ESc101 : Fundamental of Computing

I Semester 2008-09

Lecture 35

- Input from keyboard and output to Console : (Lecture 34)
- Input and Output from/to file
  (using the existing classes and method of package java.io.*)
- Sorting algorithms based on recursion
  - Quick Sort,
  - Merge Sort (in some future class)

Reminder : did you solve “Tower of Hanoi” problem

- There are three Towers : A,B,C
- Tower A has n discs arranged one above the other in the increasing order of the radii from top to bottom.
- The towers B and C are empty.
- We can move one disc only in a single step.
- **AIM :** Describe the steps to transfer all discs from tower A to tower B.

Constraint : We can never place a bigger disc on a smaller one.

Design a method Tower_of_Hanoi(n)

which prints the detailed instruction about the movement of discs in order to transfer n discs from A to B.

Input Output from/to Console : Lecture 34

Classes used :

- InputStreamReader
- BufferedReader

An interactive program for sorting numbers

file selection_sort_i.java

Input Output from/to File - without buffers

without buffer means reading or writing just one character at a time.

Input Output from/to File - without buffers

one character at a time

Classes used :

- FileReader
- FileWriter

Example : copying a file to another file.

File : IO_without_buffer.java

Output to a File - with buffers

Printing a String at a time

Classes used :

- FileWriter
- PrintWriter

Example : Storing random number in a file.

File : random_example_file.java

Input from a File - with buffers

reading a line at a time

Classes used :

- FileReader
- BufferedReader

Example : Reading numbers from a file.

File : reading_numbers_from_file.java

Selection Sort

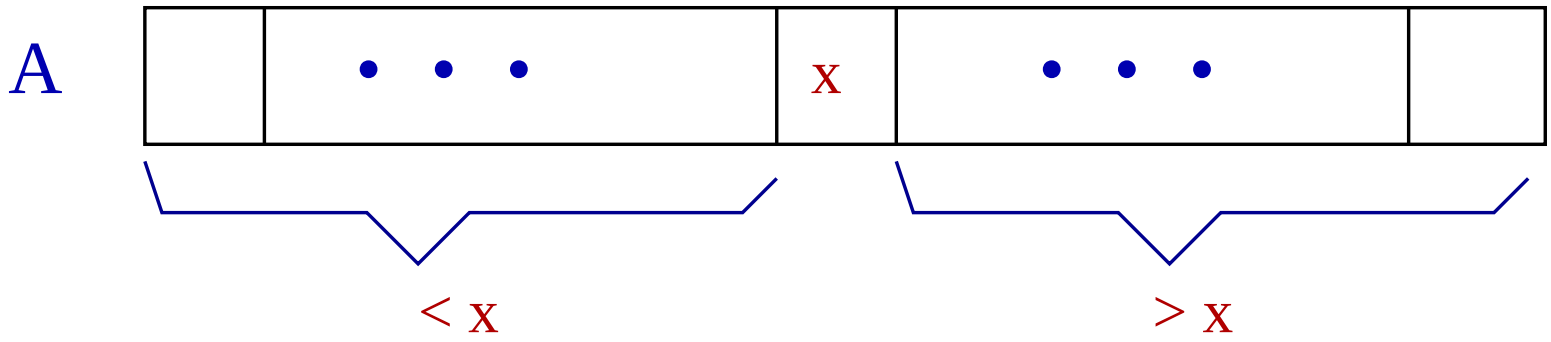
```
int  index_of_smallest_value(int[] A,int i)
//returns integer j such that A[j] is smallest among A[i], A[i+1],...

SelectionSort(int [] A)
{

    for(int count=0;count<A.length;count=count+1)
        {
            int j = index_of_smallest_value(A, count);
            if(j != count)
                swap_values_at(A,j,count);
        }
}
```

discussed in some earlier lecture

Quick Sort

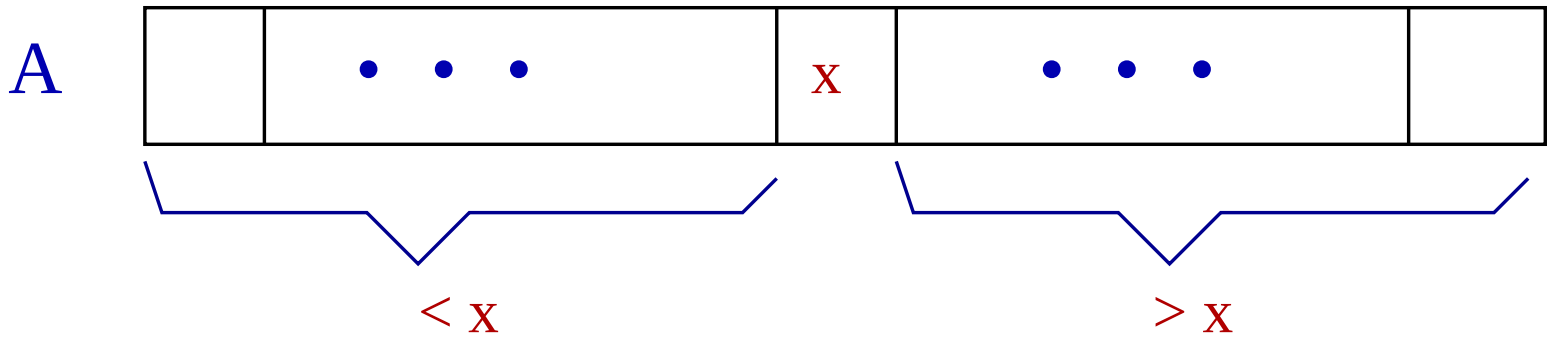
Partitioning an array into two parts

Given an array and an element $x \in A$, rearrange elements within A so that



Partitioning an array into two parts

Given an array and an element $x \in A$, rearrange elements within A so that



Can be easily done using an extra array

Partitioning an array into two parts

Write a method

```
int Partition(int[] A, int left, int right)
```

- which partitions the array with `x` = `A[right]`
- and returns the smallest integer `i` such that `A[i]=x`

Partitioning

Write a method

```
int Partition(int[] A, int left, int right)
```

- which partitions the array with `x` = `A[right]`
- and returns the smallest integer `i` such that `A[i]=x`

Homework : Given an implementation of Partition()

which does not use any extra array

Partitioning without using extra array

Solution of the homework on next slide

Read it after you have made a sincere attempt.

Partitioning without using extra array

NSE_index is abbreviation for **Index of Next Smaller Element**. This variable stores the index of array where we are going to store the next element $\leq$ x.

```
public static int partition (int[] A, int left, int right)
    {   int x=A[right];
        int NSE_index = left;
        for(int i = left; i<=right-1; i=i+1)
        {
            if(A[i]<=x)
            {   swap(A,i,NSE_index);
                NSE_index = NSE_index + 1;
            }
        }
//Finally moving x to its appropriate place.
        swap(A,right,NSE_index);
        return NSE_index;
    }
}
```

Quick Sort

```
public static void Qsort(int[] A, int left, int right)
{
    if( ??)
    {
                ???      ;
                ???      ;
                ???      ;
    }
}
```

Quick Sort

```
public static void Qsort(int[] A, int left, int right)
{
    if(left<right)
    {
                ???      ;
                ???      ;
                ???      ;
    }
}
```

Quick Sort

```
public static void Qsort(int[] A, int left, int right)
{
    if(left<right)
    {
        int mid = partition(A, left, right);
                ???      ;
                ???      ;
    }
}
```

Quick Sort

```
public static void Qsort(int[] A, int left, int right)
{
    if(left<right)
    {
        int mid = partition(A, left, right);
        Qsort(A,     ??    );
        Qsort(A,     ??    );
    }
}
```

Quick Sort

```
public static void Qsort(int[] A, int left, int right)
{
    if(left<right)
    {
        int mid = partition(A, left, right);
        Qsort(A,left,mid-1);
        Qsort(A,    ??    );
    }
}
```

Quick Sort

```
public static void Qsort(int[] A, int left, int right)
{
    if(left<right)
    {
        int mid = partition(A, left, right);
        Qsort(A,left,mid-1);
        Qsort(A,mid+1,right);
    }
}
```

You can observe that the size of problem corresponding to recursive calls decreases always. Hence the program will eventually terminate.

Show execution of Qsort on an array of 8 elements

using paper and pen.

Comparing Selection Sort and Quick Sort

```
for(n=2000,n<25000;n=n+1000)
   {    Generate an array A of size n;
        Fill it with random integers;
        Create copy B of array A;
        Execute Quick sort on A and measure time.
        Execute SelectionSort on B and measure time.
   }
```

Measuring time taken a method M

```
long start = System.currentTimeMillis();
M();
long stop = System.currentTimeMillis();
System.out.println(stop-start);
```

Note : `System.currentTimeMillis()` returns a long which corresponds to current time in milliseconds.

Comparing Selection Sort and Quick Sort

The file : **Comparing_Sorting_Algo.java**