Question 5

I developed a binary search for eight distinct arrays size of 128, 512, 2048, 8192, 32768, 131072, 524288, and 2097152 with three distinct programming languages: Python, Java, C++. For each distinct programming languages, I ran 10,000,000 failed searches. The data I retrieved for the searches are shown below.

|  |  |  |  |
| --- | --- | --- | --- |
| N-Size of Array | Python | C++ | Java |
| 128 | 24.83 seconds | 0.000000448 seconds | 0.17322 seconds |
| 512 | 32.68 seconds | 0.000000585 seconds | 0.20086 seconds |
| 2,048 | 45.56 seconds | 0.000000746 seconds | 0.30817 seconds |
| 8,192 | 61.16 seconds | 0.000000885 seconds | 0.35829 seconds |
| 32,728 | 64.99 seconds | 0.000001029 seconds | 0.42256 seconds |
| 131,072 | 78.14 seconds | 0.000001369 seconds | 0.42256 seconds |
| 524,288 | 77.21 seconds | 0.000001363 seconds | 0.52993 seconds |
| 2,097,157 | 103.65 seconds | 0.000001554 seconds | 0.5983 seconds |

I was under the assumption C++ to be the quickest since it is a high-level programming language that is compiled during runtime. C++ is objective code which goes right into your machine. On the other hand, Java is required to go through modifications since it is written in byte. Also, it is required to go through the java virtual machine before entering your computer. As a result, it is not as fast as compared to that of C++. Finally, Python was the slowest out of all three programming languages. Its code is usually about three to five times shorter than the same code as that of Java. Additionally, it is about three to five times shorter than the same code as that of C++. Python’s run time has to work much harder than C++ and Java due to the run-time typing.

Java

**import** java.lang.Math;

**public** **class** program {

**static** **int** binarySearch(**int**[] lst, **int** key, **int** min, **int** max){

**while**(max-min>1){

**int** midPos = (max-min)/2+min;

**int** mid = lst[midPos];

**if** (key==mid){

**return** midPos;

}

**else** **if** (key<mid){

max = midPos-1;

}

**else** **if** (key > mid){

min = midPos +1;

}

}

**if** (lst[min] == key){

**return** min;

}

**else** {

**return** -1;

}

}

**public** **static** **void** main (String [] args){

**int** [][] arrays = **new** **int** [8][(**int**)Math.*pow*(2, (2\*7+7))];

**for** (**int** i=0; i<8; i++){

**for**(**int** j=0; j<Math.*pow*(2, (2\*i+7)); j++){

arrays [i][j] =(**int**)(Math.*pow*(j, 2));

}

}

**for**(**int** i=0; i<8; i++){

**long** start= System.*nanoTime*();

**for** (**int** j=0; j<10000000; j++){

*binarySearch*(arrays[i],2,0,(**int**) Math.*pow*(2, (2\*i+7)-1));

}

**long** end = System.*nanoTime*();

System.***out***.println((end-start)\*Math.*pow*(10, -9) + "seconds for" + (**int**)Math.*pow*(2,(2\*i+7)));

}

}

}

C++

#include<time.h>

#include<iostream>

using namespace std;

int binarySearch(int beginning [], int data\_located, int minimum, int maximum)

{

while (maximum - minimum > 1)

{

int center\_location = (maximum - minimum)/2 + minimum;

int center = beginning[center\_location];

if (data\_located == center )

{

return center\_location;

}

else if (data\_located < center)

{

maximum = center\_location;

}

else if (data\_located > center)

{

minimum = center\_location + 1;

}

}

if (beginning[minimum] == data\_located)

{

return minimum;

}

else

{

return -1;

}

}

int power(int base, int power\_raised)

{

int rtn = 1;

for (int i= 0; i < power\_raised; i++)

{

rtn = rtn = rtn\*base;

}

return rtn;

}

int main()

{

int \* array\_destinationspot[8];

for (int i = 0; i < 8; i++)

{

array\_destinationspot[i] = new int[power(2, 2\*i+7)];

for (int j = 0; j < power(2,2\*i+7); j++)

{

array\_destinationspot[i][j] = power (j,2);

}

}

for (int i = 0; i <8; i++)

{

clock\_t start = clock();

for(int j = 0; j <10000000; j++)

{

binarySearch(array\_destinationspot[i],2,0, power(2,(2\*i+7)-1));

