Unit 3 – Advanced Algorithms and Programming Principles

Recursively Defined Algorithms

Introduction

Thus far, we have investigated ***iterative*** algorithms, which are all based on ***looping***. Sometimes, however, it is very difficult or even impossible to describe algorithms in an iterative fashion. Fortunately, in many such cases, we can resort to a ***recursive*** description.

What are Recursive Algorithms?

To understand this, it's helpful to understand both non-recursively and recursively defined sequences.

Explicitly Defined Sequences (Functions)

e.g. 1, 2, 4, 8, 16, 32, 64, …

If we let t_n represent the n^{th} term of the sequence, then $t_1=1$, $t_2=2$, $t_3=4$, $t_4=8$ and so on. If we can find a formula that expresses t_n in terms of n, then we say that t_n is ***defined explicitly in terms of n***. Such definitions are ***not recursive***. The following is an ***explicit definition*** (non-recursive definition) of the sequence given above:

$$t_n = 2^{n-1}, n \in \mathbb{N}$$

Notice that you can determine any term in the sequence just by knowing the value of n.

Recursively Defined Sequences (Functions)

e.g. 1, 1, 2, 3, 5, 8, 13, 21, …

This is the famous ***Fibonacci Sequence***. (See http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/fibnat.html for a great site on how this sequence arises in nature.) This sequence is difficult to define explicitly. However, it is extremely easy to define ***recursively***. Such definitions of sequences define t_n in terms of *previous terms of the sequence*. The sequence above can be defined recursively as follows:

$$\begin{cases} t_1 = 1 \\ t_2 = 1 \\ t_n = t_{n-2} + t_{n-1}, n \geq 3 \end{cases}$$

Notice that in this case you need to know the values of the two previous terms of the sequence to calculate t_n. Knowing n is not enough!

Questions

1. Is it possible to define $t_n = 2^{n-1}$ recursively? How?
2. Use the Internet to discover whether it is possible to define the Fibonacci sequence ***explicitly***? If you find an explicit definition, compare it to the recursive definition given above. Which do you prefer? Why?

How this applies to Programming

We can illustrate the differences between non-recursive and recursive solutions by solving a few problems in both ways!

Problem 1

Given n, calculate $n!$ (This is read "n factorial" and means $n(n-1)(n-2)\cdots(2)(1)$. For example, $6! = 6\times5\times4\times3\times2\times1$.)

Problem 2

Given n, calculate the n^{th} term of the Fibonacci sequence.

Problem 3

Given an amount to be deposited at regular intervals (x), the annual rate of interest (r), the number of deposits per year (d) and the total number of deposits (n), calculate the future value. For this question, assume that the compounding frequency equals the payment frequency and that the deposits are made at the end of each payment period (ordinary annuity).

Pseudo-code for Iterative Solutions (i.e. Solutions Involving Loops)

Problem 1

```
set product = 1

for i = 2 to n
  set product = product*i
next i

return product
```

Problem 2

```
if n=1 or n=2
  return 1
else
  set term1 = 1
  set term2 = 1

  for i=3 to n
    set temp = term2
    set term2 = term1 + term2
    set term1 = temp
  next i

  return term2
end if
```

Problem 3

```
set perRate = r/d
set futureVal = 0

for i=0 to n-1
  futureVal =futureVal + x*(1+perRate)^i
next i

return futureVal
```

Pseudo-code for Recursive Solutions

Problem 1

The recursive function that solves this problem is called `"factorial"`

```
if n>1
  return n*factorial(n-1)
else
  return 1
end if
```

Problem 2

The recursive function that solves this problem is called `"fibonacci"`

```
if n>2
  return fibonacci(n-1)+fibonacci(n-2)
else
  return 1
end if
```

Problem 3

The recursive function that solves this problem is called `"fV"`

```
set perRate = r/d
if n>1
  return x +
      (1+perRate)*fV(n-1,x,d,r)
else
  return x
end if
```

