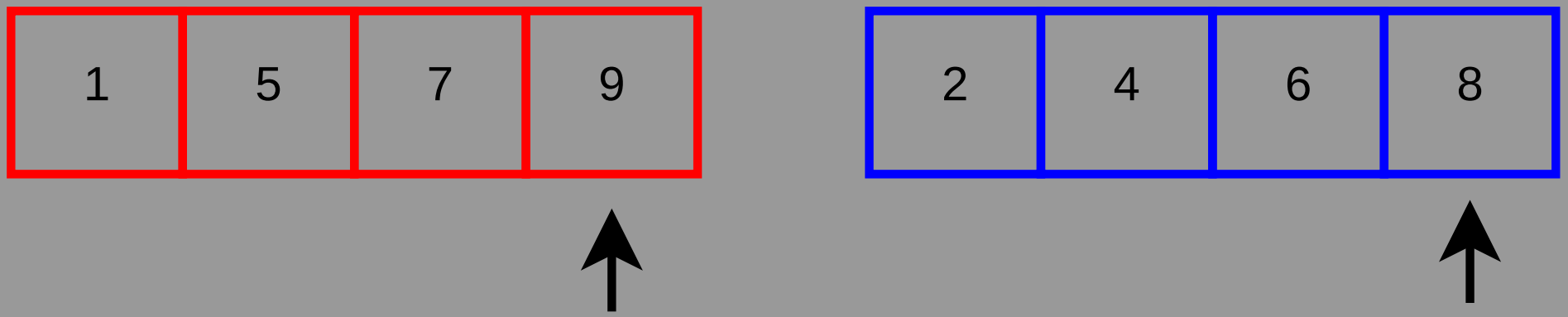
Merging Example

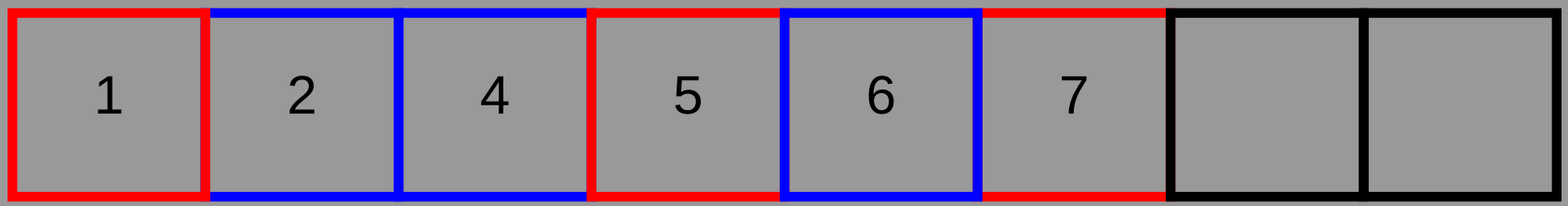


Merging Example

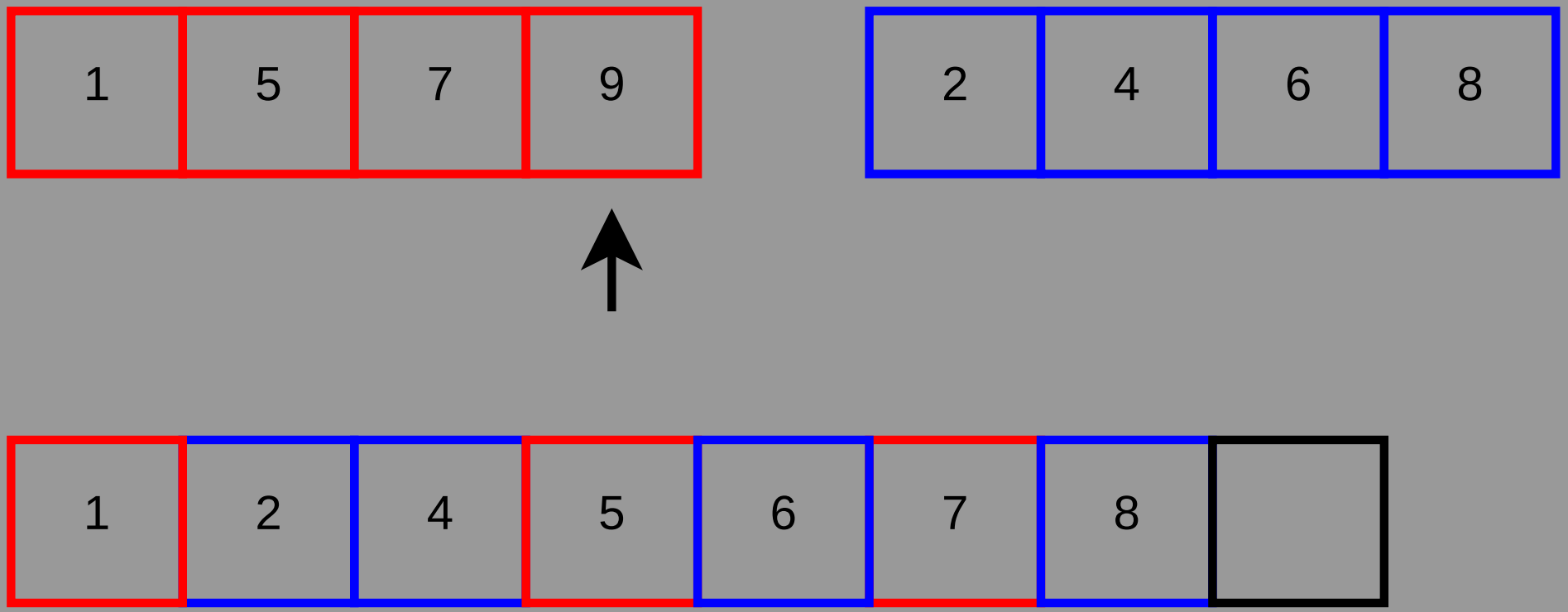


Merging Example





Merging Example



Merging Example

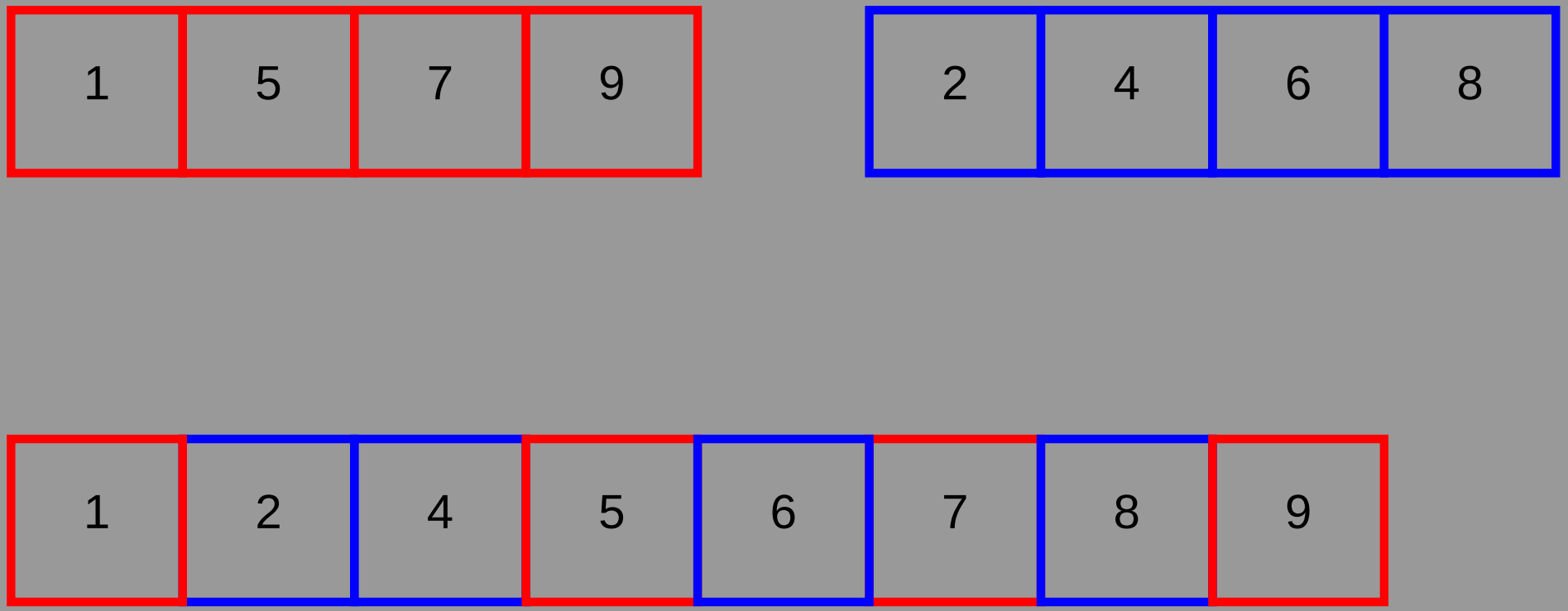


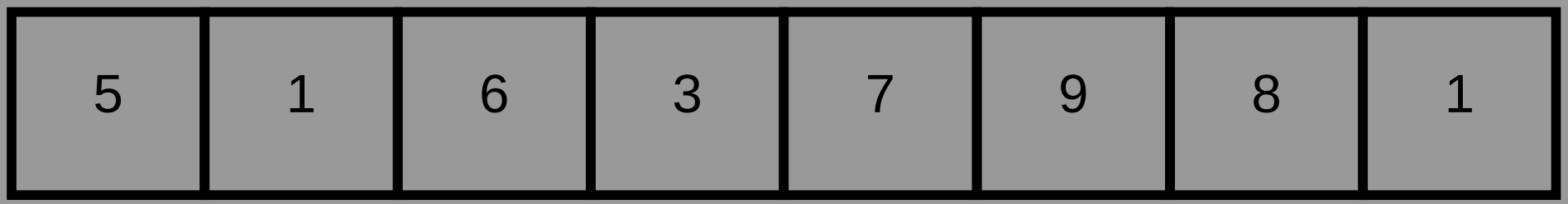
Table of Contents

Mergesort

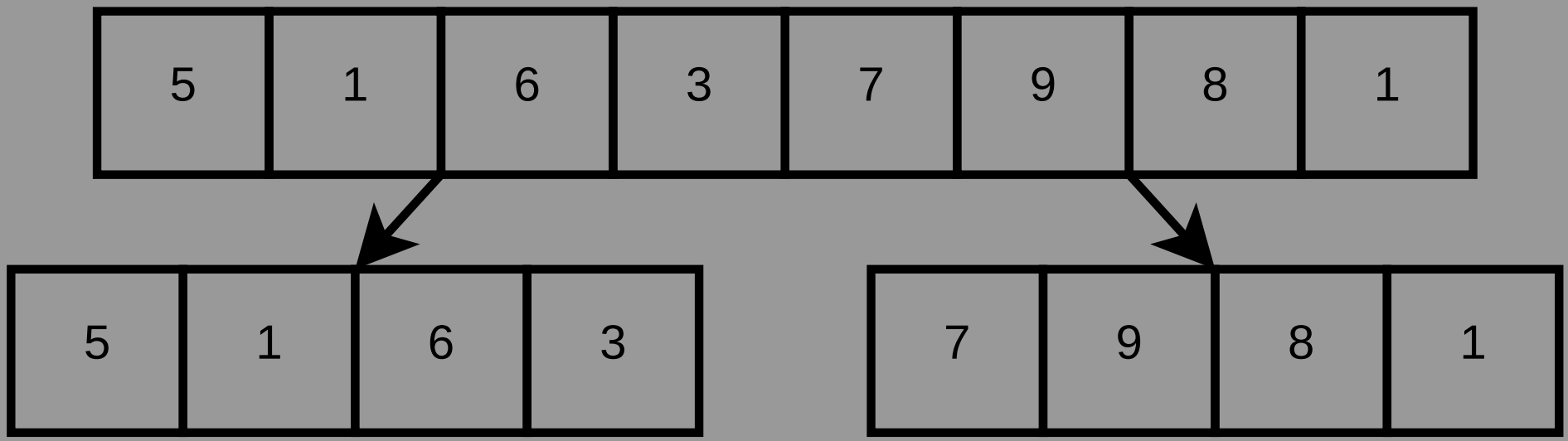
Mergesort takes advantage of the efficiency of merging already sorted lists

- Mergesort divides the array in half repeatedly until there is only one element in each partitions
- These partitions are then merged again and again until the list is sorted

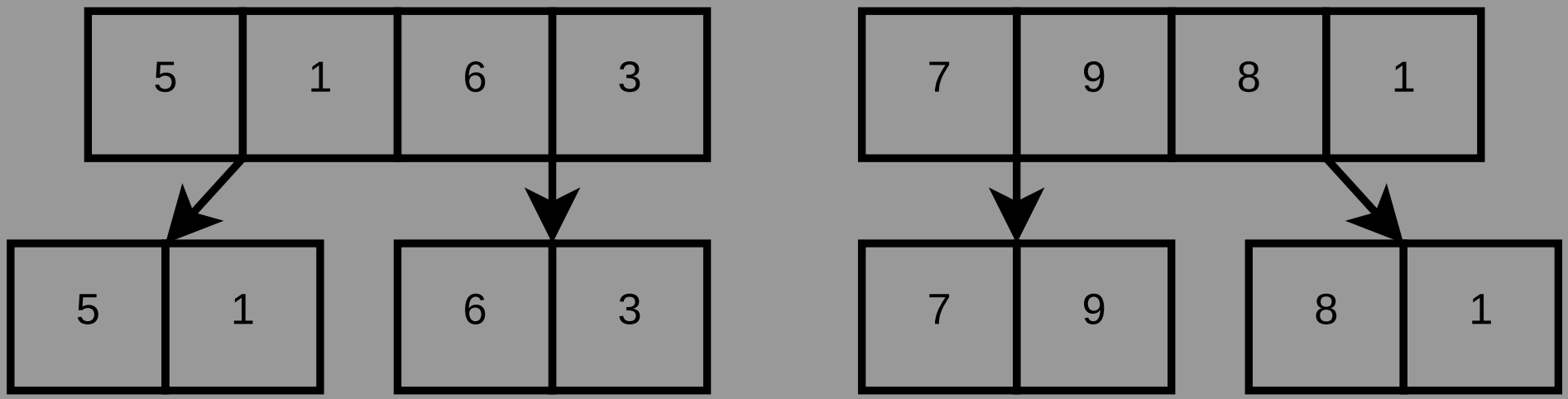
Mergesort Example



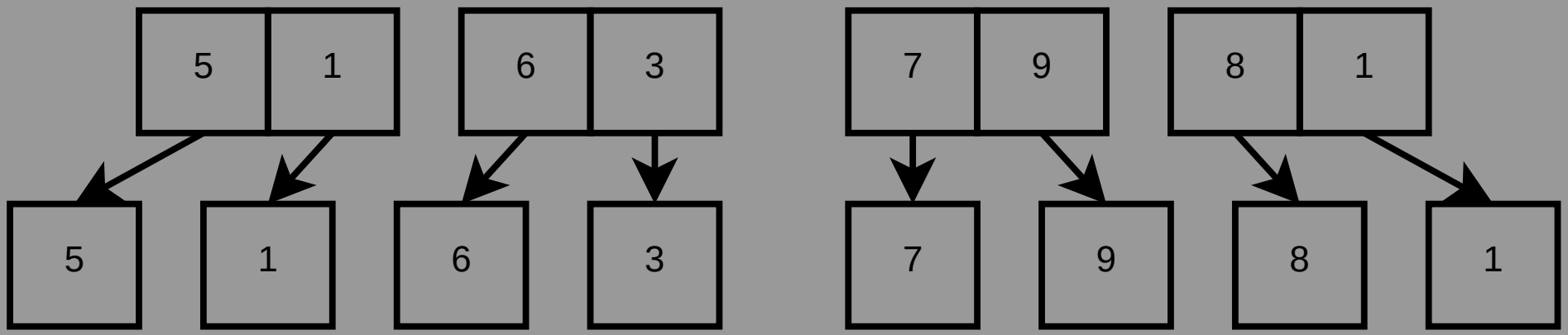
Mergesort Example



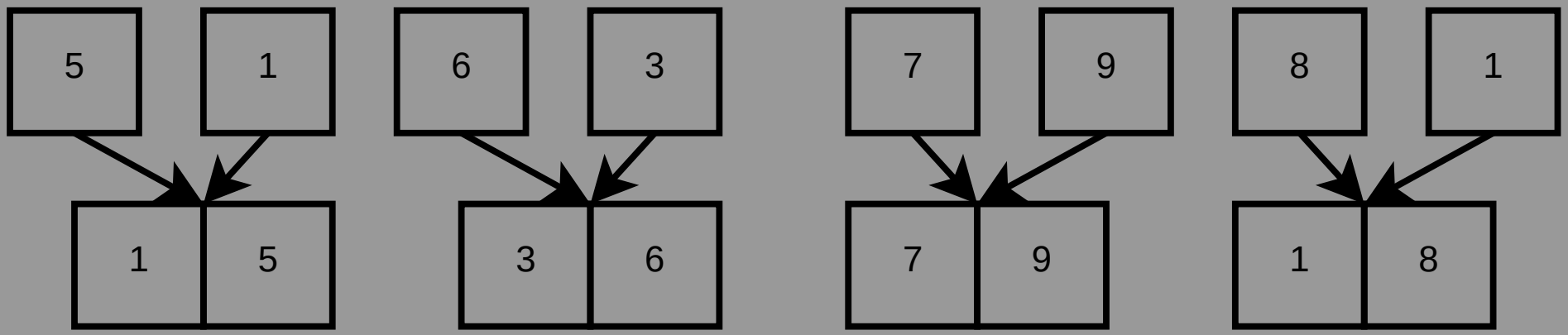
Mergesort Example



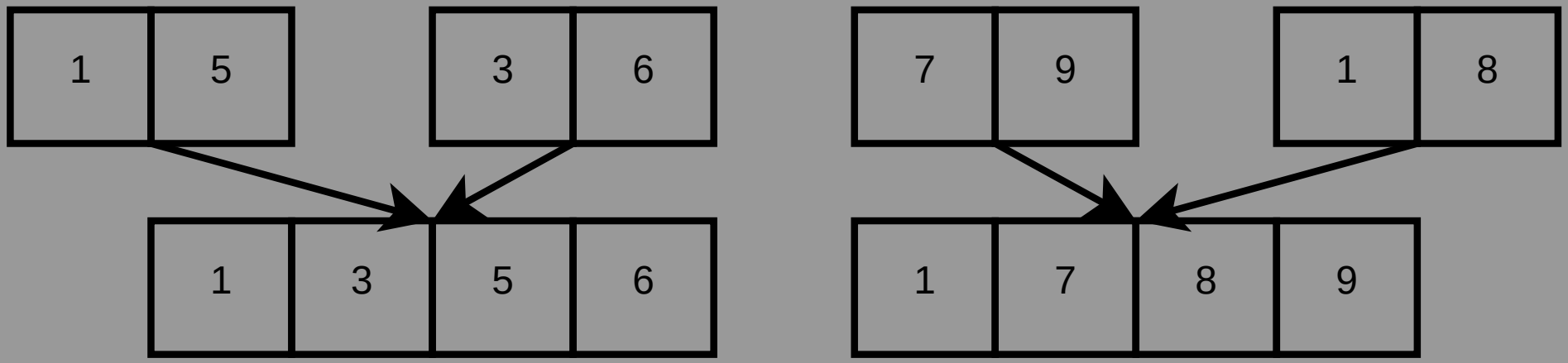
Mergesort Example



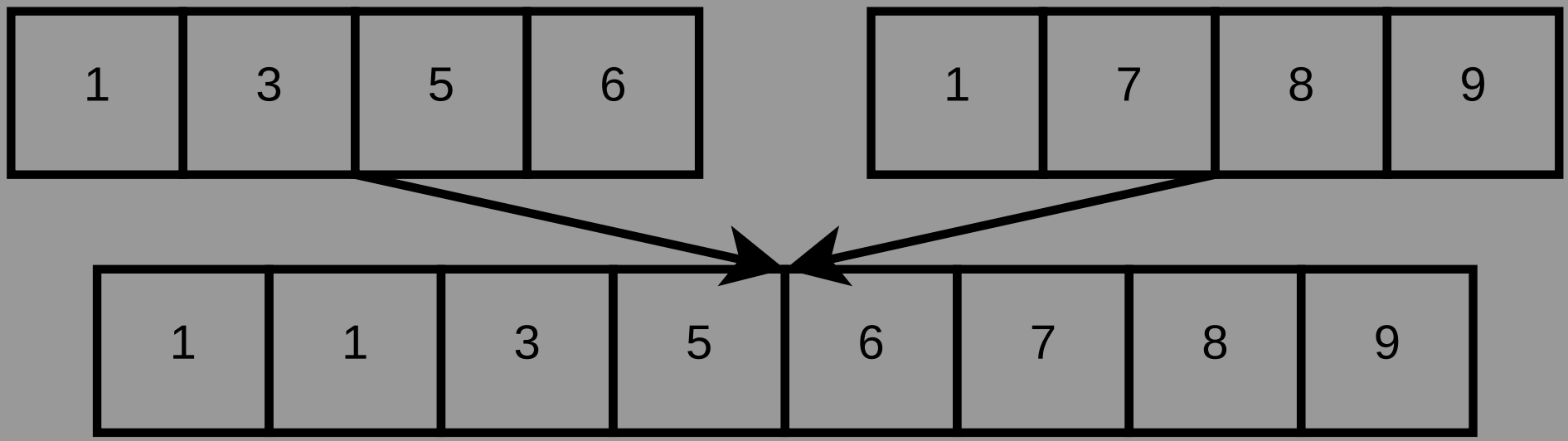
Mergesort Example



Mergesort Example



Mergesort Example



Mergesort Code

```
1 void mergeSort(int f[], int lb, int ub){
2   if(lb+1 < ub){
3     int mid = (lb+ub)/2;
4     mergeSort(f,lb,mid);
5     mergeSort(f,mid,ub);
6     merge(f,lb,mid,ub);
7   }
8 }
```

Complexity of Mergesort

- The complexity of mergesort is based on the operations that we have to perform
  - Splitting the arrays
  - Merging the arrays
- Because we are halving the size of the array every step, splitting the array happens $\log n$ times
- For every time that we split an array in two we also have to join two together, joining has a complexity of $O(n)$
- This means that the total cost of mergesort is $O(n * \log n)$

Table of Contents

Recursion

- Recursion is the practice of defining a function/method in terms of itself
- In mathematics recursion is defined by two properties:
  - A base case
  - A set of rules that we follow to get to the base case

Recursive Definition of Factorial

$$n! = \begin{cases} 1 & \text{if} \quad n = 0 \\ n * (n-1)! & \text{if} \quad n > 0 \end{cases}$$

- Here the base case is when n is equal to zero the answer is 1
- For all other values the factorial of n is defined in terms of the factorial of $n - 1$

Recursive Factorial Example

Recursive Definition of Factorial

$$n! = \begin{cases} 1 & \text{if} \quad n = 0 \\ n * (n-1)! & \text{if} \quad n > 0 \end{cases}$$

We will use this definition to calculate the value of 5!

$$n! = n * (n-1)!$$

$$5! = 5 * (5-1)! = 5 * 4!$$

$$5! = 5 * 4 * (4-1)! = 5 * 4 * 3!$$

$$5! = 5 * 4 * 3 * (3-1)! = 5 * 4 * 3 * 2!$$

$$5! = 5 * 4 * 3 * 2 * (2-1)! = 5 * 4 * 3 * 2 * 1!$$

$$5! = 5 * 4 * 3 * 2 * 1 * (1-1)! = 5 * 4 * 3 * 2 * 1 * 0!$$

$$5! = 5 * 4 * 3 * 2 * 1 * 1 = 120$$

Programming with recursion

- Using recursion allows us to program repetitive behaviours without using loops
- Instead of using a loop, we call the same function again, with slightly different parameters
- Each time we call the function it must make some progress to the base case

Recursive Definition of Factorial

$$mergesort(f, lo, hi) = \begin{cases} & \text{if} \quad hi - lo <= 1 \\ mergesort(f, lo, (lo + hi/2)) & \\ mergesort(f, (lo + hi/2), hi) & \text{if} \quad hi - lo > 1 \end{cases}$$

Mergesort Recursion Example

Recursive Definition of Factorial

$$mergesort(f, lo, hi) = \begin{cases} & \text{if} \quad hi - lo <= 1 \\ mergesort(f, lo, (lo + hi/2)) & \\ mergesort(f, (lo + hi/2), hi) & \text{if} \quad hi - lo > 1 \end{cases}$$

- If we have an array named a that is size 8 and wish to use mergesort on it, we call $mergesort(a, 0, 8)$
- As the difference between hi and lo is > 1 we call it again twice $mergesort(a, 0, 4) + mergesort(a, 4, 8)$
- The first call does the first half of the array and the second does the second half
- For both of these we again do the same and continue until we reach the base case

Table of Contents

Non-Comparison based Sorting

- So far all of the sorting algorithms we have looked at are based on making comparisons between two numbers to sort them
- Now we will look at some algorithms that do not use comparisons to sort data
  - Bucket Sort
  - Radix Sort

Linked Lists

Reminder

Last year you studied linked lists in Programming two. We will be looking at them in more detail later in this course. Here is a reminder of how they work

- A linked list is a data structure consisting of a group of nodes which together represent a sequence
- Linked lists are based on the idea of object references/pointers
- Linked lists grow and shrink when data is added or removed

Table of Contents

Bucket Sort

- Set up an array of m buckets where each bucket is responsible for one possible key value
  - Each bucket is a linked list
- Place each item in the correct bucket
- Concatenate all of the lists from the buckets to get the final sorted order
  - Start with the bucket that stores the lowest key and add the others in order

Evaluation function

Each key is mapped to the correct bucket using an evaluation function

- This takes a key as a parameter and returns the number of the bucket to place it into

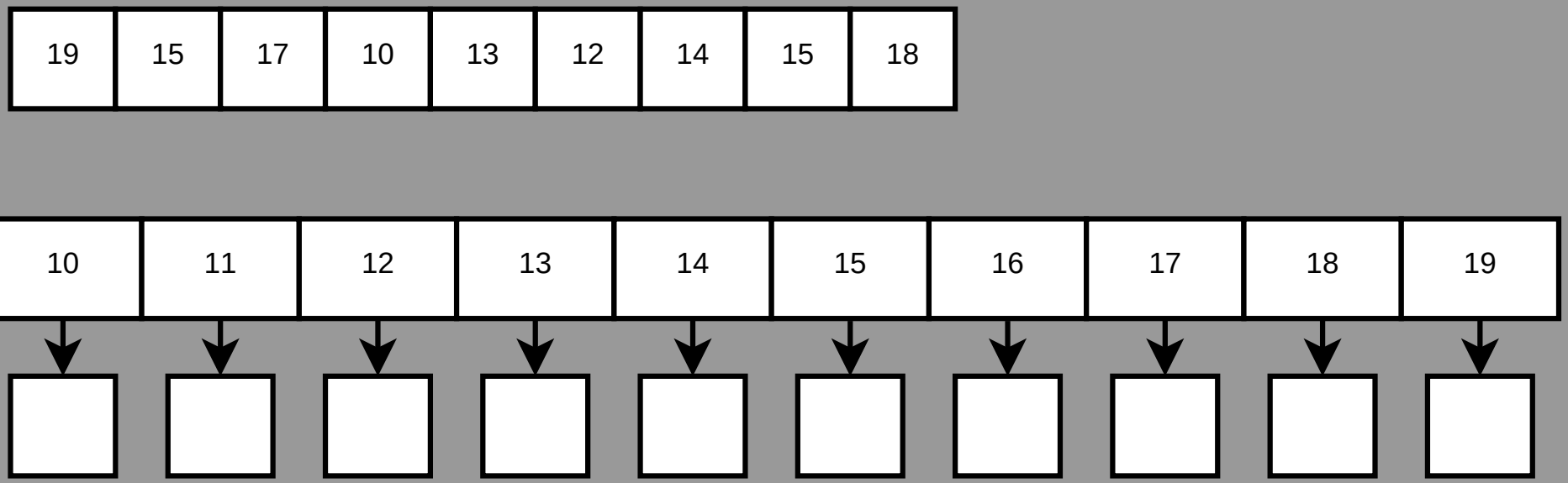
Bucket Sort

Evaluation Function

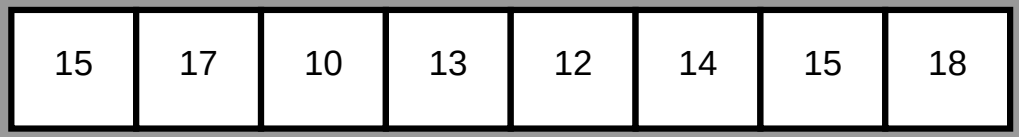
Here are some examples of possible evaluation functions.

