Sorting in C

```
//Array Sorting Program

#include <stdio.h>
int main()
{
int arr[] = {50, 20, 80, 70, 10};
int temp = 0;
int length = sizeof(arr)/sizeof(arr[0]);
printf("Elements of original array: \n");
for (int i = 0; i < length; i++) {
printf("%d ", arr[i]);
}
for (int i = 0; i < length; i++) {
for (int j = i+1; j < length; j++) {
if(arr[i] > arr[j]) {
temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
}
}
}
```

```
printf("\n");
printf("Elements of array sorted in ascending
order: \n");
for (int i = 0; i < length; i++) {
printf("%d ", arr[i]);
}
return 0;
}
```

Bubble Sort

- **Bubble sort** is a very simple sorting technique. However, this sorting algorithm is not efficient in comparison to other sorting algorithms.
- The basic idea underlying the **bubble sort** is to pass through the file sequentially several times. Each pass consists of comparing each element in the file with its successor (**x[i]** with **x[i+1]**) and interchanging the two elements if they are not in proper order.
- Example: Consider the following file,

25 57 48 37 12 92 86 33

Bubble Sort...

In first pass, following comparisons are made:

x[0]	with	x[1]	(25 with 57)	No interchange
x[1]	with	x[2]	(57 with 48)	Interchange
x[2]	with	x[3]	(57 with 37)	Interchange
x[3]	with	x[4]	(57 with 12)	Interchange
x[4]	with	x[5]	(57 with 92)	No interchange
x[5]	with	x[6]	(92 with 86)	Interchange
x[6]	with	x[7]	(92 with 33)	Interchange

Thus, after the first pass, the file is on the order:

25 48 37 12 57 86 33 92

Bubble Sort...

- After first pass, the largest element (in this case 92) gets into its proper position within the array.
- In general, ***x[n-i]*** is in its proper position after iteration ***i***. The method is thus called ***bubble sort*** because each number slowly "***bubbles***" up to its proper position after each iteration.
- Now after the second pass the file is:

  25 37 12 48 57 33 86 92

- Thus, after second pass, 86 has now found its way to the second highest position.

Bubble Sort...

- Since each iteration or pass places a new element into its proper position, a file of *n* elements requires no more than *n-1* iterations.
- The complete set of iterations is the following:

Original file:	25	57	48	37	12	92	86	33
Iteration 1:	25	48	37	12	57	86	33	92
Iteration 2:	25	37	12	48	57	33	86	92
Iteration 3:	25	12	37	48	33	57	86	92
Iteration 4:	12	25	37	33	48	57	86	92
Iteration 5:	12	25	33	37	48	57	86	92
Iteration 6:	12	25	33	37	48	57	86	92
Iteration 7:	12	25	33	37	48	57	86	92

Algorithm for Bubble Sort

- This algorithm sorts the array ***list*** with ***n*** elements:

1. Initialization,
   Set **i=0**
2. Repeat steps 3 to 6 until **i<n**
3. Set **j=0**
4. Repeat step 5 until **j<n-i-1**
5. If **list[j]>list[j+1]**
   Set **temp = list[j]**
   Set **list[j] = list[j+1]**
   Set **list[j+1] = temp**
   j=j+1
   End if
6. **i=i+1**
7. Exit

/* Bubble sort C program */

```
#include <stdio.h>
int main()
{
  int array[100], n, i, j, swap;

  printf("Enter number of elements\n");
  scanf("%d", &n);

  printf("Enter %d integers\n", n);

  for (i = 0; i < n; i++)
    scanf("%d", &array[i]);

  for (i = 0 ; i < n - 1; i++)
  {
    for (j = 0 ; j < n - i - 1; j++)
    {
      if (array[j] > array[j+1]) /* For decreasing order use '<' instead of '>' */
      {
        swap       = array[j];
        array[j]   = array[j+1];
        array[j+1] = swap;
      }
    }
  }

  printf("Sorted list in ascending order:\n");

  for (i = 0; i < n; i++)
    printf("%d\n", array[i]);

  return 0;
}
```

Efficiency of Bubble Sort

- Sorting algorithms are analyzed in terms of the number of comparisons required (i.e. the major operation).
- In ***bubble sort***, the first pass requires (n-1) comparisons to fix the highest element to its location, the second pass requires (n-2) comparisons, ..., k^{th} pass requires (n-k) comparisons and the last pass requires only one comparison to be fixed at its proper position.
- Therefore total number of comparisons:

$$f(n) = (n-1) + (n-2) + \ldots + (n-k) + \ldots + 3 + 2 + 1 = (n-1)*n/2$$

$$< 1*n^2$$

Thus, $\boldsymbol{f(n) = O(n^2)}$ *with* $g(n)=n^2$ *and* $C=1$ *whenever* $n>1$.