Java Code for Recursive Solutions

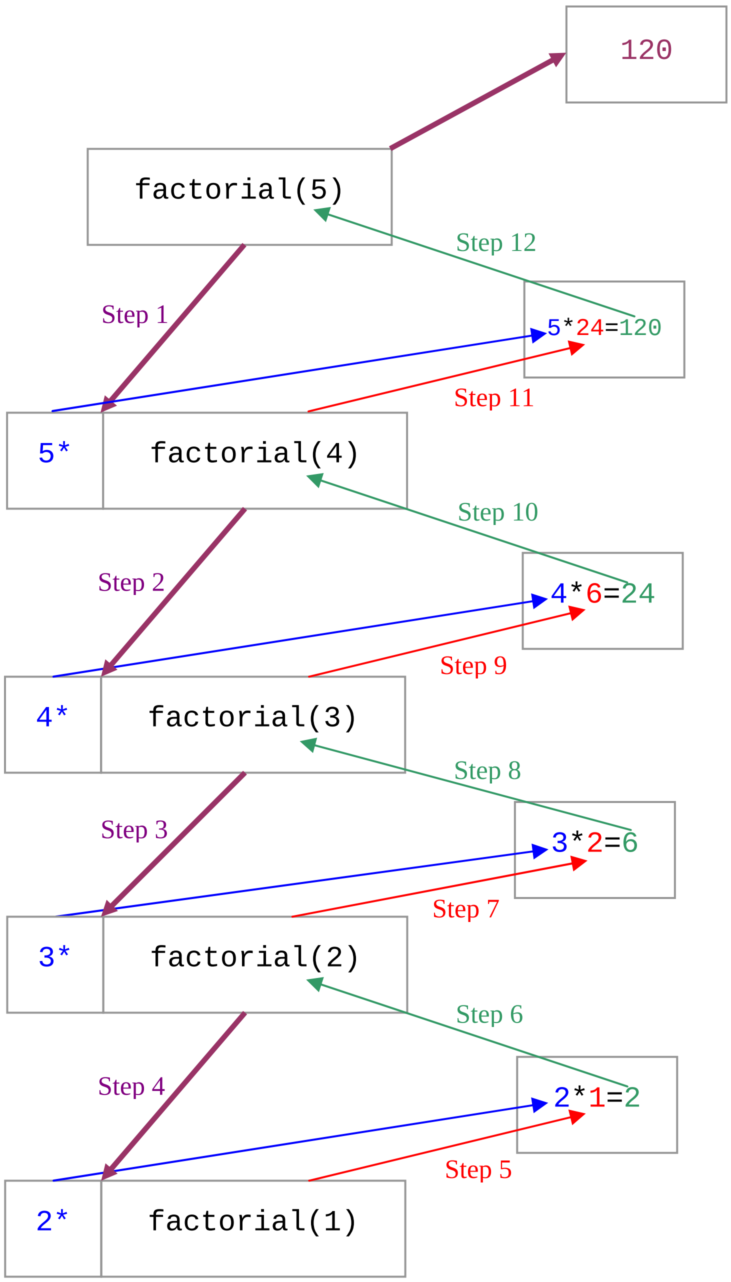
```java
public class RecursiveMethodExamples
{
   public static double factorial(int n)
   {
      if (n>1)
         return n*factorial(n-1);
      else if (n>=0)
         return 1;
      else
         return -1; //Error code: negative values cannot be passed
   }

   public static double fibonacci(int n)
   {
      if (n>2)
         return fibonacci(n-1)+fibonacci(n-2);
      else if (n>0)
         return 1;
      else
         return -1; //Error code: only positive values can be passed
   }

   //This method applies a simple recursive algorithm to calculate the future value
   //of an ordinary annuity (payment made at end of interval) given the amount
   //deposited at regular intervals.  It is assumed in this method that the
   //payment frequency is equal to the compounding frequency.
   public static double futureValue(double deposit, double annualRate,
                                    short depositsPerYear, int totalNumDeposits)
   {
      if (totalNumDeposits>1)
         return deposit+(1+annualRate/depositsPerYear)
                  *futureValue(deposit,annualRate,depositsPerYear,totalNumDeposits-1);
      else if (totalNumDeposits==1)
         return deposit;
      else
         return 0; //In case 0 or a negative integer is passed to "totalNumDeposits"
   }

}
```

Try it out yourselves!

1. Load **I:\Out\Nolfi\Ics4mo\Recursion\RecursiveMethodExamples\RecursiveMethodExamples.sln**.
2. Confirm that each recursive method returns correct answers.
3. Use breakpoints to follow the execution of each recursive method.
4. Write a short description of your observations in # 3. What seems to be happening?

Visualizing the Execution of a Recursive Method

The following diagram should help you to understand the execution of a recursive method.



Explanation

The "`factorial`" method from the previous page is used to illustrate the execution of a recursive method.

