Birla Institute of Technology & Science, Pilani
2nd Semester 2016-17 - CS F211 – Data Structures and Algorithms

Lab 5 – 15th Jan. 2017
Topics – Sorting and Practical Performance of Sorting Algorithms

1. Improving the performance of Quick Sort

Exercise 1:

The Quicksort procedure derived from a divide-and-conquer design is simple (*Version 1 – qs*):

```
qs(ls,lo,hi) {  if (lo<hi) { p=part(Ls,lo,hi,pivot(Ls,lo,hi)); qs(ls,lo,p-1); qs(ls,p+1,hi); } }

// Assumptions:
// (i) pivot returns the position (i.e. index) of the pivot element;
//  (ii) Postcondition for part: Ls[p] is the pivot; (Ls[j]<=Ls[p] for lo<=j<p) and (Ls[j]>Ls[p] for p<j<=hi)
```

Implement a simple quick sort algorithm on an array of student records, where each record must contain the following:

Student:

Name : single word (at most 20 characters)
Marks1 : double
Marks2 : double

You must sort the records based on marks of the students. Use the last element as the pivot. Also implement a partial quick sort algorithm as per the instructions given in the following table.

Key	Function	Input Format	Description
0	readData	0 M	Indicates that next M lines will contain data to be read. Each line will contain student's name followed by marks, separated by space.
1	partialQuickSort	1 t	Call quicksort but stop partitioning when size of list falls below t, and print the whole partially-sorted array. Print each name and marks (space separated) in a new line.
2	quicksort	2	Call complete quick sort and print final result.

Note:

You should call partialQuickSort and quickSort on separate copies of input list. A sample input list is given below.

Sample Input File:

```
0 12
RuOPVkrNoC      62.887092    47.739705
sAXVHLhFam      80.417675    13.723158
iugfZlARFp      52.428719    63.755227
zYpOBOuUos      91.902647    80.772452
dXBejIUsdb      6.309584     45.770174
mYUSnCtNAI      3.928034     53.160643
```

wWWLoMHsJi 88.007524 44.010453
RrlUsJXEIS 92.396979 43.955992
uQJVzlXVhT 43.195342 64.108060
GoOpxljoCv 90.680393 16.960709
1 4
2
-1

Exercise 2:

(i) Modify your QuickSort implementation such that QuickSort calls InsertionSort when the size of a sub list (as a result of partitioning) is less than a threshold t. [Note that t should be parameterized and taken from input file as per the below format– not hard-coded.]. Name this procedure QuickInSort.

(ii) Measure and compare the time taken by QuickSort and QuickInSort for large input sizes (at least a million records – for instance, one of those in-memory sorts in 2 above.).

(iii) Repeat step ii by starting with t=10 and increasing (say, by increments of 2) until performance of QuickInsort becomes better than that of QuickSort, peaks, and starts receding. [Note. The assumption is that there exists a single range of t values for which QuickInsort is better than QuickSort and that within the range QuickInsort displays monotonically increasing performance until a peak and then displays monotonically decreasing performance. End of Note.]

2. Measuring the maximum size of dynamically allocated array

The maximum possible size of an array that can be dynamically allocated can be estimated (approximately). Iteratively allocate (and free) memory dynamically until you incur failure while increasing the size: For e.g. start by allocating an array of size 1024; if allocation is successful, free it, and allocate an array of double the size; repeat the last step until allocation fails. The largest successful allocation is a limit on in-memory processing size (on the heap). Let this size be MAX_SIZE.

3. Sorting large files

In this exercise, you shall need to sort a very large file. The maximum size of an array which you can sort in memory has been found in the previous exercise. For practical implementation, assume MAX_SIZE = 10,000. Given a file of 1 Million student records containing name and marks for two subjects. You can generate this file using the following C program that takes the number of records as command line argument.

```
/* random_generator.c */
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
void rand_str(char *dest, size_t length) {
   char charset[] = "abcdefghijklmnopqrstuvwxyz"
                    "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

   while (length-- > 0) {
      size_t index = (double) rand() / RAND_MAX * (sizeof charset - 1);
      *dest++ = charset[index];
   }
   *dest = '\0';
}
```

```
double rand_double(double min, double max)
{
   return (double)rand()/(double)RAND_MAX * (max - min) + min;
}

int main(int argc, char *argv[])
{
        int N = 10;
        assert(argc == 2);
        int i = atoi(argv[1]);
        char *str = (char*) malloc (sizeof(char) * N);
        assert(str != NULL);
        //rand_str(str, N);
        while(i--)
        {
                rand_str(str, N);
                printf("%s\t%lf\t%lf\n", str, rand_double(0,100), rand_double(0,100));
        }
        return 0;
}
/* End of random_generator.c */
```

You shall need to sort all these records on **total marks** obtained by each student. You should follow the following steps in order to sort such large files.

Phase 1:

(i) Take the test file provided.

(ii) Allocate an array Ls of size MAX_SIZE.

(ii) Sort this file in two phases:

1. Phase I:
    a. j=0;
    b. Read input records totaling to MAX_SIZE from F and store them in Ls.
    c. Sort Ls using QuickSort (or QuickInSort – see below) and write the output onto a file, say out[j]
    d. j=j+1
    e. repeat the last three steps (i.e. b to d) until F becomes empty
2. Phase II:
    a. Repeatedly merge files two at a time until all records are in one sorted file.
    b. Note that this is merge sort where the atomic problem is sorting one file (because it is already sorted in Phase I).

(iv) Measure the time taken for sorting this entire file.

Note that this requires you to implement Merge Sort as well as Quick Sort. Let us call this procedure MergeQkSort.

4. Comparison of Performance

Compare the performance (time) of your implementation of MergeQkSort with the Unix/Linux command line *sort* routine for different input size.