Lecture 8 - Randomized Algorithms and QuickSort

Fall 2025, Korea University

Instructor: Gabin An (gabin_an@korea.ac.kr)

Welcome Back! 👋

In today's class, we will:

- Quickly review the four sorting algorithms we've learned so far
- Introduce a new sorting algorithm: **QuickSort** 🚀
    - This will be our **first randomized algorithm** in this course!

Let's First Review Sorting Algorithms We've Learned

- SelectionSort
- BubbleSort
- InsertionSort
- MergeSort

SelectionSort

- Idea: Repeatedly find the smallest element from the unsorted part and move it to the front.

```
# Input: [5, 4, 1, 8, 7, 2, 6, 3]
[1, 4, 5, 8, 7, 2, 6, 3]  # 1 is the smallest, swap with 5
[1, 2, 5, 8, 7, 4, 6, 3]  # 2 is next smallest, swap with 4
[1, 2, 3, 8, 7, 4, 6, 5]  # 3 is next, swap with 5
[1, 2, 3, 4, 7, 8, 6, 5]  # 4 is next, swap with 8
[1, 2, 3, 4, 5, 8, 6, 7]  # 5 is next, swap with 7
[1, 2, 3, 4, 5, 6, 8, 7]  # 6 is next, swap with 8
[1, 2, 3, 4, 5, 6, 7, 8]  # 7 is next, swap with 8
```

- Selectionsort makes $n-1$ swaps, but always scans the rest of the array.

BubbleSort

- Idea: Repeatedly swap adjacent elements if they're in the wrong order. Largest values “bubble up” to the end.

```
# Pass 1
# [ unsorted part | sorted part ]
[5, 4, 1, 8, 7, 2, 6, 3|] # compare 5, 4 (swap) -> 4, 5
[4, 5, 1, 8, 7, 2, 6, 3|] # compare 5, 1 (swap) -> 1, 5
[4, 1, 5, 8, 7, 2, 6, 3|] # compare 5, 8 (do not swap) -> 5, 8
[4, 1, 5, 8, 7, 2, 6, 3|] # compare 8, 7 (swap) -> 7, 8
[4, 1, 5, 7, 8, 2, 6, 3|] # compare 8, 2 (swap) -> 2, 8
[4, 1, 5, 7, 2, 8, 6, 3|] # compare 8, 6 (swap) -> 6, 8
[4, 1, 5, 7, 2, 6, 8, 3|] # compare 8, 3 (swap) -> 3, 8
[4, 1, 5, 7, 2, 6, 3 | 8] # the largest element 8 'bubble up' to the end!
```

BubbleSort - Continued

```
[4, 1, 5, 7, 2, 6, 3 | 8] # After Pass 1
[1, 4, 5, 2, 6, 3 | 7, 8] # After Pass 2
[1, 4, 2, 5, 3 | 6, 7, 8] # After Pass 3
[1, 2, 4, 3 | 5, 6, 7, 8] # After Pass 4
[1, 2, 3 | 4, 5, 6, 7, 8] # After Pass 5
[1, 2 | 3, 4, 5, 6, 7, 8] # After Pass 6
[1 | 2, 3, 4, 5, 6, 7, 8] # After Pass 7
```

- Both SelectionSort and Buble Sort have quadratic running times, meaning that the number of operations performed on arrays of length n scales with n^2, i.e., $O(n^2)$.

InsertionSort

```python
def insertion_sort(A):
    for i in range(1, len(A)):
        current = A[i]
        j = i - 1
        while j >= 0 and A[j] > current:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = current
```

Example:

```
[5, 4, 2, 3]
```

MergeSort

```
       [6 5 12 10 9 1]
      /              \
  [6 5 12] divide [10 9 1]
   /    \          /     \
[6]    [5 12]   [10]    [9 1]
 |      /  \      |      /  \
[6]   [5]  [12] [10]    [9] [1]
 |      \   /     |      \   /
[6]    [5 12]   [10]     [1 9]
  \    /           \     /
 [5 6 12] combine  [1 9 10]
      \             /
      [1 5 6 9 10 12]
```

These are All DETERMINISTIC Sorting Algorithms!

- SelectionSort
- BubbleSort
- InsertionSort
- MergeSort

Each time you run the algorithm on the same input, it follows the exact same steps and produces the same output. There is **NO RANDOMNESS** involved! The behavior is completely **predictable** and **repeatable**.

QuickSort Overview

- We already know one **blazingly fast** sorting algorithm: **MergeSort**.
- So why do we need another?
  - Actually, **QuickSort** is highly competitive and often faster in practice.
- QuickSort is preferred in many libraries because it offers **better space complexity** than MergeSort.
  - QuickSort can run **in place**, using only a tiny amount of extra memory.
  - It operates directly on the input array via repeated **element swaps**, avoiding the need for auxiliary arrays.

