COS10007

Week 7 Prac

Question 1

Week 7 Question 1

Using the array "data" as the example, explain the logic of the following

sorting techniques.

array = {10, 2, 9, 12, 6};

a. Bubble sort

10 2 9 12 6

2 10 9 12 6

2 9 10 12 6

2 9 10 6 12

2 9 6 10 12

2 6 9 10 12

b. Selection sort

10 2 9 12 6

2 10 9 12 6

2 6 9 12 10

2 6 9 12 10

2 6 9 10 12

c. Insertion sort

10 2 9 12 6

2 10 9 12 6

2 9 10 12 6

2 6 9 10 12

d. Merge sort

10 2 9 12 6

(10 2 9) (12 6)

(10 2) (9) (12 6)

(2 10) (9) (6 12)

(2 9 10) (6 12)

2 6 9 10 12

e. Quick sort

10 2 9 12 6

(10 2 9) (12 6)

(10 2 9) (12 6)

(2 9 10) (6 12)

2 6 9 10 12

Question 2

#include <stdio.h>

#include <string.h>

#include <time.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdbool.h>

// !!! FIXED PART OF QUESTION !!!

struct student {

char name[10];

int rank;

};

struct student Students[5];

// !!! END FIXED PART OF QUESTION !!!

typedef struct student Student;

// Make a more interesting array.

#define PEOPLE\_LEN 20

Student People[PEOPLE\_LEN];

Student People\_backup[PEOPLE\_LEN];

// Functions for assigning default values at start of execution.

void assign\_values(const size\_t apos, const char \* n, const int r);

void assign\_values\_people(const size\_t apos, const char \* n, const int r);

// Functions for reseting values to defaults before executing sorting functions.

void reset\_values(void);

void reset\_values\_people(void);

// Calls assign\_values\_people a bunch to setup for later.

void initialize\_values\_people(void);

// Print the contents of an array of type Student.

void print\_values(const Student \* s, size\_t s\_len);

// Sorting algorithms

Student \* selection\_sort(Student \* s, const size\_t s\_len);

Student \* insertion\_sort(Student \* s, const size\_t s\_len);

Student \* merge\_sort(Student \* s, const size\_t s\_len);

Student \* quick\_sort(Student \* s, const size\_t s\_len);

int main(void) {

// Initialize random number generator for later when we run the functions on a larger sample.

srandom(time(NULL));

// Unsorted

reset\_values();

printf("Values before sorting:\n");

print\_values(Students, sizeof(Students) / sizeof(\*Students));

printf("\n");

// Selction

reset\_values();

memcpy(Students, selection\_sort(Students, sizeof(Students) / sizeof(\*Students)), sizeof(Students));

printf("Values after selection sort:\n");

print\_values(Students, sizeof(Students) / sizeof(\*Students));

printf("\n");

// Insertion

reset\_values();

memcpy(Students, insertion\_sort(Students, sizeof(Students) / sizeof(\*Students)), sizeof(Students));

printf("Values after insertion sort:\n");

print\_values(Students, sizeof(Students) / sizeof(\*Students));

printf("\n");

// Merge

reset\_values();

memcpy(Students, merge\_sort(Students, sizeof(Students) / sizeof(\*Students)), sizeof(Students));

printf("Values after merge sort:\n");

print\_values(Students, sizeof(Students) / sizeof(\*Students));

printf("\n");

// Quick

reset\_values();

memcpy(Students, quick\_sort(Students, sizeof(Students) / sizeof(\*Students)), sizeof(Students));

printf("Values after quick sort:\n");

print\_values(Students, sizeof(Students) / sizeof(\*Students));

printf("\n");

/\*\*

sleep(5);

// Different array with PEOPLE\_LEN x random values this time.

// Unsorted

initialize\_values\_people();

reset\_values\_people();

printf("New values before sorting:\n");

print\_values(People, sizeof(People) / sizeof(\*People));

printf("\n");

// Selction

reset\_values\_people();

memcpy(People, selection\_sort(People, sizeof(People) / sizeof(\*People)), sizeof(People));

printf("Values after selection sort:\n");

print\_values(People, sizeof(People) / sizeof(\*People));

printf("\n");

// Insertion

reset\_values\_people();

memcpy(People, insertion\_sort(People, sizeof(People) / sizeof(\*People)), sizeof(People));

printf("Values after insertion sort:\n");

print\_values(People, sizeof(People) / sizeof(\*People));

printf("\n");

// Merge

reset\_values\_people();

memcpy(People, merge\_sort(People, sizeof(People) / sizeof(\*People)), sizeof(People));

printf("Values after merge sort:\n");

print\_values(People, sizeof(People) / sizeof(\*People));

printf("\n");

// Quick

reset\_values\_people();

memcpy(People, quick\_sort(People, sizeof(People) / sizeof(\*People)), sizeof(People));

printf("Values after quick sort:\n");

print\_values(People, sizeof(People) / sizeof(\*People));

printf("\n");

/\*\*/

return 0;