1. The execution begins when "5" is passed to the method. Since 5 > 1, the method returns 5*factorial(4). Since factorial(4) has not yet been evaluated, the first call to "factorial" is suspended until the next call (factorial(4)) returns a value.
2. The second call is factorial(4), which returns 4*factorial(3). Again, the execution of the factorial method, that is the second call to the factorial method, needs to be suspended until factorial(3) returns a value.

•
•
•

5. The calls continue in the manner described above until "1" is passed to the factorial method. This call does not need to be suspended because factorial(1) returns "1" immediately.
6. Now that factorial(1) has returned a value, the execution of factorial(2) can resume.

•
•
•

12. The returns continue in this ***cascading*** fashion until factorial(5) finally returns a value of 120.

Advantages and Disadvantages of Recursive Algorithm

Advantages

1. Code tends to be extremely short.
2. Debugging tends to be very easy because code is easy to understand.
3. Code corresponds very closely to mathematical formulation.

Disadvantages

1. Difficult to visualize execution of recursive calls.
2. Execution can be extremely slow due to large amount of overhead involved in processing method calls.
3. Extra memory must be allocated to store the "return points." The return points are stored using a data structure called a ***stack***. In many cases, the stack can grow exponentially, which can very quickly result in an "out of memory" condition.

Advantages and Disadvantages of Iterative Algorithm

Advantages

1. Execution speed tends to be fast.
2. No stack needed (no extra memory).

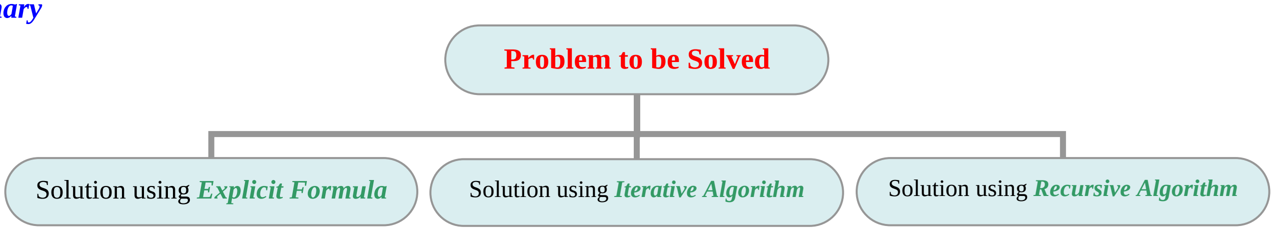
Disadvantages

1. Iterative implementation sometimes not at all straightforward (e.g. quickSort).
2. Code for iterative solution can be longer, more complex, more difficult to debug.

Questions

1. Rewrite each recursive method iteratively. Use the pseudo-code on page 2 as a guide.
2. Compare the execution speed of each recursive method with its iterative counterpart. What do you notice? Can you explain your observations?
3. As you may have learned in a previous math course, the future value of an annuity can be calculated using a simple formula (*i.e. no recursion or iteration is needed*). Rewrite "futureValue" in such a way that only a formula needs to be evaluated to compute the future value. (If you haven't learned about annuities in a previous course or you have forgotten what you have learned about them, try a search phrase such as "future value formula" to find the required formula.)
4. Is it possible to calculate the n^{th} term of the Fibonacci sequence using a formula? Is it possible to calculate $n!$ using a formula? (**Hint:** Try searching Google using the phrases "explicit formula fibonacci" and "explicit formula factorial.")

Summary



Series of Paragraphs

Write a series of paragraphs to explain the relative merits of the three types of solutions listed above. Discuss the advantages and disadvantages of each method as well as whether it is always possible to implement all three types of solutions.

Recursive Method Programming Exercises

1. Write a ***recursive method*** that can compute the sum of the integers from 1 to n, that is $t_n = 1 + 2 + \cdots + n = \sum_{i=1}^{n} i, n \geq 1$.
2. Write a ***recursive method*** that can compute the sum of the integers from m to n, that is

   $t_n = m + (m+1) + \cdots + n = \sum_{i=m}^{n} i, n \geq 0$.
3. Write a ***recursive method*** that can compute $t_n = 2^n$, where n is a non-negative integer.
4. Write a ***recursive method*** that can compute $t_n = x^n$, where x is any real number and n is any non-negative integer.

