Bubble Sort

- Bubble sort is the easiest sorting algorithm to implement.
- It is an **in-place sorting** algorithm.
- It uses no auxiliary data structures (extra space) while sorting.
- Bubble sort uses multiple passes (scans) through an array.
  - In each pass, compares the adjacent elements of the array
  - If they are in wrong order, swaps the two elements

Bubble Sort

5	**1**	**3**	**4**	**6**	**2**

- Comparison
- Data Movement
- Sorted

Bubble Sort

5	1	3	4	6	2

Comparison

Data Movement

Sorted

Bubble Sort

5	1	3	4	2	6

Comparison

Data Movement

Sorted

Bubble Sort

5	1	3	4	2	6

Comparison

Data Movement

Sorted

Bubble Sort

5	1	3	2	4	6

Comparison

Data Movement

Sorted

Bubble Sort

5	1	3	2	4	6

Comparison

Data Movement

Sorted

Bubble Sort

5	1	2	3	4	6

Comparison

Data Movement

Sorted

Bubble Sort

5	1	2	3	4	6

Comparison

Data Movement

Sorted

Bubble Sort

5	1	2	3	4	6

Comparison

Data Movement

Sorted

Bubble Sort

1	5	2	3	4	6

Comparison

Data Movement

Sorted

Bubble Sort

1	**5**	**2**	**3**	**4**	**6**

- Finally after the first pass, we see that the smallest element 1 reaches its correct position.
    - Can do this in other way around by placing the largest element in its current position

