DEsign and analysis of algorithms

By Utsav Balar

201903103510391

BTech CE-C Batch 1

LAB MANUAL

**Practical 1**

**Aim: Find Factorial of a given number using Recursive and Iterative methods. Calculate execution time.**

**Code:**

#include <stdio.h>

#include <time.h>

/\*

\* %lu support data limit upto 2^64 (precisely: 18.4 quintillion)

\* so the following code will be limited upto 65

\*/

long recur(int n);

long iter(int n);

int main() {

int num, c;

clock\_t t;

printf("Enter the number: ");

scanf("%d", & num);

printf("1 = Recursion | 2 = Iteration ? Enter 1 or 2: \n");

scanf("%d", & c);

if (num < 0)

printf("Factorial of negative numbers are not possible\n");

else if (num > 65)

printf("Data limit exceeded! Please use a lower value");

else {

switch (c) {

case 1:

printf("\nUsing Recursion: \n");

t = clock();

printf("Factorial of %d is %lu\n", num, recur(num));

t = clock() - t;

printf("Time Taken to execute: (%f seconds).\n",

((float) t) / CLOCKS\_PER\_SEC);

break;

case 2:

printf("\nUsing Iterative: \n");

t = clock();

printf("Factorial of %d is %lu\n", num, iter(num));

t = clock() - t;

printf("Time Taken to execute: (%f seconds).\n",

((float) t) / CLOCKS\_PER\_SEC);

break;

default:

printf("Invalid Input\n");

break;

}

return 0;

}

}

long recur(int n) {

if (n == 0)

return 1;

return (n \* recur(n - 1));

}

long iter(int n) {

long fact = 1;

while (n > 0) {

fact = fact \* n;

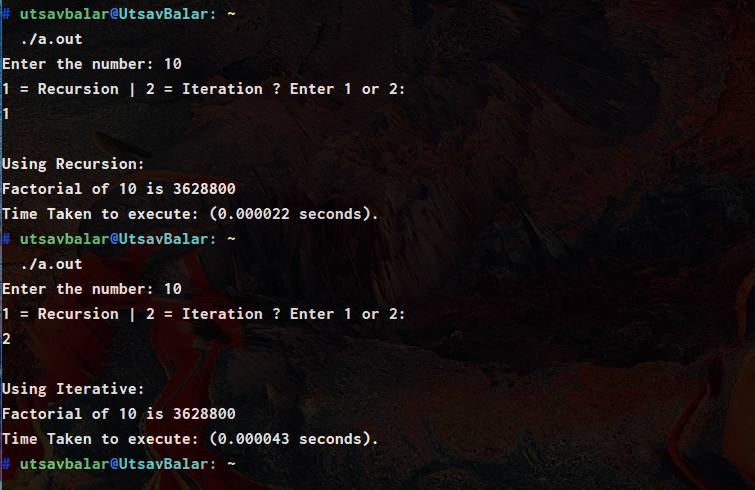
n--;

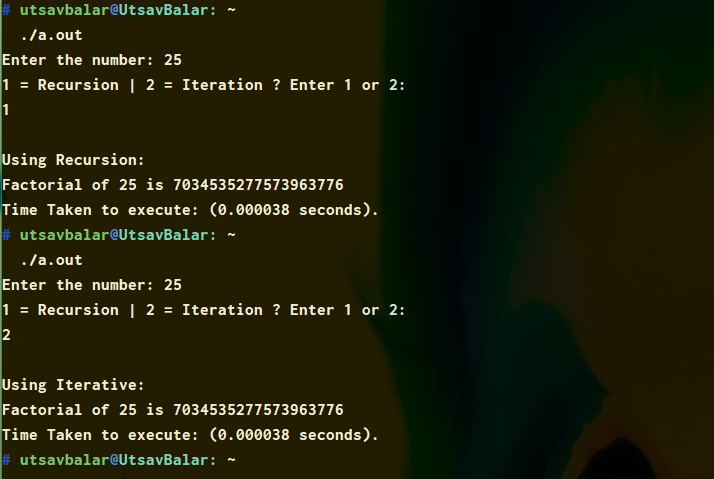
}

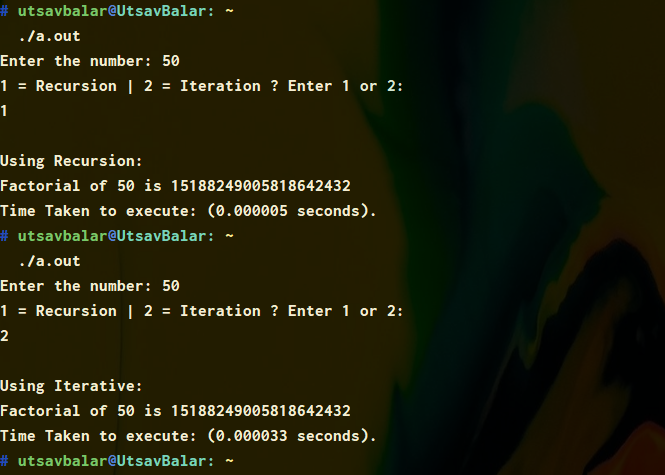
return fact;