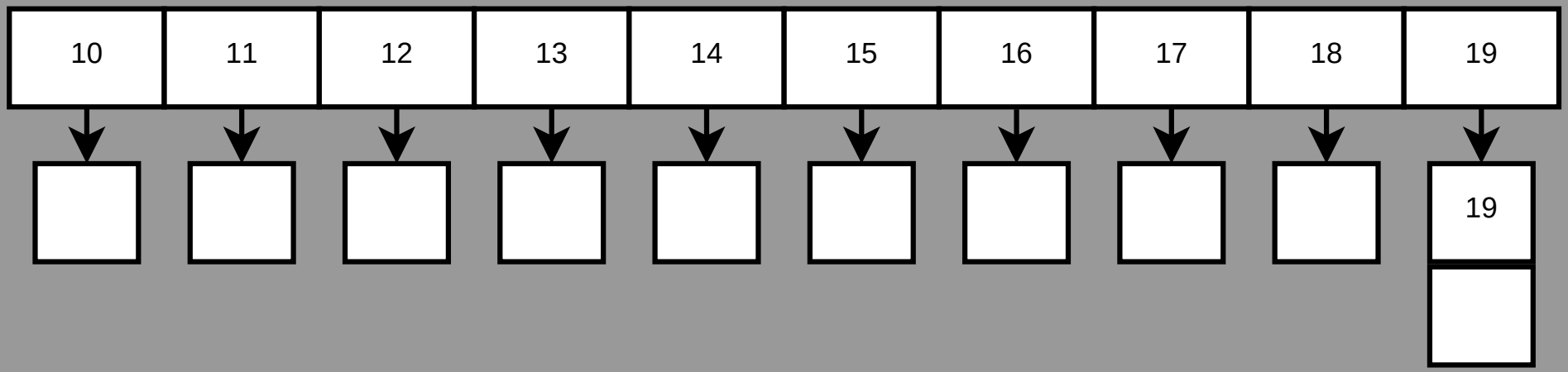
- If the keys are numbered $0...N$ and the buckets are also numbered $0...N$, then the evaluation function just returns the key itself
- If the keys are numbered $10...N$ and the buckets are numbered $0...N-10$, the the evaluation function would map a key k to $(k-10)$

