Design & analysis of algorithms  
 **COMP-SCI 5592 [Spring Semester 2023]**

**Assessment-1**

GitHub: <https://github.com/NagaSatish45/DAA_Comp_Sci_5592>

**Question # 1:**

Merge Sort and Insertion Sort Although merge sort costs O(n lg n) and insertion sort costs O (n2), but constant factors in insertion sort may result it faster on average case for small size problems.

**1.1 Experiment:** Merge Sort vs. Insertion Sort Compare time complexities of both algorithms empirically:

• For this purpose, you must code both algorithms, and execute them for different values of n and plot the obtained complexities.

• The time complexities will be approximated by counting the numbers of comparison of two values and swaps only.

• For plotting you will need to calculate time complexities for the values of n like 5, 10, 50, 90, 95, 100.

• For each of these values, you will need to generate an array of random integer values between 0 and 1000.

• Time complexity for every array of size n, will help to calculate the average complexity.

**Results:**

**1.1** **Provide the code for both algorithms and show your counters that you used for calculating time complexity.**

**Source Code:** Code for Both Merge and Insertion Sorting (in C language)

**// Merge sort function**

void mergeSort(int arr[], int l, int r){

if (l < r)

{

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

// Function to merge two subarrays

void merge(int arr[], int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

i = 0;

j = 0;

k = l;

while (i < n1 && j < n2)

{

if (L[i] <= R[j])

{

arr[k] = L[i];

i++;

}

else

{

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1)

{

arr[k] = L[i];

i++;

k++;

}

while (j < n2)

{

arr[k] = R[j];

j++;

k++;

}

}

**// Insertion sort function**

void insertionSort(int arr[], int n)

{

int i, key, j;

for (i = 1; i < n; i++)

{

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > key)

{

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

}

int main()

{

int n, i;

int arr[100]; //array of size 100 (can be changed to accommodate different n values)

// n values: 5, 10, 50, 90, 95, 100

n = 101;

srand(time(0)); // seed the random number generator with the current time

for (i = 0; i < n; i++) {

arr[i] = rand() % 1000; //generates random numbers between 0 and 999

}

clock\_t start, end;

// Calculate time complexity for merge sort

start = clock();

mergeSort(arr, 0, n - 1);

end = clock();

double time\_taken\_merge = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Time taken for Merge sort: %f\n", time\_taken\_merge);

// Calculate time complexity for insertion sort

start = clock();

insertionSort(arr, n);

end = clock();

double time\_taken\_insertion = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Time taken for Insertion sort: %f\n", time\_taken\_insertion);

}

**Execution Results:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sno** | **Number of Values(n)** | **Time Taken to sort By Merge Sorting(sec)** | **Time Taken to sort By Insertion Sorting (sec)** |
| 1 | 5 | 0.000003 | 0.000002 |
| 2 | 10 | 0.000002 | 0.000001 |
| 3 | 50 | 0.000011 | 0.000002 |
| 4 | 90 | 0.000013 | 0.000001 |
| 5 | 95 | 0.000021 | 0.000002 |
| 6 | 100 | 0.000022 | 0.000002 |

**2** .**Plot the time complexity graphs for both algorithms on the same figure.**

Using R Language to plot the Time Complexity Graph between both the algorithms for above table values

R Code:

library(ggplot2)

MergeSort <- c(0.000003,0.000002,0.000011,0.000013, 0.000021,0.000022)

InsertionSort <- c(0.000002,0.000001,0.000002,0.000001,0.000002,0.000002)

x <- c(5,10,50,90,95,100)

library(ggplot2)

ggplot(data.frame(x, MergeSort, InsertionSort), aes(x)) +

geom\_line(aes(y = MergeSort, color = "MergeSort")) +

geom\_line(aes(y = InsertionSort, color = "InsertionSort")) +

scale\_y\_log10() +

labs(x = "Input Size", y = "Seconds (log scale)",

color = "Sorting Algorithm")

Result:

Chart, line chart

Description automatically generated

**3.What is the biggest value of n after which merge sort is better than insertion sort?**

For the values I provided above ,it shows that Insertion sort is better than Merge sort with low time complexity…as I tested by giving More values such as 1 million and above by algorithm was crashing for the array size to generate Random variables. But while I tested online it was failing when the n value is greater than 1 million to 1.5 million The Merge sort is becoming more effective than Insertion sorting.