}

**Output:**







**Practical 2**

**Aim: Following is the data of height of 10 students of Sports class in school. Lined up in a random order in front of the teacher, who’s put to the task of lining all up in an ascending order of height. Now your task is to help your teacher in arranging them using following set of data and measure their execution time and time complexity.**

**Height:**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Student | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Height | 89 | 42 | 100 | 93 | 11 | 234 | 30 | 82 | 22 | 75 |

**Code:**

#include <stdio.h>

#include <stdbool.h>

#include <time.h>

void sort(int arr[], int n) {

int i, j, swap;

bool is\_sorted = false;

for (i = 0; i < n - 1; i++) {

for (j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

swap = arr[j];

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

arr[j + 1] = swap;

is\_sorted = true;

}

if (!is\_sorted)

return;

}

}

}

int main() {

clock\_t t;

int height[10] = {

89,

42,

100,

93,

11,

234,

30,

82,

22,

75

};

int n = (sizeof(height) / sizeof(height[0]));

printf("Input Array: ");

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

printf("%d ", height[i]);

t = clock();

sort(height, n);

t = clock() - t;

printf("\nSorted Array: ");

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

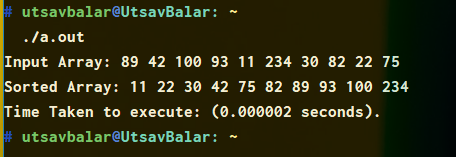
printf("%d ", height[i]);

printf("\nTime Taken to execute: (%f seconds).\n",

((float) t) / CLOCKS\_PER\_SEC);

return 0;

}



**Practical 2 [Analysis]**

**Code:**

// Time complexivity analysis of bubble sort algorithm

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Created macro for Random number generation

#define randnum(min, max)\

((rand() % (int)(((max) + 1) - (min))) + (min))

// Function to sort Array using bubble sort algorithm

void sort(int arr[], int n) {

bool isSorted = false; // Initially array is unsorted

while (!isSorted)

{

isSorted = true;

// Perform Bubble sort on odd indexed element

for (int i=1; i<=n-2; i=i+2)

{

if (arr[i] > arr[i+1])

{

swap(arr[i], arr[i+1]);

isSorted = false;

}

}

// Perform Bubble sort on even indexed element

for (int i=0; i<=n-2; i=i+2)

{

if (arr[i] > arr[i+1])

{

swap(arr[i], arr[i+1]);

isSorted = false;

}

}

}

return;

}

// Function to print array

void printarr(int arr[], int size) {

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

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

printf("\n");

}

int main(void) {

clock\_t t1, t2, t3;

int start = 1, end, i, temp, r;

printf("Enter the largest number for array: ");

scanf("%d", &end);

int size = end - start + 1;

int arr[size];

// Fill array with numbers in range (start, end)

for (temp = 0, i = start; temp < size; i++, temp++)

arr[temp] = i;

// To avoid repetition with each run

srand((unsigned int) time(NULL));

// Store all the random numbers in Array arr[]

for (i = size - 1; i > 0; i--) {

r = randnum(start, end);

temp = arr[i];

arr[i] = arr[r];

arr[r] = temp;

}

printf("Unsorted Array: (Average Case): \n");

printarr(arr, size);

t1 = clock();

sort(arr, size);

t1 = clock() - t1;

printf("\nSorted Array: (Average Case): \n");

printarr(arr, size);

t2 = clock();

sort(arr, size);

t2 = clock() - t2;

printf("\nSorted Array: (Best Case): \n");

printarr(arr, size);

// Reverse the numbers in Array arr[]

int tmp = 0, x = 0, y = size - 1; // End value of array

while (x < y) {

tmp = arr[x];

arr[x] = arr[y];

arr[y] = tmp;

x++;

y--;

}

printf("\nUnsorted Array: (Worst Case): \n");

printarr(arr, size);

t3 = clock();

sort(arr, size);

t3 = clock() - t2;

printf("\nSorted Array: (Worst Case): \n");

printarr(arr, size);

printf("\nTime Taken to execute for\n"

"Average Case\t: (%f seconds)\n"

"Best Case\t: (%f seconds)\n"

