**1.**

#include <stdio.h>

#include <stdlib.h>

#define ARRAY\_LENGTH 5

//Printing the array

void printArray(int \*arr, int len) {

printf("\n---Array elements: --\n");

for (int i=0;i<len;i++) {

printf("%d\n",arr[i]);

}

printf("---------------\n");

}

//Wrapper for malloc

void\* Malloc(int size) {

void \*p = malloc(size);

if (p==NULL) {

printf("malloc failed\n");

return NULL;

}

else

return p;

}

//insert array definition

int\* insert(int \*origArray, int length, int index, int value) {

if (length==0){

int \*newArray = (int\*) Malloc(sizeof(int));

\*newArray = value;

return newArray;

}

else if ((index<0) || (index>length)) {

printf("New value cannot be inserted at pos = %d. Index not in range

[0-%d]\n",index,length);

return NULL;

}

else {

int \*newArray = (int\*) Malloc(sizeof(int)\*(length+1));

//Loop through until the point of insertion

for (int i=0;i<index;i++) {

newArray[i] = origArray[i];

}

//Insert new element at the given index

newArray[index] = value;

//After new element, insert rest of elements from old array into new array.

for (int j=index+1;j<=length;j++) {

newArray[j] = origArray[j-1];

}

if(origArray != NULL) {

free(origArray);

}

return newArray;

}

}

int main(int argc, char \*\*argv) {

int \*origArray = (int\*)Malloc(sizeof(int) \* ARRAY\_LENGTH);

int \*n;

printf("Input elements of array (one per line)\n");

for (int i=0;i<ARRAY\_LENGTH;i++) {

scanf("\n%d",origArray+i);

}

n = insert(origArray,ARRAY\_LENGTH,5,74);

if(n != NULL) {

printArray(n,ARRAY\_LENGTH+1);

free(n);

}

return EXIT\_SUCCESS;

}

**2.**

int main() {

/\* Setting to allow fine-tuning the granularity of the readings \*/

unsigned int INSERTS\_PER\_READING = 1500;

/\* Start with an empty array \*/

int \*array = NULL;

unsigned int length = 0;

unsigned int value = rand();

unsigned int index = 0;

unsigned long startTime,stopTime;

double timePerInsert;

printf("\nArray length Seconds per insert\n\n");

/\* Take 60 readings \*/

for (int i=0;i<60;i++) {

startTime = clock();

for (int j=0;j<INSERTS\_PER\_READING;j++) {

array = insert(array,length,index,value);

length += 1;

index = rand() %length;

value = rand();

}

stopTime = clock();

timePerInsert = (double)(stopTime - startTime) /

(CLOCKS\_PER\_SEC\*INSERTS\_PER\_READING);

printf("%5d %25.10lf\n",length,timePerInsert);

}

if (array != NULL) {

free(array);

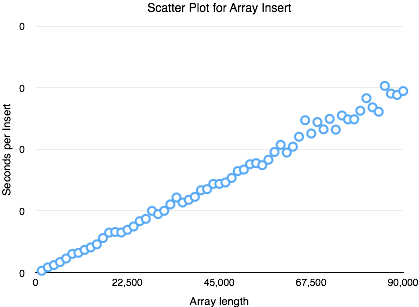
}

return EXIT\_SUCCESS;

