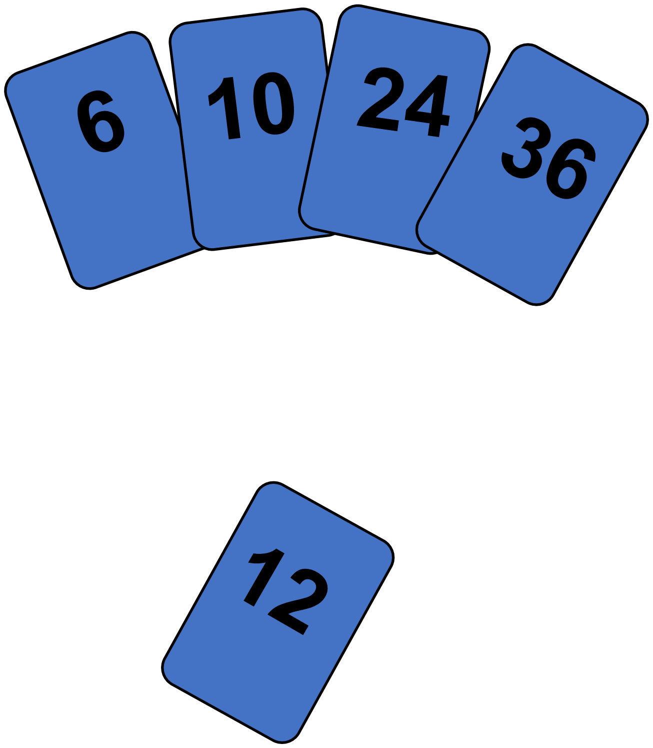
SORTING
(Insertion and Merge)

Insertion Sort

Insertion Sort

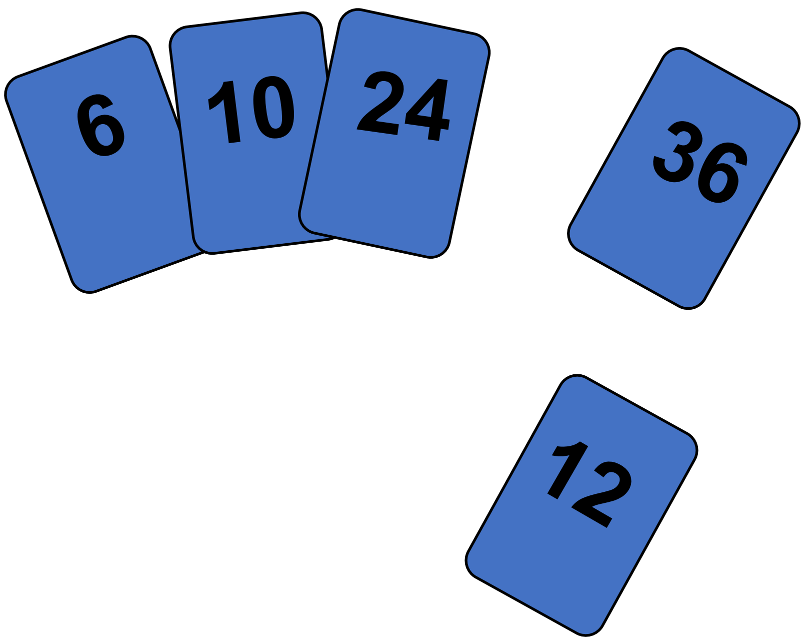
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

Insertion Sort

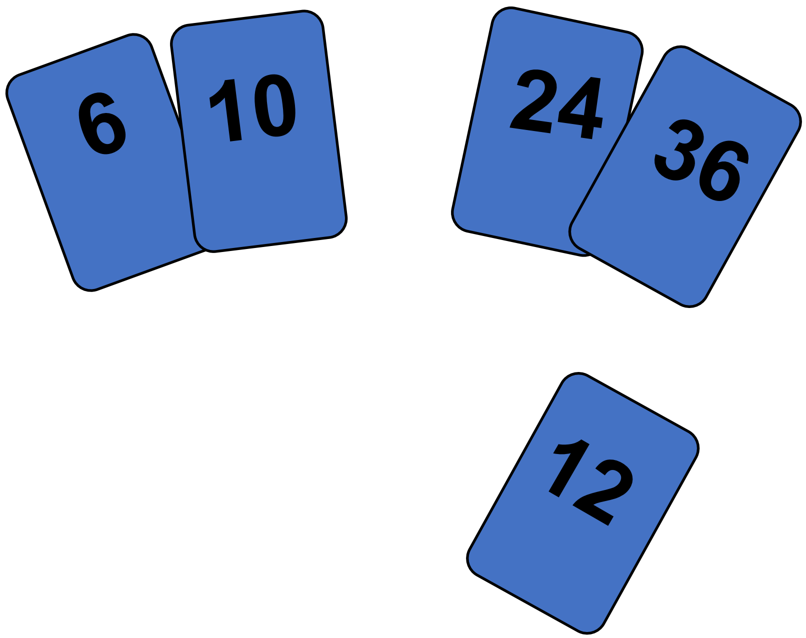


To insert 12, we need to make room for it by moving first 36 and then 24.