Comparison

Data Movement

Sorted

Bubble Sort

1	5	2	3	4	6

Comparison

Data Movement

Sorted

Bubble Sort

1	5	2	3	4	6

Comparison

Data Movement

Sorted

Bubble Sort

1	5	2	3	4	6

Comparison

Data Movement

Sorted

Bubble Sort

1	5	2	3	4	6

Comparison

Data Movement

Sorted

Bubble Sort

1	2	5	3	4	6

Comparison

Data Movement

Sorted

Bubble Sort

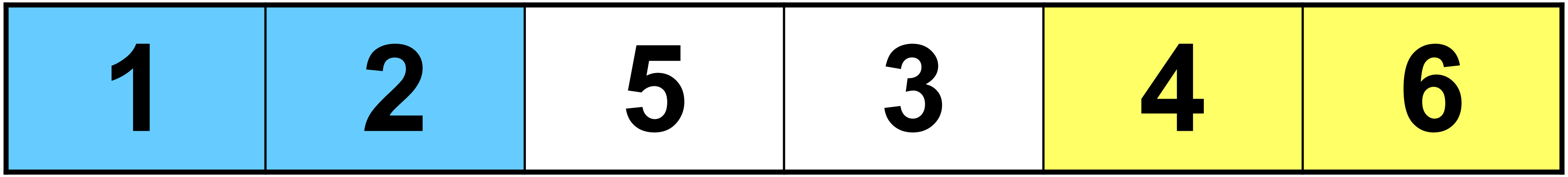
1	2	5	3	4	6

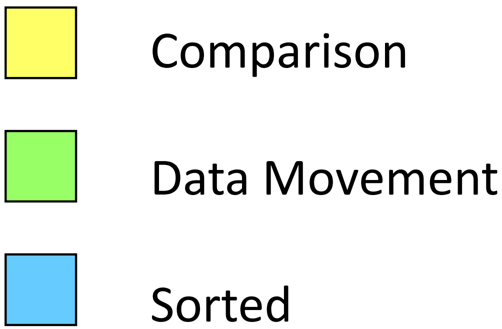
Comparison

Data Movement

Sorted

Bubble Sort





Bubble Sort

1	2	5	3	4	6

Comparison

Data Movement

Sorted

Bubble Sort

1	2	5	3	4	6

Comparison

Data Movement

Sorted

Bubble Sort

1	2	3	5	4	6

Comparison

Data Movement

Sorted

Bubble Sort

1	2	3	5	4	6

Comparison

Data Movement

Sorted

Bubble Sort

1	2	3	5	4	6

Comparison

Data Movement

Sorted

Bubble Sort

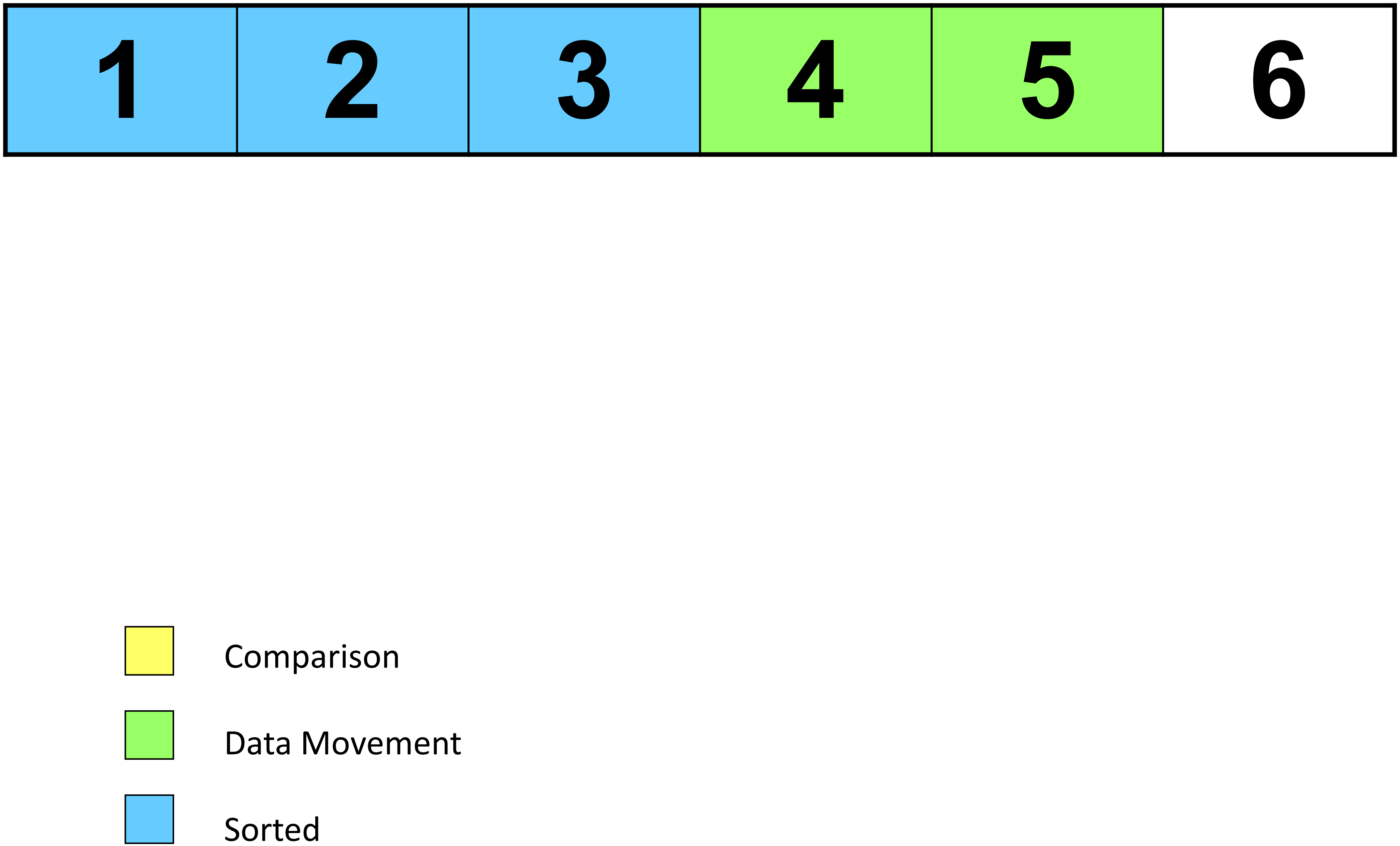
1	2	3	5	4	6

Comparison

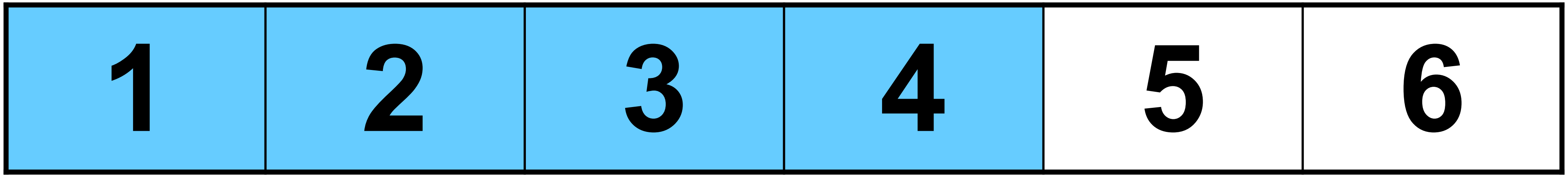
Data Movement

Sorted

Bubble Sort



Bubble Sort



Comparison

Data Movement

Sorted

Bubble Sort

1	2	3	4	5	6

Comparison

Data Movement

Sorted

Bubble Sort

1	2	3	4	5	6

Comparison

Data Movement

Sorted

Bubble Sort

```python
def bubble_sort(A):
    n = len(A)
    for i in range(n - 1):
        for j in range(n - i - 1):
            if A[j] > A[j + 1]:
                A[j], A[j + 1] = A[j + 1], A[j]
    return A


A = [5, 2, 9, 2, 1]
print(bubble_sort(A))  # prints [1, 2, 2, 5, 9]
```

```python
def bubble_sort(A):
    n = len(A)
    while True:
        swapped = False
        for i in range(n - 1):
            if A[i] > A[i + 1]:
                A[i], A[i + 1] = A[i + 1], A[i]
                swapped = True
        if not swapped:
            break
    return A
```

- To avoid extra comparisons, we maintain a flag (**swapped**) variable.

- Once we need to swap adjacent values for correcting their wrong order, the value of flag variable is set to 1.

- If we encounter a pass where flag == 0, then it is safe to break the outer loop and declare the array is sorted.

Bubble Sort _ Complexity analysis

- Bubble sort uses two loops- inner loop and outer loop.
- The inner loop deterministically performs O(n) comparisons.
- In worst case, the outer loop runs O(n) times.

the worst-case time complexity of bubble sort is
$O(n \times n) = O(n^2)$.

- In best case, the array is already sorted but still to check, bubble sort performs O(n) comparisons.
- the best-case time complexity of bubble sort is O(n).

