Unit 7: Sorting

a. Introduction
b. Bubble sort
c. Insertion
d. Selection
e. Quick
f. Merge
g. Comparison and Efficiency of Sorting

TU Exam Question (2065)

- What is sorting? Describe the insertion.

Introduction

- Sorting is the process of arranging data in some sequence, i.e. in increasing order or decreasing order.
- ***Example:*** Suppose an array say ***value*** contains 10 elements as follows: 7, 9, 3, 6, 8, 2, 1, 4, 18, 5. After sorting in increasing order, the array ***value*** will contain its elements as: 1, 2, 3, 4, 5, 6, 7, 8, 9, 18.
- Sorting is categorized as: ***Internal Sorting*** and ***External Sorting***.
- ***Internal sorting*** is the sorting of data within an array only which is in computer's primary memory, while ***external sorting*** is the sorting of data from the external file by reading it from the secondary memory.

Introduction…

- ***Note:***
  - We will use the term ***file of size n*** to refer to a sequence of ***n*** items *r[0], r[1], …, r[n-1]*.
  - Each item in the file is called a ***record***.

TU Exam Question (2066)

- Write a short note on External Sorting.

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…

- It is clear that after the 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:

Iteration 0: (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 5 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
    End if
6.  Exit
```

C Function for Bubble Sort

Function call: ***bubble_sort(list, n);***

Function Definition:

```
void bubble_sort(int list[], int n)
{
int temp, i, j;

for(i=0;i<n;i++)
    {
    for(j=0;j<n-i-1;j++)
            {
            if(list[j]>list[j+1])
                        {
                        temp=list[j];
                        list[j]=list[j+1];
                        list[j+1]=temp;
                        }
            }
    }
}
```

Classwork

- Write C program for ***bubble sort***.

C Program for Bubble Sort

```
#include <stdio.h>
void bubble_sort(int [], int);
void main()
{
int list[100], 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]);

bubble_sort(list, n);

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

```
void bubble_sort(int list[], int n)
{
int temp, i, j;

for(i=0;i<n;i++)
    {
    for(j=0;j<n-i-1;j++)
            {
            if(list[j]>list[j+1])
                    {
                    temp=list[j];
                    list[j]=list[j+1];
                    list[j+1]=temp;
                    }
            }
    }
}
```

Efficiency of Bubble Sort

- In the ***bubble sort***, the first pass requires (n-1) comparisons to fix the highest element to its location, the second pass requires (n-2) comparisons, …, kth 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:

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

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

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

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

Model Question (2008)

- Explain why the straight selection sort is more efficient than the bubble sort.

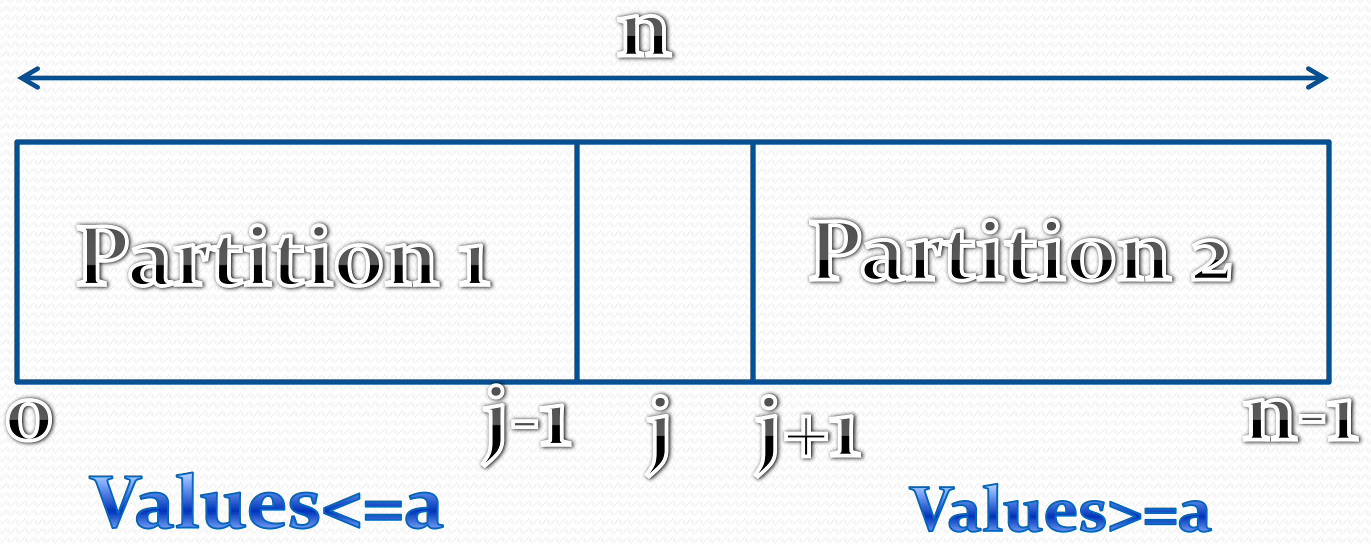
<u>Selection Sort</u>

Model question (2008)

- What is Big-O notation? Analyze the efficiency of quick sort.

Quick Sort

- It is also called ***partition exchange sort*** because it works by partitioning the array to be sorted.
- ***<u>Basic Idea:</u>***
  - Let ***x*** be an array, and ***n*** be the number of elements in the array.
  - Let us choose an element (called *pivot*) ***a*** from a specific position within the array (for example, ***a*** can be chosen as the first element so that ***a****=****x[0]***).
  - Now suppose that the elements of array ***x*** are partitioned in such a way that ***a*** is placed into position ***j*** of the array and the following conditions hold:
    1. Each of the elements in positions ***0*** through ***j-1*** is less than or equal to ***a***.
    2. Each of the elements in positions ***j+1*** through ***n-1*** is greater than or equal to ***a***.



- If these two conditions hold for a particular ***a*** and ***j***, ***a*** is the jth smallest element of array ***x***, so that ***a*** remains in position ***j*** when the array is completely sorted.
- If the same process is repeated for subarrays ***x[0]*** through ***x[j-1]*** and ***x[j+1]*** through ***x[n-1]*** and any subarrays created by the process in successive iterations, the final result is a sorted file.

Working of Quick Sort

- Let ***a=x[lb]*** be the pivot element whose exact final position is required.
- There are two pointers, ***up*** and ***down***, that are initialized to the upper and lower bounds of the subarray, respectively.
- At any point during execution, each element in a position above ***up*** is greater than or equal to ***a***, and each element in a position below ***down*** is less than or equal to ***a***.
- The two pointers ***up*** and ***down*** are moved towards each other in the following fashion:

  Step 1: Repeatedly increase the pointer ***down*** by one position while ***x[down]<=a***.

  Step 2: Repeatedly decrease the pointer ***up*** by one position while ***x[up]>a***.

  Step 3: If ***down<up***, interchange ***x[down]*** with ***x[up]***.

This process is repeated until the condition in step 3 fails (***down>=up***), at which point ***x[up]*** is interchanged with ***x[lb]*** (which equals pivot element ***a***), whose final position was required, and the process is repeated again recursively for the two subarrays on the left and right of the pivot element ***a***.

Trace quick sort for the following data:

25 57 48 37 12 92 86 83

- We illustrate the process of quick sort showing the positions of ***up*** and ***down*** as they are adjusted.
- The direction of the scan is indicated by an arrow at the pointer being moved.
- Let the pivot element ***a=x[lb]=25***.

down→							up
25	57	48	37	12	92	86	33

	down						up
25	57	48	37	12	92	86	33

	down						<-up
25	57	48	37	12	92	86	33

	down					<-up	
25	57	48	37	12	92	86	33
	down				<--up		
25	57	48	37	12	92	86	33
	down			up			
25	57	48	37	12	92	86	33

Interchange

	down			up			
25	12	48	37	57	92	86	33
	down→			up			
25	12	48	37	57	92	86	33

```
          down        up
25   12   48   37   57   92   86   33

          down     <--up
25   12   48   37   57   92   86   33

          down <--up
25   12   48   37   57   92   86   33

      <--up, down
25   12   48   37   57   92   86   33

     up   down
25   12   48   37   57   92   86   33
↑    ↑
└────┘
```

Interchange x[up] with x[lb] since up<=down

```
      up   down
12    25   48   37   57   92   86   33
```

➢At this point, 25 is in its proper position (position 1), and every element to its left is less than or equal to 25, and every element to its right is greater than or equal to 25.

➢Now we could proceed to sort the two subarrays (12) and (48 37 57 92 86 33) by applying the same method of quick sort.

➢Let's sort the second subarray now. For the second subarray, the pivot element, ***a=x[lb]=48***.

```
down→                    up
48   37   57   92   86   33

     down→               up
48   37   57   92   86   33

          down           up
48   37   57   92   86   33
          ↑              ↑
          └─Interchange──┘
```

		down			up
48	37	33	92	86	57
		down→			up
48	37	33	92	86	57
			down		up
48	37	33	92	86	57
			down		<--up
48	37	33	92	86	57
			down	<--up	
48	37	33	92	86	57
			<--up, down		
48	37	33	92	86	57

$$\begin{array}{cccccc} & & \text{up} & \text{down} & & \\ 48 & 37 & 33 & 92 & 86 & 57 \end{array}$$

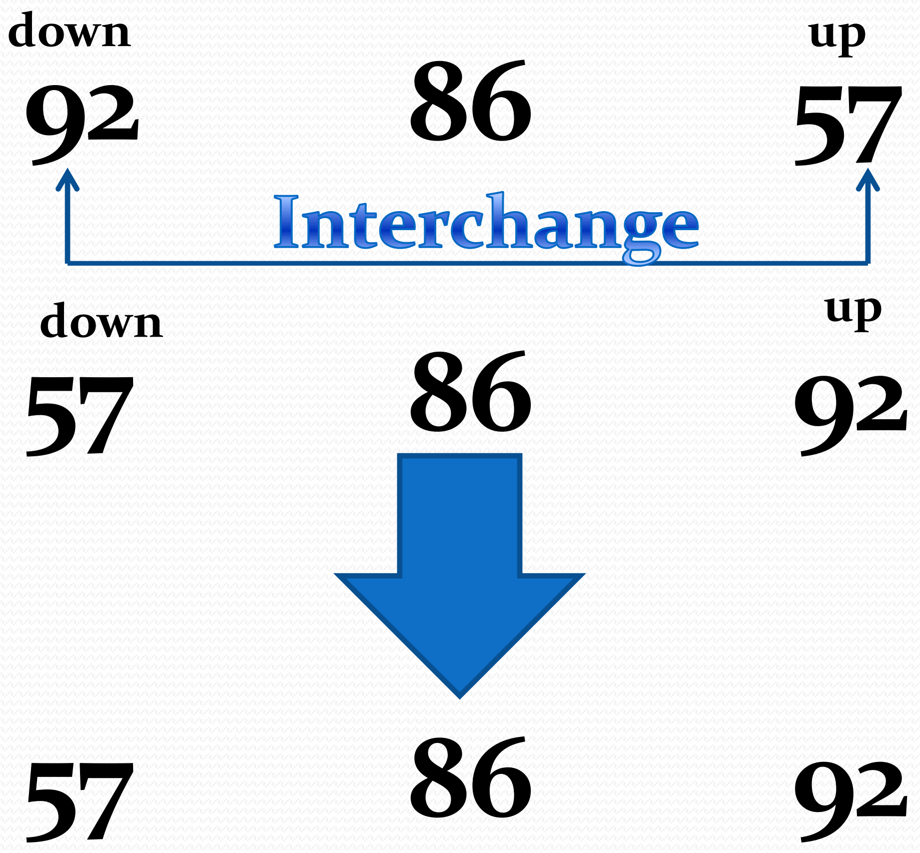
Interchange x[up] with x[lb] since up<=down

$$\begin{array}{cccccc} & & \text{up} & \text{down} & & \\ 33 & 37 & 48 & 92 & 86 & 57 \end{array}$$

- At this point, 48 is in its proper position, and every element to its left is less than or equal to 48, and every element to its right is greater than or equal to 48.
- Now we could proceed to sort the two subarrays (33 37) and (92 86 57) by applying the same method of quick sort.
- Let's sort the first subarray (33 37)now. For the first subarray ***a=x[lb]=33***.

$$\begin{array}{cc} \text{down} & \text{up} \\ 33 & 37 \end{array} \longrightarrow \begin{array}{cc} 33 & 37 \end{array}$$

➢Let’s sort the second subarray (92 86 57) now. For the second subarray ***a=x[lb]=92***.



➢***Hence the final sorted array is:***

12 25 33 37 48 57 86 92

C Program for Quick Sort

```
#include <stdio.h>
void quick_sort(int [], int, int);
void main()
{
int list[100], 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]);

quick_sort(list,0,n);

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

```
void quick_sort(int list[], int lb, int ub)
{
      int pivot, down, up;
      int temp;
                  pivot=list[lb];
                  down=lb;
                  up=ub;

      if(lb>=ub)
                  return;

      while(down<up)
                  {
                  while(list[down]<=pivot && down<ub)
                              down++;                   //move up the array
                  while(list[up]>pivot)
                              up--;                     //move down the array
                  if(down<up)                           //interchange
                              {
                              temp=list[down];
                              list[down]=list[up];
                              list[up]=temp;
                              }
                  }
      list[lb]=list[up];
      list[up]=pivot;

      quick_sort(list, lb, up-1);
      quick_sort(list, up+1, ub);
}
```

Homework

Sort the following file using quick sort:

45, 26, 77, 14, 68, 61, 97, 39, 99, 90

Efficiency of Quick Sort

- Let us assume that the file size ***n*** is a power of 2, say $\mathbf{n=2^m}$. Thus $\mathbf{m=\log_2 n}$.
- Let us also assume that the proper position of the pivot always turns out to be the exact middle of the subarray.
- In that case there are approximately ***n*** comparisons on the first pass, after which the file is split into two subfiles each of size **n/2**, approximately. For each of these two files there are approximately **n/2** comparisons and a total of four files each of size **n/4** are formed. Each of these files requires **n/4** comparisons yielding a total of **n/8** subfiles. After halving the subfiles **m** times, there are **n** files of size 1.

Efficiency of Quick Sort…

- Thus the total number of comparisons for the entire sort is approximately

  n + 2*(n/2) + 4*(n/4) + 8*(n/8) + … + n*(n/n)

  = n + n + n + n + … + n (m terms) comparisons.

- There are **m** terms because the file is subdivided **m** times.
- Thus the total number of comparisons is O(n*m) or ***O(n log n)***.
- Thus the efficiency of quick sort is ***O(n log n)***.

Efficiency of Quick Sort…

- ***Note:***
  - There is an absurd property of quick sort: it works ***best*** for files that are "***completely unsorted***" (**O(n log n)**) and ***worst*** for files that are "***completely sorted***".
  - ***Worst case analysis:*** Let the original array is sorted. Then ***x[lb]*** is in its correct position. Now the original file is split into subfiles of size ***0*** and ***n-1***. If this process continues, a total of ***n-1*** subfiles are sorted, the first of size ***n***, the second of size ***n-1***, the third of size ***n-2***, and so on. Assuming ***k*** comparisons to rearrange a file of size ***k***, the total number of comparisons to sort the entire file is:

$$n + (n-1) + (n-2) + \ldots + 2$$

which is $O(n^2)$.

Merge Sort

TU Exam Question (2066)

- Divide and conquer algorithm taking reference to merge sort.

TU Exam Question (2066)

- Explain the use of Big-O notation in analyzing algorithms. Compare sorting time, efficiencies of Quick Sort and Merge Sort.