MODULE 5

SORTING

DATA STRUCTURE AND ALGORITHMS

FACULTY OF COMPUTING
UNIVERSITI TEKNOLOGI MALAYSIA

MODULE 5: SORTING

OBJECTIVES FOR STUDENTS

1. To describe the purpose of sorting technique as operations on data structure.
2. To write source codes for the implementation of simple sort algorithms : Bubble Sort, Insertion Sort and Selection Sort
3. To write source codes for the implementation of divide and conquer sorting algorithms : Merge Sort and Quick Sort.
4. To identify the efficiency of the sorting algorithms and determine the suitable sorting techniques for certain problem.

KEY CONCEPT

1.0 INTRODUCTION TO SORTING

1.1. **Sorting definition -** A process in which data in a list are organized in certain order; either ascending or descending order.

1.2. **Advantages** of sorted lists:
   i. Easier to understand and analyze data collection.
   ii. Searching process will be much faster.
   - Sorting Example :
      i. Sorted in Ascending order: phone directory and dictionary
      ii. Sorted in Descending order; number of scores/points earned by every team in a competition. The winner gets the highest score.

1.3. Sorting Algorithms **Categories**:
   i. An internal sort
      - Requires that the collection of data fit entirely in the computer's main memory. Suitable to sort a small size of list.
   ii. An external sort
      - The collection of data will not fit in the computer's main memory all at once, but must reside in secondary storage. Suitable to sort large

size of data.

1.4. Types of lists to be sorted:

- List of simple data types, such as integers, char or strings
- Examples: list of numbers (**int** type) or list of book titles (**string** type) as shown in the figure below:

List of student's age (int)
22
28
18
23
19

List of Book Titles (string)
Struktur Data
Learning English
Mathematics for Kids
Effective Communication
Learn C++

1.5. Sorting list of records

- Each record contains a field called the **key.**
- Record key – field that become the identifier to the record.
- For sorting purposes, the records will be sorted based on the sorting key, which is part of the data item that we consider when sorting a data collection.
- Example: A list that contains student's information

Sorting key
The list can be sorted either by student's name, matrix number or CPA.

Indeks	Student Name	Matrix Number	CPA
[0]	Hisham	A5021	3.09
[1]	Zainal	A1051	2.55
[2]	Maria	A2000	3.60
[3]	Adam	A5501	3.00
[4]	Zahid	A2233	2.95

List of records

2.0 SORTING PROCESS

2.1. Two main **activities** in the sorting process:

i. **Compare:** compare between two elements. If they are not in correct

order, then
ii. **Swap:** Change the position of the elements in order to get the right order.

2.2. The **efficiency** of sorting algorithm is measured based on :
- the number of comparisons and
- the number of swapping between elements

2.3. The sorting efficiency is measured based on the execution time of the algorithm when tested using sample **cases** of data as follows:
- **Worst-case analysis** considers the maximum amount of work an algorithm will require on a problem of a given size. (Data is totaly unsorted).
- **Best-case analysis** considers the minimum amount of work an algorithm will require on a problem of a given size. (Data is almost sorted).
- **Average-case analysis** considers the expected amount of work that an algorithm will require on a problem of a given size.

3.0 SORTING ALGORITHMS

3.1. There are several sorting algorithms. In this module, two **strategies** of sorting techniques will be discussed in detail. They are:
i. Quadratic Sorting Algorithms
ii. Divide and Conquer Sorting Algorithms

3.2. **Quadratic** Sorting Algorithms work straight-forward and sorting methods is usually people think of sorting things in general. The quadratic sorting algorithms are not very fast and with quadratic efficiency.

3.3. Three quadratic sorting algorithms are:
i. Bubble Sort
ii. Insertion Sort
iii. Selection Sort

3.4. Divide and Conquer Sorting Algorithms strategy solves a problem by :
i. Breaking into sub problems that are themselves smaller instances of the same type of problem.
ii. Recursively solving these sub problems.
iii. Appropriately combining their answers.

3.5. Two types of sorting algorithms which are based on this divide and conquer algorithm :
i. Merge Sort
ii. Quick Sort

4.0 BUBBLE SORT

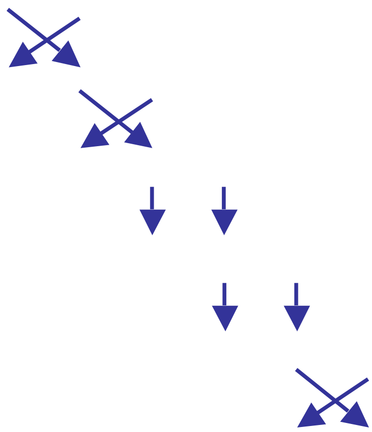
4.1. Bubble sort is a simple sorting technique in which will arrange the elements of the list by comparing each pair of adjacent items and swapping them if they are in the wrong order.

4.2. To sort an array of record with bubble sort, it works by taking multiple passes over the array with the following main activities
- i. **Compare** adjacent elements in the list
- ii. **Exchange** the elements if they are out of order
- iii. Each pass **moves the largest** (or smallest) elements to the end of the array
- iv. **Repeating** this process eventually sorts the array into ascending (or descending) order.

4.3. Bubble sort is a quadratic algorithm *O(n^2)*. The algorithm only suitable to sort array with small size of data.

4.4. Example of Bubble Sort operation with list of 8 elements.

[0] [1] [2] [3] [4] [5] **Pass 1: Unsorted List**



1. Compare, swap (0, 1)
2. Compare, swap (1, 2)
3. Compare, no swap
4. Compare, no swap
5. Compare, swap (4, 5)
6. 99 is in right position

[0]	[1]	[2]	[3]	[4]	[5]
8	3	12	21	1	99
3	8	12	21	1	99
3	8	12	21	1	99
3	8	12	21	1	99
3	8	12	1	21	99
3	8	12	1	21	99

Pass 2
1. Compare, swap (0, 1)
2. Compare, no swap
3. Compare, no swap
4. Compare, swap (3, 4)
5. 22 is in right position

[0]	[1]	[2]	[3]	[4]	[5]
3	8	12	1	21	99
3	8	12	1	21	99
3	8	12	1	21	99
3	8	1	12	21	99

Pass 3
1. Compare, no swap
2. Compare, no swap
3. Compare, swap (2, 3)
4. 12 is in right position

3	8	1	12	21	99
3	8	1	12	21	99
3	1	8	12	21	99

Pass 4
1. Compare, no swap
2. Compare, swap (1, 2)
3. 8 is in right position

3	1	8	12	21	99
1	3	8	12	21	99
1	3	8	12	21	99

Pass 5

4. Compare, swap (0, 1)
5. 1 & 3 are in right position - DONE

4.5. Bubble Sort implementation:

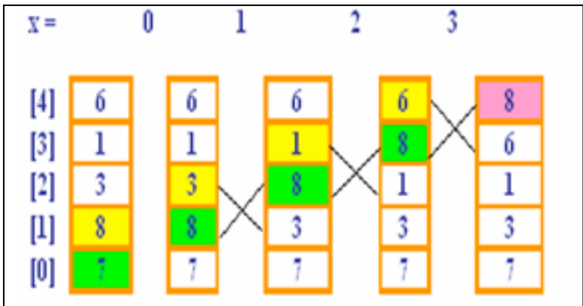
If statement is used to compare the adjacent elements.

External for loop is used to control the number of passes needed.

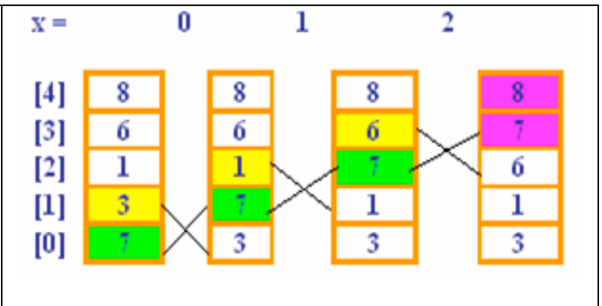
```
//Program 5.1
// Sorts items in an array into ascending order.
void BubbleSort(dataType data[], int listSize)
{
  int pass, tempValue;
    for ( pass =1;pass < listSize; pass++ )
    {
     // moves the largest element to the
     // end of the array
       for (int x = 0; x < listSize - pass; x++)
          //compare adjacent elements
          if ( data[x]>data[x+1] )
          {   // swap elements
              tempValue = data[x];
              data[x] = data[x+1];
              data[x+1] = tempValue;
          }
       }
} // end Bubble Sort
```

Internal for loop is used to compare adjacent elements and swap elements if they are not in order. After the internal loop has finished execution, the largest element in the array will be moved at the top.

