jgromero@inf.uc3m.es

Universidad
Carlos III de Madrid
www.uc3m.es

Lesson 7

Search, sort and merge algorithms

Programming

Grade in Industrial Technology Engineering

Outline

1. **Search**
2. **Sort**
3. **Merge**

Search, sort and merge algorithms

Introduction

Algorithms to:

- *Search* a value in a list
- *Sort* a list of values
- *Merge* two lists of values

Lists are represented as **one-dimension arrays**

Search

- Ordered list
- Unordered list

Sort

- Bubblesort
- Insertionsort
- Selectionsort

Merge

Outline

1. **Search**
2. Sort
3. Merge

Search algorithms aim at finding a value in a collection (usually, the first occurrence)

Input: Array of values *list*, length *n*, value to find *e*

Output: Position of *e* in *list*; -1 if not found

Find a value in an array of integers

```
int find(int list[], int n, int e)
```

Input:

```
list ← {5, 6, 3, 1, 8, 9, 0, 3, 4, 1}
n ← 10
e ← 1
```

Output:

```
3
```

The algorithm looks sequentially for the value *e* in the list:

```
location = -1;
i = 0;
found = false;

while ( (!found) && (i < n))

    if (list[i] == e)
        location = i;
        found = true;
    else
        i++;

return location;
```

Linear search (also named sequential search) is used for unsorted lists

1. Search
Linear search

```c
int main(void) {

   int L[]  = {5, 6, 3, 1, 8, 9, 0, 3, 4, 1};
   int size = 10;
   int pos;
   int number;

   printf("Enter value: ");
   scanf("%i", &number);

   pos = linearSearch(L, size, number);
   printf("Position: %i", pos);

   return 0;
}
```

```c
int linearSearch(int list[], int n, int e) {
    int location = -1;
    int i = 0;
    int found = 0;

    while( !found && i<n ) {
       if(list[i] == e) {
           location = i;
           found = 1;
       } else {
           i++;
       }
    }

    return location;
}
```

Ordered lists

Optimizations can be applied to speed up the searching procedure when the list is ordered

(Similar to a dictionary)

Optimized linear search

The value is searched until:

- It is found
- The loop index is beyond the position in which it should be stored

Binary search

The central element of the list is checked

- If it is the searched element, the algorithm ends
- If not, the search is repeated in the corresponding half of the list

Optimized linear search

```
location = -1;
i = 0;
found = false;

while ( (!found) && (i < n) && (list[i] <= e) )

    if (list[i] == e)
        location = i;
        found = true;
    else
        i++;

return location;
```