"Worst Case\t: (%f seconds)\n",

(((float) t1) / CLOCKS\_PER\_SEC),

(((float) t2) / CLOCKS\_PER\_SEC),

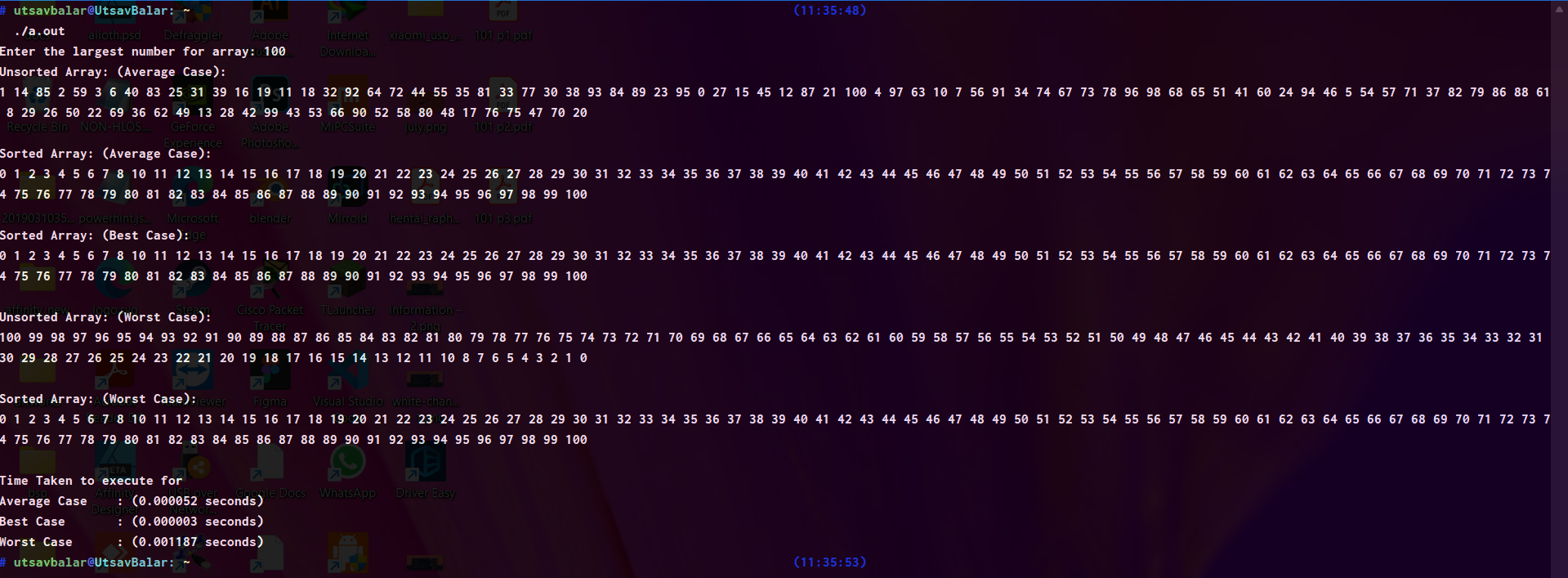
(((float) t3) / CLOCKS\_PER\_SEC));

return 0;

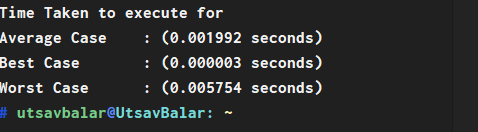
}

**Output:**

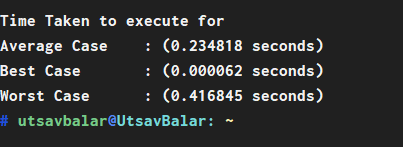
1. 100 Random numbers



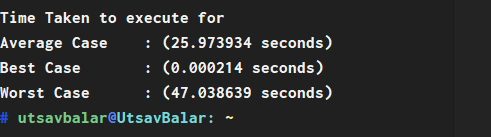
1. 1000 Random numbers



1. 10000 Random numbers



1. 100000 Random numbers



**Practical 3**

**Aim: Suppose you are having a set of trump cards, each card is having one letter on it. To play a game it would be better if all the trump cards will be arranged. To arrange trump cards following technique should be adopted.**

**Take very first card as a Key element and try to place it on its final position of arrangement. Repeat this procedure until all the cards will be arranged.**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Card No | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Value | D | W | A | S | E | U | G |

**Code:**

#include <stdio.h>

#include <string.h>

#include <time.h>

void printarr(const char \*\*array, int size);

void sort(const char \*\*array, int size);

int main()

{

clock\_t t;

const char \*cards[] = {

"D",

"W",

"A",

"S",

"E",

"U",

"G"};

int size = (sizeof(cards) / sizeof(const char \*));

printf("Unsorted Array\t: ");

printarr(cards, size);

t = clock();

sort(cards, size);

t = clock() - t;

printf("\nSorted Array\t: ");

printarr(cards, size);

printf("\nTime Taken to execute: (%f seconds)\n",

(((float)t) / CLOCKS\_PER\_SEC));

return 0;

}

void sort(const char \*\*array, int size)