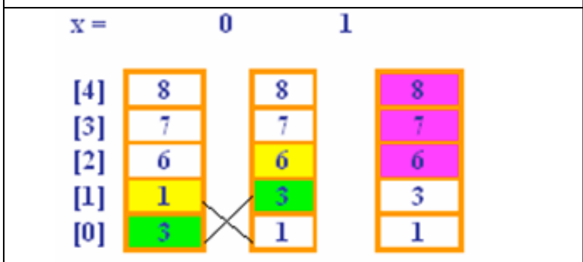
4.6. Example of Bubble Sort implementation to sort array of integer [7 8 3 1 6] into ascending order:

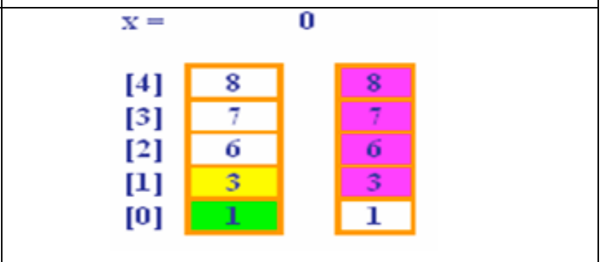


pass = 1 **listSize** = 5



pass = 2 **listSize** = 5





pass = 3 **listSize** = 5	**pass** = 3 **listSize** = 5

4.7. Bubble sort analysis:

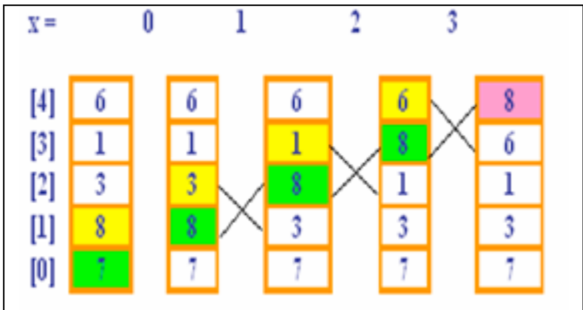
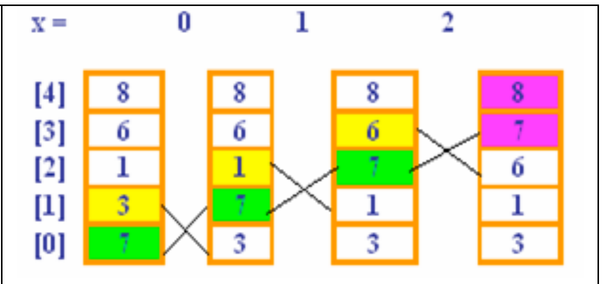
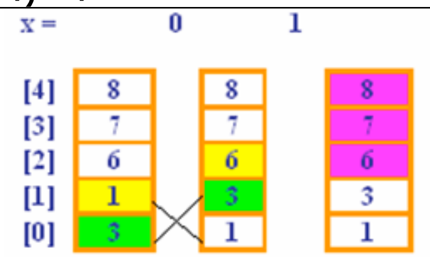
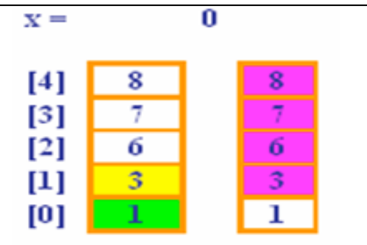
- To determine the efficiency of Bubble Sort algorithm the following number need to be identified:
    - o the number of comparison between elements and
    - o the number of exchange between elements.

- Generally, the number of comparisons between elements in Bubble Sort can be stated as follows:

$$(n\text{-}1)+(n\text{-}2)+\ldots\ldots.+2+1= n(n\text{-}1)/2 = O(n^2)$$

- However, in any cases, (worst case, best case or average case) the number of comparisons between elements are the same.

4.8. An example of Bubble sort analysis for array [7 8 3 1 6]:

	
Pass 1 : **Comparison (listSize-pass):** **(5-1) = 4**	**Pass 2 :** **Comparisons (listSize-pass):** **(5-2) = 3**
	
Pass 3 : **Comparison(listSize-pass):** **(5-3) = 2**	**Pass 4 :** **Comparisons (listSize-pass):** **(5-4) = 1**

- The number of comparisons:

  $(n\text{-}1)+(n\text{-}2)+\ldots\ldots\ldots\ldots.+2+1= n(n\text{-}1)/2 = O(n^2)$
- The number of comparisons for array [7 8 3 1 6]:

  (5-1) + (5-2) + (5-3) + (5-4) = 4 + 3 + 2 + 1 = 10.

4.9. In any cases, (worst case, best case or average case) to sort the list in ascending order the number of comparisons between elements are the **same**.

i. Worst Case [8 7 6 3 1]
ii. Average Case [7 8 3 1 6]
iii. Best Case [1 3 6 7 8]

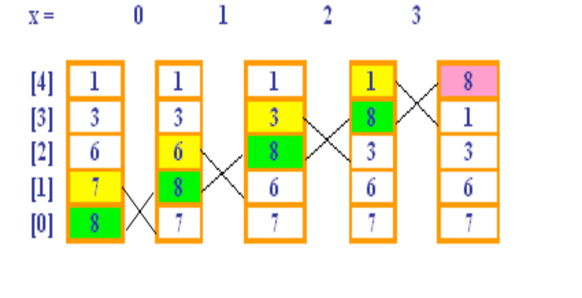
- The number of comparisons for all cases:

  (n-1)+(n-2)+…………..+2+1= n(n-1)/2 = O(n²)
- All lists with 5 elements need 10 comparisons to sort all the data.

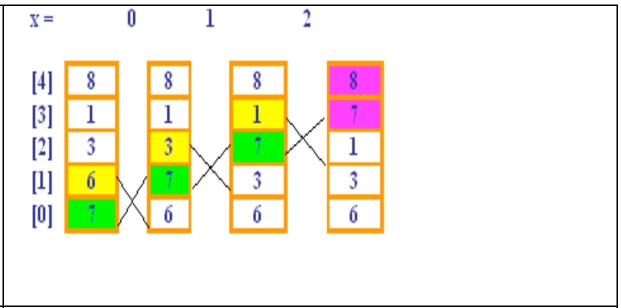
4.10. Example of worst case analysis for array [8 7 6 3 1]:

- The number of comparisons to sort data in this list:

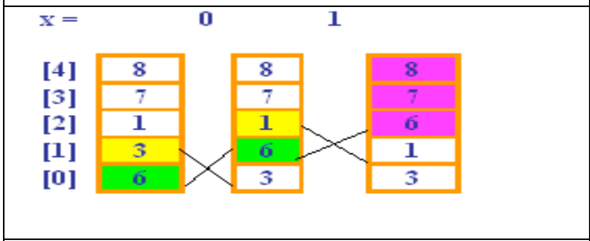
  (5-1) + (5-2) + (5-3) + (5-4) = 4 + 3 + 2 + 1 = 10.



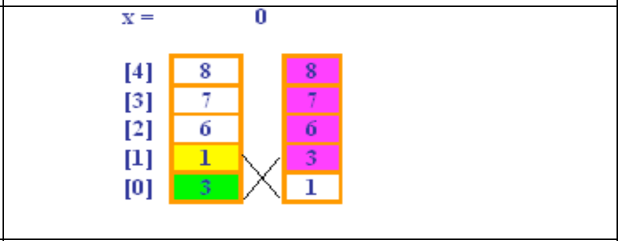
Pass 1 : (5-1=4 Comparisons)



Pass 2 : (5-2=3 Comparisons)



Pass 3 : (5-3= 2 Comparisons)

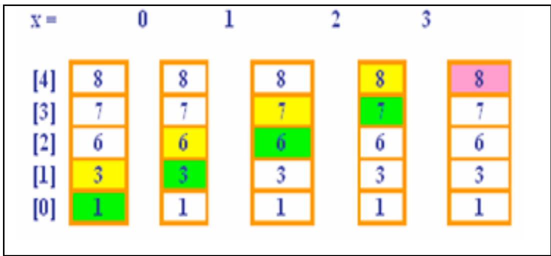


Pass 4 : (5-4=1 Comparison)

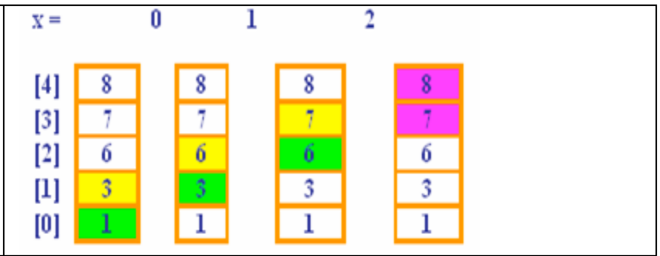
4.11. Example of best case analysis for array [1 3 6 7 8]:

- The number of comparisons to sort data in this list:

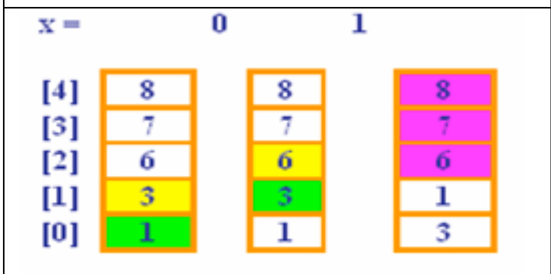
  (5-1) + (5-2) + (5-3) + (5-4) = 4 + 3 + 2 + 1 = 10.



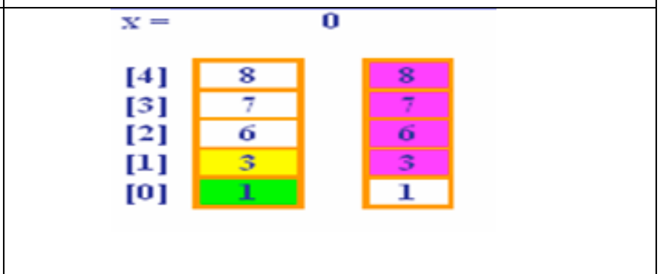
Pass 1 : (5-1=4 Comparisons)



Pass 2 : (5-2=3 Comparisons)



Pass 3 : (5-3=2 Comparisons)



Pass 4 : (5-4=1 Comparison)

- In the example given, it can be seen that the number of comparison for worst case and best case is the same - with 10 comparisons.

- The difference can be seen in the number of swapping elements. Worst case has maximum number of swapping: 10, while best case has no swapping since all data is already in the right position.
- For the best case, we can observe that starting with pass one, there is no exchange of data occur.
- From the example, it can be concluded that in any pass, if there is no exchange of data occur, the list is already sorted. The next pass shouldn't be continued and the sorting process should stop.