1. Search

Optimized linear search

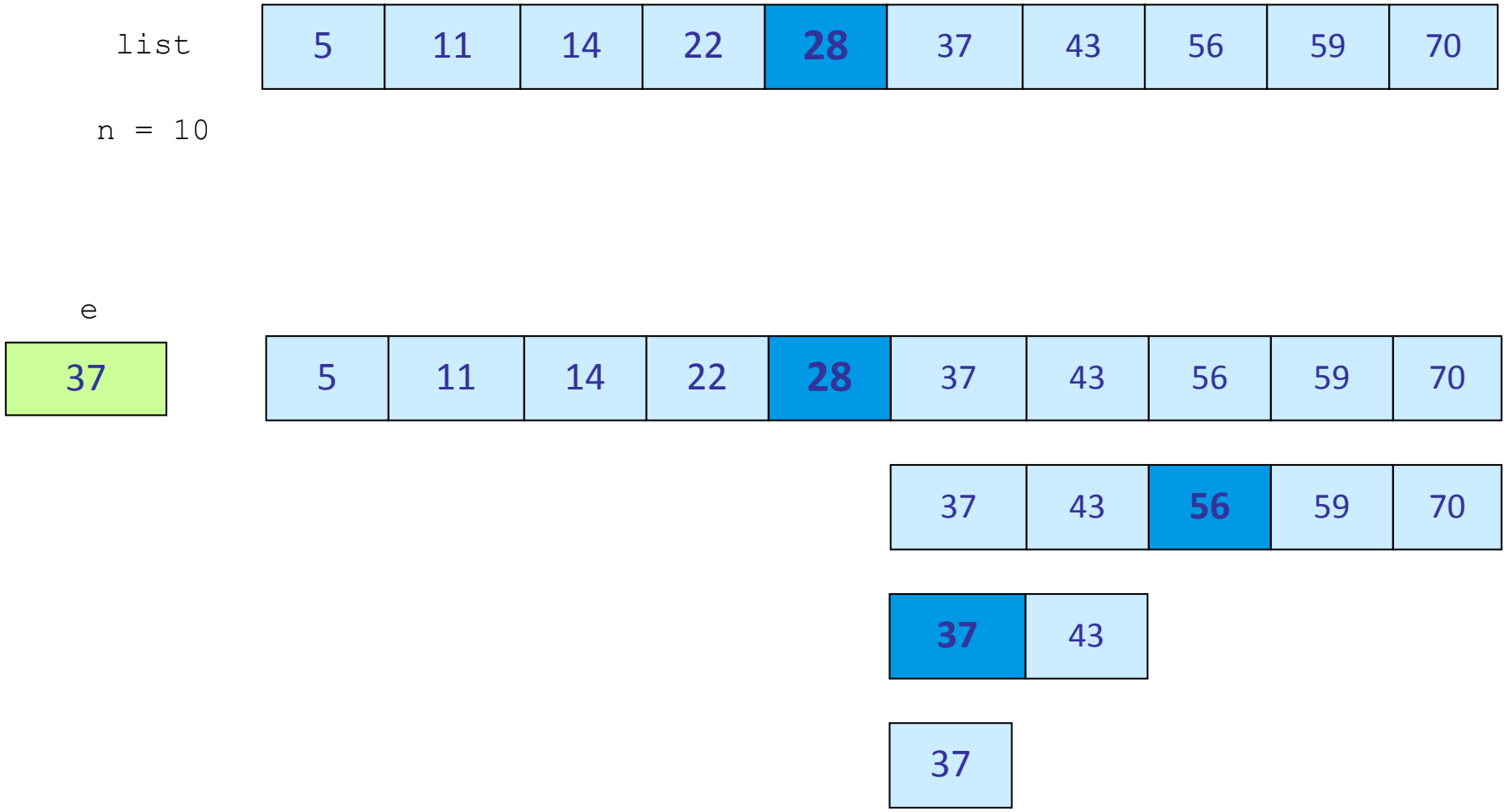
```c
int main(void) {

   int L[] = {0, 1, 1, 3, 3, 4, 5, 6, 8, 9};
   int size = 10;
   int number;
   int pos;

   printf("Enter value: ");
   scanf("%i", &number);

   pos = optimizedLinearSearch(L, size, number);
   printf("Position: %i", pos);

   return 0;
}
```

```c
int optimizedLinearSearch(int list[], int n, int e) {
   int location = -1;
    int i = 0;
    int found = 0;

    while( !found && i<n && list[i]<=e) {
        if(list[i] == e) {
            location = i;
            found = 1;
        } else {
            i++;
        }
    }

    return location;
}
```

```
location = -1;
left   = 0;
right  = n - 1;
middle = (left+right) / 2;
found  = false;

while ( (left <= right) and (!found) )

    if ( list[middle] == e )
        found = true;
        location = middle;

    else
        if (e < list[middle] )
            right = middle – 1;
        else
            left = middle + 1;

        middle = (left+right) / 2;

return location;
```

