# Data Structures and Algorithms

# Lecture 7: Quick Sort

Department of Computer Science & Technology
United International College

# Outline

- Introduction to Quick Sort
- Quick Sort Components
  - Partitioning
  - Small Array Strategy
  - Picking the Pivot
- Cost Analysis

# Introduction

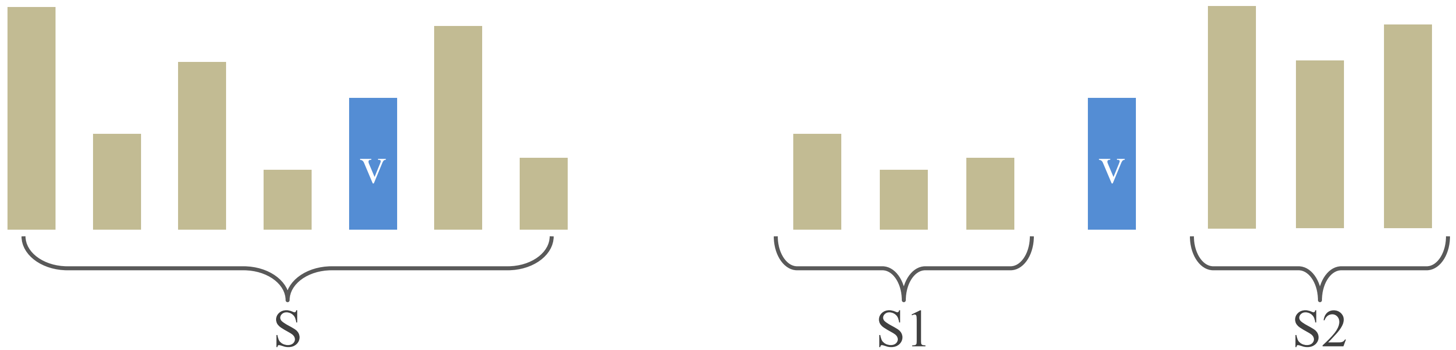
- Fastest sorting algorithm in practice
  - A lot of variations exist
- Not Stable
  - Average case cost: O(N log N)
  - Worst case cost: $O(N^2)$
    - But, the worst case seldom happens.
- Another divide-and-conquer algorithm

Quick sort, another divide-and-conquer algorithm

- **DIVIDE**
- **CONQUER**
- **COMBINE**

# Divide

- Pick an element $v$ in $S$
  - $v$ is called the pivot
  - Many ways to pick a pivot
- Partition $S-\{v\}$ into two disjoint groups
  - `S1 = {x ∈ S - {v} | x <= v}`
  - `S2 = {x ∈ S - {v} | x >= v}`
- Recursively divide $S1$ and $S2$



# Conquer

- If there is no more than 1 element in `S`, return directly.

# Combine

- No action is needed.
- The sorted `S1` (when the recursion is done) followed by $v$, followed by the sorted `S2` (when the recursion is done), make a sorted new list.

# Example

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Pick a pivot | 2 | 6 | 1 | 4 | 9 | 5 | 3 | 0 | 7 | 8 |
| Partition | 2 | 3 | 1 | 0 | 4 | 5 | 6 | 9 | 7 | 8 |
| Pick a pivot | 2 | 3 | 1 | 0 | 4 | 5 | 6 | 9 | 7 | 8 |
| Partition | 0 | 1 | 3 | 2 | 4 | 5 | 6 | 9 | 7 | 8 |
| Pick a pivot | Conquer | 1 | 3 | 2 | 4 | 5 | 6 | 9 | 7 | 8 |
| Partition | | 1 | 2 | 3 | 4 | 5 | 6 | 9 | 7 | 8 |

Conquer

The right half can be solved similarly

Nothing is done in conquer and combine

# “DIVIDE” IS THE KEY

# Animation

- [Animation]
- Note that
  - There are various methods to choose a pivot
  - There are various methods to partition a sub-array

