មហាវិទ្យាល័យវិស្វកម្ម
FACULTY OF ENGINEERING

Data Structure & Algorithm II

Lecture 2
Quicksort

Chhoeum Vantha, Ph.D.
Telecom & Electronic Engineering

Content

- Quicksort
- Issues To Consider
- Partitioning Strategy
- Picking the Pivot
- Quick Sort: Pseudo-code

Quicksort: Main Idea

- QuickSort is based on divide-and-conquer approach.
- QuickSort is an **in place** sorting algorithm.
- A sorting algorithm is said to be in place if it requires very little additional space beside the initial array holding the elements that are to be sorted.

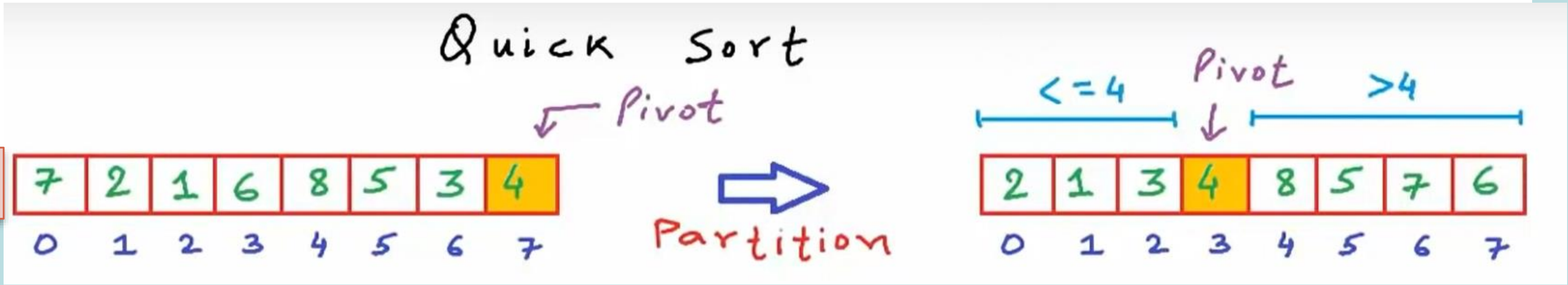
Quicksort: Main Idea

- The key to the Algorithm is the Partition Procedure, which rearranges the subarray a[p..r] in place.
- Partition selects an element **x** as a pivot element around which to partition the subarray a[p..r].