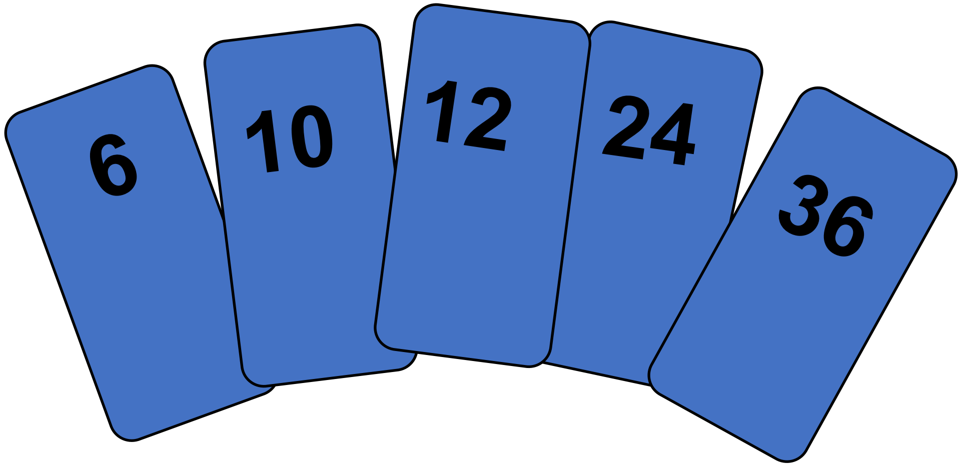
Insertion Sort



Insertion Sort

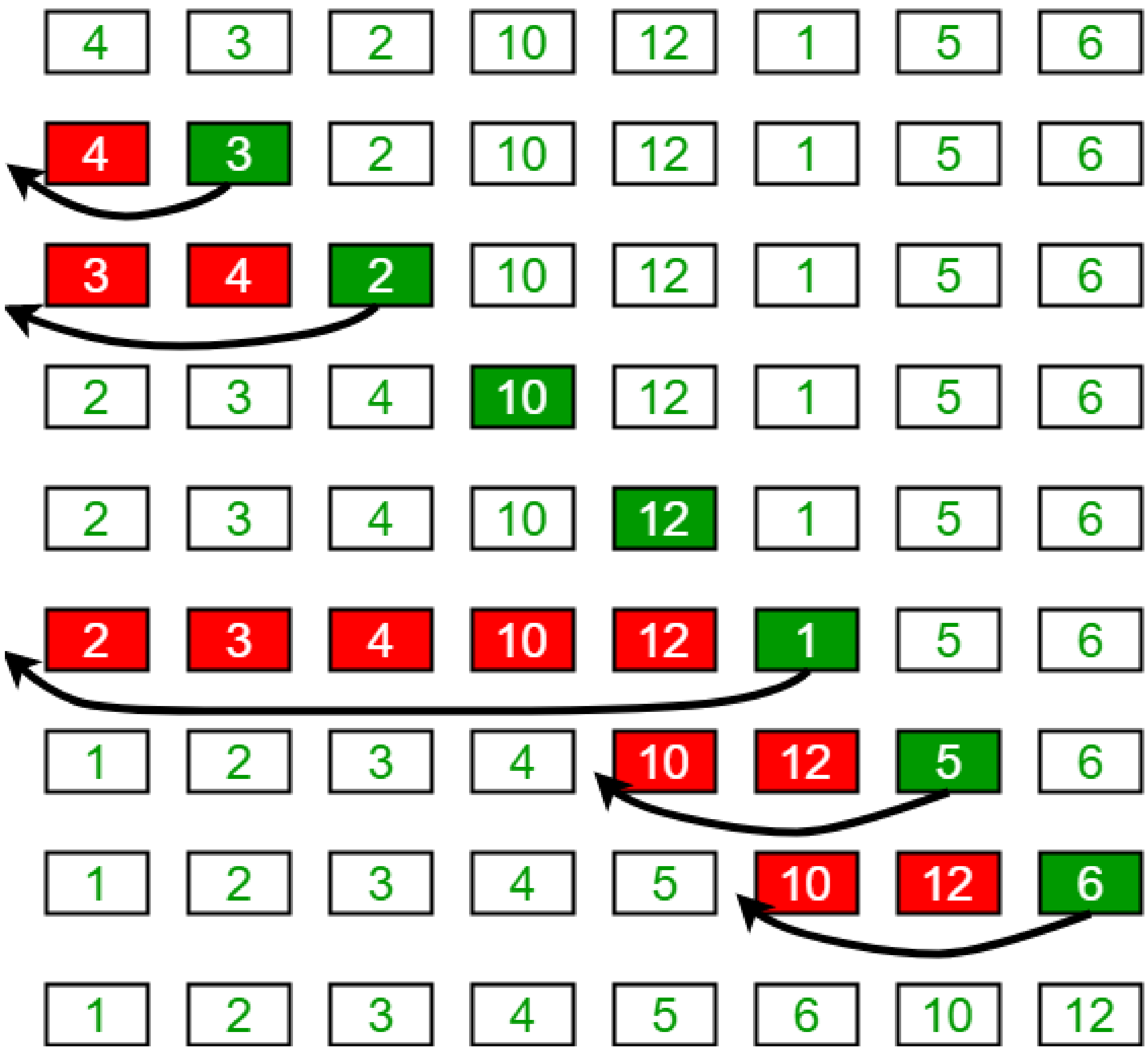


Insertion Sort



Insertion Sort

Insertion Sort Execution Example



Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.
- After i^{th} iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	2.78	7.42	0.56	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 0: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- After i^{th} iteration, `a[0]` through `a[i]` contain first **i+1** elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	2.78	7.42	0.56	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 1: step 0.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	2.78	0.56	7.42	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 2: step 0.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	2.78	7.42	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 2: step 1.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	2.78	7.42	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 2: step 2.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	2.78	1.12	7.42	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 3: step 0.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	2.78	7.42	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 3: step 1.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	2.78	7.42	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 3: step 2.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	2.78	1.17	7.42	0.32	6.21	4.42	3.14	7.71

Iteration 4: step 0.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	1.17	2.78	7.42	0.32	6.21	4.42	3.14	7.71

Iteration 4: step 1.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	1.17	2.78	7.42	0.32	6.21	4.42	3.14	7.71

Iteration 4: step 2.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	1.17	2.78	0.32	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 0.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	1.17	0.32	2.78	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 1.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	0.32	1.17	2.78	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 2.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	0.32	1.12	1.17	2.78	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 3.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 4.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 5.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	6.21	7.42	4.42	3.14	7.71

Iteration 6: step 0.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	6.21	7.42	4.42	3.14	7.71

Iteration 6: step 1.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	6.21	4.42	7.42	3.14	7.71

Iteration 7: step 0.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	4.42	6.21	7.42	3.14	7.71

Iteration 7: step 1.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	4.42	6.21	7.42	3.14	7.71

Iteration 7: step 2.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	4.42	6.21	3.14	7.42	7.71

Iteration 8: step 0.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	4.42	3.14	6.21	7.42	7.71

Iteration 8: step 1.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Iteration 8: step 2.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Iteration 8: step 3.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Iteration 9: step 0.

Insertion Sort

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Iteration 10: DONE.

