CS 6301 Section 16

Implementation of Advanced DS&A

Project 1 : Empirical Study of Merge Sort

Submitted By : Arjun Gopal

Net ID : axg145630

Project Requirement

Merge sort to be implemented in three different methods which are

1. Simple Merge Sort by allocating dynamic memory for L and R (where L and R are two arrays for storing the left and right side of the array during merging) - MergeSortA
2. Creating a single auxiliary array B and passing it to the MergeSort and Merge. In each instance of Merge , data is copied from A to B and merged back to A. – MergeSortB
3. Creating a single auxiliary array B and pass it to the MergeSort and Merge. Data alternates between A and B in alternate recursion steps. – MergeSort C

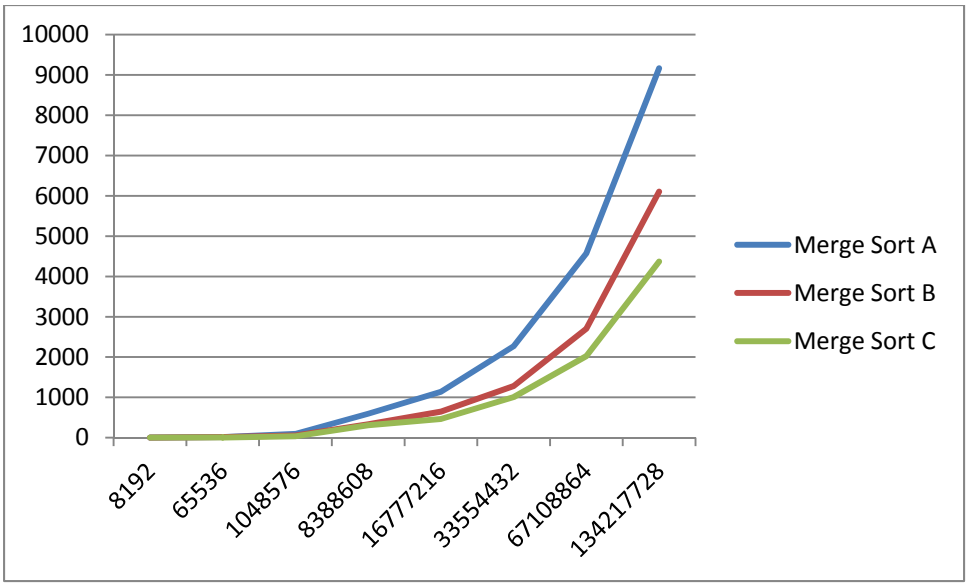
Specifications

Program takes one integer from the command line (n) , creates an array of that size and populates it with numbers in reverse order. It sorts the array using merge sort and verifies whether the array is in sorted order. It prints just one line of output indicating whether the algorithm succeeded in sorting the array.

Running Time of Each Merge Sort

MergeSort A is the implementation of first requirement, MergeSort B for the second one and MergeSortC being the implementation of the third requirement.

Input	Time Taken (ms)		
n	**Merge Sort A**	**Merge Sort B**	**Merge Sort C**
8192	2	3	2
65536	14	14	4
1048576	95	56	36
8388608	594	331	307
16777216	1140	645	463
33554432	2269	1280	1009
67108864	4573	2704	2025
134217728	9165	6104	4371



X Axis denotes the input test cases and Y axis denotes the time taken for each test cases.

Eligibility for the Extra Credit

The third Merge Sort (Merge Sort) which is attached with this report may be considered for the extra credit .

MergeSort B has to use an auxiliary array B which is passed to the Sort function and data should alternate between A (main array) and B. This avoids the copying of data between auxiliary array and main array each time. The challenge was to make this happen for all values of n. Following explains how this is achieved in this project.

An auxiliary array is created which is basically a clone of the main array and passed through the main sort function.

<u>The role of input and auxiliary array is switched in each recursive call.</u>

mergeSort(**int[] A,int[] B**,int low,int high)

{

If(low>=high)

Return;

mergeSort(**B,A**,low,mid) // Within each recursion A and B are reversed.

mergeSort(**B,A**,mid+1,high)

merge(**B,A**,low,mid,high) // Switching the control of B and A

}

and merge is defined like

void merge**(int[] A,int []B**,int low,int mid,int high)

{

//Merge from **A to B**

}

Basically when merge is called in each recursive steps data alternates between A and B. I have done the swapping when the arrays are passed to the recursive methods which seems to be faster than swapping with a temporary variable. This is achieved in a simple way without complicating the code.

Other Optimizations

1. Insertion sort is called if the length of array is less than 11 . (This speeds up the thing by avoiding more recursion steps.)
2. Before calling the merge, a check is done whether the array is sorted or not. If sorted, then return.

    If (A[middle+1]) > A[middle]) return;

    All these optimizations are done in the three variants of the merge sort.

Conclusion

All the three variants of the merge sort which was given as a part of the project requirement was implemented and a considerable variation in the time of execution was noticed from one to another, MergeSortC being the fastest. Almost 52 % improvement in time was achieved from MergeSortA to MergeSort C and 28% from MergeSort B to MergeSort C.