 *DEPARTMENT OF COMPUTER ENGINEERING*

|  |  |
| --- | --- |
| Semester | S.E. Semester IV – Computer Engineering |
| Subject | Analysis of Algorithms |
| Lectures Professor In-charge | Prof. Ashwin Ganesan |
| Practicals Professor In-Charge | Prof. Ashwin Ganesan |
| Laboratory number | 314A |

|  |  |  |  |
| --- | --- | --- | --- |
| Student Name | Harsh Daga | | |
| Roll Number | 13106B0033 | | |
| Grade |  | Subject Teacher’s Signature |  |

|  |  |
| --- | --- |
| Experiment Number | 02 |
| Experiment Title | Time complexity of Bubble Sort, Selection Sort, Merge Sort, Merge Sort and Quick Sort. |
| Resources / Apparatus Used | Code Blocks IDE |
| Objectives (Skill Set/ Knowledge tested/imparted) | 1. To sort n random numbers starting from 10 to 100000 in increasing steps of multiples of 10. 2. To measure individual time taken by bubble sort, selection sort and insertion sort to arrange 10, 100, 1000, 10000 & 100000 numbers respectively. 3. To infer that Merge and Quick algorithms are poly logarithmic time algorithms. Their time complexity is O(n log n). |

**Description of the experiment:**

The input to the program below is n random numbers generated by a random number generator and stored in a file. The program takes n random numbers in steps of multiples of 10, starting from 10 going through 100,1000,10000,100000. At every step 5 algorithms sort these random numbers in ascending order one at a time. We keep track of system times before the sorting begins and after it for all the steps of n for all 5 algorithms. We then subtract these two system times and convert them to seconds to get the amount of time taken by the individual algorithms to sort the sets of numbers. We display the amount of time taken by each algorithm for different sizes of input.

**Program:**

#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>  
#include <string.h>  
#include <malloc.h>  
   
#define NUMELEMENTS 1000000  
#define NUMDIGITS 5  
const int VERIFY = 0;  
   
inline void swap ( int \*left, int \*right )  
{  
 int temp = \*left;  
 \*left = \*right;  
 \*right = temp;  
}  
   
// Bubble Sort method which returns time elapsed while sorting  
double bubblesort ( int a[ ], int n )  
{  
 clock\_t time1 = clock ( );  
 int i, j;  
 for ( i = 0; i < n - 1; i++ )  
 for ( j = 0; j<n - 1; j++ )  
 if ( a[ j ]>a[ j + 1 ] )  
 swap ( &a[ j ], &a[ j + 1 ] );  
 return (double) ( clock ( ) - time1 ) / CLOCKS\_PER\_SEC;  
}  
   
// Selection Sort method which returns time elapsed while sorting  
double selectionsort ( int a[ ], int n )  
{  
 clock\_t time1 = clock ( );  
 int i, j, min, pos;  
 for ( i = 0; i < n - 1; i++ )  
 {  
 min = a[ pos = i ];  
 for ( j = i + 1; j < n; j++ )  
 if ( a[ j ] < min )  
 min = a[ pos = j ];  
 a[ pos ] = a[ i ];  
 a[ i ] = min;  
 }  
 return (double) ( clock ( ) - time1 ) / CLOCKS\_PER\_SEC;  
}  
   
// Insertion Sort method which returns time elapsed while sorting  
double insertionsort ( int a[ ], int n )  
{  
 clock\_t time1 = clock ( );  
 int i, j, val;  
 for ( i = 0; i<n - 1; i++ )  
 {  
 val = a[ i + 1 ];  
 for ( j = i; j >= 0; j-- )  
 {  
 if ( a[ j ]>val )  
 a[ j + 1 ] = a[ j ];  
 else  
 break;  
 }  
 a[ j + 1 ] = val;  
 }  
 return (double) ( clock ( ) - time1 ) / CLOCKS\_PER\_SEC;  
}  
   
void merge ( int x[ ], int lb, int m, int ub )  
{  
 int i = lb, j = m + 1, k = lb;  
 while ( i <= m && j <= ub )  
 if ( x[ i ] < x[ j ] )  
 x[ k++ ] = x[ i++ ];  
 else  
 x[ k++ ] = x[ j++ ];  
 while ( i <= m )  
 x[ k++ ] = x[ i++ ];  
 while ( j <= ub )  
 x[ k++ ] = x[ j++ ];  
}  
   
void mergesort\_worker ( int a[ ], int l, int u )  
{  
 if ( l < u )  
 {  
 int mid = ( l + u ) / 2;  
 mergesort\_worker ( a, l, mid );  
 mergesort\_worker ( a, mid + 1, u );  
 merge ( a, l, mid, u );  
 }  
}  
   
void quicksort\_worker ( int a[ ], int low, int high )  
{  
 int pivot, j, i;  
 if ( low < high )  
 {  
 pivot = i = low;  
 j = high;  
 while ( i < j )  
 {  
 while ( ( a[ i ] <= a[ pivot ] ) && ( i < high ) )  
 i++;  
 while ( a[ j ] > a[ pivot ] )  
 j--;  
 if ( i < j )  
 swap ( &a[ i ], &a[ j ] );  
 }  
 swap ( &a[ pivot ], &a[ j ] );  
 quicksort\_worker ( a, low, j - 1 );  
 quicksort\_worker ( a, j + 1, high );  
 }  
}  
   
void print\_times ( double times[ ], int n )  
{  
 int i, mul = 1;  
 for ( i = 0; i < n; i++ )  
 {  
 mul \*= 10;  
 printf ( "%d\t:\t%.3fs\n",  
 mul,  
 times[ i ] );  
 }  
}  
   
