Beirut Arab University
Faculty of Science
Mathematics and Computer Science Department

Course: Fundamentals of Algorithms
Semester: Spring 2024-2025
Lab 5

Randomized Algorithms and Quick Sort

Objectives:

1. **Understand Randomized Algorithms:** Learn how randomness can optimize algorithm efficiency.
2. **Implement QuickSort with Random Pivoting:** Explore QuickSort with random pivot selection for optimal performance.
3. **Analyze Complexity:** Compare the expected and worst-case runtimes of QuickSort.

In randomized algorithms, some decisions are made randomly, aiming to achieve good performance on average rather than worst-case guarantees. QuickSort, when using a random pivot, is a classic example. It has an expected time complexity of O(nlog n)O(n \log n)O(nlogn) while still handling large arrays efficiently in practice.

Exercise 1: Implement Randomized QuickSort

In this exercise, we implement QuickSort with a random pivot to achieve a more balanced partitioning, minimizing the likelihood of worst-case scenarios.

Steps:

1. **Pivot Selection:** Select a pivot randomly within the array range.
2. **Partitioning:** Partition the array around the pivot so that elements on the left are smaller and on the right are larger.
3. **Recursive Sort:** Recursively apply QuickSort to the left and right subarrays

```
RandomizedQuickSort(arr, low, high)
    if low < high then
        pivotIndex = RandomizedPartition(arr, low, high)
        RandomizedQuickSort(arr, low, pivotIndex - 1)
RandomizedQuickSort(arr, pivotIndex + 1, high)
```

```
RandomizedPartition(arr, low, high)
    pivotIndex = Random(low, high)  // Select a random index between low
and high
    swap(arr[pivotIndex], arr[high])  // Move the random pivot to the end
    return Partition(arr, low, high)

Partition(arr, low, high)
    pivot = arr[high]  // Use the last element as the pivot
    i = low - 1
    for j = low to high - 1 do
        if arr[j] <= pivot then
            i = i + 1
            swap(arr[i], arr[j])  // Place element smaller than pivot to
the left
    swap(arr[i + 1], arr[high])  // Place pivot in its correct position
    return i + 1  // Return the pivot index
swap(x, y)
    temp = x
    x = y
    y = temp
```

1. **randomizedPartition:** Chooses a random pivot and swaps it to the end of the array, where the partition method expects it.
2. **partition:** Rearranges the array so that elements less than the pivot are on the left and elements greater are on the right.
3. **quickSort:** Recursively sorts each partitioned subarray.

Example on an initial array `[10, 7, 8, 9, 1, 5]` with target sort in ascending order:

```
1. Initial Array: [10, 7, 8, 9, 1, 5]
   Random pivot selected (e.g., pivot = 7)
2. Partition around pivot:
   [5, 1] | 7 | [10, 8, 9]  (pivot index 2)
3. Recursive Sort on [5, 1]
   - Result after sort: [1, 5]
4. Recursive Sort on [10, 8, 9]
   - Result after sort: [8, 9, 10]
Final Result: [1, 5, 7, 8, 9, 10]
```

Exercise 2:

QuickSort with Lomuto Partition (Last Element as Pivot)

In this exercise, you will implement the QuickSort algorithm using Lomuto's partitioning scheme, where the pivot is chosen as the last element of the array. The goal is to understand how elements are rearranged based on the pivot and how QuickSort recursively sorts the subarrays.

Steps

1. **Initialize Pointers:**
2. **Set two pointers, i and j.**

i tracks where elements smaller than the pivot should go, while j iterates through each element in the array.

- **Partitioning:**
  - Choose the last element as the pivot.
  - As j moves through the array, whenever it encounters an element smaller than the pivot, swap it with the element at index i, then increment i.
  - Once all elements have been checked, swap the pivot (last element) with the element at index i. This places the pivot in its final sorted position.

3. **Recursive Sort:**

Apply QuickSort recursively to the left and right partitions created by the pivot.

```
QuickSort(arr, low, high)
    if low < high then
        pivotIndex = LomutoPartition(arr, low, high)
        QuickSort(arr, low, pivotIndex - 1)
        QuickSort(arr, pivotIndex + 1, high)
LomutoPartition(arr, low, high)
    pivot = arr[high]  // Choose the last element as pivot
    i = low - 1

    for j = low to high - 1 do
        if arr[j] <= pivot then
            i = i + 1
            swap(arr[i], arr[j])  // Place element smaller than pivot to the left
    swap(arr[i + 1], arr[high])  // Place pivot in its correct position
    return i + 1  // Return the pivot index
swap(x, y)
    temp = x
    x = y
    y = temp
```

```
Example Walkthrough
Let's use the array [8, 7, 1, 3, 5, 6, 4] to demonstrate how Lomuto's
partitioning scheme works when 4 is chosen as the pivot (as shown in the
image).
Initial Setup:
```

```
Array: [8, 7, 1, 3, 5, 6, 4]Pivot: 4 (last element)
Initialize i = -1 and j = 0.
Partition
 Steps:
Iterate through the array with j from 0 to high - 1.
Compare each element to the pivot (4):
If arr[j] <= pivot, increment i and swap arr[i] with arr[j].
After the loop, place the pivot at arr[i + 1].
Result:
After partitioning, the array becomes [1, 3, 4, 7, 5, 6, 8], with 4 placed in
its final sorted position.
```

Question:

In QuickSort, choosing a pivot can significantly impact the algorithm's performance. Compare the following two pivot selection strategies:

1. **Random Pivot Selection**: The pivot is chosen randomly from the array. (Part i)
2. **Last Element as Pivot (Lomuto's Partition)**: The last element of the array is chosen as the pivot. (part ii)

Discussion Points:

- **What are the expected time complexities** for each strategy on a randomly ordered array?
- **How would the worst-case time complexity** differ if the array is already nearly sorted or reverse sorted?
- **Why does using a random pivot generally lead to better average-case performance** than always choosing the last element?