Insertion Sort

input array

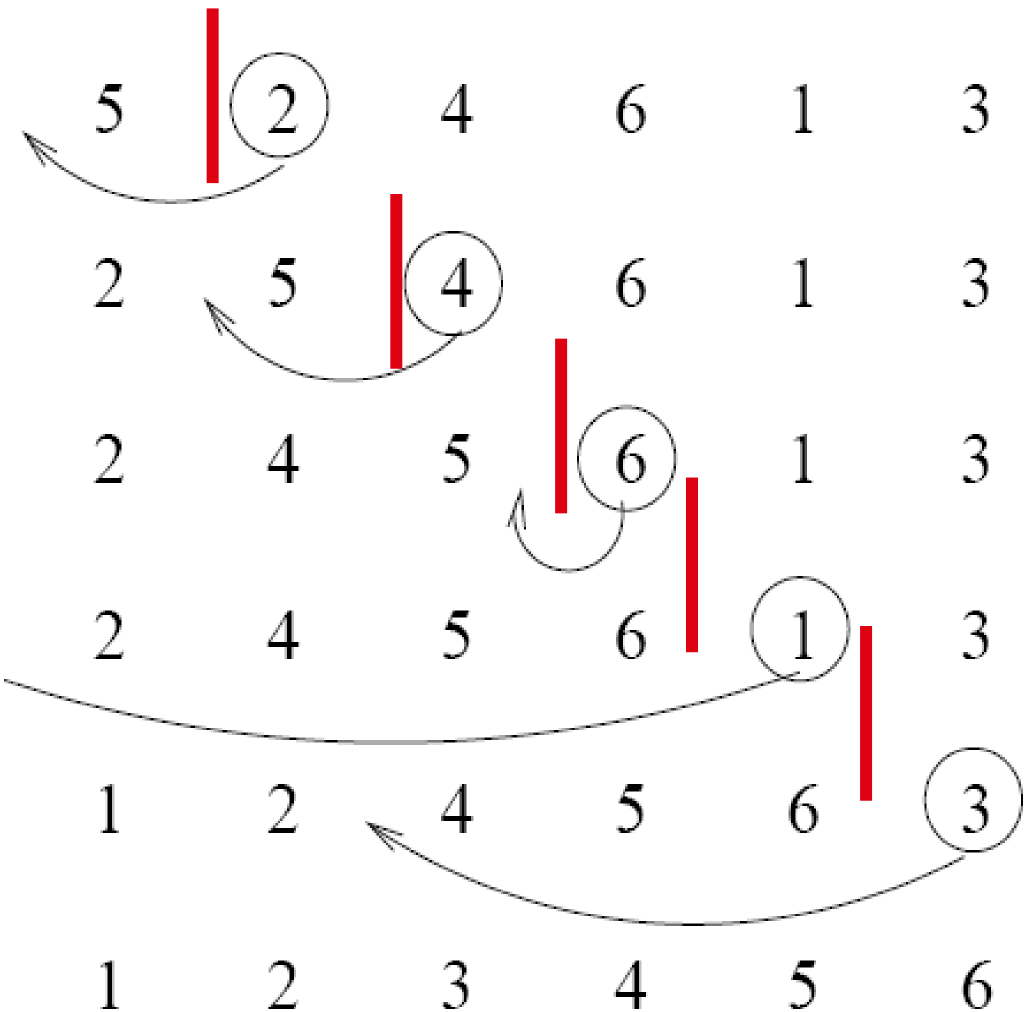
5 2 4 6 1 3

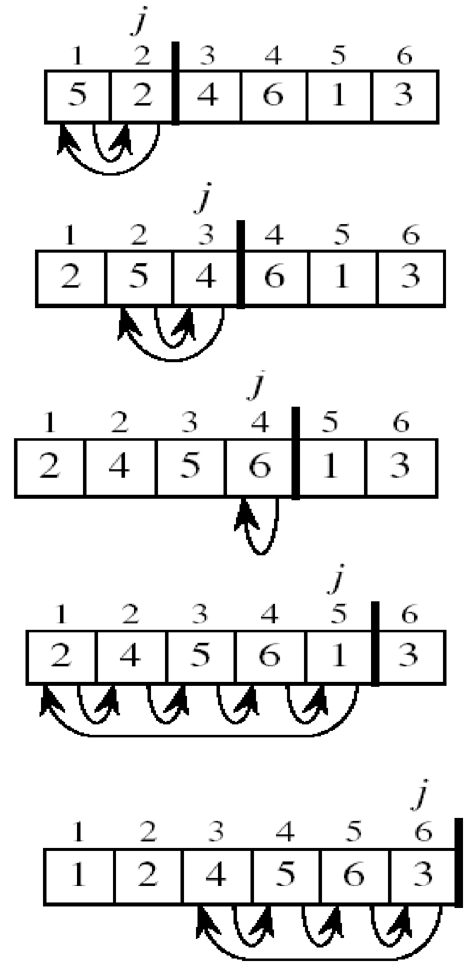
at each iteration, the array is divided in two sub-arrays:

left sub-array right sub-array

sorted unsorted

Insertion Sort





INSERTION SORT ALGORITHM

1	2	3	4	5	6	7	8
a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8

Sorts an array ‘A’ with ‘n’ elements using the INSERTION sort.

Step1: Repeat steps 2 to 4 for **i=2 to n**

Step2: Set **Temp = A[i]** and **k = i-1**

Step3: Repeat While **Temp<A[k] AND k>0**

Set **A[k+1] = A[k]**

Set **k = k-1**

[End loop]

Step4: Set **A[k+1] = Temp**

[End loop]

Step5: Exit

Is it right code for Insertion Sort?

```
for (i = 1 ; i < n ; i++)
 {    key = A [ i ];
      j = i - 1;
     while(j >= 0 && A [ j ] > key)
      {   A [ j+1 ] = A [ j ];
           j--;
       }
      A [ j+1 ] = key;
}
```

```c
#include <stdio.h>
main()
{ int i,n=12,j,key, A[12] = { 97,556,43,-5,6,404,55,3,22,-122,4,35};
for( i=0;i<n;i++)    printf(" %d ",A[i]);
for (i = 1 ; i < n ; i++)
{  key = A [ i ];
   j = i - 1;
   while(j >= 0 && A [ j ] > key)
   {   A [ j+1 ] = A [ j ];
       j--;
    }
    A [ j+1 ] = key;
}
```

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

Analysis of Insertion Sort

- **Which is the WORST CASE for Insertion Sort ?**

  WORST CASE for Insertion Sort is when elements of the list are in the reverse order of the desired one.