Core Idea of QuickSort

QuickSort is very similar to the `select` algorithm we studied in the last class.

```
def quickSort(A):
    if len(A) <= 1:
        return A
    choose a pivot element p # to be implemented
    partition A around p # to be implemented
    recursively sort first part of A
    recursively sort second part of A
```

Example

`quickSort([3,8,2,5,1,4,7,6])`

Suppose the pivot is `4`

1. Partition around pivot `4`
    - `[3,2,1]`
    - `[8,5,7,6]`
2. So the array is conceptually split as:
    - `quickSort([3,2,1])` + `[4]` + `quickSort([8,5,7,6])`

Remaining To-Do List

1. ➡️ How do we implement the partitioning subroutine?
2. How should we choose the pivot element?
3. What's the running time of `QuickSort`?

Partitioning Around a Pivot Element - The Easy Way Out

```
def partition_simple(A, p):
    A_less, A_greater = [], []
    for item in A:
        if item < p:
            A_less.append(item)
        elif item > p:
            A_greater.append(item)
```

- It's easy to come up with a linear-time partioning subroutine if we don't care about allocating additional memory.
- How do we partition an array around a pivot element while allocating almost **no additional memory**?

In-Place Partitioning Implementation - The High-Level Plan

- Assume the pivot is the first element.
- Maintain the following structure during the scan:

```
| p |   <p    |     >p      |    unseen     |
```

- If we succeed with this plan, by the end of the scan, the array will look like:

```
| p |         <p         |         >p       |
```

- To complete the partioning, we can **swap** the pivot with the last element less than it:

```
|         <p         | p |         >p       |
```

Implementation for Partition (a.k.a., Lomuto Partition Algorithm)

```python
def partition(A, left, right):
    pivot = A[left]
    i = left + 1
    for j in range(left + 1, right + 1):
        if A[j] < pivot:
            A[i], A[j] = A[j], A[i]
            i += 1
    A[left], A[i - 1] = A[i - 1], A[left]
    return i - 1
```

- **Invariant**: all elements between the pivot and i are less than the pivot, all elements between i and j are greater than the pivot.
- Maintain the following structure during the scan:

```
left               i         j          right
| p |   <p     |     >p      |   unseen     |
```

Example:

```
A = [3,8,2,5,1,4,7,6]
pivot_index = partition(A, 0, len(A) - 1)
print(A, pivot_index)
```

Output: `[1, 2, 3, 5, 8, 4, 7, 6], 2`

```
```

New Pseudocode of QuickSort based on In-Place Partition

```
def quickSort(A, left, right):
    if left >= right:
        return A

    pivot_index = choose_pivot_index(A, left, right) # to-be-implemented

    A[left], A[pivot_index] = A[pivot_index], A[left] # make pivot first
    new_pivot_index = partition(A, left, right) # ✅ in-place partition!

    quickSort(A, left, new_pivot_index - 1)
    quickSort(A, new_pivot_index + 1, right)
```

Remaining To-Do List

1. ✅ How do we implement the partitioning subroutine?
2. ➡️ How should we choose the pivot element?
3. What's the running time of `QuickSort`?

The Importance of Good Pivots

- For QuickSort to be quick, it's important that **good** pivot elements are chosen.
- Example
  - `A = [3,8,2,5,1,4,7,6]`
    - If pivot is `5`: the subarrays are `[1,2,3,4]`, `[6,7,8]`.
    - If pivot is `8`: the subarrays are `[1,2,3,4,5,6,7]` and `[]`.
- Let's assume that we choose the k-th smallest element as pivot, then
  - $T(n) \leq cn + T(k-1) + T(n-k)$.
- For the worst pivot choice (the max or min element in the array, e.g., `1` or `8` in the example), the recurrence becomes $T(n) = T(n-1) + \Theta(n)$, i.e., $T(n) = \Theta(n^2)$.

What If the Array Is Already Sorted? ⚠️

Consider: `A = [1,2,3,4,5,6,7,8]`

- Choosing the first or last element as pivot leads to highly unbalanced partitions.
  - This results in worst-case performance, i.e., $\Theta(n^2)$.
- What if we use the exact median as pivot?
  - Finding the median requires running `select(A, k = len(A) // 2)`, i.e., $\Theta(n)$.
  - $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \Rightarrow T(n) = \Theta(n \log n)$.

Strategy	Pros	Cons
Arbitrary (First or Last)	Fast pivot selection: $O(1)$	Worst-case runtime: $O(n^2)$
True Median	Worst-case runtime: $O(n \log n)$	Slow pivot selection: $O(n)$

Can We Get the Best of Both Worlds? 🤔

- Idea: Use a **Random Pivot**!! 🎲
  - Select a random element from the array
  - Just as fast and simple as choosing the first or last element, i.e., $O(1)$.
  - But much more likely to land near the middle, **avoiding extreme cases**
- Why random pivot works?
  - Each element has an **equal chance** of being chosen (uniform distribution)
  - Very low probability of consistently picking the **worst-case** (min or max)

Randomized QuickSort

```python
import random

def randomizedQuickSort(A, left, right):
    if left >= right:
        return A

    pivot_index = random.randint(left, right) # ⬅️ random pivot selection

    A[left], A[pivot_index] = A[pivot_index], A[left] # make pivot first
    new_pivot_index = partition(A, left, right) # ✅ in-place partition!

    randomizedQuickSort(A, left, new_pivot_index - 1)
    randomizedQuickSort(A, new_pivot_index + 1, right)
```

Remaining To-Do List

1. ✅ How do we implement the partitioning subroutine?
2. ✅ How should we choose the pivot element?
3. ➡️ What's the **expected** running time of **randomized** `QuickSort`?

