```
//for Standard IO
#include <stdio.h>
//for dynamic allocation, FILE IO, and random
#include <stdlib.h>
//for Random
#include <time.h>
//To use C style string more easily
#include <string.h>
//array type
#define array_t char*
//Compare count or Swap count sometime overflow integer
#define int_s unsigned int
//For using is-- varaible
#define true 1
#define false 0
//For using max char_num
#define MAX 20

//handle error with message
void Error(const char* msg){
    fprintf(stderr, "[Error]: %s\n", msg);
    exit(1);
}

//for Testing array
void print_array(array_t* array, int_s size){
    printf("\n");
    for(int_s i = 0 ; i < size; i++)
        printf("%s\t", array[i]);

    printf("\n\n");
}

//initilaze array value for random character
void init_array(array_t* array, int_s size, int char_num){
    for(int_s i = 0 ; i < size; i++){
        for(int j = 0 ; j < char_num - 1; j++)
            (array[i])[j] = 'a' + rand() % 25 + 0;
        (array[i])[char_num - 1] = '\0';
    }
//checking random array is gernerated well(when submin, erase this section)
//  print_array(array, size);
}

//For more inuitive coding
enum array_index{bubble = 0, insert, quick, merge, heap};

//save information in struct
typedef struct array_info{
    char* array_name;
    array_t* result;
    double do_time;
    int_s swap_count;
    int_s compare_count;
    int isStable;
}array_i;

//Duplicate random value array (To sorting array)
array_t* Duplicate(array_t* array, int_s size, int char_num)
{
    array_t* new_array = (array_t*)malloc(size * sizeof(array_t));
    if(new_array == NULL)
        Error("Failed to allocate Memory");

    for(int_s i = 0; i < size; i++){
        new_array[i] = (array_t)malloc(char_num);
        strncpy(new_array[i], array[i], char_num - 1);
        new_array[i][char_num - 1] = '\0';
    }
    return new_array;
}

//To Count swap count
int swap(array_t x, array_t y)
{
    array_t temp = (array_t)malloc(strlen(x) + 1);
        strcpy(temp, x);
    strcpy(x, y);
    strcpy(y, temp);

    x[strlen(x)] = '\0';
    y[strlen(y)] = '\0';

    free(temp);

    return 1;
}

//print out functions
void print_result(array_i* ai, int_s size);
int file_w(array_i* ai, int_s size);

//Sorting Algorithms
array_i Bubble(array_t* array, int_s size, int char_num)
{
    array_t* sort_array = Duplicate(array, size, char_num);
    int_s s_count = 0, c_count = 0;
    clock_t start, end;

    printf("=>Starting - Bubble Sort\n");
    start = clock();

    for(int_s i = 0; i < size - 1; i++)
    {
        for(int_s j = 0; j < size - 1; j++)
        {
            c_count++;
            if(strncmp(sort_array[j], sort_array[j + 1], char_num - 1) > 0)
                s_count += swap(sort_array[j], sort_array[j + 1]);
        }
    }

    end = clock();
    printf("=>Finished\n");

    array_i ai;
    ai.array_name = "Bubble";
    ai.do_time = (double)(end - start) / CLOCKS_PER_SEC;
    ai.swap_count = s_count;
    ai.compare_count = c_count;
    ai.isStable = true;

    file_w(&ai, size);

//checking sorting is working well (before submit, erase this section)
    //print_array(sort_array, size);
    free(sort_array);

    return ai;
}

array_i Insert(array_t* array, int_s size, int char_num)
{
    array_t* sort_array = Duplicate(array, size, char_num);
    int_s s_count = 0, c_count = 0;
    int i, j;
    array_t key = (array_t)malloc(char_num);
    if(key == NULL)
        Error("Failed to allocate Memeory");

    clock_t start, end;

    printf("=> Starting - Insert Sort\n");
    start = clock();
    for(i = 1; i < (int)size; i++)
    {
        strncpy(key, sort_array[i], char_num - 1);
        key[char_num - 1] = '\0';
        (sort_array[i])[char_num - 1] = '\0';

        for(j = i - 1; j >= 0 && strncmp(sort_array[j], key, char_num - 1) > 0; j-
-)
        {
            strncpy(sort_array[j + 1], sort_array[j], char_num - 1);
            (sort_array[j + 1])[char_num - 1] = '\0';
            (sort_array[j])[char_num - 1] = '\0';
            s_count++;
            c_count++;
        }
```

```
            c_count++;