Selection Sort

5	**1**	**3**	**4**	**6**	**2**

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

↑
Current

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

↑
Current

Comparison

Data Movement

Sorted

Selection Sort

5	**1**	**3**	**4**	**6**	**2**

↑
Current

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

↑
Current

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

↑
Current

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

↑
Current

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

↑
Current

Comparison

Data Movement

Sorted

Selection Sort

5	**1**	**3**	**4**	**6**	**2**

↑ **Current** ↑ **Smallest**

Comparison

Data Movement

Sorted

Selection Sort

1	5	3	4	6	2

↑ **Current** ↑ **Smallest**

Comparison

Data Movement

Sorted

Selection Sort

1	5	3	4	6	2

Comparison

Data Movement

Sorted

Selection Sort

1	**5**	**3**	**4**	**6**	**2**

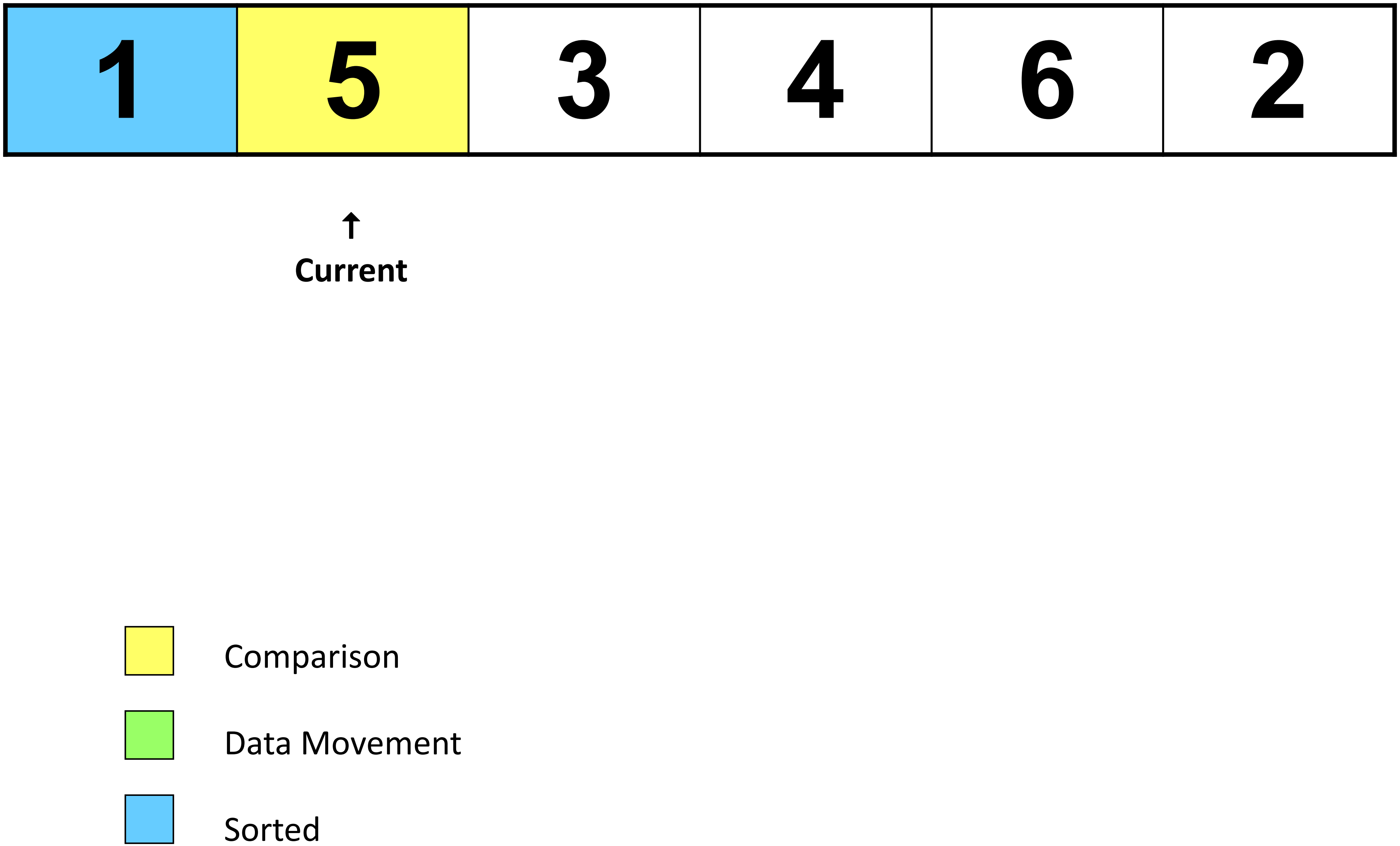
↑
Current

Comparison

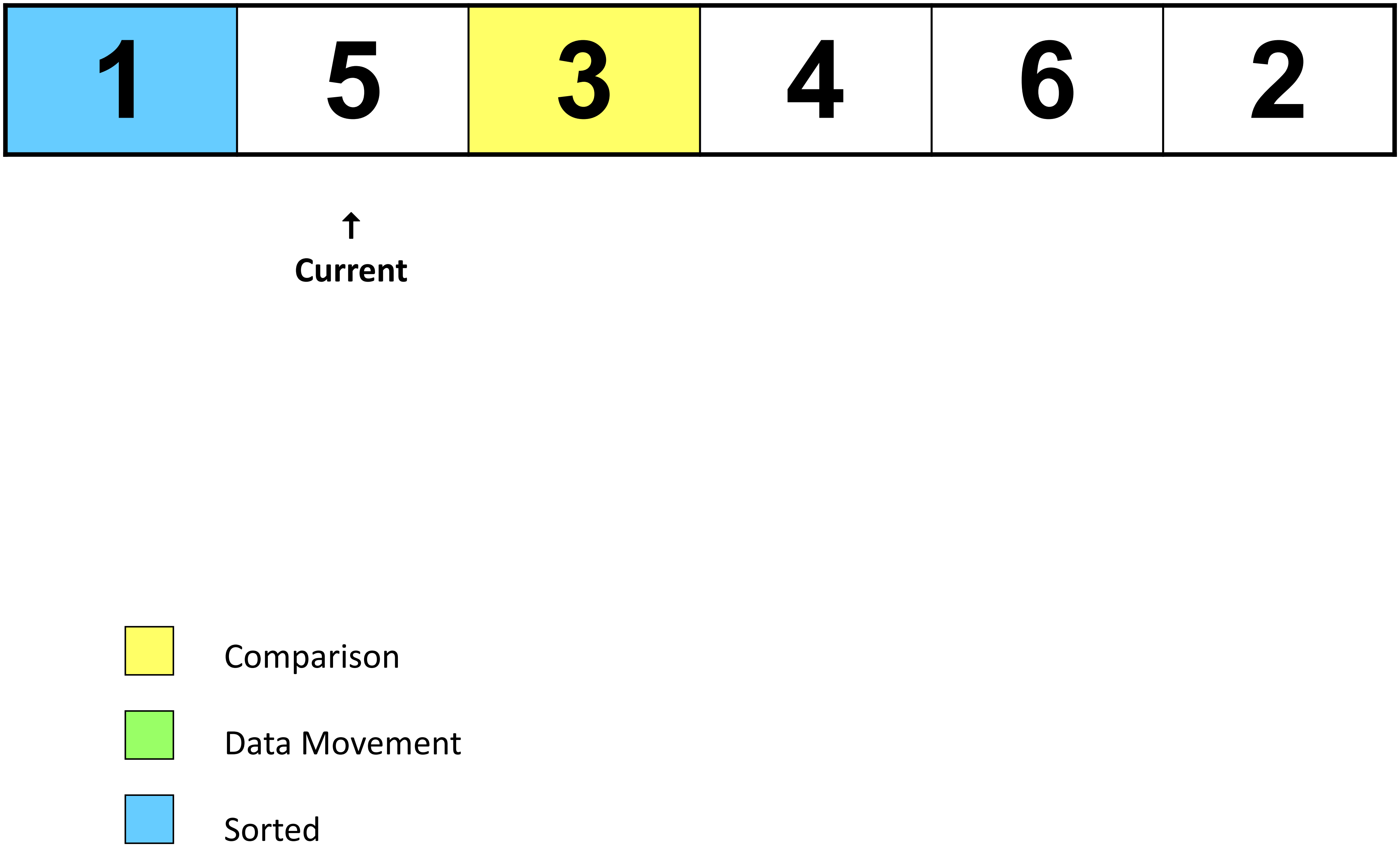
Data Movement

Sorted

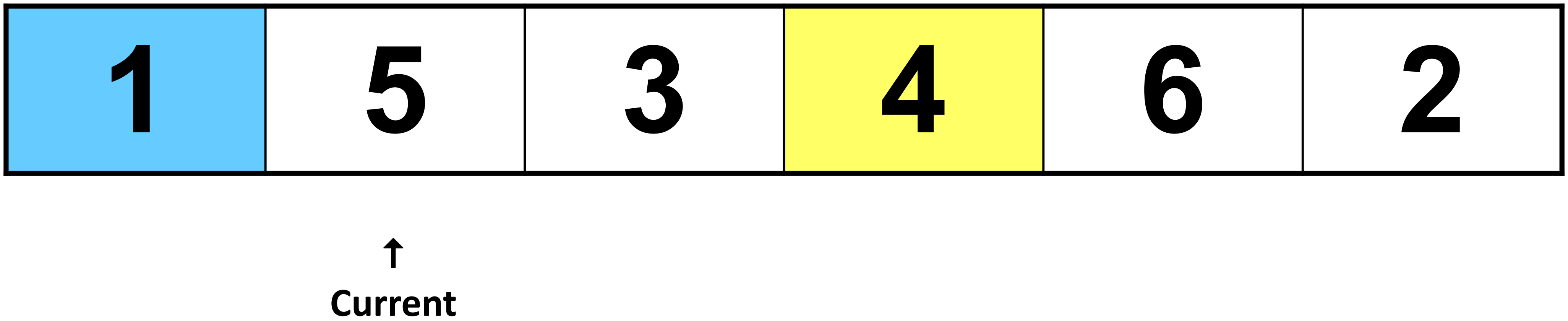
Selection Sort

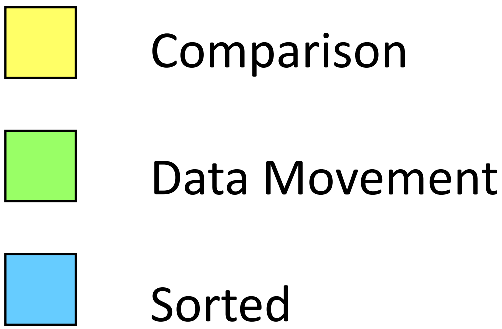


Selection Sort

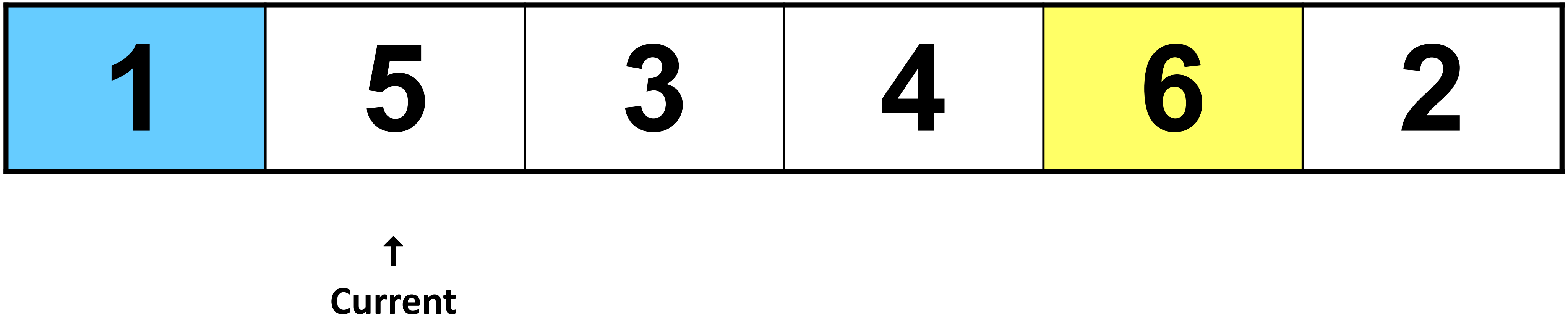


Selection Sort





Selection Sort

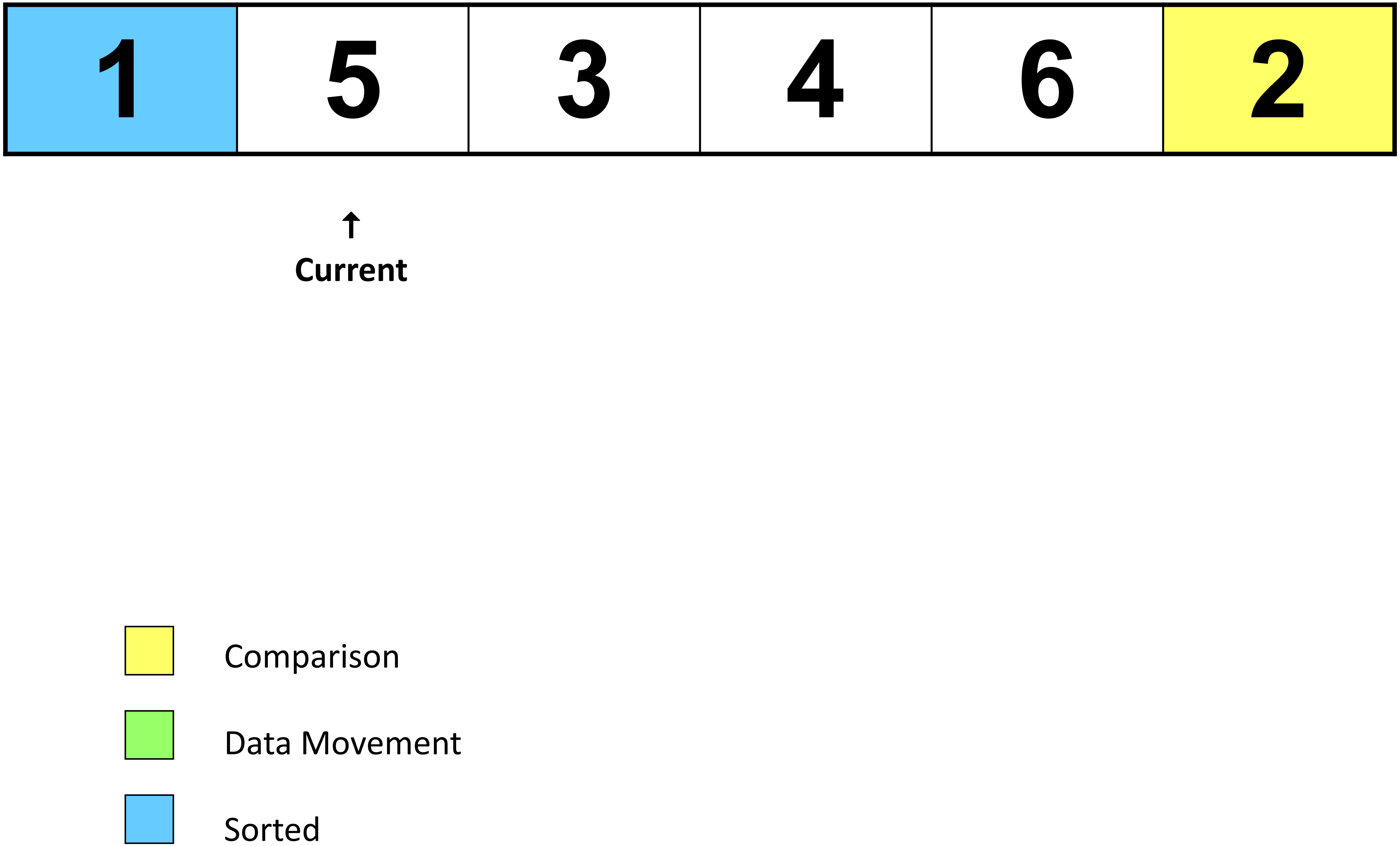


Comparison

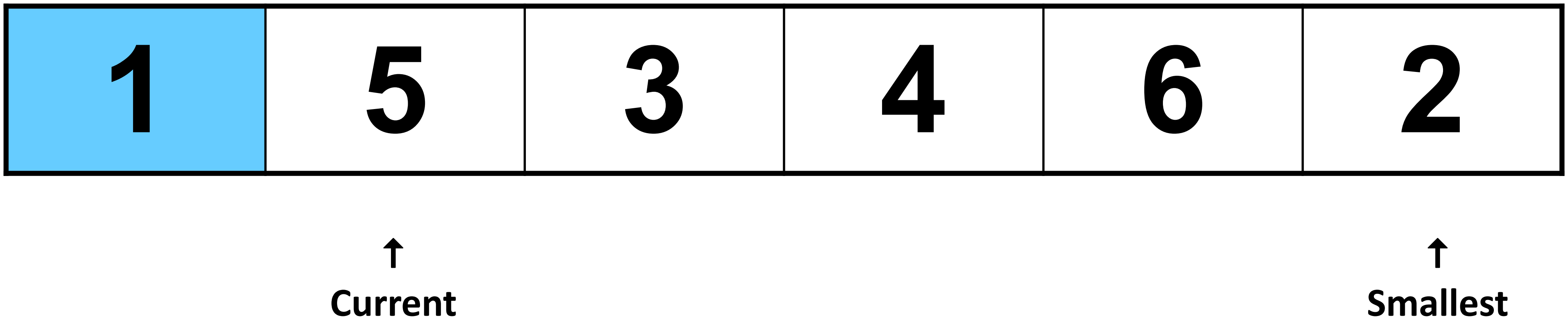
Data Movement

Sorted

Selection Sort



Selection Sort

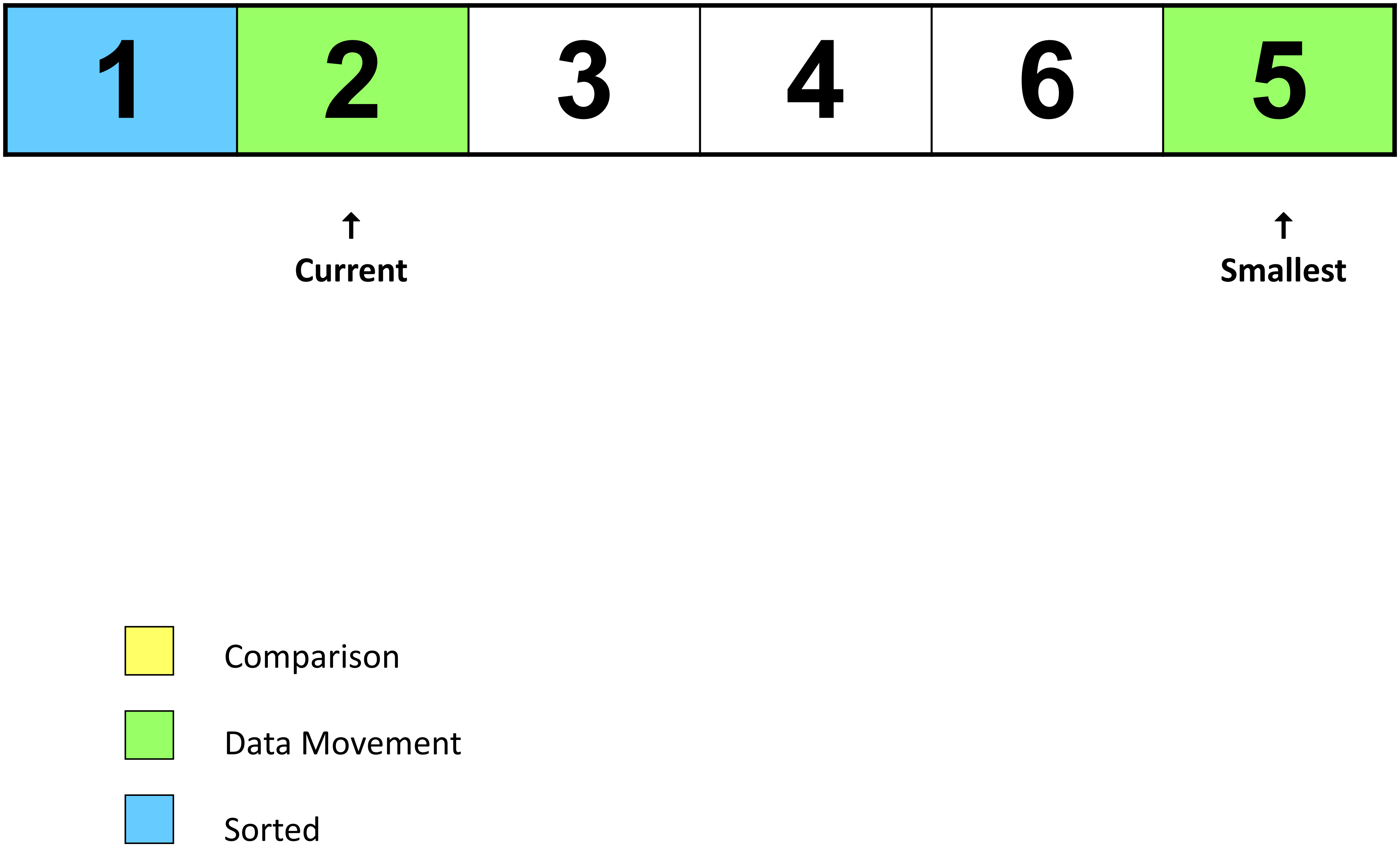


Comparison

Data Movement

Sorted

Selection Sort



Selection Sort

1	2	3	4	6	5

Comparison

Data Movement

Sorted

Selection Sort

