Project 1

(Improving Performance of Quicksort)

Algorithms and Data Structures

(ITCS 6114 – 091)

Fall 2012

By

Abdul Aaquib

#800795370

10/01/2012

**Problem Statement**

One way to improve the performance of Quick Sort is to switch to Insertion Sort when a sub-file has <= M elements instead of recursively calling itself. Implement a recursive Quick Sort with a cutoff to Insertion Sort for sub-files with M or less elements. Include counters in the codes, empirically determine the value of M for which it performs fewest total number of key comparisons and key assignments on inputs of 10000 random natural numbers each less than K for K = 100, 1000, 10000, 100000. You should repeat the experiments 1000 times before making conclusions. Does the optimal value M depend on K? Prepare a project report to present your approach and finding. The source code listing should be documented and attached to your project report.

**Solution**

I started by first implementing the Quick Sort algorithm and the Insertion Sort algorithm separately. I then incorporated the Insertion Sort function into the Quick Sort program by modifying the program to call the insertion sort function whenever the number of elements left to be sorted is less than or equal to some value M.

**Quick Sort**

Quick Sort is a fast divide and conquer style sorting algorithm which is widely used in practice. It’s average expected running time is O(nlogn) which is much faster than other sorting algorithms for large quantities of data.

The pseudo code for Quick Sort is as follows.

Begin partition( a[], left, right)

x = a[left];

i = left – 1;

j = right + 1;

Begin do-while

Begin {do-while}

j--;

while (x <a[j])

End {do-while}

Begin {do-while}

i++;

while (x >a[i])

End {do-while}

if (i < j)

Begin

temp = a[i];

a[i] = a[j];

a[j] = temp;

End {If}

while (i < j)

End {do-while}

return j;

End {Partition}

Begin QuickSort(a[], left, right)

// left = subscript of beginning of array

// right = subscript of end of array

if(left<right)

Begin

pivot = partition(a, left, right);

QuickSort(a, left, pivot); // sort first section

QuickSort(a, pivot+1, right); // sort second section

End {If}

End {QuickSort}

**Insertion Sort**

Insertion Sort is a simple sorting algorithm which moves elements one at a time into their correct position. Insertion Sort can be very fast and efficient when used with smaller arrays and is actually used in practice to sort small arrays of data.  Unfortunately, it loses this efficiency when dealing with large amounts of data. On average, Insertion Sort has an overall complexity of O(n2).

The pseudo code for the Insertion Sort function is as follows.

Begin InsertionSort( array[])

for j = 0 to length(array)-1

do

key = a[j];

i = j - 1;

while i >= 0 and array[i] > key

do

a[i+1] = a[i];

i= i – 1;

end {while}

a[i+1] = key; //Put key into its proper location

end {for}

end {InsertionSort}

**Quick Sort with cut-off to insertion sort**

Quick Sort algorithm with a complexity of O(nlogn) is suitable for large quantities of data. However, when the data to be sorted is less, it is not the most efficient algorithm. Insertion Sort on the other hand is suitable and even outperforms Quick Sort for a small amount of data but it is inefficient for large quantities of data. Hence, we modify the Quick Sort algorithm to cut-off to insertion sort whenever the sub-file has less than or equal to M elements. This leads to a small performance improvement for Quick Sort while sorting a large amount of data.

The pseudo code for the modified Quick Sort function is as follows.

Begin QuickSortwithCutoff (a[], left, right)

// left = subscript of beginning of array

// right = subscript of end of array

If ((right-left) < M) //Cut-off to insertion sort if sub-array has <= M elements

Begin

InsertionSort (a, left, right);

return;

End {If}

Else if (left<right)

Begin

pivot = partition(a, left, right);

QuickSortwithCutoff (a, left, pivot); // sort first section

QuickSortwithCutoff (a, pivot+1, right); // sort second section

End {If}

End {QuickSort}

**The Experiment**

For getting the best performance out of this hybrid algorithm, it is necessary to choose an optimum value for M for which the algorithm is most efficient.

According to the problem statement, we have to empirically determine the value of M, for which it performs the fewest total number of key assignments and key comparisons on inputs of 10000 random natural numbers each less than K for K=100, 1000, 10000, 100000.

First, I create an integer array of 10000 elements and fill the array with random natural numbers each less than K=100, using the random number generator function rand().

I seed the rand() function with srand() to ensure that I get different random numbers every time I run the program.

int a[10000];

int K = 100;

srand( (unsigned) time(0) );

for (i=0;i<10000;i++)

a[i] = rand() % K;

Second, I add counters to the code to count the total number of key comparisons and key assignments performed by the algorithm. For every key comparison in the algorithm, I increment a comparison counter (c\_counter++) and for every key assignment, I increment an assignment counter (a\_counter++). At the end of the algorithm, I display the value of the counters and the total in the output window.

Then I record the total number of key assignments and key comparisons for different values of M and empirically determine the optimum value of M for which the total number of assignments and comparisons is the fewest.

I experiment with different values of M for each K and repeat the experiment many times to get a more optimum value of M for which the total number of key comparisons and key assignments is the least.