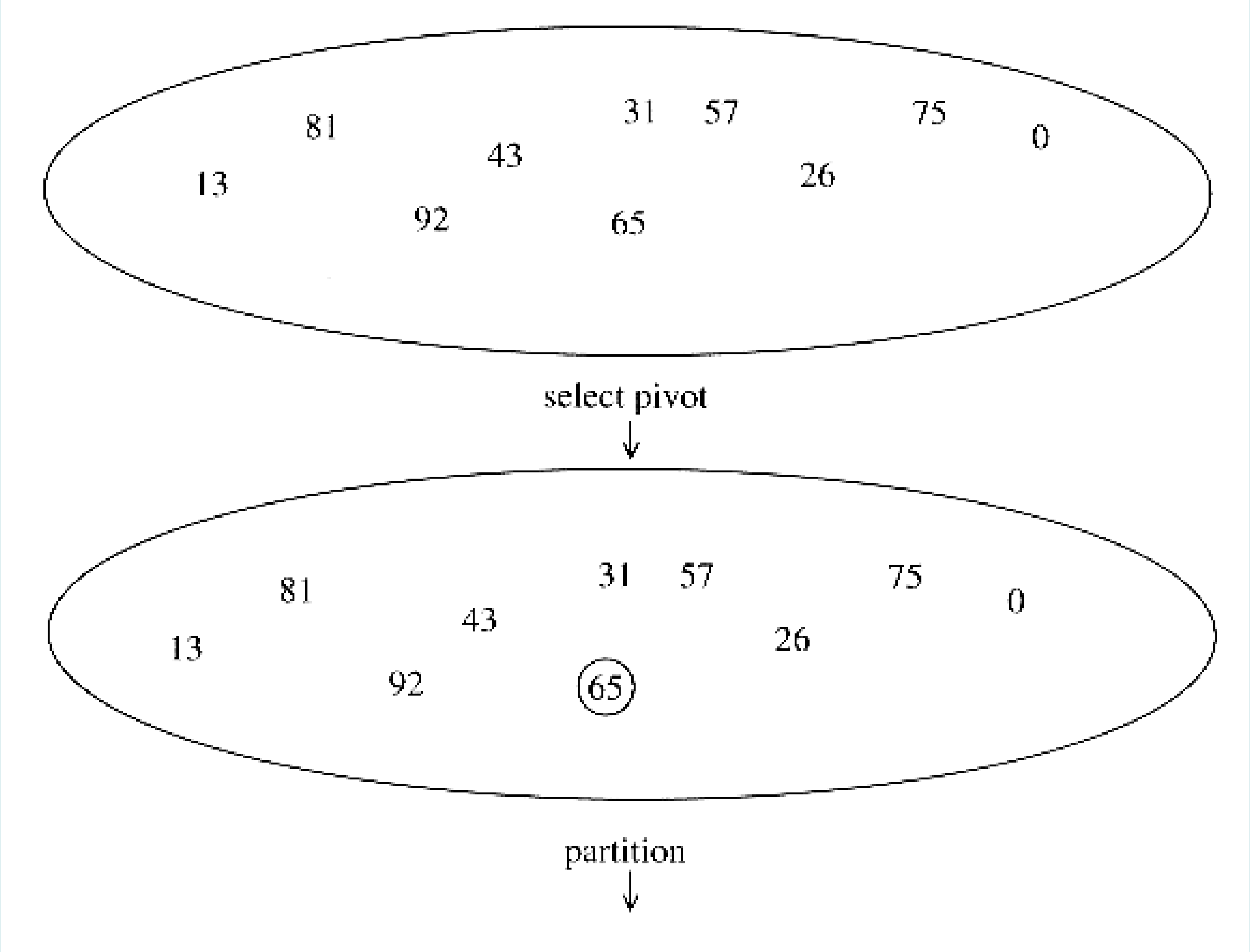
Quick Sort: Main Idea

1. If the number of elements in **S** is **0** or **1**, then return (base case).
2. Pick any element v in **S** (called the pivot).
3. Partition the elements in S except v into two disjoint groups:
    1. $S_1 = \{x \in S - \{v\} \mid x \leq v\}$
    2. $S_2 = \{x \in S - \{v\} \mid x \geq v\}$
4. Return {QuickSort(S_1) + v + QuickSort(S_2)}

