HACETTEPE UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING BBM204

Name : Mehmet Taha

Surname : Usta

Number : 21527472

E~mail : b21527472@cs.hacettepe.edu.tr

Subject : Analyzing the Complexity of Sorting Algorithms

Programming Language : JAVA

1.Problem Definition

Implement the selected 3 sort algorithm. See the time difference between these 3 sort algorithms according to given file and feature index number. if the last parameter is T, the program input file is sorted. The program saves the listed data on the input file. So that the big o notation concept can be easily understood. Reduce the type of conversion of the target data,shorten the running of sorting algorithms and save time

2.Average Result Tables

I run 5 times the number given on the program to increase the accuracy of the results. I took the average. All table results are in seconds. Processor is i5-4210U(2.4 ghz), ram size 8gb

NUMBER=11	TrafficFlow100	TrafficFlow1000	TrafficFlow50000	TrafficFlow100000	TrafficFlowAll
Selection Sort	0.0032	0.0446	5.541	29.9622	203.5958
Quick Sort	0.0018	0.04	2.1372	9.6164	33.4822
Merge Sort	0.0014	0.0032	0.0262	0.0368	0.0748

NUMBER=44	TrafficFlow100	TrafficFlow1000	TrafficFlow50000	TrafficFlow100000	TrafficFlowAll
Selection Sort	0.0028	0.0466	5.1516	29.1534	216.7808
Quick Sort	0.002	0.0164	3.1196	13.933	53.973
Merge Sort	0.0008	0.003	0.0288	0.0484	0.0982

NUMBER=79	TrafficFlow100	TrafficFlow1000	TrafficFlow50000	TrafficFlow100000	TrafficFlowAll
Selection Sort	0.0026	0.045	3.0794	15.2808	115.4902
Quick Sort	0.0028	0.0502	9.8742	48.7054	317.8722
Merge Sort	0.0012	0.0024	0.0248	0.0342	0.0636

3. Discussion about the time complexity and memory requirements of every algorithm you choose.

I chose selection sort,quick sort and merge sort.

Pseudo-code to for Selection Sort

A - an array containing the list of numbers
numitems - the number of numbers in the list

```
for i = 0 to numItems - 1
   for  j = i+1 to numItems
      if A[i] > A[j]
         // Swap the entries
         Temp = A[i]
         A[i] = A[j]
         A[j] = Temp
      End If
   Next j
Next i
```

Worst case performance for selection sort: O(n2)

Best case performance for selection sort: O(n2)

Average case performance for selection sort: O(n2)

Worst case space complexity for selection sort: O(n) total, O(1) auxiliary

Analysis for selection sort

Selecting the lowest element requires scanning all *n* elements (this takes *n*-1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning all *n*-1 elements and so on, for (*n*-1) + (*n*-2) + ... + 2 + 1 ($O(n^2)$) comparisons. Each of these scans requires one swap for *n*-1 elements.

Pseudo-code to for Quick Sort

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high){
   if (low < high)    {
      /* pi is partitioning index, arr[pi] is now
        at right place */
      pi = partition(arr, low, high);

      quickSort(arr, low, pi - 1);  // Before pi
      quickSort(arr, pi + 1, high); // After pi
   }
}
partition (arr[], low, high){
   // pivot (Element to be placed at right position)
   pivot = arr[high];
    i = (low - 1)  // Index of smaller element

   for (j = low; j <= high- 1; j++)    {
      // If current element is smaller than or
      // equal to pivot
      if (arr[j] <= pivot)       {
         i++;    // increment index of smaller element
         swap arr[i] and arr[j]
      }
   }
   swap arr[i + 1] and arr[high])
   return (i + 1)
}
```

Worst case performance for quick sort: O(n log(n))

Best case performance for quick sort: O(n log(n))

Average case performance for quick sort: O(n2)

Worst case space complexity for quick sort: O(logn)

Analysis for quick sort

When quicksort always has the most unbalanced partitions possible, then the original call takes cn cnc, n time for some constant c cc, the recursive call on n-

1 n−1n, minus, 1 elements takes c(n-1) c(n−1)c, left parenthesis, n, minus, 1, right parenthesis time, the recursive call on n-2 n−2n, minus, 2 elements takes c(n-2) c(n−2)c, left parenthesis, n, minus, 2, right parenthesis time, and so on.

Pseudo-code to for Merge Sort

```
merge(A, B):
  C = empty list

  While A and B are not empty:
    If the first element of A is smaller than the first element of B:
      Remove first element of A.
      Add it to the end of C.
    Otherwise:
      Remove first element of B.
      Add it to the end of C.

  If A or B still contains elements, add them to the end of C.
mergeSort(A):
  if length of A is 1:
    return A

  Split A into two lists, L and R.

  Q = merge(mergeSort(L), mergeSort(R))
  return Q
```

Worst case performance for merge sort: O(n log (n))
Best case performance for merge sort: O(n log (n)) typical
Average case peroformance for merge sort: O(n log (n))
Worst case space complexity for merge sort: O(n) total, O(n) auxiliary

Analysis for Merge sort

Assumption: N is a power of two.
For N = 1: time is a constant (denoted by 1)
Otherwise: time to mergesort N elements = time to mergesort N/2 elements plus
time to merge two arrays each N/2 elements.
Time to merge two arrays each N/2 elements is linear, i.e. N