1	2	3	4	6	5

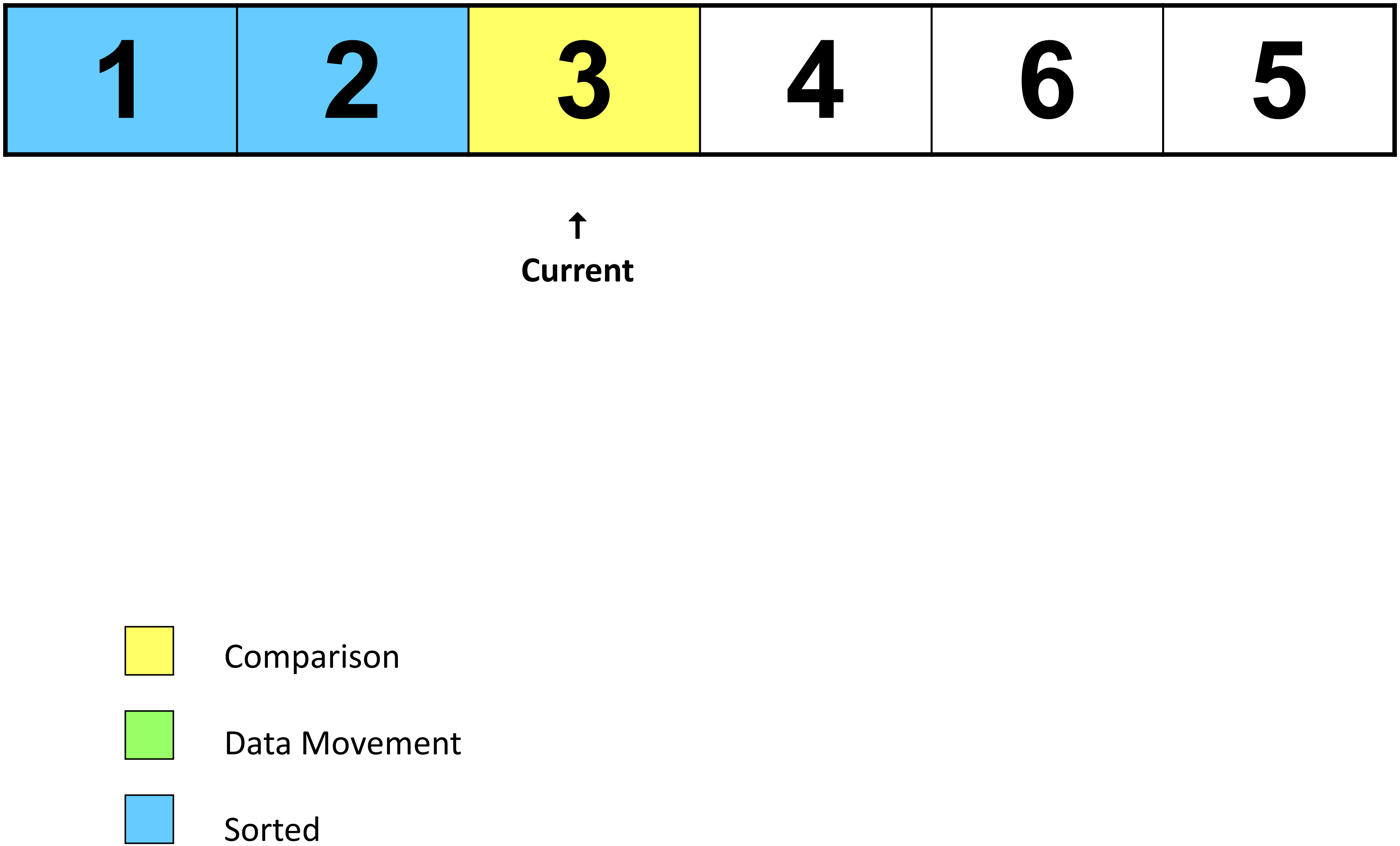
↑
Current

Comparison

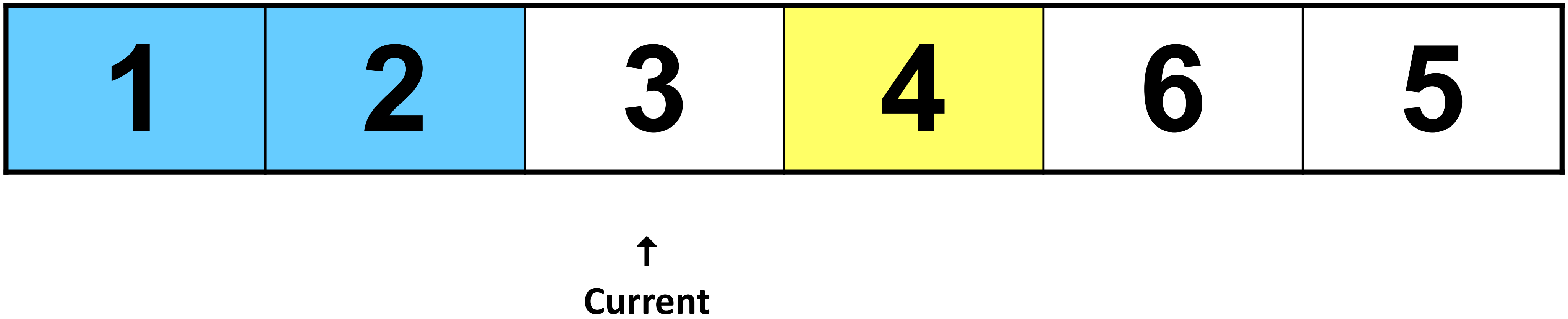
Data Movement

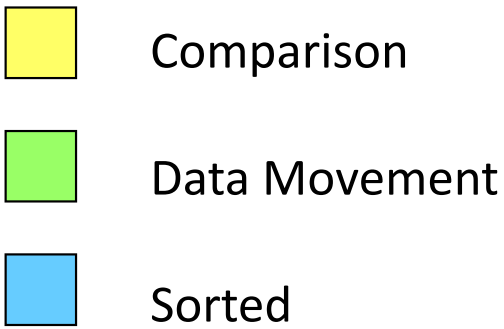
Sorted

Selection Sort

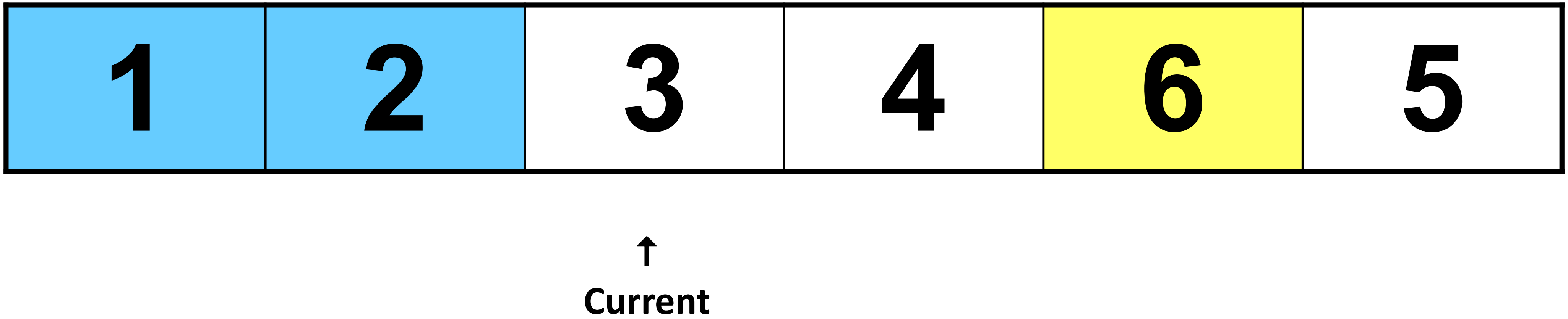


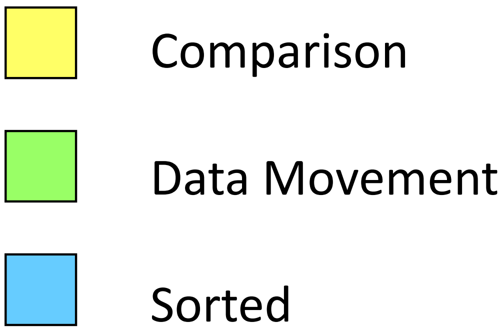
Selection Sort





Selection Sort





Selection Sort

1	2	3	4	6	5

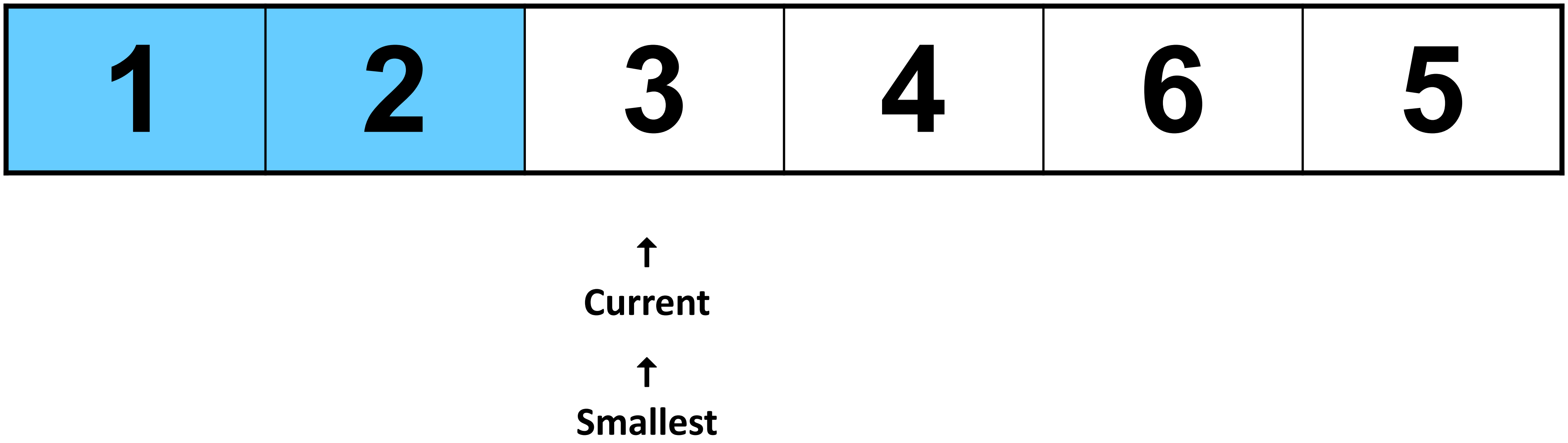
↑
Current

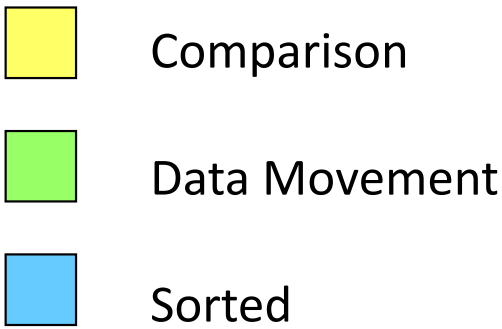
Comparison

Data Movement

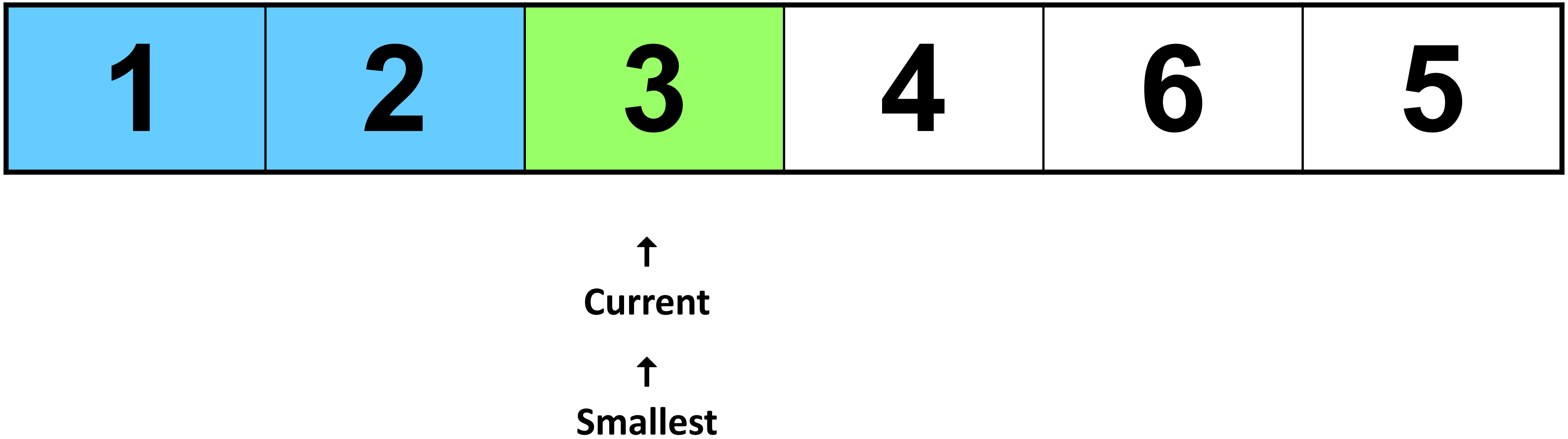
Sorted

Selection Sort





Selection Sort



Comparison

Data Movement

Sorted

Selection Sort

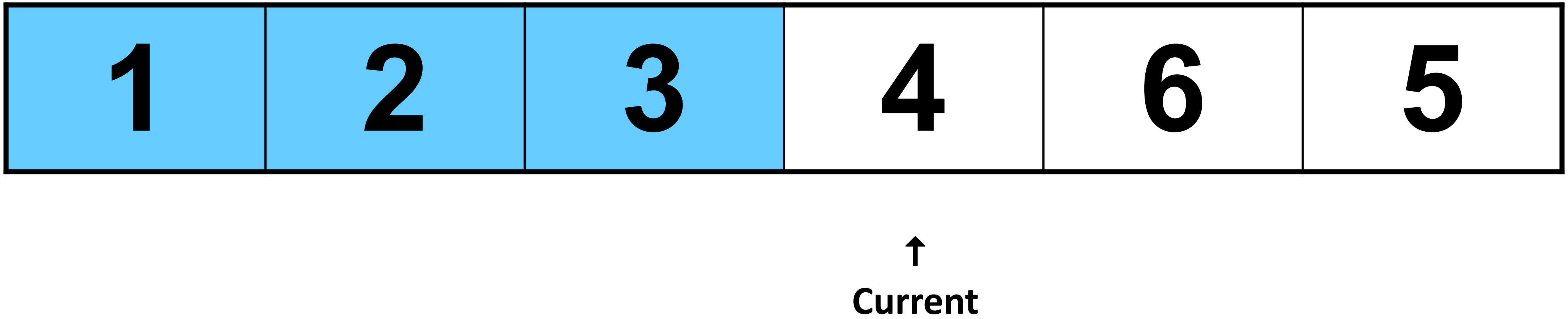
1	2	3	4	6	5

Comparison

Data Movement

Sorted

Selection Sort

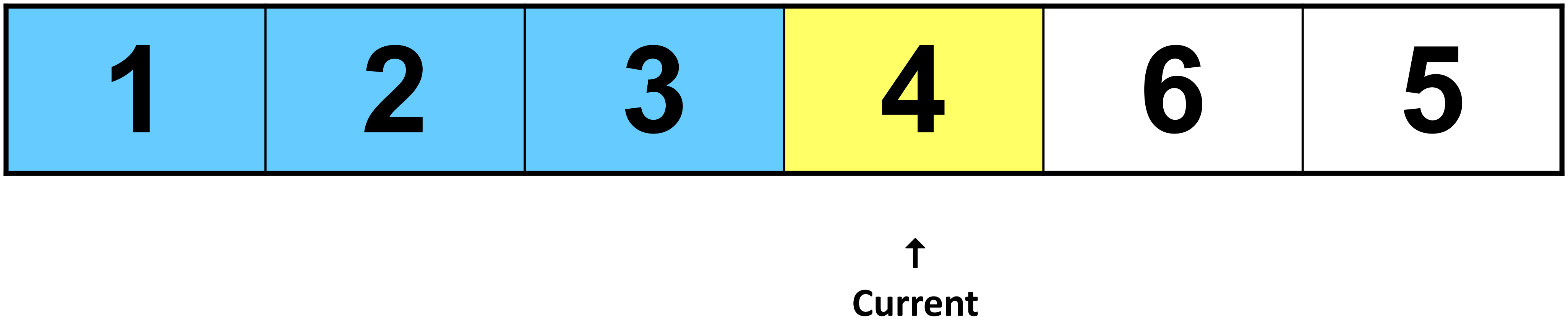


Comparison