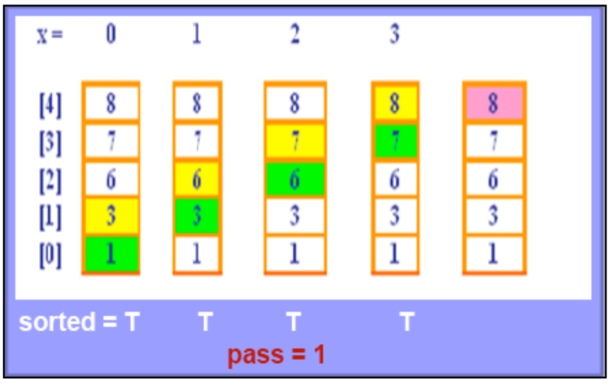
4.12. Improvement of the Bubble Sort algorithm

i. To improve the efficiency of Bubble Sort, a condition that check whether the list is sorted should be add at the external loop
ii. A Boolean variable, **sorted** is added in the algorithm to signal whether there is any exchange of elements occur in certain pass.
iii. In external loop, sorted is set to true. If there is exchange of data inside the inner loop, sorted is set to false.
iv. Another pass will continue, if sorted is false and will stop if sorted is true.

```
//Program 5.2
// Improved Bubble Sort
// Sorts items in an array into ascending order.

void bubbleSort(DataType data[], int n)
{  int temp;
   bool sorted = false;  // false when swaps occur
   // improvements i and ii.
   for (int pass = 1; (pass<n) && !sorted; ++pass)
   { // assume sorted - improvement iii.
     sorted = true;
      for (int x = 0; x < n-pass; ++x)
      {  if (data[x] > data[x+1])
         {  // exchange items
            temp = data[x];
            data[x] = data[x+1];
            data[x+1] = temp;
            sorted = false;
         //signal exchange -improvement iii and iv.
         }  // end if
      }  // end for
   }  // end for
}  // end bubbleSort
```

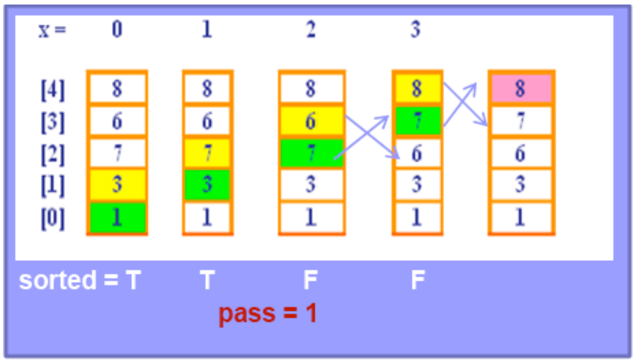
2 1 2 2 2 3	

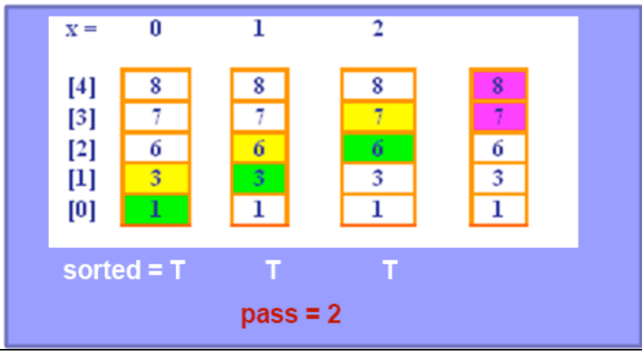
4.13. Example of improved Bubble sort for best case analysis for array [1 3 6 7 8]:



- In pass 1, there is no exchange of data occurs and variable sorted is always True. Therefore, condition statement in external loop will become false and the loop will stop execution. In this example, pass 2 will not be continued.
- For best case, the number of comparison between elements is 4, (n-1) which is O(n).

4.14. Example of improved Bubble sort for average case analysis for array [1 3 7 6 8]:





- For average case [1 3 7 6 8] we have to go through 2 passes only. The subsequent passes are not continued since the array is already sorted.
- Conclusion for improved Bubble Sort, the sorting time and the number of comparisons between data in average case and best case can be minimized.

4.15. Summary of bubble sort algorithm complexity (time consuming operations compares, swaps)

i. # of Compares
   - a *for* loop embedded inside a *while* loop
   - Worst Case (n-1)+(n-2)+(n-3) …+1 , or O(n2)
   - Best Case – (n-1) or O(n)

ii. # of Swaps
   - inside a conditional -> #swaps data dependent !!
   - Best Case 0, or O(1)
   - Worst Case (n-1)+(n-2)+(n-3) …+1 , or O(n2)

iii. Space
   - size of the array
   - an *in-place* algorithm

5.0 SELECTION SORT

5.1. Selection sort performs sorting by repeatedly find the next largest (or smallest) element in the array and put the element in the unprocessed portion of the array to the end of the unprocessed portion until the whole array is sorted.

5.2. To sort an array of record with selection sort, it works by taking multiple passes over the array with the following main activities

i. **Choose** the largest/smallest item in the array and place the item in its correct place.
ii. Choose the next larges/next smallest item in the array and place the item in its correct place.
iii. **Repeat** the process until all items are sorted.

5.3. Does not depend on the initial arrangement of the data and only appropriate for small *n* - O(n^2) algorithm.

[0]	[1]	[2]	[3]	[4]	[5]	
12	8	3	21	99	1	**Start - Unsorted List**
12	8	3	21	99	1	**Pass 1**- Find the largest element in the array (99) and put at the last index of the array
12	8	3	21	1	99	
12	8	3	21	1	99	**Pass 2**- Find the second largest element in the array (21) and put at the second last index of the array
12	8	3	1	21	99	
12	8	3	1	21	99	**Pass 3**- Find the next largest element in the array (12) and put at the current last index of the array
1	8	3	12	21	99	
1	8	3	12	21	99	**Pass 4**-Find the next largest element in the array (8) and put at the current last index of the array
1	3	8	12	21	99	
1	3	8	12	21	99	**Pass 5**-Find the next largest element in the array (3) and put at the current last index of the array
1	3	8	12	21	99	

5.4. Two functions in selection sort implementation are; **selectionSort()**and and **swap()**. Program 5.3 and Program 5.4 are the implementation of the two functions in C++.

`last` : index of the last item in the subarray of items yet to be sorted.

```
//Program 5.3
void selectionSort(DataType Data[], int n)
{
  for (int last = n-1; last >= 1; --last)
  {// select largest item in theArray
    int largestIndex = 0;
    // largest item is assumed start at index 0
    for (int p=1;p <= last; ++p)
    {  if (Data[p] > Data[largestIndex])
          largestIndex = p;
} // end for
   // swap largest item Data[largestIndex] with
   // Data[last]
    swap(Data[largestIndex],Data[last]);
  } // end for
}   // end selectionSort
```

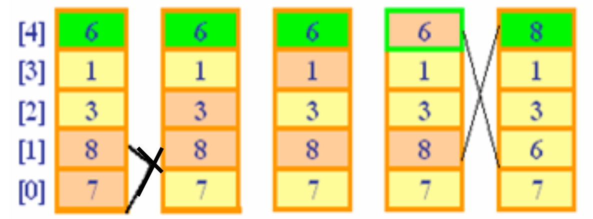
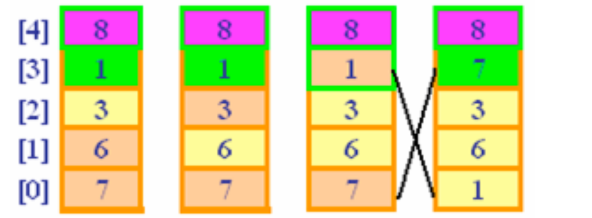
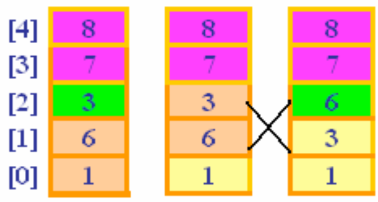
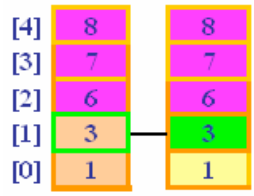
`largestIndex` : index of the largest item found

`swap`: change largest value with item at last index of the subarray.

Need to pass x and y by reference

```
//Program 5.4
void swap(DataType& x, DataType& y)
{  DataType temp = x;
   x = y;
   y = temp;
}  // end swap
```

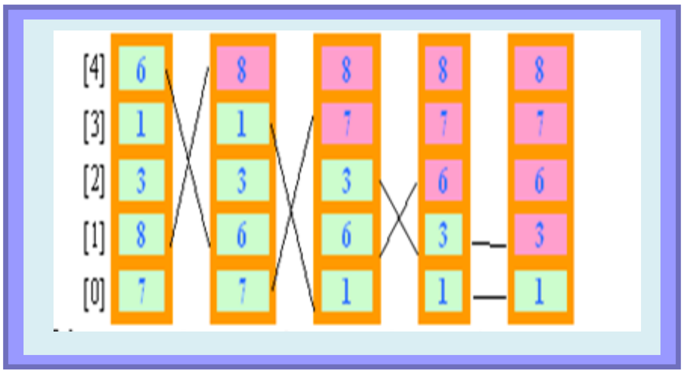
5.5. Example of selection sort implementation to sort array of integer [7 8 3 1 6] into ascending order:

	
Pass 1 **last** = 4 **largestIndex** = 0, 1, 1, 1 **p** = 1, 2, 3, 4	**Pass 2** **last** = 3 **largestIndex** = 0, 0, 0 **p** = 1, 2, 3, 4
	
Pass 3 **last** = 2 **largestIndex** = 0, 1 **p** = 1, 2	**Pass 4** **last** = 1 **largestIndex** = 0, 1 **p** = 1

- In pass 1, the largest value in the array will be searched from index 1 to index 4. The largest value is 8 and was found at index 1 and will be put at last(4). There are four comparisons in this pass.
- Below shows step by step changes in the list that show the swapping process during selection sort implementation on array [7 8 3 1 6].

