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.

Grading: Remember if you complete the practical, add the code to your GitHub repo which needs to be submitted at the end of the course **for an extra 5%**

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.

***please add this function to your class of sorts that you created last week.**

First implement mergeSort:

```
function mergeSort (int[] a){

N = array.length;

//base case
if (n == 1){
return array;
}

//create left and right sub-arrays
left = mergeSort(left);
right = mergeSort(right);

mergeArray = merge(left, right);

return mergedArray;
}
```

Second implement the recursive merge:

```
function merge (int[] a, int[] b){

//repeat while both arrays have elements in them
while (a.notEmpty() && b.notEmpty()){

//if element in 1st array is <= 1st element in 2nd array
if (a.firstElement <= b.firstElement){
S.insertLast(a.removeFirst());
} else if (b.firstElement <= a.firstElement){
S.insertLast(b.removeFirst());
}

//when while loop ends
If (a.notEmpty()){
//add remaining elements in a to S
} else if (b.notEmpty()){
//add remaining elements in b to S
}

return S;
```

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;
         }
   ```

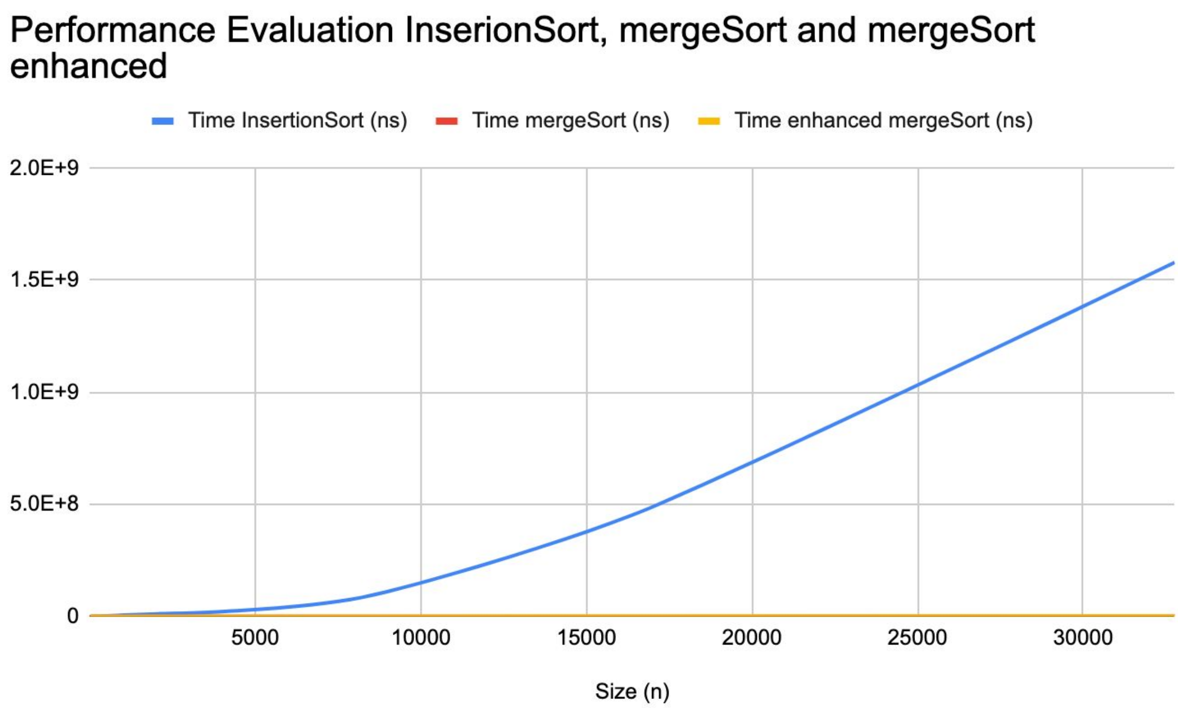
2) test whether the array is already in order. We can reduce the running time to be linear for arrays that are already in order by adding a test to skip call to merge() if a[mid] is less than or equal to a[mid+1]. **In other words, if the last element in the first sorted array is less than or equal to 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.

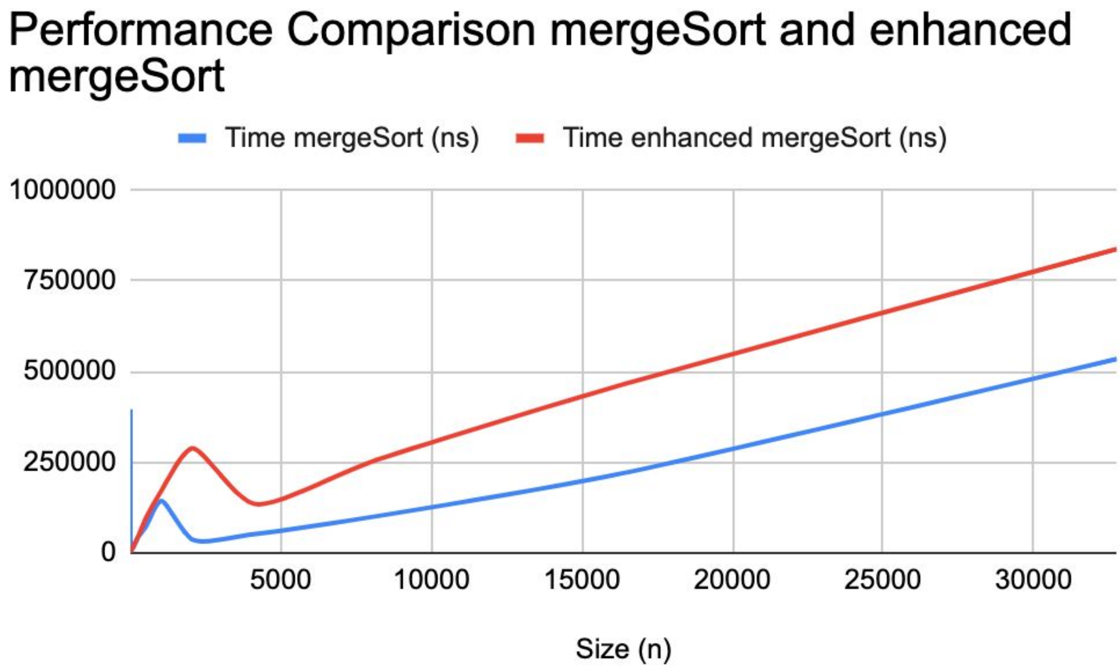
Part 3

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.

Size (n)	Time InsertionSort (ns)	Time mergeSort (ns)	Time enhanced mergeSort (ns)
4	624737	395229	9936
8	6076	1609	2116
16	22325	2590	3119
32	27400	4511	5685
64	98970	9622	10565
128	374427	17694	22420
256	1188730	42130	42301
512	1794757	70379	93229
1024	5543784	142491	168956
2048	11071865	36503	287633
4096	21722053	50527	134666
8192	83689021	100408	255879
16384	451879859	219377	464458
32768	1579829275	534439	837957





We can clearly see in the first graph that insertion sort is an asymptotic magnitude above mergeSort. Rather more interesting is the second graph where we see a translation of the function when enhancing the algorithm which is related with the 10-15% enhancement commented above.