Search & Sort

Data Structures – Coding Course

Search

Linear search

```c
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++)
        if (arr[i] == key)
            return i;
    return -1;
}
```

O(n)

Jump search

- Sorted structures

```
int jumpSearch(int arr[], int n, int key) {
    int step = sqrt(n);
    int prev = 0;
    // Find the block
    while (arr[min(step, n) - 1] < key) {
        prev = step;
        step += sqrt(n);
        if (prev >= n)
            return -1;
    }
    // Linear
    for (int i = prev; i < min(step, n); i++){
        if (arr[i] == key)
            return i;}
    return -1;
    }
```

O(√n)

Binary search

- Sorted structures

```
int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
       if (arr[mid] == key) return mid;
       else if (arr[mid] < key) low = mid + 1;
       else high = mid - 1;
    }
    return -1;}
```

O(logn)

Sort

Bubble sort

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1])
                swap(arr[j], arr[j + 1]);
}
```

$O(n^2)$

Selection sort

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++)
            if (arr[j] < arr[minIndex])
minIndex = j;
        swap(arr[i], arr[minIndex]);
    }
}
```

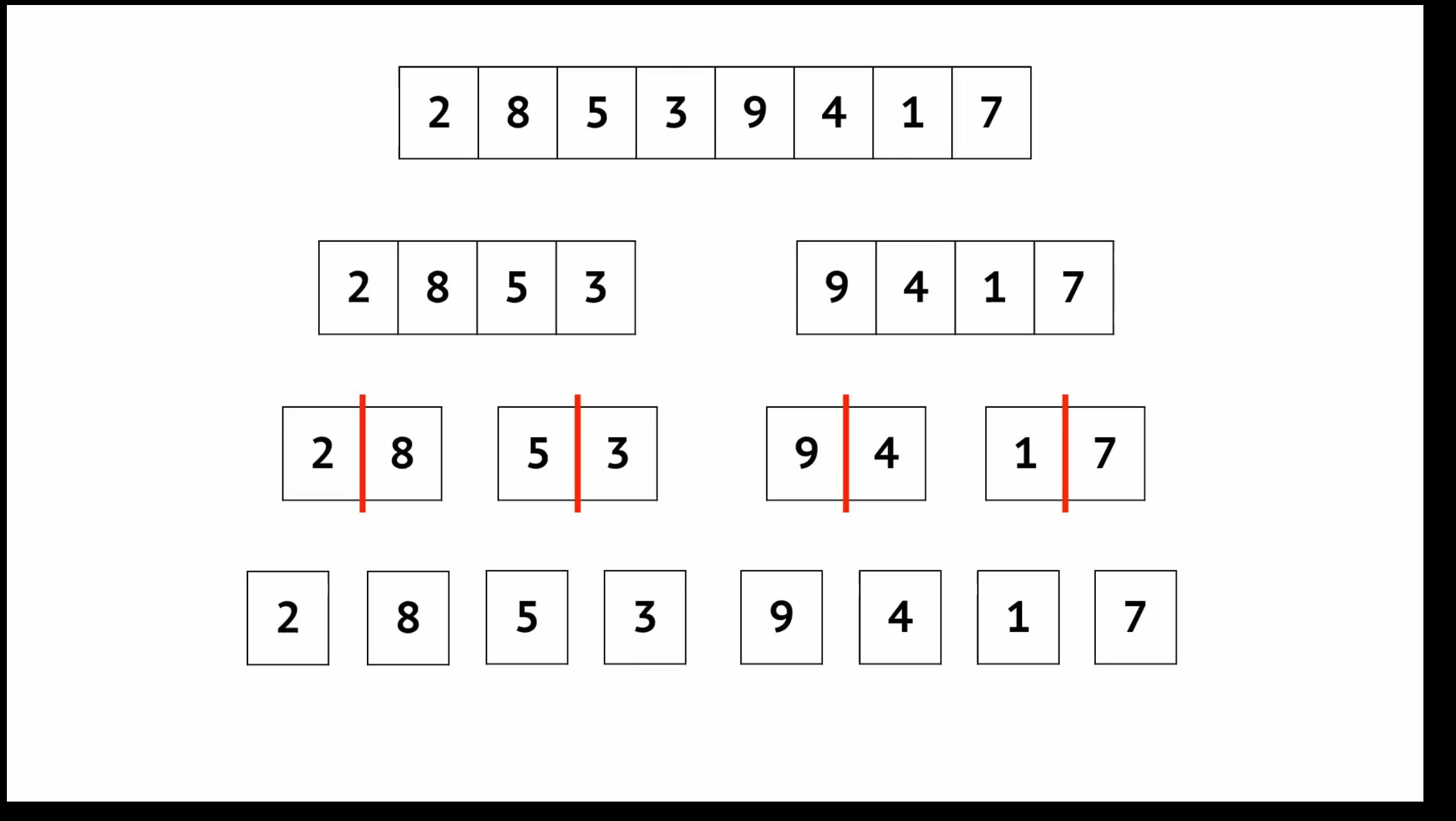
$O(n^2)$

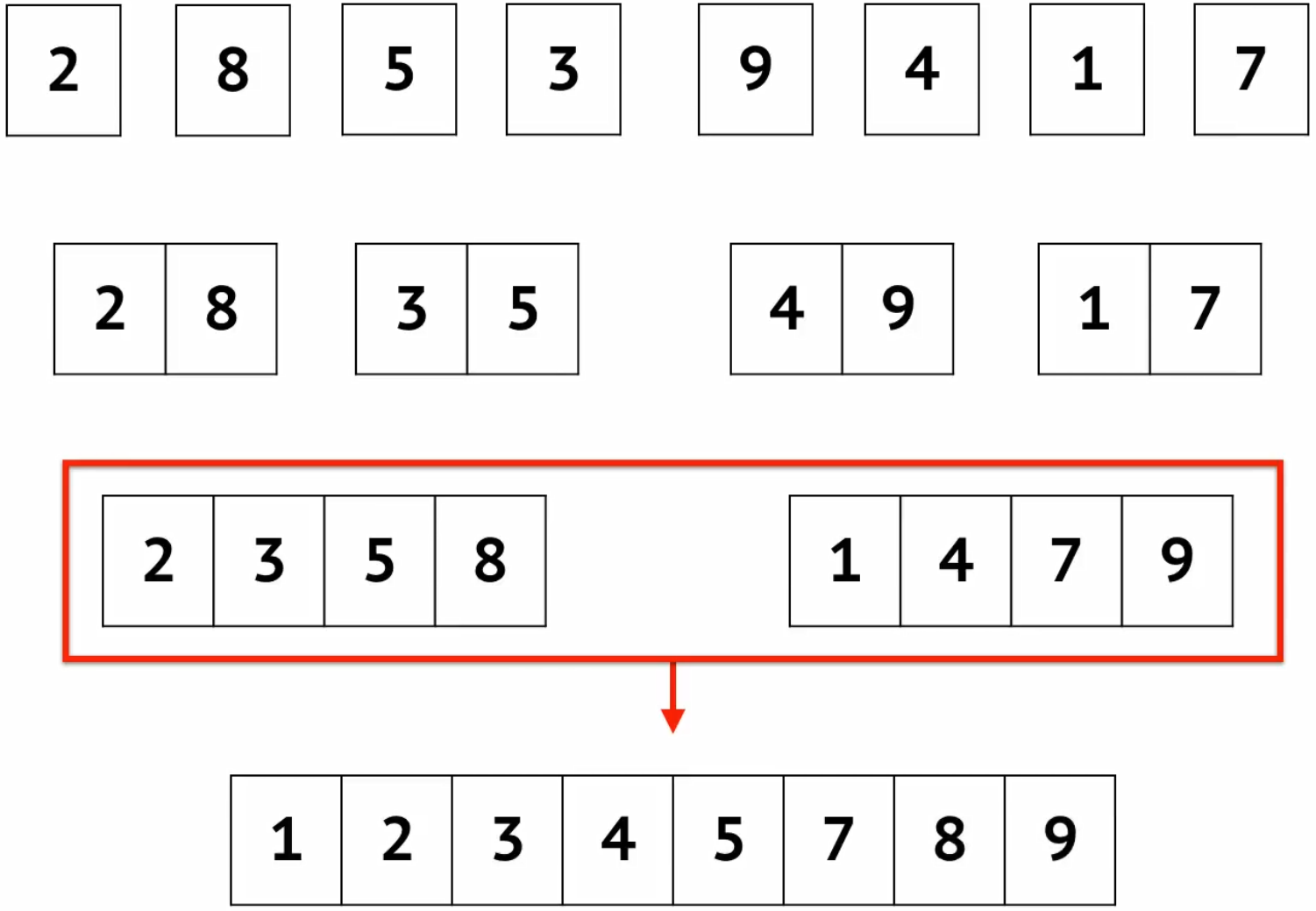
Merge sort

- Usually recursively
- Devide & conquer

```
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1, n2 = r - m;
    int L[100], R[100];
    for (int i = 0; i < n1; i++) L[i] = arr[l
+ i];
    for (int j = 0; j < n2; j++) R[j] = arr[m
+ 1 + j];
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2)
        arr[k++] = (L[i] <= R[j]) ? L[i++] :
R[j++];
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];}
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }}
```

O(n log n)


