Practical 6: Advanced Sorting Algorithms

What am I doing today?

Today's practical focuses on:

1. Implement Quicksort from pseudo-code
2. Develop a separate Enhanced QuickSort algorithm
3. ***Optional*** Assess the performance difference between QuickSort and Enhanced QuickSort

Instructions

Try all the questions. Ask for help from the demonstrators if you get stuck.

QuickSort Algorithm Implementation

Let's start by implementing a version of the Quick Sort algorithm (using the pseudo-code below) that sort values in ascending order.

Remember Quick Sort is an algorithm that takes a divide-and-conquer approach. The steps are:

1. Pick an element, called a pivot, from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

To implement QuickSort you will need to create two functions: A recursive implementation of QuickSort typically involves two functions: 1) the recursive QuickSort function and 2) a partition function. The key function really is the partition function which given an array and a

pivot element, it should output with the pivot element in its correct position with all smaller elements before it and all greater elements after it.

There are many different ways to pick the pivot in QuickSort including:

1. Always pick the first element as the pivot.
2. Always pick the last element as the pivot
3. Pick a random element as the pivot
4. Pick the median from a random selection as the pivot

Pseudocode for QuickSort:

```
/* array is the input to be sorted, start is the starting index,  end is the end index */
quickSort(array[], start index, end index)
{

//base case for this recursive function  If
(start < end){

 /* piv is the partitioning index */
piv = partition(array, start, end);
// sort before piv
quickSort(array, start, piv - 1);
// sort after piv
quickSort(array, piv + 1, end);
```

```
}
}
```

Pseudocode for Partitioning:

```
/* */

    partition (Array, start index, end index) {

    //pivot (element to be placed the correct position pivot
    = Array[end];
    pi = start;

    for (i = start, i <= end-1; i++){
    if (Array[i] <= pivot){ swap
    Array[pi] with A[i]
    pi++
    }
    }

    swap Array[i] with Array[end]
    return pi;
    }
```

Part 2

Write a second version of QuickSort (you can call it **enhancedQuickSort**) that implements the two improvements that we covered in the lecture:

1) Similar to your mergesort implementation last week, add a cutoff for small subarrays (e.g., <10 but you can try a few different cutoffs) and use the insertion sort algorithm you wrote before to handle them. ***Note** We can improve most recursive algorithms by handling small cases differently.

   Pseudo-code example:
   ```
   if (hi <= lo + CUTOFF) {
   insertionSort(array, lo, hi);
        return;
    }
   ```

2) As we saw in the lecture, random shuffling the input array first improves performance and protects against the worse case performance. Add a shuffle function that takes the input array and shuffles the elements. You can use the helper shuffle algorithms in the repo.
3) As we saw in the lecture, choosing a partition where the value is near the middle or exactly the middle of the elements in the arrays values means our sort will perform better. In quicksort with median-of-three partitioning the pivot item is selected as the median between the first element, the last element, and the middle element (decided

using integer division of n/2). In the cases of already sorted lists this should take the middle element as the pivot thereby reducing the inefficiency found in normal quicksort.

Look at the first, middle and last elements of the array, and choose the median of those three elements as the pivot (e.g,, int median = medianOf3(a, lo, lo + (hilo)/2, hi);

Part 3 Optional

Compare the performance of your QuickSort and QuickSortEnhanced on a range of inputs (N= 10, 1000, 10000, 100000 etc.) and graph the results of your experiments.

Input Size	Quick Sort	Quick Sort Enhanced
10	**0.0**	**0.001**
100	**0.0**	**0.001**
1000	**0.001**	**0.001**
10000	**0.002**	**0.03**
100000	**0.018**	**1.5**