Bucket Sort Example

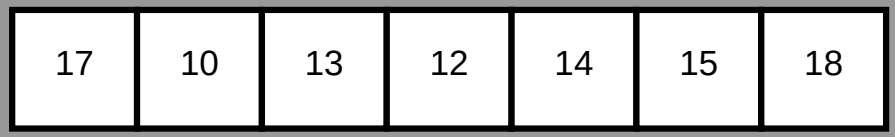


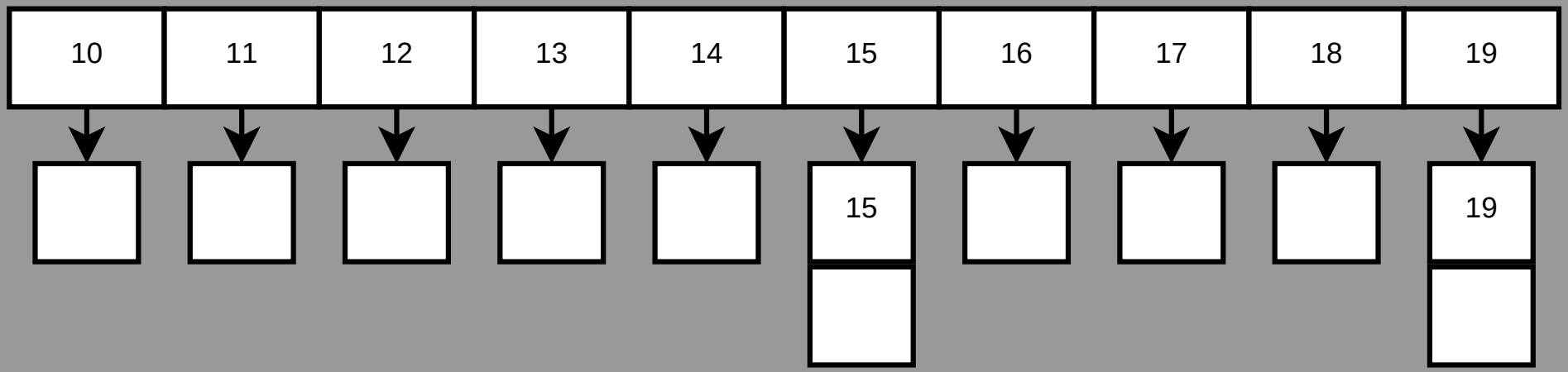
Bucket Sort Example



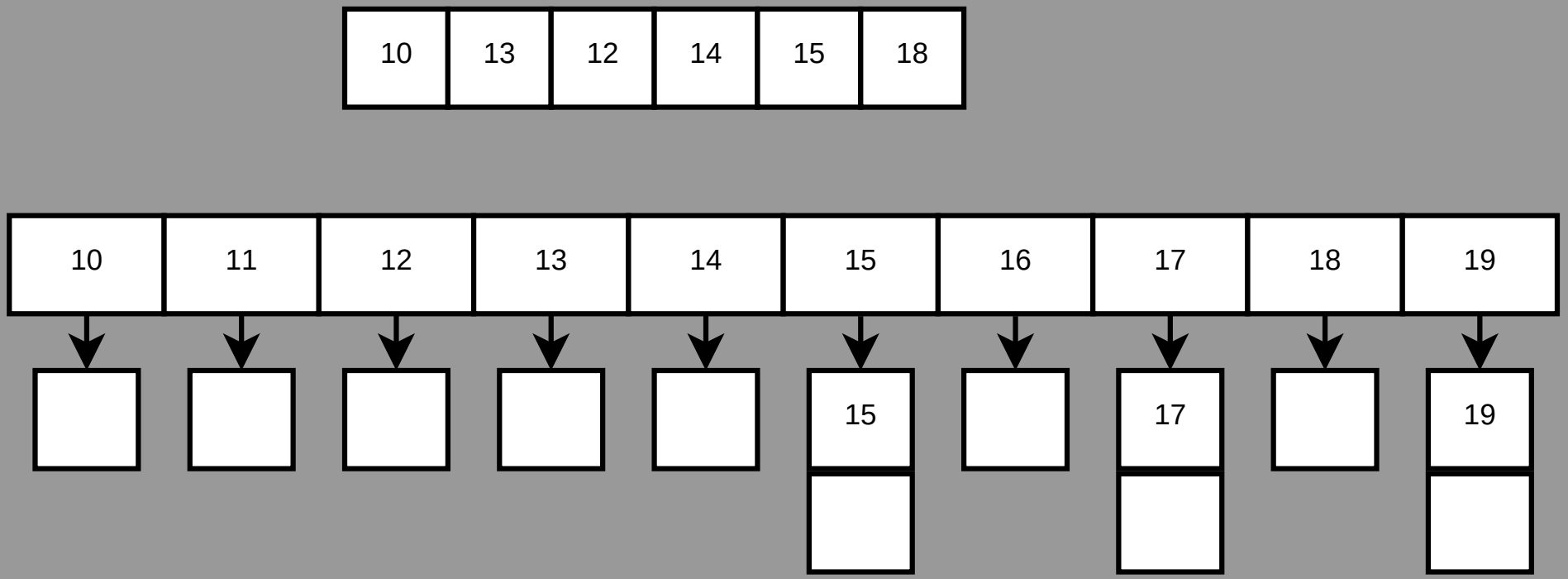


Bucket Sort Example

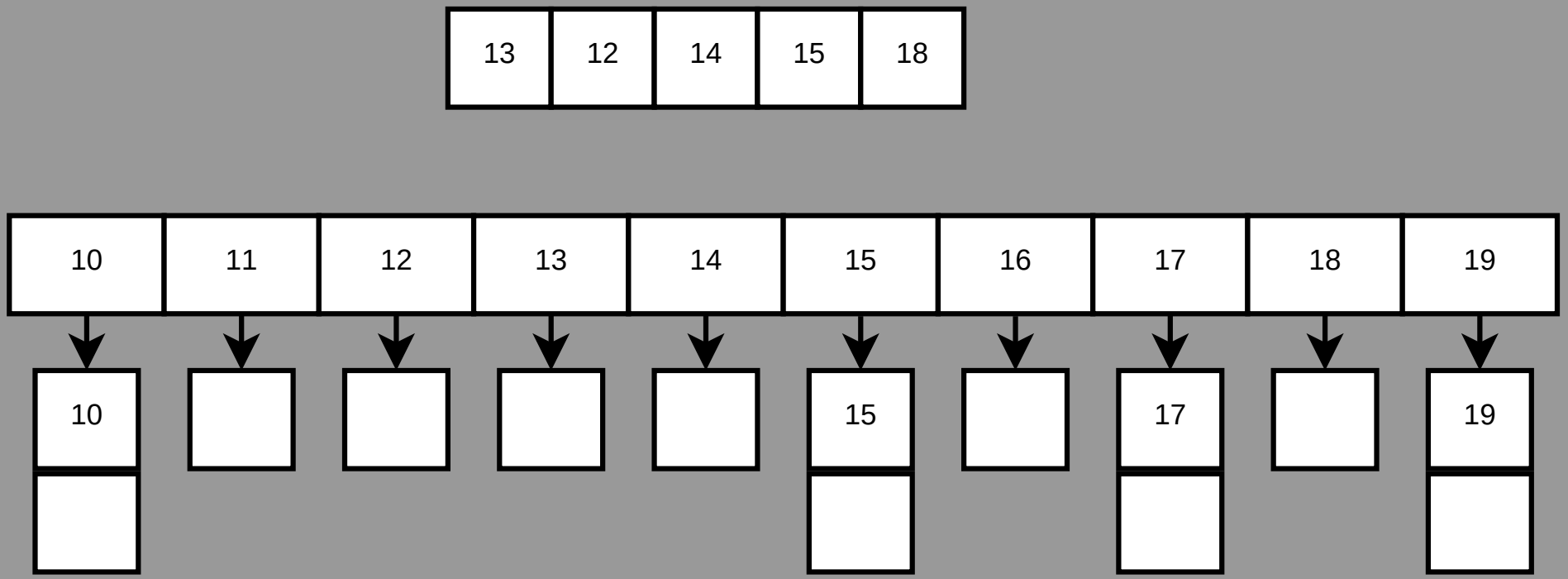




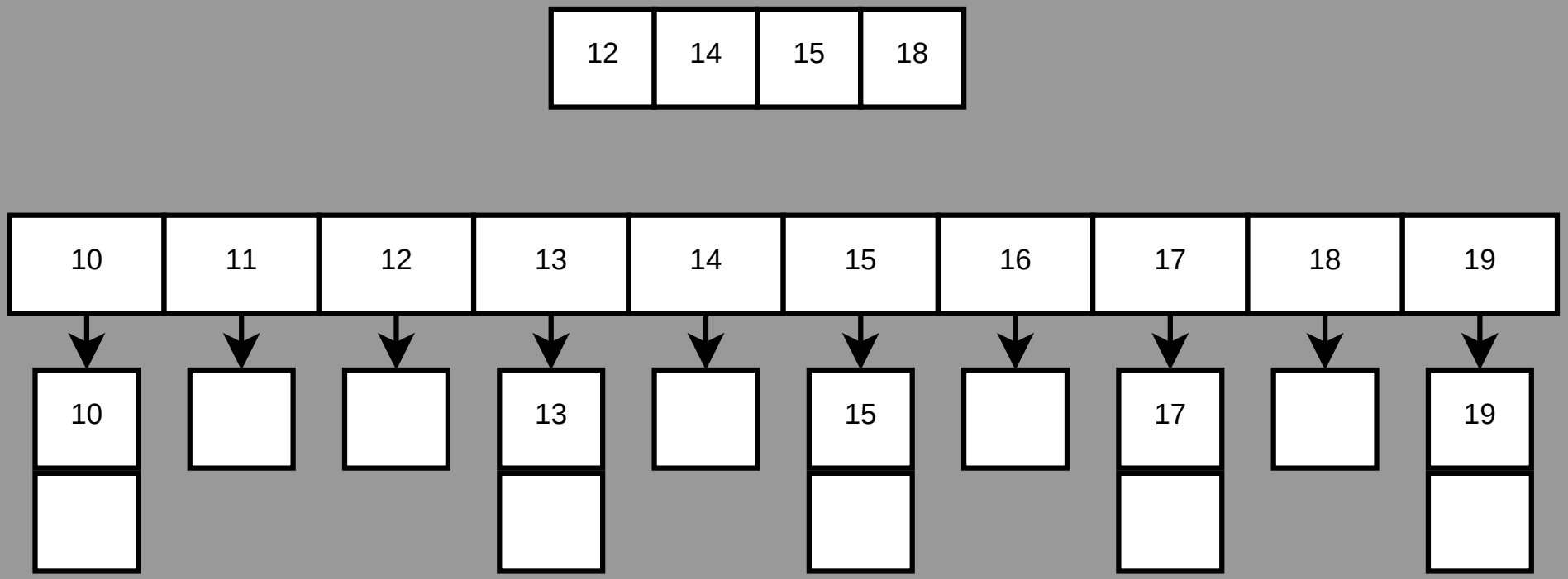
Bucket Sort Example



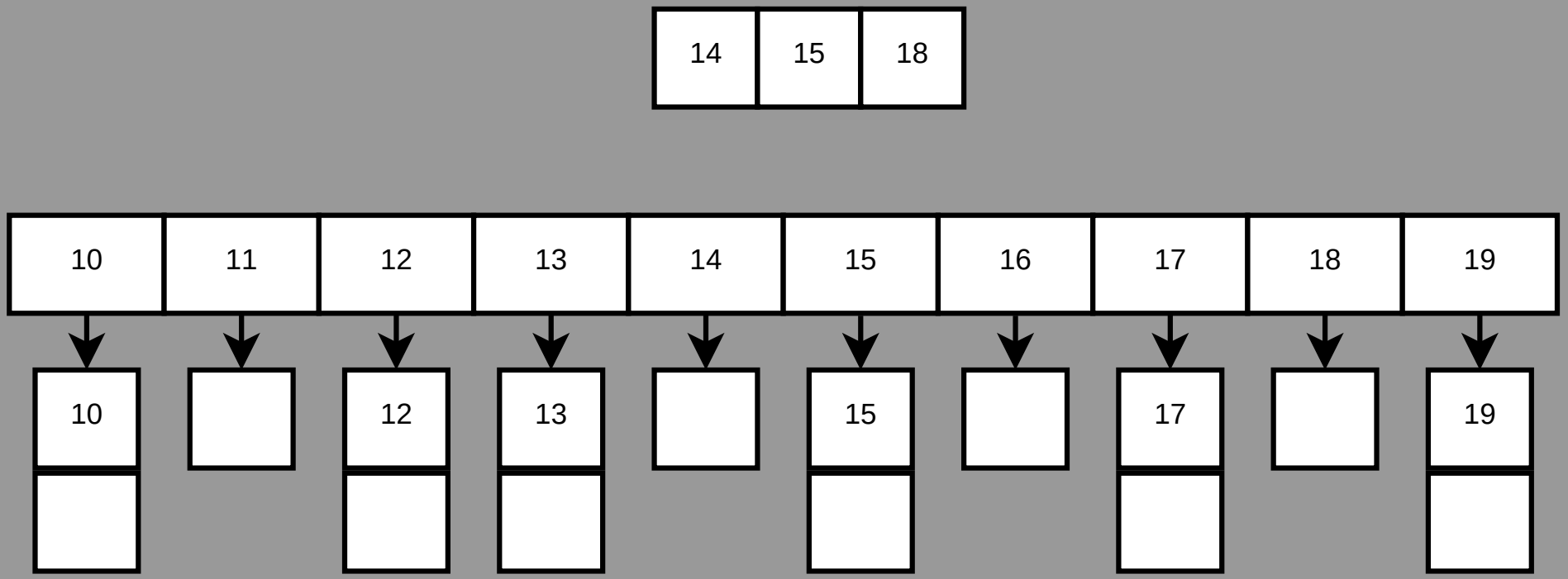
Bucket Sort Example



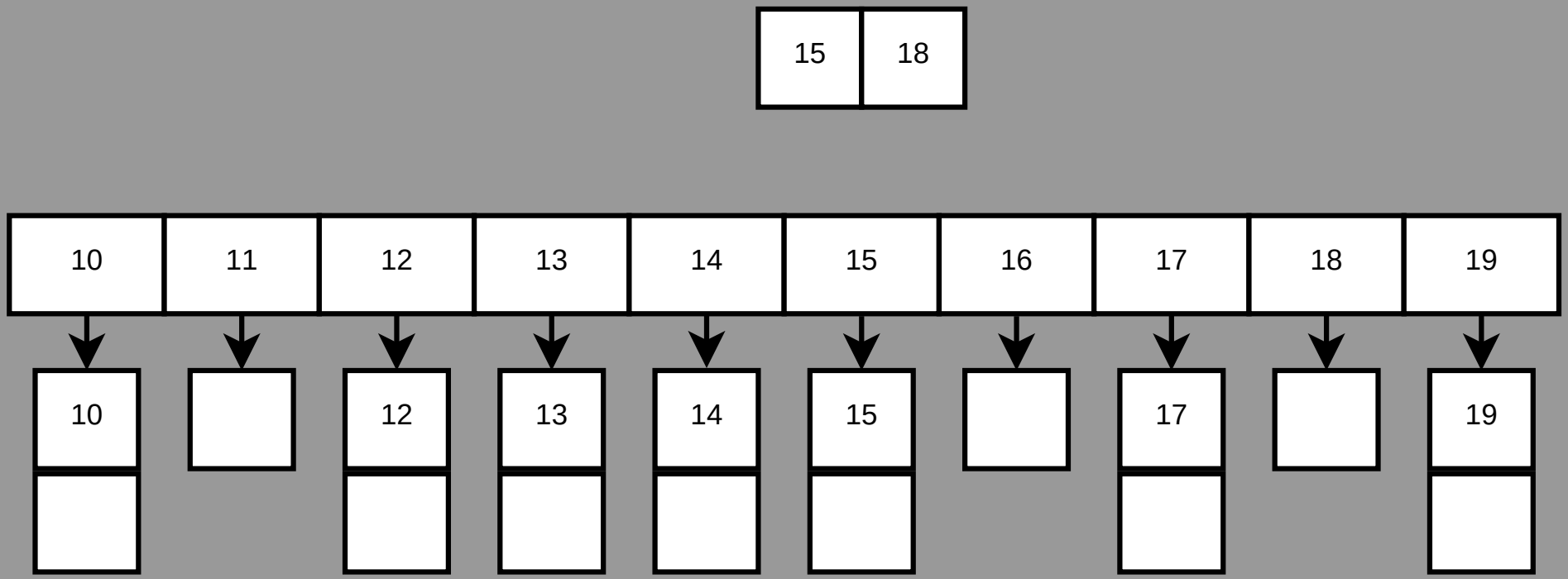
Bucket Sort Example



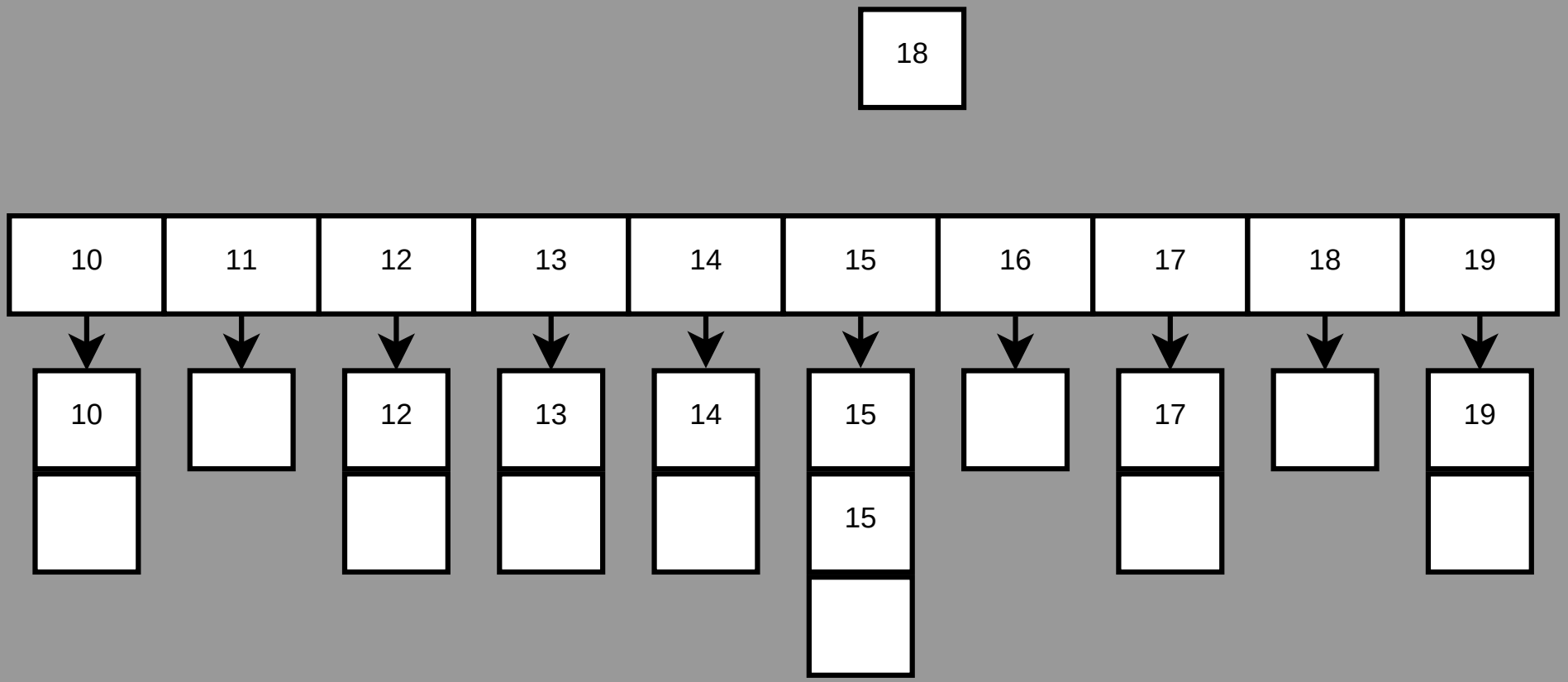
Bucket Sort Example



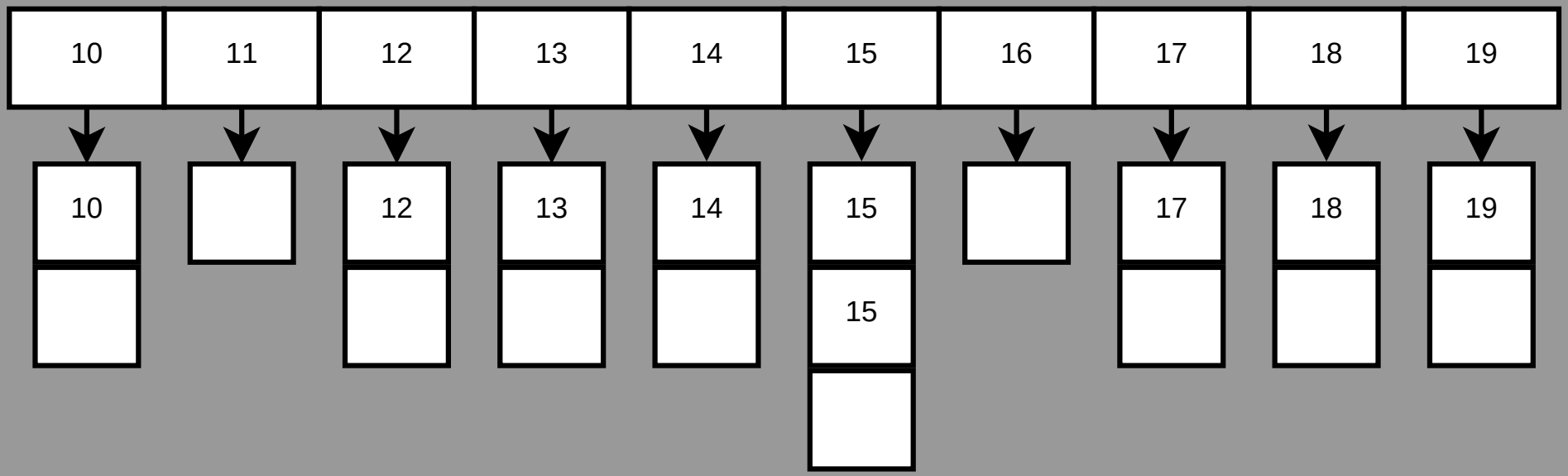
Bucket Sort Example



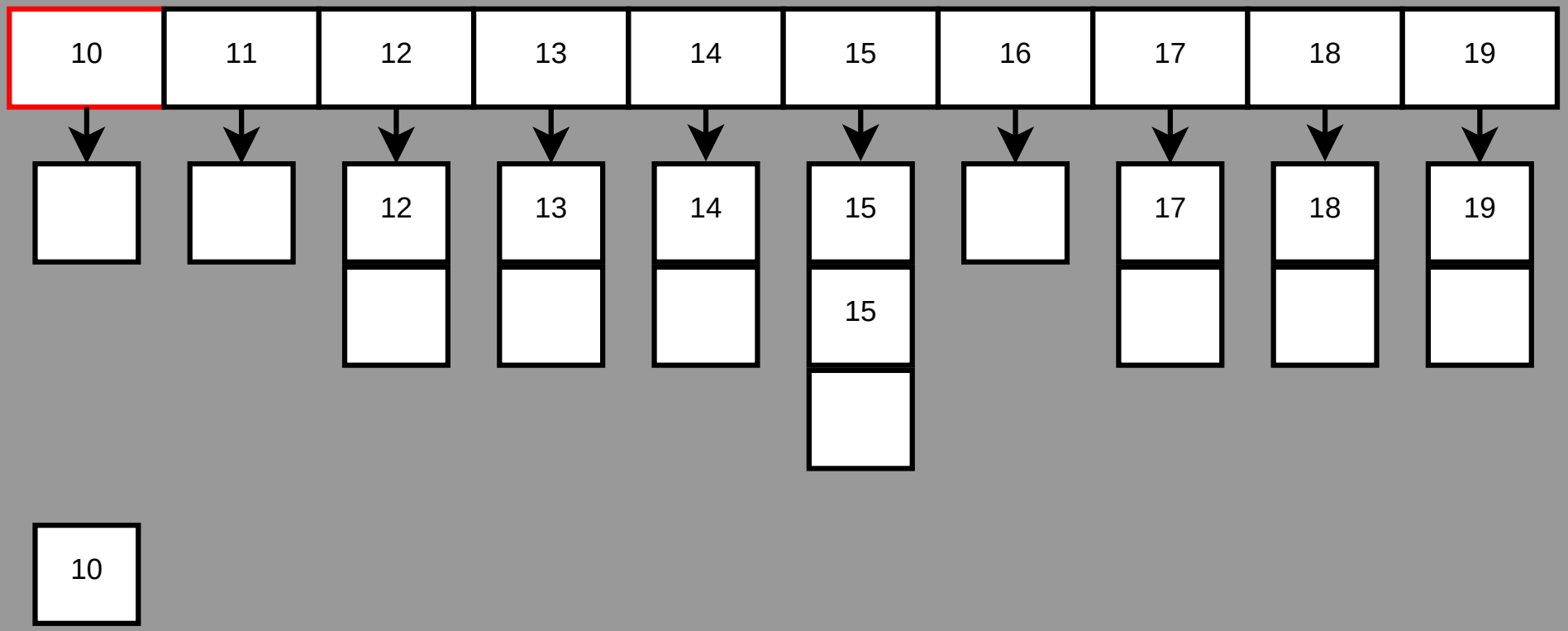
Bucket Sort Example



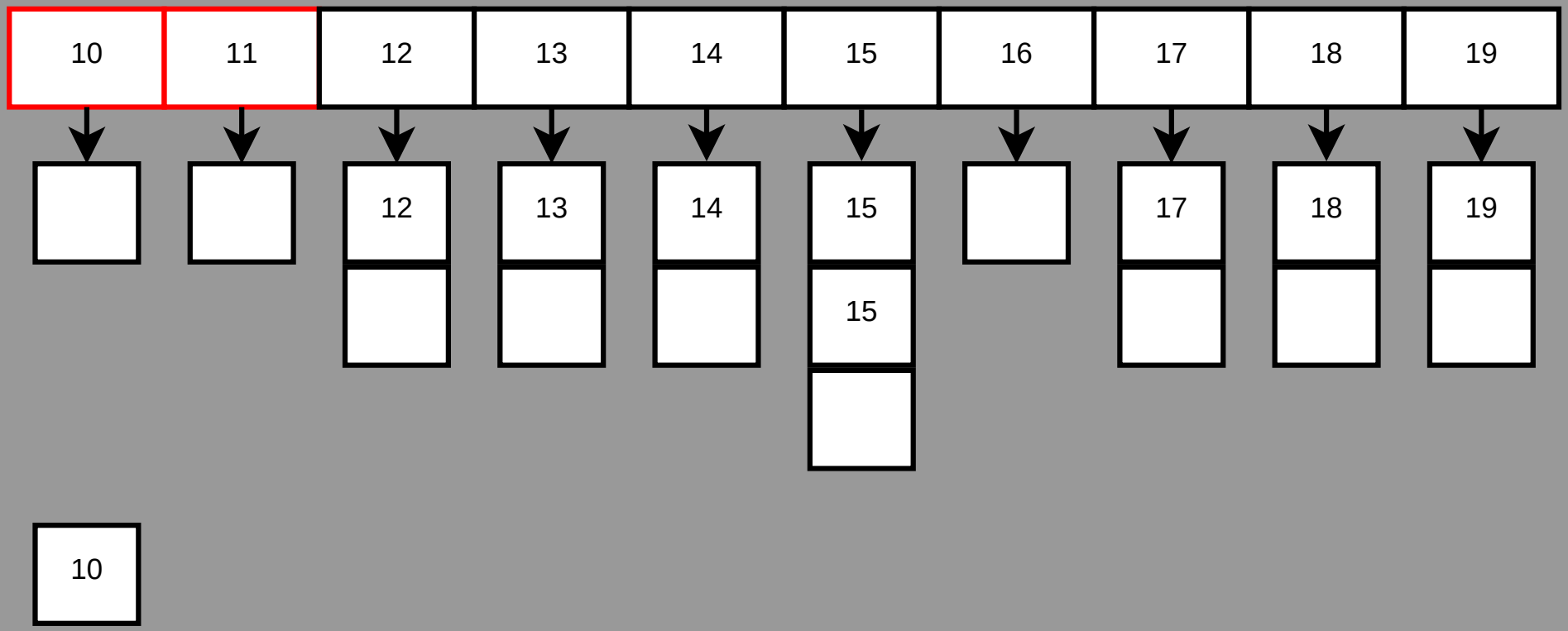
Bucket Sort Example



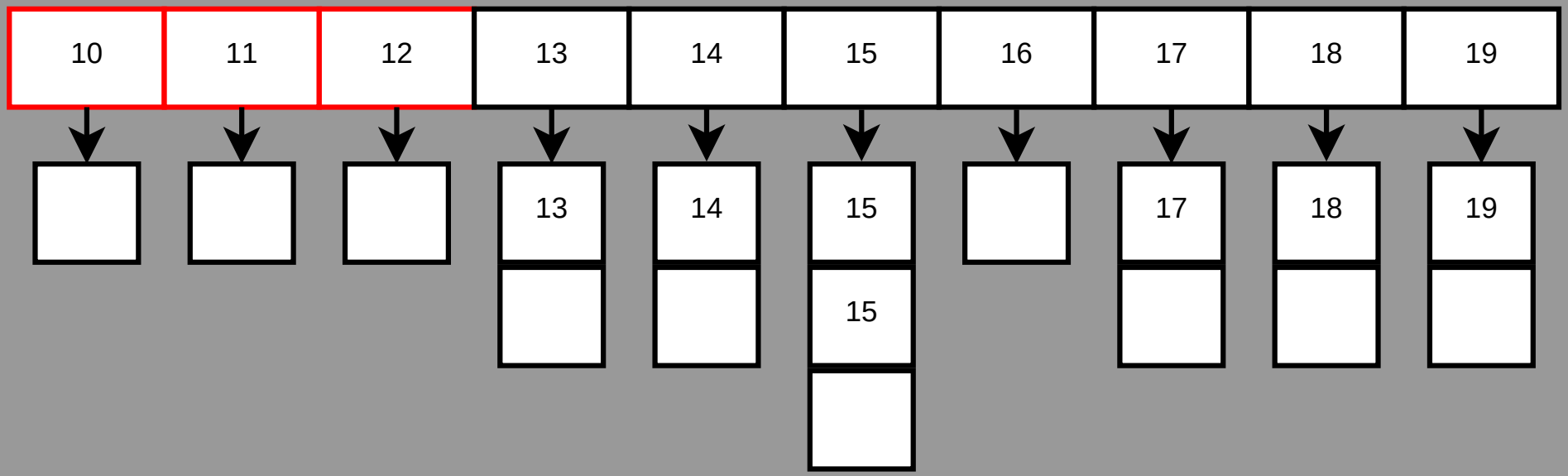
Bucket Sort Example

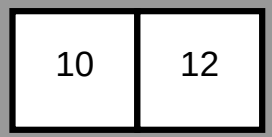


Bucket Sort Example

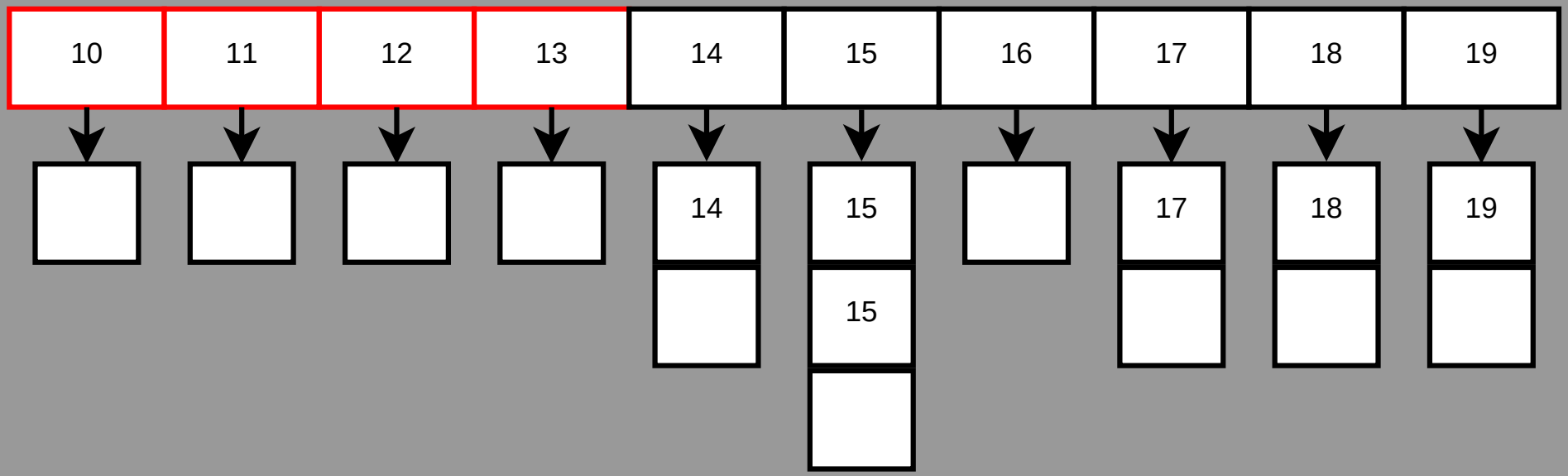


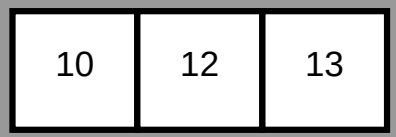
Bucket Sort Example



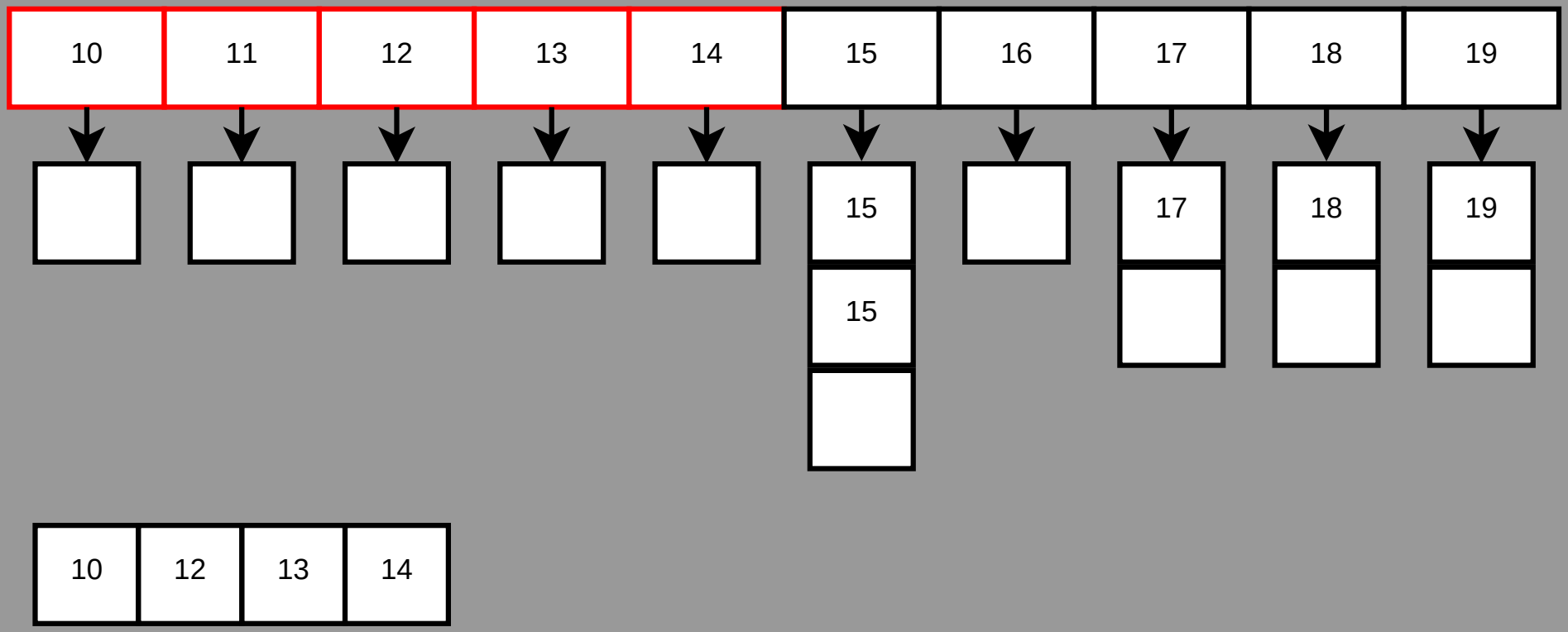


Bucket Sort Example

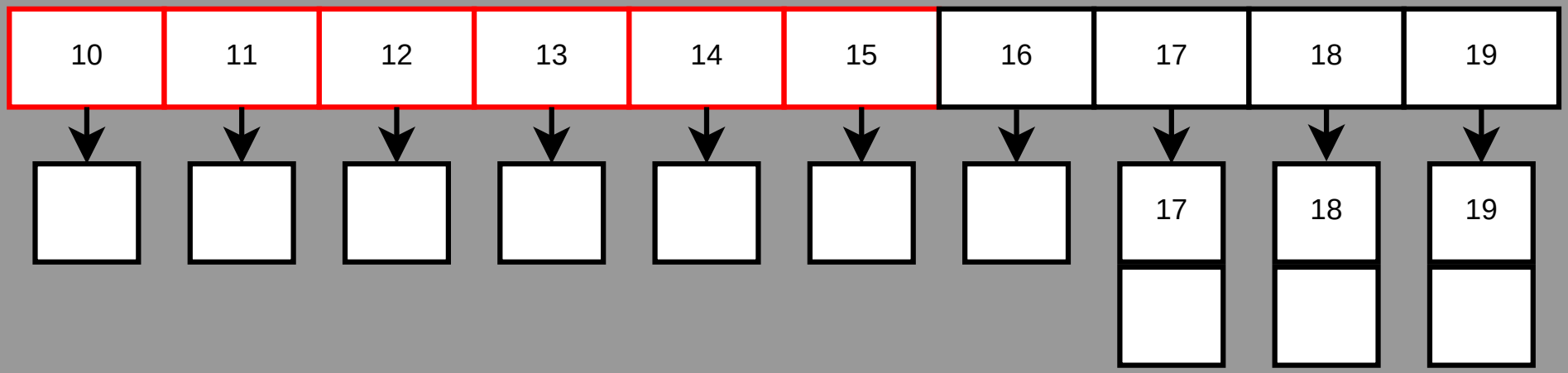




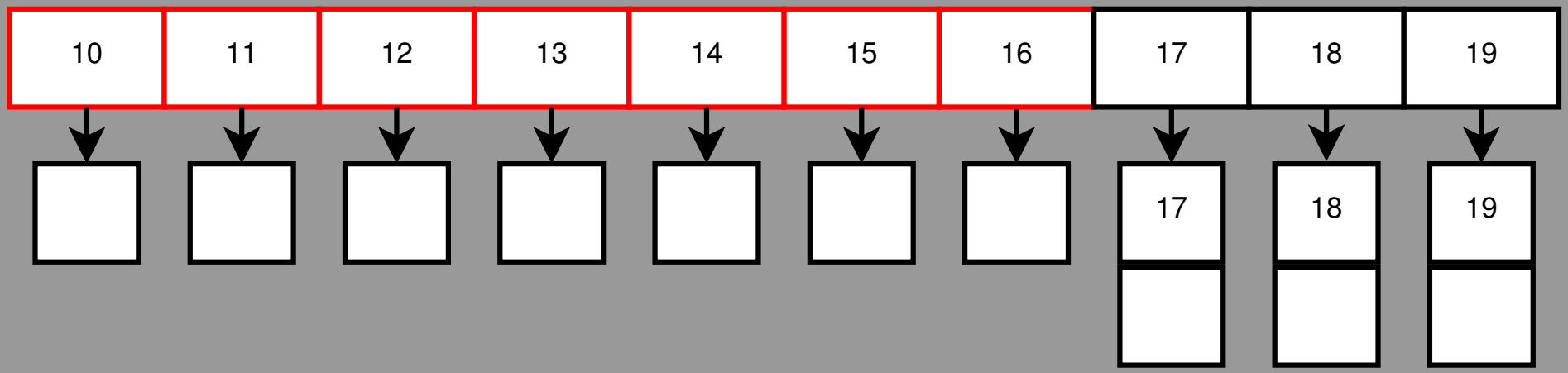
Bucket Sort Example

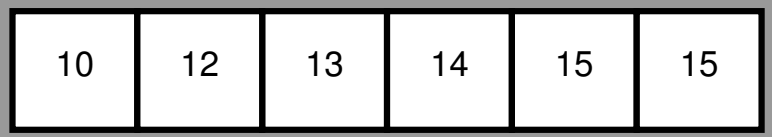


Bucket Sort Example

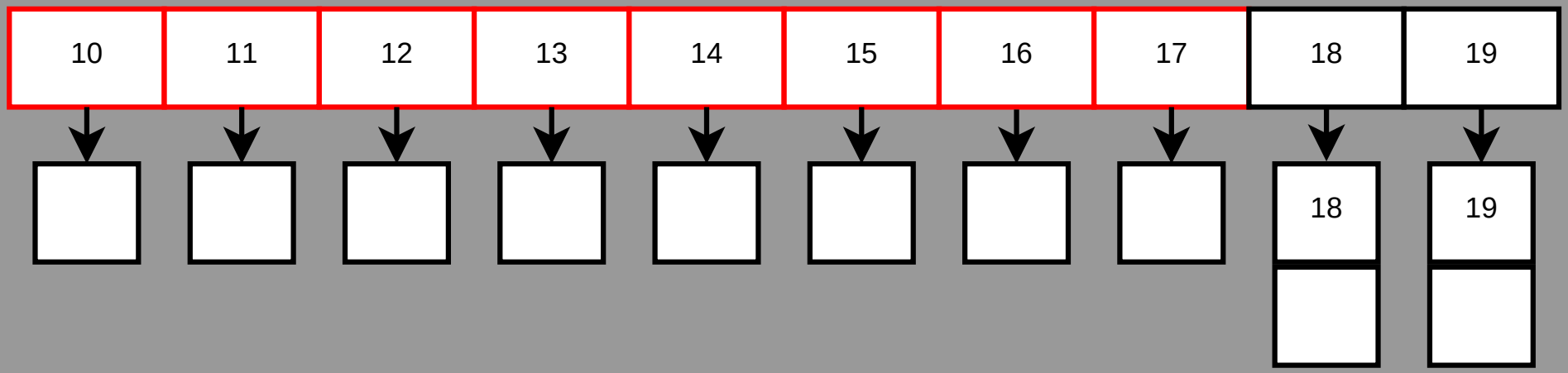


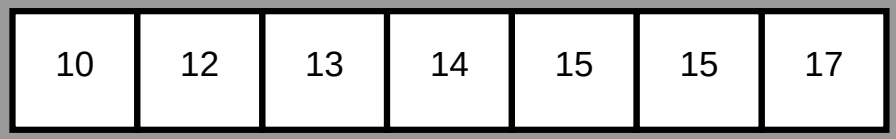
Bucket Sort Example



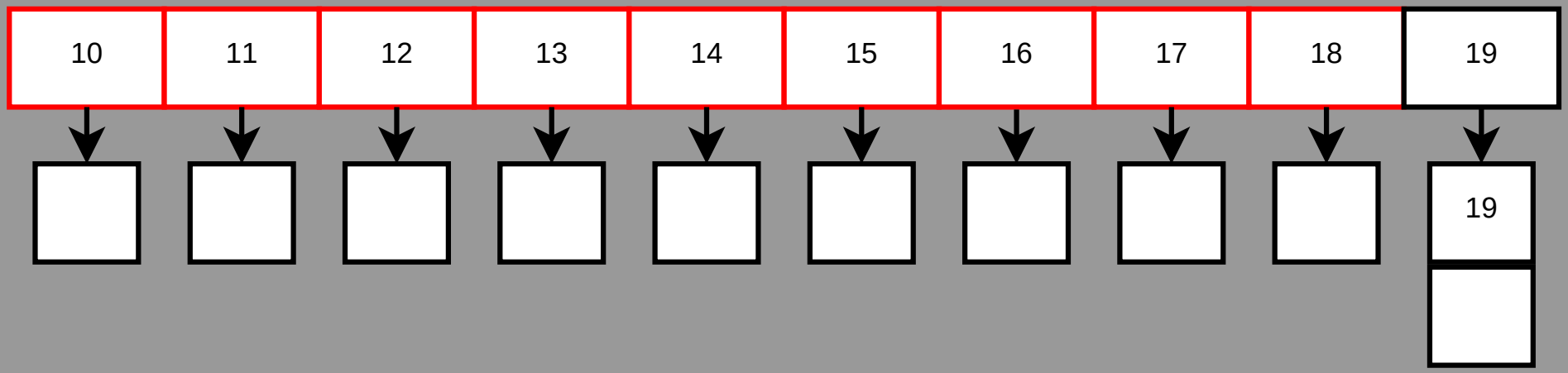


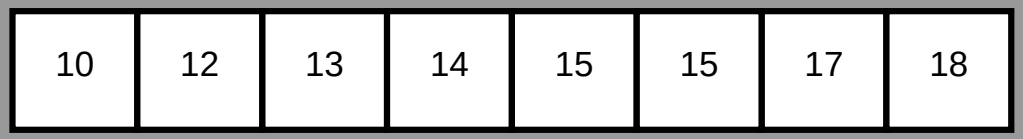
Bucket Sort Example



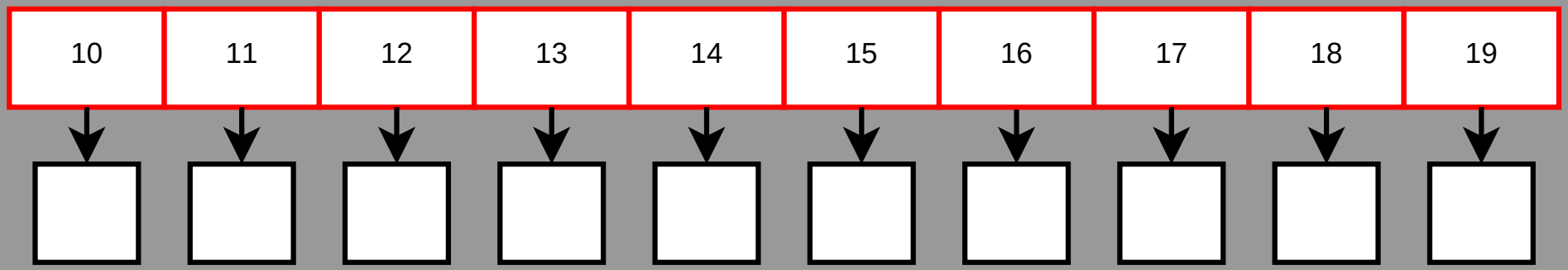


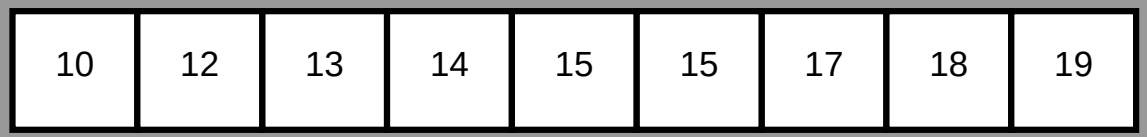
Bucket Sort Example





Bucket Sort Example





Multiple keys in the same bucket

- It is also possible to store different keys in the same buckets
- In this case each bucket usually has a range of keys e.g. 0 - 100
- Items inside each bucket still need to be sorted
  - Usually insertion sort (or some similar sorting algorithm) is used to sort each bucket

Complexity of Bucket Sort

- If we have an array of n elements with N possible key values (i.e. N buckets)
- Then inserting consist of the following operations:
  - Calling the evaluation function $O(1)$
  - Finding the correct index in the array $O(1)$
  - Inserting the value into the linked list $O(1)$
  - This gives us an overall insertion complexity of $O(1)$
- Every element is added to a bucket once:
  - n insertions of $O(1) = O(n)$
- When we remove the items from the buckets:
  - Every bucket is visited once N times
  - In total we will remove n elements from the buckets n times
  - Cost of removing an element is $O(1)$
  - N buckets + n elements * $O(1)$ time $= O(N + n)$

Table of Contents

Radix Sort

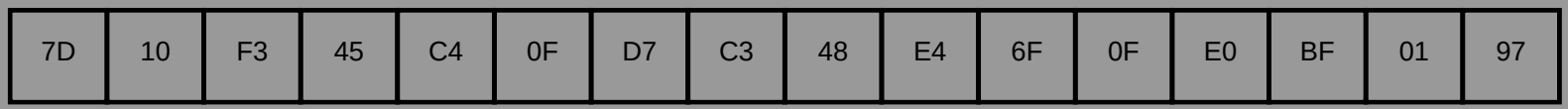
Radix

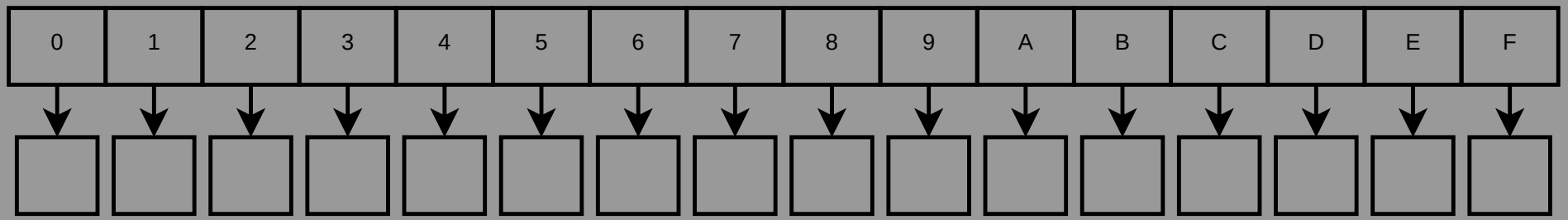
A radix is the base of a numbering system, it means the number of unique digits.
The radix of binary is 2, and decimal is 10.