# Pseudo Code

```
QUICKSORT(A, left, right)
    IF left >= right
        return
    q = PARTITION(A, left, right)
    //q is the position of the pivot
    QUICKSORT(A, left, q-1)
    QUICKSORT(A, q+1, right)
```

# Partitioning

- Partitioning
  - This is a key step of the quicksort algorithm
  - Goal: given the picked pivot, partition the remaining elements into two smaller sets
  - Many ways to implement how to partition
    - Even the slightest deviations may cause surprisingly bad results.
- We will learn an easy and efficient partitioning strategy here.
- How to pick a pivot will be discussed later

# Partitioning Strategy

Want to partition an array A[left .. right]

1. Get the pivot element out of the way by swapping it with the last element. (Swap pivot and A[right])
2. Let i start at the first element and j start at the next-to-last element
    1. i = left, j = right – 1

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Pick a pivot | 2 | 6 | 1 | 4 | 9 | 5 | 3 | 0 | 7 | 8 |
| Swap | 2 | 6 | 1 | 8 | 9 | 5 | 3 | 0 | 7 | 4 |
| | i | | | | | | | | j | |

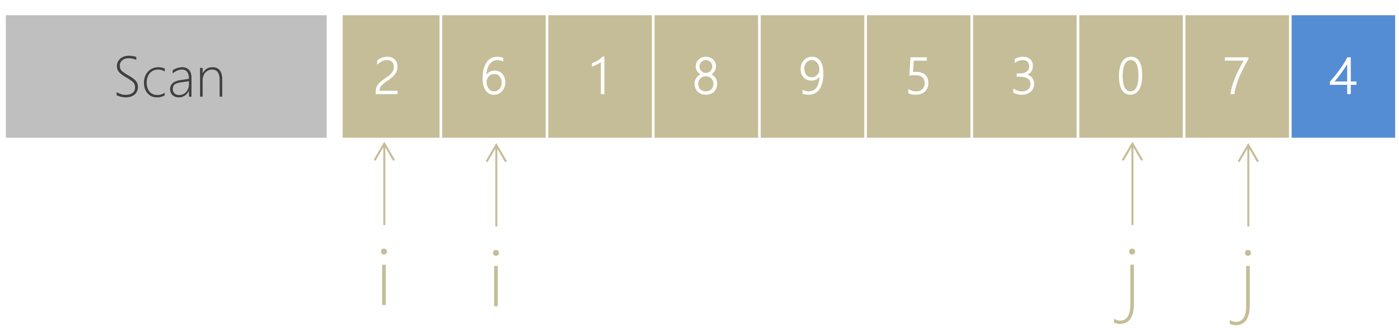
# Partitioning Strategy

Goal:

- A[left..i] are smaller or equal to the pivot
- A[j..right] are greater or equal to the pivot

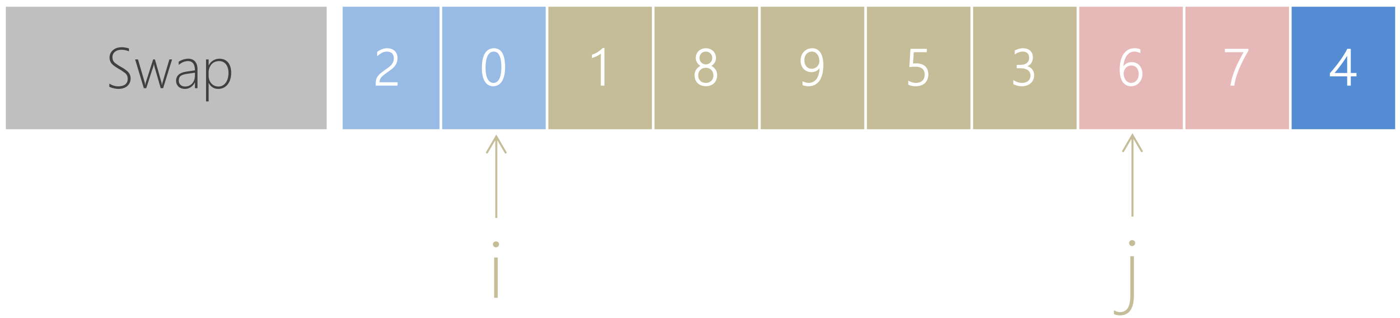
Strategy:

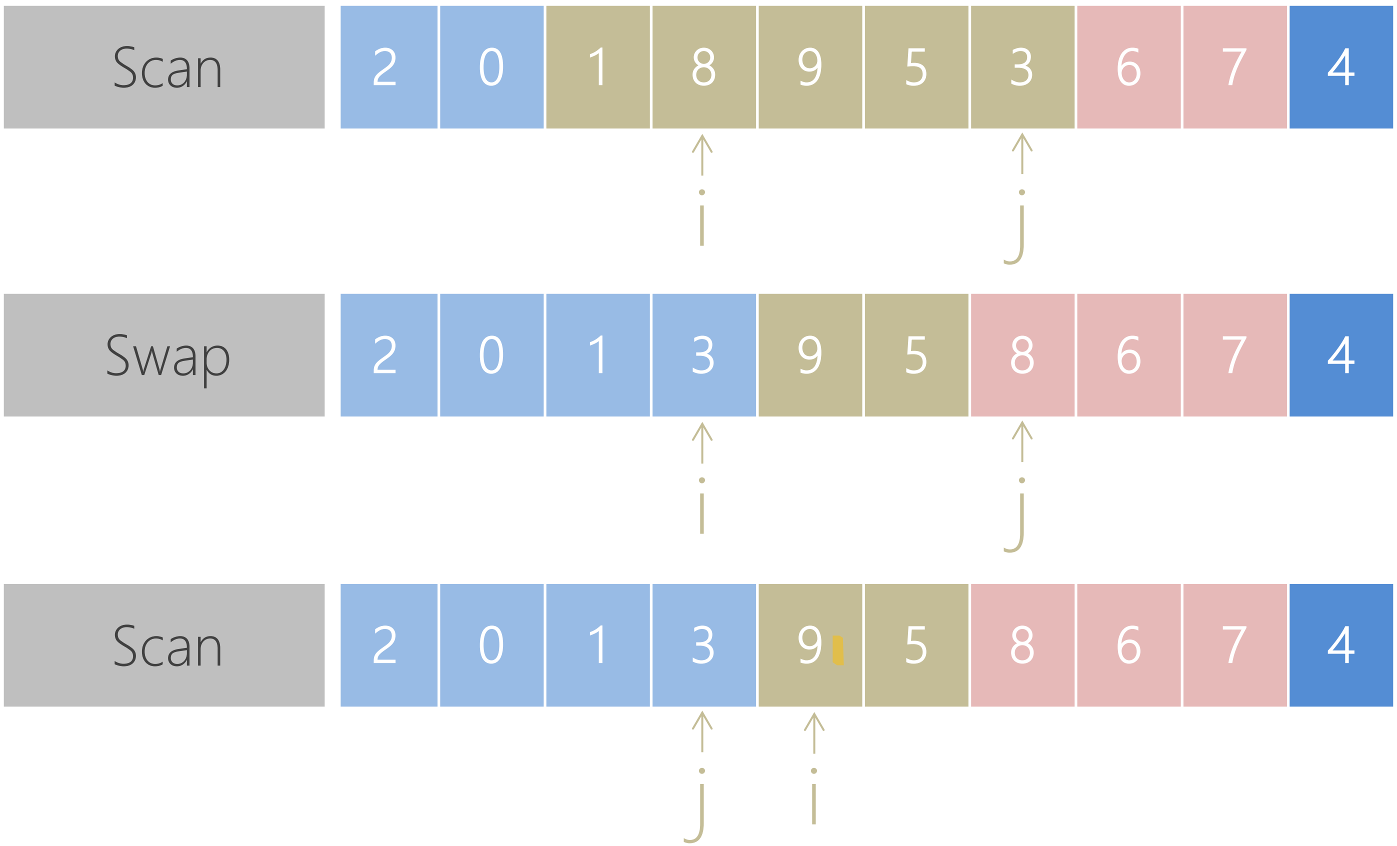
- When i < j
  - Move i right, skipping over elements smaller than the pivot
  - Move j left, skipping over elements greater than the pivot
  - When both i and j have stopped
    - A[i] >= pivot
    - A[j] <= pivot { A[i] and A[j] should now be swapped}



