Lab 7

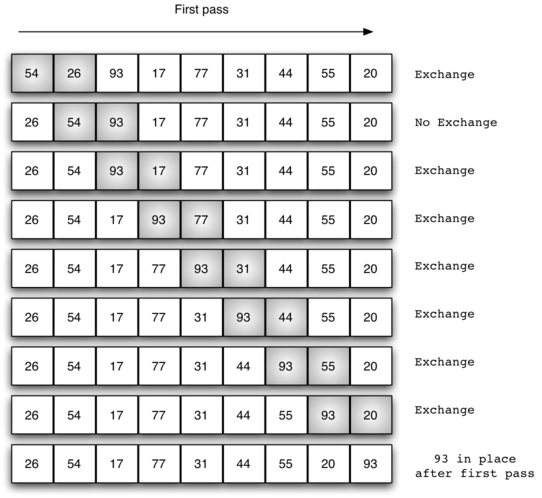
**1. Using an integer array that contains minimum six elements, explain the logic of the following sorting techniques. Use diagrams wherever necessary**

**a. Bubble sort**

Bubble sort iterates through the array of numbers, and looks at each pair of adjacent numbers. Bubble sort will then place the lower number on the left, towards the beginning of the array, and the higher number on the right, towards the end. In other words, it will swap the numbers if ar[i]>ar[i+1](considering, we are sorting in ascending order). At the end of first iteration, the largest number in array would be at the last position. This process is repeated and bubble sort will continue to loop through the array until no swaps are made, thus leaving a sorted array.

It has time complexity of O(n2) and space complexity of O(1).

Below is the example of Bubble Sort:

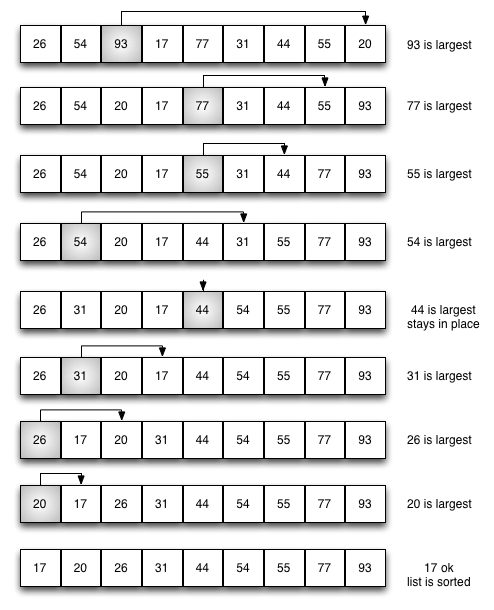


**b. Selection sort**

Here, we assume that the first element is the minimum element. Then, we check if an element lower than that is present in the rest of the array. If there is, we swap the assumed minimum and the actual minimum, else we move on to the next element and find for the next smallest element. In this way, the whole list become sorted.

It also has time complexity of O(n2) and space complexity of O(1).

Below is the example of Selection Sort:

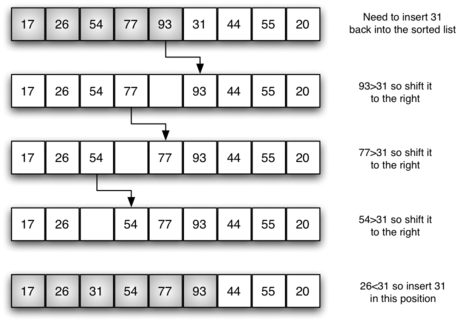


**c. Insertion sort**

In this, for each element, we check if the order is correct until the current element. That is, the left of current element is sorted and in this left sorted array we try to find correct position of our current element and place it there. Since the first element is in order, we start from second element and check if the order is maintained. If not, then we swap them. So, on any given element, we check if the current element is greater than previous element. If it is not, we keep swapping elements until our current element is greater than previous element.

It has worst case time complexity of O(n2), best case time complexity of O(n) and space complexity of O(1).

Below is the example of Insertion Sort:

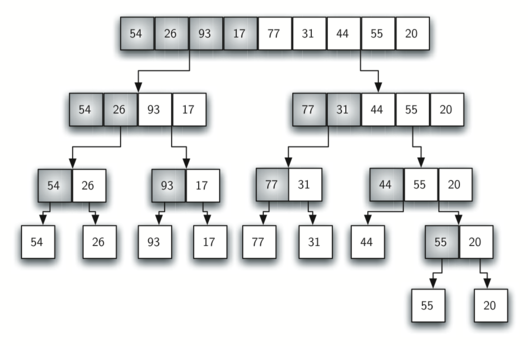


**d. Merge sort**

Merge sort operates by first breaking an array into its individual components. It then ‘pairs up’ an individual with another, putting them into their proper place (sorted) with reference to each other. Merge sort then continues to pair up each sublist of numbers and sort them in the process. This is continued until there is just one list remaining — the sorted array.