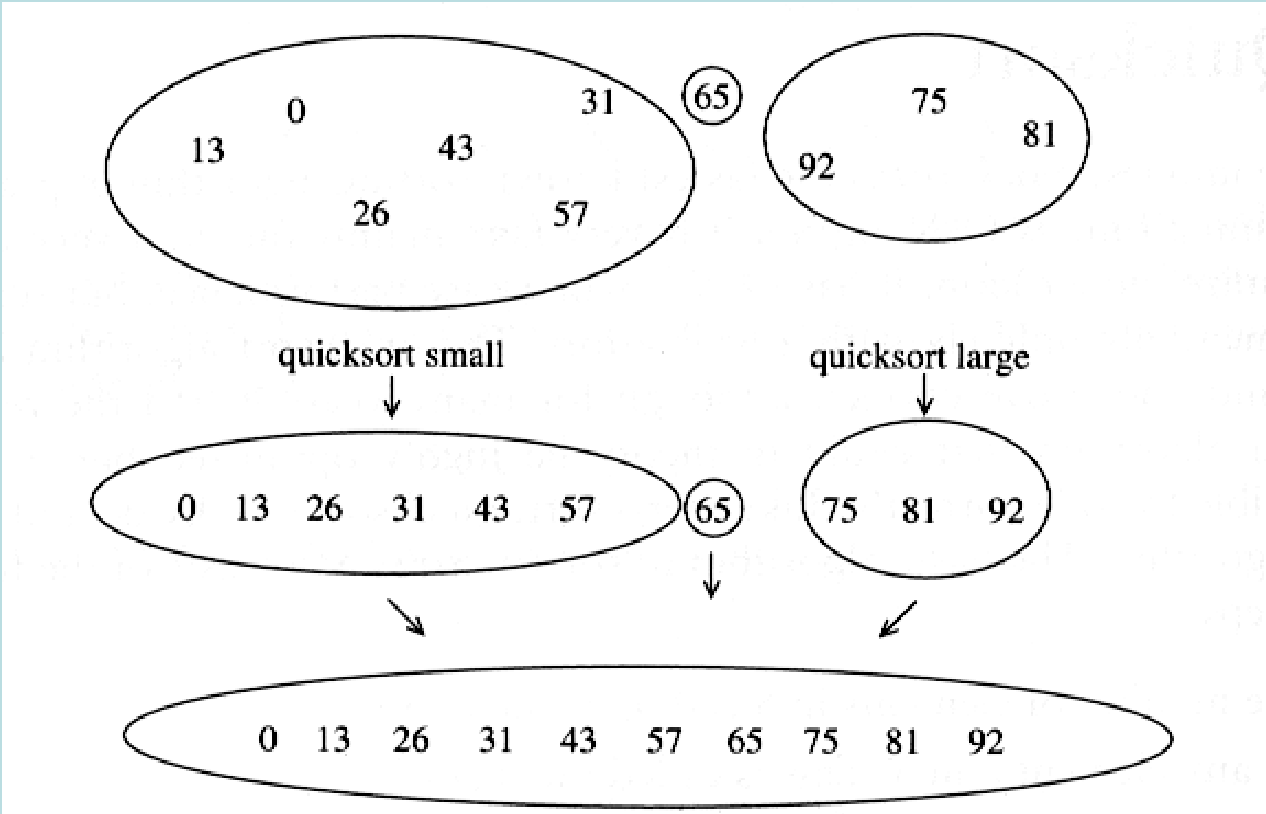
S



Quick Sort: Example



Example of Quick Sort...