2 8 5 3 9 4 1 7
2 8 5 3
9 4 1 7
2 8
5 3
9 4
1 7
2 8 5 3 9 4 1 7


2 8 5 3 9 4 1 7
2 8 3 5 4 9 1 7
2 3 5 8 1 4 7 9
1 2 3 4 5 7 8 9

Quick sort

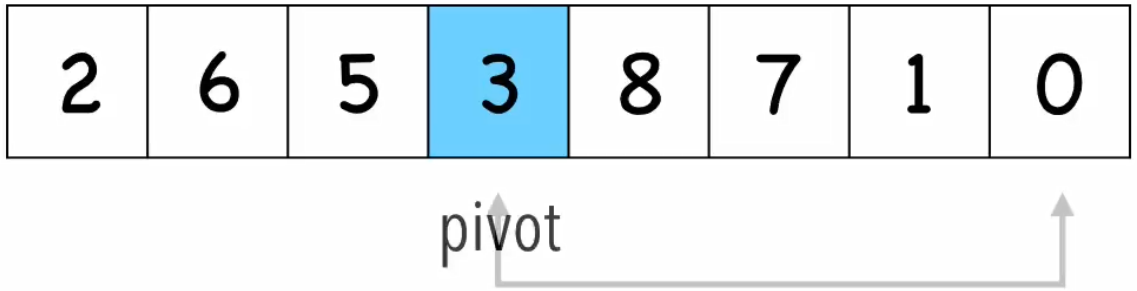
```
int partition(int arr[], int low, int high) {
    int pivot = arr[high], i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

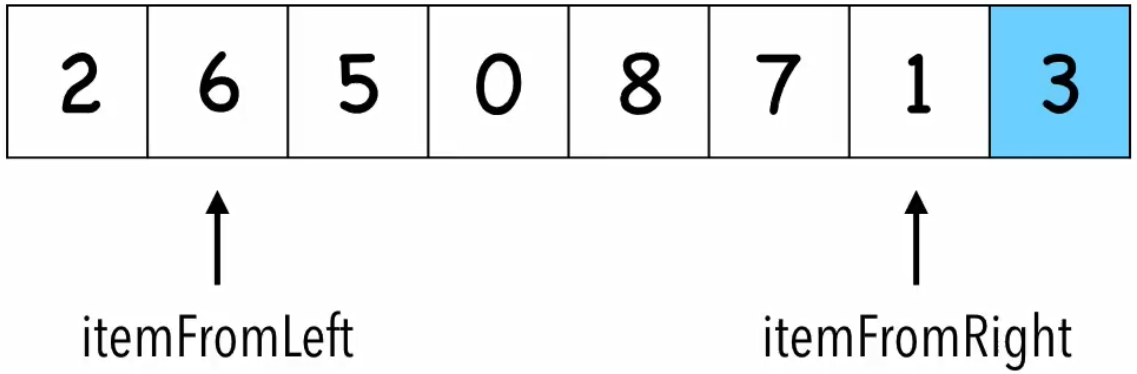
O(n log n)

quick sort = pivot

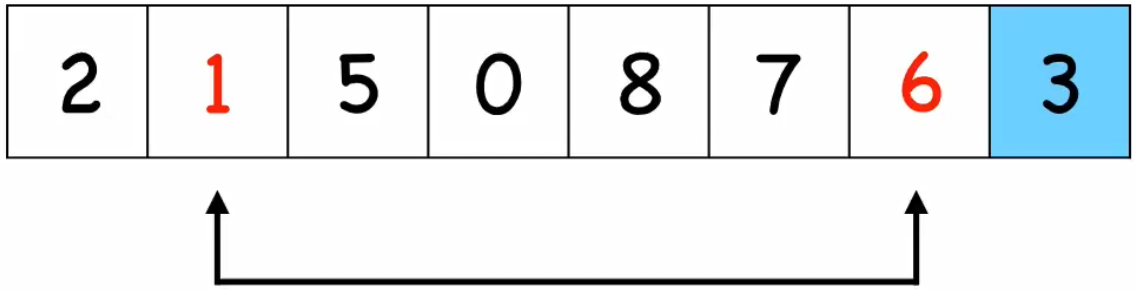