It has time complexity of O(nlogn) and space complexity of O(n).

Below is the example of Merge Sort:

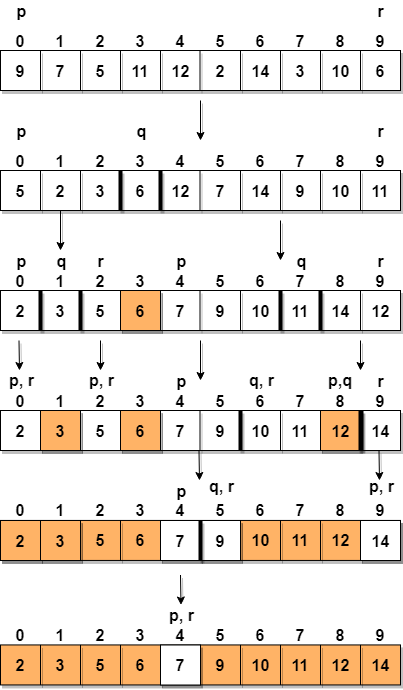


**e. Quick sort**

This algorithm uses the concept of Divide and Conquer. First, we decide a pivot element. We then find the correct index for this pivot position and then divide the array into two subarrays. One subarray will contain elements that are lesser than our pivot and other subarray will contain elements that are greater than our pivot. We then recursively call these two subarrays and the process goes on until we can’t further divide the array.

It has worst case time complexity of O(n2), best case time complexity of O(nlogn) and space complexity of O(n).

Below is the example of Quick Sort:



**2.**

*struct student {*

*char name[10];*

*int rank;*

*};*

**Using the student structure given above, create an array of size 5 students. Then write a complete C program to sort the students array based on the students rank. Use the following sorting techniques in your code.**

**Don’t forget to print initial array and final (sorted) array.**

**a. Bubble sort**

**b. Selection sort**

**Code:**

#include <stdio.h>

struct student {

int rank;

char name[10];

};

// bubble sort function

void sortbubb(struct student class\_record[5], int s)

{

int i, j;

struct student t;

for (i = 0; i < s - 1; i++)

{

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

{

if (class\_record[j].rank > class\_record[j + 1].rank)

{

t = class\_record[j];

class\_record[j] = class\_record[j + 1];

class\_record[j + 1] = t;

}

}

}

}

// selection sort function

void sortsel(struct student class\_record[5], int s)

{

int i, j, min\_idx;

struct student t;

for (i = 0; i < s - 1; i++)

{

min\_idx = i;

for (j = i + 1; j < s; j++)

if (class\_record[j].rank < class\_record[min\_idx].rank)

min\_idx = j;// finding minimum element from array

t = class\_record[i];// swapping minimum element with first element

class\_record[i] = class\_record[min\_idx];

class\_record[min\_idx] = t;

}

}

// Function to displays the contents of the array of structures

void display(struct student class\_record[5])

{

int i, len = 5;

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

printf("Roll number : %d",

class\_record[i].rank);

printf(" Name: %s",

class\_record[i].name);

printf("\n");

}

}

// Driver Code

int main()

{

// Initialize of an array of structures

int choice;

struct student class\_record[5]

= { { 3, "Json" },

{ 2, "Hope" },

{ 1, "Batman" },

{ 4, "Hose" },

{ 5, "William" }

};

display(class\_record);

printf("Which technique to use for sorting: 1. Bubble sort\n2.Selection sort\n");

scanf("%d", &choice);

if (choice == 1)

{

printf("After Bubble Sort\n");

sortbubb(class\_record, 5);

}

if (choice == 2) {

printf("After Selection Sort\n");

sortsel(class\_record, 5);

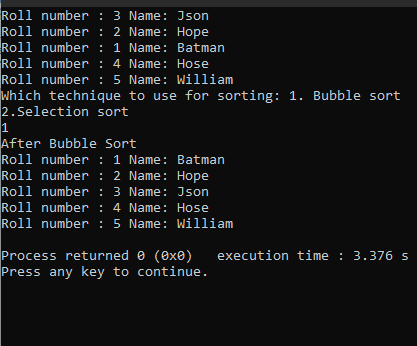
}

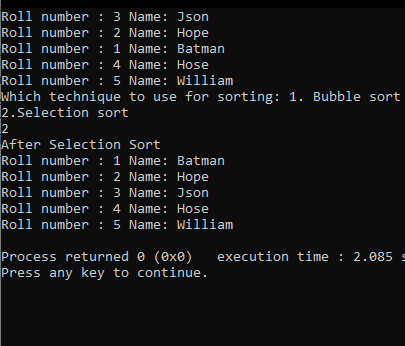
display(class\_record);

return 0;

}

**Output:**





**3.**

*struct grade {*

*char id[10];*

*int mark;*

*};*

**Using the student structure given above, create an array of size 5 students. Then write a complete C program to sort the students array based on the students rank. Use the following sorting techniques in your code.**

