**Fatma Kausar**

**CSC382**

**Professor Yumei Huo**

**Lab#1**

///@file Source.cpp

#include <iostream>

#include <string>

#include <time.h>

#include <vector>

#include <algorithm> //for random\_shuffle

#include <chrono> //for chronotime

#include <math.h> // for pow and log

using namespace std;

long long int steps=0;

long long int stepCount=0;

long double cApprox = 0;

//InsertionSort algorithm

void insertionSort(vector<int> &arr)

{

int i, key, j;

stepCount += 4;

for (i = 1; i < arr.size(); i++)

{

stepCount += 2;

key = arr[i];

j = i - 1;

stepCount += 2;

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

{

stepCount += 2;

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

j = j - 1;

stepCount += 2;

}

stepCount++;

arr[j + 1] = key;

stepCount++;

}

stepCount++;

}

// merges two subarrays of array[].

void merge(vector<int> &arr, int start, int middle, int end) {

vector<int> leftArray(middle - start + 1);

vector<int> rightArray(end - middle);

steps += 2;

// fill in left array

for (int i = 0; i < leftArray.size(); ++i)

{

steps += 2;

leftArray[i] = arr[start + i];

steps++;

}

// fill in right array

for (int i = 0; i < rightArray.size(); ++i)

{

steps += 2;

rightArray[i] = arr[middle + 1 + i];

steps++;

}

/\* Merge the temp arrays \*/

// initial indexes of first and second subarrays

int leftIndex = 0, rightIndex = 0;

steps += 2;

// the index we will start at when adding the subarrays back into the main array

int currentIndex = start;

steps++;

// compare each index of the subarrays adding the lowest value to the currentIndex

while (leftIndex < leftArray.size() && rightIndex < rightArray.size()) {

steps += 2;

if (leftArray[leftIndex] <= rightArray[rightIndex]) {

steps++;

arr[currentIndex] = leftArray[leftIndex];

leftIndex++;

steps += 2;

}

else {

arr[currentIndex] = rightArray[rightIndex];

rightIndex++;

steps += 2;

}

currentIndex++;

steps++;

}

// copy remaining elements of leftArray[] if any

while (leftIndex < leftArray.size())

{

steps++;

arr[currentIndex++] = leftArray[leftIndex++];

steps++;

}

// copy remaining elements of rightArray[] if any

while (rightIndex < rightArray.size())

{

steps++;

arr[currentIndex++] = rightArray[rightIndex++];

steps++;

}

}

// main function that sorts array[start..end] using merge()

void mergeSort(vector<int> &arr, int start, int end) {

// base case

if (start < end) {

steps++;

// find the middle point

int middle = (start + end) / 2;

steps++;

mergeSort(arr, start, middle); // sort first half

mergeSort(arr, middle + 1, end); // sort second half

steps += 2;

// merge the sorted halves

merge(arr, start, middle, end);

steps++;

}

}

//Printing the vector arrays function

void printArray(vector<int> &arr)

{

for (int x : arr)

{

cout << x << " ";

}

}

//Generating sorted arrays for n input

void generateSort(vector<int> &arr, int n)

{

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

{

arr.push\_back(i);

}

}

//Generate reverse arrays for n input

void generateReverse(vector<int> &arr, int n)

{

for (int i = n; i >= 1; i--)

{

arr.push\_back(i);

}

}

//Generate random permutation arrays for n input

void generateRandom(vector<int> &arr, int n)

{

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

{

arr.push\_back(i);

}

random\_shuffle(arr.begin(), arr.end());

}

int main()

{

// Array of inputs n

int inputs[8] = { 100, 200, 300, 400, 500, 1000, 4000, 10000 };

//InsertionSort for Sorted Arrays, Reverse Arrays, Random Permutation Arrays

//C approximation will be stepCount/n^2

cout << "---------------------------------INSERTION SORT----------------------------------------" << endl;

cout << "SORTED ARRAYS" << endl;

for (int i = 0; i < sizeof(inputs) / sizeof(inputs[0]); i++)

{

stepCount = 0;

cApprox = 0;

vector<int> arr;

generateSort(arr, inputs[i]);

auto t1 = chrono::high\_resolution\_clock::now();

insertionSort(arr);

auto t2 = chrono::high\_resolution\_clock::now();

chrono::duration<double, milli> timer = t2 - t1;

cApprox = (stepCount) / (pow(inputs[i], 2));

cout << arr.size() << " ELEMENTS: Steps = " << stepCount << " | Runtime = " << timer.count() << " miliseconds. | C Approximation: " << cApprox << endl;

arr.clear();

}

cout << "REVERSED ARRAYS" << endl;

for (int i = 0; i < sizeof(inputs) / sizeof(inputs[0]); i++)

