**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#include <windows.h>

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

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

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

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

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

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

swap(arr, j, j + 1);

}

}

}

}

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;

}

}

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

int i, j, min\_idx;

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

min\_idx = i;

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

if (arr[j] < arr[min\_idx]) {

min\_idx = j;

}

}

swap(arr, i, min\_idx);

}

}

int main() {

int n;

printf("Enter the length of the array (n): ");

scanf("%d", &n);

srand(GetTickCount());

int arr[n];

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

arr[i] = rand() % (5 \* n + 1);

}

LARGE\_INTEGER start, end, frequency;

QueryPerformanceFrequency(&frequency);

QueryPerformanceCounter(&start);

bubbleSort(arr, n);

QueryPerformanceCounter(&end);

LARGE\_INTEGER start1, end1, frequency1;

QueryPerformanceFrequency(&frequency1);

QueryPerformanceCounter(&start1);

insertionSort(arr, n);

QueryPerformanceCounter(&end1);

LARGE\_INTEGER start2, end2, frequency2;

QueryPerformanceFrequency(&frequency2);

QueryPerformanceCounter(&start2);

selectionSort(arr, n);

QueryPerformanceCounter(&end2);

double elapsed\_time = ((double)(end.QuadPart - start.QuadPart) / frequency.QuadPart) \* 1000.0;

double elapsed\_time1 = ((double)(end1.QuadPart - start1.QuadPart) / frequency1.QuadPart)\* 1000.0;

double elapsed\_time2 = ((double)(end2.QuadPart - start2.QuadPart) / frequency2.QuadPart) \* 1000.0;

printf("Execution Time Of BUBBLE SORT: %f milliseconds\n", elapsed\_time);

printf("Execution Time Of INSERTION SORT: %f milliseconds\n", elapsed\_time1);

printf("Execution Time Of SELECTION SORT: %f milliseconds\n", elapsed\_time2);

}

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#include <windows.h>

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

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

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

int pivot = arr[high];

int i = low - 1;

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

if (arr[j] < pivot) {

i++;

swap(arr, i, j);

}

}

swap(arr, i + 1, high);

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);

}

}

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++;

}

}

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

if (l < r) {

// Same as (l+r)/2, but avoids overflow for large l and r

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

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

void heapify(int arr[], int n, int i) {

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < n && arr[left] > arr[largest]) {

largest = left;

}

if (right < n && arr[right] > arr[largest]) {

largest = right;

}

if (largest != i) {

swap(arr, i, largest);

heapify(arr, n, largest);

}

}

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

for (int i = n / 2 - 1; i >= 0; i--) {

heapify(arr, n, i);

}

for (int i = n - 1; i > 0; i--) {

swap(arr, 0, i);

heapify(arr, i, 0);

}

}

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

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

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

}

printf("\n");

}

int main() {

int n;

printf("Enter the length of the array (n): ");

scanf("%d", &n);

srand(GetTickCount());

int arr[n];

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

arr[i] = rand() % (5 \* n + 1);

}

LARGE\_INTEGER start, end, frequency;

QueryPerformanceFrequency(&frequency);

QueryPerformanceCounter(&start);

quicksort(arr, 0, n - 1);

QueryPerformanceCounter(&end);

LARGE\_INTEGER start1, end1, frequency1;

QueryPerformanceFrequency(&frequency1);

QueryPerformanceCounter(&start1);

mergeSort(arr, 0, n - 1);

QueryPerformanceCounter(&end1);

LARGE\_INTEGER start2, end2, frequency2;

QueryPerformanceFrequency(&frequency2);

QueryPerformanceCounter(&start2);

heapSort(arr, n);

QueryPerformanceCounter(&end2);

double elapsed\_time = ((double)(end.QuadPart - start.QuadPart) / frequency.QuadPart) \* 1000.0;

double elapsed\_time1 = ((double)(end1.QuadPart - start1.QuadPart) / frequency1.QuadPart) \* 1000.0;

double elapsed\_time2 = ((double)(end2.QuadPart - start2.QuadPart) / frequency2.QuadPart) \* 1000.0;

printf("Execution Time Of QUICK SORT: %f milliseconds\n", elapsed\_time);

printf("Execution Time Of MERGE SORT: %f milliseconds\n", elapsed\_time1);

printf("Execution Time Of HEAP SORT: %f milliseconds\n", elapsed\_time2);

}

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#include <windows.h>

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

int max = arr[0];

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

if (arr[i] > max) {

max = arr[i];

}

}

int\* count = (int\*)malloc((max + 1) \* sizeof(int));

if (count == NULL) {

fprintf(stderr, "Memory allocation failed\n");

exit(EXIT\_FAILURE);

}

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

count[i] = 0;

}

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

count[arr[i]]++;

}

for (int i = 1; i <= max; i++) {

count[i] += count[i - 1];}

int\* output = (int\*)malloc(n \* sizeof(int));

if (output == NULL) {

fprintf(stderr, "Memory allocation failed\n");

free(count);

exit(EXIT\_FAILURE);

}

for (int i = n - 1; i >= 0; i--) {

output[count[arr[i]] - 1] = arr[i];

count[arr[i]]--;

}

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

arr[i] = output[i];

}

free(count);

free(output);

}

int getMax(int arr[], int n) {

int max = arr[0];

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

if (arr[i] > max) {

max = arr[i];

}

}

return max;

}

void countingSort(int arr[], int n, int exp) {

const int base = 10;

int\* output = (int\*)malloc(n \* sizeof(int));

if (output == NULL) {

fprintf(stderr, "Memory allocation failed\n");

exit(EXIT\_FAILURE);

}

int\* count = (int\*)malloc(base \* sizeof(int));

if (count == NULL) {

fprintf(stderr, "Memory allocation failed\n");

free(output);

exit(EXIT\_FAILURE);

}

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

count[i] = 0;

}

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

count[(arr[i] / exp) % base]++;

}

for (int i = 1; i < base; i++) {

count[i] += count[i - 1];

}

for (int i = n - 1; i >= 0; i--) {

output[count[(arr[i] / exp) % base] - 1] = arr[i];

count[(arr[i] / exp) % base]--;

}

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

arr[i] = output[i];

}

free(output);

free(count);

}

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

int max = getMax(arr, n);

for (int exp = 1; max / exp > 0; exp \*= 10) {

countingSort(arr, n, exp);

}

}

int main() {

int n;

printf("Enter the length of the array (n): ");

scanf("%d", &n);

srand(GetTickCount());

int arr[n];

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

arr[i] = rand() % (5 \* n + 1);

}

LARGE\_INTEGER start, end, frequency;

QueryPerformanceFrequency(&frequency);

QueryPerformanceCounter(&start);

countSort(arr, n);

QueryPerformanceCounter(&end);

double elapsed\_time = ((double)(end.QuadPart - start.QuadPart) / frequency.QuadPart) \* 1000.0;

printf("Execution Time of Count Sort: %f milliseconds\n", elapsed\_time);

LARGE\_INTEGER start1, end1, frequency1;

QueryPerformanceFrequency(&frequency1);

QueryPerformanceCounter(&start1);

radixSort(arr, n);

QueryPerformanceCounter(&end1);

double elapsed\_time1 = ((double)(end1.QuadPart - start1.QuadPart) / frequency1.QuadPart) \* 1000.0;

printf("Execution Time of radix Sort: %f milliseconds\n", elapsed\_time);

}