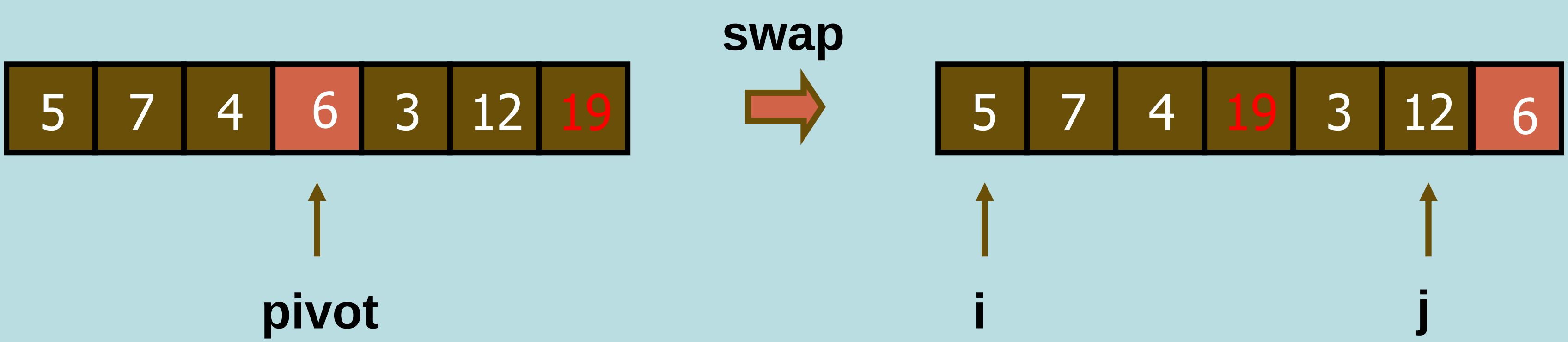


Issues To Consider

- How to pick the pivot?
  - Many methods (discussed later)
- How to partition?
  - Several methods exist.
  - The one we consider is known to give good results and to be easy and efficient.
  - We discuss the partition strategy first.

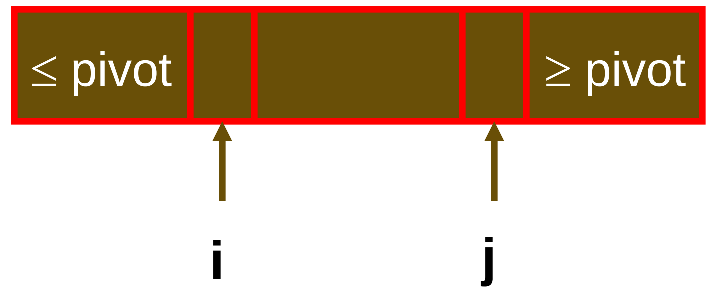
Partitioning Strategy

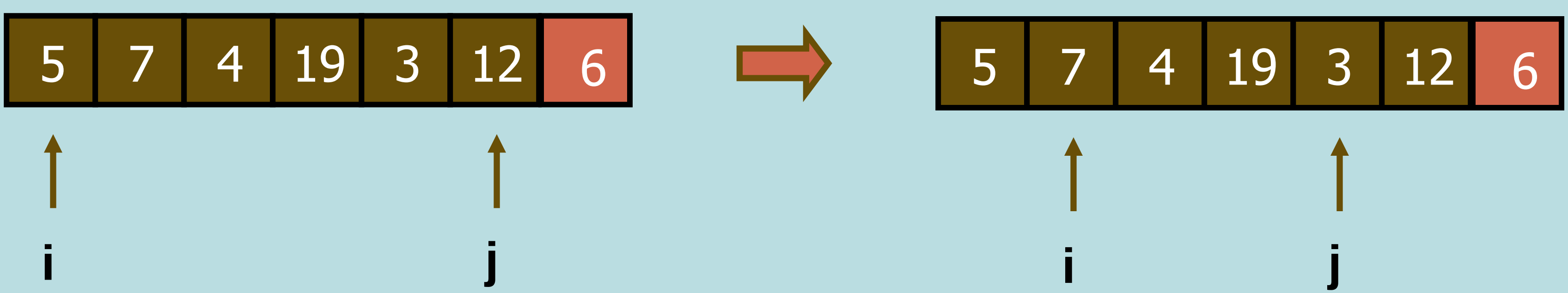
- For now, assume that **pivot** = average or median
- We want to partition array A[left .. right].
- First, get the pivot element out of the way by swapping it with the last element (swap pivot and A[right]).
- Let i start at the first element and j start at the next-to-last element (i = left, j = right – 1)