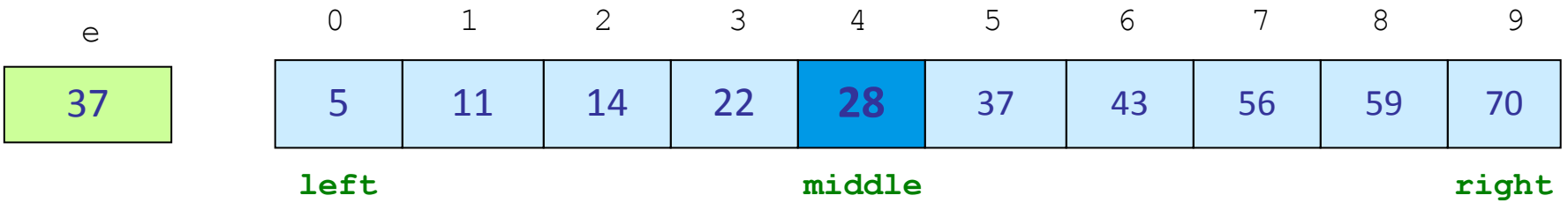
1. Search

Binary search



1. Search

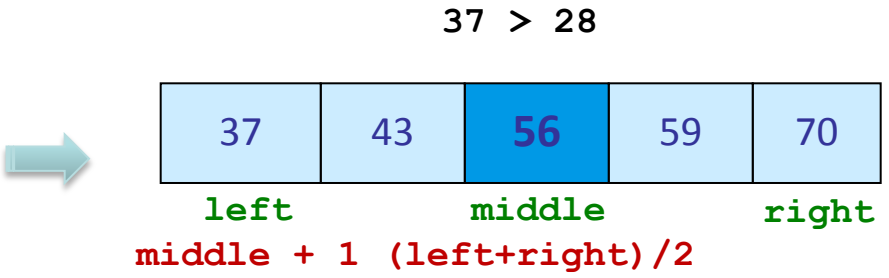
Binary search



Iteration 1

Value not found

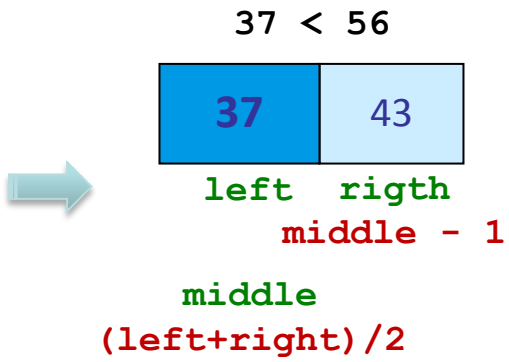
Continue with the second half of the array



Iteration 2

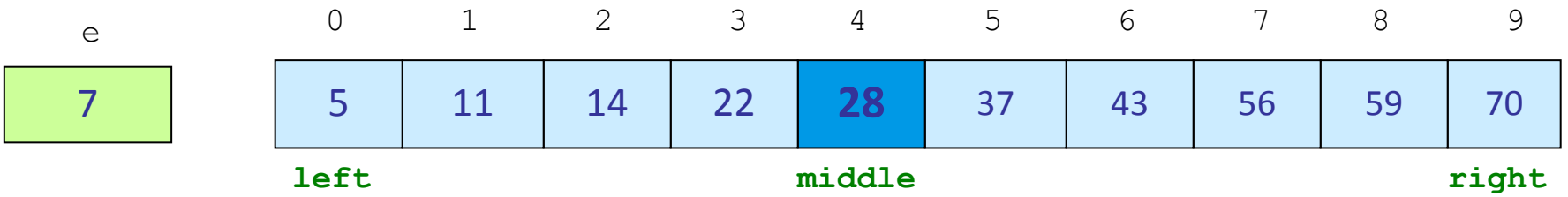
Value not found

Continue with the first half of the array

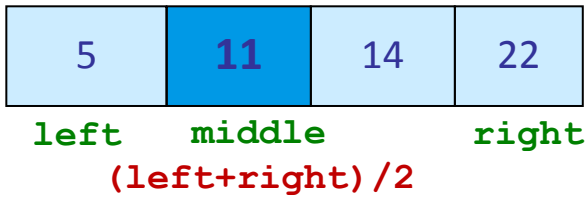


Iteration 3

Value found



7 < 28



Iteration 1
Value not found
Continue with the first half of the array

7 < 11

left
right
middle - 1
middle
(left+right)/2

Iteration 2
Value not found
Continue with the first half of the array

Iteration 3
Value not found
Cannot continue

1. Search
Binary search

```c
int main(void) {

   int L[] = {5, 11, 14, 22, 28, 37, 43, 56, 59, 70};
   int size = 10;
   int number;
   int pos;

   printf("Enter value: ");
   scanf("%i", &number);

   pos = binarySearch(L, size, number);
   printf("Position: %i", pos);

   return 0;
}
```

```c
int binarySearch(int list[], int n, int e) {
   int location = -1;
   int left = 0;
   int right = n-1;
   int middle = (left+right) / 2;
   int found = 0;

   while( left <= right && !found) {
      if(list[middle] == e) {
         found = 1;
         location = middle;
      } else {
         if(e < list[middle])
            right = middle-1;
         else
            left = middle + 1;

         middle = (left + right) / 2;
      }
   }

   return location;
}
```