}

clock\_t end = clock();

cout <<"The time needed to complete ten million iterations: " << (end - start)\*0.000000001 << " of size"

<< power(2,(2\*i+7)) << endl;

}

for (int i = 0; i < 8; i++)

{

delete array\_destinationspot[i];

}

return 0;

}

Python

import time

def binarySearch(lst, key, min, max):

while max - min > 1:

midPos = (max-min)//2+ min

mid = lst[midPos]

if key== mid:

return midPos

elif key < mid:

max = midPos - 1

elif key > mid:

min = midPos + 1

if lst[min]==key:

return min

else:

return -1

arrays = [[],[],[],[],[],[],[],[]]

for i in range(8):

for j in range(2\*\*(2\*i+7)):

arrays[i].append(j\*\*2)

for i in range(8):

start=time.time()

for j in range (10000000):

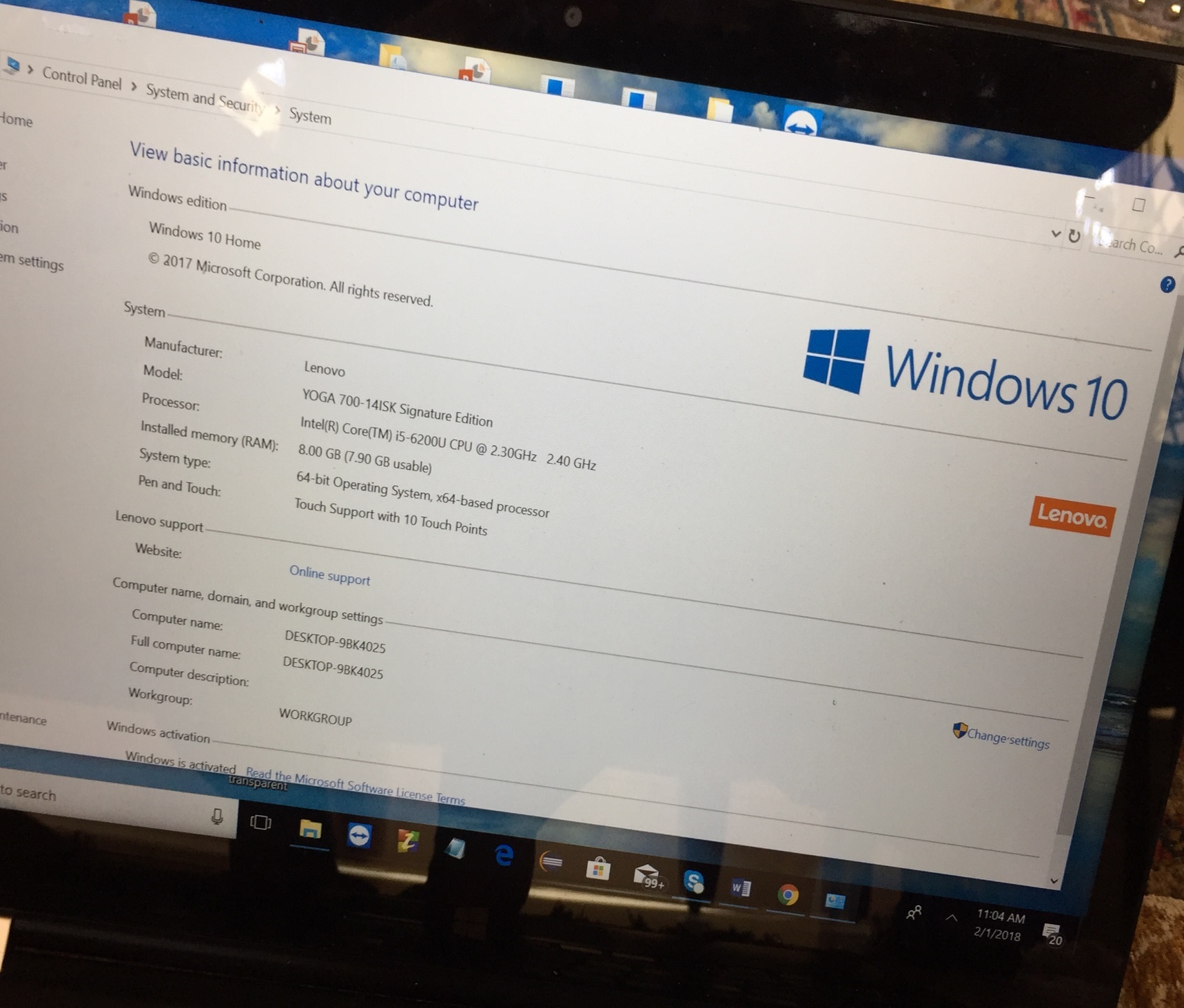
binarySearch(arrays[i], 2, 0, 2\*\*(2\*i+7)-1)

end=time.time()

print("This is the complete ten milion itterations:" , end -start , "seconds")