- **Desired order is Ascending**

Input List is : 896, 467, 235, 198, 89,75, 64, 15, 6

Analysis of Insertion Sort

- **Which is the BEST CASE for Insertion Sort ?**
    - **BEST CASE for Insertion Sort is when elements of the list are already sorted in desired order.**
- **Desired order is Descending**
- **Input List is : 896, 467, 235, 198, 89,75, 64, 15, 6**

Analysis of Insertion Sort

- The worst case for Insertion sorting is, when elements of the list are in the reverse order of the desired one.
- In the worst case, each element of the unsorted part will be compared with all the elements of the sorted part of the list. Thus, in worst case,
- To position the 2^{nd} element of the list at proper position, number of comparisons = 1
- To position the 3^{rd} element of the list at proper position, number of comparisons = 2
- To position the 4^{th} element of the list at proper position, number of comparisons = 3

Analysis of Insertion Sort

- :
- :
- To position the n^{th} element of the list at proper position, number of comparisons = n-1
- So, the total number. of comparisons in the worst case.

$$f(n) = 1+2+3+4+5+6+.....+(n-2)+(n-1)$$
$$= n*(n-1)/2$$
$$= O(n^2)$$

- Thus, for the worst case the complexity of the insertion sort is $O(n^2)$.
- The best case for this algorithm is when elements are already sorted. Then the complexity will be $O(n)$.
- However, for the average case complexity will be $O(n^2)$.