5.6. Example of best case analysis for array [2 4 6 8 10]:

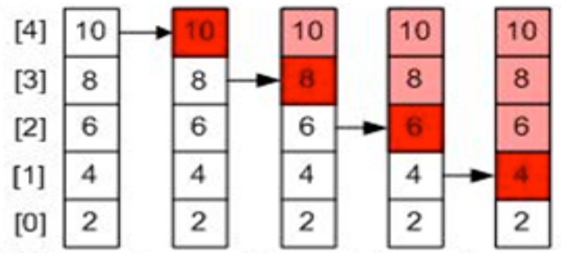
`last =` **4 3 2 1**



`largestIndex =` **1 0 1 1**

- Step by step changes in the list that show the swapping process during selection sort implementation on array [2 4 6 8 10]

`last =` **4 3 2 1**



`largestIndex =` **4 3 2 1**

5.7. Selection sort analysis:

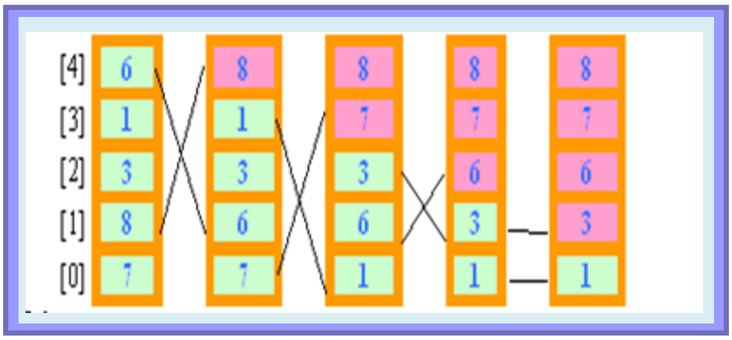
- For an array with size n, the external loop will iterate from *n-1* to *1*.
  for (int last = n-1; last>=1; --last)
- For each iteration, to find the largest number in subarray, the number of comparison inside the internal loop must is equal to the value of last.
  for (int p=1;p <=last; ++p)
- Therefore the total comparison for Selection Sort in each iteration is :
  (n-1) + (n-2) + ….. 2 + 1.
- Generally, the number of comparisons between elements in Selection Sort can be stated as follows:

$$(n-1)+(n-2)+.......+2+1=\frac{n(n-1)}{2}=O(n^2)$$

5.8. Similar to Bubble Sort, in any cases of Selection Sort (worst case, best case or average case) the number of comparisons between elements is the same.

- Example selection sort analysis for array [7 8 3 1 6]:

last = 4 3 2 1

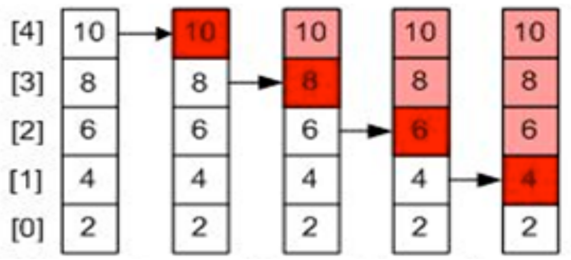


largestIndex = 1 0 1 1

Number of Comparisons: 4 + 3 + 2 + 1 = 10
For array n= 5 => (n-1) + (n-2) + ….+ 2 + 1 = n(n-1)/2 = O(n^2)

- Example selection sort analysis for best case array [10 8 6 4 2]:

last = 4 3 2 1



largestIndex = 4 3 2 1

Number of Comparisons for best case : 4 + 3 + 2 + 1 = 10
For array n= 5 => (n-1) + (n-2)+ …. + 2 + 1 = n(n-1)/2 = O(n^2)

5.9. Selection sort issues and improvement of the selection sort algorithm.

- It can be seen that the swapping process occur even though the largest index is at last. This is not efficient and can be improved by putting a condition statement as follows:

```
If (largestIndex !=last);
    swap(Data[largestIndex],Data[last]);
```

5.10. Summary of selection sort algorithm complexity:

- Time Complexity for Selection Sort is the same for all cases - worst case, best case or average case *O(n^2)*.
- The number of comparisons between elements is the same.
- The efficiency of Selection Sort does not depend on the initial arrangement of the data.

6.0 INSERTION SORT

6.1. Insertion sort performs sorting by repeatedly removes an element from the unsorted region, inserting it into the correct position in sorted region list, until all regions become sorted.

6.2. To sort an array of record with selection sort, it works by taking multiple passes over the array with the following main activities

- Partition the array into two regions: sorted and unsorted
  - i. Take each item from the unsorted region and insert it into its correct order in the sorted region
  - ii. Find next unsorted element and Insert it in correct place, relative to the ones already sorted

6.3. Insertion Sort is appropriate for small arrays due to its simplicity.

6.4. Example of Insertion sort operation with list of 6 elements [21 8 3 12 99 1] is as follows:

[0]	[1]	[2]	[3]	[4]	[5]	
21	8	3	12	99	1	**Start - Unsorted List**
21	8	3	12	99	1	**Insert 8 before 21:**
8	21	3	12	99	1	Created the sorted region from index [0] to [1], the unsorted region from index [2] to [5]
8	21	3	12	99	1	**Insert 3 before 8:**
3	8	21	12	99	1	Created the sorted region from index [0] to [2], the unsorted region from index [3] to [5]
3	8	21	12	99	1	**Insert 12 before 21:**
3	8	12	21	99	1	Created the sorted region from index [0] to [3], the unsorted region from index [4] to index [5]
3	8	12	21	99	1	**Keep 99 in place:**
3	8	12	21	99	1	Created the sorted region from index [0] to [4], the unsorted region from index [5]
3	8	12	21	99	1	**Insert 1 before 3:**
1	3	8	12	21	99	All elements are in sorted region

6.5. Insertion sort implementation:

```
//Program 5.5
void insertionSort(dataType data[])
 { dataType item;
    int pass, insertIndex;
    for(pass=1; pass<n; pass++)
   {
       item = data[pass];
       insertIndex = pass;
      while((insertIndex >0) &&
            (data[insertIndex -1]>item))
     {
       //insert the right item
      data[insertIndex]= data[insertIndex -1];
        insertIndex --;
     }
      data[insertIndex] = item;
     //insert item at the right place
   }
}
```

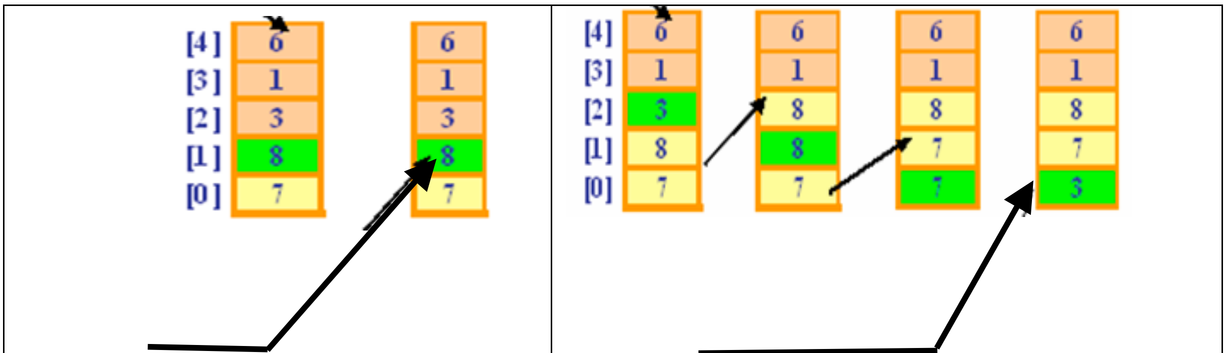
External loop to control sorted and unsorted regions

`item` : `item` in the unsorted region yet to be sorted in the sorted region

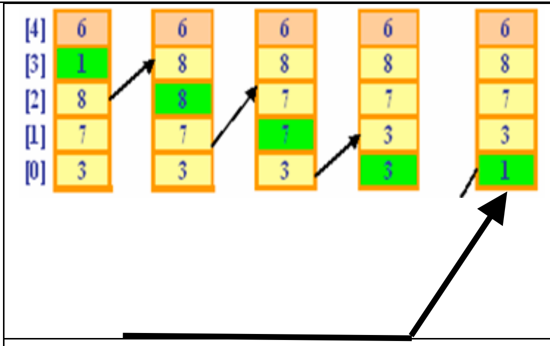
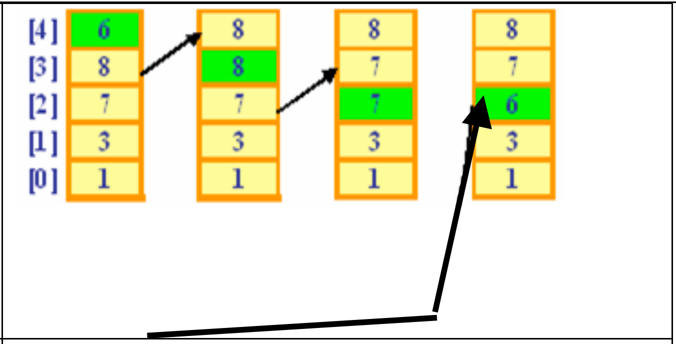
Internal loop to search for insertion spot & shift the sorted item to provide space to insert item