**Question # 2:** Repeat the same experiment as above for Quick Sort and Insertion Sort.

**1.2 Experiment:** Quicksort vs. Insertion Sort Compare time complexities of both algorithms empirically:

• For this purpose, you must code both algorithms, and execute them for different values of n and plot the obtained complexities.

• The time complexities will be approximated by counting the numbers of comparison of two values and swaps only.

• For plotting you will need to calculate time complexities for the values of n like 5, 10, 50, 90, 95, 100.

• For each of these values, you will need to generate an array of random integer values between 0 and 1000.

• Time complexity for every array of size n, will help to calculate the average complexity.

**Results:**

**1.1** **Provide the code for both algorithms and show your counters that you used for calculating time complexity.**

**Source Code:** Code for Both Quick and Insertion Sorting (in C language)

int partition(int arr[], int low, int high) {

int pivot = arr[high]; // Selecting last element as pivot

int i = (low - 1);

for (int j = low; j <= high - 1; j++) {

if (arr[j] < pivot) {

i++;

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// Swapping arr[i + 1] and arr[high]

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return (i + 1);

}

void quickSort(int arr[], int low, int high)

{

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

// Insertion sort function

void insertionSort(int arr[], int n)

{

int i, key, j;

for (i = 1; i < n; i++)

{

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > key)

{

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

}

int main()

{

int n, i;

int arr[10000]; //array of size 100 (can be changed to accommodate different n values)

// n values: 5, 10, 50, 90, 95, 100

n = 100;

srand(time(0)); // seed the random number generator with the current time

for (i = 0; i < n; i++) {

arr[i] = rand() % 1000; //generates random numbers between 0 and 999

}

clock\_t start, end;

// Calculate time complexity for merge sort

start = clock();

quickSort(arr, 0, n - 1);

end = clock();

double time\_taken\_merge = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Time taken for Quick sort: %f\n", time\_taken\_merge);

// Calculate time complexity for insertion sort

start = clock();

insertionSort(arr, n);

end = clock();

double time\_taken\_insertion = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Time taken for Insertion sort: %f\n", time\_taken\_insertion);

}

**Execution Results:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sno** | **Number of Values(n)** | **Time Taken to sort By Quick Sorting (sec)** | **Time Taken to sort By Insertion Sorting(sec)** |
| 1 | 5 | 0.000002 | 0.000001 |
| 2 | 10 | 0.000003 | 0.000001 |
| 3 | 50 | 0.000004 | 0.000001 |
| 4 | 90 | 0.000007 | 0.000001 |
| 5 | 95 | 0.000012 | 0.000002 |
| 6 | 100 | 0.000008 | 0.000001 |

**2** .**Plot the time complexity graphs for both algorithms on the same figure.**

Using R Language to plot the Time Complexity Graph between both the algorithms for above table values

R Code:

library(ggplot2)

QuickSort <- c(0.000002,0.000003,0.000004,0.000007,0.000012,0.000008)

InsertionSort <- c(0.000001, 0.000001,0.000001,0.000001,0.000002,0.000001)

x <- c(5,10,50,90,95,100)

library(ggplot2)

ggplot(data.frame(x, MergeSort, InsertionSort), aes(x)) +

geom\_line(aes(y = MergeSort, color = "QuickSort")) +

geom\_line(aes(y = InsertionSort, color = "InsertionSort")) +

scale\_y\_log10() +

labs(x = "Input Size", y = "Seconds (log scale)",

color = "Sorting Algorithm")

Result:

Chart, line chart

Description automatically generated

**3.What is the biggest value of n after which Quick sort is better than insertion sort?**

In general, quick sort is considered to be faster than insertion sort for larger arrays due to its better average-case time complexity of O(n log n). On the other hand, insertion sort has a time complexity of O(n^2) in the worst case, which makes it slower than quick sort for larger inputs.

The exact value of n after which quick sort becomes faster than insertion sort can vary depending on the specifics of the implementation and the distribution of the data. However, it is generally considered that quick sort is faster than insertion sort for n >= 20. This value is just a rough estimate, and the actual value can be much larger or smaller depending on the specifics of the situation.