// Merge Sort method which returns time elapsed while sorting  
double mergesort ( int a[ ], int n )  
{  
 clock\_t time1 = clock ( );  
 mergesort\_worker ( a, 0, n );  
 return (double) ( clock ( ) - time1 ) / CLOCKS\_PER\_SEC;  
}  
   
// Quick Sort method which returns time elapsed while sorting  
double quicksort ( int a[ ], int n )  
{  
 clock\_t time1 = clock ( );  
 quicksort\_worker ( a, 0, n );  
 return (double) ( clock ( ) - time1 ) / CLOCKS\_PER\_SEC;  
}  
   
// Check if array is sorted  
int verify ( int a[ ], int n )  
{  
 int i;  
 for ( i = 0; i != n - 1; i++ )  
 if ( a[ i ] > a[ i + 1 ] )  
 return 0;  
 return 1;  
}  
   
int main ( )  
{  
 int \*rand\_collection, \*a,  
 check[ 5 ] = { 0 }, // To verify that the array is sorted  
 i, n = 1;  
 double  
 bubble\_times[ NUMDIGITS ],  
 selection\_times[ NUMDIGITS ],  
 insertion\_times[ NUMDIGITS ],  
 merge\_times[ NUMDIGITS ],  
 quick\_times[ NUMDIGITS ];  
 rand\_collection = (int\*) malloc ( NUMELEMENTS \* sizeof ( int ) );  
 a = (int \*) malloc ( NUMELEMENTS \* sizeof ( int ) );  
 srand ( (unsigned) clock ( ) );  
 for ( i = 0; i < NUMELEMENTS; i++ )  
 rand\_collection[ i ] = rand ( ) % 1000;  
 for ( i = 0; i < NUMDIGITS; i++ )  
 {  
 n = n \* 10;  
 memcpy ( a, rand\_collection, n \* sizeof ( int ) );  
 bubble\_times[ i ] = bubblesort ( a, n );  
 if ( VERIFY )  
 check[ 0 ] |= verify ( a, n );  
 memcpy ( a, rand\_collection, n \* sizeof ( int ) );  
 selection\_times[ i ] = selectionsort ( a, n );  
 if ( VERIFY )  
 check[ 1 ] |= verify ( a, n );  
 memcpy ( a, rand\_collection, n \* sizeof ( int ) );  
 insertion\_times[ i ] = insertionsort ( a, n );  
 if ( VERIFY )  
 check[ 2 ] |= verify ( a, n );  
 memcpy ( a, rand\_collection, n \* sizeof ( int ) );  
 merge\_times[ i ] = mergesort ( a, n );  
 if ( VERIFY )  
 check[ 3 ] |= verify ( a, n );  
 memcpy ( a, rand\_collection, n \* sizeof ( int ) );  
 quick\_times[ i ] = quicksort ( a, n );  
 if ( VERIFY )  
 check[ 4 ] |= verify ( a, n );  
 }  
 printf ( "Execution times for Merge Sort:\n" );  
 print\_times ( merge\_times, NUMDIGITS );  
 printf ( "Execution times for Quick Sort:\n" );  
 print\_times ( quick\_times, NUMDIGITS );  
 if ( VERIFY )  
 for ( i = 0; i < 5; i++ )  
 printf ( "%d\t", check[ i ] );  
 getchar ( );  
 return 0;  
}

**Output run:**

Execution times for Bubble Sort:

10 : 0.000s

100 : 0.001s

1000 : 0.008s

10000 : 0.952s

100000 : 101.710s

Execution times for Selection Sort:

10 : 0.000s

100 : 0.000s

1000 : 0.001s

10000 : 0.125s

100000 : 12.635s

Execution times for Insertion Sort:

10 : 0.000s

100 : 0.000s

1000 : 0.001s

10000 : 0.059s

100000 : 6.249s

Execution times for Merge Sort:

10 : 0.000s

100 : 0.000s

1000 : 0.000s

10000 : 0.001s

100000 : 0.015s

1000000 : 0.151s

Execution times for Quick Sort:

10 : 0.000s

100 : 0.000s

1000 : 0.001s

10000 : 0.002s

100000 : 0.035s

1000000 : 1.415s

**Conclusions:**

Clearly, we can notice that with the different algorithms the time taken for sorting is different. As we can see, bubble sort takes more time than selection sort which in turn takes more time than Insertion sort. This emphasizes the efficiency of insertion, and selection sort and the inefficiency of bubble sort. Thus we can conclude that with varying algorithms, on the same system, the time taken for sorting varies according to the algorithm. Of course, as the system changes, at different operating speeds, things are a little different but the relative time taken is the same.