# Partitioning Strategy

- When i and j have stopped and i is to the left of j (thus legal)
  - Swap A[i] and A[j]
    - And then both elements are on the "correct" side
  - After swapping
    - A[i] <= pivot
    - A[j] >= pivot
  - Repeat the process until i and j cross

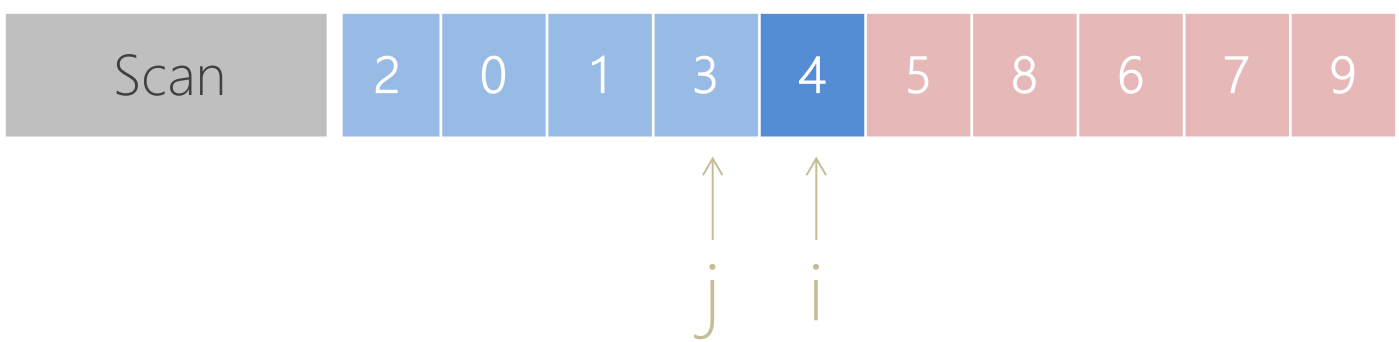





# i and j cross now!

# Partitioning Strategy

- When i and j have crossed
  - Swap A[i] and pivot
- Result:
  - A[p] <= pivot, for p < i
  - A[p] >= pivot, for p > I
- Partition complete



# Pseudo Code

```
PARTITION(A, left, right)
    p = PIVOT(A, left, right)
    //p is the position of the pivot
    swap A[p] and A[right]
    i = left, j = right-1, pivot = A[right]
    WHILE true
        WHILE i<right AND A[i]<pivot
            i++
        WHILE j>=left AND A[j]>pivot
            j--
        IF i<j
            swap A[i] and A[j]
            i++, j--
        ELSE
            BREAK
    swap A[i] and A[right]
```

# Small arrays

- For very small arrays, quicksort does not perform as well as insertion sort
  - how small depends on many factors, such as the time spent making a recursive call, the compiler, etc
- Do not use quicksort recursively for small arrays
  - Instead, use a sorting algorithm that is efficient for small arrays, such as insertion sort

# Quick Sort + Small Array Strategy

```
QUICKSORT(A, left, right)
    IF left >= right - 10
        INSERTIONSORT(A, left, right)
        RETURN
    q = PARTITION(A, left, right)
    //q is the position of the pivot
    QUICKSORT(A, left, q-1)
    QUICKSORT(A, q+1, right)
```

# Picking the **PIVOT**

# Strategy I

- 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 S2 (or S1)
    - this happens consistently throughout the recursive calls
    - Results in $O(n^2)$ behavior

