Sir Syed CASE Institute of Technology

Data Structure and Algorithm

SUBMITTED TO:

Sir. Kaleem

SUBMITTED BY:

Ahtisham Ahmed

FA21-BSCS/014

Ahtisham Ahmed
014-BSCS/Fa21

BUBBLE SORT:

Code:

```
#include <iostream>
using namespace std;

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

        int count = 0;

        while (count < size) {

                for (int i = 0; i < size - count; i++) {
                        //cout << size-count;
                        if (arr[i] > arr[i + 1]) {
                                int temp = arr[i];
                                arr[i] = arr[i + 1];
                                arr[i + 1] = temp;
                        }
                }

                count++;
        }
}

void display(int arr[], int size) {
        for (int i = 0; i <= size; i++) {
                cout << arr[i] << "\t";

        }
}

int main() {
        int arr[6] = { 4,3,1,7,8,0 };
        int size = 5;

        cout << "Before : ";

        display(arr, size);
        cout << endl;

        cout << "After  : ";
        bubble_sort(arr, size);
        display(arr, size);
}
```

Explanation:

BEST CASE	AVERAGE CASE	WORST CASE
O(n)	O(n^2)	O(n^2)

BEST CASE:

In best case O (n) the array which has to be sorted is already sorted.

Ahtisham Ahmed
014-BSCS/Fa21

WORST CASE:

In worst case O (n^2) the array which have to be sorted is not sorted and we will have to use 2 loops in order to sort the array.

SELECTION SORT:

Code:

```
#include <iostream>
using namespace std;

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

	for (int first = 0; first < size - 1; first++) {

		for (int second = first + 1; second < size; second++) {

			if (arr[first] > arr[second]) {

				int temp = arr[first];

				arr[first] = arr[second];

				arr[second] = temp;

			}
		}
	}

}

void display(int arr[], int size) {
	int i = 0;
	while (i < size) {
		cout << arr[i] << "\t";

		i++;
	}
	cout << endl;
}

int main() {

	int arr[5] = { 4,3,1,7,8 };
```

Ahtisham Ahmed
014-BSCS/Fa21

```
        int size = 5;

        cout << "Before : ";
        display(arr, size);

        cout << "\n";

        cout << "After  : ";

        bubble_sort(arr, size);

        display(arr, size);

}
```

Explanation:

BEST CASE	AVERAGE CASE	WORST CASE
$O(n^2)$	$O(n^2)$	$O(n^2)$

BEST CASE:

In best case O (n^2) even if the array is already sorted we will need 2 loops.

WORST CASE:

In worst case O (n^2) the array which have to be sorted is not sorted and we will have to use 2 loops in order to sort the array.

INSERTION SORT:

Code:

```
void insertionSort(int *array, int size) {

  int key, j;

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

    key = array[i];

    j = i;

    while(j > 0 && array[j-1]>key) {

      array[j] = array[j-1];
      j--;

    }

    array[j] = key;

  }

}
```

Explanation:

BEST CASE	AVERAGE CASE	WORST CASE
O(n)	O(n^2)	O(n^2)

BEST CASE:

In best case O (n) even if the array is already sorted we will need 1 loop tochek if the array is sorted.

WORST CASE:

In worst case O (n^2) the array which have to be sorted is not sorted and we will have to use 2 loops in order to sort the array.

QUICK SORT:

Code:

```
#include <iostream>
using namespace std;

int quick_sort(int arr[], int p, int q) {

        int pivot = p;

        int low = p;

        p = p + 1;

        while (p < q) {

                while (arr[p] <= arr[pivot]) {  //to check p
                        p++;
                }
```

Ahtisham Ahmed
014-BSCS/Fa21

```
                while (arr[q] > arr[pivot]) { //to check q
                        q--;
                }


                if (p < q) {

                        int temp = arr[p];

                        arr[p] = arr[q];

                        arr[q] = temp;

                }

        }

        if (p > q) {
                int temp = arr[low];

                arr[low] = arr[q];

                arr[q] = temp;
        }

        return q;
}

void initial_sort(int arr[], int low, int high) {

        int pivot;

        if (low < high) {

                pivot = quick_sort(arr, low, high);

                quick_sort(arr, low, pivot - 1); //solves first half

                quick_sort(arr, pivot+1, high); //solves second half

        }

}

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

        for (int i = 0; i <= size; i++) {
                cout << arr[i] << "\t";
        }
        cout << endl;
}

int main() {
        int arr[8] = { 35,50,15,25,80,20,90,45 };

        int size = 7;

        cout << "Before  : ";
        display(arr, size);

        cout << endl << endl;

        cout << "After  : ";

        initial_sort(arr, 0, size);

        display(arr, size);
}
```

Explanation:

BEST CASE	AVERAGE CASE	WORST CASE
O(n log(n))	O(n log(n))	O(n^2)

BEST CASE:

In best case O(n log(n)) even if the array is already sorted we will need to partition the array and check if the array is sorted through a loop.