**Case 1: K=100**

I started with an array of 10000 elements with each element being a random natural number less than or equal to 100. Initially, I set M=1000 and this is the result I got:

M= 1000 K= 100

no. of assignments= 1319415

no. of comparisons= 1372861

Total= 2692276

Then I reduced M to 500, and the result now was:

M= 100 K= 100

no. of assignments= 61809

no. of comparisons= 159454

Total= 221263

The total number of key assignments and comparisons reduced significantly, so I further reduced M to 50 which gave the following output:

M= 50 K= 100

no. of assignments= 29283

no. of comparisons= 140436

Total= 169719

The total number of key assignments and comparisons reduced further, but the change was smaller. This indicated that the optimum value of M was close. Hence, I further reduced M to 20 which resulted in:

M= 20 K= 100

no. of assignments= 20146

no. of comparisons= 164143

Total= 184289

The total number of key assignments and key comparisons was higher than that for M=50. So, after performing a number of trials, I deduced that the optimum value should lie between 20 and 50.

Hence I repeated the experiment many times with different values of M between 20 and 50 and finally selected 35 as the optimum value of. One of the results for M=35, for which I got the least number of assignments and comparisons, is as follows:

M= 35 K= 100

no. of assignments= 23720

no. of comparisons= 138370

Total= 162090

Hence, for **K=100**, the optimum value of M was found to be **35.**

**Case 2: K=1000**

I initially set M to 500 and got the following output:

M= 500 K= 1000

no. of assignments= 849845

no. of comparisons= 911769

Total= 1761614

The total was very high so I reduced M to 50, then to 30, and then to 20. The total number of key assignments and comparisons kept reducing.

After performing many experiments for each value of M between 1 and 20, the total number of key assignments and comparisons were least for M=11.

Hence for **K=1000**, the optimum value of M was found to be **11**.

**Case 3: K=10000**

In the last 2 cases, the optimum value of M decreased when K increased. Hence, I guessed that the optimum value of M would be even lower for K=10000. Hence, after a lot of trials with different values of M less than 20, the total number of key assignments and comparisons were least for M=7.

Hence for **K=10000**, the optimum value of M was found to be **9**.

**Case 4: K=100000**

In this case, I observed that the rand() function does not generate very large random numbers. It cannot be used to generate numbers greater than 32767. However, I performed the following operation to generate random numbers upto 100000 :

data[j]=(rand()\*10 % K)+rand()%10

After, performing a number of experiments for different values of M between 1<=M<=15, I got the optimum value of M as 8.

Hence for **K=100000**, the optimum value of M was found to be **8**.

**Does value of M depend on K?**

Yes, the value of M depends on K, i.e. the optimum value of M below which cut-off to insertion sort occurs depends on the size of each element in the array (each element <= K). As the value K increases, the optimum value of M decreases.

Moreover, as the value of K increased from 10000 to 100000, the value of M decreased by 1. This shows that for very large values of K, the optimum value of M remains more or less the same.

**Code**

#include <fstream>

#include <iostream>

#include <time.h>

using namespace std;

int a\_counter=0 ,c\_counter=0;

void InsertionSort( long int a[], int left, int right)

{

int i, j, key;

for(j = left+1; j <= right; j++)

{

key = a[j]; a\_counter++;

for(i = j - 1; (i >= 0) && (a[i] > key); i--) // Smaller values move up

{

c\_counter++;

a[i+1] = a[i]; a\_counter++;

}

a[i+1] = key; //Put key into its proper location

a\_counter++;

}

}

int partition(long int a[], int left, int right)

{

int x = a[left]; a\_counter++;

int i = left - 1;

int j = right + 1;

int temp;

do

{

do

{

j--;

c\_counter++;

}while (x <a[j]);

do

{

i++;

c\_counter++;

} while (x >a[i]);

if (i < j)

{

temp = a[i];

a[i] = a[j];

a[j] = temp;

}

c\_counter++;

}while (i < j);

return j; // returns middle subscript

}

void Quick\_Ins\_Sort(long int a[], int left, int right,int ISORT\_CUTOFF)

{

// left = subscript of beginning of array

// right = subscript of end of array

int pivot;

if((right-left) < ISORT\_CUTOFF)

{

InsertionSort(a, left, right);

return;

}

else if(left<right)

{

pivot = partition(a, left, right); a\_counter++;

Quick\_Ins\_Sort(a, left, pivot,ISORT\_CUTOFF); // sort first section

Quick\_Ins\_Sort(a, pivot+1, right,ISORT\_CUTOFF); // sort second section

}

}

int main()

{

long int data[10000];

int i, average=0;

int K=100000;

int M;

ofstream myfile;

myfile.open("C:\\output(K=100000).txt", ios::out | ios::app); //output to file

srand ( (unsigned) time(0) );

for(M=1;M<=50;M++) //Experiment with different values of M

{

average=0;

for(int trial = 1; trial<=3; trial++) //perform 3 trials for each M

{

a\_counter=0;

c\_counter=0;

for(int j=0;j<10000;j++)

{

data[j]=(rand()\*10 % K)+rand()%10 ; // To generate random //numbers greater than //RAND\_MAX

}

Quick\_Ins\_Sort(data, 0, 9999, M);

average=average+a\_counter+c\_counter;

if (myfile.is\_open())

{

myfile << "\nM= "<<M;

myfile << "\tK= "<<K;

myfile << "\nno. of assignments= "<<a\_counter;

myfile << "\nno. of comparisons= "<<c\_counter;

myfile << "\nTotal= "<<a\_counter+c\_counter;

}

}

if(myfile.is\_open())

{

myfile<< "\naverage total = "<<average/3<<"\n";

}

}

myfile.close();

std::cout<<"Sorted Array \n";

for(i=0; i<10000; i++)

{

cout<<data[i]<<"\t";

if(i%10+1==0)

std::cout<<"\n";

}

std::cout<<"done";

getchar();

return 0;

}