            strncpy(sort_array[j + 1], key, char_num - 1);
            s_count++;
        }
        end = clock();
        free(key);
        printf("=> Finished\n");

        array_i ai;
        ai.array_name = "Insert";
        ai.do_time = (double)(end - start) / CLOCKS_PER_SEC;
        ai.swap_count = s_count;
        ai.compare_count = c_count;
        ai.isStable = true;

    //checking sorting is working well (before submit, erase this section)
        //print_array(sort_array, size);
        free(sort_array);

        file_w(&ai, size);

        return ai;
    }

    int partition(array_t* arr, int low, int high, int char_num, int* c_count, int*
    s_count) {
        array_t pivot = arr[high];
        int i = low - 1;
        for(int j = low; j < high; j++) {
            (*c_count)++;
            if(strncmp(arr[j], pivot, char_num - 1) <= 0) {
                i++;
                *s_count += swap(arr[i], arr[j]);
            }
        }
        *s_count += swap(arr[i + 1], arr[high]);
        return i + 1;
    }

    // Recursive quicksort
    void quicksort(array_t* arr, int low, int high, int char_num, int* c_count, int*
    s_count) {
        if(low < high) {
            int pi = partition(arr, low, high, char_num, c_count, s_count);
            quicksort(arr, low, pi - 1, char_num, c_count, s_count);
            quicksort(arr, pi + 1, high, char_num, c_count, s_count);
        }
    }

    array_i Quick(array_t* array, int_s size, int_s char_num)
    {
        array_t* sort_array = Duplicate(array, size, char_num);
        int c_count = 0, s_count = 0;
        clock_t start, end;
        printf("=> Starting - Quick Sorting\n");
        start = clock();

        quicksort(sort_array, 0, size - 1, char_num, &c_count, &s_count);

        end = clock();
        printf("=> Finished\n");

        array_i ai;
        ai.array_name = "Quick";
        ai.do_time = (double)(end - start) / CLOCKS_PER_SEC;
        ai.swap_count = s_count;
        ai.compare_count = c_count;
        ai.isStable = false;

    //checking sorting is working well (before submit, erase this section)
    //  print_array(sort_array, size);
        free(sort_array);

        file_w(&ai, size);

        return ai;
    }

    void merge_merge(array_t* arr, int left, int mid, int right, int char_num, int*
    c_count, int* s_count) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        array_t* L = (array_t*)malloc(n1 * sizeof(array_t));
        array_t* R = (array_t*)malloc(n2 * sizeof(array_t));
        for(int i = 0; i < n1; i++) {
            L[i] = (array_t)malloc(char_num);
            strncpy(L[i], arr[left + i], char_num - 1);
            L[i][char_num - 1] = '\0';
        }
        for(int j = 0; j < n2; j++) {
            R[j] = (array_t)malloc(char_num);
            strncpy(R[j], arr[mid + 1 + j], char_num - 1);
            R[j][char_num - 1] = '\0';
        }

        int i = 0, j = 0, k = left;
        while(i < n1 && j < n2) {
            (*c_count)++;
            if(strncmp(L[i], R[j], char_num - 1) <= 0) {
                if(strcmp(arr[k], L[i]) != 0) (*s_count)++;
                strncpy(arr[k], L[i], char_num - 1);
                arr[k][char_num - 1] = '\0';
                i++;
            } else {
                if(strcmp(arr[k], R[j]) != 0) (*s_count)++;
                strncpy(arr[k], R[j], char_num - 1);
                arr[k][char_num - 1] = '\0';
                j++;
            }
            k++;
        }
        while(i < n1) {
            if(strcmp(arr[k], L[i]) != 0) (*s_count)++;
            strncpy(arr[k], L[i], char_num - 1);
            arr[k][char_num - 1] = '\0';
            i++; k++;
        }
        while(j < n2) {
            if(strcmp(arr[k], R[j]) != 0) (*s_count)++;
            strncpy(arr[k], R[j], char_num - 1);
            arr[k][char_num - 1] = '\0';
            j++; k++;
        }
        for(int i = 0; i < n1; i++) free(L[i]);
        for(int j = 0; j < n2; j++) free(R[j]);
        free(L); free(R);
    }

    // Recursive merge sort