WORST CASE:

In worst case O (n^2) the array which have to be sorted is not sorted and we will have to use 2 loops in order to sort the array.

MERGE SORT:

Code:

```
void merge(int *array, int l, int m, int r)

{   int i, j, k, nl, nr;

  //size of left and right sub-arrays

  nl = m-l+1; nr = r-m;

  int larr[nl], rarr[nr];

  //fill left and right sub-arrays

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

    larr[i] = array[l+i];

  for(j = 0; j<nr; j++)
```

Ahtisham Ahmed
014-BSCS/Fa21

```
    rarr[j] = array[m+1+j];
  i = 0; j = 0; k = l;
  //marge temp arrays to real array
  while(i < nl && j<nr) {
    if(larr[i] <= rarr[j]) {
      array[k] = larr[i];
      i++;
    }else{
      array[k] = rarr[j];
      j++;
    }
    k++;
  }
  while(i<nl) {       //extra element in left array
    array[k] = larr[i];
    i++; k++;
  }
  while(j<nr) {     //extra element in right array
    array[k] = rarr[j];
    j++; k++;
  }
}
void mergeSort(int *array, int l, int r) {
  int m;
  if(l < r) {
    int m = l+(r-l)/2;
    // Sort first and second arrays
    mergeSort(array, l, m);
    mergeSort(array, m+1, r);

    merge(array, l, m, r);

  }
```

Explanation:

BEST CASE	AVERAGE CASE	WORST CASE
O(n log(n))	O(n log(n))	O(n log(n))

BEST CASE:

In best case O(n log(n)) even if the array is already sorted we will need to partition the array and check if the array is sorted through a loop.

WORST CASE:

In worst case O(n log(n)) the array which have to be sorted is not sorted and we will have to partition the array and sort using a loop.

CONCULSION

In conclusion the best sorting algorithm according to bigO notation is merge sort. Merge sort best and worst cases are:

BEST CASE	AVERAGE CASE	WORST CASE
O(n log(n))	O(n log(n))	O(n log(n))

The reason merge sort is the ideal sorting algorithm because merging sort uses divide and conquer. It is really fast and the best and worst and average case are the same. The only downside of merge sort is that it take quite a lot of memory. Quick sort is almost the same merge sort but in quick sort the worst case scenario is O (n^2) .if you pick the wrong pivot element it could take a lot of time.

BINARY SEARCH:

Ahtisham Ahmed
014-BSCS/Fa21

```
#include <iostream>
using namespace std;

void binary_search(int arr[], int size, int to_find) {

        int low = 0;
        int high = size;
        int half = (low+high)/2;

        if (arr[half] == to_find) {
                cout << "The number is found at index number :  " << half << endl;
        }

        else {
                while (low!=high+1) {
                        half = (low + high) / 2;

                        if (arr[half] == to_find) {
                                cout << "The number is found at index number :  " << half << endl;
                                break;
                        }

                        else if (arr[half] < to_find) {
                                low++;
                        }

                        else {
                                low--;
                        }


                }
        }

}

void display(int arr[], int size) {
        int i = 0;
        while (i < size) {
                cout << arr[i] << "\t";

                i++;
        }
        cout << endl;
}

int main() {

        int arr[5] = { 1,2,3,4,5 };

        int size = 4;

        int to_find;

        cout << "Array : ";
        display(arr, size+1);

        cout << "\n";
```

```
	cout << "Enter the number you want to find : "; cin >> to_find; cout << endl;

	binary_search(arr, size, to_find);



}
```

SEQUENTIAL SEARCH:

```
#include <iostream>
using namespace std;

void sequential_search(int arr[], int size, int to_find) {

	int i = 0;

	while (i <= size) {
		if (arr[i] == to_find) {
			cout << "The number is present at index number : " << i << endl;
			break;
		}
		else {
			i++;
		}
	}

}

void display(int arr[], int size) {
	int i = 0;
	while (i < size) {
		cout << arr[i] << "\t";

		i++;
	}
	cout << endl;
}

int main() {

	int arr[5] = { 1,2,3,4,5 };

	int size = 4;

	int to_find;

	cout << "Array : ";
	display(arr, size+1);

	cout << "\n";

	cout << "Enter the number you want to find : "; cin >> to_find; cout << endl;
	sequential_search(arr, size, to_find);


}
```