1. Correct position in final, sorted array
2. Items to the left are smaller
3. Items to the right are larger

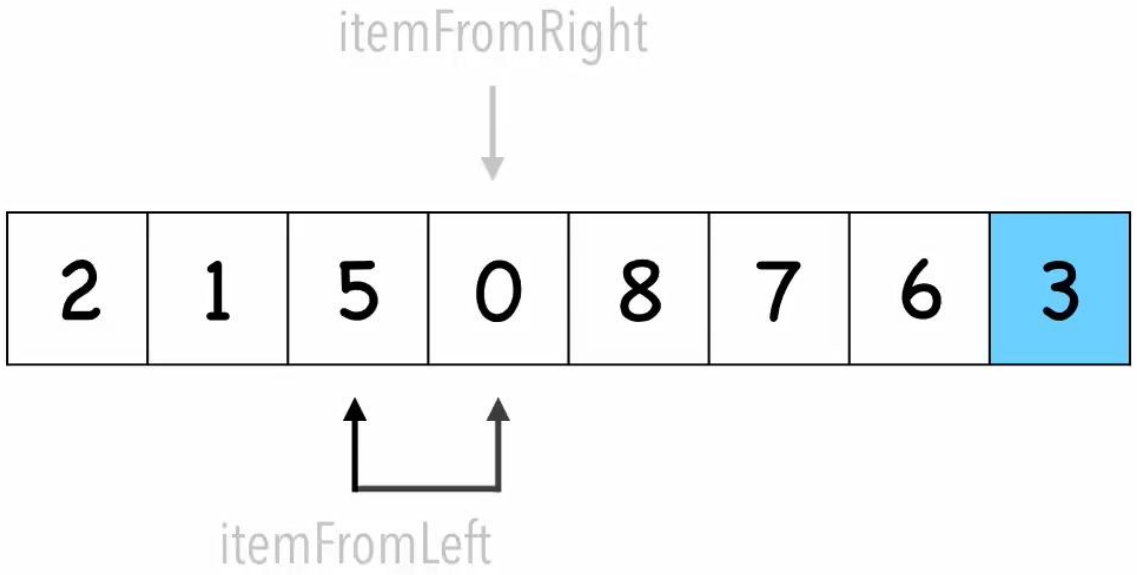

2
6
5
3
8
7
1
0
pivot

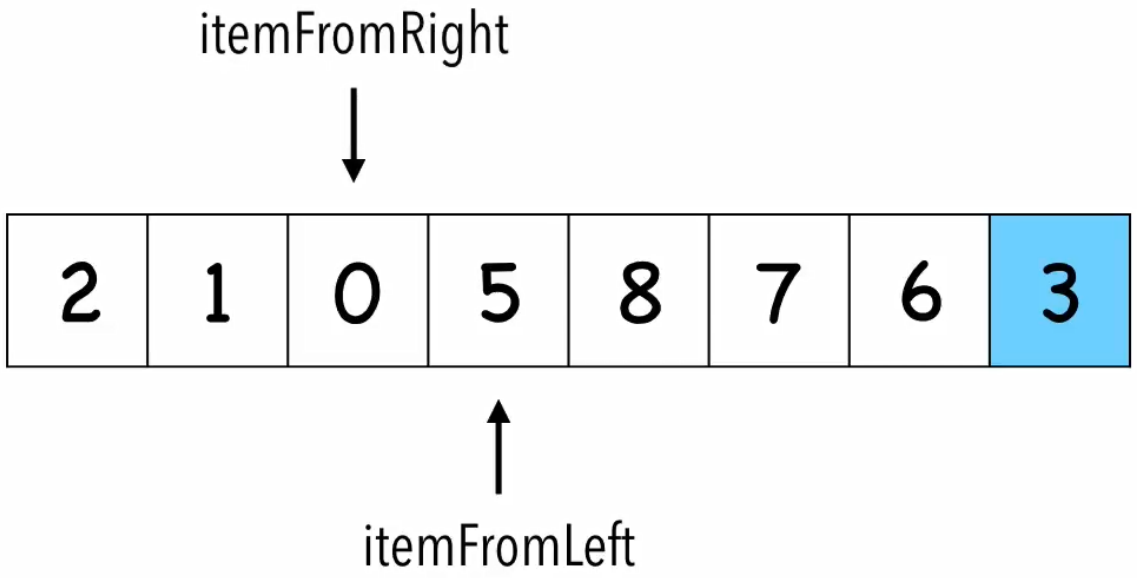
1. itemFromLeft that is larger than pivot
2. itemFromRight that is smaller than pivot



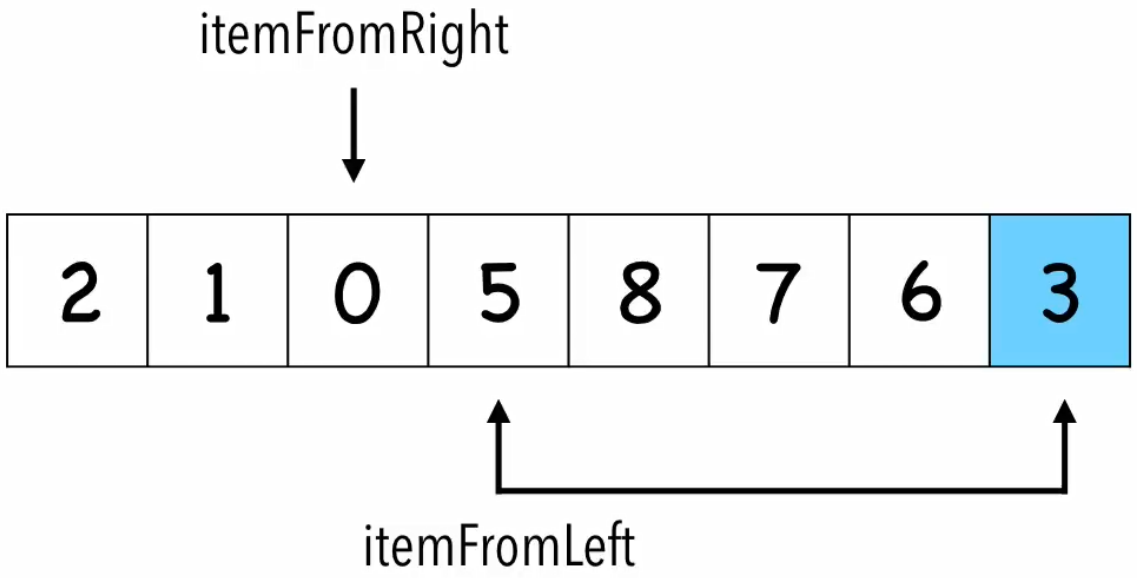
1. itemFromLeft that is larger than pivot
2. itemFromRight that is smaller than pivot




itemFromRight
2
1
5