Data Movement

Sorted

Selection Sort

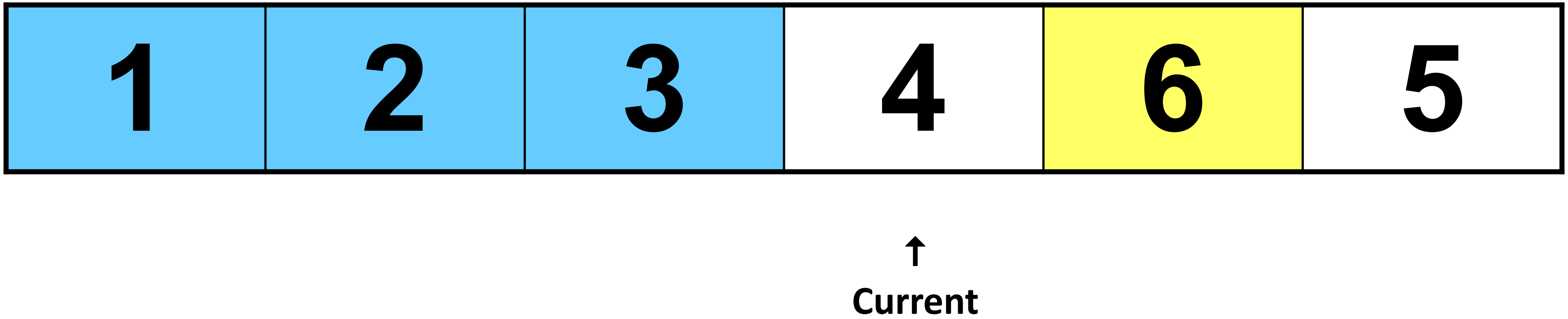


Comparison

Data Movement

Sorted

Selection Sort

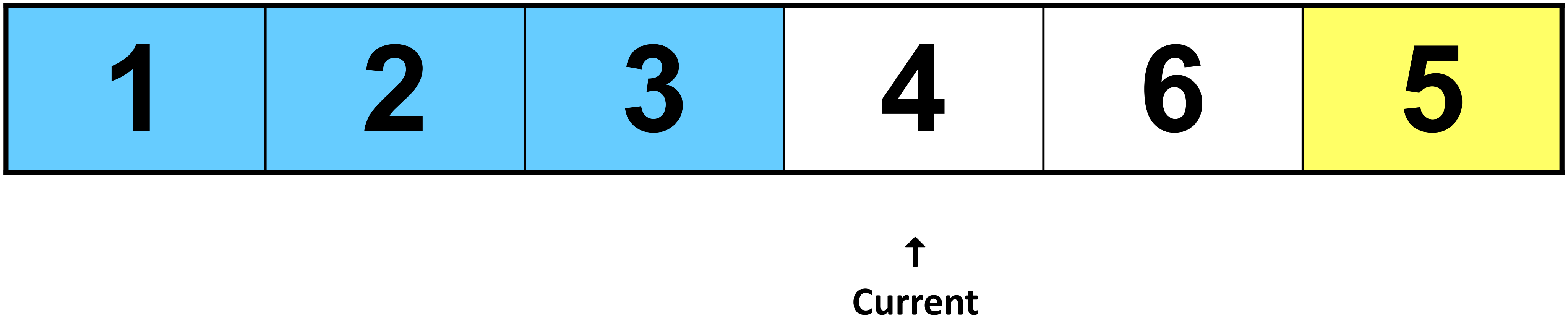


Comparison

Data Movement

Sorted

Selection Sort

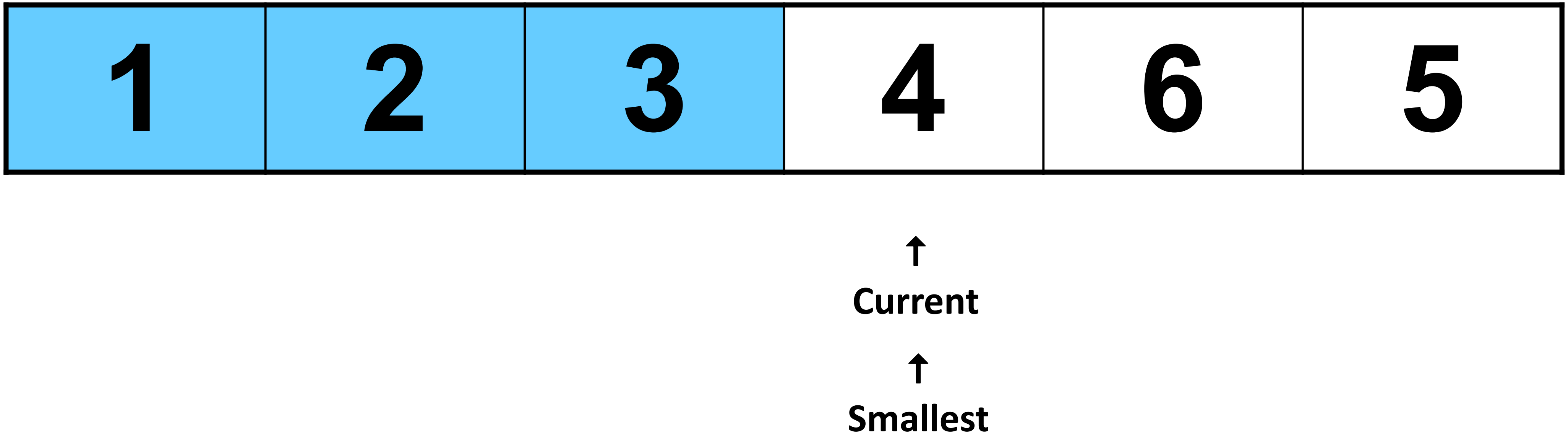


Comparison

Data Movement

Sorted

Selection Sort

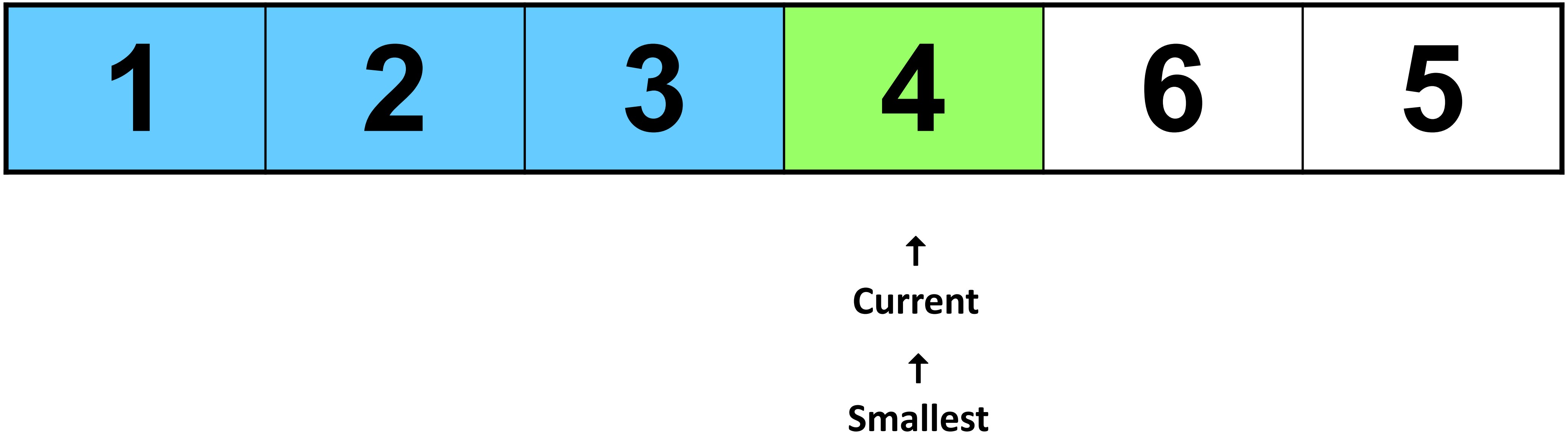


Comparison

Data Movement

Sorted

Selection Sort

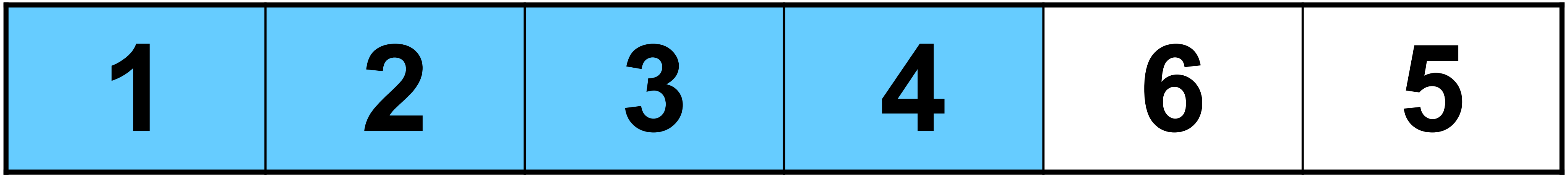


Comparison

Data Movement

Sorted

Selection Sort



Comparison

Data Movement

Sorted

Selection Sort

1	2	3	4	6	5

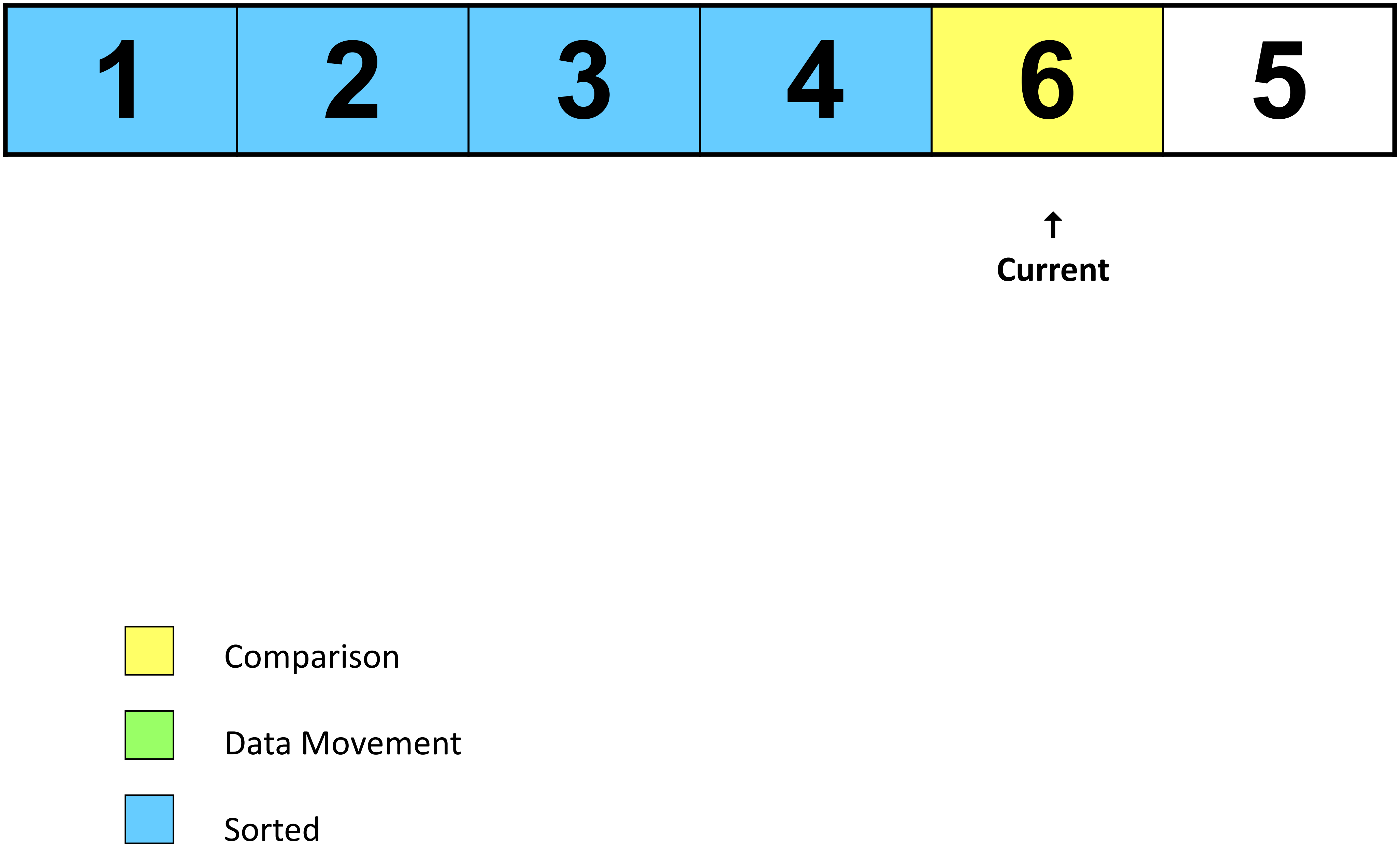
↑
Current

Comparison

Data Movement

Sorted

Selection Sort



Selection Sort

1	2	3	4	6	5

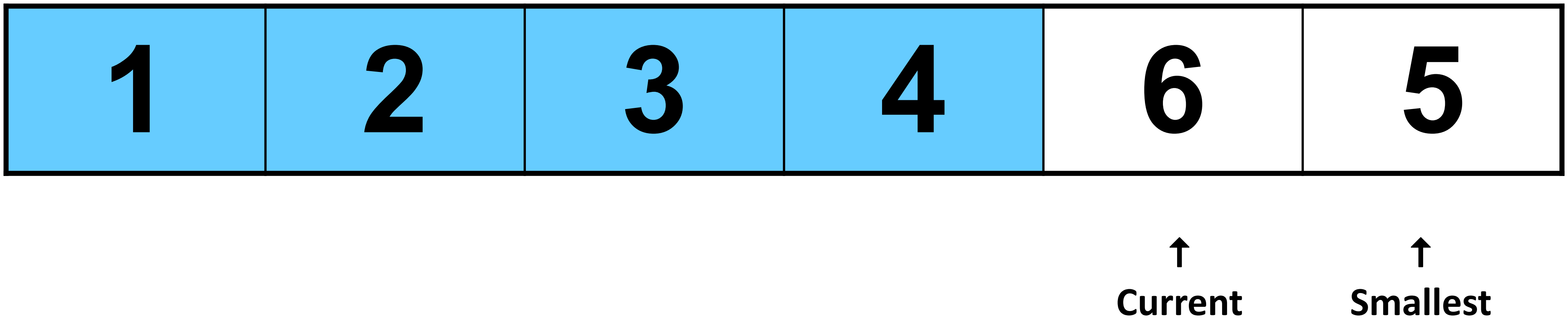
↑
Current

Comparison