Insert item in the sorted region

6.6. Example of insertion sort implementation to sort array of integer [7 8 3 1 6] into ascending order:

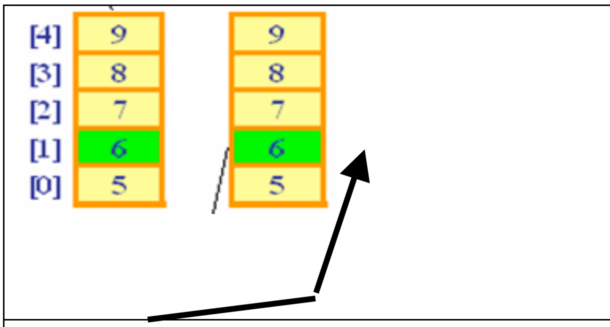
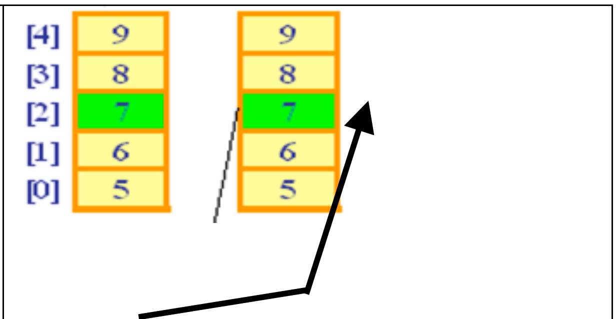
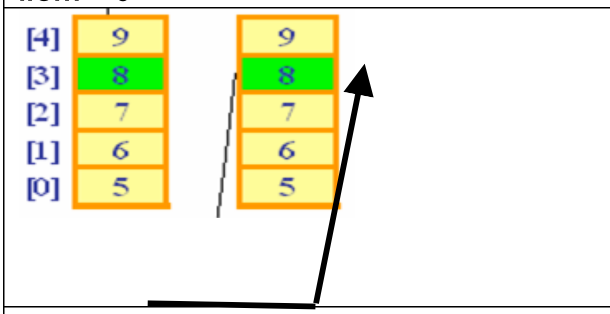
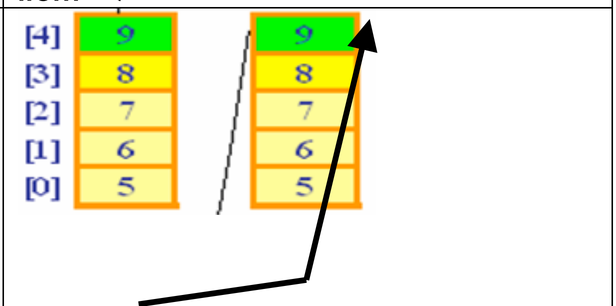


Pass 1	Pass 2
insertIndex : 1	insertIndex : 2 1 0
item = 8	item = 3

	
Pass 3 **insertIndex :** **3 2 1 0** **item = 1**	**Pass 3** **insertIndex :** **4 3 2 2** **item = 6**

- In Pass 1, item=8 > data[0]=7. while loop condition is false, therefore data[1] will be assigned with item = 8. The number of comparison is 1.
- In Pass 2, item to be insert is 3. Insertion point is from indeks 0-2, which is between 7 and 8. The number of comparison is 2.
- In Pass 3, item to be insert is 1. Insertion point is from indeks 0-3, which is between 3, 7 and 8. The number of comparison is 3.
- In Pass 4, item to be insert is 6. Insertion point is from indeks 0-4, which is between 1,3, 7 and 8. at index, item (6) > data[1]=3, while loop condition is false and therefore data[2] is assigned with value for item = 6. The number of comparison is 4.

6.7. Example of a best case analysis for array [5 6 7 8 9]:

	
Pass 1 **insertIndex = 1** **item = 6**	**Pass 2** **insertIndex = 2** **item = 7**
	
Pass 3 **insertIndex = 3** **item = 8**	**Pass 4** **insertIndex = 4** **item = 9**

- In Pass 1 item=6 > data[0]=1, while condition is false and data[1] is assigned with item=6. The number of comparison is 1.
- In Pass 2 item=7 > data[1]=1, while condition is false and data[2] is assigned with item=7. The number of comparison is 1.
- In Pass 3 item=8 > data[1]=1, while condition is false and data[3] is assigned with item=8. The number of comparison is 1.
- In Pass 3 item=9 > data[1]=1, while condition is false and data[4] is assigned with item=9. The number of comparison is 1.
- There are 4 passes to sort array with elements [5 6 7 8 9]. In each pass there is only 1 comparison.

  Example:

  Pass 1, 1 comparison
  Pass 2, 1 comparison
  Pass 3, 1 comparison
  Pass 4, 1 comparison

- In this example, the total comparisons for an array with size 5 are 4. Therefore, for best case, the number of comparison is *n-1* which gives linear time complexity - linear *O(n)*.

6.8. The worst case for insertion sort is when we have totally unsorted data. In each pass, the number of iteration for while loop is maximum.

6.9. For example worst case with 4 elements array.

Pass 4, 4 comparison - (n-1)
Pass 3, 3 comparison -(n-2)
Pass 2, 2 comparison -(n-3)
Pass 1, 1 comparison - (n-4)

- The number of comparisons between elements in Insertion Sort can be stated as follows:

$$\sum_{i=1}^{n-1} i = (n-1)+(n-2)+.......+2+1 = \frac{n(n-1)}{2} = O(n^2)$$

- Example of worst case analysis for array [9 7 5 3 1]:

  Pass 4 - Have to compare data at data[4-1], data[4-2], data[4-3] and data[4-4].
  Pass 3 - Have to compare data at data[3-1], data[3-2] and data[3-3].
  Pass 2 - Have to compare data at data[2-1], and data[2-2].
  Pass 1 - Have to compare data at data[1-1] only

  - The number of comparison is 10. i.e. (5-1)+ (5-2) + (5-3) + (5-4) = 10.

6.10. Summary of insertion sort algorithm complexity:

- How many **compares** are done?
  - 1+2+…+(n-1), *O(n^2)* for *worst case*
  - (n-1)* 1 , O(n) for *best case*
- How many element **shifts** are done?
  - 1+2+...+(n-1), *O(n^2)* for *worst case*
  - 0 , *O(1)* for *best case*
- How much **space**?
  - In-place algorithm

7.0 SUMMARY OF QUADRATIC SORTING ALGORITHMS COMPLEXITY

Efficiency	Insertion	Bubble	Selection
Comparisons:			
Best Case	O (n)	O(n^2)	O(n^2)
Average Case	O(n^2)	O(n^2)	O(n^2)
Worst Case	O(n^2)	O(n^2)	O(n^2)
Swaps			
Best Case	0	0	O(n)
Average Case	O(n^2)	O(n^2)	O(n)
Worst Case	O(n^2)	O(n^2)	O(n)

8.0 MERGE SORT

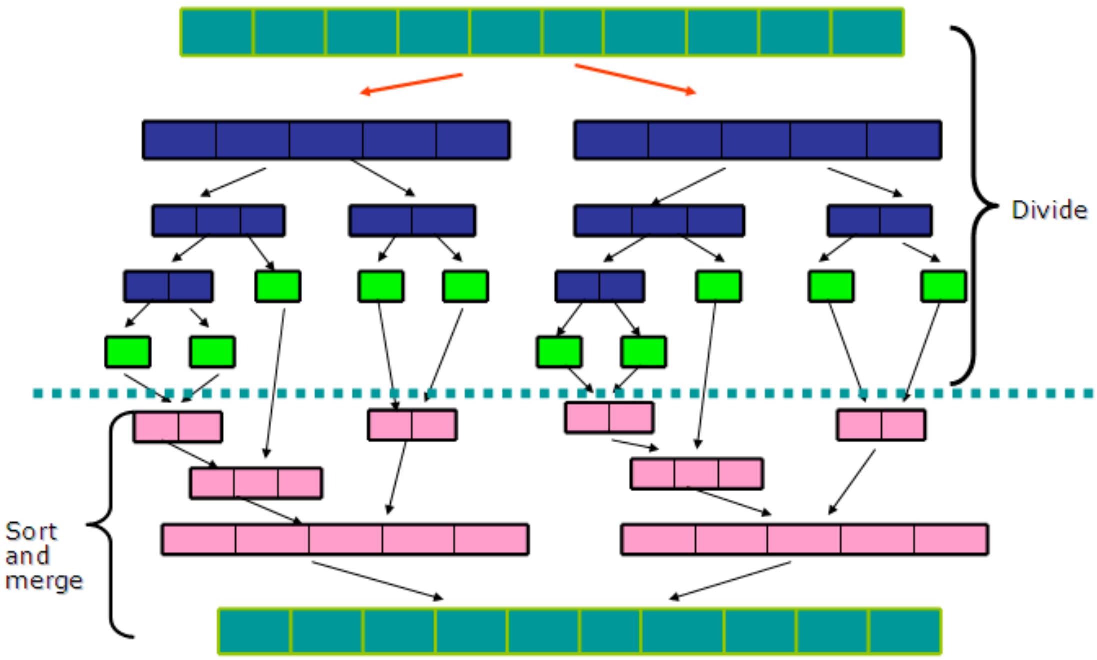
8.1. Merge Sort applies **divide** and **conquer** strategy. First, the list to be sorted is separated into two groups (*Divide*), *recursively* each group is sorted independently (*Conquer*) and then the two sorted groups are merged to a sorted sequence (*Combine*).

8.2. Three main steps in Merge Sort algorithm:

i. **Divide** an array into halves
ii. **Sort** each half
iii. **Merge** the sorted halves into one sorted array

8.3. The performance is independent of the initial order of the array items.

8.4. Illustration of the recursive Merge Sort algorithm strategy.



8.5. Two functions in merge sort implementation are;
MergeSort() and **Merge()**.

i. **mergeSort()**function
   - A recursive function that divide the array into pieces until each piece contain only one item.
   - The small pieces are merge into larger sorted pieces until one sorted array is achieved.

ii. **merge()**function
   - Compares an item into one half of the array with item in the other half of the array and,
   - Moves the smaller item into temporary array.
   - Then, the remaining items are simply moved to the temporary array.
   - The temporary array is copied back into the original array.