MERGE SORT

MERGE SORT

- Merge sort is one of the best sorting techniques which is based on the divide and conquer strategy.
- It is an in-place sorting algorithm; It uses no auxiliary data structures (extra space) while sorting.
- Partition the n > 1 elements into two smaller instances.
- First (n/2) elements define one of the smaller instances; remaining (n/2) elements define the second smaller instance.
- Merge Sort reiterates on both the newly formed partitions by dividing them in half and continue the process.

MERGE SORT

- This process of dividing stop when the partition size reaches to one item.
- At this point it has created many one-item lists. Any one-item list is naturally in sorted order.
- The next step is to merge these one-item lists together for creating the sorted list.
- To combine two sorted lists, the merge sort compare successive pairs of elements, one from each list.
- If the list A has any element < than all the elements of list B, then it will be chosen to be append to the aggregated list or vice versa.

MERGE SORT

- And when all the elements of one list are added to the aggregated list then all the remaining elements of other list will be append directly to the aggregated list.
- Merge Sort repeats the process of combining sorted sub-lists into larger sorted sub-lists until all have been successfully integrated into a single sorted list.
- This merging process takes at least (n/2) comparisons but not more than (n-1) comparisons.
- Complexity is O(n log n).

Merge Two Sorted Lists

A = (2, 5, 6)

B = (1, 3, 8, 9, 10)

C = ( )

Compare smallest elements of A and B and merge smaller into C.

A = (2, 5, 6)

B = (3, 8, 9, 10)

C = (1)

Merge Two Sorted Lists

A = (5, 6)

B = (3, 8, 9, 10)

C = (1, 2)

A = (5, 6)

B = (8, 9, 10)

C = (1, 2, 3)

A = (6)

B = (8, 9, 10)

C = (1, 2, 3, 5)

Merge Two Sorted Lists

A = ( )

B = (8, 9, 10)

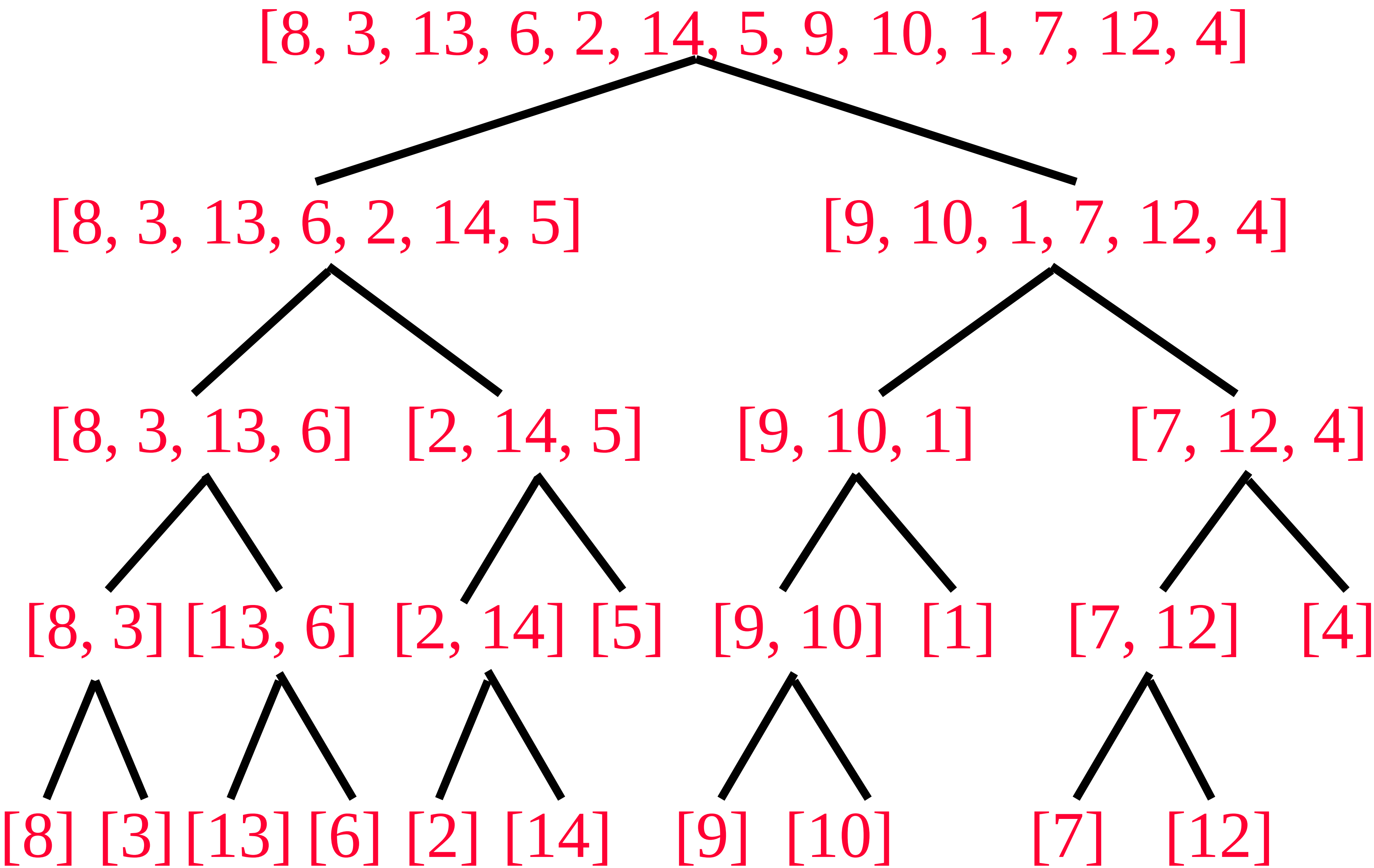
C = (1, 2, 3, 5, 6)