}

void assign\_values(const size\_t apos, const char \* n, const int r) {

strcpy(Students[apos].name, n);

Students[apos].rank = r;

}

void assign\_values\_people(const size\_t apos, const char \* n, const int r) {

strcpy(People\_backup[apos].name, n);

People\_backup[apos].rank = r;

}

void reset\_values(void) {

assign\_values(0, "David", 10);

assign\_values(1, "James", 2);

assign\_values(2, "Bruce", 9);

assign\_values(3, "Jane", 12);

assign\_values(4, "Edith", 6);

}

void initialize\_values\_people(void) {

for(int i = 0; i < PEOPLE\_LEN; i++) {

assign\_values\_people(i, "A", random() % 100);

}

}

void reset\_values\_people(void) {

memcpy(People, People\_backup, sizeof(People));

}

void print\_values(const Student \* s, size\_t s\_len) {

for(int i = 0; i < s\_len; i++) {

printf("Name: %s\t", s[i].name);

printf("Rank: ");

// Keep all of the numbers lined up.

if(s[i].rank < 10) {printf(" ");}

printf("%d\n", s[i].rank);

}

}

// Question 2 Part a

Student \* selection\_sort(Student \* s, const size\_t s\_len) {

// Iterate position in array.

for(int i = 0; i < s\_len; i++) {

int minimum = s[i].rank;

size\_t minpos = i;

// Find the next lowest value.

for(int j = i + 1; j < s\_len; j++) {

if(s[j].rank < minimum) {

minpos = j;

minimum = s[minpos].rank;

}

}

// Switch the position of the values.

Student temp = s[i];

s[i] = s[minpos];

s[minpos] = temp;

}

return s;

}

// Question 2 Part b

Student \* insertion\_sort(Student \* s, const size\_t s\_len) {

// Iterate through array.

for(int i = 0; i < s\_len; i++) {

// Shuffle items down list till we find the correct location.

for(int j = i + 1; s[j].rank < s[j - 1].rank && j < s\_len; j--) {

Student temp = s[j];

s[j] = s[j - 1];

s[j - 1] = temp;

}

}

return s;

}

// Question 2 Part c

Student \* merge\_sort(Student \* s, const size\_t s\_len) {

// Sort two arrays into one with the assumption that they're either of length 1 or are already internally

// sorted.

Student \* conquer( Student \* merge,

const size\_t merge\_len,

const Student \* left,

const size\_t left\_len,

const Student \* right,

const size\_t right\_len) {

size\_t left\_pos = 0, right\_pos = 0;

// Merge the arrays into one array.

for(int i = 0; i < merge\_len; i++) {

if( left[left\_pos].rank < right[right\_pos].rank &&

left\_pos != left\_len ||

right\_pos == right\_len) {

merge[i] = left[left\_pos++];

} else {

merge[i] = right[right\_pos++];

}

}

/\* Debug info \*

printf("Merged values:\n");

print\_values(merge, merge\_len);

printf("\n");

usleep(500000);

/\*\*/

return merge;

}

// Divide arrays recursively into smaller arrays eventually of either 1 or two values.

Student \* divide(Student \* split, const size\_t arr\_len) {

// Find midpoint.

const size\_t midpoint = arr\_len / 2;

// If there is only one value in the array just return it.

if(arr\_len <= 1) {return split;}

// Work on left half.

const size\_t left\_len = midpoint;

Student left[left\_len];

// Copy left contents.

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

left[i] = split[i];

}

// Recursively call this function again.

memcpy(left, divide(left, left\_len), sizeof(left) / sizeof(\*left));

// Work on right half.

const size\_t right\_len = arr\_len - midpoint;

Student right[right\_len];

// Copy right contents.

for(int i = midpoint; i < arr\_len; i++) {

right[i - midpoint] = split[i];

}

// Recursively call this function again.

memcpy(right, divide(right, right\_len), sizeof(right) / sizeof(\*right));

// Join halves together.

return conquer(split, arr\_len, left, left\_len, right, right\_len);

}

return divide(s, s\_len);

}

// Question 2 Part d

Student \* quick\_sort(Student \* s, const size\_t s\_len) {

void swap(Student \* a, Student \* b) {

Student temp = \*a;

\*a = \*b;

\*b = temp;

}

int parsechunk(Student \* s, int lo, int hi) {

// Save the pivot point.

Student pivot = s[hi];

// Search for things that should be left of the midpoint.

for(int i = lo; i <= hi; i++) {

// Shift values if low.

if(s[i].rank < pivot.rank) {

swap(&s[lo++], &s[i]);

}

}

// Dump pivot into midpoint.

swap(&s[lo], &s[hi]);

return lo;

}

Student \* sortfork(Student \* s, int lo, int hi) {

if(lo < hi) {

int m = parsechunk(s, lo, hi);

memcpy(s, sortfork(s, lo, m - 1), sizeof(Student));

memcpy(s, sortfork(s, m + 1, hi), sizeof(Student));

}

return s;

}

return sortfork(s, 0, s\_len - 1);

}