# Strategy II

- Choose the pivot randomly
  - generally safe
  - random number generation can be expensive

# Strategy III

- Use the median of the array
  - The median is the middle element if the array is sorted. For example, if there are 9 elements in the array, the median is the $5^{th}$ largest one.
  - Partitioning always cuts the array into roughly half
  - An optimal quicksort: O(N log N)
  - However, expensive to find the exact median
    - e.g., sort an array to pick the value in the middle

# Strategy IV

- We will use median of three
  - Compare just three elements: the left most, right most 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 pivot is at second last position
    - **WHY?**

# Median3 Example

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| median3 | 2 | 6 | 1 | 4 | 9 | 5 | 3 | 0 | 8 | 7 |
| reposition | 2 | 6 | 1 | 4 | 7 | 5 | 3 | 0 | 8 | 9 |
| pick pivot | 2 | 6 | 1 | 4 | 7 | 5 | 3 | 0 | 8 | 9 |
| swap | 2 | 6 | 1 | 4 | 8 | 5 | 3 | 0 | 7 | 9 |
| | | ↑ i | | | | | | ↑ j | | |

# Partition With Median3

```
PARTITION(A, left, right)
    MEDIAN3(A, left, right)
    // MEDIAN3 repositions the left, center
    // and the right elements
    i = left+1, j = right-2, pivot = A[right-1]
    WHILE true
        WHILE A[i]<pivot
            i++
        WHILE A[j]>pivot
            j--
        IF i<j
            swap A[i] and A[j]
            i++, j--
        ELSE
            BREAK
    Swap A[i] and A[right-1]
```

No boundary check. Why?

# Quicksort Faster than Mergesort

- Both quicksort and mergesort take O(N log N) in the average case.
- Why is quicksort faster than mergesort?
  - The inner loop consists of an increment/decrement (by 1, which is fast), a test and a jump.
  - There is no extra juggling as in mergesort.

# Analysis

- Assumptions
  - Pivot Selection: Median of 3
  - Base Case: Array size <= 10
- Running time T(n)
  - Divide
    - Pivot selection: O(1)
    - Partitioning: O(n)
    - Recursive calls: T(i) + T(n-i-1)
      - i: number of elements in S1
  - Conquer and Combine: O(1)

$$T(n)=T(i)+T(n-i-1)+O(n)$$

# Worst-Case Analysis

- What will be the worst case?
  - The pivot is the smallest element, all the time
  - Partition is always unbalanced

$$
\begin{aligned}
T(N) &= T(N-1) + cN \\
T(N-1) &= T(N-2) + c(N-1) \\
T(N-2) &= T(N-3) + c(N-2) \\
&\vdots \\
T(2) &= T(1) + c(2) \\
T(N) &= T(1) + c\sum_{i=2}^{N} i = O(N^2)
\end{aligned}
$$

# Best-case Analysis

- What will be the best case?
  - Partition is perfectly balanced
  - Pivot is always in the middle (median of the array)

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2[2T(n/2^2) + n/2] + n \\
&= 2^2T(n/2^2) + 2n \\
&= 2^3T(n/2^3) + 3n \\
&= 2^iT(n/2^i) + i*n
\end{aligned}
$$

Let i=log(n),

$$
\begin{aligned}
&= nT(n/n) + n*log(n) \\
&= O(n*log(n))
\end{aligned}
$$

# Average-Case Analysis

- Assume
  - Each of the sizes for S1 is equally likely
- This assumption is valid for our pivoting (median-of-three) strategy
- On average, the running time is O(N log N)