When one of A and B becomes empty, append the other list to C.

O(1) time needed to move an element into C.

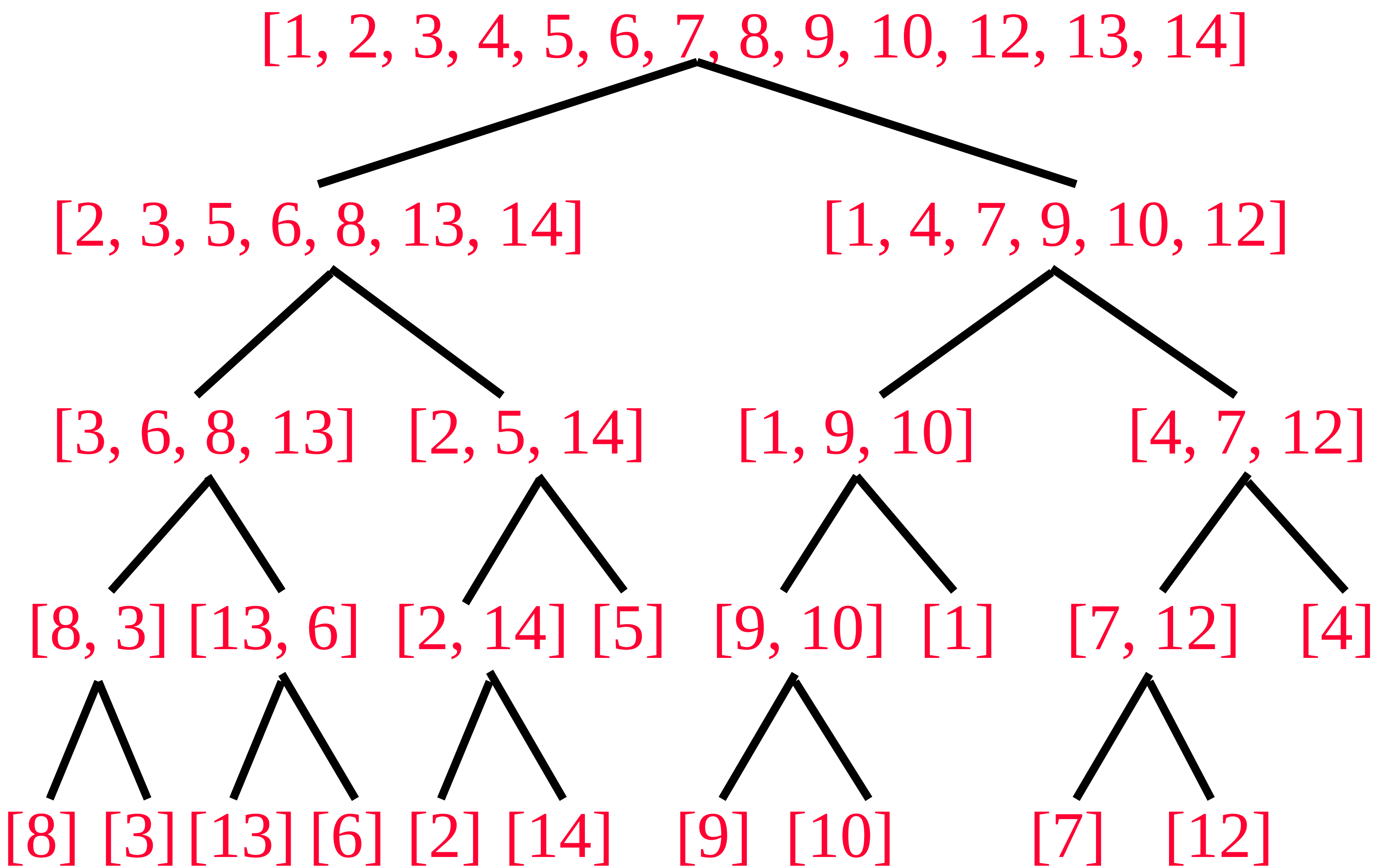
Total time is O(n + m), where n and m are, respectively, the number of elements initially in A and B.