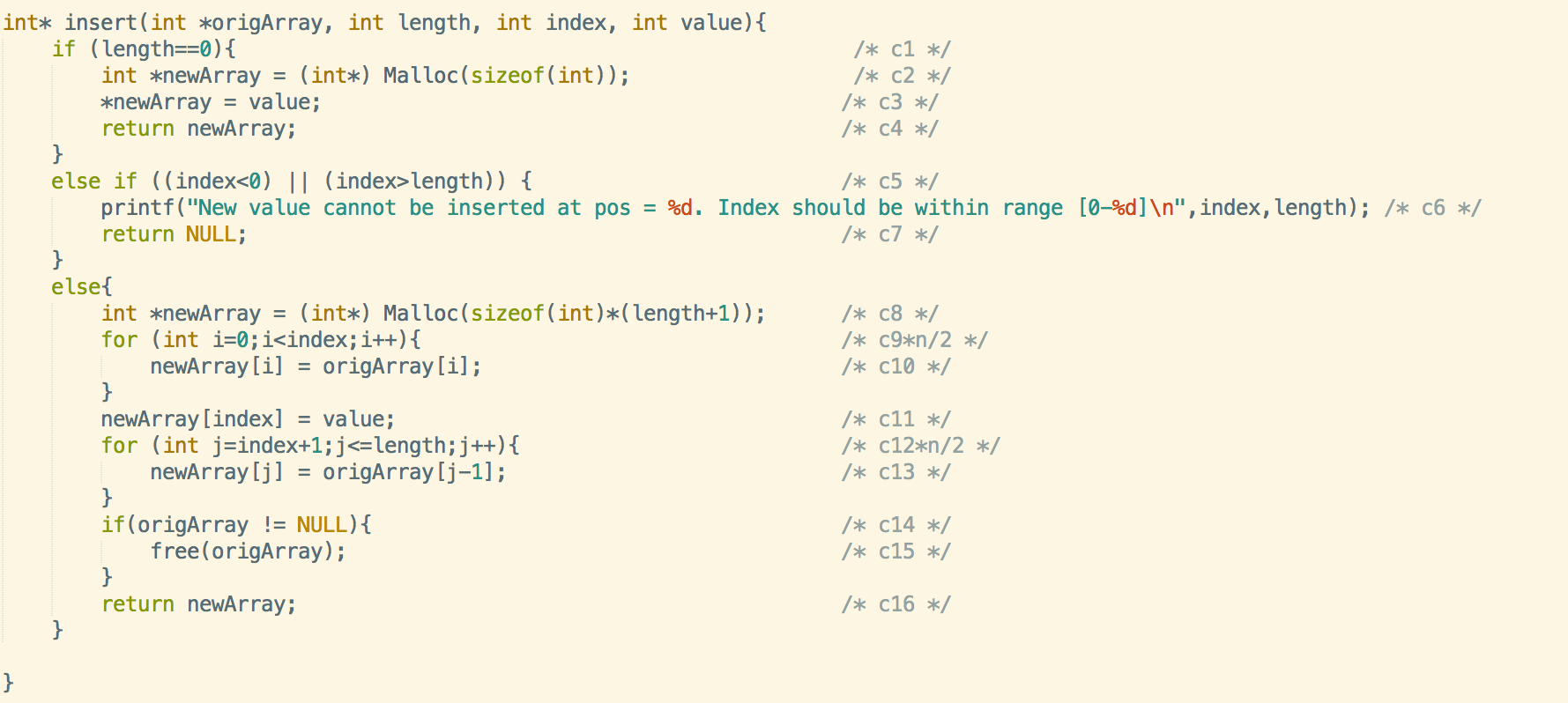
}

|  |  |
| --- | --- |
|  |  |
| Array length | Seconds per insert |
| 1500 | 0.0000020740 |
| 3000 | 0.0000062427 |
| 4500 | 0.0000090640 |
| 6000 | 0.0000126547 |
| 7500 | 0.0000170367 |
| 9000 | 0.0000226613 |
| 10500 | 0.0000239327 |
| 12000 | 0.0000275053 |
| 13500 | 0.0000305040 |
| 15000 | 0.0000343127 |
| 16500 | 0.0000419653 |
| 18000 | 0.0000485813 |
| 19500 | 0.0000492893 |
| 21000 | 0.0000486640 |
| 22500 | 0.0000519753 |
| 24000 | 0.0000560193 |
| 25500 | 0.0000624727 |
| 27000 | 0.0000652360 |
| 28500 | 0.0000750100 |
| 30000 | 0.0000710960 |
| 31500 | 0.0000750867 |
| 33000 | 0.0000828960 |
| 34500 | 0.0000912613 |
| 36000 | 0.0000851593 |
| 37500 | 0.0000884320 |
| 39000 | 0.0000920453 |
| 40500 | 0.0001001047 |
| 42000 | 0.0001014400 |
| 43500 | 0.0001078233 |
| 45000 | 0.0001077967 |
| 46500 | 0.0001095767 |
| 48000 | 0.0001150700 |
| 49500 | 0.0001234453 |
| 51000 | 0.0001252547 |
| 52500 | 0.0001317993 |
| 54000 | 0.0001330520 |
| 55500 | 0.0001305967 |
| 57000 | 0.0001372313 |
| 58500 | 0.0001466773 |
| 60000 | 0.0001553687 |
| 61500 | 0.0001460060 |
| 63000 | 0.0001531480 |
| 64500 | 0.0001652053 |
| 66000 | 0.0001853173 |
| 67500 | 0.0001691280 |
| 69000 | 0.0001831200 |
| 70500 | 0.0001742227 |
| 72000 | 0.0001869640 |
| 73500 | 0.0001738760 |
| 75000 | 0.0001910860 |
| 76500 | 0.0001863713 |
| 78000 | 0.0001864187 |
| 79500 | 0.0001968587 |
| 81000 | 0.0002121013 |
| 82500 | 0.0002009827 |
| 84000 | 0.0001957033 |
| 85500 | 0.0002271200 |
| 87000 | 0.0002177427 |
| 88500 | 0.0002160767 |
| 90000 | 0.0002207907 |

**3.**



**4.**



Big-O PERFORMANCE

= O ( (c1 + c2 + c3 + c4) **or** (c5 + c6 + c7) **or**  ( c8 +c9n/2 \*c10 + c11 + c12n/2\*c13+c14+c15+c16) )

= O( (c8+c11+c14\_c15+c16) + (c9\*c10 + c12\*c13)n/2 )

= O(n/2)

= O(n)

This function’s runtime is linearly proportional to the length of array. Therefore , Big-O is the linear run-time O(n).

Copying the old array elements up until the point of insertion consumes lot of processor cycles. Also, after the new element insertion, copying the remainder of the old array into the new array takes up significant processor cycles. Both these copying operations contribute most heavily to the Big-O performance.

**5.**

The performance degrades as the length of the array grows. With increasing array length, it takes proportionally longer to form the new array with the new element inserted. This is also evident from the scatter plot that shows the “Seconds per Insert” increasingly linearly with array size. The Big-O analysis matches the results of running the program.