{

int j;

for (j = 1; j < size; j++)

{

int i = j - 1;

const char \*key = array[j];

for (; i >= 0; i--)

{

if (strcasecmp(array[i], key) < 0)

break;

printf("\nCard moved\t: %s\n", key);

array[i + 1] = array[i];

printf("Card inserted\t: %s, at position : %d", key, i);

}

array[i + 1] = key;

printf("\nIteration %d\t: ", j);

printarr(array, size);

}

}

void printarr(const char \*\*array, int size)

{

printf("[ ");

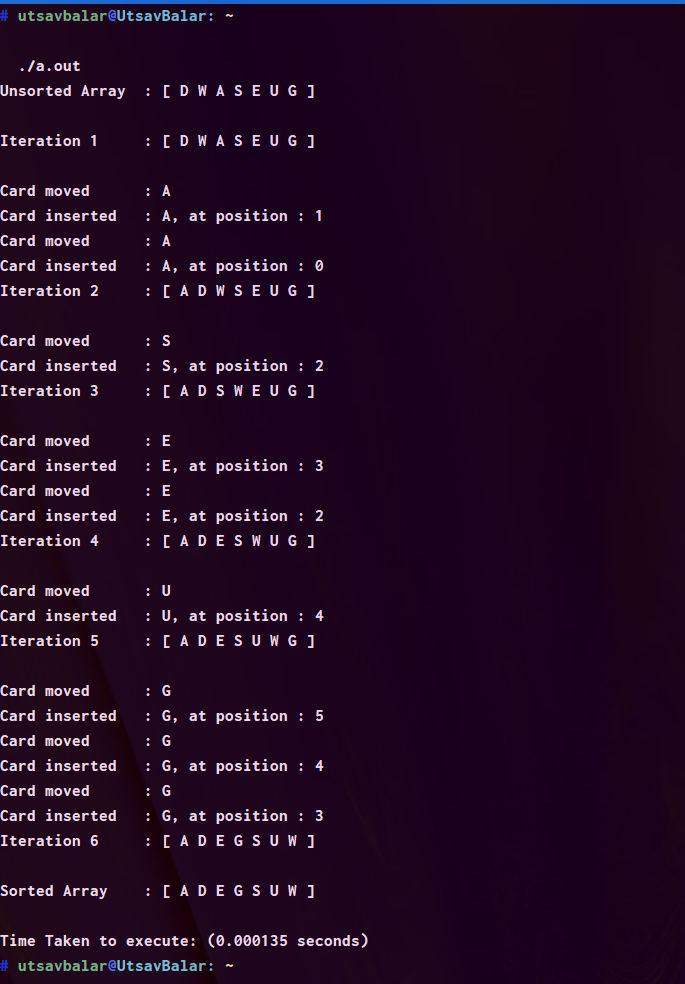
for (int i = 0; i < size; i++)

printf("%s ", array[i]);

printf("]\n");

}

**Output:**



**Practical 3 [Analysis]**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

// Created macro for Random number generation

#define randnum(min, max)\

((rand() % (int)(((max) + 1) - (min))) + (min))

void printarr(int array[], int size);

void sort(int array[], int size);

void swap(int array[], int i, int j);

int main() {

clock\_t t1, t2, t3;

int start = 1, end = 0, i, temp = 0, r;

int \* array;

// Loop until user enters a valid size

while (end <= 0) {

printf("\nPlease enter the size of random array: ");

scanf("%d", & end);

if (end <= 0)

printf("Please enter a number above 0.");

}

int size = end - start + 1;

// Set the size as your input size

array = malloc(sizeof(int) \* size);

if (array == NULL) {

printf("malloc of size %d failed!\n", size);

return -1;

}

// Fill array with numbers in range (start, end)

for (temp = 0, i = start; temp < size; i++, temp++)

array[temp] = i;

// To avoid repetition with each run

srand((unsigned int) time(NULL));

// Store all the random numbers in Array arr[]

for (i = size - 1; i > 0; i--) {

r = randnum(start, end);

swap(array, i, r);

}

printf("Unsorted Array\t: ");

printarr(array, size);

t1 = clock();

sort(array, size);

t1 = clock() - t1;

printf("\nSorted Array: (Average Case): \n");

printarr(array, size);

t2 = clock();

sort(array, size);

t2 = clock() - t2;

printf("\nSorted Array: (Best Case): \n");

printarr(array, size);

// Reverse the numbers in Array arr[]

int x = 0, y = size - 1; // End value of array

while (x < y) {

swap(array, x, y);

x++;

y--;

}

printf("\nUnsorted Array: (Worst Case): \n");

printarr(array, size);

t3 = clock();

sort(array, size);

t3 = clock() - t2;

printf("\nSorted Array: (Worst Case): \n");

printarr(array, size);

printf("\nTime Taken to execute for\n"