- Radix sort is a multiple pass distribution sort
  - It uses the bucket sort algorithm multiple times to properly sort data
  - At each stage it distributes each item into a bucket based on part of the item's key
  - After each pass the items are collected and kept in the same order
  - They are then distributed again using a different part of the key
- For numbers, it can sort by digits or a number of digits, for strings it sorts character by character

Radix Sort Example

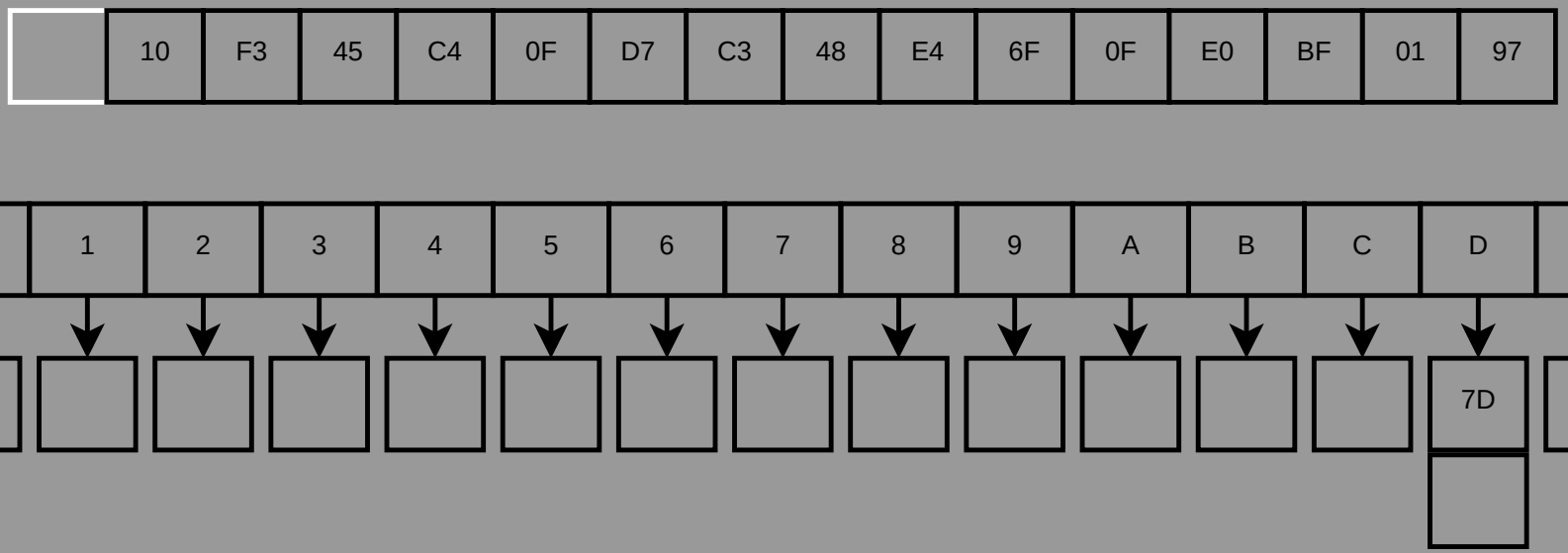
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the least significant digit (rightmost)





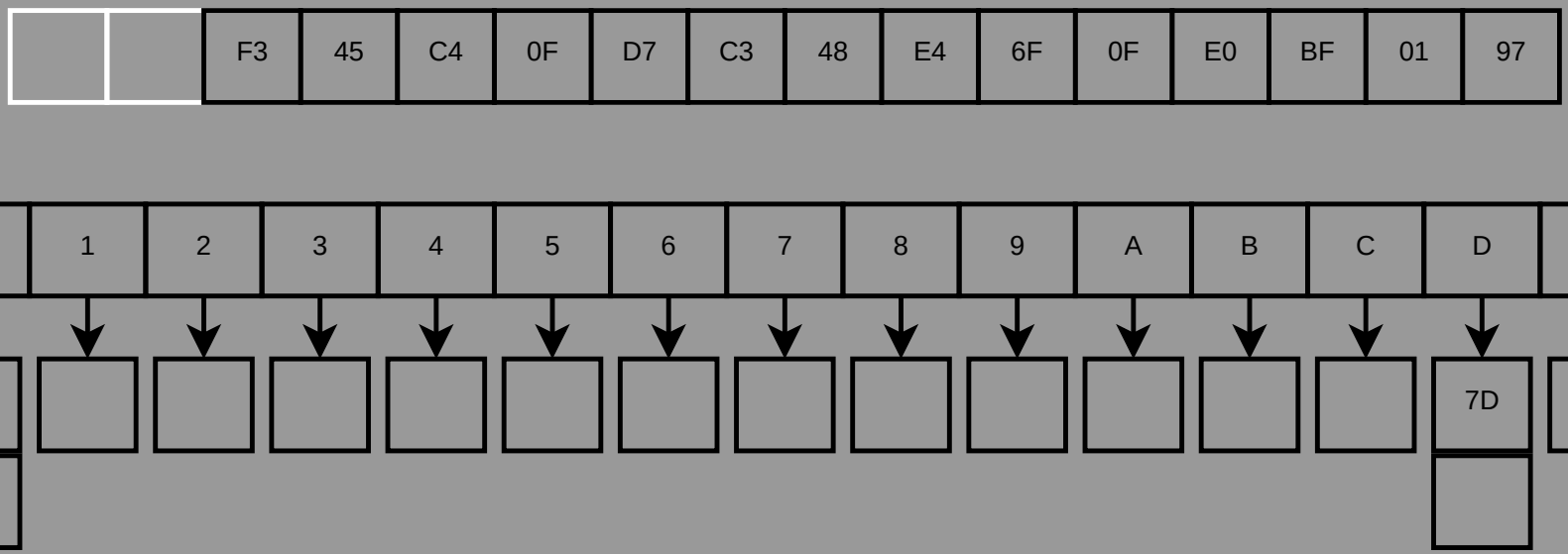
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the least significant digit (rightmost)



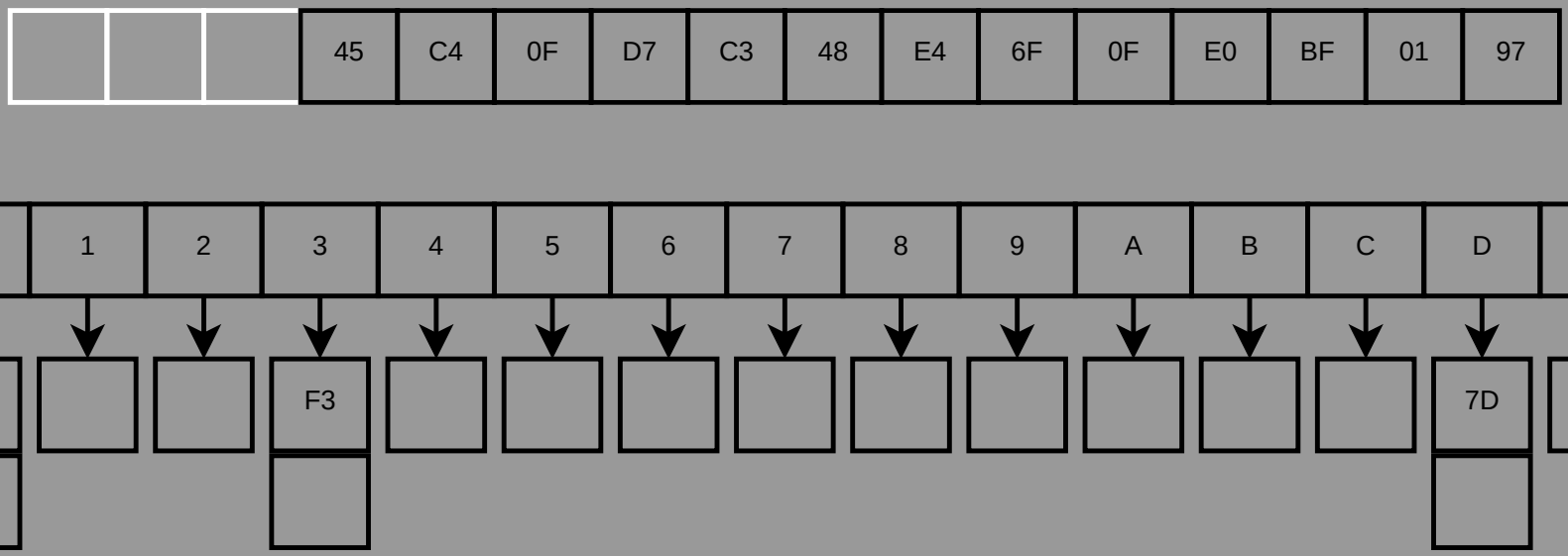
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the least significant digit (rightmost)



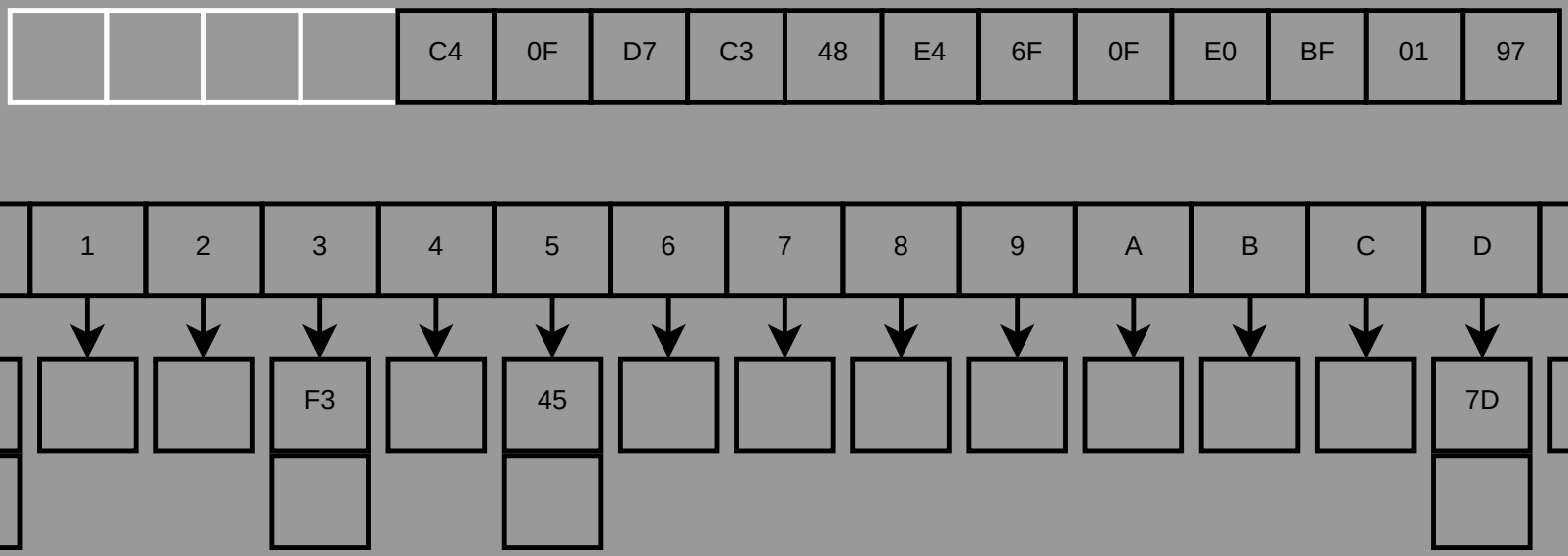
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the least significant digit (rightmost)



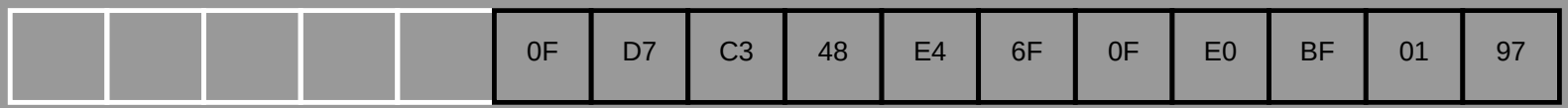
Radix Sort Example

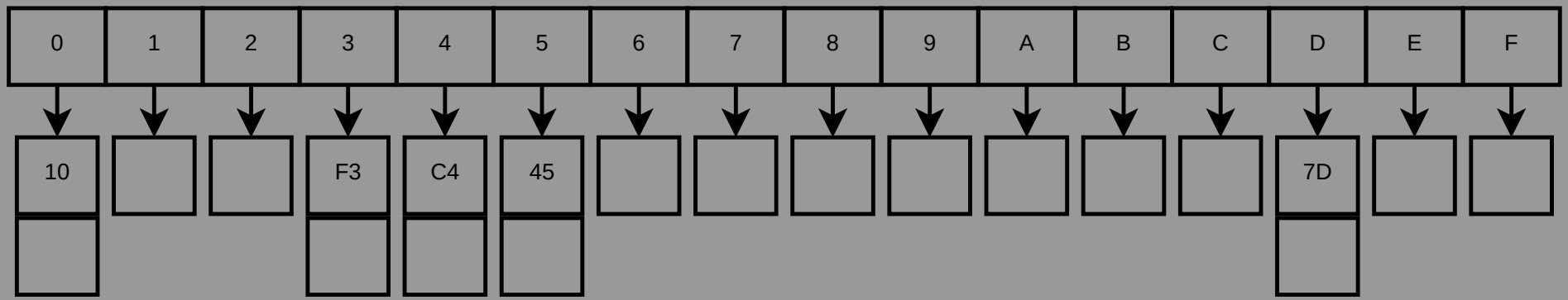
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the least significant digit (rightmost)



Radix Sort Example

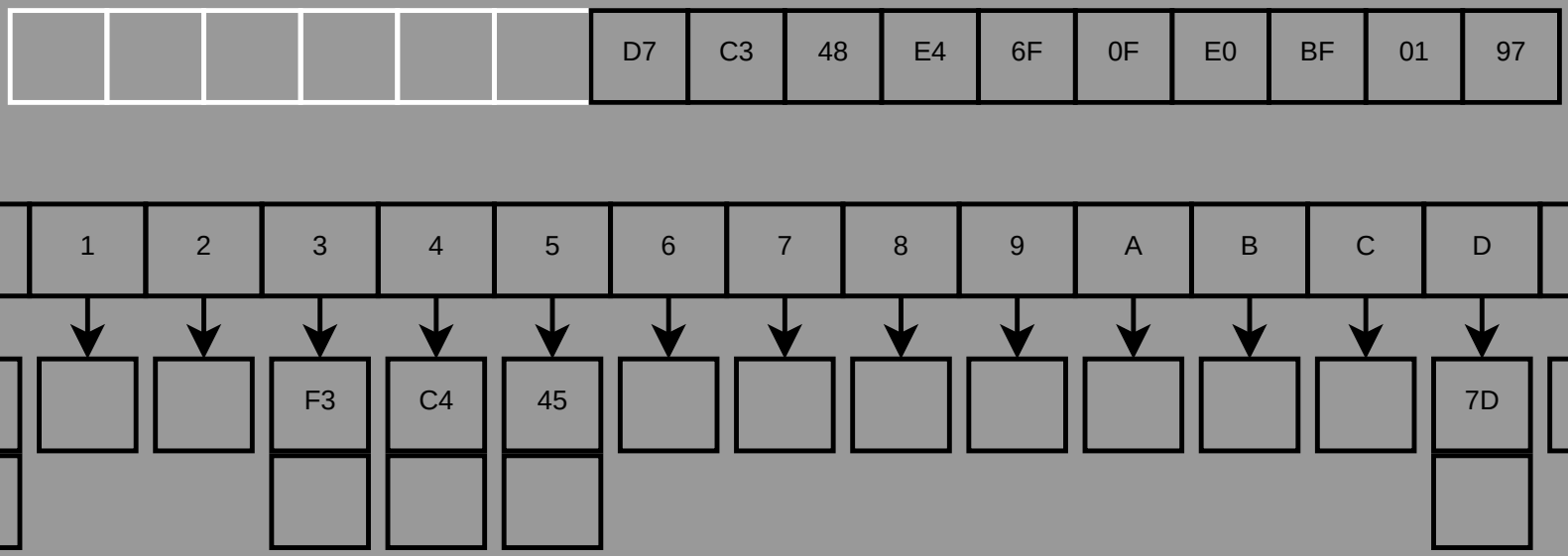
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the least significant digit (rightmost)





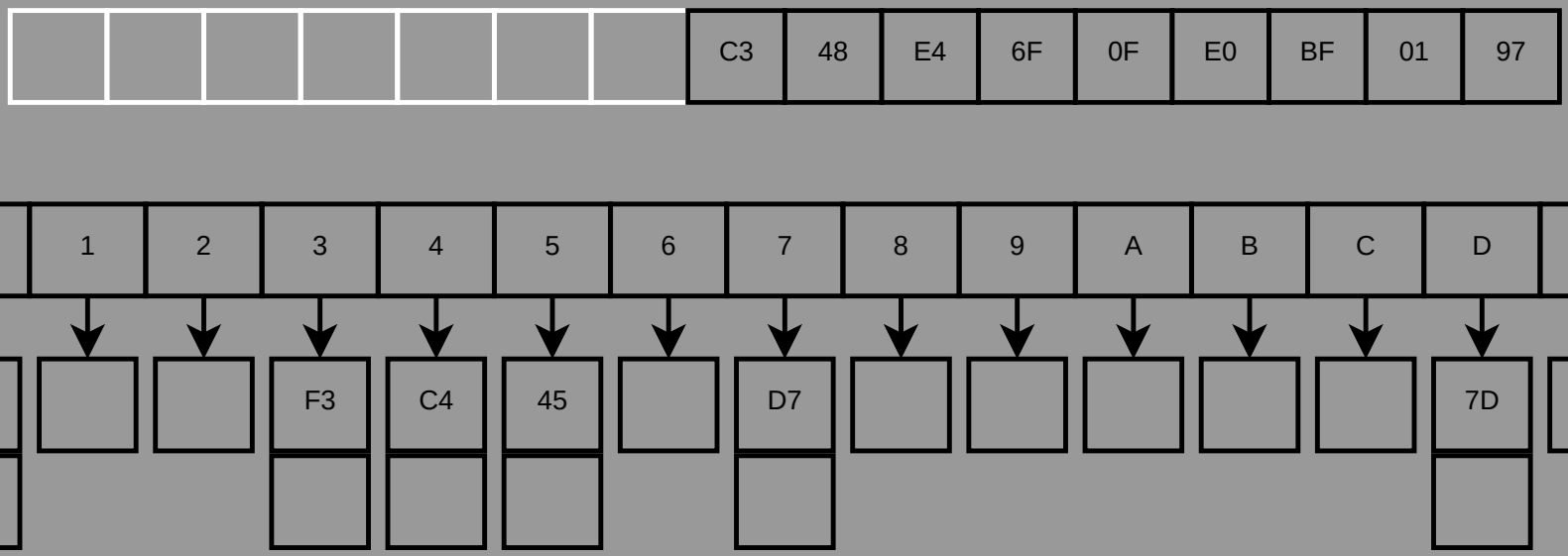
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the least significant digit (rightmost)



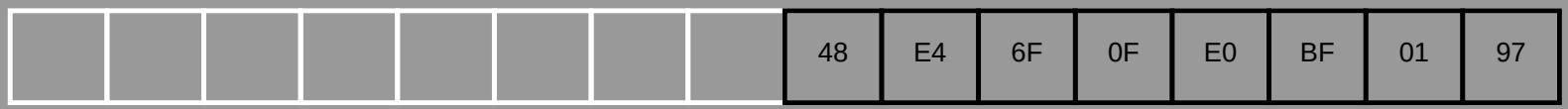
Radix Sort Example

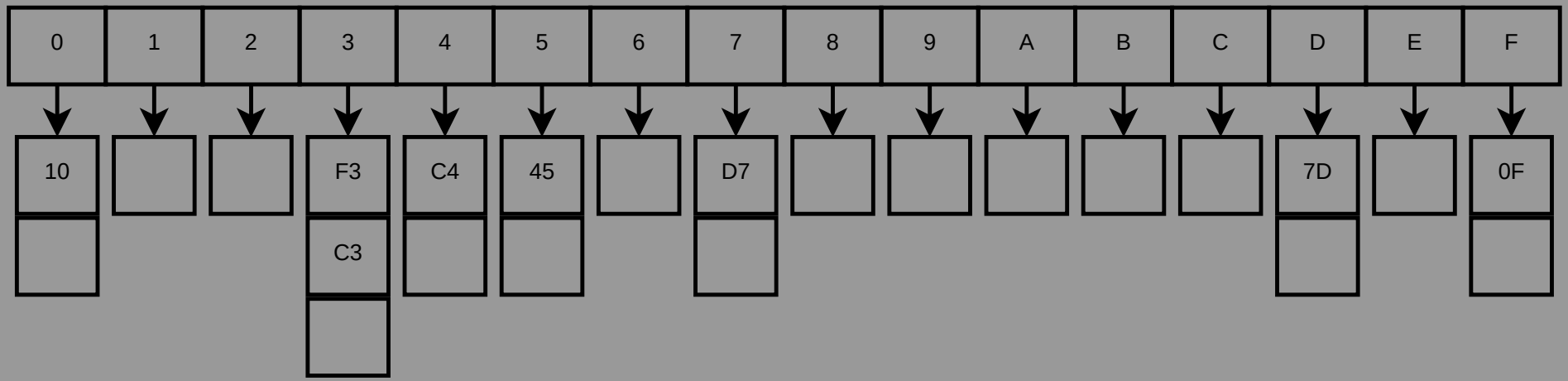
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the least significant digit (rightmost)



Radix Sort Example

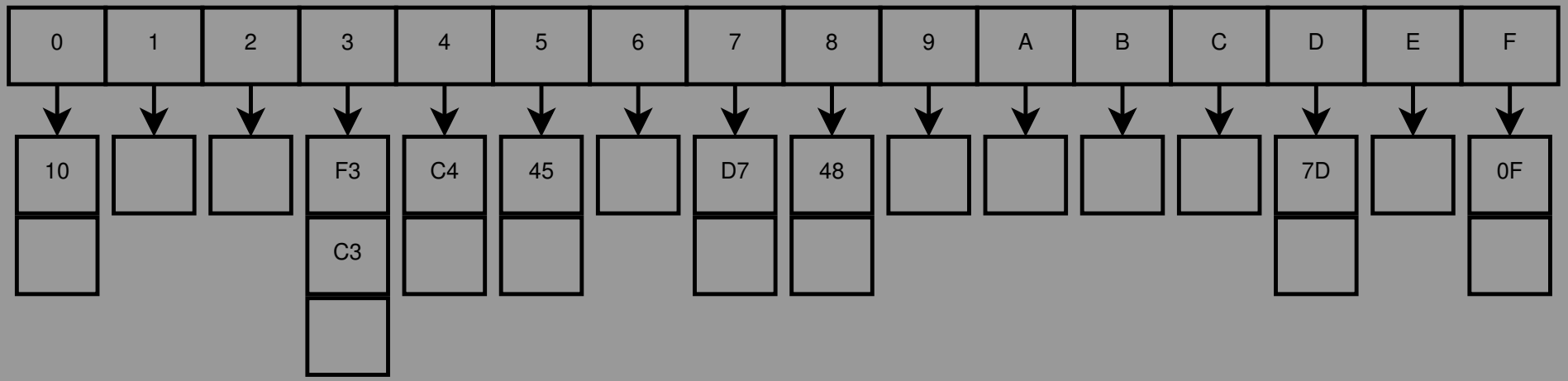
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the least significant digit (rightmost)





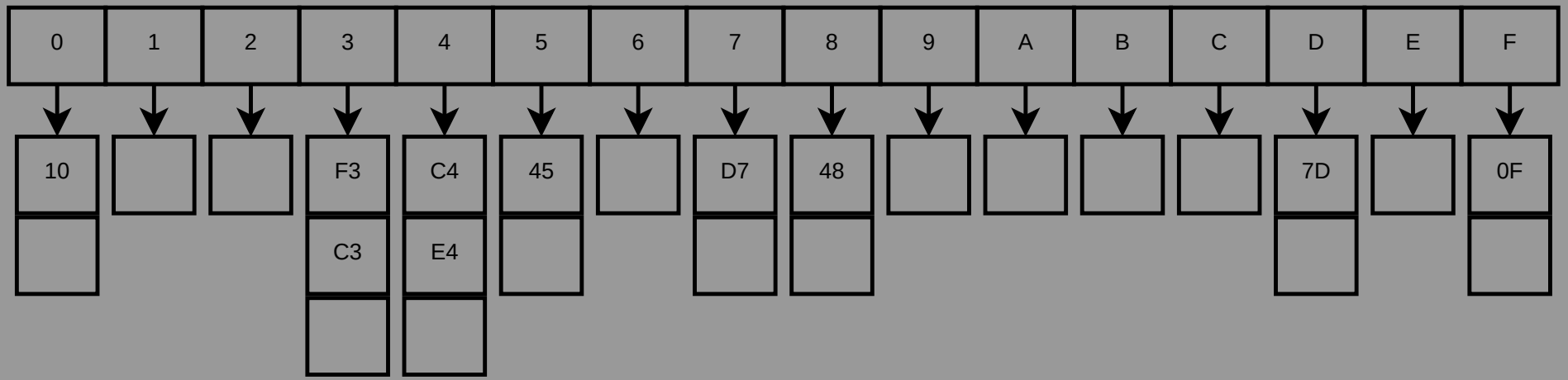
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the least significant digit (rightmost)