Merge Sort



[8, 3, 13, 6, 2, 14, 5, 9, 10, 1, 7, 12, 4]
[8, 3, 13, 6, 2, 14, 5]
[9, 10, 1, 7, 12, 4]
[8, 3, 13, 6]
[2, 14, 5]
[9, 10, 1]
[7, 12, 4]
[8, 3]
[13, 6]
[2, 14]
[5]
[9, 10]
[1]
[7, 12]
[4]
[8]
[3]
[13]
[6]
[2]
[14]
[9]
[10]
[7]
[12]

Merge Sort


[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14]
[2, 3, 5, 6, 8, 13, 14]
[1, 4, 7, 9, 10, 12]
[3, 6, 8, 13]
[2, 5, 14]
[1, 9, 10]
[4, 7, 12]
[8, 3]
[13, 6]
[2, 14]
[5]
[9, 10]
[1]
[7, 12]
[4]
[8]
[3]
[13]
[6]
[2]
[14]
[9]
[10]
[7]
[12]

Time Complexity

- Let t(n) be the time required to sort n elements.
- t(0) = t(1) = c, where c is a constant.
- When n > 1,
  - t(n) = t(ceil(n/2)) + t(floor(n/2)) + dn,
  - where d is a constant.
- To solve the recurrence, assume n is a power of 2 and use repeated substitution.
- t(n) = O(n log n).

Merge Sort

- Downward pass over the recursion tree.
  - Divide large instances into small ones.
- Upward pass over the recursion tree.
  - Merge pairs of sorted lists.
- Number of leaf nodes is n.
- Number of nonleaf nodes is n-1.