- In case of ***bubble sort***,

  Worst case complexity = Best case complexity = Average case complexity = $\boldsymbol{O(n^2)}$ because the comparisons will always take place.

Selection Sort:-

Selection sort is the selection of an element & keepingit in sorted orde us take an array arr[0]……..arr[N-1]. First find the position of smallest eleme from arr[0] to arr [n-1]. Then interchange the smallest element from arr[1] to arr[n-1], then interchanging the smallest element with arr[1]. Similarly, the process will be for arr[0] to arr[n-1] & so on.

Algorithm:-

Pass 1:- search the smallest element for arr[0] ……..arr[N-1].

- Interchange arr[0] with smallest element

Result : arr[0] is sorted.

Pass 2:- search the smallest element from arr[1],…….arr[N-1]

- Interchange arr[1] with smallest element

Result: arr[0], arr[1] is sorted.

……

Pass N-1:-

- search the smallest element from arr[N-2] & arr[N-1]
- Interchange arr[N-1] with smallest element

Result: arr[0]…… Arr[N-1] is sorted.

Q. Show all the passes using selecting sort.

	75	35	42	13	87	27	64	57
Pass 1	(75)	35	42	(13)	87	27	64	57
Pass 2	13	(35)	42	75	87	(27)	64	57
Pass 3	13	27	(42)	75	87	35	64	57
Pass 4	13	27	35	(75)	87	(42)	64	57
Pass 5	13	27	35	42	(87)	75	64	(57)
Pass 6	13	27	35	42	57	(75)	(64)	87
Pass 7	13	27	35	42	57	64	75	87

```
arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

ALGORITHM

Step 1 – Set MIN to location 0
Step 2 – Search the minimum element in the list
Step 3 – Swap with value at location MIN
Step 4 – Increment MIN to point to next element
Step 5 – Repeat until list is sorted

```c
#include <stdio.h>
int main()
{
  int array[100], n, i, j, position, t;

  printf("Enter number of elements\n");
  scanf("%d", &n);

  printf("Enter %d integers\n", n);

  for (i = 0; i < n; i++)
    scanf("%d", &array[i]);

  for (i = 0; i < (n - 1); i++) //
finding minimum element (n-1) times
  {
    position = i;
for (j = i + 1; j < n; j++)
    {
      if (array[position] > array[j])
        position = j;
    }
    if (position != i)
    {
      t = array[i];
      array[i] = array[position];
      array[position] = t;
    }
  }

  printf("Sorted list in ascending
order:\n");

  for (i= 0; i < n; i++)
    printf("%d\n", array[i]);