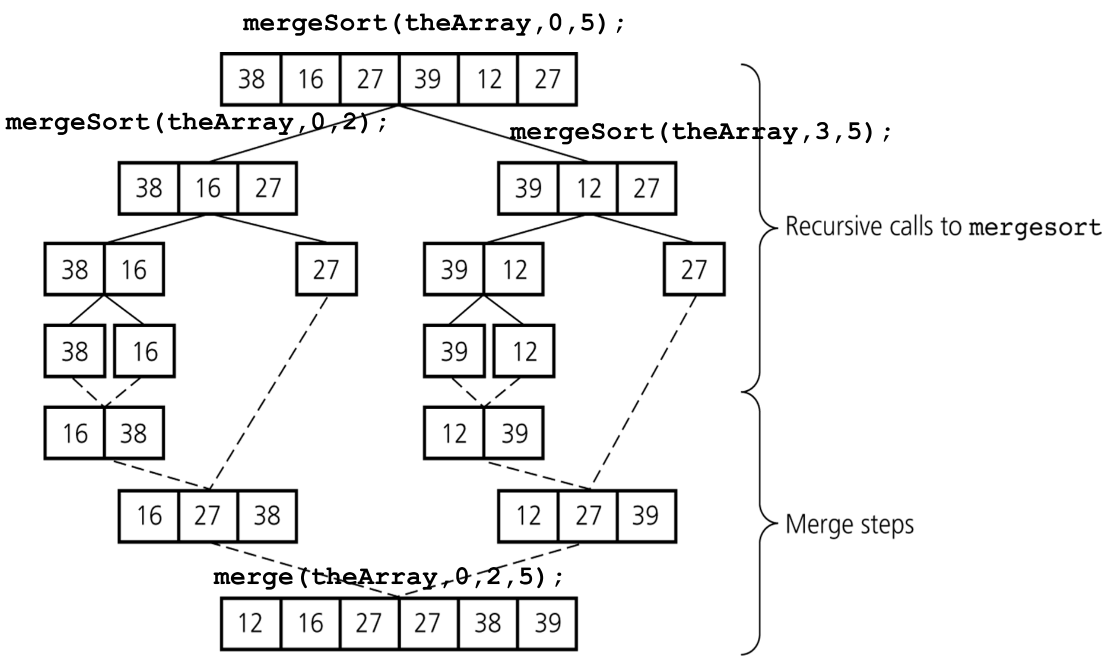
8.6. Program 5.6 is the implementation of the **mergeSort()** function in C++.

```
//Program 5.6
void mergeSort(DataType theArray[],int first,int last)
{ if (first < last)
  { // sort each half
    int mid = (first + last)/2;
    // index of midpoint
    // sort left half theArray[first..mid]
        mergesort(theArray, first, mid);
        //sort right half theArray[mid+1..last]
        mergesort(theArray, mid+1, last);
        // merge the two halves
        merge(theArray, first, mid, last);
} // end if
} // end mergesort
```

divide the array into pieces

small pieces are **merged**

8.7. Example of calling **mergeSort(theArray,0,5)**function in merge sort implementation to sort array of integer [38 16 27 39 12 27] into ascending order. In function **mergeSort(theArray,0,5)** three function will be called are **mergeSort(theArray,0,2)**, **mergeSort(theArray,3,5)** and **merge(theArray,0,2,5).**



8.8. Program 5.7 is the implementation of the **merge()** function in C++.

```
//Program 5.7
const int MAX_SIZE = maxNmbrItemInArry;
void merge(DataType theArray[],
          int first, int mid, int last)
{  DataType tempArray[MAX_SIZE];  // temp array
   int first1 = first;  // first subarray begin
   int last1  = mid;   // end of first subarray
   int first2 = mid + 1; // secnd subarry begin
   int last2  = last;   // end of secnd subarry

  // while both subarrays are not empty,
  // copy the smaller item into the temporary array
  int index = first1;
  // next available location in tempArray
  for (; (first1 <= last1) && (first2 <= last2); ++index)
  {if (theArray[first1] < theArray[first2])
     {  tempArray[index] = theArray[first1];
        ++first1;      }
     else
     {  tempArray[index] = theArray[first2];
        ++first2;      }
   } // end if


   for (; first1 <= last1; ++first1, ++index)
      tempArray[index] = theArray[first1];
   // finish off the second subarray, if necessary

   for (; first2 <= last2; ++first2, ++index)
      tempArray[index] = theArray[first2];


   // copy the result back into the original array
   for (index = first;index <= last; ++index)
      theArray[index] = tempArray[index];
   } // end merge function
```

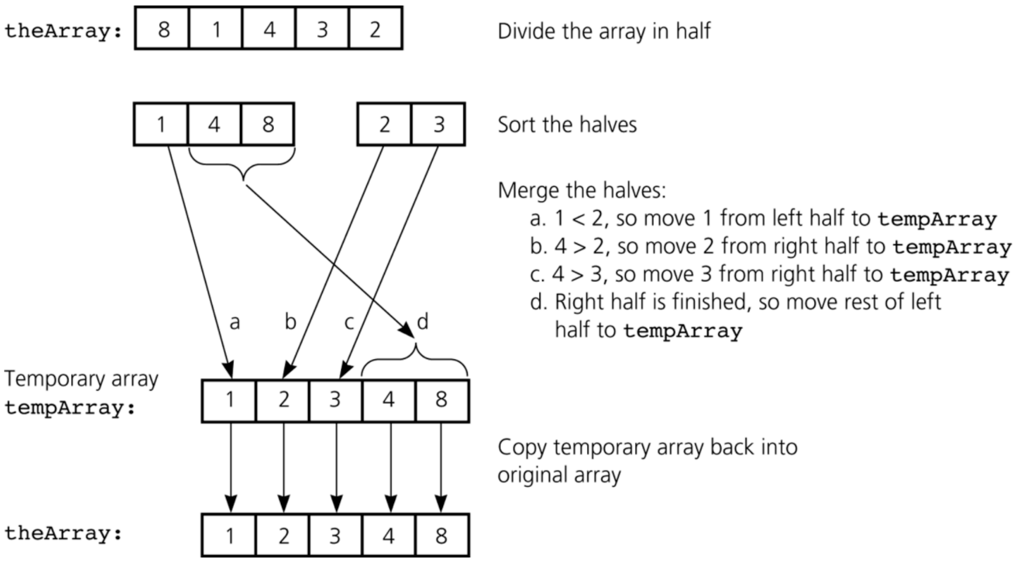
Duplicate the positions

Moves the smaller item into temporary array

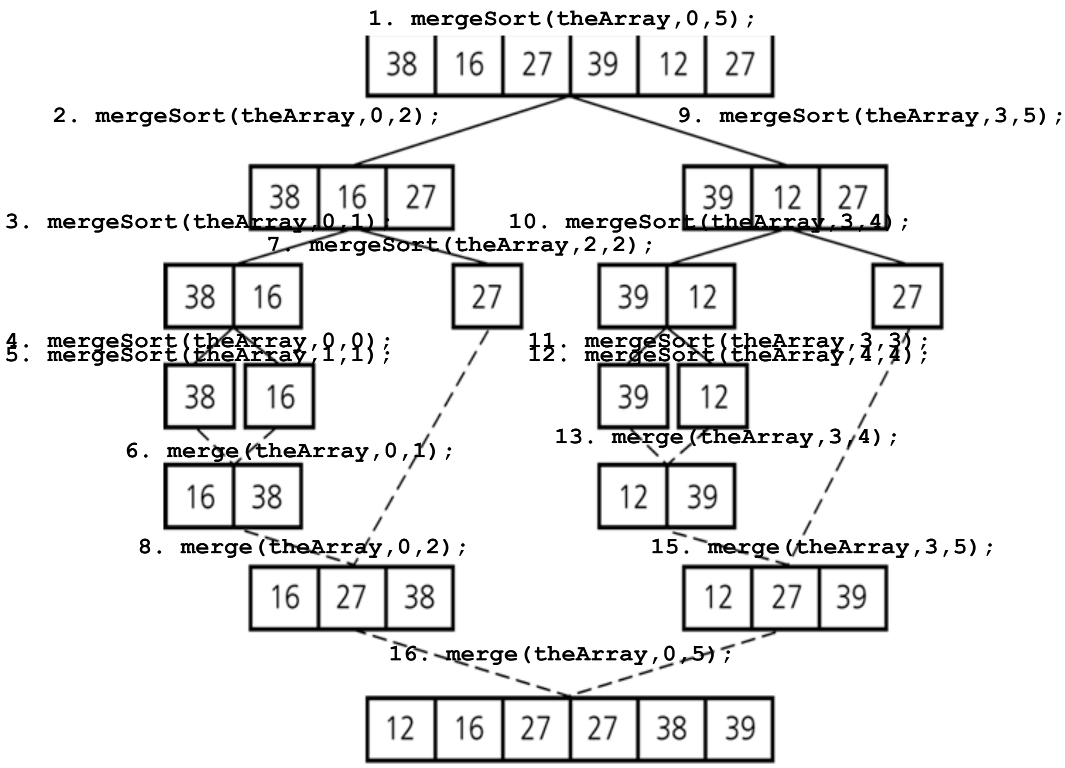
move the remaining items to the temporary array

The temporary array is copied back into the original array

8.9. Example of calling **merge(theArray,0,2,4)**function in merge sort implementation to sort array of integer [8, 1, 4, 3. 2] into ascending order.



