Practical 5: Advanced Sorting Algorithms

What am I doing today?
Today's practical focuses on:
1. Implementing Mergesort from pseudo-code and comparing Mergesort to Insertion Sort for increasing input sizes
2. Implementing enhanced Mergesort
3. Compare the performance between 3 sorting algorithms

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

*Don't forget to pull the latest update from the github classroom repository to start

Quick Questions

1. Mergesort guarantees to sort an array in _______ time, regardless of the input:

   A. Linear time
   B. Quadratic time
   C. Linearithmic time
   D. Logarithmic time

2. The main disadvantage of MergeSort is:

   A. It is difficult to implement
   B. It uses extra space in proportion to the size of the input
   C. It is an unstable sort
   D. None of the above

3. Merge sort makes use of which common algorithm strategy?

   A. Dynamic Programming
   B. Branch-and-bound
   C. Greedy approach
   D. Divide and conquer

4. Which sorting algorithm will take the least time when all elements of the input array are identical?

   A. Insertion Sort
   B. MergeSort
   C. Selection Sort
   D. Bogo Sort

5. Which sorting algorithm should you use when the order of input is not known?

   A. Mergesort
   B. Insertion sort
   C. Selection sort
   D. Shell sort

Algorithmic Development

Part 1

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

Merge Sort divides the input array into 2 parts. There are 2 main components to the function: merge() and sort() *review your lecture notes if you need to refresh your memory

Your merge function assumes the input arrays are sorted and then merges them into one array.

The sort function is a recursive call that divides the array down and down until it reaches the base case and then sorts as it returns.

First implement merge (pseudocode):

```
Function merge(array a, array aux, int lo, int mid, int hi){

//copy the array a to an aux array
For loop through k {
Aux[k] = a[k];
}
//merge back to a[]
I = lo
J = id+1

For loop to merge a[] back
{
//insert your code here
}
```

Second implement the recursive sort (pseudocode):

```
Function sort (array[] array, int l, int r){
If (l < r) {
```

```
//find the middle point
M = l + (r-1)/2;
//sort first and second halves
Sort (array, l, m);
Sort (array, m+1, r);

//merge the sorted halves
Merge(array, l, m, r);
}
```

Part 2

Write a second version of MergeSort that implements the two improvements to mergesort that we covered in the lecture:

1) add a cutoff for small subarrays and use insertion sort (written last time) to handle them. We can improve most recursive algorithms by handling small cases differently.

Pseudo-code:

```
if (hi <= lo + CUTOFF) {
    insertionSort(dst, lo, hi);
        return;
     }
```

2) test whether the array is already in order. We can reduce the running time to be linear for arrays that are already in order by adding a test to skip call to merge() if a[mid] is less than or equal to a[mid+1]. **In other words, if the last element in the first sorted array is less than or equal to the first element in the second sorted array then you can just add the entire second array in without the need for comparisons.** With this change, we still do all the recursive calls, but the running time for any sorted subarray is linear.

Part 3

Compare the performance of Insertion Sort, MergeSort and MergeSortEnhanced on a range of inputs (N= 10, 1000, 10000, 100000 etc.).

Solution note: Switching to insertion sort for small subarrays will improve the running time of a typical mergesort implementation by 10 to 15 percent.