    void merge_sort(array_t* arr, int left, int right, int char_num, int* c_count,
    int* s_count) {
        if(left < right) {
            int mid = left + (right - left) / 2;
            merge_sort(arr, left, mid, char_num, c_count, s_count);
            merge_sort(arr, mid + 1, right, char_num, c_count, s_count);
            merge_merge(arr, left, mid, right, char_num, c_count, s_count);
        }
    }

    array_i Merge(array_t* array, int_s size, int char_num)
    {
        array_t* sort_array = Duplicate(array, size, char_num);
        int c_count = 0, s_count = 0;
        clock_t start, end;
        printf("=> Starting - Merge Sorting\n");
        start = clock();

        merge_sort(sort_array, 0, size - 1, char_num, &c_count, &s_count);

        end = clock();
        printf("=> Finished\n");

        array_i ai;
        ai.array_name = "Merge";
        ai.do_time = (double)(end - start) / CLOCKS_PER_SEC;
        ai.swap_count = s_count;
        ai.compare_count = c_count;
        ai.isStable = true;

        file_w(&ai, size);
```

```c
//checking sorting is working well (before submit, erase this section)
//  print_array(sort_array, size);
    free(sort_array);

    return ai;
}

// Heapify function for string arrays
void heapify(array_t* arr, int_s n, int_s i, int char_num, int* c_count, int*
s_count) {
    int_s largest = i;
    int_s l = 2 * i + 1;
    int_s r = 2 * i + 2;

    if (l < n) {
        (*c_count)++;
        if (strncmp(arr[l], arr[largest], char_num - 1) > 0)
            largest = l;
    }
    if (r < n) {
        (*c_count)++;
        if (strncmp(arr[r], arr[largest], char_num - 1) > 0)
            largest = r;
    }
    if (largest != i) {
        *s_count += swap(arr[i], arr[largest]);
        heapify(arr, n, largest, char_num, c_count, s_count);
    }
}

array_i Heap(array_t* array, int_s size, int char_num)
{
    array_t* sort_array = Duplicate(array, size, char_num);
    int c_count = 0, s_count = 0;
    clock_t start, end;
    printf("=> Starting - Heap Sorting\n");
    start = clock();

    // Build heap (rearrange array)
    for (int i = (int)size / 2 - 1; i >= 0; i--)
        heapify(sort_array, size, i, char_num, &c_count, &s_count);

    // One by one extract elements from heap
    for (int i = (int)size - 1; i > 0; i--) {
        s_count += swap(sort_array[0], sort_array[i]);
        heapify(sort_array, i, 0, char_num, &c_count, &s_count);
    }

    end = clock();
    printf("=> Finished\n");

    array_i ai;
    ai.array_name = "Heap";
    ai.do_time = (double)(end - start) / CLOCKS_PER_SEC;
    ai.swap_count = s_count;
    ai.compare_count = c_count;
    ai.isStable = false;

//checking sorting is working well (before submit, erase this section)
//  print_array(sort_array, size);
    free(sort_array);

    file_w(&ai, size);

    return ai;
}


int main(void)
{
//for random value
    srand((int_s)time(NULL));

//FILE Save more good to see
//  system("rm -rf Result");
//  system("mkdir Result");
    int_s size;

//Size of array
    printf("Number of Instances: ");
    if(!(scanf("%u", &size)))
        Error("Invalid Input");

//array for take random value (sorting destination)
    array_t* array = (array_t*)malloc(size * sizeof(array_t));
    if(array == NULL)
        Error("Failed to allocate Memory");

//dynamic character size
    int char_num = rand()% MAX + 1;
    char_num++;

//allocate size to array
    for(int_s i = 0; i < size; i++){
        array[i] = (array_t)malloc(char_num);
        if(array[i] == NULL){
            free(array);
            for(int_s j = 0; j < i; j++)
                free(array[j]);
            Error("Failed to allocate Memory");
        }
    }

//initialize array for random string value
    init_array(array, size, char_num);

//array information struct save
    array_i* ai = (array_i*)malloc(sizeof(array_i) * 5);
    if(ai == NULL)
        Error("Failed to Allocate Memory");

//Do Sorting
    ai[bubble] = Bubble(array, size, char_num);
    ai[insert] = Insert(array, size, char_num);
    ai[quick] = Quick(array, size, char_num);