Partitioning Strategy

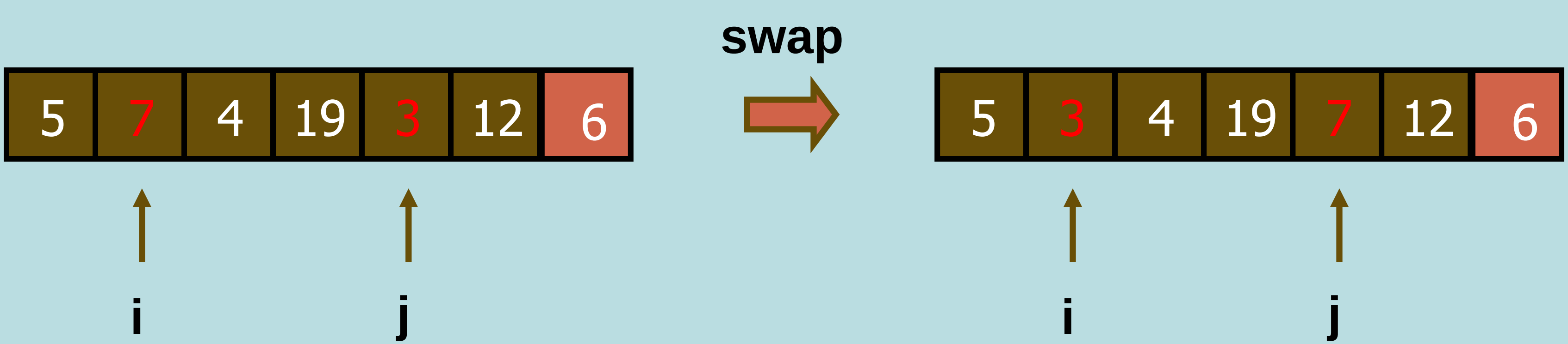
- Want to have
  - A[k] ≤ pivot, for k < i
  - A[k] ≥ pivot, for k > j
- When i < j
  - Move i right, skipping over elements smaller than the pivot
  - Move j left, skipping over elements greater than the pivot
  - When both i and j have stopped
    - A[i] ≥ pivot
    - A[j] ≤ pivot ⇒ A[i] and A[j] should now be swapped





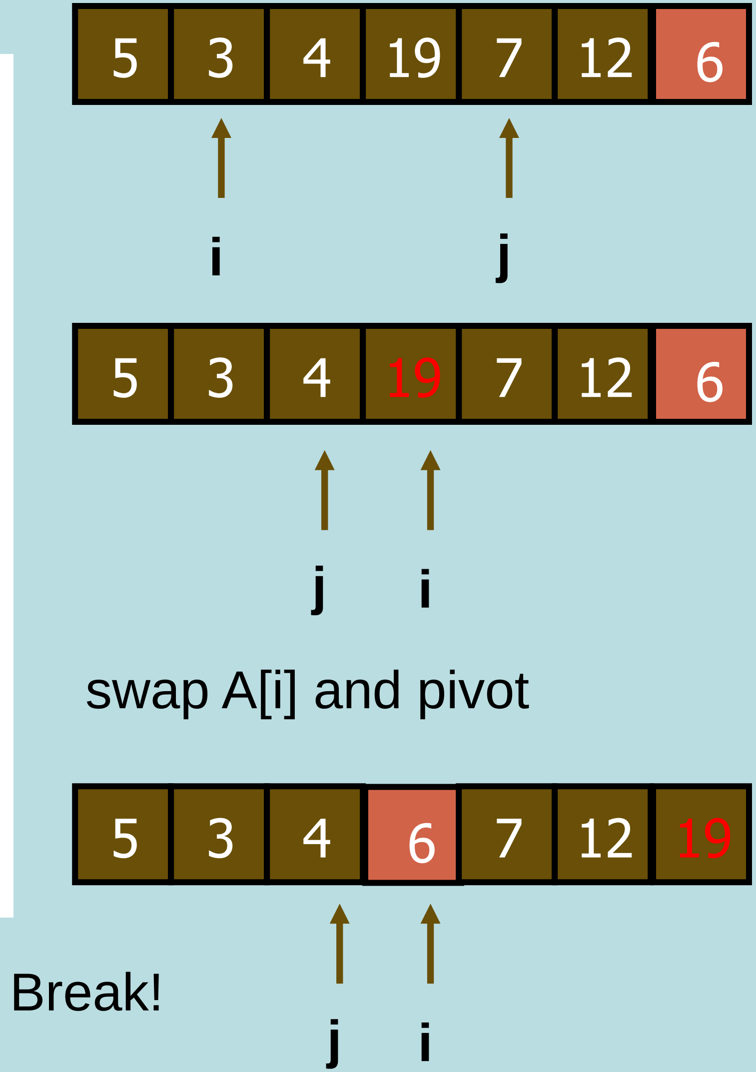
Partitioning Strategy (2)