Outline

1. Search
2. **Sort**
3. Merge

Sort algorithms aim at rearranging the values of a collection to position them in order (usually, in increasing order)

Input: Array of values *list*, list size *n*

Output: Array of values *list** ordered

```
void sort(int list[], int n)
```

Input:

```
list ← {7, 2, 8, 5, 4}
n ← 5
```

Output:

```
list ← {2, 4, 5, 7, 8}
```

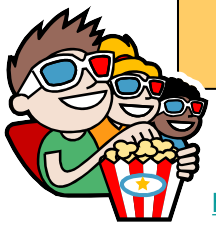
Idea:

- Compare an element *list[i]* with the adjacent value *list[i+1]*
- If *list[i]* > *list[i+1]*, the values are swapped (increasing order)
- Repeat the procedure while swaps are performed

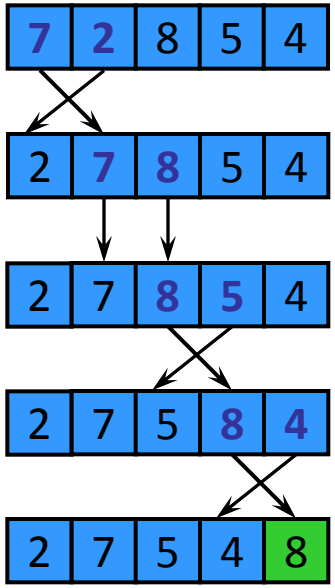
```
do {
    swapped = false;

    for (i=0; i <= n-2; i++)
        if (list[i] > list[i+1])
            swap(list[i], list[i+1]);
            swapped = true;

} while (swapped);
```

To change the order, change the comparison operator

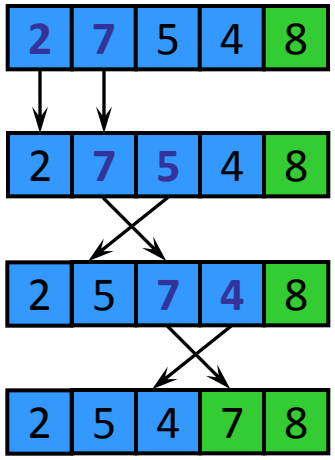
The same algorithm can be applied to other data types



http://www.youtube.com/watch?v=UnK5ueUgc88

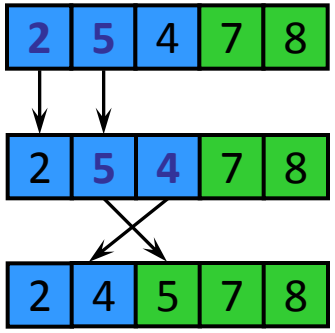


1st iteration
The largest value is pushed towards the the end of the array

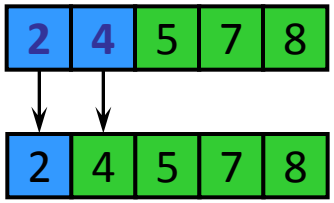


2nd iteration
The second largest value is pushed towards the end of the array

It is not necessary to compare with the last element of the array



3rd iteration
The three largest values are at the end of the array



END

4th iteration
No swapping

The swap variable is set to false and the procedure ends

At most, n-1 iterations are required

At least, n-i comparisons required in each iteration

2. Sort
Bubblesort

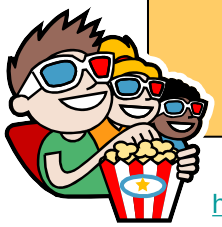
```c
int main(void) {

   int L[] = {7, 2, 8, 5, 4};
   int size = 5;
   int k;

   bubblesort(L, size);
   for(k=0; k<size; k++)
       printf("%i ", L[k]);

   return 0;
}
```

```c
void bubblesort(int list[], int n) {
    int swapped;

    do {
        int i;

        swapped = 0;
        for(i=0; i <= n-2; i++) {
            if(list[i] > list[i+1]) {
                int temp;
                temp = list[i];
                list[i] = list[i+1];
                list[i+1] = temp;

                swapped = 1;
            }
        }

    } while (swapped);

    return;
}
```