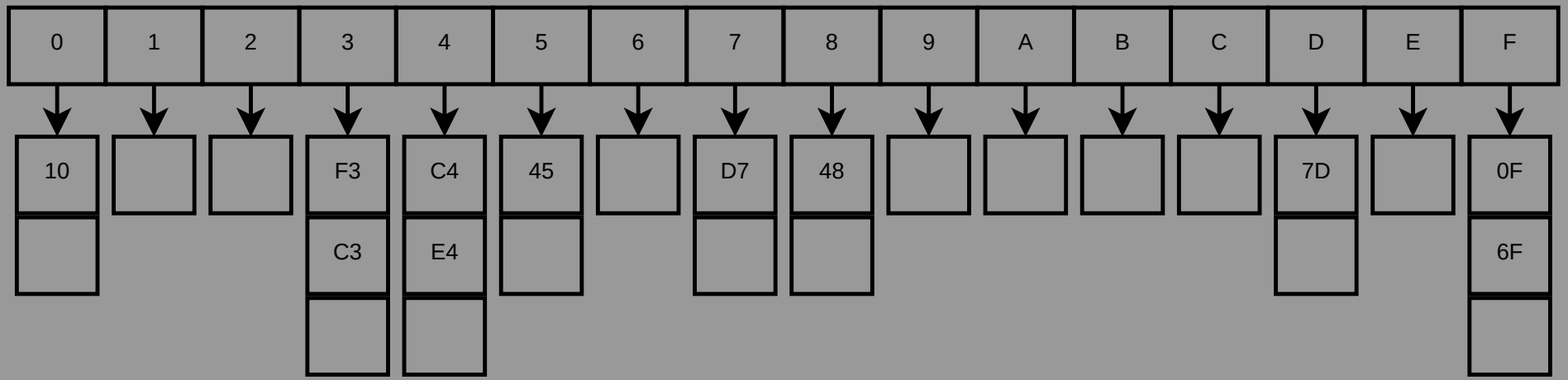
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the least significant digit (rightmost)



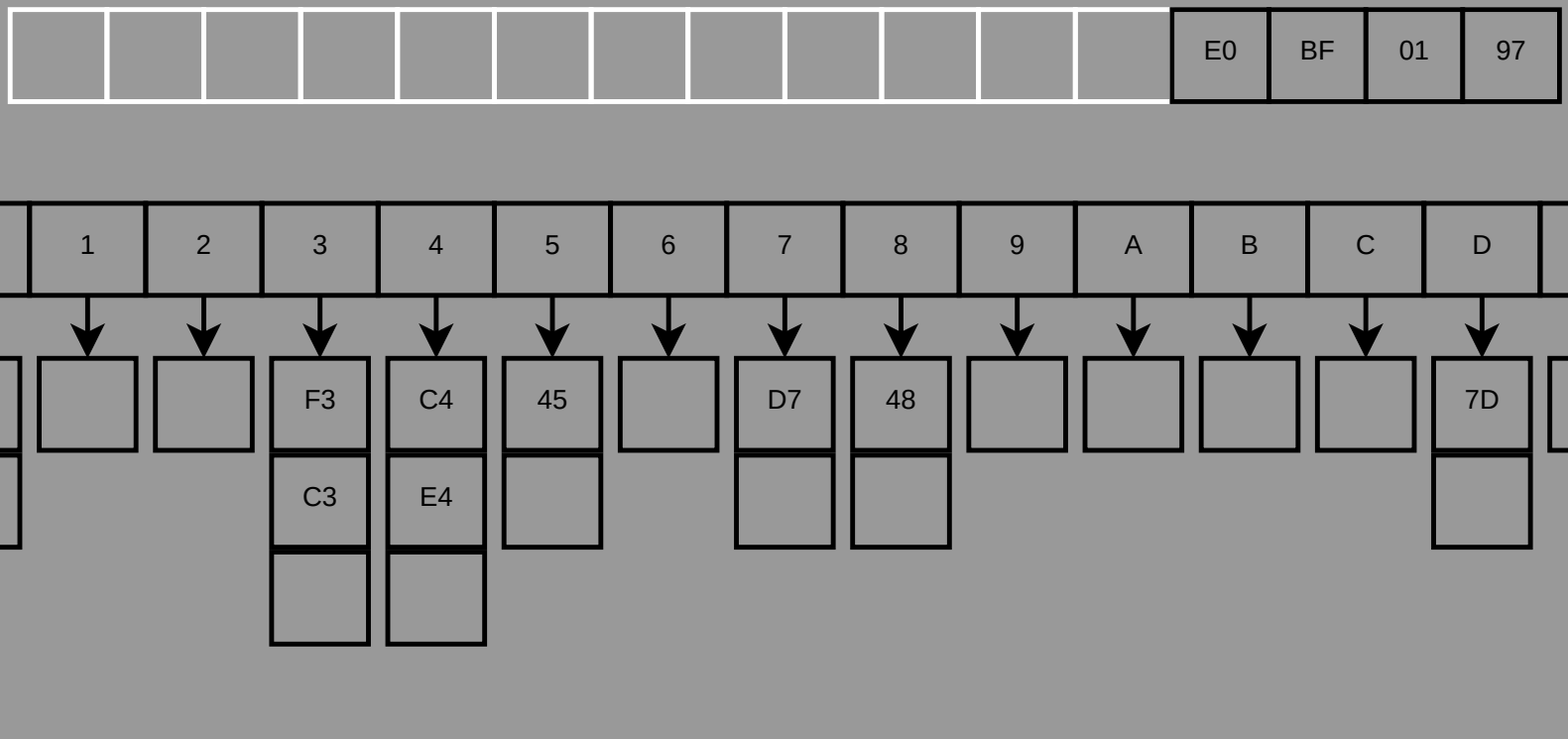
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the least significant digit (rightmost)



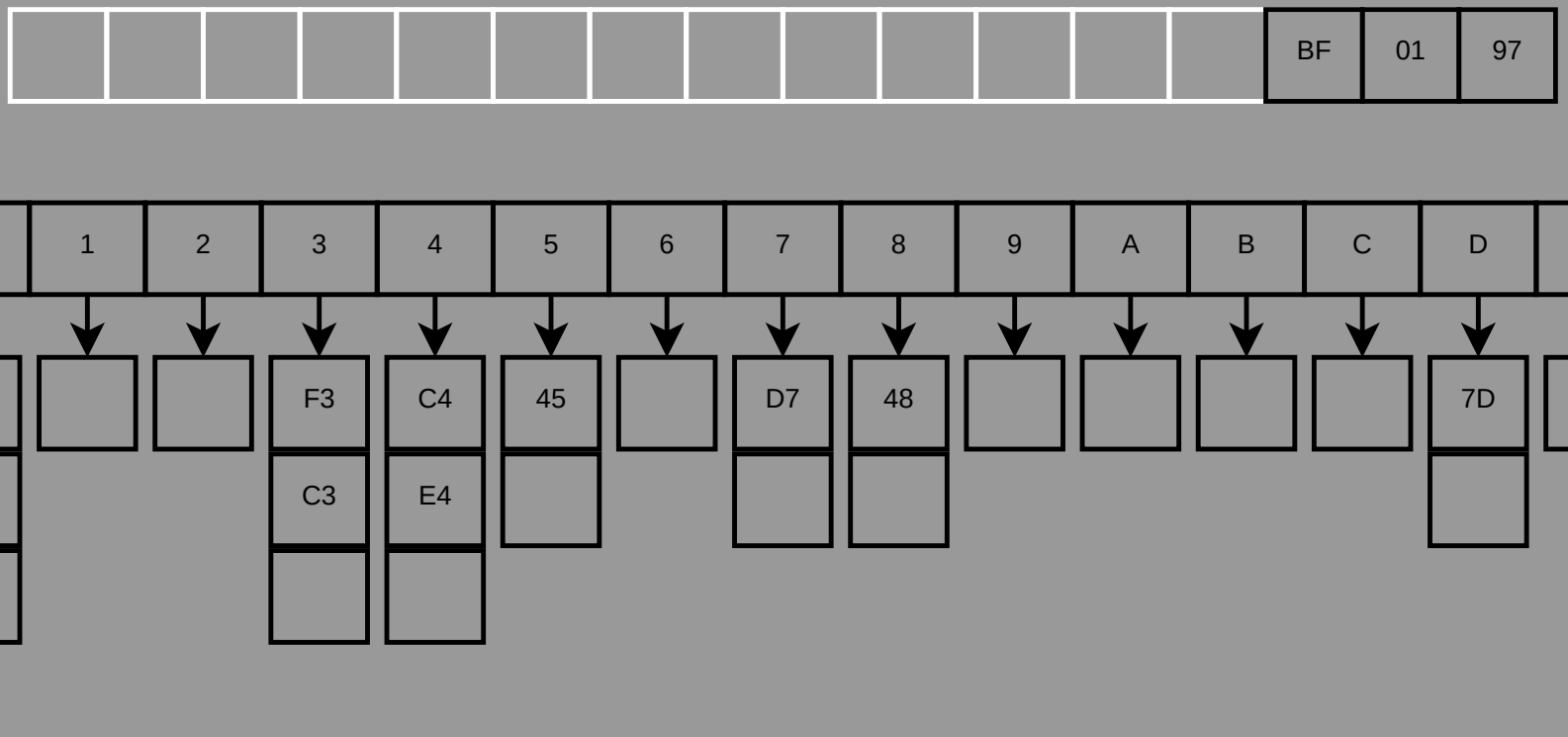
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the least significant digit (rightmost)



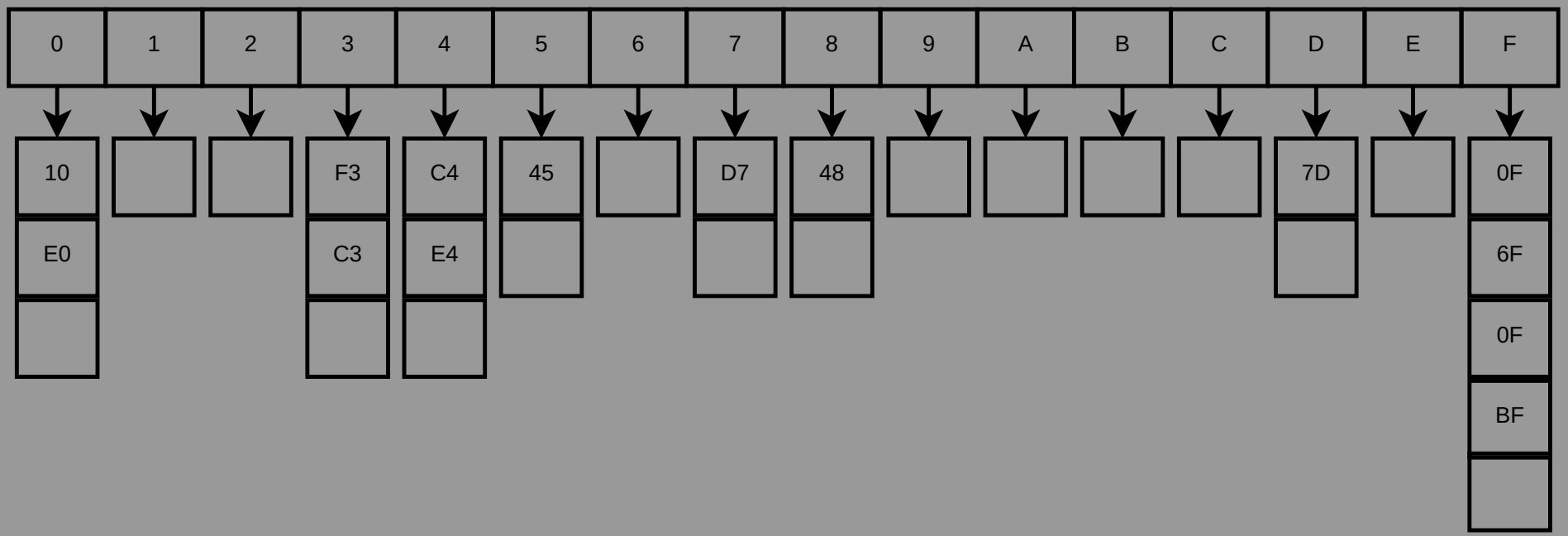
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the least significant digit (rightmost)



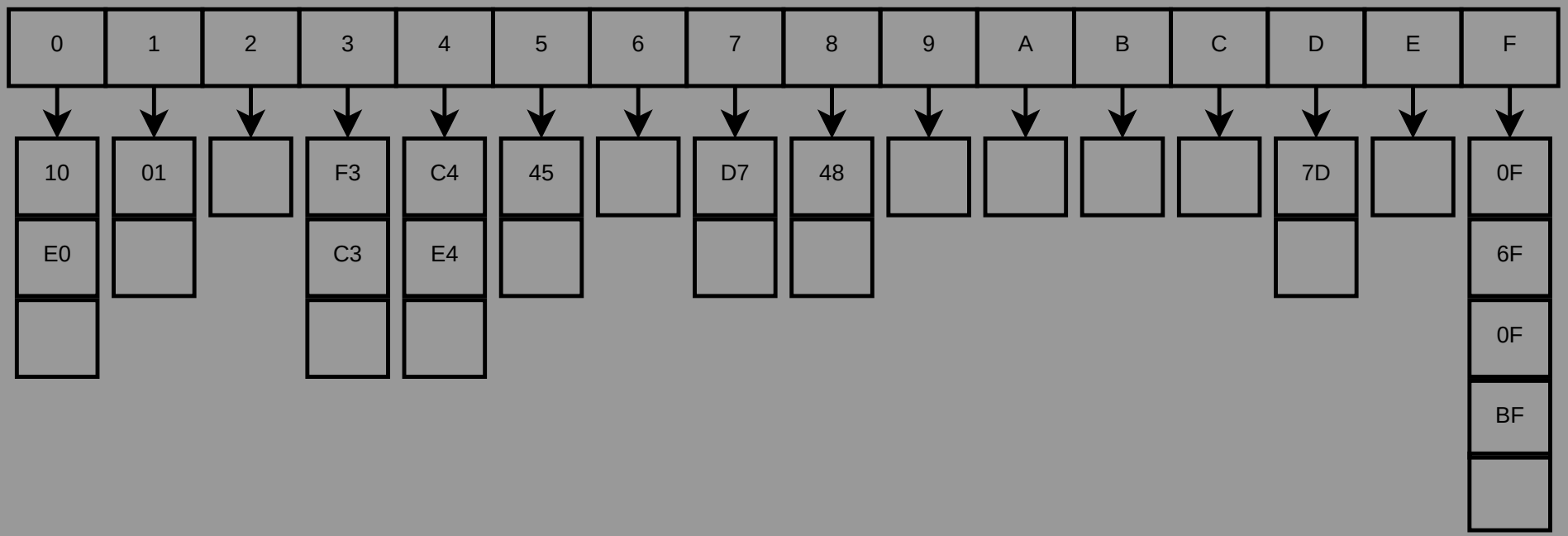
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the least significant digit (rightmost)



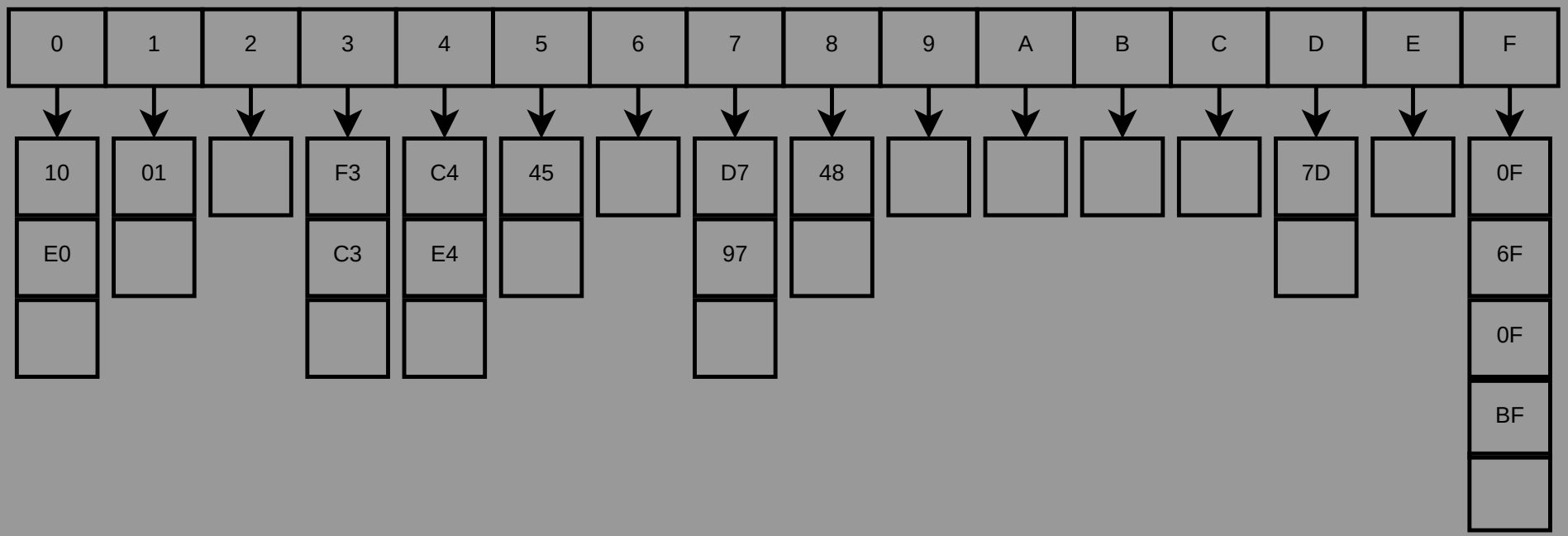
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the least significant digit (rightmost)



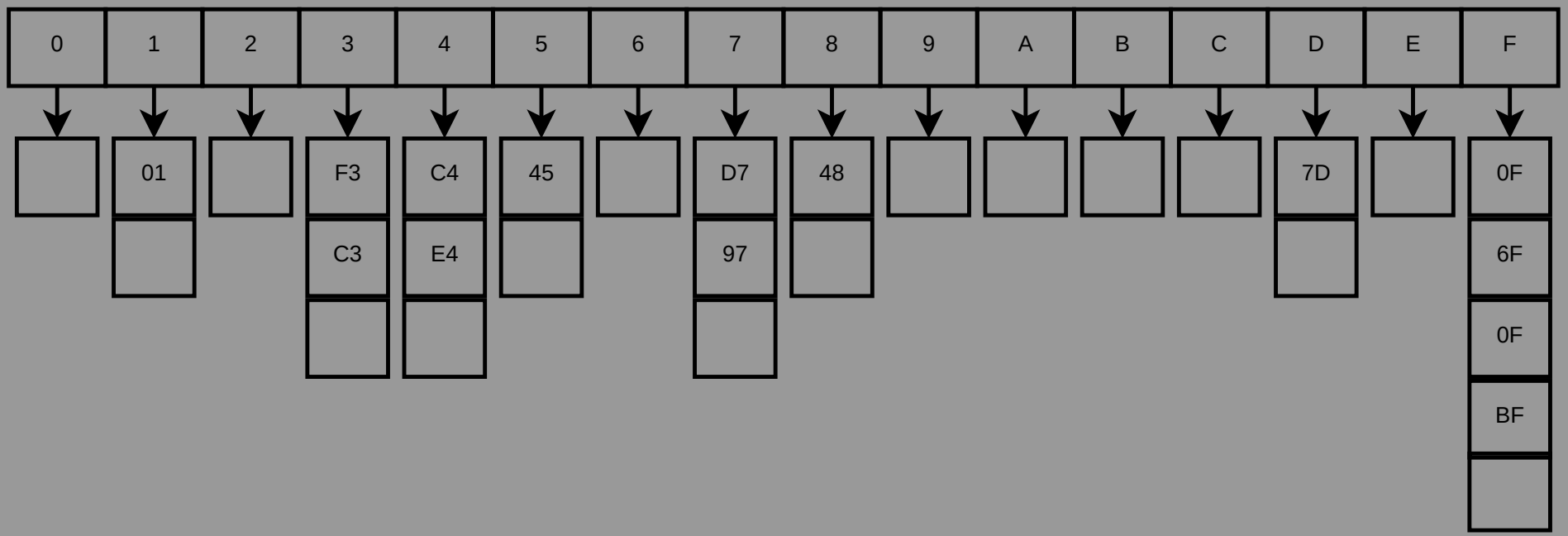
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F



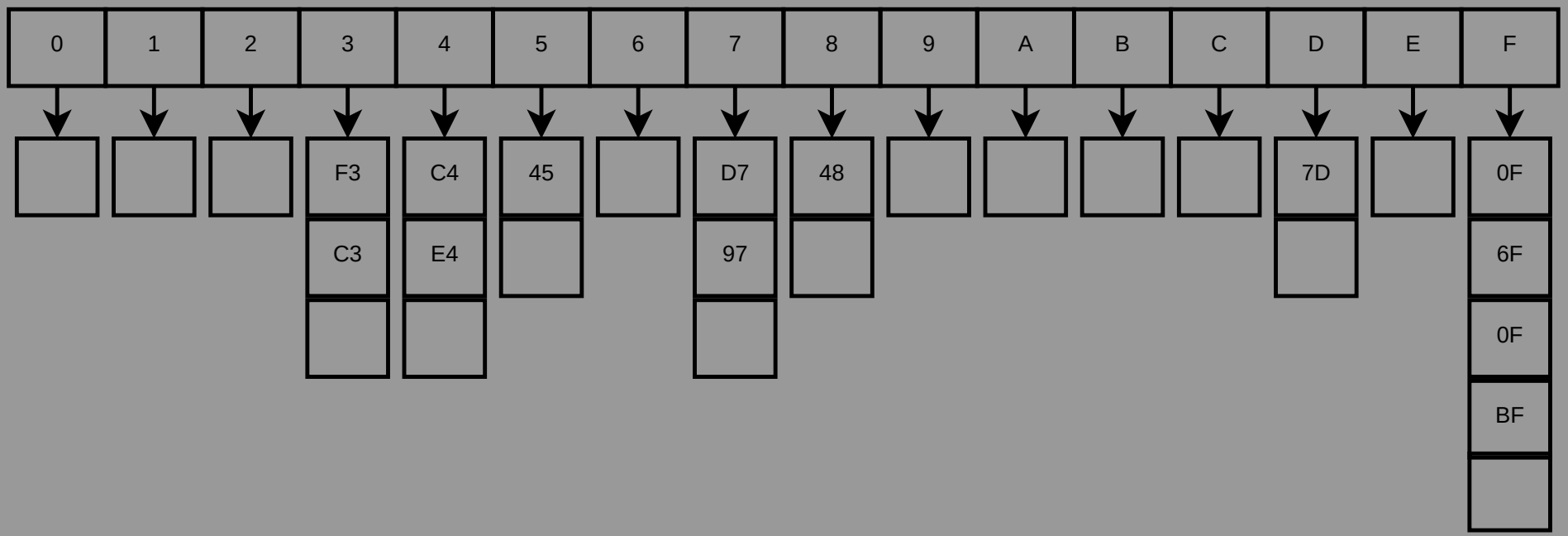
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F



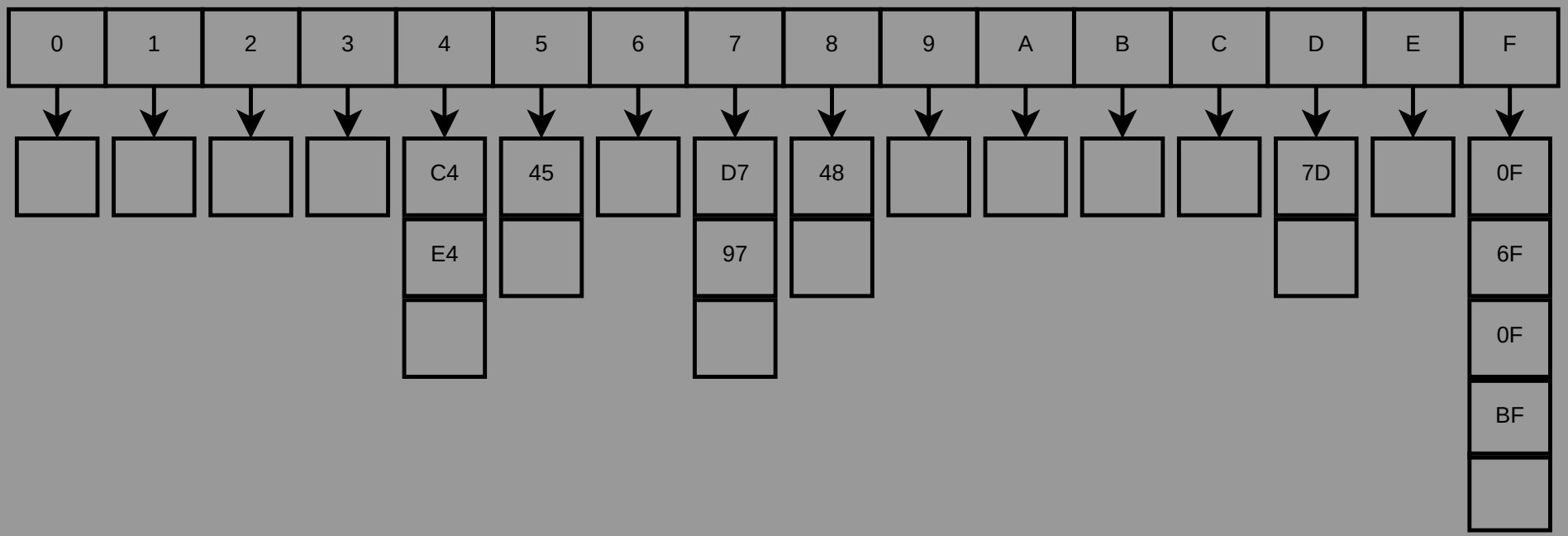
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F