- When i and j have stopped and i is to the left of j (thus legal)
  - Swap A[i] and A[j]
    - The large element is pushed to the right and the small element is pushed to the left
  - After swapping
    - A[i] ≤ pivot
    - A[j] ≥ pivot
  - Repeat the process until i and j cross



Partitioning Strategy (3)

- When i and j have crossed
  - swap A[i] and pivot
- Result:
  - A[k] ≤ pivot, for k < i
  - A[k] ≥ pivot, for k > i



Picking the Pivot

- There are several ways to pick a pivot.

- Objective:
  - Choose a pivot so that we will get 2 partitions of (almost) equal size.

Picking the Pivot

- Use the first element as pivot
  - if the input is random, ok.
  - if the input is presorted (or in reverse order)
    - all the elements go into S_2 (or S_1).
    - this happens consistently throughout the recursive calls.
    - results in $O(N^2)$ behavior (we analyze this case later).
- Choose the pivot randomly
  - generally safe,
  - but random number generation can be expensive and does not reduce the running time of the algorithm.

Picking the Pivot

- Use the median of the array (ideal pivot)
  - The $\lceil N/2 \rceil$ *th* largest element
  - Partitioning always cuts the array into roughly half
  - An optimal quick sort (O(N log N))
  - However, hard to find the exact median

- Median-of-three partitioning
  - eliminates the bad case for sorted input.
  - reduces the number of comparisons by 14%.

Median of Three Method

- Compare just three elements: the leftmost, rightmost and center
  - Swap these elements if necessary so that
    - A[left] = Smallest
    - A[right] = Largest
    - A[center] = Median of three
  - Pick A[center] as the **pivot**.
  - Swap A[center] and A[right – 1] so that the pivot is at the second last position (why?)

```
int center = ( left + right ) / 2;
if( a[ center ] < a[ left ] )
    swap( a[ left ], a[ center ] );
if( a[ right ] < a[ left ] )
    swap( a[ left ], a[ right ] );
if( a[ right ] < a[ center ] )
    swap( a[ center ], a[ right ] );

    // Place pivot at position right - 1
swap( a[ center ], a[ right - 1 ] );
```

Median of Three: Example

2	5	6	4	13	3	12	19	6

A[left] = 2, A[center] = 13, A[right] = 6

2	5	6	4	6	3	12	19	13

Swap A[center] and A[right]

2	5	6	4	6	3	12	19	13

Choose A[center] as pivot

pivot

2	5	6	4	19	3	12	6	13

Swap pivot and A[right – 1]

pivot

We only need to partition A[left + 1, …, right – 2]. Why?

Quick Sort Summary

- Recursive case: QuickSort(a, left, right)

  pivot = median3(a, left, right);

  Partition a[left … right] into a[left … i-1], i, a[i+1 … right];

  QuickSort(a, left, i-1);

  QuickSort(a, i+1, right);

- Base case: when do we stop the recursion?
  - In theory, when left >= right.
  - In practice, …

Quick Sort: Pseudo-code

```
if( left + 10 <= right )
{
    Comparable pivot = median3( a, left, right );

        // Begin partitioning
    int i = left, j = right - 1;
    for( ; ; )
    {
        while( a[ ++i ] < pivot ) { }
        while( pivot < a[ --j ] ) { }
        if( i < j )
            swap( a[ i ], a[ j ] );
        else
            break;
    }

    swap( a[ i ], a[ right - 1 ] );  // Restore pivot

    quicksort( a, left, i - 1 );     // Sort small elements
    quicksort( a, i + 1, right );    // Sort large elements
}
else  // Do an insertion sort on the subarray
    insertionSort( a, left, right );
```