    ai[merge] = Merge(array, size, char_num);
    ai[heap] = Heap(array, size, char_num);

//Print out Result
    print_result(ai, size);
    free(ai);

    return 0;
}

void print_result(array_i* array, int_s size)
{
    printf("==== Sorting Result Summary ===\n");
    if(size < 1000){
        printf("Algorithm\t|\tTimes(s)\t|\tCompare\t|\tSwap\t|\tStable\n");
        printf("-----------------------------------------------------------------
--------------------------\n");
        for(int i = 0; i < 5; i++)
        {
            if(array[i].array_name == NULL)
                break;
            printf("%s Sort\t|\t%.6lf\t|\t%3u\t|\t%3u\t|\t", array[i].array_name,
array[i].do_time, array[i].compare_count, array[i].swap_count);
            if(array[i].isStable == 1)
                printf("YES");
            else
                printf("NO");
            printf("\n");
        }
    }
    else
    {
        printf("Algorithm\t|\tTimes(s)\t|\tCompare\t  \t|\tSwap\t\t|\tStable\n");
        printf("-----------------------------------------------------------------
----------------------------------------\n");
        for(int i = 0; i < 5; i++)
        {
            if(array[i].array_name == NULL)
                break;
            printf("%s Sort\t|\t%.6lf\t|\t%10u\t|\t%10u\t|\t",
array[i].array_name, array[i].do_time, array[i].compare_count,
array[i].swap_count);
            if(array[i].isStable == 1)
                printf("YES");
            else
                printf("NO");
            printf("\n");
        }
    }
    return;
}

int file_w(array_i* array, int_s size)
{
    FILE* fp;
    char* file_name;

    if(array->array_name == NULL)
        return false;
    file_name = (char*)malloc(strlen(array->array_name) + 9 + 1 /*+
strlen("Result/")*/);
    if(file_name == NULL)
        Error("Failed to allocate Memory");

    //make file name for dynamically
    /*strncpy(file_name, "Result/", strlen("Result/"));
    file_name[strlen("Result/")] = '\0';*/
    strncpy(file_name, array->array_name, strlen(array->array_name));
    file_name[strlen(array->array_name) /*+ strlen("Result/")*/] = '\0';
    file_name = strcat(file_name, "_sort.out");

    //make lower case
    if(file_name[0/*strlen("Result/")*/] >= 'A' &&
file_name[0/*strlen("Result/")*/] <= 'Z')
        file_name[0/*strlen("Result/")*/] += 'a' - 'A';

    fp =  fopen(file_name, "w");
    if(fp == NULL)
        Error("Failed to open FILE");


    if(size < 1000)
    {
        //Type out Menu
        fprintf(fp, "==== Sorting Result Summary ===\n");
        fprintf(fp, "Algorithm\t|\tTimes(s)\t|\tCompare\t|\tSwap\t|\tStable\n");
        fprintf(fp, "--------------------------------------------------------------
------------------------------\n");

        //Type out Array Algorithms
        fprintf(fp, "%s Sort\t|\t", array->array_name);

        //Type out Algorithms times
        fprintf(fp, "%.6lf\t|\t", array->do_time);

        //Type out Algorithmes compare count
        fprintf(fp, "%3d\t|\t", array->compare_count);

        //Type out Algorithms swap count
        fprintf(fp, "%3d\t|\t", array->swap_count);
        //is Stable Algorithm or not
        if(array->isStable == 1)
            fprintf(fp, "YES");
        else
            fprintf(fp, "NO");
        fprintf(fp, "\n");
    }
    else
    {
        fprintf(fp, "==== Sorting Result Summary ===\n");
        fprintf(fp, "Algorithm\t|\tTimes(s)\t|\tCompare\t
\t|\tSwap\t\t|\tStable\n");
        fprintf(fp, "--------------------------------------------------------------
--------------------------------------------\n");

        fprintf(fp, "%s Sort\t|\t", array->array_name);
        fprintf(fp, "%.6lf\t|\t", array->do_time);
        fprintf(fp, "%10u\t|\t", array->compare_count);
        fprintf(fp, "%10u\t|\t", array->swap_count);

        if(array->isStable == true)
            fprintf(fp, "Yes");
        else
            fprintf(fp, "No");
        fprintf(fp ,"\n");
    }

    fclose(fp);

    printf("=> Created - %s\n", file_name);

    free(file_name);

    return true;
}
```