Covered in

**DESIGN AND ANALYSIS OF ALGORITHMS**

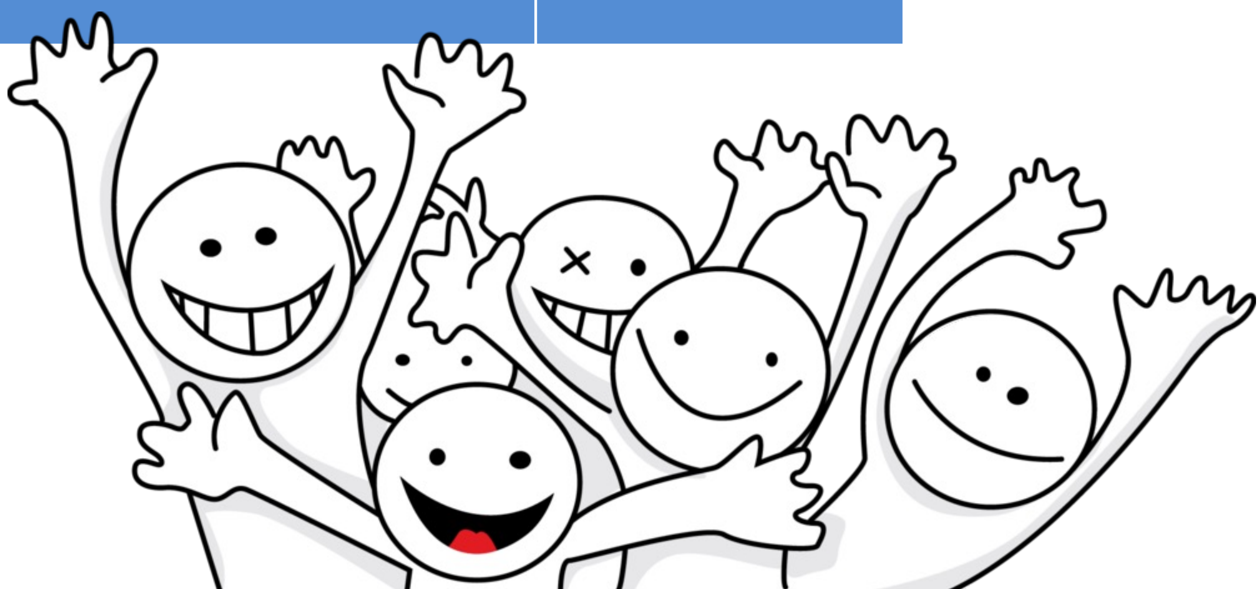
# Consider special cases

- When all elements are the same?
- Other cases?

# Analysis of Quick Sort

| | |
|---|---|
| Best-case Running Time | O(nlog(n)) |
| Worst-case Running Time | $O(n^2)$ |
| Average Running Time | O(nlog(n)) |

- Quick sort is not stable
- But it is the fastest in practice
- The worst case seldom happens



# Task

- Create a class, Sorting3, which includes at least the following static methods
  - public static void insertionSort(int[] A)
  - public static void mergeSort(int[] A)
  - Two overloading quickSort methods
    - public static void quickSort()
    - private static void quickSort(int[] A, int left, int right)
  - public static void main(String[] args)
- Auxiliary methods may be defined
- Submit Sorting3.java to iSpace

# Task

- public static void quickSort(int[] A)
  - It calls the recursive quickSort method to sort *A*
  - *quickSort(A, 0, A.length-1)*
- private static void quickSort(int[] A, int left, int right)
  - Sort sub-array *A*[*left..right*] recursively using quick sort
  - Pivot is picked using median-of-3
  - Switch to insertionSort for small arrays
    - Test on your computer and find the best BASE CASE for you
      - When shall we switch to insertionSort?
    - Overload the existing insertionSort so that it can sort a sub-array
      - private static void insertionSort(int[] A, int left, int right)

# Task

- public static void main(String[] args)
  - Generate an array *A1* consisting of $10^5$ random integers which are in range [0, 999]
  - Generate arrays *A2, A3* which are identical to *A1*
  - Sort *A1* using insertionSort, *A2* using mergeSort, and *A3* using quickSort
  - Print the elapsed time in milliseconds during which all the search functions run, respectively
- You can use the sample code on iSpace for existing methods