Choose pivot

Partitioning

Recursion

For small arrays

Partitioning Part

- The partitioning code we just saw works only if pivot is picked as median-of-three.
  - A[left] ≤ pivot and A[right] ≥ pivot
  - Need to partition only A[left + 1, …, right – 2]

- j will not run past the beginning
  - because A[left] ≤ pivot

- i will not run past the end
  - because A[right-1] = pivot

```
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```

Analysis

Assumptions:

- A random pivot (no median-of-three partitioning)
- No cutoff for small arrays (to make it simple)

1. If the number of elements in S is 0 or 1, then return (**base case**).
2. Pick an element v in S (called the pivot).
3. Partition the elements in S except v into two disjoint groups:
   1. $S_1 = \{x \in S - \{v\} \mid x \leq v\}$
   2. $S_2 = \{x \in S - \{v\} \mid x \geq v\}$
4. Return {QuickSort(S_1) + v + QuickSort(S_2)}

W2 – Lab

Analysis on Recursion and Quicksort

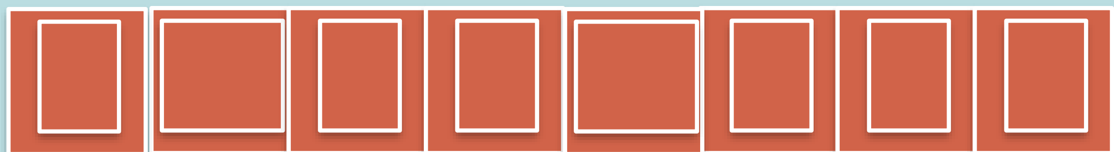
Check the given array above:

- int arr [] = {8, 15, 4, 3, 18, 7, 1, 4}
- first_index =0
- last_index = 7,
- pivot = arr[last_index];

8	15	4	3	18	7	1	4

Analysis on Recursion and Quicksort

1. Using the code below swap(arr, 2, 6) the array and show the arr after swap.



```cpp
#include <iostream>
using namespace std;
void swap(int arr[] , int pos1, int pos2)
{
    int temp;
    temp = arr[pos1];
    arr[pos1] = arr[pos2];
    arr[pos2] = temp;
}
```

Analysis on Recursion and Quicksort

2. Show the partition (arr, first_ind, last_ind, pivot) using swap and partition function to the original array below:

8	15	4	3	18	7	1	4

```
#include <iostream>
using namespace std;
void swap(int arr[] , int pos1, int pos2)
{
    int temp;
    temp = arr[pos1];
    arr[pos1] = arr[pos2];
    arr[pos2] = temp;
}
```

Analysis on Recursion and Quicksort

- Partition **Function:**

```
int partition(int arr[], int first_ind, int last_ind, int pivot){
    // low = first_ind, higt = last_ind
    int i = first_ind;
    int j = first_ind;
    while( i <= last_ind){
        if(arr[i] > pivot){
            i++;
        }
        else{
            swap(arr,i,j);
            i++;
            j++;
        }
    }
    return j-1;
}
```

Analysis on Recursion and Quicksort

3. Show the quicksort(arr, 0, 7) to original array below:

arr []

8	15	4	3	18	7	1	4

```
void quickSort(int arr[], int first_ind, int last_ind){
    if(first_ind < last_ind){
    int pivot = arr[last_ind];
    int pos = partition(arr, first_ind, last_ind, pivot);

    quickSort(arr, first_ind, pos-1);
    quickSort(arr, pos+1, last_ind);
    }
}
```

Exercise

1. Implementation of Recursion and Quicksort to singly linked list
2. Implementation of Recursion and Quicksort to a doubly linked list

Preparation Reading/Research

1. What is Quicksort?
2. Compare the Strengths and Weaknesses of Quicksort.
3. What are the factors to consider before using Quicksort?

=> Write down your answer on your note!

Thanks!