Idea:

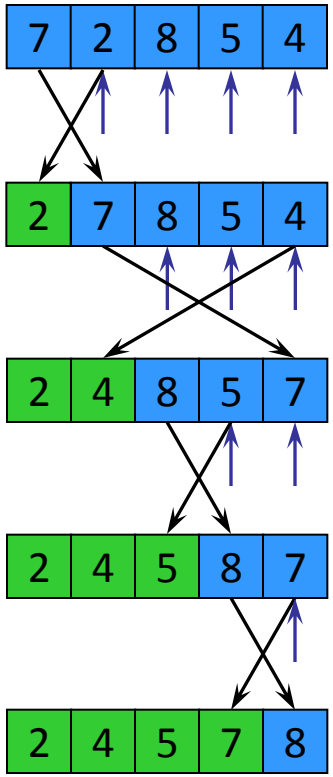
- For each value of the list (at position i),
  - Find the smallest value (at position *minPos*) of the elements *i+1,…, n-1*
  - If *list[i]* > *list[minPos]*, the values are interchanged

```
for (i=0; i <= n-2; i++)
    minPos = i;
    for (j=i+1; j < n; j++)

        if (list[j] < list[minPos])
            minPos = j;

    swap(list[i], list[minPos])
```



http://www.youtube.com/watch?v=TW3_7cD9L1A

j = 1 … n-1



First iteration
First position is 0
i=0, minPos=1

j = 2 … n-1

Second iteration
i=1, minPos=4

j = 3 … n-1

Third iteration
i=2, minPos=3

j = 4 … n-1

Forth iteration
i=3, minPos=4

END

n-1 iterations of the outer loop are required

```c
int main(void) {

   int L[] = {7, 2, 8, 5, 4};
   int size = 5;
   int k;

   selectionsort(L, size);
   for(k=0; k<size; k++)
       printf("%i ", L[k]);

   return 0;
}
```

```c
void selectionsort(int list[], int n) {
    int i, j, k;

    for(i=0; i<=n-2; i++) {
        int temp;

        int minPos = i;
        for(j=i+1; j<n; j++)
            if(list[j] < list[minPos])
                minPos = j;


        temp = list[i];
        list[i] = list[minPos];
        list[minPos] = temp;
    }

    return;
}
```

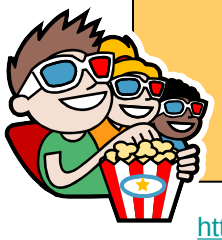
Idea:

- Assume that the elements *0, …, i-1* of the list are ordered

- Find the position *k* in *0, …, i-1* where the element at position *i* should be placed

- (Simultaneously) Shift to the right the values at *k, …, i-1* and insert *list[i]* at position *k*

```
for (i=1; i < n; i++)
    e = list[i];
    j = i-1;
    while( (j >= 0) && (list[j] > e) )
        list[j+1] = list[j];
        j = j-1;

    list[j+1] = e;
```



http://www.youtube.com/watch?v=Fr0SmtN0IJM

2. Sort

Insertionsort

```c
int main(void) {

   int L[] = {7, 2, 8, 5, 4};
   int size = 5;
   int k;

   insertionsort(L, size);
   for(k=0; k<size; k++)
       printf("%i ", L[k]);

   return 0;
}
```

```c
void insertionsort(int list[], int n) {
   int i, j, k;

   for(i=1; i<n; i++) {
       int e;

       e = list[i];
       j = i-1;

       while( j>=0 && list[j] > e) {
           list[j+1] = list[j];
           j = j-1;
       }

       list[j+1] = e;
   }

   return;
}
```

How long does it take to sort an array of *n* elements?