"Average Case\t: (%f seconds)\n"

"Best Case\t: (%f seconds)\n"

"Worst Case\t: (%f seconds)\n",

(((float) t1) / CLOCKS\_PER\_SEC),

(((float) t2) / CLOCKS\_PER\_SEC),

(((float) t3) / CLOCKS\_PER\_SEC));

free(array);

return 0;

}

void sort(int array[], int size) {

for (int j = 1; j < size; j++) {

int key = array[j];

int i = j - 1;

for (; i >= 0; i--) {

if (array[i] < key)

break;

array[i + 1] = array[i];

}

array[i + 1] = key;

}

}

void printarr(int array[], int size) {

printf("[ ");

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

printf("%d ", array[i]);

printf("]\n");

}

void swap(int array[], int i, int j) {

int temp = 0;

temp = array[i];

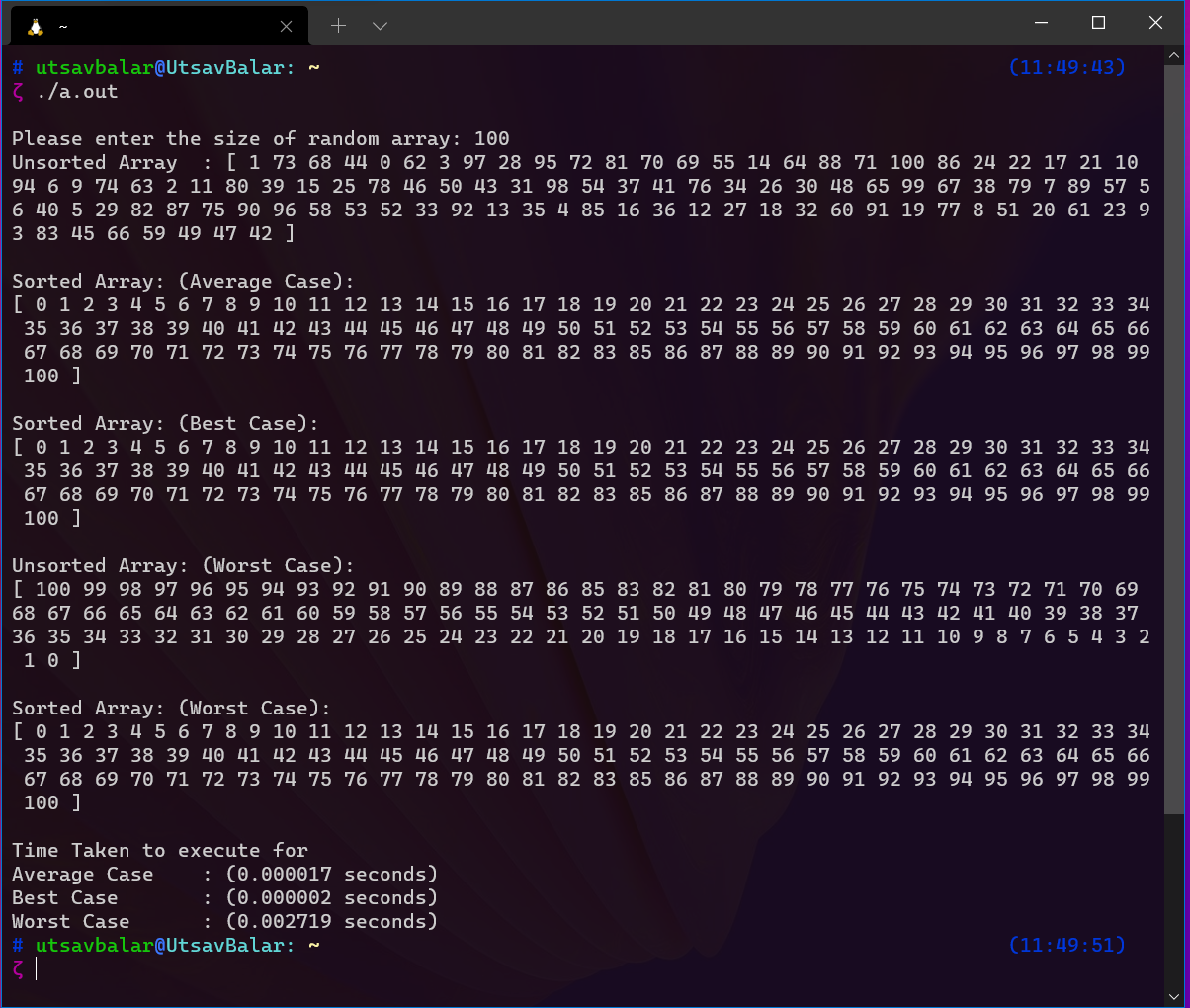
array[i] = array[j];

array[j] = temp;

