COMP 301 Analysis of Algorithms, Fall 2021
Instructor: Zafer Aydın
Lab Assignment 3

## Introduction

In this lab you will try to improve the running time of merge sort using insertion sort for small blocks. Submit your answers to the questions below in a text file (e.g. Word document). Name your file in name_surname.docx format. Submit your solution document and Java codes as a compressed folder (.zip, .rar) in name_surname format to Canvas.

## Background

The idea you will implement is based on Problem 2.1 of textbook. Instead of going till the end of the recursion tree where there is one element per leaf, you will go until a higher level where there are $n/k$ blocks each having $k$ elements. Then you will sort each of these blocks using insertion sort only at this base condition and merge those using the standard merge procedure of merge sort (i.e. merge them pairwise till all elements are merged into a single block). Insertion sort will be applied only at a single lowest level of the recursion. The rest will be sorted using the merge procedure.

***2-1 Insertion sort on small arrays in merge sort***

Although merge sort runs in $\Theta(n \lg n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to ***coarsen*** the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which $n/k$ sublists of length $k$ are sorted using insertion sort and then merged using the standard merging mechanism, where $k$ is a value to be determined.

## Assignment

1. The ***insertion sort*** algorithm is given below. You can use the code template `merge.java`, which includes the implementation of this algorithm in the method called `insertion_sort`. Alternatively you can also implement this algorithm yourself.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

where A is an aray of numbers and the array indices start from 1.

The merge sort procedure is also given below

COMP 301 Analysis of Algorithms, Fall 2021
Instructor: Zafer Aydın
Lab Assignment 3

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

You can use the code template for merge sort, which is available in `merge.java`. Alternatively you can also implement this method yourself (the version that does not use sentinels as in lab 2).

(a) Choose an integer array of size 65536. Initialize your array by random numbers from 0 to 99. Sort the numbers in this array using merge sort. Include the time it takes to sort these numbers to your report. You can use the commands given at the end of this assignment to measure the time in nano seconds.

(b) Modify the `merge_sort` procedure (rename it as `merge_sort_improved`) so that the method also includes an integer input parameter called `level`. This parameter will be incremented by 1 inside the `merge_sort` before the two recursive calls. It will keep track of the level index of the recursion tree. You can set this parameter to 1 before calling `merge_sort`. Based on this, the index for the first level of the recursion tree becomes

COMP 301 Analysis of Algorithms, Fall 2021
Instructor: Zafer Aydın
Lab Assignment 3

1.

(c) Modify the `merge_sort_improved` so that the method includes another input parameter called `max_level`. This parameter stores the maximum level index we should reach in the recursion tree. For instance if max_level is set to 2 then we should go until the second level of the recursion tree. Use this parameter to update the base case condition for terminating the recursive calls (i.e. no recursive calls should be made when the current level index is greater than or equal to this parameter).

(d) Set the `max_level` to 13. This will produce 4096 blocks each having 16 integers when the recursive calls reach that level of the recursion tree. Then modify `merge_sort_improved` so that when the method reaches the base case condition (i.e. level index is 13), the method will sort the blocks of 16 integers using insertion sort and then will call the merge procedure. Call the insertion sort only at the base condition of recursion. You can consider defining a second insertion sort method that also accepts the indices between which the input array will be sorted (this is given as the code template called `insertion_sort_2`). Run the `merge_sort_improved` method for the same input array as in part (a). Measure the running time and include it to your report. Can you get better running time as compared to part (a)?

(e) Repeat part (d) choosing different values for `max_level`. You can try values from 1 to 17. Which value gives the shortest running time? Include this to your report. Can you get better running time than parts (a) and (d)?

You can find the following Java commands useful in this lab

*To import class libraries*

```
import java.util.Arrays;

import java.util.Random;
```

*To generate and store random numbers in an array*

```
int data[ ] = new int[10];

Random rand = new Random();

rand.setSeed(System.currentTimeMillis());

for (int i = 0; i < data.length; i++)

 data[i] = rand.nextInt(100);
```

*To measure the running time put the following code before and after your selection sort block*

```
long startTime = System.currentTimeMillis();

/* Your selection sort block will go here */

long endTime = System.currentTimeMillis();

long elapsed = endTime - startTime;
```

or you can also use the nanoTime which could be more convenient:

```
long startTime = System.nanoTime();

/* Your sorting block will go here */

long endTime = System.nanoTime();

long elapsed = endTime - startTime;
```