Data Movement

Sorted

Selection Sort

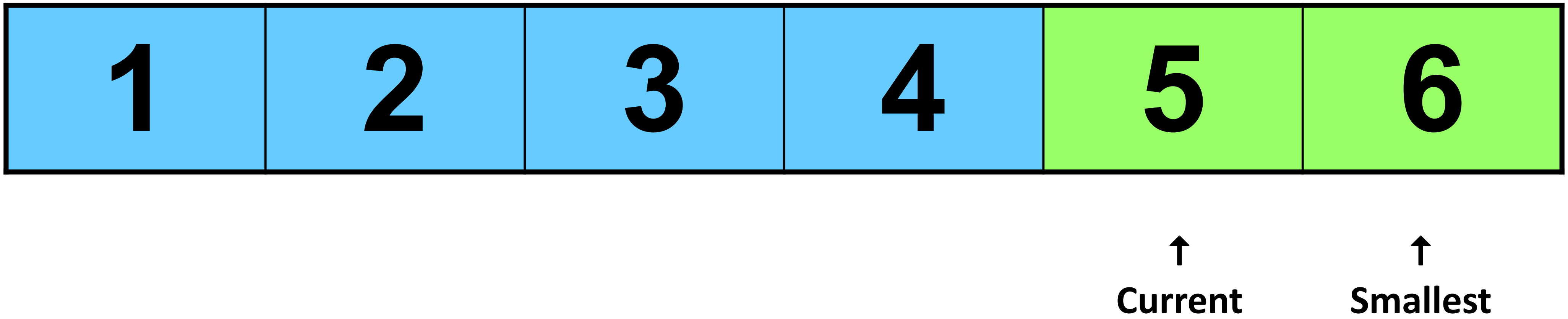


Comparison

Data Movement

Sorted

Selection Sort



Comparison

Data Movement

Sorted

Selection Sort

1	2	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

- It is also an in-place sorting algorithm because it uses no auxiliary data structures while sorting.

```
def selection_sort(A):
    n = len(A)
    for i in range(n - 1):
        min_index = i
        for j in range(i + 1, n):
            if A[j] < A[min_index]:
                min_index = j
        A[i], A[min_index] = A[min_index], A[i]
    return A


A = [5, 2, 9, 2, 1]
print(selection_sort(A))  # prints [1, 2, 2, 5, 9]
```

Insertion Sort

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

Insertion Sort

```
Value of j= 1
0 94 1 93
[93, 94, 92, 91, 90]
Value of j= 2
1 94 2 92
[93, 92, 94, 91, 90]
0 93 1 92
[92, 93, 94, 91, 90]
Value of j= 3
2 94 3 91
[92, 93, 91, 94, 90]
1 93 2 91
[92, 91, 93, 94, 90]
0 92 1 91
[91, 92, 93, 94, 90]
Value of j= 4
3 94 4 90
[91, 92, 93, 90, 94]
2 93 3 90
[91, 92, 90, 93, 94]
1 92 2 90
[91, 90, 92, 93, 94]
0 91 1 90
[90, 91, 92, 93, 94]
[90, 91, 92, 93, 94]
```

```python
def insertion_sort(lst):
    for i in range(1, len(lst)):
        j = i
        print("Value of j=", j)
        while j > 0 and lst[j - 1] > lst[j]:
            print(j-1, lst[j-1], j, lst[j])
            lst[j], lst[j - 1] = lst[j - 1], lst[j]
            print(lst)
            j -= 1
    return lst


print(insertion_sort([94, 93, 92, 91, 90]))  #[90, 91, 92, 93, 94]
```