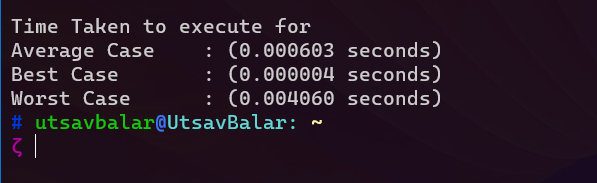
}

**Output:**

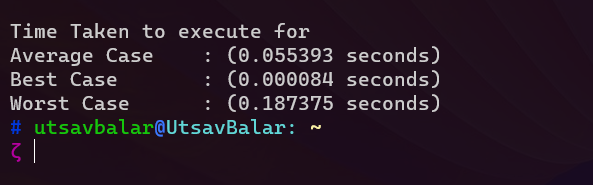
1. 100 Random numbers



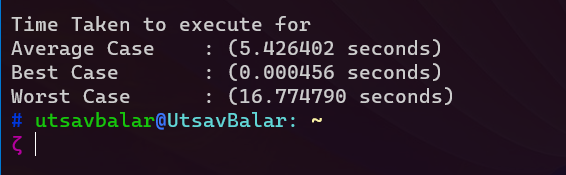
1. 1000 Random numbers



1. 10000 Random numbers



1. 100000 Random numbers



**Practical 3**

**Aim: Creates two empty arrays to hold elements less than the pivot value and elements greater than the pivot value, and then recursively sort the sub arrays. There are two basic operations in the algorithm, swapping items in place and partitioning a section of the array. Take following results and note down your own observation.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| I/P Size | Performance Measurement Model | Avg. Run Time | No. Instruction executed | Observation |
| 100 | Worst |  |  |  |
| 100 | Average |  |  |  |
| 100 | Best |  |  |  |

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Created macro for Random number generation

#define randnum(min, max) \

((rand() % (int)(((max) + 1) - (min))) + (min))

// Function to swap two numbers

void swap(int \*x, int \*y)

{

int temp = \*x;

\*x = \*y;

\*y = temp;

}

// Function to partition array and return the pivot element index

int partition(int arr[], int start, int end)

{

int pivot = arr[end];

int pindex = start - 1;

int i;

for (int i = start; i < end; i++)

{

if (arr[i] <= pivot)

{

pindex++;

swap(&arr[i], &arr[pindex]);

}

}

swap(&arr[pindex + 1], &arr[end]);

return pindex + 1;

}

// Function to sort an array using quicksort algorithm

void quicksort(int arr[], int start, int end)

{

if (start < end)

{

int pindex = partition(arr, start, end);

quicksort(arr, start, pindex - 1);

quicksort(arr, pindex + 1, end);

}

}

// Function to all the elements in an array

void printArr(int array[], int size)

{

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

{

printf("%d ", array[i]);

}

printf("\n");

}

int main()

{

// clock variables to store execution time for each case

clock\_t t1, t2, t3;

int size = 100, i, temp = 0, r;

int \*array;

// Set the size as your input size

array = malloc(sizeof(int) \* size);

if (array == NULL)

{

printf("malloc of size %d failed!\n", size);

return -1;

}

// Fill array with numbers in range (1, size)

for (temp = 0, i = 1; temp < size; i++, temp++)

array[temp] = i;

// To avoid repetition with each run

srand((unsigned int)time(NULL));

// Store all the random numbers in the array

for (i = size - 1; i > 0; i--)

{

r = randnum(1, size);

swap(&array[i], &array[r]);

}

/\*

\* Average Case

\*/

printf("\nUnsorted Array:(Average Case) \t");

printArr(array, size);

// Start measuring time

t1 = clock();

// Call quicksort() function

quicksort(array, 0, size - 1);

// Stop measuring time and calculate the elapsed time

t1 = clock() - t1;

printf("Sorted Array:(Average Case) \t");

printArr(array, size);

/\*

\* Best Case

\*/

printf("\nUnsorted Array:(Best Case) \t");

printArr(array, size);

// Start measuring time

t2 = clock();

// Call quicksort() function

quicksort(array, 0, size - 1);

// Stop measuring time and calculate the elapsed time

t2 = clock() - t2;

printf("Sorted Array:(Best Case) \t");

printArr(array, size);

/\*

\* Worst Case

\*/

// Reverse the numbers in Array arr[]

int x = 0, y = size - 1; // End value of array

while (x < y)

{

swap(&array[x], &array[y]);

x++;

y--;

}

printf("\nUnsorted Array:(Worst Case) \t");

printArr(array, size);

// Start measuring time

t3 = clock();

// Call quicksort() function

quicksort(array, 0, size - 1);

// Stop measuring time and calculate the elapsed time

t3 = clock() - t3;

printf("Sorted Array:(Worst Case) \t");

printArr(array, size);

printf("\nTime Taken to execute for\n"

"Average Case\t: (%f seconds)\n"