  return 0;
}
```

Efficiency of Selection Sort

- In the ***selection sort***, the first pass requires (n-1) comparisons to fix the minimum element to its location i.e. location 0, the second pass requires (n-2) comparisons to fix the next minimum element to location 1, ..., k^{th} pass requires (n-k) comparisons to fix the k^{th} minimum element at its location (k-1) and the last pass requires only one comparison to be fixed at the last position of the array.
- Therefore total number of comparisons:

$$f(n) = (n-1) + (n-2) + \ldots + (n-k) + \ldots + 3 + 2 + 1 = n*(n-1)/2$$

Thus, $\boldsymbol{f(n) = O(n^2)}$.

- In case of ***selection sort***,

Worst case complexity = Best case complexity = Average case complexity = $\boldsymbol{O(n^2)}$.

Insertion Sort

- An ***insertion sort*** is one that sorts a set of values by inserting values into an existing sorted file.
- Suppose that ***list[]*** is a list of ***n*** elements in memory. The insertion sort technique scans the list ***list[]*** from ***list[1]*** to ***list[n-1]***, inserting each element ***list[k]*** into its proper position in the previously sorted sublist ***list[0]***, ***list[1]***, ..., ***list[n-1]***.
- ***Illustration:***

  Let ***list[]*** be an array of 7 elements: 25, 15, 30, 9, 99, 20, 26. Initial scenario,

25	15	30	9	99	20	26
0	1	2	3	4	5	6

Pass 1: list[1]<list[0], so interchange the position of elements.

15	25	30	9	99	20	26
0	1	2	3	4	5	6

Pass 2: list[2]>list[1], so position of elements remain same.

15	25	30	9	99	20	26
0	1	2	3	4	5	6

Pass 3: list[3] is less than list[2],list[1] and list[0], thus list[3] is inserted at list[0] position and others are shifted by one position.

9	15	25	30	99	20	26
0	1	2	3	4	5	6

Pass 4: list[4]>list[3], so position of elements remain same.

9	15	25	30	99	20	26
0	1	2	3	4	5	6

Pass 5: list[5] is less than list[4], list[3], and list[2], so list[5] is inserted at the position of list[2] and others are shifted by one position.

9	15	20	25	30	99	26
0	1	2	3	4	5	6

Pass 6: list[6] is less than list[5] and list[4], so list[6] is inserted at the position of list[4] and others are shifted by one position.

9	15	20	25	30	99	26
0	1	2	3	4	5	6

After the pass 6, the list is completely sorted

Algorithm for insertion sort

- This algorithm sorts the array ***list*** with ***n*** elements. Let ***temp*** be a temporary variable to interchange the two values, ***k*** be the total number of passes and ***j*** be another control variable for sorting.

1. Initialization,
   Set **k=1**
2. For **k=1** to **(n-1)**
   Set **temp=a[k]**
   Set **j=k-1**
   While **temp<a[j]** and **(j>=0)** perform the following steps
   Set **a[j+1]=a[j]**
   Set **j=j-1**
   [End of loop structure]
   Assign the value of **temp** to **list[j+1]**
   [End of for loop structure]
3. Exit

```c
/* Insertion sort ascending order */
#include <stdio.h>
int main()
{
  int n, array[1000], c, d, t, flag = 0;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);

  for (c = 1 ; c <= n - 1; c++) {
    t = array[c];
    for (d = c - 1 ; d >= 0; d--) {
      if (array[d] > t) {
        array[d+1] = array[d];
        flag = 1;
      }
      else
        break;
    }
```

```c
 if (flag)
      array[d+1] = t;
  }
  printf("Sorted list in ascending
order:\n");
  for (c = 0; c <= n - 1; c++) {
    printf("%d\n", array[c]);
  }
  return 0;
}
```

Merge Sort

- Merging is the process of combining two or more sorted files into a third sorted file.
- ***Procedure***: In merge sort, we divide the file into ***n*** subfiles of size 1 and then merge adjacent pair of files. We then have ***n/2*** files of size 2. We repeat this process until there is only one file remaining of size ***n***.
- Let the file to sort be:

38 27 43 3 9 82 10

Then the merge sort algorithm works as follows:

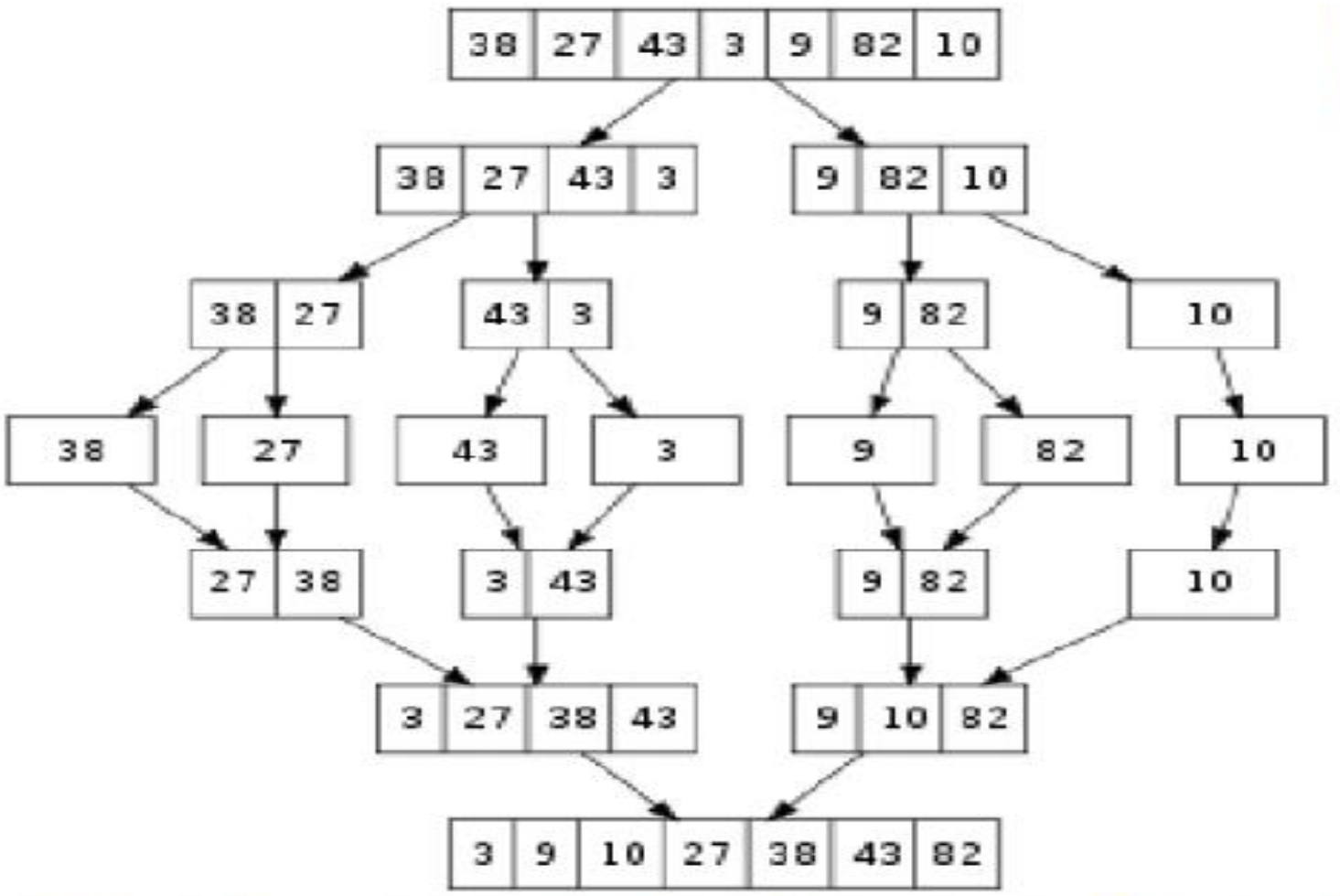

Fig: A recursive merge sort algorithm used to sort an array of 7 integer values. (These are the steps a human would take to simulate merge sort (top-down)).
Reference: http://en.wikipedia.org/wiki/Merge_sort

C Program for Merge Sort