   (Note that the value of 0^0 has been the subject of some debate in the past. Nowadays, the value of 0^0 is generally taken to be 1 for most purposes. This definition of 0^0 does not create inconsistencies in the vast majority of cases.)
5. A *tribonacci number* is like a Fibonacci number. Instead of starting with two predetermined terms, however, the sequence starts with three and each term afterwards is the sum of the preceding three terms. The first few tribonacci numbers are: 1, 1, 2, 4, 7, 13, 24, 44, 81, 149, 274, 504, 927, 1705, 3136, 5768, 10609, 19513, 35890, 66012. Write a ***recursive method*** that can compute any term in the tribonacci sequence.
6. The first few terms of the ***Repfigit*** (**REP**etitive **FI**bonacci-like di**GIT**) numbers (or Keith numbers) are 14, 19, 28, 47, 61, 75, 197, 742, 1104, 1537, 2208, 2580 and 3684. This sequence is formed as shown in the following examples:

   14 $\rightarrow$ 1, 4, 5, 9, 14

   (The number "14" has ***two*** digits. If a "Fibonacci-like" sequence is formed starting with "1" and "4" as the initial terms, 14 will eventually be reached by adding the two previous terms to obtain the next term.)

   19 $\rightarrow$ 1, 9, 10, 19

   197 $\rightarrow$ 1, 9, 7, 17, 33, 57, 107, 197

   (Since 197 has 3 digits, the previous ***three*** terms must be added to form the next term. Similarly, if a term t_n in the repfigit sequence has m digits, then it must be obtained by using a "Fibonacci-like" sequence that begins with the m digits of t_n and in which the next term is obtained by adding the previous m terms of the sequence.)

   Write a ***recursive method*** that can compute any term in the ***Repfigit*** sequence. (It's best to split the solution to this problem into two or more methods.)

Quicksort: A Very Fast Recursive Sorting Algorithm

In grade 11 we explored various sorting algorithms, all of which were very easy to program but unfortunately, also very slow. Now that we understand recursion, we are in a position to explore a much more efficient sorting algorithm called "quick sort," which was developed in 1962 by C. A. R. Hoare. (Alright, try to restrain yourselves from making cheap jokes about Hugh Grant.)

Pseudo-Code for Quicksort

Suppose that the data are stored in an array with indices running from 0 to n.

1. Choose a "Pivot."

There are many methods that can be used to perform this step, however, none of them is optimal! There is no way of choosing the pivot in such a way that worst case performance can be avoided (see question 2 below).
e.g. choose "middle" element as pivot, pick a random pivot, choose either the leftmost or rightmost element, etc.

2. "Partition" the Array

Reorganize the array in such a way that the array is divided into three parts. The ***left partition*** consists of all elements ≤ the pivot and the ***right partition*** consists of all elements ≥ the pivot.

Left Partition all elements ≤ pivot	**Pivot**	**Right Partition** all elements ≥ pivot

3. Repeat Steps 1 and 2 on each Partition

If the left partition has 2 or more elements
    Repeat steps 1 and 2 on the left partition
Else
    Return (do nothing)
If the right partition has 2 or more elements
    Repeat steps 1 and 2 on the right partition
Else
    Return (do nothing)

Quicksort Example

In this example, the "middle" element is always chosen as the pivot. The pivot(s) is(are) always displayed in **red**.

First, choose the middle element (element 5) as the pivot.

	0	1	2	3	4	5	6	7	8	9	10
Step 1	17	7	0	6	31	**4**	13	27	18	21	13

Then reorganize the array in such a way that the elements of the left partition are ≤ the pivot and all the elements of the right partition are ≥ the pivot. Once this is done, the pivot lands exactly in its final resting place.

	0	1	2	3	4	5	6	7	8	9	10
Step 2	0	4	7	6	13	13	18	17	31	21	27

Since the left partition consists of only one element, it requires no further processing. The right partition has 9 elements, so the quicksort algorithm is applied to it.

	0	1	2	3	4	5	6	7	8	9	10
Step 1	0	4	7	6	13	13	**18**	17	31	21	27

Now reorganize the right partition.

	0	1	2	3	4	5	6	7	8	9	10
Step 2	0	4	7	6	13	13	17	18	31	21	27

Continuing in this manner, we obtain the following:

	0	1	2	3	4	5	6	7	8	9	10
Step 1	0	4	7	6	**13**	13	17	18	31	**21**	27

	0	1	2	3	4	5	6	7	8	9	10
Step 2	0	4	7	6	13	13	17	18	21	27	31

	0	1	2	3	4	5	6	7	8	9	10
Step 1	0	4	**7**	6	13	**13**	17	18	21	**27**	31

	0	1	2	3	4	5	6	7	8	9	10
Step 2	0	4	6	7	13	13	17	18	21	27	31

	0	1	2	3	4	5	6	7	8	9	10
	0	4	6	7	13	13	17	18	21	27	31

Questions

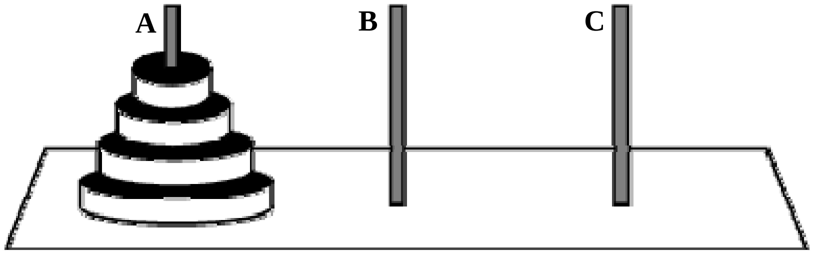
1. Using a *tree diagram*, explain why quicksort is usually so fast.
2. Again using a *tree diagram*, describe a scenario that would cause quicksort to perform very poorly. How could a hacker exploit this to launch an attack on a Web site?

```java
public final class SortingMethods
{
	/**
	 * The following three methods implement quickSort with median pivot. There are a variety of
	 * other ways of choosing the pivot, including a randomly chosen pivot.
	 */
	public static void quickSort(int[] a, int left, int right)
	{
		if (left<right)
		{
			int pivotIndex=partition(a, left, right);
			quickSort(a,left,pivotIndex-1); //Sort left partition
			quickSort(a,pivotIndex+1,right); //Sort right partition
		}

		else
			return; //Do nothing if partition contains fewer than 2 elements
	}

	private static void swap(int a[], int i, int j)
	{
		int temp=a[i];
		a[i]=a[j];
		a[j]=temp;
	}

	/**
	 * The "partition" method is used to reorganize the array into three parts, the left
	 * partition, the pivot and the right partition. The left partition contains all
	 * elements <= pivot and the right partition contains all elements >= pivot. The index
	 * of the pivot is normally chosen as the average (midpoint) of the indices "left"
	 * and "right." If the pivot is <= or >= both a[left] and a[right], then the "pivotIndex"
	 * is set either to "left" or "right," depending on whether a[left] or a[right]
	 * is "in the middle."  This method returns "pivotIndex," the index of the pivot.
	 */
	private static int partition(int a[], int left, int right)
	{
		int i, pivot, pivotIndex, mid=(left+right)/2;

		//Choose the index of the pivot.
		if (a[left]<=a[mid] && a[mid]<=a[right] || a[right]<=a[mid] && a[mid]<=a[left])
			pivotIndex=mid;
		else if (a[right]<=a[left] && a[left]<=a[mid] || a[mid]<=a[left] && a[left]<=a[right])
			pivotIndex=left;
		else
			pivotIndex=right;

		//Reorganize the array so that a[i] <= a[pivotIndex] if left <= i <= pivotIndex
		//and a[i] >= a[pivotIndex] if pivotIndex <= i <= right.
		swap(a,left,pivotIndex);//Place pivot at left end of array
		pivotIndex=left;
		pivot=a[pivotIndex];

		for (i=left+1; i<=right; i++)
		{
			if (a[i]<pivot)
				swap(a,++pivotIndex,i);
		}

		swap(a,left,pivotIndex); //Put pivot in its proper place

		return pivotIndex;
	}
}
```

Exercise

1. Use array diagrams to explain how the "partition" method reorganizes an array "a" in such a way that a[i] <= a[pivotIndex] for "i" ranging from "left" up to "pivotIndex-1" and a[i] >= a[pivotIndex] for "i" ranging from "pivotIndex+1" up to "right."

A Recursive Solution to the "Tower of Hanoi" Problem



The "Tower of Hanoi," commonly known as the "*Towers* of Hanoi," is a puzzle invented by E. Lucas in 1883. This puzzle involves three rods and a stack of *n* disks that is placed on one of the rods. The disks are initially arranged from largest on the bottom to smallest on top. The objective of the puzzle is to determine the minimum number of moves required to move the stack from one rod to another. Only one disk may be moved at a time from the top of any stack to the top of any other stack. Smaller disks may be placed on top of larger disks but larger disks ***cannot*** be placed on top of smaller disks.