8.10. Example of merge sort implementation to sort array of integer [38 16 27 39 12 27] into ascending order. The numbered function is the calling sequence of the functions in the algorithms.



- The execution of the C++ program to sort array of integers :
  [38 16 27 39 12 27]
  into ascending order gives the following sequence of output tracing.

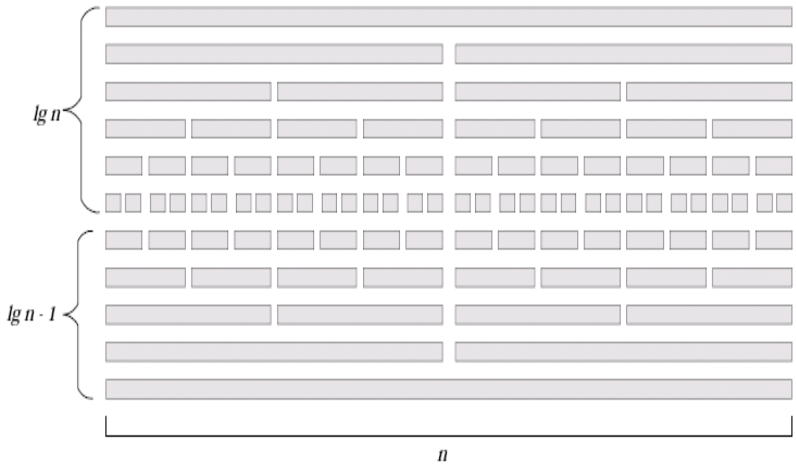
```
Unsorted data [38 16 27 39 12 27]
                                  14. mergeSort(theArray,5,5);
Content of divided sublist with first=0 & last=5 [38 16 27 39 12 27]
Content of divided sublist with first=0 & last=2 [38 16 27]
Content of divided sublist with first=0 & last=1 [38 16]
Content of divided sublist with first=0 & last=0 [38]
Content of divided sublist with first=1 & last=1 [16]
Content of merged list with first=0 & last=1 [16 38]
Content of divided sublist with first=2 & last=2 [27]
Content of merged list with first=0 & last=2 [16 27 38]
Content of divided sublist with first=3 & last=5 [39 12 27]
Content of divided sublist with first=3 & last=4 [39 12]
Content of divided sublist with first=3 & last=3 [39]
Content of divided sublist with first=4 & last=4 [12]
Content of merged list with first=3 & last=4 [12 39]
Content of divided sublist with first=5 & last=5 [27]
Content of merged list with first=3 & last=5 [12 27 39]
Content of merged list with first=0 & last=5 [12 16 27 27 38 39]

Sorted data [12 16 27 27 38 39]
Press any key to continue . . .
```

8.11. Merge sort analysis:

- The list is **always divided into two balanced** list (or almost balanced for odd size of list)
- The number of calls to repeatedly divide the list until there is one item left in the list is:

$$n + 2\frac{n}{2} + 4\frac{n}{4} + 8\frac{n}{8} + 16\frac{n}{16} + \ldots\ldots\ x\frac{n}{x}$$

- Assuming that the left segment and the right segment of the list have the equal size (or almost equal size), then $x \approx \lg n$. The number of iteration is approximately *n* lg *n*.
- The same number of repetition is needed to sort and merge the list (refer to the following illustration). Thus, as a whole the number of steps needed to sort data using merge sort is 2*n lg n,* which is O(*n lg n*).



8.12. Summary of merge sort analysis:

- Worse Case Analysis : O(n * log2n)
- Average case Analysis: O(n * log2n)
- Performance is independent of the initial order of the array items
- Advantage – Merge sort is an extremely fast algorithm
- Disadvantage – Merges sort requires a second array (temporary array) as large as the original array

9.0 QUICK SORT

9.1. Quick sort is similar with Merge sort in using **divide** and **conquer** technique.

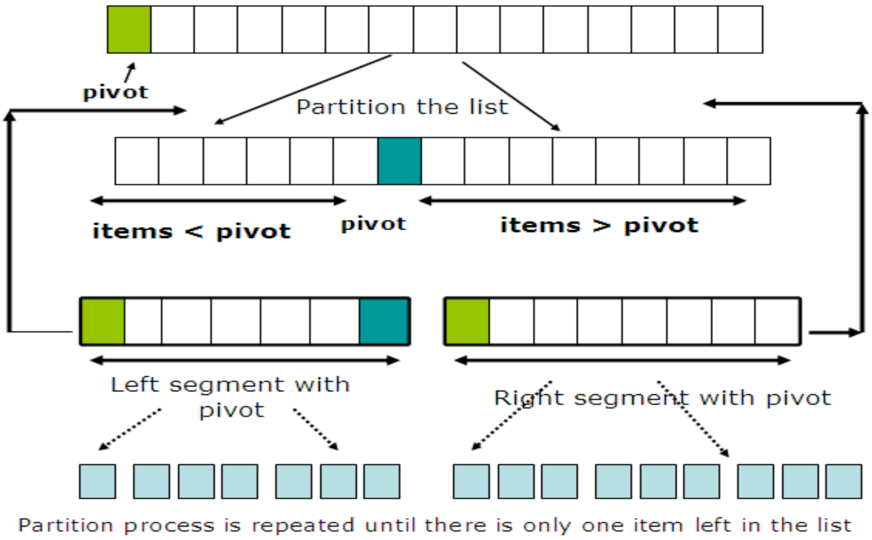
9.2. Differences of Quick sort and Merge sort :

Quick Sort	Merge Sort
Partition the list based on the *pivot* value	Partition the list by dividing the list into two
No merge operation is needed since when there is only one item left in the list to be sorted, all other items are already in sorted position.	Merge operation is needed to sort and merge the item in the left and right segment.

9.3. The **divide-and-conquer** algorithm strategy:

i. **Choose** a pivot (first element in the array)
ii. Partition the array about the pivot
   - o items < pivot
   - o items >= pivot
   - o Pivot is now in correct sorted position
iii. Sort the left section again until there is one item left
iv. Sort the right section again until there is one item left

9.4. Illustration of the recursive Quick Sort algorithm strategy.



9.5. Two functions in quick sort implementation are;
quickSort() and **partition()**.

i. **quickSort()**function
   - A recursive function that will partition the list into several sub lists until there is one item left in the sub list

ii. **partition()**function
   - The function organizes the data so that the items with values less than pivot will be on the left of the pivot, while the values at the right pivot contains items that are greater or equal to pivot.

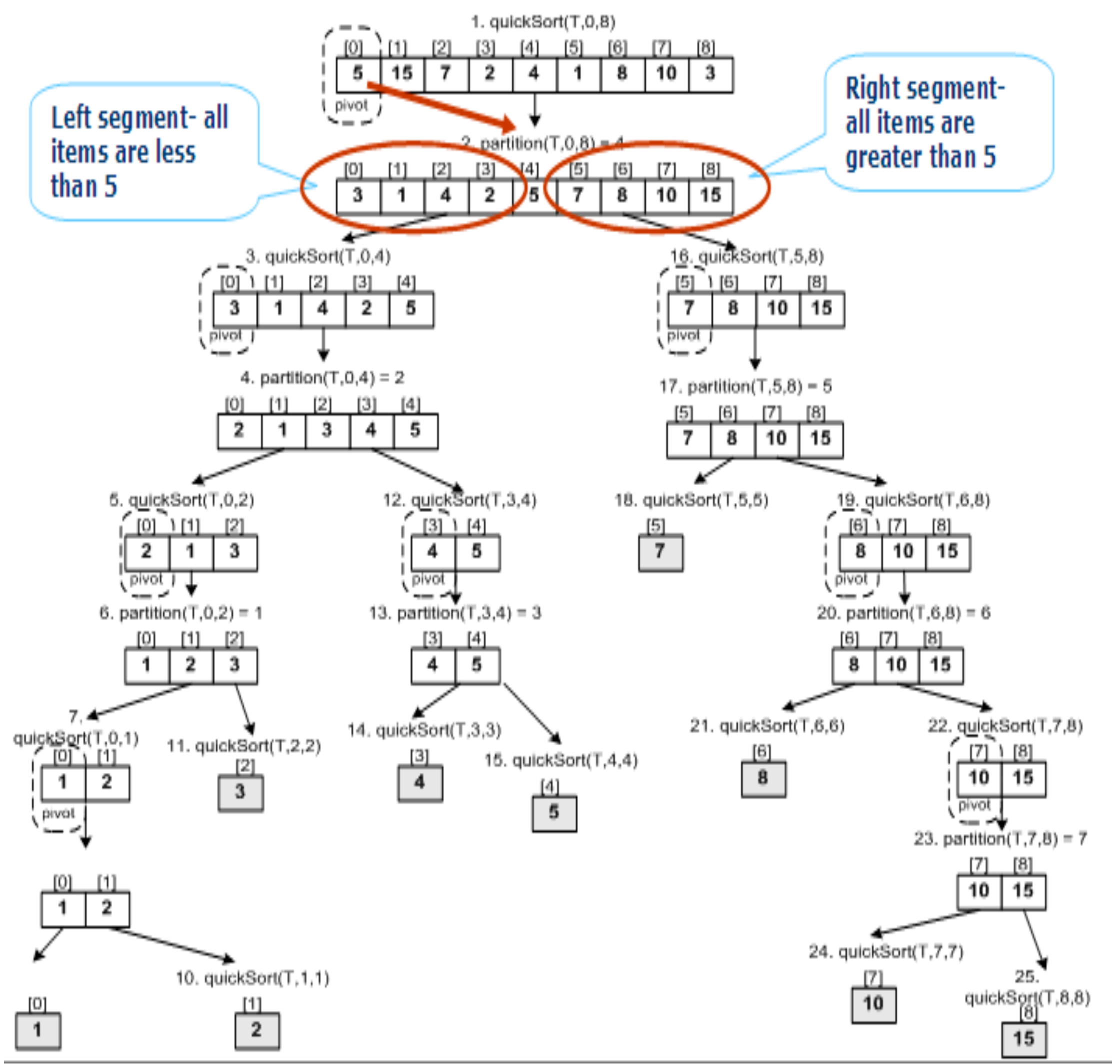
9.6. Program 5.8 is the implementation of the **quickSort()** function in C++.