```
#define SIZE 100
void merge_sort(int [], int);
void m_sort(int [], int, int);
void merge(int [], int, int, int);

void main()
{
int list[SIZE], n, i;
clrscr();
printf("\n How many elements in the array:");
scanf("%d", &n);
printf("\n Enter %d values to sort:",n);
for(i=0;i<n;i++)
    scanf("%d", &list[i]);

merge_sort(list, n-1);

printf("\n The sorted list is:");
for(i=0;i<n;i++)
    printf("%d\t", list[i]);
getch();
}
```

```
void merge_sort(int list[], int n)
{
m_sort(list, 0, n);
}

void m_sort(int list[], int left, int right)
{
int mid;
if(right>left)
   {
   mid=(right+left)/2;
   m_sort(list, left, mid);
   m_sort(list, mid+1, right);
   merge(list, left, mid+1, right);
   }
}
```

```
void merge(int list[], int left, int mid, int right)
{
int i;
int temp[SIZE];
int left_end, n, temp_pos;
left_end=mid-1;
temp_pos=left;
n=right-left+1;

while(left<=left_end && mid<=right)
    {
    if(list[left]<=list[mid])
            temp[temp_pos++]=list[left++];
    else
            temp[temp_pos++]=list[mid++];
    }

while(left<=left_end)
    temp[temp_pos++]=list[left++];

while(mid<=right)
    temp[temp_pos++]=list[mid++];

for(i=0;i<n;i++)
    list[right]=temp[right--];
}
```

Efficiency of Merge Sort

- Let us assume that the file size ***n*** is a power of 2, say $\mathbf{n=2^m}$. Thus $\mathbf{m=\log_2 n}$.
- It is therefore obvious that there are no more than **m** or $\mathbf{\log_2 n}$ passes in merge sort (since each pass divides the file into two parts), with each pass involving at most ***n*** comparisons.
- Thus total number of comparisons in merge sort is at most = $\mathbf{n*m = n*\log_2 n}$.
- Hence the time complexity of merge sort = ***O(nlogn)***.
- Average case complexity = Worst case complexity = Best case complexity = ***O(nlogn)***.