Algorithm performance is measured according to the number of comparison and swapping operations that are required to obtain the solution

This number strongly depends on the starting situation: sorted array, unsorted, reversed, etc.

Some algorithms *realize* that the list is ordered and perform better

In general, basic sorting algorithms (bubble, selection, insertion) are not suitable for large arrays (the differences for small arrays are not significant)

There are more efficient (and complex) sorting algorithms: shell, heap, merge, quicksort, etc.

Algorithms can be compared according to the number of operations performed in the best case, worst case, and average case

Being *n* the length of the array, the upper *complexity* of the algorithm is bounded by:

Algorithm	Best ≈	Worst ≈	Average ≈
Bubble	n	n^2	n^2
Selection	n^2	n^2	n^2
Insertion	n	n^2	n^2
Quicksort	$n \cdot \log(n)$	n^2	$n \cdot \log(n)$

Yes, we can… but not with *bubblesort.*
http://www.youtube.com/watch?v=k4RRi_ntQc8

Bubble sort is the simplest, but also has a the higher worst-case execution time. Nevertheless, it behaves very well with ordered arrays

Selection sort is easy to implement and more efficient that *Bubble sort*, but it behaves very bad even if the array is ordered (it cannot be known if the array is already ordered at any iteration)

Insertion sort is simple to implement and behaves quite well for almost ordered arrays. It is also faster in practice

Outline

1. Search
2. Sort
3. **Merge**

Merging consists in combining sorted sequences of values into a single sorted sequence

Input: Sorted array of values *list1* and *list2*, list size *n1, n2*

Output: New sorted array of values *list* with size *n1+n2*

```
void merge(int list1[], int n1, int list2[], int n2,
           int list[])
```

Input:

```
list1 ← {2, 4, 5, 7, 8}
n1 ← 5
list2 ← {1, 3, 8}
n2 ← 3
```

Output:

```
list ← {1, 2, 3, 4, 5, 7, 8, 8}
```

Some advanced sorting algorithms use merging

- The list is divided in smaller pieces to be ordered and, finally, the parts are merged
  - E.g.: Mergesort

- 'Divide & Conquer' strategy: a problem can be solved by splitting it into parts, solving the parts, and joining the partial solutions

Merging is also convenient if the number of values to sort is larger than the memory size

Idea:

- Define two indices `i,j` to traverse all the elements of `list1,list2` respectively
- Determine the value to add to the merged array by comparing `list1[i]` and `list2[j]`
- Copy the remaining elements of the list that has not been completely traversed to `list`

3. Merge
Algorithm

```
i = 0; j = 0; k = 0;

while(i < n1 && j < n2)
    if(list1[i] < list2[j]
        list[k] = list1[i]
        i = i+1
    else
        list[k] = list2[j]
        j = j+1
    k = k+1

if(i < n1)
    for(l=i; l < n1; l++)
        list[k] = list1[l]
        k = k+1
else
    for(l=j; l < n2; l++)
        list[k] = list2[l]
        k = k+1
```

3. Merge
Implementation

```
int main(void) {
   int L1[] = {2, 4, 5, 7, 8};
   int size1 = 5;
   int L2[] = {1, 3, 8};
   int size2 = 3;

   int L[8];
   int size=size1+size2;

   int k;

   merge(L1, size1, L2, size2, L);
   for(k=0; k<size; k++)
       printf("%i ", L[k]);

   return 0;
}
```

```
void merge(int list1[], int n1, int list2[], int n2, int list[]) {

   int i, j, k, l;

   i=0;
   j=0;
   k=0;

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

       if(list1[i] < list2[j]) {
           list[k] = list1[i];
           i++;
       } else {
           list[k] = list2[j];
           j++;
       }

       k++;
   }

   if(i<n1)
       for(l=i; l<n1; l++) {
           list[k] = list1[l];
           k++;
       }
   else
       for(l=j; l<n2; l++) {
           list[k] = list2[l];
           k++;
       }

   return;
}
```

Basic

- Paul J. Deitel, Harvey M. Deitel. C: *How to Program*. Prentice Hall, 2006 (5th Edition) – Chapter 16 (sections 1, 3, 4)