{

stepCount = 0;

cApprox = 0;

vector<int> arr;

generateReverse(arr, inputs[i]);

auto t1 = chrono::high\_resolution\_clock::now();

insertionSort(arr);

auto t2 = chrono::high\_resolution\_clock::now();

chrono::duration<double, milli> timer = t2 - t1;

cApprox = (stepCount) / (pow(inputs[i], 2));

cout << arr.size() << " ELEMENTS: Steps = " << stepCount << " | Runtime = " << timer.count() << " miliseconds. | C Approximation: " << cApprox << endl;

arr.clear();

}

cout << "RANDOM PERMUTATION ARRAYS" << endl;

for (int i = 0; i < sizeof(inputs) / sizeof(inputs[0]); i++)

{

stepCount = 0;

cApprox = 0;

vector<int> arr;

generateRandom(arr, inputs[i]);

auto t1 = chrono::high\_resolution\_clock::now();

insertionSort(arr);

auto t2 = chrono::high\_resolution\_clock::now();

chrono::duration<double, milli> timer = t2 - t1;

cApprox = (stepCount) / (pow(inputs[i], 2));

cout << arr.size() << " ELEMENTS: Steps = " << stepCount << " | Runtime = " << timer.count() << " miliseconds. | C Approximation: " << cApprox << endl;

arr.clear();

}

//MergeSort for Sorted Arrays, Reverse Arrays, Random Permutation Arrays

//C approximation will be steps/n\*log(n)

cout << endl << endl << "---------------------------------MERGE SORT----------------------------------------" << endl;

cout << "SORTED ARRAYS" << endl;

for (int i = 0; i < sizeof(inputs) / sizeof(inputs[0]); i++)

{

steps = 0;

cApprox = 0;

vector<int> arr;

generateSort(arr, inputs[i]);

auto t1 = chrono::high\_resolution\_clock::now();

mergeSort(arr, 0, (arr.size() - 1));

auto t2 = chrono::high\_resolution\_clock::now();

chrono::duration<double, milli> timer = t2 - t1;

cApprox = (steps) / (inputs[i] \* log(inputs[i]));

cout << arr.size() << " ELEMENTS: Steps = " << steps << " | Runtime = " << timer.count() << " miliseconds. | C Approximation: " << cApprox << endl;

arr.clear();

}

cout << "REVERSED ARRAYS" << endl;

for (int i = 0; i < sizeof(inputs) / sizeof(inputs[0]); i++)

{

steps = 0;

cApprox = 0;

vector<int> arr;

generateReverse(arr, inputs[i]);

auto t1 = chrono::high\_resolution\_clock::now();

mergeSort(arr, 0, (arr.size() - 1));

auto t2 = chrono::high\_resolution\_clock::now();

chrono::duration<double, milli> timer = t2 - t1;

cApprox = (steps) / (inputs[i] \* log(inputs[i]));

cout << arr.size() << " ELEMENTS: Steps = " << steps << " | Runtime = " << timer.count() << " miliseconds. | C Approximation: " << cApprox << endl;

arr.clear();

}

cout << "RANDOM PERMUTATION ARRAYS" << endl;

for (int i = 0; i < sizeof(inputs) / sizeof(inputs[0]); i++)

{

steps = 0;

cApprox = 0;

vector<int> arr;

generateRandom(arr, inputs[i]);

auto t1 = chrono::high\_resolution\_clock::now();

mergeSort(arr, 0, (arr.size() - 1));

auto t2 = chrono::high\_resolution\_clock::now();

chrono::duration<double, milli> timer = t2 - t1;

cApprox = (steps) / (inputs[i] \* log(inputs[i]));

cout << arr.size() << " ELEMENTS: Steps = " << steps << " | Runtime = " << timer.count() << " miliseconds. | C Approximation: " << cApprox << endl;

arr.clear();

}

cout << endl << endl << "---------------------------------50 INSTANCES----------------------------------------" << endl;

//50 instances of Random inputs InsertionSort

cout << "50 INSTANCES INSERTION SORT" << endl;

for (int i = 0; i < sizeof(inputs) / sizeof(inputs[0]); i++)

{

stepCount = 0;

cApprox = 0;

double totalT = 0;

for (int x = 0; x < 50; x++)

{

vector<int> arr;

for (int j = 1; j <= inputs[i]; j++)

{

arr.push\_back(rand() % inputs[i] + 1);

}

auto t1 = chrono::high\_resolution\_clock::now();

insertionSort(arr);

auto t2 = chrono::high\_resolution\_clock::now();

chrono::duration<double, milli> timer = t2 - t1;

totalT += timer.count();

arr.clear();

}

cApprox = (stepCount / 50) / (pow(inputs[i],2));

cout << "50 INSTANCES OF " << inputs[i] << " ELEMENTS: Steps Average = " << stepCount/50 << " | Runtime = " << totalT << " miliseconds | Average time: " << totalT/50 << " miliseconds. | C Approximation: " << cApprox << endl;

}

//50 instances of random inputs mergeSort

cout << "50 INSTANCES MERGE SORT" << endl;

for (int i = 0; i < sizeof(inputs) / sizeof(inputs[0]); i++)

{

steps = 0;

cApprox = 0;

double totalT = 0;

for (int x = 0; x < 50; x++)