0
8
7
6
3
itemFromLeft

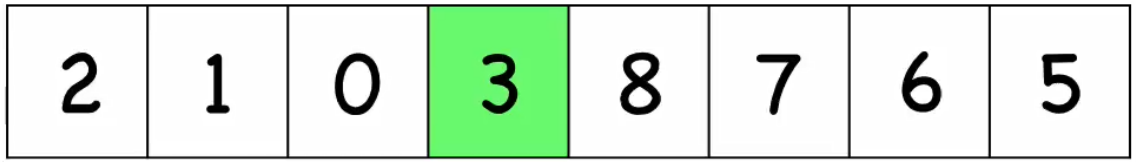


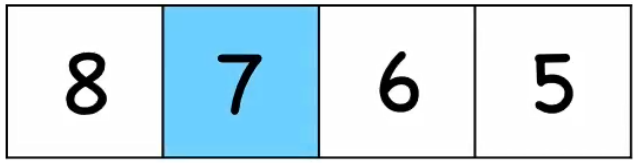
Stop when index of itemFromLeft > index of itemFromRight

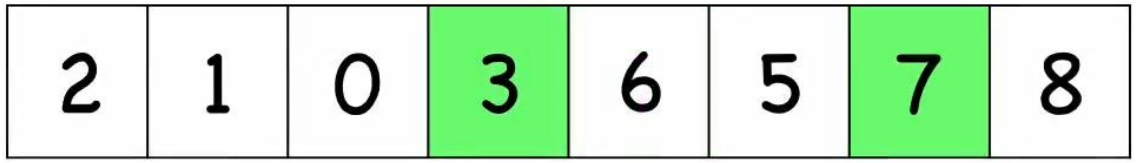


Swap itemFromLeft and pivot

1. Correct position in final, sorted array
2. Items to the left are smaller
3. Items to the right are larger

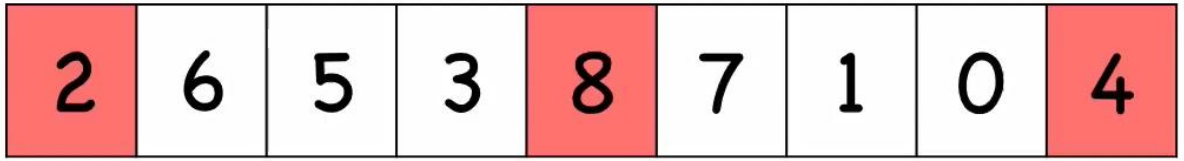


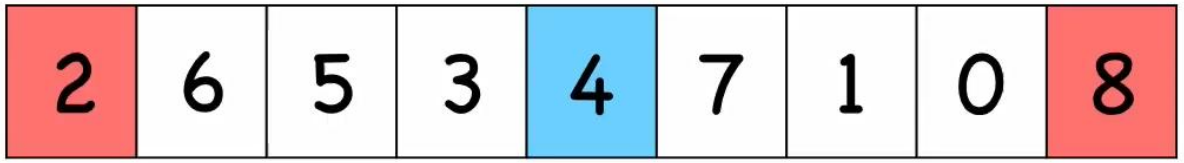

8
7
6
5
pivot


2
1
0
3
6
5
7
8

How to choose the pivot?

median–of–three


2 6 5 3 8 7 1 0 4


2
6
5
3
4
7
1
0
8
pivot

C++ Search built-in methods

Method	Function	How it works	Time Complexity
`std::find()`	`find(begin,end,key)`	Linear search	**O(n)**
`std::binary_search()`	`binary_search(begin,end,key)`	Binary search	**O(log n)**

C++ Sort built-in methods

Method	Function	How it works	Time Complexity
`std::sort()`	`sort(begin,end)`	Mix	O(n log n)