SC2001/CE2101/CZ2101 ALGORITHM DESIGN AND ANALYSIS

Project 1: Integration of Mergesort & Insertion Sort

In Mergesort, when the sizes of subarrays are small, the overhead of many recursive calls makes the algorithm inefficient. Therefore, in real use, we often combine Mergesort with Insertion Sort to come up with a hybrid sorting algorithm for better efficiency. The idea is to set a small integer **S** as a threshold for the size of subarrays. Once the size of a subarray in a recursive call of Mergesort is less than or equal to **S**, the algorithm will switch to Insertion Sort, which is efficient for small-sized input.

(a) **Algorithm implementation:** Implement the above hybrid algorithm.

(b) **Generate input data:** Generate arrays of increasing sizes, in a range from 1,000 to 10 million. For each of the sizes, generate a random dataset of integers in the range of [1, …, x], where x is the largest number you allow for your datasets.

(c) **Analyze time complexity:** Run your program of the hybrid algorithm on the datasets generated in Step (b). Record the number of key comparisons performed in each case.

   i. With the value of **S** fixed, plot the number of key comparisons over different sizes of the input list n. Compare your empirical results with your theoretical analysis of the time complexity.

   ii. With the input size n fixed, plot the number of key comparisons over different values of **S**. Compare your empirical results with your theoretical analysis of the time complexity.

   iii. Using different sizes of input datasets, study how to determine an optimal value of **S** for the best performance of this hybrid algorithm.

(d) **Compare with original Mergesort:** Implement the original version of Mergesort (as learnt in lecture). Compare its performance against the above hybrid algorithm in terms of the number of key comparisons and CPU times on the dataset with 10 million integers. You can use the optimal value of **S** obtained in (c) for this task.

Merge Sort

```
mergesort (array a)
    if ( n == 1 )
      return a

    arrayOne = a[0] ... a[n/2]
    arrayTwo = a[n/2+1] ... a[n]

    arrayOne = mergesort ( arrayOne )
    arrayTwo = mergesort ( arrayTwo )

    return merge ( arrayOne, arrayTwo )
```

O(nlogn)

```
merge ( array a, array b )
    array c

    while ( a and b have elements )
      if ( a[0] > b[0] )
        add b[0] to the end of c
        remove b[0] from b
      else
        add a[0] to the end of c
        remove a[0] from a

    // At this point either a or b is empty

    while ( a has elements )
      add a[0] to the end of c
      remove a[0] from a

    while ( b has elements )
      add b[0] to the end of c
      remove b[0] from b

    return c
```

Insert Sort

```
for i : 1 to length(A) -1
    j = i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j = j - 1
```

$O(n^2)$

Comparison and Swaps