**Don’t forget to print initial array and final (sorted) array.**

**a. Insertion sort**

**b. Merge sort**

**c. Quick sort**

**Code:**

#include<stdio.h>

struct grade

{ char id[1];

int marks;

};

// FUNCTION DECLARATIONS

void insertion\_sort(struct grade students[]);

void merge(struct grade students[], int left, int middle, int right); void swap(int\* a, int\* b);

void print(struct grade students[]);

void merge\_sort(struct grade students[],int left,int right); void quick\_sort(struct grade students[], int low, int high); int partition (struct grade students[], int low, int high);

int main()

{

int i,n=0;

struct grade students[5];

for(i=0;i<5;i++)

{

printf("Enter id and marks of student %d:-\n",i+1); scanf("%s",&students[i].id); scanf("%d",&students[i].marks);

}

// UNSORTED DATA

printf("(UNSORTED DATA)\n");

printf("ID\tMarks\n");

for(i=0;i<5;i++)

{

printf("%s\t",students[i].id); printf("%d\n",students[i].marks);

}

printf("\nEnter 1 for insertion sort.\nEnter 2 for merge sort.\nEnter 3 for Quick sort sort.\n"); scanf("%d",&n);

switch(n)

{

case 1:

// SORTING VIA INSERTION SORT

insertion\_sort(students);

printf("\n(SORTING METHOD= INSERTION SORT)\n");

print(students); break;

case 2:

//SORTING VIA MERGE SORT

merge\_sort(students, 0,4);

printf("\n(SORTING METHOD= MERGE SORT)\n");

print(students); break;

case 3:

// SORTING VIA QUICK SORT

merge\_sort(students, 0,4);

printf("\n(SORTING METHOD= QUICK SORT)\n");

print(students);

break;

default:

printf("Enter valid option.\n");

}

}

// INSERTION SORT FUNCTION

void insertion\_sort(struct grade students[])

{

int i, key, j;

for (i = 1; i < 5; i++) { key = students[i].marks; j = i - 1;

while (j >= 0 && students[j].marks > key) { students[j+1].marks = students[j].marks;

j = j - 1;

}

students[j+1].marks = key;

}

}

//FUNCTIONS REQUIRED FOR MERGE SORT->

void merge(struct grade students[], int left, int middle, int right)

{

int i, j, k;

int n1 = middle - left + 1; int n2 = right - middle;

/\* create temp arrays \*/ struct grade L[n1]; struct grade R[n2];

/\* Copy data to temp arrays L[] and R[] \*/ for (i = 0; i < n1; i++)

L[i].marks = students[left + i].marks; for (j = 0; j < n2; j++)

R[j].marks = students[middle + 1 + j].marks;

/\* Merge the temp arrays back into arr[l..r]\*/

i = 0;

j = 0; // Initial index of second subarray

k = left; // Initial index of merged subarray

while (i < n1 && j < n2) {

if (L[i].marks <= R[j].marks) { students[k].marks = L[i].marks; i++;

}

else {

students[k].marks = R[j].marks; j++;

} k++;

}

/\* Copy the remaining elements of L[], if there are any \*/

while (i < n1) { students[k].marks = L[i].marks; i++;

k++;

}

/\* Copy the remaining elements of R[], if there are any \*/

while (j < n2) { students[k].marks = R[j].marks; j++;

k++;

}

}

// MERGE SORT FUNCTION

void merge\_sort(struct grade students[],int left,int right)

{

if (left < right) {

// large l and h

int middle = left + (right - left) / 2;

// Sort first and second halves merge\_sort(students, left, middle); merge\_sort(students, middle + 1, right);

merge(students, left, middle, right);

}

}

// FUNCTIONS REQUIRED FOR QUICK SORT->

void swap(int\* a, int\* b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

/\* This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot)

to left of pivot and all greater elements to right of pivot \*/

int partition (struct grade students[], int low, int high)

{

int pivot = students[high].marks; // pivot

int i = (low - 1);

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

{

// If current element is smaller than the pivot if (students[j].marks < pivot)

{

i++; // increment index of smaller element

swap(&students[i].marks, &students[j].marks);

}

}

swap(&students[i + 1].marks, &students[high].marks); return (i + 1);

}

void quick\_sort(struct grade students[], int low, int high)

{

if (low < high)

{

/\* pi is partitioning index, arr[p] is now at right place \*/

int pi = partition(students, low, high);

// Separately sort elements before

// partition and after partition quick\_sort(students, low, pi - 1); quick\_sort(students, pi + 1, high);

}

}

void print(struct grade students[])

{ int i; printf("\nID\tMarks\n"); for(i=0;i<5;i++)

{

printf("%s\t",students[i].id); printf("%d\n",students[i].marks);

}

}

**Output**