Question 4

The computer that this program is being executed on is a Lenovo YOGA 700-14ISK with a total of physical memory of 8GB, running on a Intel i5 processor with Microsoft Windows 10 64-bit operating system.



Towards the beginning, I had to figure out the m value. First, I equaled m to be 100000 which worked. Afterwards, I increased m to be 1000000 which caused the program to completely crash. Finally, I went back and reduced m to be 50000, which was the highest level of m value I was able to work with. During some intervals, the program would still crash, and I would consider this to be a series of fluctuation operations occurring in the background of my PC. The average time elapsed for the first allocation for over seven runs was 13.53500 seconds. The average time elapsed for the second allocation for over seven runs was 10.81800 seconds. As memory is released using the free() command in C, memory blocks are left available in order to use in the main memory bank. If the program attempts to provide memory spaces larger than those that have been left by the free() command, the computer attempts to provide as much as it is possible with the banks that are right next to these particular positions. Considering there is no full certainty that are will be an adequate amount of free memory in order to protect the memory that has been developed by other programs running in the computer, it is the primary reason the program stops functioning properly and exits.

C Code

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MNUMARRAYS 500000

void allocateArray(int \*\*mainArray, int numOfArrays, int sizeOfArrays) {

int i;

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

mainArray[i] = (int \*) malloc(sizeof(int) \* sizeOfArrays);

}

}

/\* Loop through arrays and free memory every even \*/

void deallocateEvenArrays(int \*\*mainArray, int numOfArrays) {

int i;

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

if(i % 2 == 0) {

free(mainArray[i]);

}

}

}

int main() {

clock\_t first\_startTime, first\_endTime;

clock\_t second\_startTime, second\_endTime;

int \*\* mainArray1;

int \*\* mainArray2;

first\_startTime = clock();

mainArray1 = (int \*\*) malloc(sizeof(int) \* 3 \* MNUMARRAYS); /\* Allocate3\*MNUMBERRAYS arrays \*/

allocateArray(mainArray1, 3 \* MNUMARRAYS, 800000);

first\_endTime = clock();

deallocateEvenArrays(mainArray1, 3 \* MNUMARRAYS);

second\_startTime = clock();

mainArray2 = (int \*\*) malloc(sizeof(int) \* 3 \* MNUMARRAYS);

allocateArray(mainArray2, 3 \* MNUMARRAYS, 900000);

second\_endTime = clock();

/\* Print out the difference for both sequences & dealloc \*/

printf("FIRST ALLOCATION TIME: %f seconds\n", (double)(((double)first\_endTime - (double)first\_startTime)/(double)CLOCKS\_PER\_SEC));

printf("SECOND ALLOCATION TIME: %f seconds\n", (double)(((double)second\_endTime - (double)second\_startTime)/(double)CLOCKS\_PER\_SEC));

getchar();

}

Question 3

a): If we choose the pivot as the last the last element, it will leave one of the partitions empty.

If the pivot is chosen as the last element, A[lo:m] and A[m:hi] won’t be of the same length.

As a result, it can’t be optimal. Picking the the pivot element shows the complexity of the

algorithm. It can either be n\*logn or quadratic time. The solution would if be if we have N

elements, the pivot would be the higher of N/2 elements.

An example would be a set of numbers from 2,6,4,8,3,1. In this case the pivot would be 4

when you arrange in the ascending order of 1, 2, 3, 4, 6, 8. The next would be to swap the

pivot with the last number. Next, define two pointers, first and last. The first would be the

first index and last to be the last index. If first is less than last, we keep increasing first until

we find an element larger than 4. Similarly, if last in larger than first, we keep decrementing

last until we find an element less than four. After both first and last stop, we swap the

elements. The final sorted array is [2, 3,1][4][6,8].

b): When we need to QuickSort an array in increasing order, the algorithm will run in the

slowest possible way when the array is already arranged in decreasing ordering.

An example would be a set of numbers 35,24,16,10,7,5.