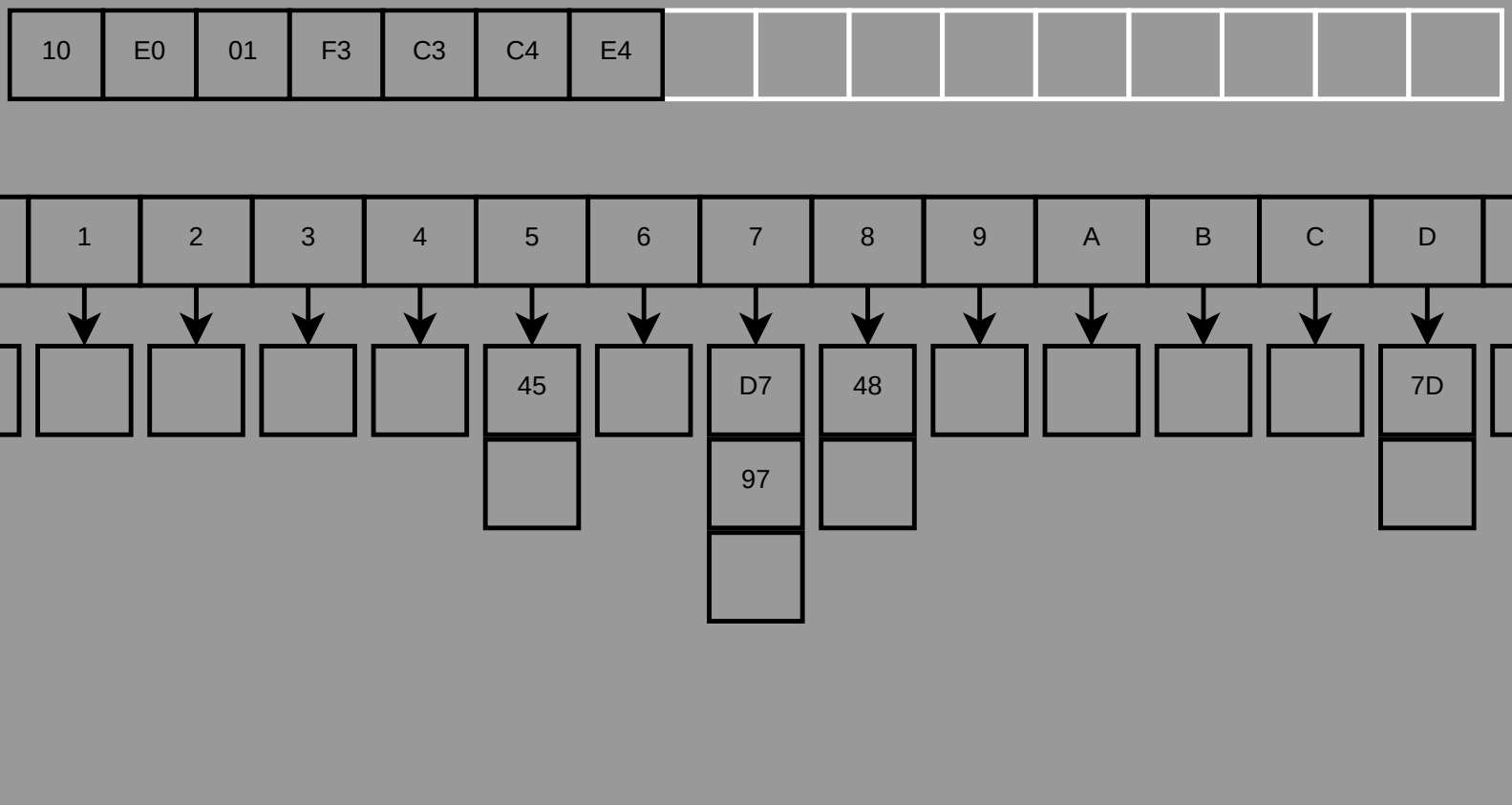
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F



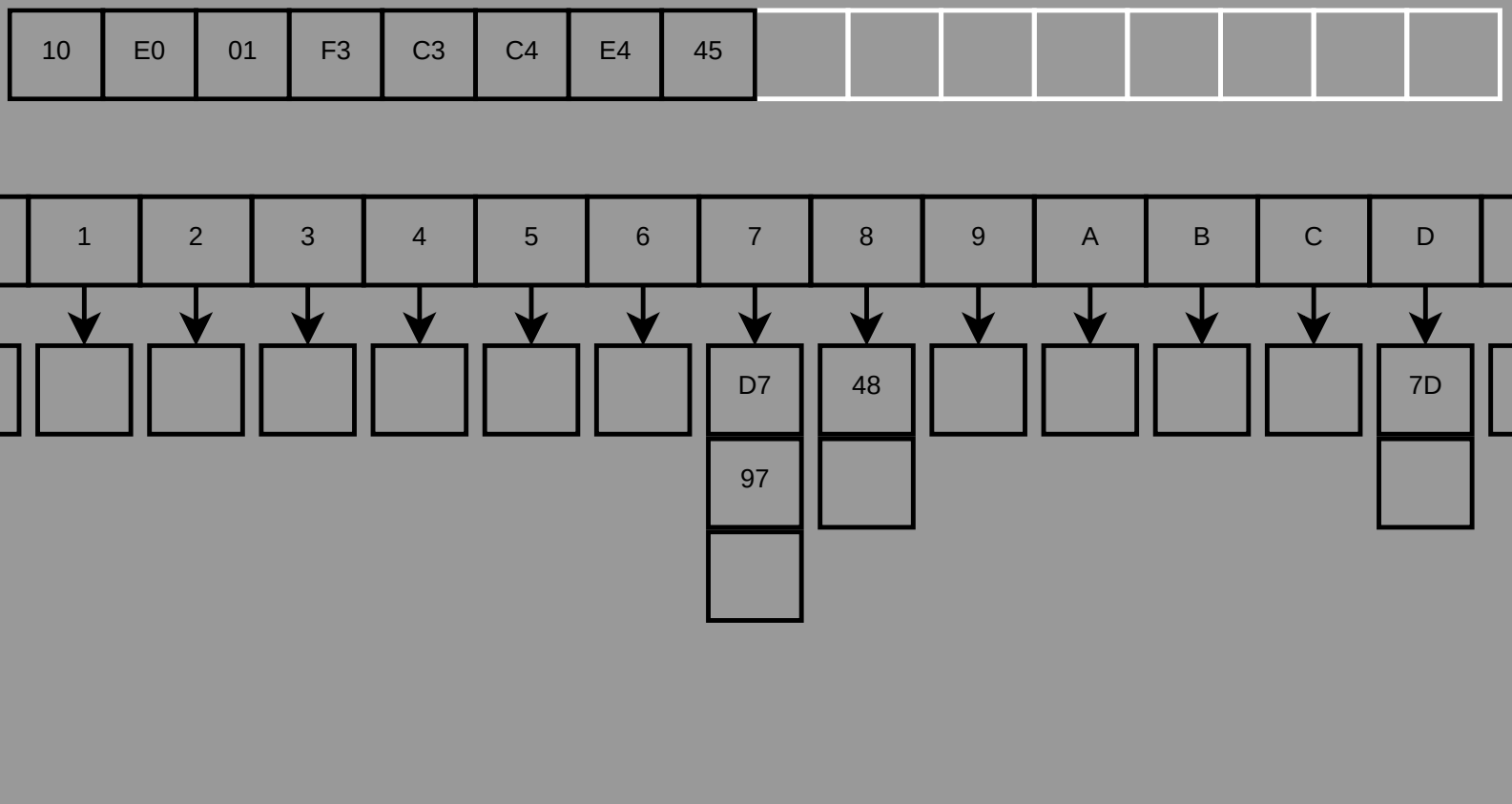
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F



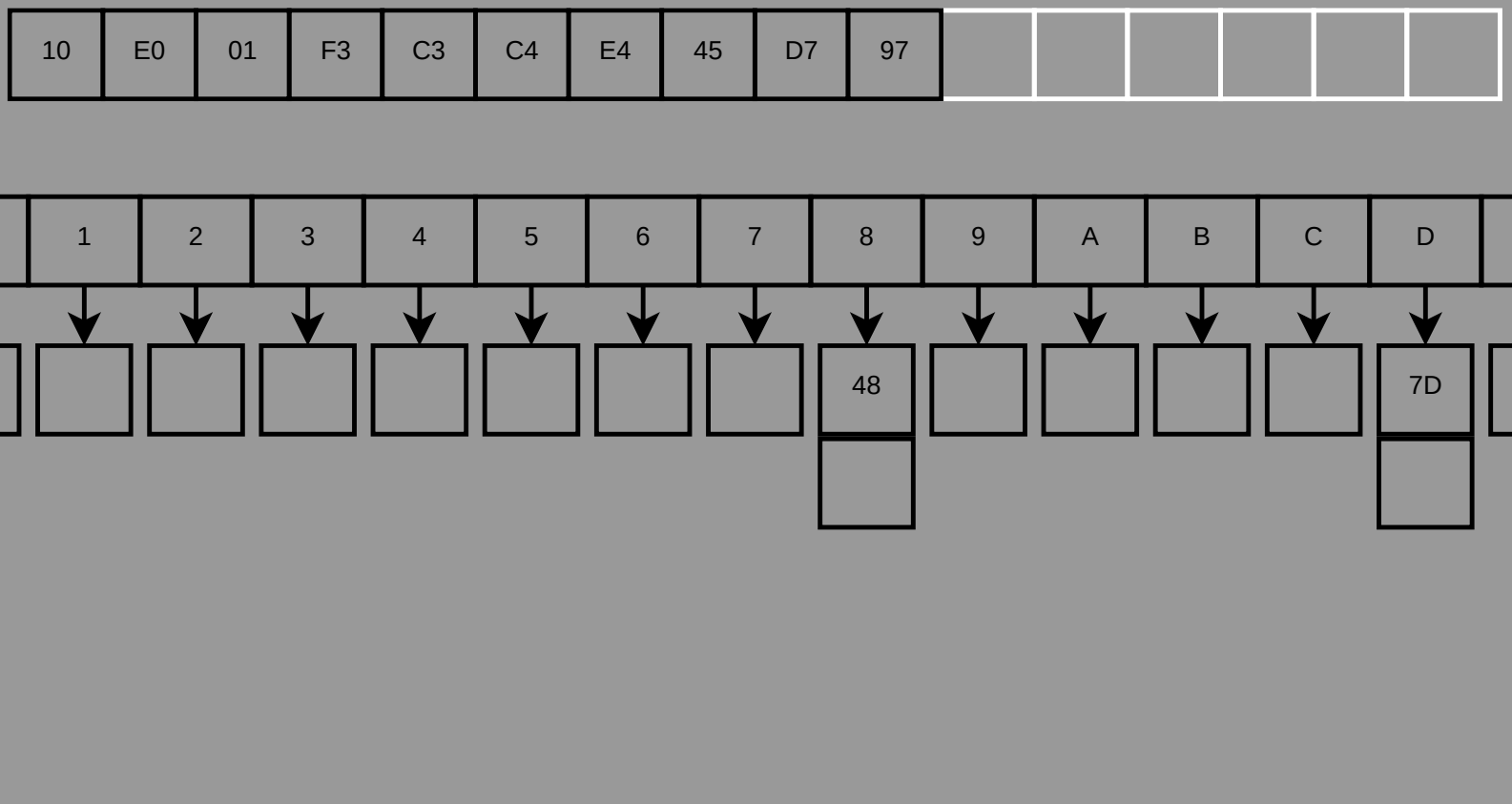
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F



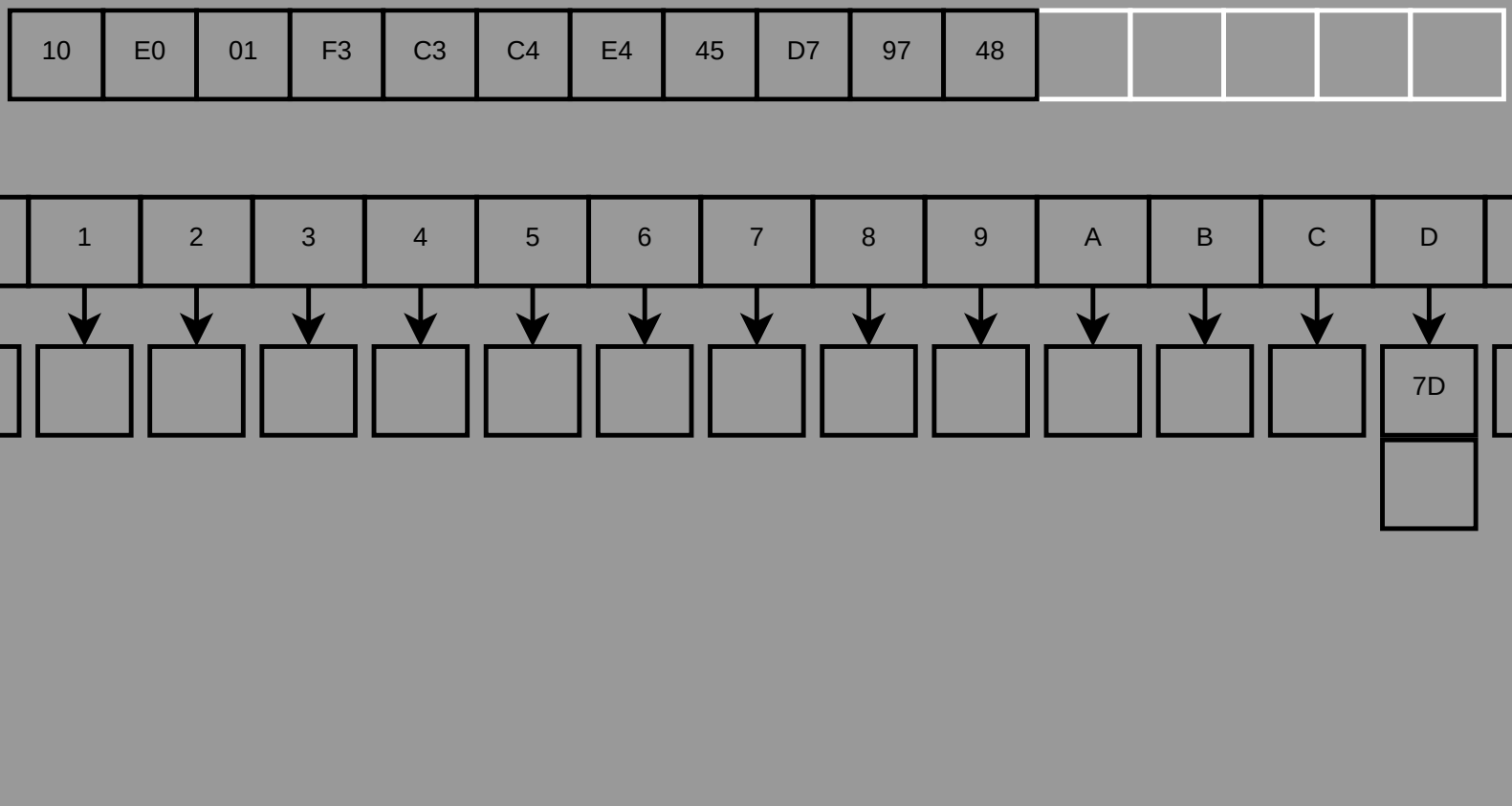
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F



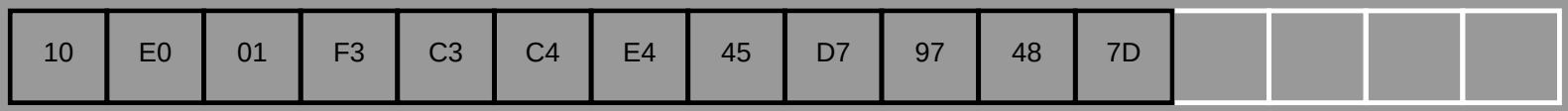
Radix Sort Example

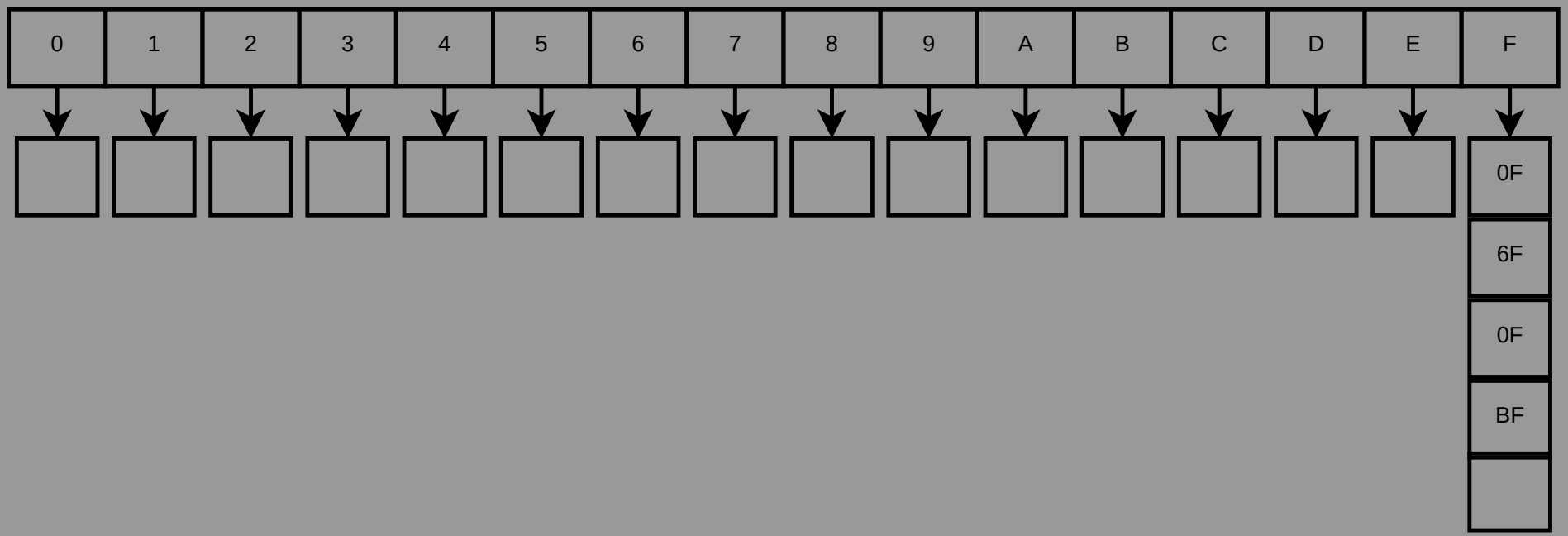
- Here we are using the radix 16 e.g. 0, 1, ..., D, F



Radix Sort Example

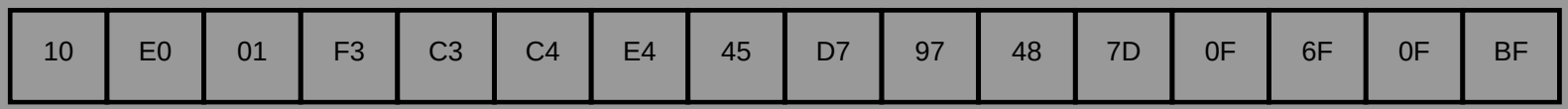
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)

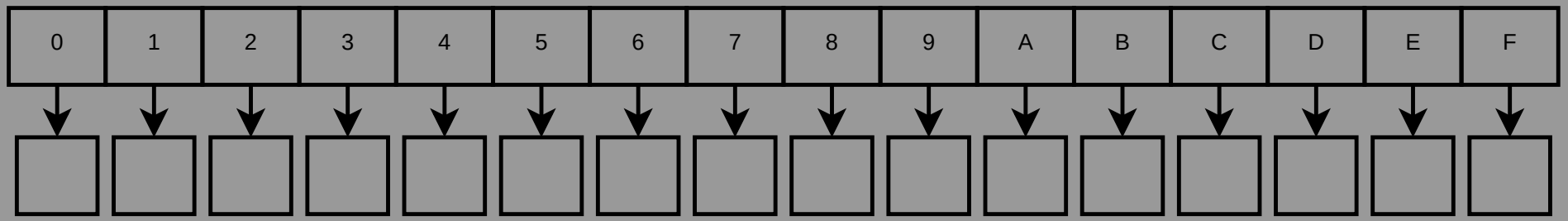




Radix Sort Example

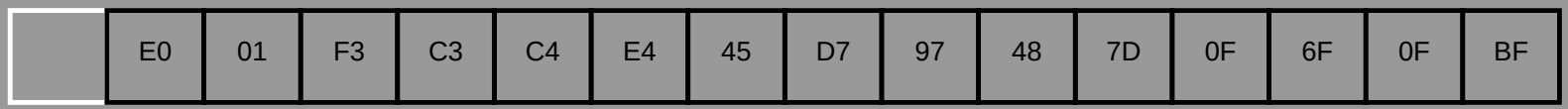
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)

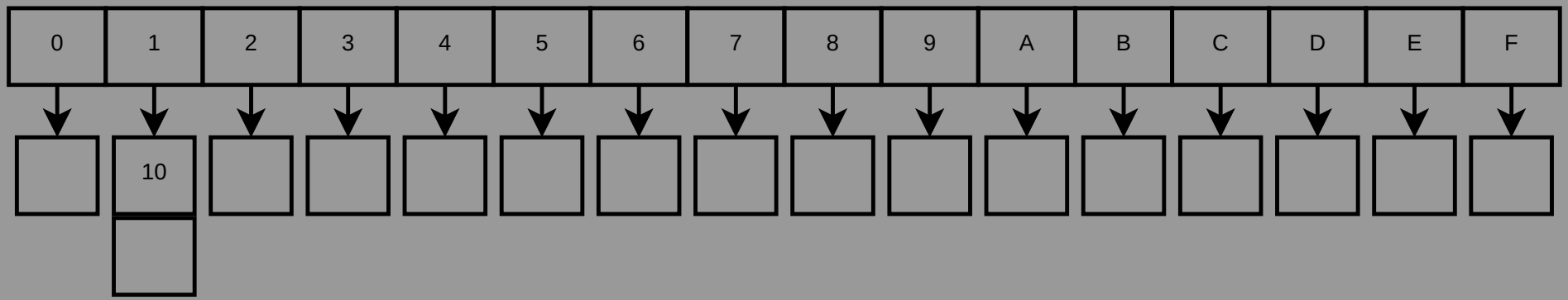




Radix Sort Example

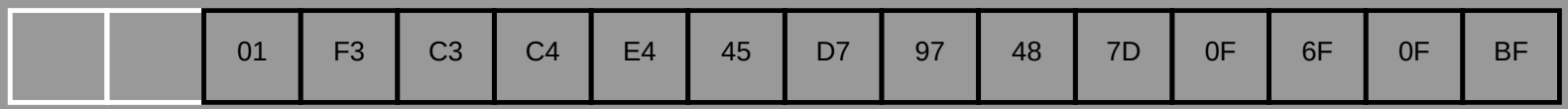
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)

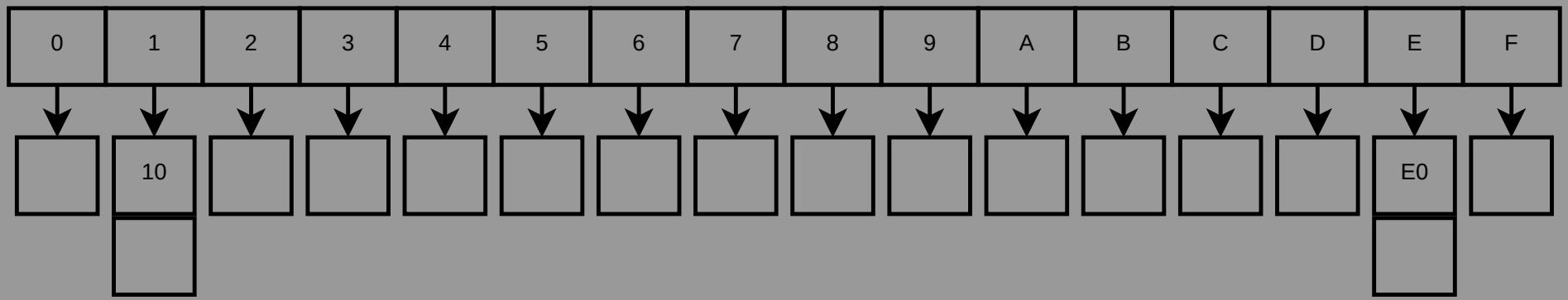




Radix Sort Example

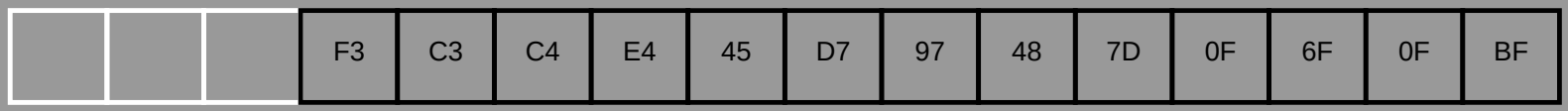
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)

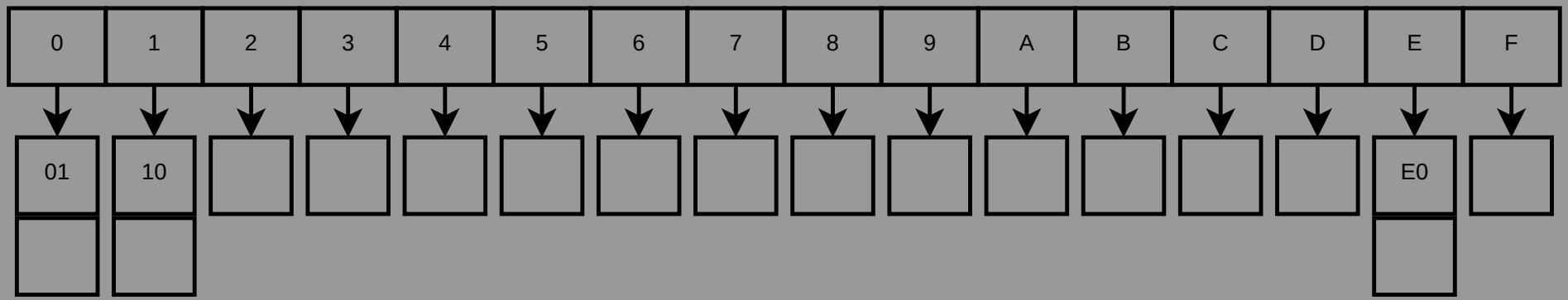




Radix Sort Example

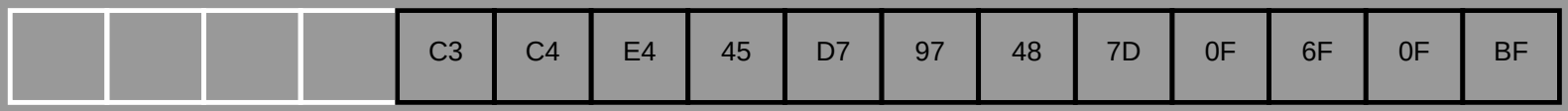
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)

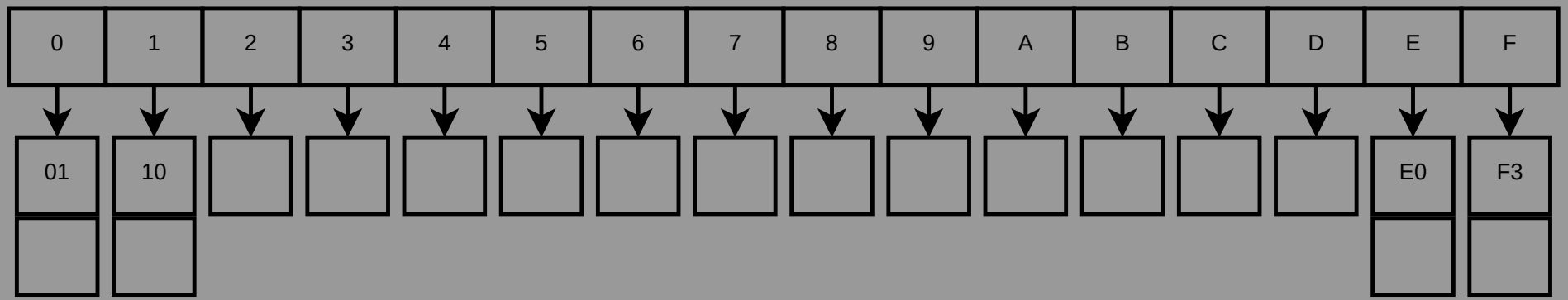




Radix Sort Example

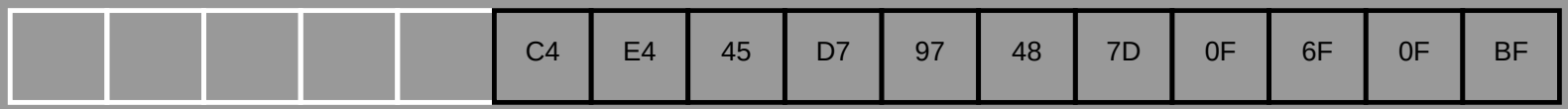
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)

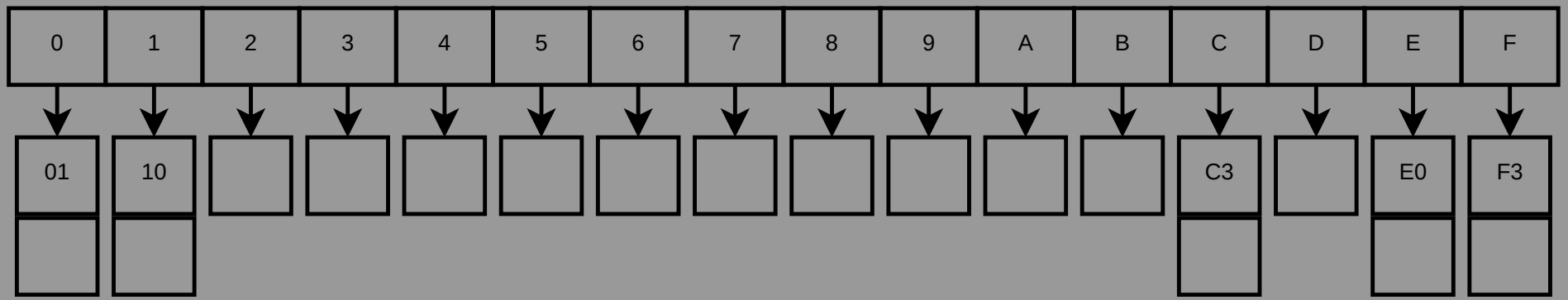




Radix Sort Example

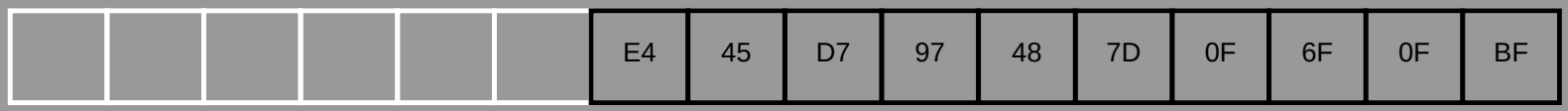
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)

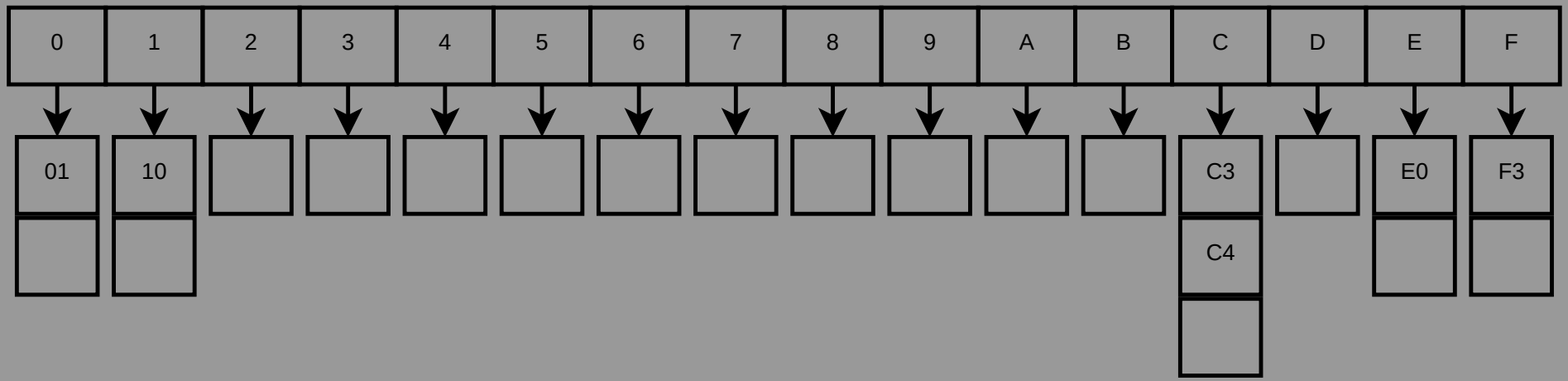




Radix Sort Example

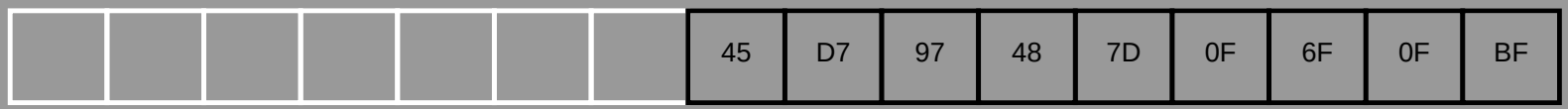
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)

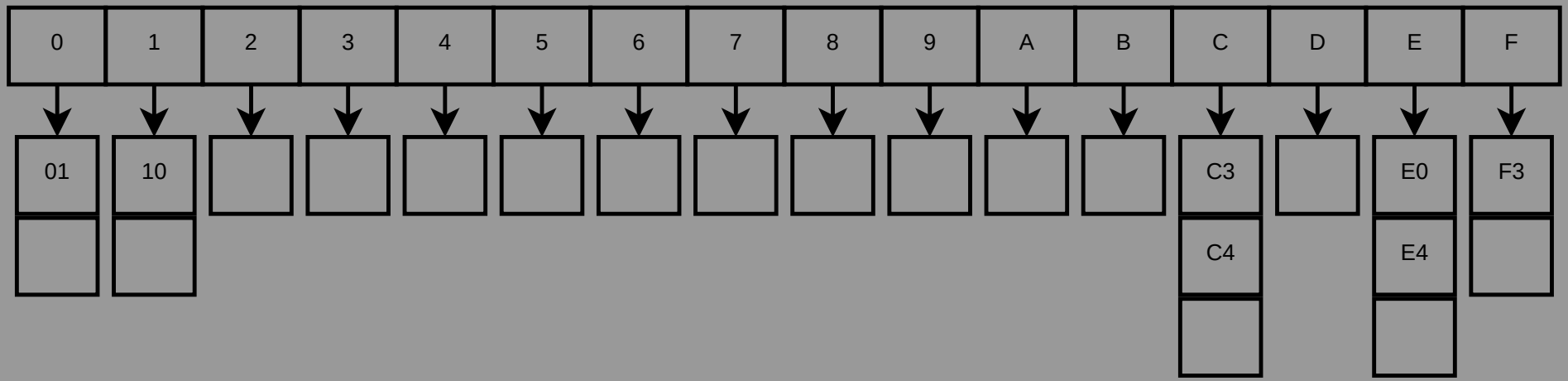




Radix Sort Example

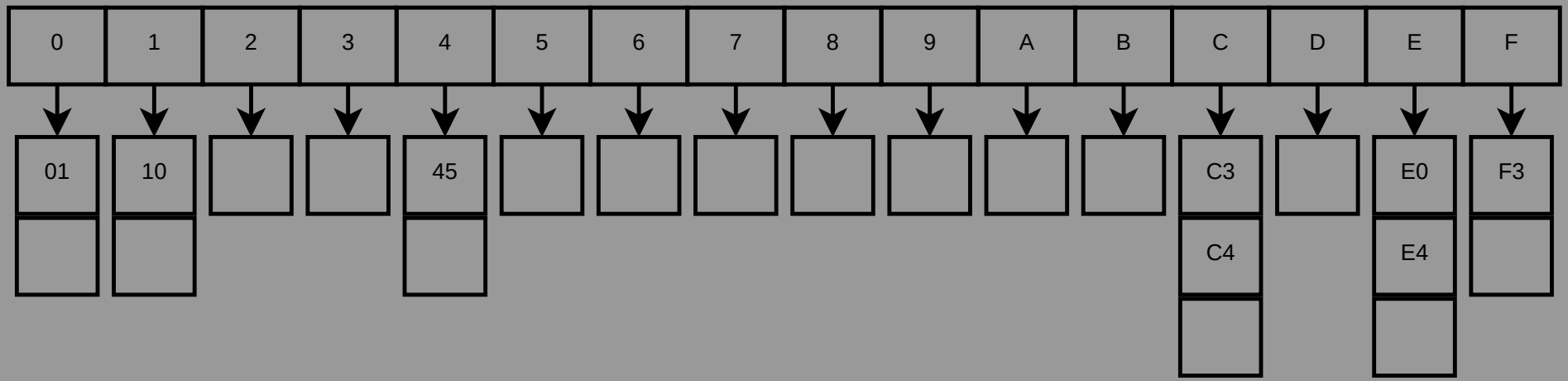
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)





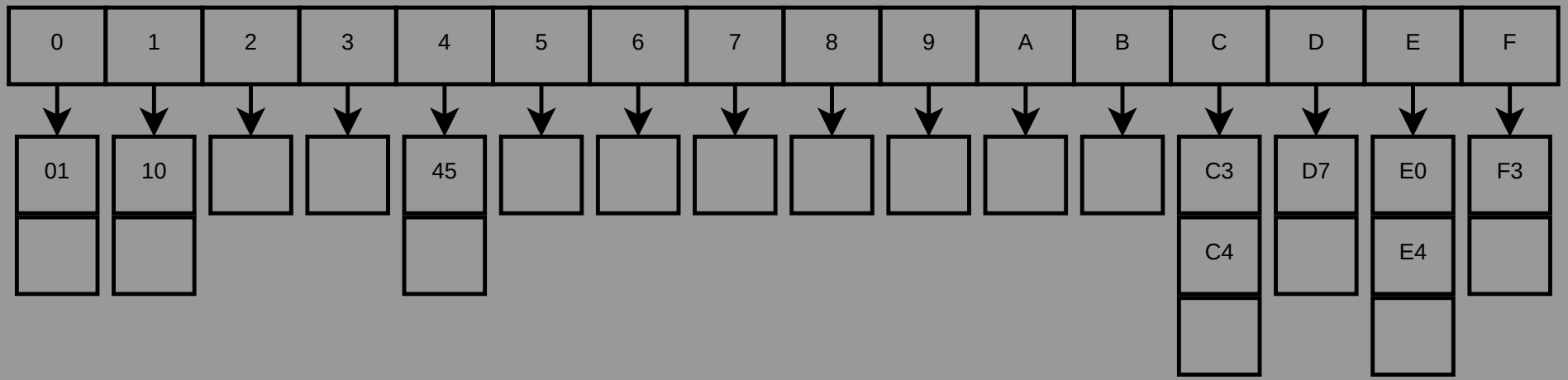
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)



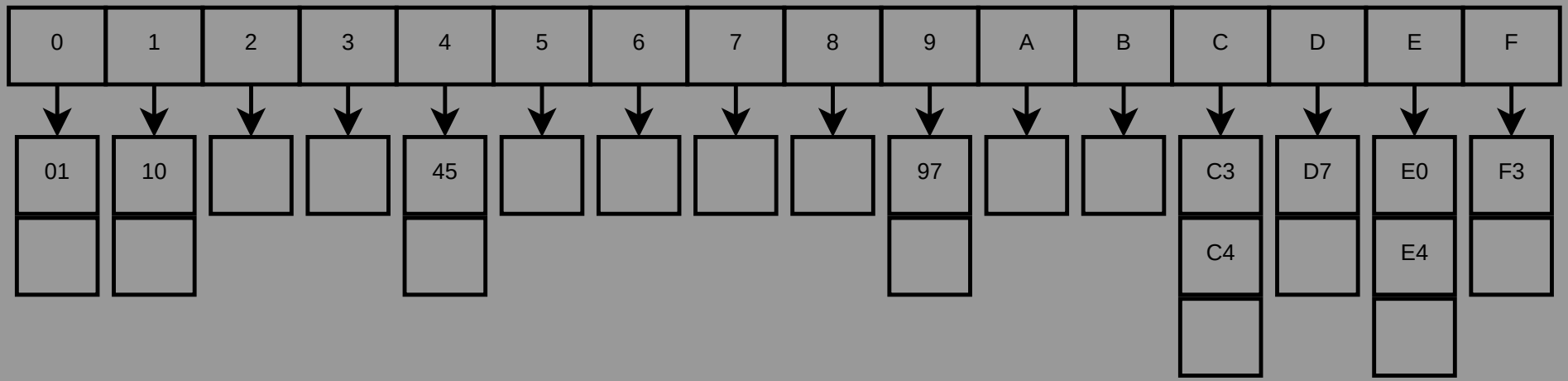
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)



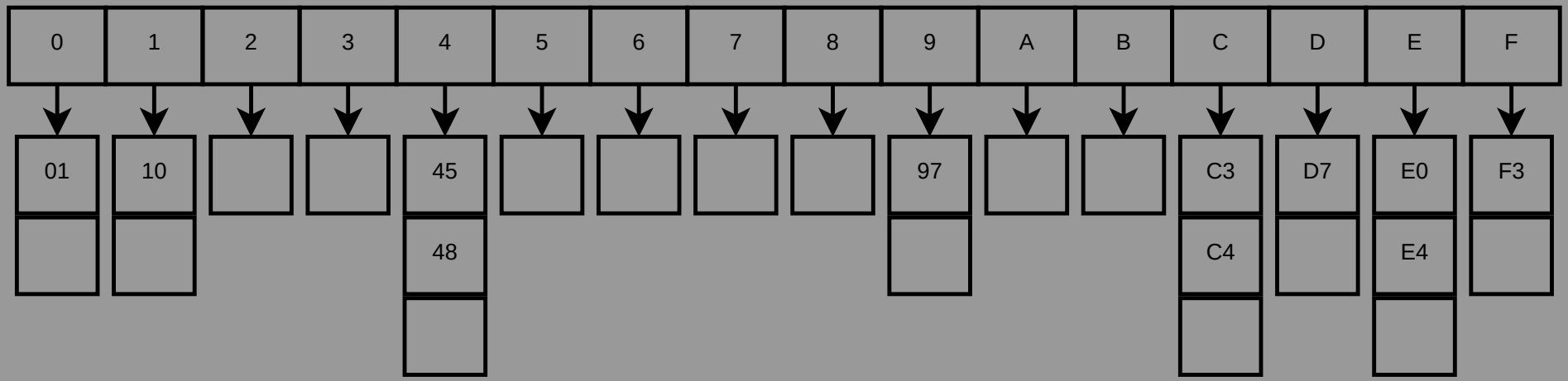
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)



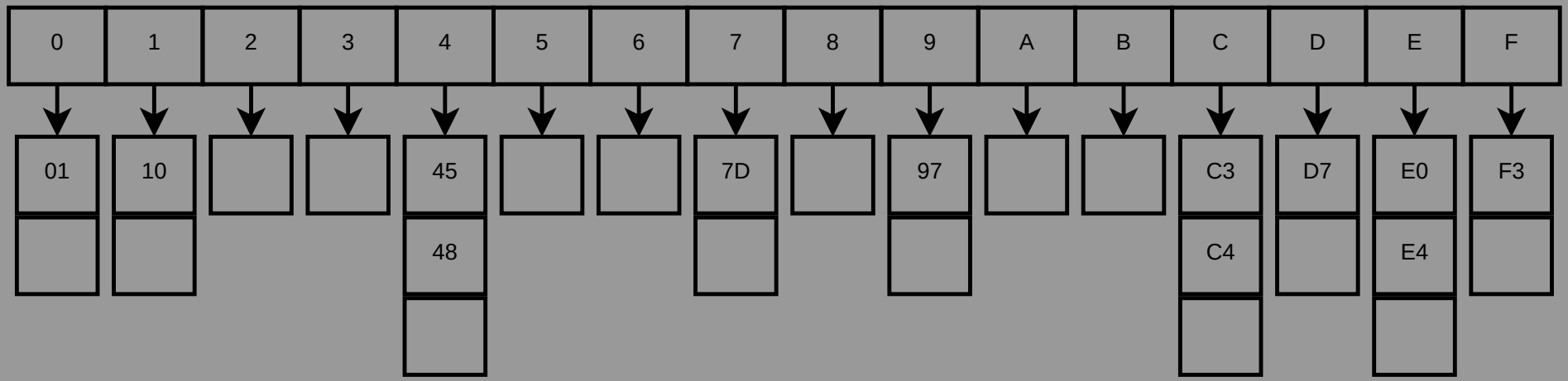
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)



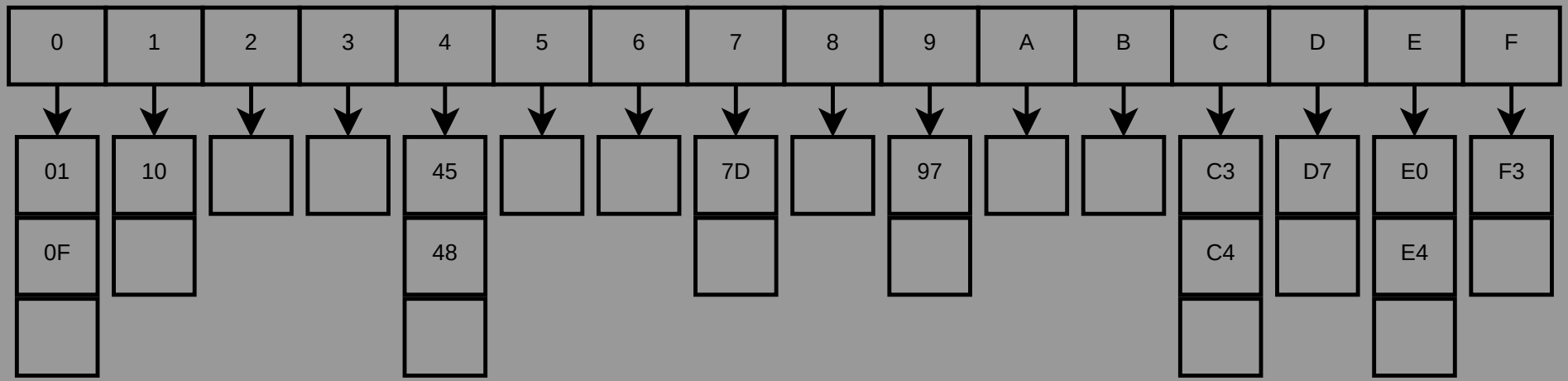
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)



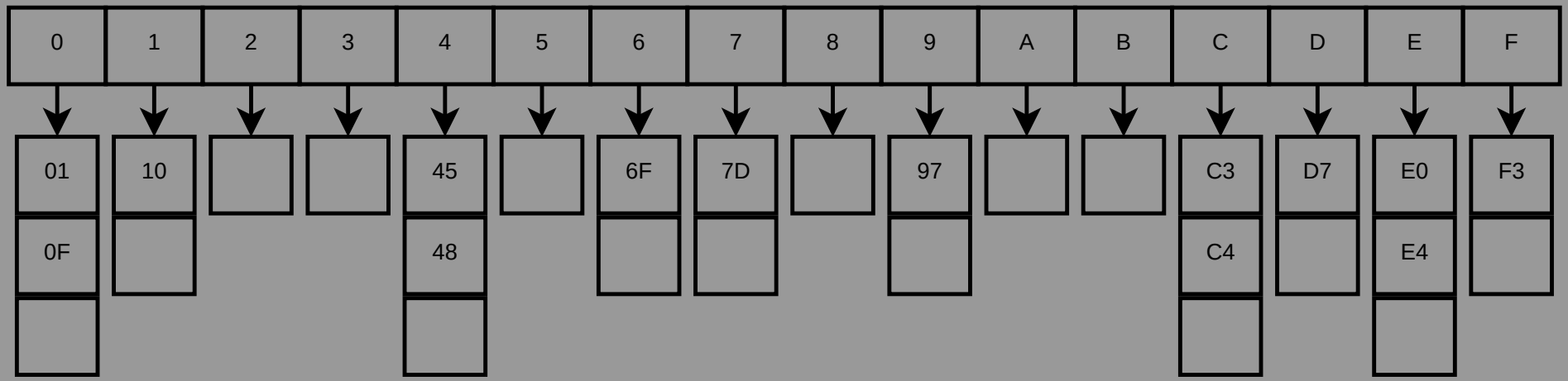
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)



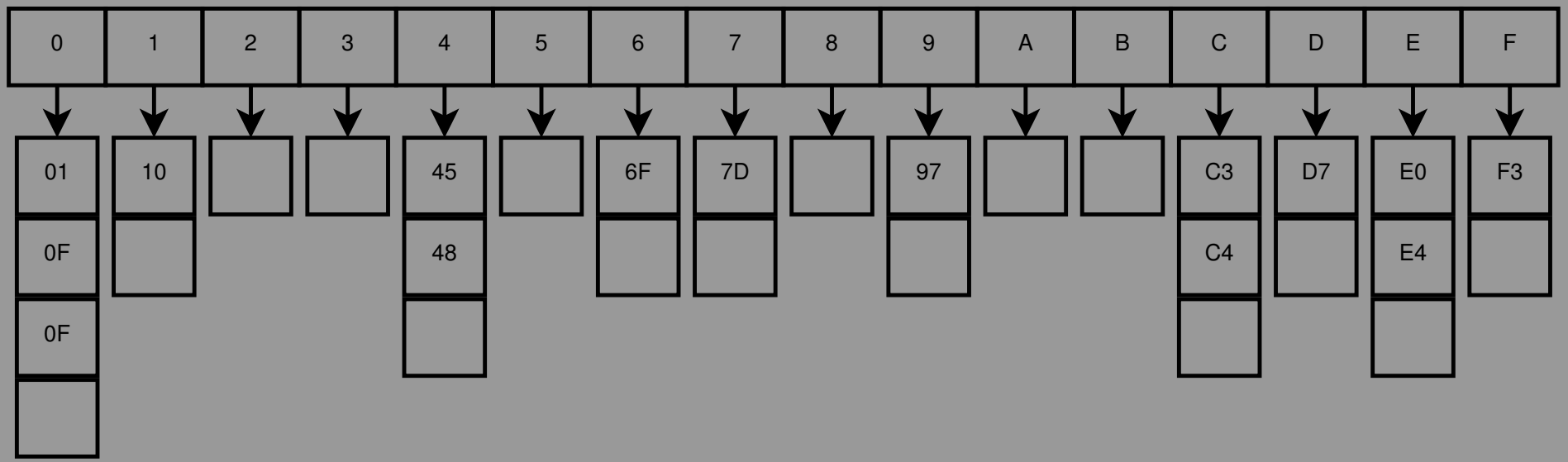
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)



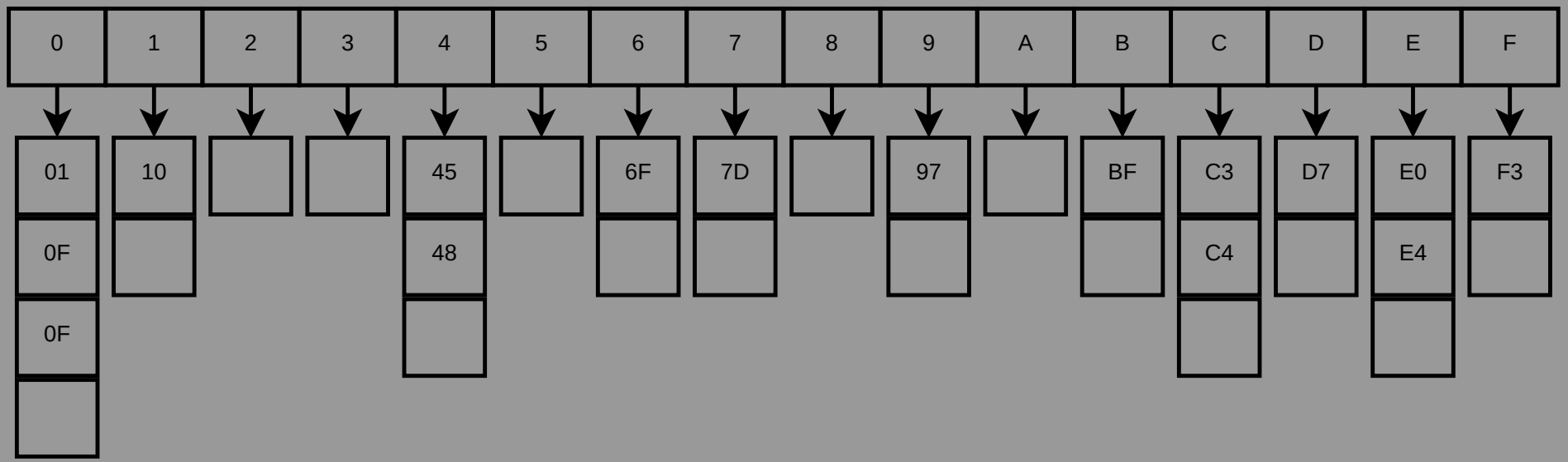
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)



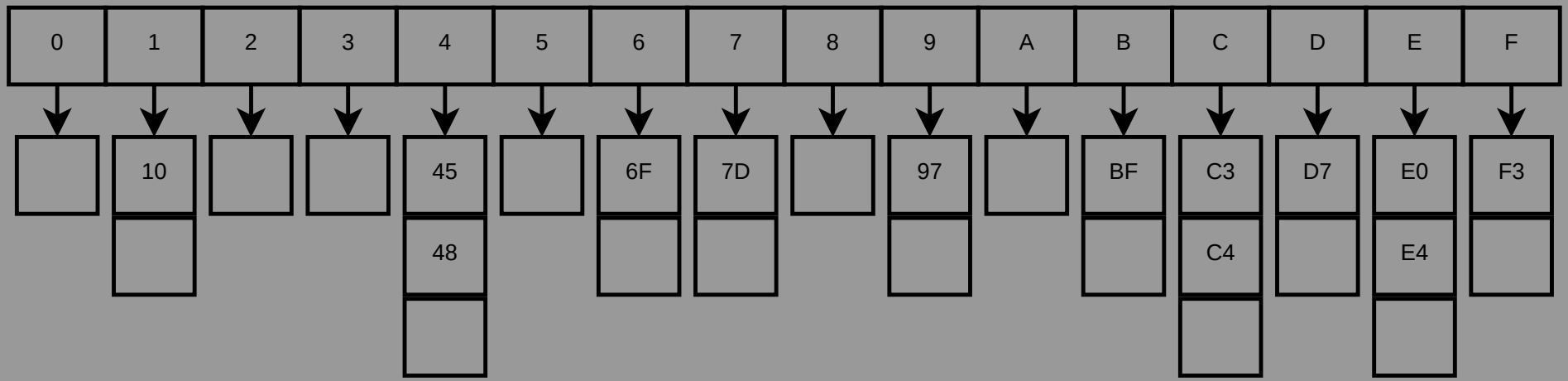
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)



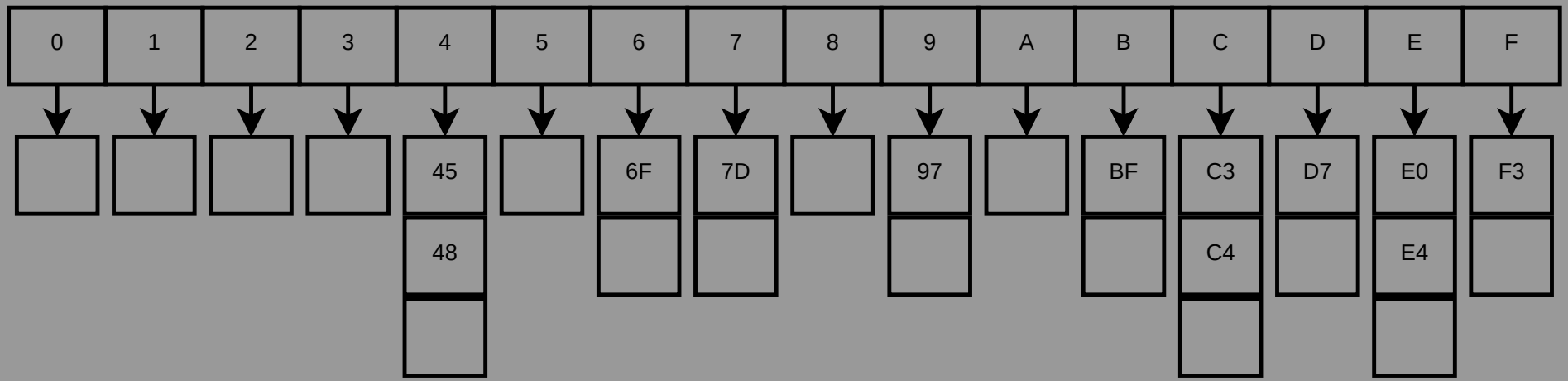
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)



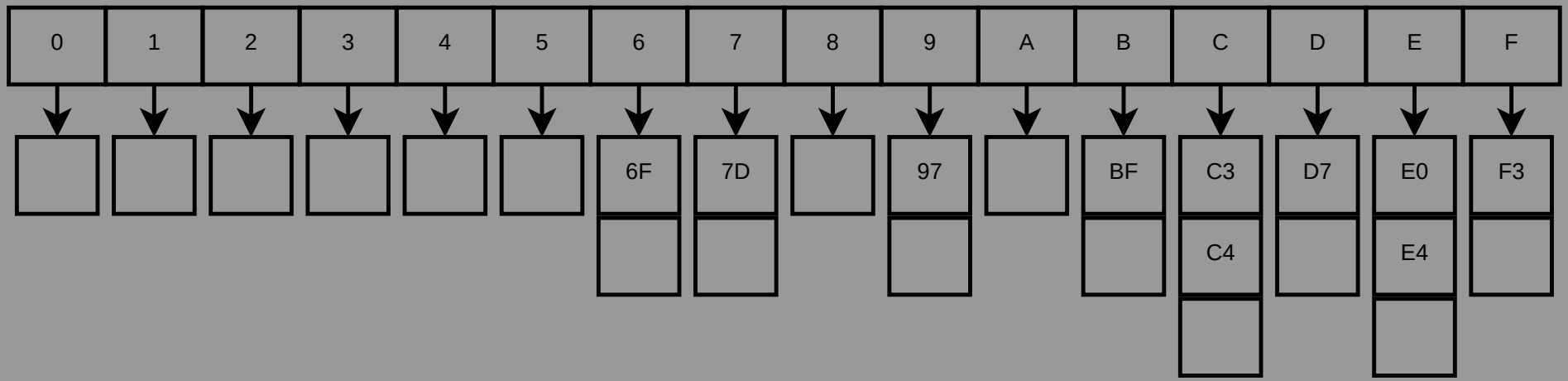
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)



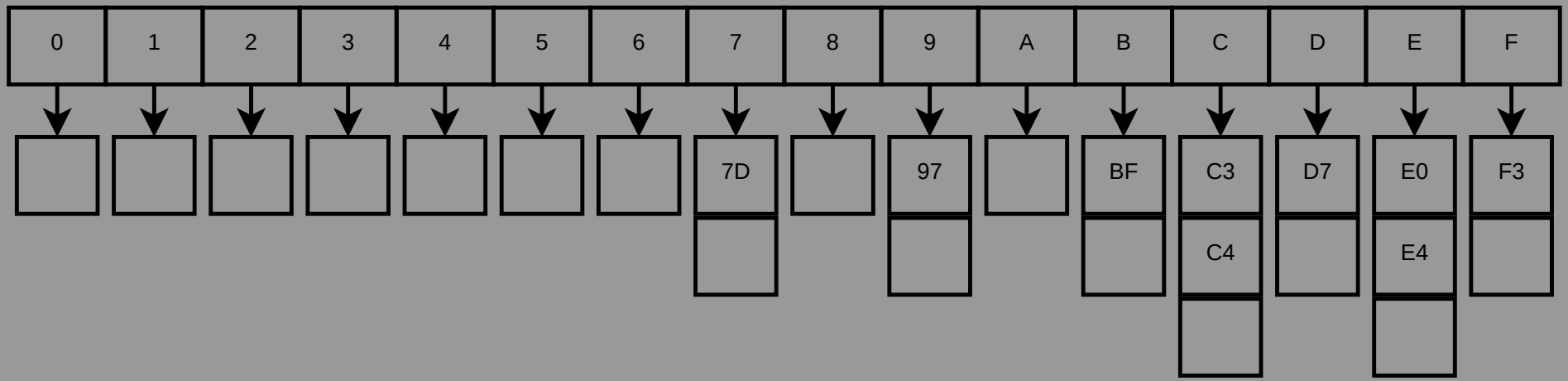
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)



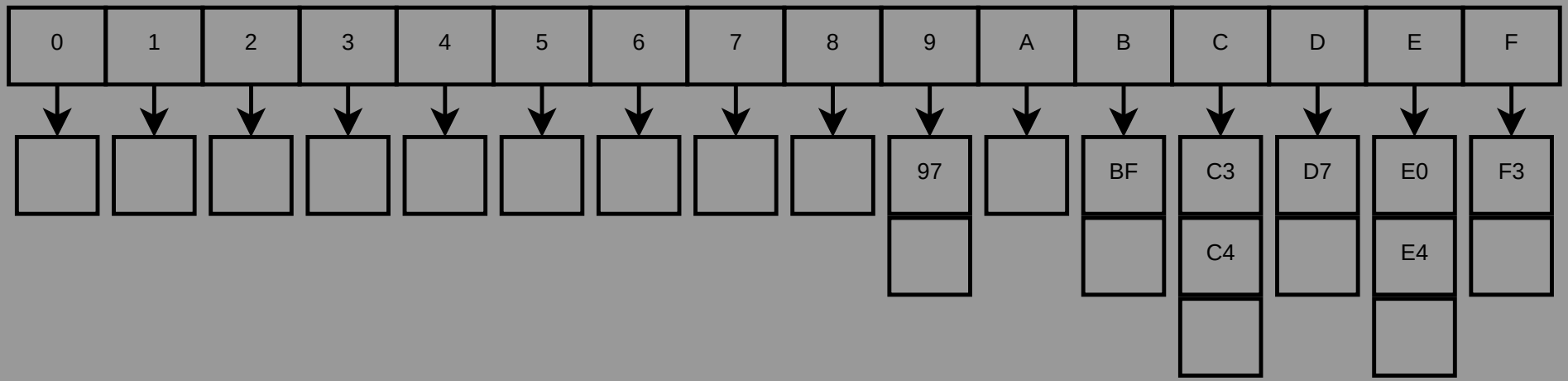
Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)



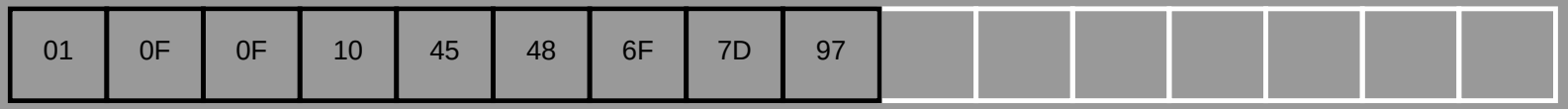
Radix Sort Example

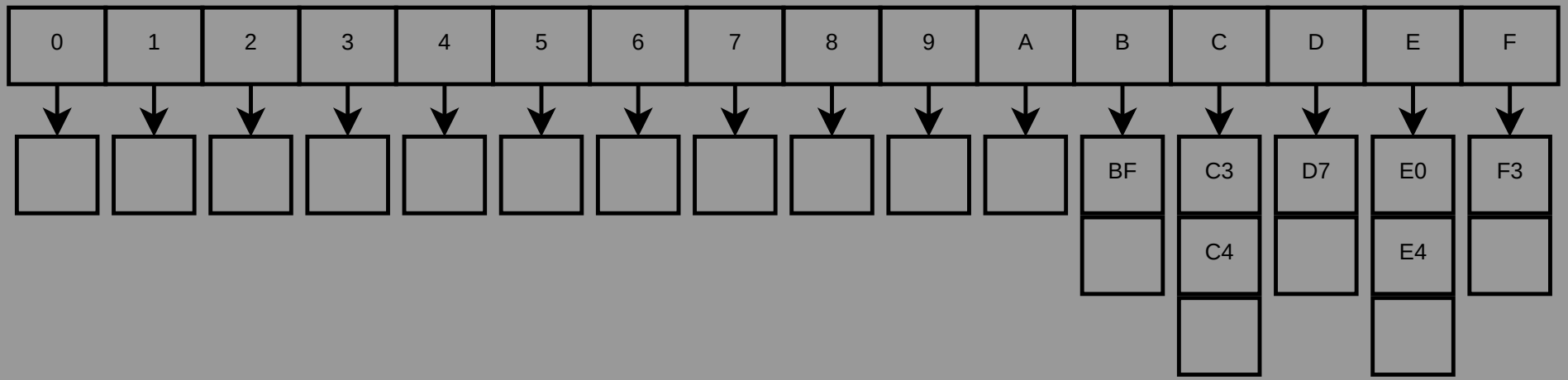
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)



Radix Sort Example

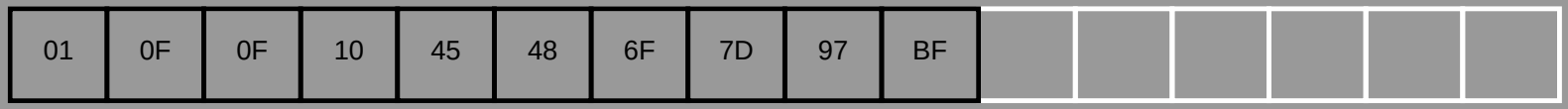
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)

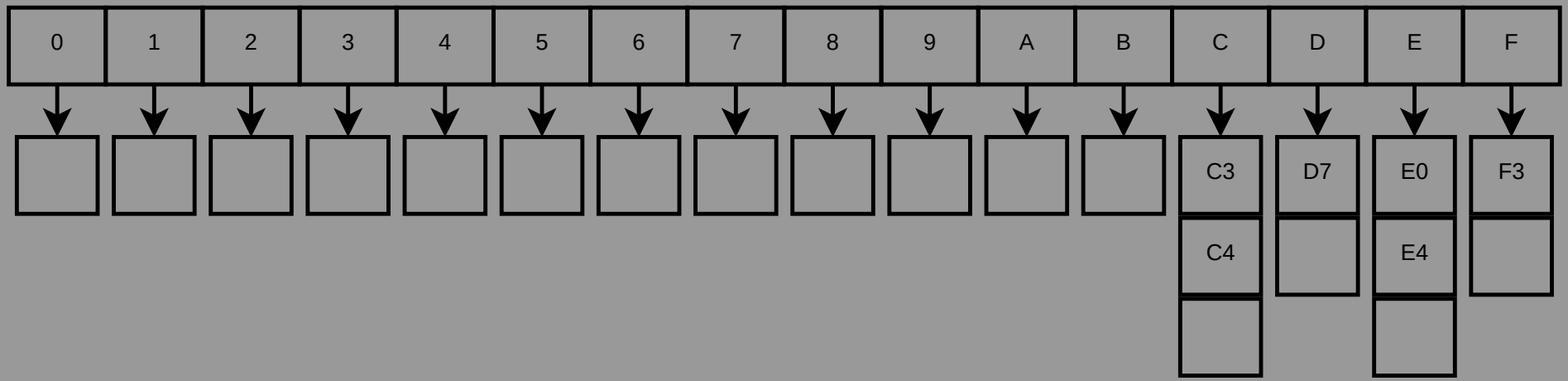




Radix Sort Example

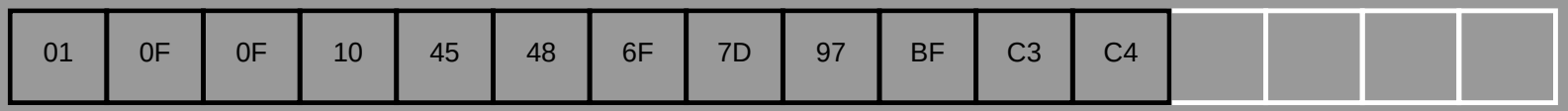
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)

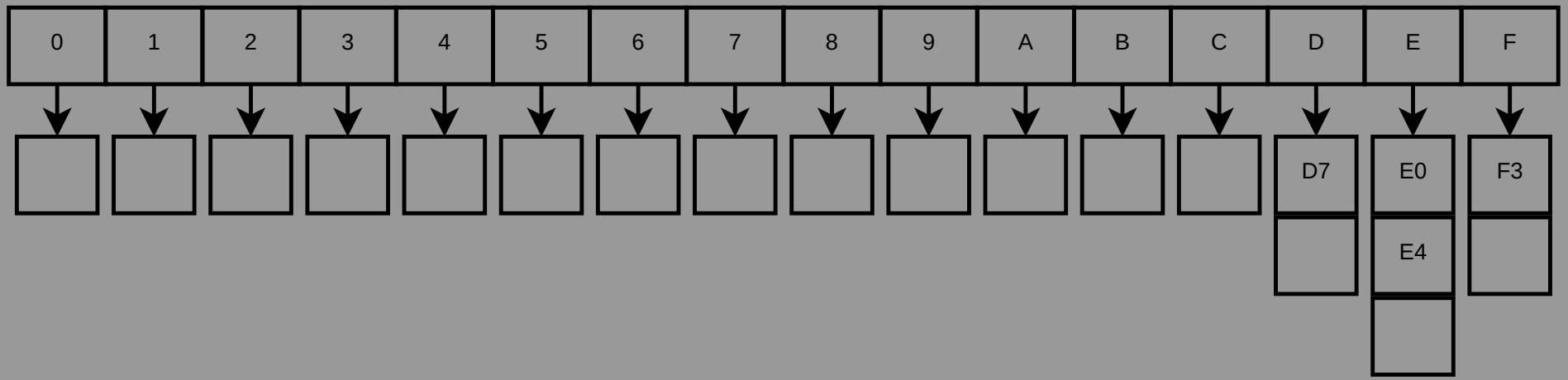




Radix Sort Example

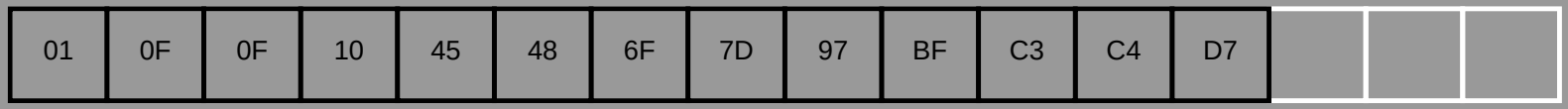
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)

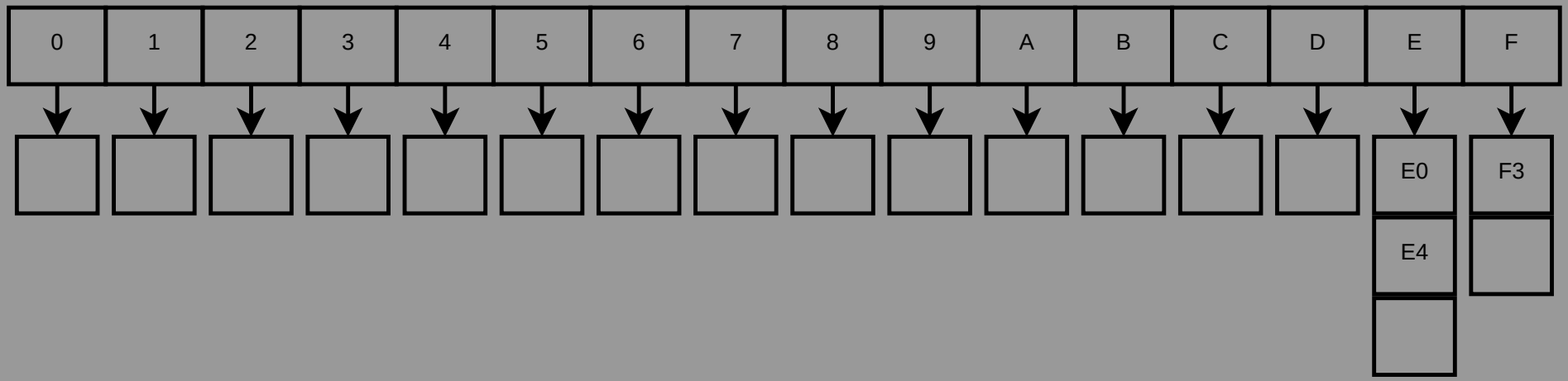




Radix Sort Example

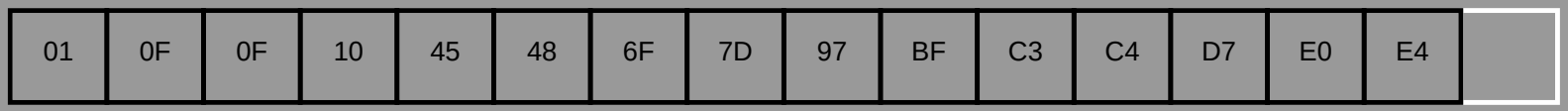
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)

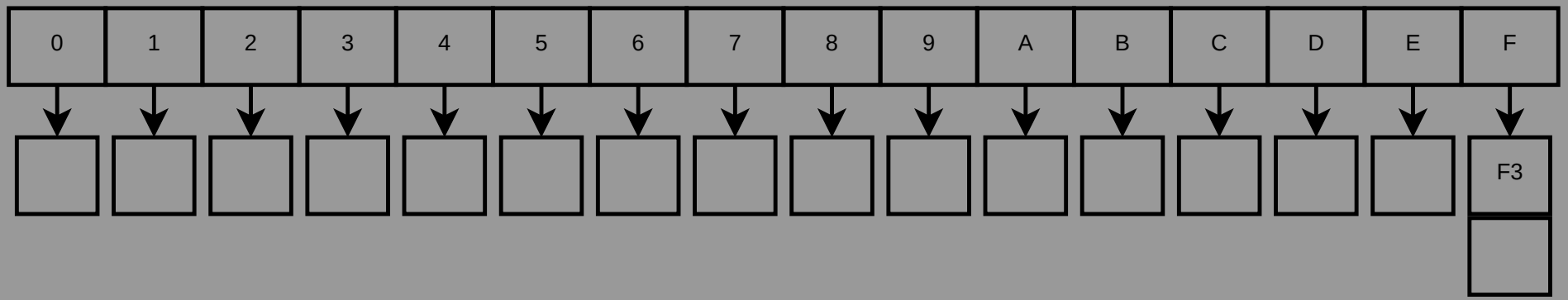




Radix Sort Example

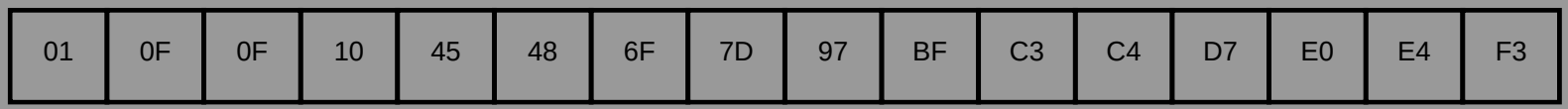
- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)

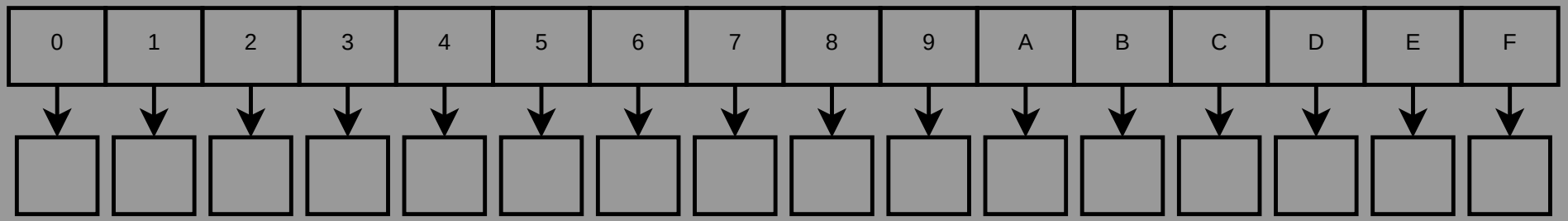




Radix Sort Example

- Here we are using the radix 16 e.g. 0, 1, ..., D, F
  - In this pass we only look at the most significant digit (leftmost)





Complexity of Radix Sort

In order to calculate the complexity of radix sort, we need to understand how many times the bucket sort algorithm will be applied

- This is related to the number of digits in the largest value
  - We have to perform bucket sort once for each digit
  - This value is $k = \log_b(max(n))$ where b is the base we are using
  - e.g. If the largest number is 50015 and we are using base 10 then $\log_{10} 50015 \simeq 5$
- If we consider that bucket sort is $O(n)$ then the complexity of then the complexity of radix sort is $k * O(n)$
  - This means that when keys are short the algorithm is more efficient
  - Typically for 32 bit integers, the radix 256 would be used, meaning only 4 passes have to be done

Table of Contents

Operations in Radix Sort

- Radix sort required breaking all of the keys into smaller components (like a single digit)
- Performing these calculations using the division and modulus operators would be very slow
- This can be made much more efficient if we use a radix that is a power of 2
- This means that we can use bitwise operators to perform the calculations
  - Bitwise operators are very efficient

Bitwise Operators

There are a number of operators in Java that allow you to alter the individual bits inside integers

- Bit shifting is where we move the bits in the number to the left or the right

  $>> n$ This shifts the pattern of bits towards the right by n places, the least significant bits are removed e.g. 101010 >> 2 would be 1010

  $<< n$ This shifts the pattern of bits towards the left by n places, the most significant bits are removed and zeroes are added at the least significant places e.g. 101010 << 2 would be 10101000

- Logical operators can be applied between two number

  $a\&b$ This is the AND operator, the result is the pattern where the same position was 1 in both a and b e.g. 111001&101010 would be 101000

  $a|b$ This is the OR operator, the result is the pattern where the position is 1 if either a or b are 1 at that position e.g. 111001|101010 would be 111011

  $a\hat{}b$ This is the Exclusive OR operator, the result is the pattern where the position is 1 if either a or b are 1 at that position but not both e.g. 111001|101010 would be 010011

Bitwise operators for radix sort

Now that we know how the individual bitwise operators work, how do we use these to perform radix sort

- If the radix we are using is 16, this means that every single hex digit is represented by 4 binary bits
- To find the final 4 bits of any number we use the AND operator
- If we have a key x, then $x\&1111$ gives us a result that is only the last 4 bits of x
  - This gives us the number of the bucket for the first pass
- To get the number of the bucket for the second pass we need to use the bit shift operator
- If we have a key x, then $x >> 4$ moves the second group of 4 binary digits to the bottom, we can now use the AND again to get only those digits

Further Information and Review

If you wish to review the materials covered in this lecture or get further information, read the following sections in Data Structures and Algorithms textbook.

- 12.1 - Merge-Sort
- 12.2 - Quick-Sort
- 12.3 - Studying Sorting through and Algorithmic Lens
- 5 - Recursion