"Best Case\t: (%f seconds)\n"

"Worst Case\t: (%f seconds)\n",

(((float)t1) / CLOCKS\_PER\_SEC),

(((float)t2) / CLOCKS\_PER\_SEC),

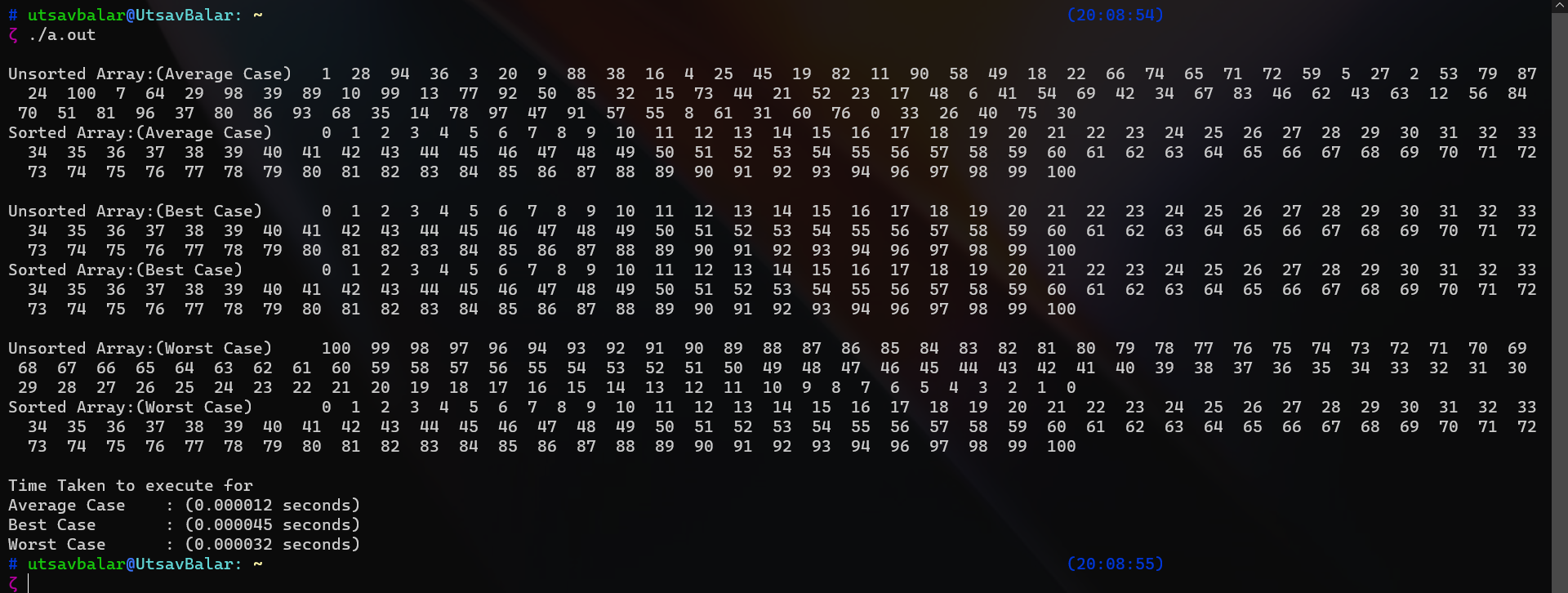
(((float)t3) / CLOCKS\_PER\_SEC));

free(array);