- Recursive function that will partition the list into several sub lists until there is one item left in the sub list.
- Example of calling **quickSort(T,0,8)** function in quick sort implementation to sort array of integer [5 15 7 2 4 1 8 10 3] into ascending order. In function **quickSort(T,0,8)** three function will be called are **partition(T,0,8)**, **quickSort(T,0,4)** and **merge(T,5,8).** Refer to the following figure shows the calling function.

```
//Program 5.8
void quickSort (dataType arrayT[],
                int first , int last)
{
    int cut;
    if (first<last){
      cut = partition(T, first,last);
      quickSort(T,                    first,cut);
      quickSort (T, cut+1,                    last);
    }
}
```

Identify pivot or cutting point & **rearrange** the list based on pivot value

Cut the list into 2 sub lists based on cut value



9.7. Program 5.9 is the implementation of the **partition()** function in C++.

- Organize the data so that the items with values less than pivot will be on the left of the pivot, while the values at the right pivot contains items that are greater or equal to pivot.

```
//Program 5.9
int partition(int T[], int first,int last)
{
  int pivot, temp;
  int loop, cutPoint, bottom, top;
  pivot=T[first];
  bottom=first; top= last;
  loop=1;    //always TRUE

  while (loop) {
    while (T[top]>pivot){
    // find smaller value than
   // pivot from top array
      top--;
```

Identify pivot

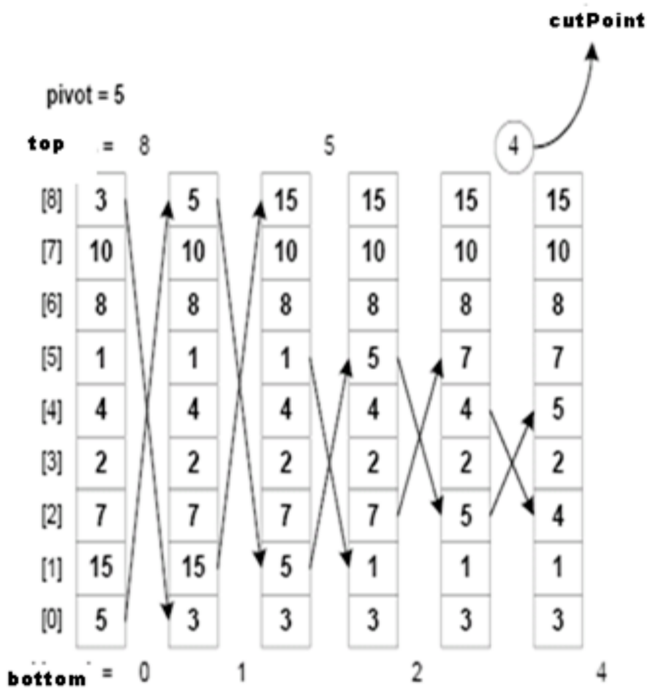
From top
Find value < pivot
& **skip** value > pivot

```
      }

      while(T[bottom]<pivot){
      //find larger value than
          //pivot from bottom
      bottom++;
      }

         if (bottom<top) {
      // change pivot place
        temp=T[bottom];
        T[bottom]=T[top];
        T[top]=temp;
      }
      else {
        loop=0; //loop false
        cutPoint = top;
     }//end if
   }// end while
   return cutPoint;
}//end function
```

From bottom
Find value > pivot
& **skip** value < pivot

Stop loop

- The following figure shows example of calling **partition(T,0,8)** function in quick sort implementation to sort array of integer [5 15 7 2 4 1 8 10 3] into ascending order. After execution of function **partition()**, pivot 5 will be placed at index 4 and the value 4, will be returned to function **quickSort()** for further partition.



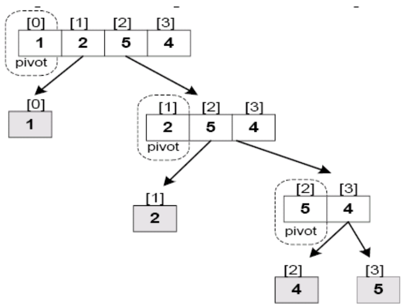
9.8. Referring to the quick sort implementation figure at point 9.6, the number at the sequence of calling functions for **quickSort()**and **partition()** functions can be mapped with the following output display.

```
Content of the array before sorting : 5 15 7 2 4 1 8

The sublist -> 1 with pivot = 5
3 15 7 2 4 1 8 10 3
The sublist -> 2 with pivot = 3
3 1 4 1 2 5
The sublist -> 3 with pivot = 2
2 1 3
The sublist -> 4 with pivot = 1
1 2
The sublist -> 5 with one piece item = 1

The sublist -> 6 with one piece item = 2

The sublist -> 7 with one piece item = 3

The sublist -> 8 with pivot = 4
4 5
The sublist -> 9 with one piece item = 4

The sublist -> 10 with one piece item = 5

The sublist -> 11 with pivot = 7
7 8 10 15
The sublist -> 12 with one piece item = 7

The sublist -> 13 with pivot = 8
8 10 15
The sublist -> 14 with one piece item = 8

The sublist -> 15 with pivot = 10
10 15
The sublist -> 16 with one piece item = 10
8 10 15
The sublist -> 17 with one piece item = 15
8 10 15
```

9.9. Quick sort analysis.

- The efficiency of quick sort depends on the **pivot value**.
- This class chose the first element in the array as pivot value.
- However, pivot can also be chosen at **random**, or **from the last element** in the array.
- The **worst case** for quick sort occurs when the **smallest** item or the **largest** item always be chosen as **pivot** value causing the left partition and the right partition not balance.

9.10. Example of the worst case quick sort: sorted array [1 2 5 4] causing imbalance partition.

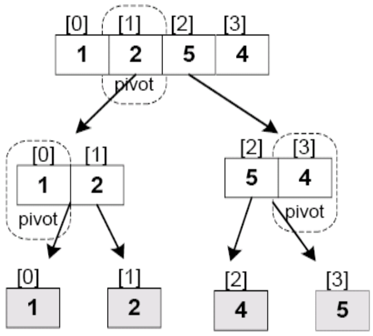


9.11. The **best** happens partition into **case** for quick sort when the list is **balance segment**.

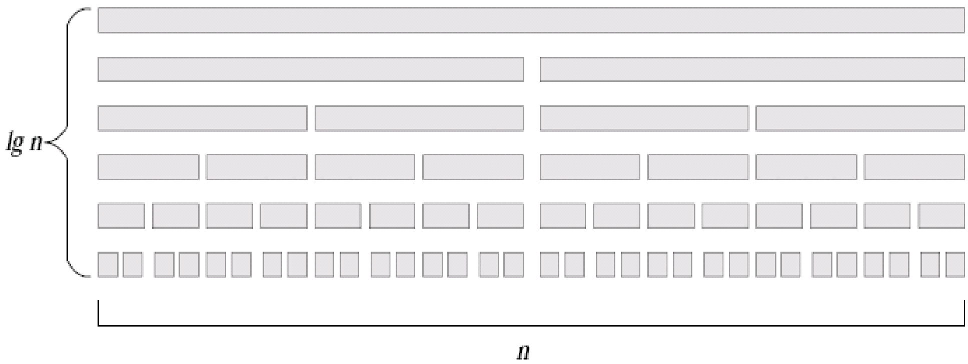
- Must pivot other balance choose the right that can put items in situation.
- The number of comparisons in partition process for base case situation is as follows:

$$n + 2\frac{n}{2} + 4\frac{n}{4} + 8\frac{n}{8} + 16\frac{n}{16} + ... x\frac{n}{x}$$

- The **best case** for quick sort happen when the **left segment and the right segment is balanced** (have the same size) with value x ≈ lg *n* .
- Example of best case quick sort: array[1 2 5 4].



9.12. The number of steps to get the balance segment while partitioning the array is *lg n* and the number of comparisons depend on the size list, *n*.



9.13. Summary of quick sort analysis:

- Average case: O(n * $\log_2 n$)
- Worst case: O(n^2) - When the array is already sorted and the smallest item is chosen as the pivot
- Quicksort is usually extremely fast in practice
- Even if the worst case occurs, quicksort's performance is acceptable for moderately large arrays

10.0 SUMMARY

10.1. Order-of-magnitude analysis and Big O notation measure an algorithm's time requirement as a function of the problem size by using a growth-rate function.

10.2. To compare the efficiency of algorithms

i. Examine growth-rate functions when problems are large
ii. Consider only significant differences in growth-rate functions

10.3. Worst-case and average-case analyses

- **Worst-case analysis** considers the maximum amount of work an algorithm will require on a problem of a given size
- **Average-case analysis** considers the expected amount of work that an algorithm will require on a problem of a given size

10.4. Order-of-magnitude analysis can be the basis of your choice of an ADT implementation.

10.5. Selection sort, Bubble sort, and Insertion sort are all O(n^2) algorithms. Quick sort and merge sort are two very fast recursive sorting algorithms.

10.6. Approximate growth rates of time required for eight sorting algorithms.

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	$n * \log n$	$n * \log n$
Quicksort	n^2	$n * \log n$

10.7. A comparison of growth-rate functions shows that *O(n log n)* algorithm is significantly faster than $O(n^2)$ algorithm.

Function	n = 10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$