Randomized Algorithms

- Algorithms that **use randomness** in their logic
  - QuickSort with a random pivot is our first example!
- How do we analye a running time of randomized algorithm?

Expected Running Time

- We analyze the randomized algorithms using **expected** running time.
- Similar in spirit to worst-case analysis,

    i. We don't assume anything about the input.

    - The analysis therefore holds for **any input**.

    ii. We compute the average performance **over all possible random choices**.

    - e.g., pivot selections in QuickSort.

Expected Running Time of Randomized QuickSort

> **Proposition**: For every input array of size `n`, the expected running time of Quicksort is $O(n \log n)$.

- The runtime becomes a **random variable**, depending on pivot choices.
- We compute the **expectation** over all possible pivot sequences.
- Let C be the total number of element comparisons. Then:
    - All partitioning logic is based on element comparisons.
    - **Other operations are either linear or bounded by the number of comparisons.**
    - Hence, the total running time is: $O(\mathbb{E}[C] + n)$.

Comparison Probability

- Let z_i and z_j be elements in the sorted array, and define $X_{ij}(\sigma) = 1$ if they are compared, 0 otherwise for a given series of pivot choices σ.
- z_i and z_j are compared **at most once** during the entire execution because after the array is split using a pivot from $[z_i, \ldots, z_j]$, they can no longer be compared.
- In fact, they are compared **if and only if** one is the **first pivot** chosen from the subarray containing both elements.

$$\mathbb{P}[z_i \text{ and } z_j \text{ are compared}] = \mathbb{P}[z_i \text{ or } z_j \text{ is the first pivot picked from}[z_i, \ldots, z_j]] = \frac{2}{j-i+1}$$

- **Linearity of Expectation**: We exploit the fact: $\mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$
- We know $\#\ \text{comparisions} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}(\sigma)$, then:

$$
\begin{aligned}
\mathbb{E}\left[\#\ \text{comparisons}\right] &= \mathbb{E}\left[\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}(\sigma)\right] \\
&= \sum_{i=1}^{n} \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}(\sigma)\right] \\
&= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{E}\left[X_{ij}(\sigma)\right] \\
&= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{P}\left[X_{ij}(\sigma) = 1\right] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{P}[z_i \text{ and } z_j \text{ are compared}]
\end{aligned}
$$

Bounding the Expected Comparisons

$$
\begin{aligned}
\mathbb{E}\left[\#\text{ comparisons}\right] &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{P}[z_i \text{ and } z_j \text{ are compared}] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= 2\sum_{i=1}^{n} \left(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-i+1}\right) \leq 2\sum_{i=1}^{n} \left(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}\right) \\
&= 2\sum_{i=1}^{n} \ln n \quad \left(\because \sum_{k=2}^{n} \frac{1}{k} \leq \ln n\right) \\
&= 2n \ln n
\end{aligned}
$$

- Hence: $\mathbb{E}[\text{runtime}] = O(\mathbb{E}[C] + n) = O(n \log n)$

✅ Quicksort has **expected** $O(n \log n)$ time **for any input**, thanks to **randomization**.

Sorting Lower Bounds

- The sorting algorithms we have learned in this course (i.e., SelectionSort, BubbleSort, InsertionSort, MergeSort, QuickSort) are all **COMPARISON-based** sorting algorithms!
  - i.e., they sort an array via asking *whether a given element is greater than, less than, or equal to some other element*.
- For such algorithms, there exists theoretical lower bounds for running time.
  - Any correct algorithm (even a randomized one!) will require $\Omega(n \log n)$.
  - Why? Please refer to Avrim Blum's notes on sorting lower bounds
  - We'll also prove it together next week!

Credits & Resources

Lecture materials adapted from:

- Stanford CS161 slides and lecture notes
  - https://stanford-cs161.github.io/winter2025/
- *Algorithms Illuminated* by Tim Roughgarden
  - https://algorithmsilluminated.com/

Image Source:

- QuickSort Visualization: https://favtutor.com/blogs/quick-sort-cpp

✅ Why are comparisons the dominant cost?

- In each recursive call, Quicksort selects a pivot and **partitions** the array into two parts: elements `< pivot` and elements `> pivot`.
- Partitioning requires **scanning the array once**, performing $O(k)$ **work** for an array of size k.
- The majority of this work includes:
  - **Comparing elements to the pivot** — these are the key operations we count.
  - **Other operations** (e.g., swapping or rearranging elements) are constant-time per element and thus linear overall, or **bounded by the number of comparisons**.