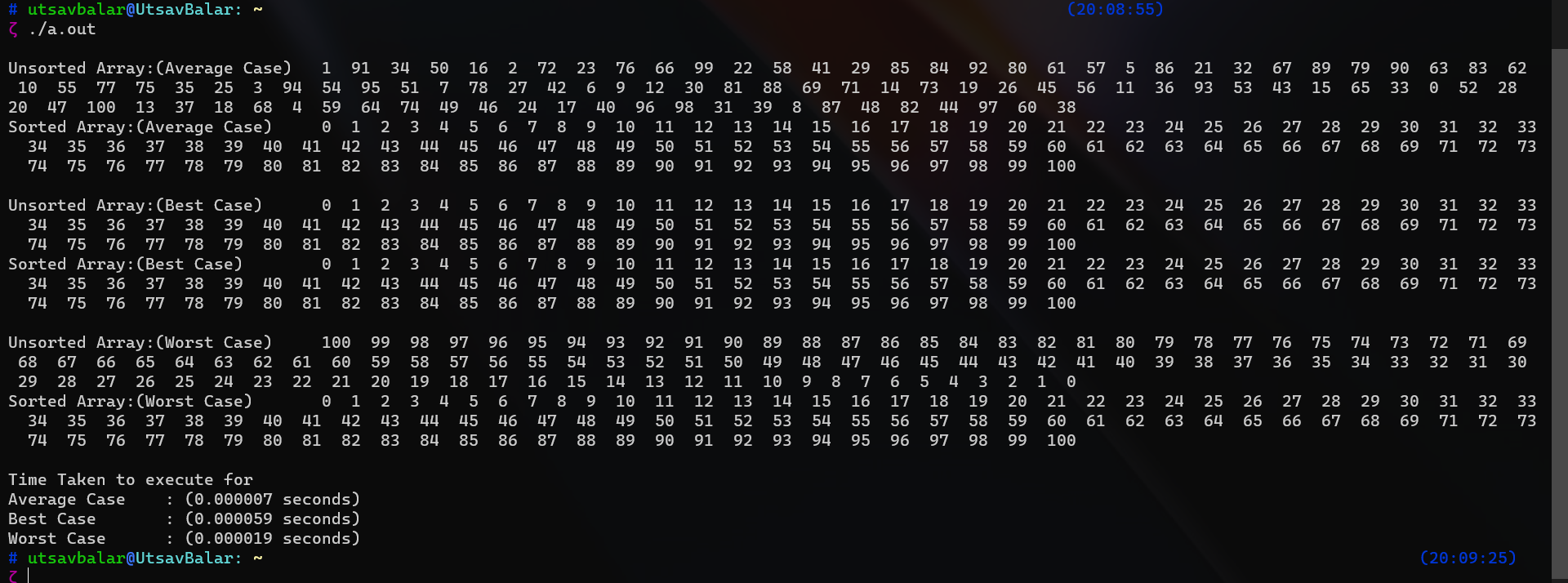
return 0;

}

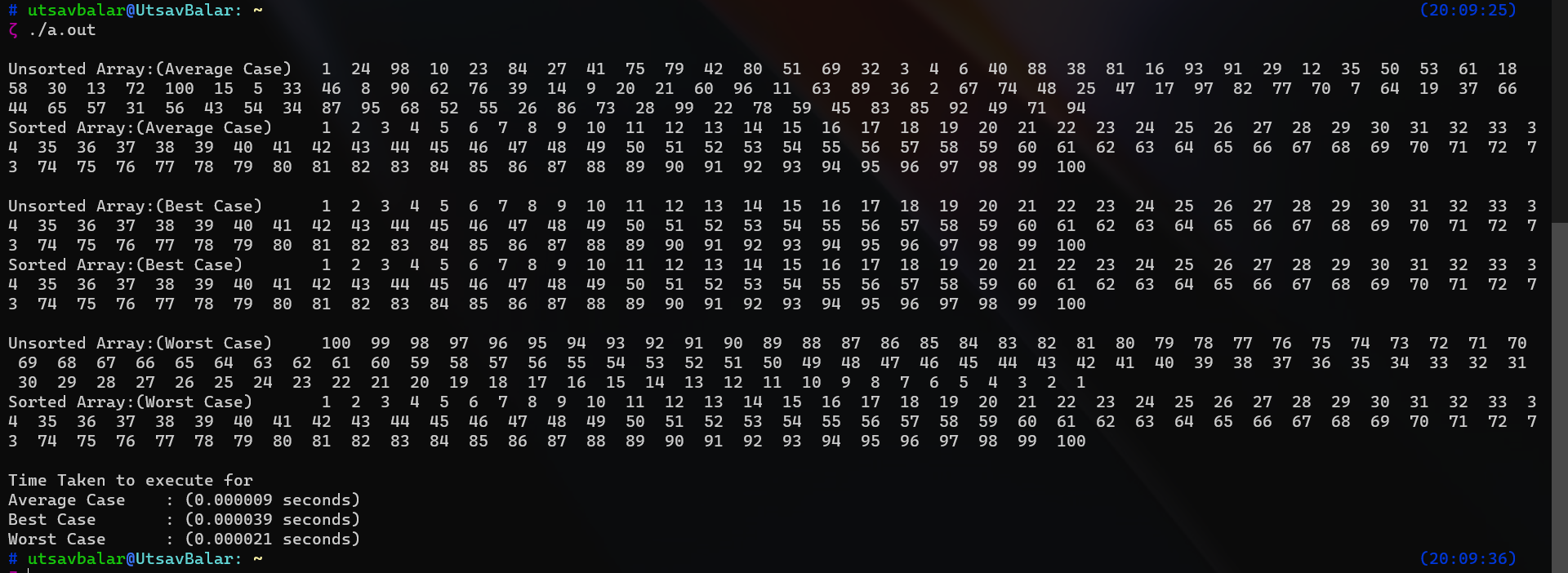
Run1



Run2



Run3



**Analysis of quick sort:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| I/P Size | Performance Measurement Model | Avg. Run Time | No. Instruction executed | Observation |
| 100 | Worst | 0.000024s |  |  |
| 100 | Average | 0.000009s |  |  |
| 100 | Best | 0.000048s |  |  |