{

vector<int> arr;

for (int j = 1; j <= inputs[i]; j++)

{

arr.push\_back(rand() % inputs[i] + 1);

}

auto t1 = chrono::high\_resolution\_clock::now();

mergeSort(arr, 0, (arr.size() - 1));

auto t2 = chrono::high\_resolution\_clock::now();

chrono::duration<double, milli> timer = t2 - t1;

totalT += timer.count();

arr.clear();

}

cApprox = (steps / 50) / (inputs[i] \* log(inputs[i]));

cout << "50 INSTANCES OF " << inputs[i] << " ELEMENTS: Steps Average = " << steps/50 << " | Runtime = " << totalT << " miliseconds | Average time: " << totalT/50 << " miliseconds. | C Approximation: " << cApprox << endl;

}

system("PAUSE");

return 0;

} //end main

**A screen shot of a computer

Description automatically generatedOutput:**

**A screen shot of a computer

Description automatically generated**

**A screen shot of a computer

Description automatically generated**

**Insertion Sort Running Time In Milliseconds**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input Size | Sorted | Reverse | Random Permutation | 50 Instances Average |
| 100 | 0.2295 | 5.2381 | 2.3583 | 3.72549 |
| 200 | 0.3208 | 22.2707 | 9.2919 | 13.8807 |
| 300 | 0.299 | 44.4023 | 28.1532 | 29.0095 |
| 400 | 0.4017 | 69.5943 | 44.6818 | 53.344 |
| 500 | 0.4986 | 110.2 | 68.7498 | 83.6949 |
| 1000 | 1.5906 | 458.628 | 225.628 | 329.323 |
| 4000 | 6.4903 | 8439.44 | 4412.94 | 5319.72 |
| 10000 | 14.1005 | 51244 | 30699.4 | 38942.9 |

**Merge Sort Running Time In Milliseconds**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input Size | Sorted | Reverse | Random Permutation | 50 Instances Average |
| 100 | 4.8864 | 3.6226 | 3.7307 | 4.48856 |
| 200 | 9.2835 | 11.6945 | 8.4183 | 9.79402 |
| 300 | 14.1179 | 12.3364 | 14.0176 | 15.2257 |
| 400 | 16.8485 | 16.2653 | 18.2677 | 21.0907 |
| 500 | 21.7617 | 20.9747 | 23.7114 | 30.1851 |
| 1000 | 43.7531 | 46.2174 | 48.6439 | 58.8891 |
| 4000 | 198.855 | 239.178 | 216.227 | 271.214 |
| 10000 | 533.399 | 612.249 | 711.775 | 760.396 |

**Insertion Sort Steps**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input Size | Sorted | Reverse | Random Permutation | 50 Instances Average |
| 100 | 599 | 20399 | 10039 | 10380 |
| 200 | 1199 | 80799 | 43219 | 40841 |
| 300 | 1799 | 181199 | 92555 | 91610 |
| 400 | 2399 | 321599 | 160691 | 161028 |
| 500 | 2999 | 501999 | 259711 | 251573 |
| 1000 | 5999 | 2003999 | 1000019 | 1003541 |
| 4000 | 23999 | 32015999 | 15846627 | 16051039 |
| 10000 | 59999 | 200039999 | 99309943 | 100074763 |

**Merge Sort Steps**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input Size | Sorted | Reverse | Random Permutation | 50 Instances Average |
| 100 | 5774 | 5298 | 6225 | 6257 |
| 200 | 12958 | 11906 | 14202 | 14215 |
| 300 | 20662 | 18970 | 22766 | 22802 |
| 400 | 28726 | 26422 | 31873 | 31832 |
| 500 | 36518 | 34078 | 40912 | 40948 |
| 1000 | 80046 | 74666 | 90393 | 90365 |
| 4000 | 376214 | 350694 | 429529 | 429548 |
| 10000 | 1044102 | 961894 | 1190897 | 1190759 |

**C Approximation Insertion Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input Size | Sorted | Reverse | Random Permutation | 50 Instances  Average |
| 100 | 0 | 2 | 1 | 1 |
| 200 | 0 | 2 | 1 | 1 |
| 300 | 0 | 2 | 1 | 1 |
| 400 | 0 | 2 | 1 | 1 |
| 500 | 0 | 2 | 1 | 1 |
| 1000 | 0 | 2 | 1 | 1 |
| 4000 | 0 | 2 | 1 | 1 |
| 10000 | 0 | 2 | 1 | 1 |

**C Approximation Merge Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input Size | Sorted | Reverse | Random Permutation | 50 Instances  Average |
| 100 | 12 | 11 | 13 | 13 |
| 200 | 12 | 11 | 13 | 13 |
| 300 | 12 | 11 | 13 | 13 |
| 400 | 12 | 11 | 13 | 13 |
| 500 | 12 | 11 | 13 | 13 |
| 1000 | 12 | 11 | 13 | 13 |
| 4000 | 12 | 11 | 13 | 13 |
| 10000 | 12 | 11 | 13 | 13 |

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated**