Response Time - How long does it take to execute a transaction?

Transaction - Personal software - word processing, etc.
- **E-commerce transaction**
- **Fetch web page**
- **execute web search function**
- **solve a differential equation**
- **sort a list**
- **search a list.**

End-to-end Response Time - The time taken by a user of a system to a response from the system.

End-to-end Response Time is the composition of the execution of many different pieces of software. We will see that one poor choice of algorithms or programs can lead to poor end-to-end response. Therefore we study the execution behavior of individual algorithms and programs. If each program is efficient then the system will usually be efficient.

Topics for Lecture

Memory requirements of algorithms and programs.

Execution time for algorithms and programs.
  - **Operation counts for algorithms**
  - **Operation counts for programs.**

Searches and Sorts as examples
  - **Generic Programming**
  - **The Comparable interface**
  - **Search/Sort Algorithms**

Measurement the execution time of programs.
  - **Use of system clocks**

Profiling the execution of a program.
  - **Where is the time of execution being spent?**

Space (memory cost) - The memory used by a program is determined by the algorithm it executes, the compiler which generates the program, the data set which is being processed, the skill with which the program is written, etc.

If an algorithm uses excessive memory then that memory is not available for use by other programs. Excessive memory use may degrade performance by over-using the virtual memory system of the computer.

Normally memory use is dominated by the size of the data being processed.

Memory usage is not always obvious - recursive programs.(We will return to recursion later.)

Invocation paths and stack and heap storage.

Time Complexity of Algorithms

Operation Count as metric for execution time.

Choose some set of dominant operations and count the number of occurrences of these operations during the execution of the algorithm

Polynomial Evaluation - simple summation

```
y = P(x) c[n]*x**n + c[n-1]*x**(n-1)……. +c[0];
y = coeff[0];
{
for (i =1,i<=n,i=i+1)
y = y + coeff[i]*x;
x = x *x;
}
```

n additions and 2 n multiplications

o(n) operations

Polynomial Evaluation - Horner’s Rule

y = P(x) = (….(c[n]*x +c[n-1])*x + c[n-2])*x…..)*x + c[0]

```
y = coeff[n];
for i = n-1,i>=0, i = i-1)
{
y = y *x;
y = y +coeff[I];
}
```

n additions and n multiplications
(o) n operations but coefficient of n is smaller.

This algorithm is faster than the simple algorithm.

Table of Operation Counts for various operation count functions

N	n	nlogn	n**2	n**4	n**10	2**n
10	.01	.03	.1	10	10**7	1
100	.10	.66	10	10**5	3000y	10**13y
10**6	10**3	2*10**4	17M	32y	10**43y	

Entries are: **in microseconds if no units**
in minutes if M is units
in years in y is units.

Assumption is that the computer executing the program operates at 1x109 operations per second.**

Best Case, Worst Case and Average Case Operation Counts

```
public class SequentialSearch
{
   /** Search the unordered array a for x.
     * CAUTION: Object.equals must be overridden for
this
     * method to work correctly
     * @return position of x if found; -1 otherwise */
   public static int sequentialSearch(Object [] a,
       Object x)
   {
      int i;
      for (i = 0; i < a.length && !x.equals(a[i]); i++);
      if (i == a.length) return -1;
      else return i;
   }
}
```

Assumptions:

1. The elements in the array have values chosen from a random distribution in the length of the array.

```
RandomIntGenerator values = new RandomIntGenerator(0,a.length);
a[i] = values.draw();
```

2. The values for which searches are made are chosen randomly from a distribution in the length of the array.

```
searchValue  = values.draw();
```

What is the smallest number of comparisons which may have to be made to find if the searchValue is in a[]?
What is the largest number of comparisons which will have to be made to find if the searchValue is in a[]?
What is the average number of comparisons which will have to be made to find if the searchValue is in a[]?

Performance Issues - Choice of Algorithms, Measurement

```
public class BinarySearch                    {
  /** Search the ordered array a for x.
* @return position of x if found; -1 otherwise */
  public static int binarySearch(Comparable [] a, Comparable x)
  {// Search a[0] <= a[1] <= ... <= a[a.length-1] for x.
    int left = 0;
    int right = a.length - 1;
    while (left <= right)
    {
      int middle = (left + right)/2;
      if (x.equals(a[middle]))
        return middle;
      if (x.compareTo(a[middle]) > 0)
        left = middle + 1;
      else
        right = middle - 1;
    }
    return -1; // x not found
  }
                                             }
```

Performance Issues - Choice of Algorithms, Measurement

Analysis of Binary Search

Assumptions:

1. The elements in the array have values chosen from a random distribution in the length of the array.

RandomIntGenerator values = new RandomIntGenerator(0,a.length);
a[i] = values.draw();

2. The values for which searches are made are chosen randomly from a distribution in the length of the array.

searchValue = values.draw();

What is the smallest number of comparisons which may have to be made to find if the searchValue is in a[]?
What is the largest number of comparisons which will have to be made to find if the searchValue is in a[]?
What is the average number of comparisons which will have to be made to find if the searchValue is in a[]?

Performance Issues - Choice of Algorithms, Measurement

Operation Counts, Execution Time, Algorithm Choice - Sort Algorithms

	best	average	worst	Sort 10000 ints
1. Bubble Sort -	n		n*n	4096ms
2. Selection Sort -	n*n		n*n	2143ms
3. Insertion Sort -	n		n*n	1222ms
4. Mergesort -	nlogn	n*logn	nlogn	40ms
5. Quicksort -	nlogn	nlogn	n**2	210ms

There may be occasions when one knows the properties of the list to be sorted so that choice of an algorithm with a good best case behavior will be a good choice. For example, bubble sort works well on a nearly sorted list.

Performance Issues - Choice of Algorithms, Measurement

Bubble Sort

1. **Traverse the sequence of values comparing each pair of values during the traverse.**
2. **Exchange each pair of values so that the smaller (larger) of each pair precedes the larger (smaller) value.**
3. **Continue traversals until no exchanges are made.**
4. **Then the sequence is sorted by value.**

40 2 1 43 3 65 0 -1 58 3 42 4
2 1 40 3 43 0 -1 58 3 42 4 65
1 2 3 40 0 -1 43 3 42 4 58 65

Performance Issues - Choice of Algorithms, Measurement

```
public class BubbleSortBig
{           public static void sort(int[] a)
            {int n = a.length;
             boolean sorted = false;
             int i, temp;
            while (!sorted)
                        { sorted = true;
                                    for (i=0; i < n-1; i++)
                                                { if (a[i] > a[i+1])
                                                  temp = a[i];
                                                 a[i] = a[i+1];
                                                 a[i+1] = temp;
                                                 sorted = false;
                                                   }
                        }
            }
}
```

Performance Issues - Choice of Algorithms, Measurement

```
public static void main(String[] args)
  {
            System.out.println(" Enter Number of items to be sorted");
            int n = SavitchIn.readLineInt();
            int[] a = new int[n];
            int i;
            // fill the array with random values
            for (i = 0; i < a.length; i++)
                        a[i] = (int)(Math.random() * 100);
            long startTime = System.currentTimeMillis();
            sort(a);
            long endTime = System.currentTimeMillis();
            long runTime = endTime - startTime;
            System.out.println("Run time in milliseconds = " + runTime);
            float runTimePerItem = (float)runTime/(float)n;
            System.out.println(" Run time per item = " + runTimePerItem);
  }
}
```

Performance Issues - Choice of Algorithms, Measurement

```
import structure.*;
import corejava.*;
public class BubbleSort
{
   public static void main(String args[])
   {
            RandomIntGenerator values = new RandomIntGenerator(1,10000);
            System.out.println(" Enter Number of items to be sorted");
            int n = SavitchIn.readLineInt();
            int data[] = new int[n];
            int i;
            for (i = 0; i < n; i++)
            {
               data[i] = values.draw();
            }
```

Performance Issues - Choice of Algorithms, Measurement

```
            long startTime = System.currentTimeMillis();
            bubbleSort(data,n);
            long endTime = System.currentTimeMillis();
            long runTime = endTime - startTime;
            System.out.println("Run time in milliseconds = " + runTime);
            float runTimePerItem = (float)runTime/(float)n;
            System.out.println(" Run time per item = " + runTimePerItem);
   }

   //+bubbleSort
   public static void bubbleSort(int data[], int n)
   // pre: 0 <= n <= data.length
   // post: values in data[0..n-1] in ascending order
   {
            int numSorted = 0;  // number of values in order
            int index;                // general index
```

Performance Issues - Choice of Algorithms, Measurement

```
while (numSorted < n)
        {
          // bubble a large element to higher array index
          for (index = 1; index < n-numSorted; index++)
          {
                if (data[index] < data[index-1])
                  swap(data,index,index-1);
          }
          // at least one more value in place
          numSorted++;
        }
  }
  //-bubbleSort
```

Performance Issues - Choice of Algorithms, Measurement

```
//+swap
  public static void swap(int data[], int i, int j)
  // pre: 0 <= i,j < data.length
  // post: data[i] and data[j] are exchanged
  {
        int temp;
        temp = data[i];
        data[i] = data[j];
        data[j] = temp;
        //-symmetry
        //-symmetryA
        //-symmetryB
  }
  //-swap
}
```

Performance Issues - Choice of Algorithms, Measurement

```
//+vectorSort

  public static void swap(Vector data, int i, int j)
  // pre: 0 <= i,j < data.size()
  // post: i-th j-th elements of data are exchanged
  {
        Object temp;
        temp = data.elementAt(i);
        data.setElementAt(data.elementAt(j),i);
        data.setElementAt(temp,j);
  }
  //-vectorSort
```

Performance Issues - Choice of Algorithms, Measurement

Selection Sort

1. Pick the largest value in a sequence and put it at the end of the sequence.
2. Repeat for each value in the sequence.

40 2 1 43 3 65 0 -1 58 3 42 4
40 2 1 43 3 4 0 -1 58 3 42 65
40 2 1 43 3 4 0 -1 42 3 58 65

Actually the pointers to the values are usually moved.

Performance Issues - Choice of Algorithms, Measurement

```
public class SelectionSort
{
public static void main(String args[])
{           RandomIntGenerator values = new RandomIntGenerator(1,10000);
            System.out.println(" Enter Number of items to be sorted");
            int n = SavitchIn.readLineInt();
            int data[] = new int[n];
            int i;
            for (i = 0; i < n; i++)
            {  data[i] = values.draw();      }
            long startTime = System.currentTimeMillis();
            selectionSort(data,n);
            long endTime = System.currentTimeMillis();
            long runTime = endTime - startTime;
            System.out.println("Run time in milliseconds = " + runTime);
            float runTimePerItem = (float)runTime/(float)n;
            System.out.println(" Run time per item = " + runTimePerItem);
   }
 }    //+selectionSort
```

Performance Issues - Choice of Algorithms, Measurement

```
public static void selectionSort(int data[], int n)
// pre: 0 <= n <= data.length
// post: values in data[0..n-1] are in ascending order
{           int numUnsorted = n;
            //+selectMaximum
            int index;
            // general index
            int max;
            // index of largest value
            //-selectMaximum
            while (numUnsorted > 0)
                        {
                        //+selectMaximum
                        // determine maximum value in array
                        max = 0;
                        for (index = 1; index < numUnsorted; index++)
                                    {     if (data[max] < data[index]) max = index;         }
//-selectMaximum
swap(data,max,numUnsorted-1);
numUnsorted--;      }    }
//-selectionSort
```

Performance Issues - Choice of Algorithms, Measurement

```
//+swap
 public static void swap(int data[], int i, int j)
// pre: 0 <= i,j < data.length
// post: data[i] and data[j] are exchanged    {
            int temp;
            temp = data[i];
            data[i] = data[j];
            data[j] = temp;    }
//-swap

//+vectorSort
public static void swap(Vector data, int i, int j)
// pre: 0 <= i,j < data.size()
// post: i-th j-th elements of data are exchanged    {
            Object temp;
            temp = data.elementAt(i);
            data.setElementAt(data.elementAt(j),i);
            data.setElementAt(temp,j);    }
//-vectorSort}
```

Performance Issues - Choice of Algorithms, Measurement

Insertion Sort

1. Partitions the list into sorted and unsorted regions. The sorted region initially has one entry.
2. Take the first entry in the unsorted region and insert it in the proper location in the sorted region and increase the size of the sorted region by one.
3. This continues until the unsorted region has no entries.

```
40   2   1   43   3   65   0    -1   58   3   42   4
2   40   1   43   3   65   0    -1   58   3   42   4
1    2  40   43   3   65   0   -1   58   3   42   4
1    2  40   43   3   65   0   -1   58   3   42   4
1    2   3   40  43   65   0   -1   58   3   42   4
```

In the case where the entries are already sorted the algorithm makes n-1 passes but makes no exchanges.

Performance Issues - Choice of Algorithms, Measurement

```
public class InsertionSort{    public static void main(String args[])
{                              System.out.println(" Enter Number of items to be sorted");
               int n = SavitchIn.readLineInt();
                int[] a = new int[n];
                int i;
               // fill the array with random values
               for (i = 0; i < a.length; i++)
                              a[i] = (int)(Math.random() * 100);
               long startTime = System.currentTimeMillis();
               insertionSort(a,n);
               long endTime = System.currentTimeMillis();
               long runTime = endTime - startTime;
               System.out.println("Run time in milliseconds = " + runTime);
               float runTimePerItem = (float)runTime/(float)n;
               System.out.println(" Run time per item = " + runTimePerItem);
}
```

Performance Issues - Choice of Algorithms, Measurement

```
 public static void insertionSort(int data[], int n)
 // pre: 0 <= n <= data.length
// post: values in data[0..n-1] are in ascending order
{ int numSorted = 1;              // number of values in place    int index;
  int index;              // general index
               while (numSorted < n)
               {              // take the first unsorted value
                              int temp = data[numSorted];
                              // ...and insert it among the sorted:
                              for (index = numSorted; index > 0; index--)
                              {              if (temp < data[index-1])
                                                            { data[index] = data[index-1]; }
                                             else {  break; }
                              }
                // re-insert value
               data[index] = temp;
               numSorted++;
               }
 }
```

Performance Issues - Choice of Algorithms, Measurement

```
//+swap    public static void swap(int data[], int i, int j)
 // pre: 0 <= i,j < data.length
// post: data[i] and data[j] are exchanged
{int temp;
temp = data[i];
data[i] = data[j];
data[j] = temp;    }
//-swap}
```

Performance Issues - Choice of Algorithms, Measurement

Merge Sort

1. **Split sequence in half into two subsequences.**
2. **Continue splitting until sequence is a set of pairs.**
3. **Exchange positions of pairs so that smaller (larger) is first.**
4. **Merge pairs choosing smallest (largest) among the pairs to be first in the merged sequence.**
5. **Continue until all of the subsequences have been merged.**

Comment - Uses extra memory.

```
40    2   1    43   3   65  |  0   -1   58   3    42   4
40    2   1  | 43   3   65  |  0   -1   58  | 3    42   4
40 |  2   1  | 43 |  3   65  | 0  |  -1   58  | 3 |  42   4
1   2   40 |  43   3   65   | 0     -1   58  | 3     42   4
1   2   3    40  43  65    | -1     0    3   4      42   58
-1  0   1     2     3   3      4     40   42  43     58   65
```

Performance Issues - Choice of Algorithms, Measurement

```
import structure.*;
public class MergeSort
{    public static void main(String args[])
   {          System.out.println(" Enter Number of items to be sorted");
              int n = SavitchIn.readLineInt();
              int[] a = new int[n];
              int i;
              // fill the array with random values
              for (i = 0; i < data.length; i++)
                          {  a[i] = (int)(Math.random() * 100);      }
              long startTime = System.currentTimeMillis();
              mergeSort(data,n);
              long endTime = System.currentTimeMillis();
              long runTime = endTime - startTime;
              System.out.println("Run time in milliseconds = " + runTime);
              float runTimePerItem = (float)runTime/(float)n;
              System.out.println(" Run time per item = " + runTimePerItem);
  }
```

Performance Issues - Choice of Algorithms, Measurement

```
public static void mergeSort(int data[], int n)
 // pre: 0 <= n <= data.length
 // post: values in data[0..n-1] are in ascending order
 {
              mergeSortRecursive(data,new int[n],0,n-1);
 }
//-mergeSort
 //+mergeSortRecursive
 private static void mergeSortRecursive(int data[],  int temp[], int low, int high)
// pre: 0 <= low <= high < data.length
// post: values in data[low..high] are in ascending order
{             int n = high-low+1;
              int middle = low + n/2;
              int i;
              if (n < 2) return;
```

Performance Issues - Choice of Algorithms, Measurement

```
// move lower half of data into temporary storage
for (i = low; i < middle; i++)  {
    temp[i] = data[i]; }
// sort lower half of array
mergeSortRecursive(temp,data,low,middle-1);
// sort upper half of array
mergeSortRecursive(data,temp,middle,high);
// merge halves together
merge(data,temp,low,middle,high);    }
//-mergeSortRecursive
//+mergeOperation
```

Performance Issues - Choice of Algorithms, Measurement

```
private static void merge(int data[], int temp[], int low, int middle, int high)
// pre: data[middle..high] are ascending
//      temp[low..middle-1] are ascending
 // post: data[low..high] contains all values in ascending order
{             int ri = low;  // result index
              int ti = low;   // temp index
              int di = middle;   // destination index
              // while two lists are not empty merge smaller value
              while (ti < middle && di <= high)
              {              if (data[di] < temp[ti])
                             {      data[ri++] = data[di++];  // smaller is in high data          }
                             else {  data[ri++] = temp[ti++]; // smaller is in temp  }
              }
              // possibly some values left in temp array
              while (ti < middle)
              {                 data[ri++] = temp[ti++];     }
              // ...or possibly some values left (in correct place) in data array
}
```

Performance Issues - Choice of Algorithms, Measurement

```
//+swap
 public static void swap(int data[], int i, int j)
// pre: 0 <= i,j < data.length
// post: data[i] and data[j] are exchanged
 {            int temp;
              temp = data[i];
              data[i] = data[j];
              data[j] = temp;
}
 //-swap
```

Performance Issues - Choice of Algorithms, Measurement

QuickSort

1. Choose a value.
2. Determine its final position in the array of values and put it in that position.
3. Partition the values into two sets, one below the final position established in step 2 and one above the final position established in step 2.
4. Apply steps two and three to each of these partitions.
5. Continue step 4 until all of the partitions are of size 1.
6. The array is then sorted.

40	2	1	43	3	65	0	-1	58	3	42	4
4	2	1	3	3	-1	0	40	58	65	42	43
0	2	1	3	3	-1	4	40	43	42	58	65
-1	0	1	3	3	2	4	40	42	43	58	65
-1	0	1	3	3	2	4	40	42	43	58	65
-1	0	1	3	3	2	4	40	42	43	58	65
-1	0	1	2	3	3	4	40	42	43	58	65
-1	0	1	2	3	3	4	40	42	43	58	65
-1	0	1	2	3	3	4	40	42	43	58	65

Green numbers are pivots, blue numbers are in their final positions.

Performance Issues - Choice of Algorithms, Measurement

```
import structure.*;
public class QuickSort
{    public static void main(String args[])
     {       System.out.println(" Enter Number of items to be sorted");
             int n = SavitchIn.readLineInt();
             int a[] = new int[n];
             int i;
             for (i = 0; i < a.length; i++)
                        { a[i] = (int)(Math.random() * 100);     }
             long startTime = System.currentTimeMillis();
             quickSortIterative(a,n);
             long endTime = System.currentTimeMillis();
             long runTime = endTime - startTime;
             System.out.println(" Run time in milliseconds = " + runTime);
             float runTimePerItem = (float)runTime/(float)n;
             System.out.println(" Run time per item = " + runTimePerItem);
  }
}
```

Performance Issues - Choice of Algorithms, Measurement

The Java package java.util provides a class, Arrays, which defines and implements many useful operations on arrays including sorting and searching. Please look at the material on Arrays at

http://java.sun.com/products/jdk/1.2/docs/api/java/util/Arrays.html#sort(java.lang.Object[])

Take a look at the several methods defined in Arrays including the method public void sort(Object[] O) at

http://java.sun.com/products/jdk/1.2/docs/api/java/util/Arrays.html#sort(java.lang.Object[])

Then click on the link to the interface Comparable which we have already discussed once.

http://java.sun.com/products/jdk/1.2/docs/api/java/lang/Comparable.html

Performance Issues - Choice of Algorithms, Measurement

A sort() method defined in class Arrays in java.util.

public static void sort(Object[] a)

Sorts the specified array of objects into ascending order, according to the natural ordering of its elements.

All elements in the array must implement the Comparable interface. Furthermore, all elements in the array must be mutually comparable (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the array).

This sort is guaranteed to be stable: equal elements will not be reordered as a result of the sort. The sorting algorithm is a modified mergesort (in which the merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist). This algorithm offers guaranteed n*log(n) performance, and can approach linear performance on nearly sorted lists.

Parameters:

a - the array to be sorted.

Throws:

ClassCastException - if the array contains elements that are not mutually comparable (for example, strings and integers).

See Also:

Comparable

Performance Issues - Choice of Algorithms, Measurement

```
import java.util.*;
import corejava.*;

public class EmployeeSortTestLibrary
{  public static void main(String[] args)
   {  Employee[] staff = new Employee[3];

      staff[0] = new Employee("Harry Hacker", 35000,
         new Day(1989,10,1));
      staff[1] = new Employee("Carl Cracker", 75000,
         new Day(1987,12,15));
      staff[2] = new Employee("Tony Tester", 38000,
         new Day(1990,3,15));
      Arrays.sort(staff);
      int i;
      for (i = 0; i < staff.length; i++)
         System.out.println(staff[i]);
    }
}
```

Performance Issues - Choice of Algorithms, Measurement

```
import java.util.*;
import corejava.*;

public class Employee implements Comparable
{  public Employee(String n, double s, Day d)
   {  name = n;
      salary = s;
      hireDate = d; }

   public void raiseSalary(double byPercent)
   {  salary *= 1 + byPercent / 100; }
```

Performance Issues - Choice of Algorithms, Measurement

```
   public String getName() { return name; }

   public double getSalary() { return salary; }

   public String toString()
   {  return name + " " + salary + " " + hireYear();}
   public int hireYear()
   {  return hireDate.getYear();}

   public int compareTo(Object b)
   {  Employee eb = (Employee)b;
      if (salary < eb.salary) return -1;
      if (salary > eb.salary) return 1;
      return 0;
   }
   private String name;
   private double salary;
   private Day hireDate;
}
```

Performance Issues - Choice of Algorithms, Measurement

binarySearch method of java.util.Arrays

public static int binarySearch(Object[] a, Object key)

Searches the specified array for the specified object using the binary search algorithm. The array must be sorted into ascending order according to the natural ordering of its elements (as by Sort(Object[]), above) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements equal to the specified object, there is no guarantee which one will be found.

Parameters:

- **a - the array to be searched.**
- **key - the value to be searched for.**

Returns:

index of the search key, if it is contained in the list; otherwise, (-(insertion point) - 1). The insertion point is defined as the point at which the key would be inserted into the list: the index of the first element greater than the key, or list.size(), if all elements in the list are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

See Also:

Comparable, sort(Object[])