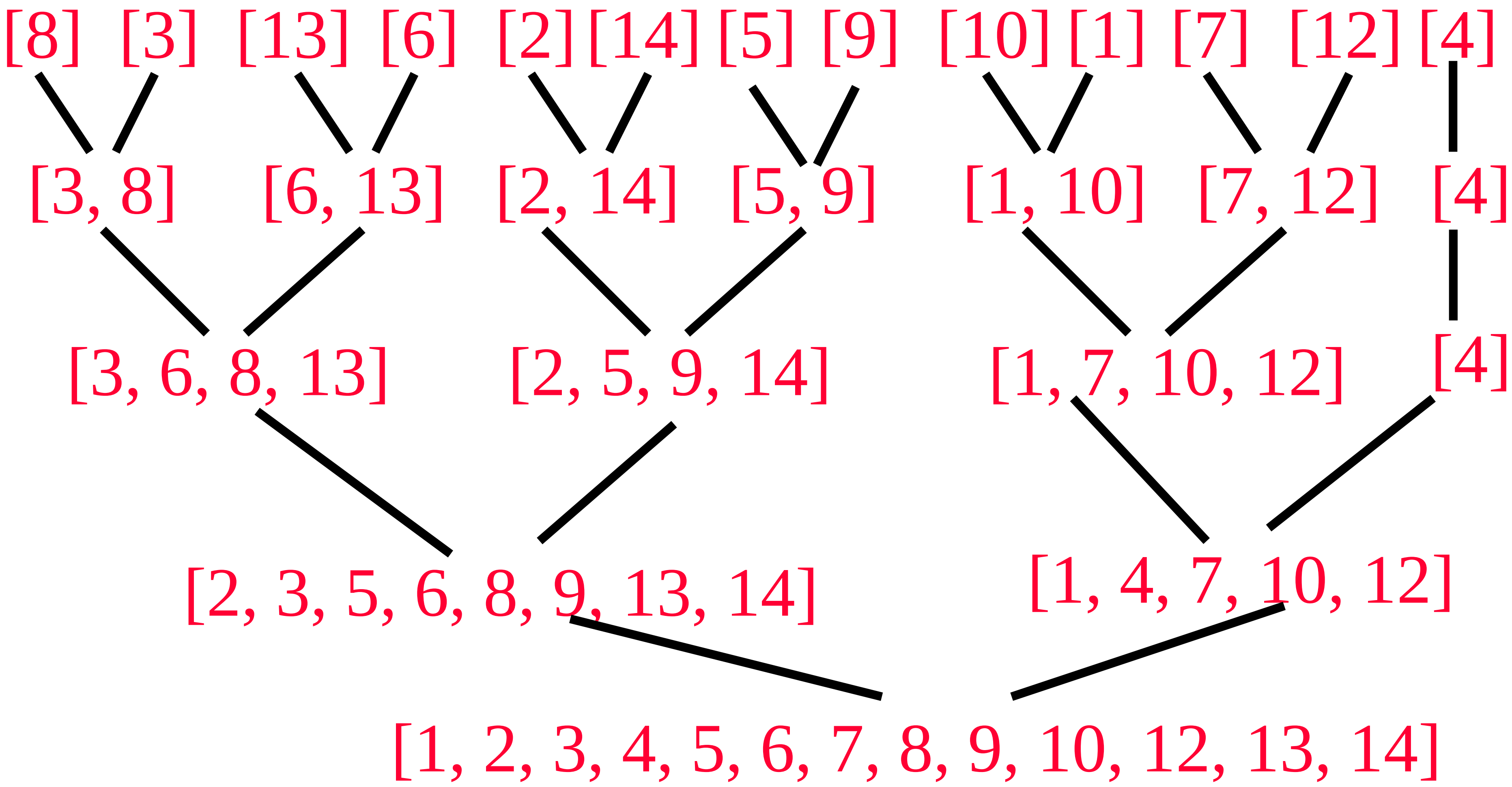
Time Complexity

- Downward pass.
  - O(1) time at each node.
  - O(n) total time at all nodes.
- Upward pass.
  - O(n) time merging at each level that has a nonleaf node.
  - Number of levels is O(log n).
  - Total time is O(n log n).

Nonrecursive Version

- Eliminate downward pass.
- Start with sorted lists of size 1 and do pairwise merging of these sorted lists as in the upward pass.

Nonrecursive Merge Sort



Complexity

- Sorted segment size is 1, 2, 4, 8, …
- Number of merge passes is ceil($\log_2 n$).
- Each merge pass takes O(n) time.
- Total time is O(n log n).
- Need O(n) additional space for the merge.
- Merge sort is slower than insertion sort when n <= 15 (approximately). So define a small instance to be an instance with n <= 15.
- Sort small instances using insertion sort.
- Start with segment size = 15.

Natural Merge Sort

- Initial sorted segments are the naturally occurring sorted segments in the input.
- Input = [8, 9, 10, 2, 5, 7, 9, 11, 13, 15, 6, 12, 14].
- Initial segments are:
  - [8, 9, 10] [2, 5, 7, 9, 11, 13, 15] [6, 12, 14]
- 2 (instead of 4) merge passes suffice.
- Segment boundaries have a[i] > a[i+1].

Merge Sort C Program

```
// Merge Sort C Program
#include<stdlib.h>
#include<stdio.h>
int main()
{ //int arr[] = {85, 24, 63, 45, 17, 31, 96, 50};
 int arr[] = {-5,406,3,425,317,31,6,50,55,54,66,78,99,43,72};
 int arr_size = sizeof(arr)/sizeof(arr[0]);
  printf("\n Given array is : ");
  printArray(arr, arr_size);
  mergeSort(arr, 0, arr_size - 1);
  printf("\n Sorted array is : ");
  printArray(arr, arr_size);
}
```

Merge Sort C Program

```
void printArray(int A[], int size)
{ int i;
  for (i=0; i < size; i++)
       printf("%d ", A[i]);
  printf("\n");
}
void mergeSort(int arr[], int l, int r)
{ if (l < r)
  { int m = l+(r-l)/2;
    mergeSort(arr, l, m);
    mergeSort(arr, m+1, r);
    merge(arr, l, m, r);
   }
}
```

Merge Sort C Program

```
// Merge Function
void merge(int arr[], int l, int m, int r)
{ int i, j, k;
  int n1 = m - l + 1;
  int n2 = r - m;
  int L[n1], R[n2];  // Create temp arrays
  for (i = 0; i < n1; i++)
     L[i] = arr[l + i];     // Copy data to temp array
  for (j = 0; j < n2; j++)
     R[j] = arr[m + 1 + j];
```

Merge Sort C Program

```
// Merge temp arrays
 i = 0;
 j = 0;
 k = l;
 while (i < n1 && j < n2)
 { if (L[i] <= R[j])
   { arr[k] = L[i];
     i++;
    }
   else
   { arr[k] = R[j];
     j++;
   }
   k++;
 }
```

Merge Sort C Program

```
while (i < n1) // Copy the remaining elements of L[]
 { arr[k] = L[i];
   i++;
   k++;
  }
 while (j < n2)// Copy the remaining elements of R[]
 { arr[k] = R[j];
   j++;
   k++;
  }
}
```

Merge Sort C Program

Given array is : 85 24 63 45 17 31 96 50

Sorted array is : 17 24 31 45 50 63 85 96

Given array is : -5 406 3 425 317 31 6 50 55 54 66 78 99 43 72

Sorted array is : -5 3 6 31 43 50 54 55 66 72 78 99 317 406 425

Merge Sort C Program

Given array is : -55 -44 -33 0 3 6 10 15 24 26 37 49 53 66 70 72

Sorted array is : -55 -44 -33 0 3 6 10 15 24 26 37 49 53 66 70 72

Given array is : 555 333 244 100 93 86 60 55 44 36 27 25 23 16 7 2

Sorted array is : 2 7 16 23 25 27 36 44 55 60 86 93 100 244 333 555