Activity

1. Use the "Tower of Hanoi" Java applet on the "Puzzles" page of www.misternolfi.com (or any other Web-based version of this puzzle) to complete the following table:

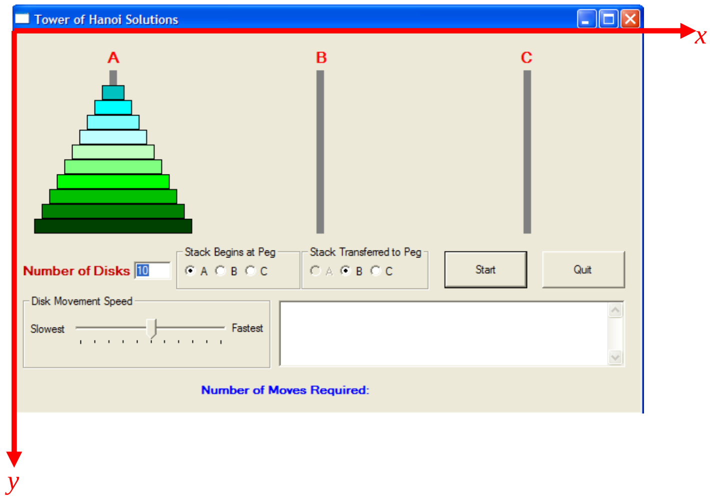
Number of Disks (n)	*Number of Moves Required to Solve*	*Record of Moves*
1	1	A → B (or A → C)
2	3	A → B, A → C, B → C (or A → C, A → B, C → B)
3		
4		
5		
6		
7		

2. Now observe your results very carefully. Can you see how a solution for *n*=1 can be used to build a solution for *n*=2? Can you see how a solution for *n*=2 can be used to build a solution for *n*=3? Can you see how a solution for *n*=3 can be used to build a solution for *n*=4? Can you see how a solution for *n*=*k* can be used to build a solution for *n*=*k*+1? Express your results using recursion.

3. Using your observations from question 2, try to write a Java method that can solve the tower of Hanoi problem for a stack of *n* disks.

MULTI-DIMENSIONAL ARRAYS

A Solution to the "Tower of Hanoi" Problem that uses two Two-Dimensional Arrays



You can find the "Tower of Hanoi Solutions" program in **I:\Out\Nolfi\Ics4m0\TowerOfHanoi**.

Once you load this program into J++, read through the code carefully. You should notice the following:

1. Many arrays are used, including two ***two-dimensional arrays***.
2. The code in the "FormTower" class is neatly divided into ***two sections***, one for ***data fields*** and another for ***methods***. The data field section is further subdivided into one section for ***global constants*** and another for ***global variables and objects***. Furthermore, the method portion consists of three different subsections (one for ***constructor methods***, another for ***event handling methods*** and yet another for ***all other methods***).
3. Most of the concepts learned in this course can be found in this program. Therefore, the "Tower of Hanoi" program can ***serve as an excellent tool for studying for the final exam!***

The ***two-dimensional array*** declared below, known as a 3×10 (read "3 by 10") array because it has 3 rows and 10 columns, stores integers representing the disks present in each stack. The disks are numbered from 0 (largest disk) to 9 (smallest disk). A value of –1 (constant "NO_DISK") is assigned if a disk is absent. Each row of the matrix represents a stack on one of the pegs.

```
private int[][] stack = new int[3][10];
```

For example, the initial arrangement of the disks on "peg A" would be stored as follows in the "`stack`" array:

Column Indices (0 to 9)

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1
2	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

Row Indices (0 to 2)

The following ***two-dimensional constant array*** stores the (left, top) co-ordinates of the disks when they rest on "peg A." These values are used when the disks are moved from one stack to another. When a disk is moved, its new position is based on the values in this array.

```
private final static int[][] STACK_A_DISK_COORD =
   { {20,200},{28,184},{36,168},{44,152},{52,136},
       {60,120},{68,104},{76,88},{84,72},{92,56} };
```

Column Indices (0 to 1)

Row Indices (0 to 9)

	0	1
0	20	200
1	28	184
2	36	168
3	44	152
4	52	136
5	60	120
6	68	104
7	76	88
8	84	72
9	92	56

This array is called a 10×2 (read "10 by 2") array because it has 3 rows and 10 columns.

Multi-Dimensional Array Exercises

1. Write a method that can calculate the sum of any row, column or diagonal of any ***n×n two-dimensional array***. (Note that two-dimensional arrays are also called ***matrices***.)

2. Write a method that can calculate the sum of any, row, column, diagonal or layer of an ***n×n×n three-dimensional array***. (A description of how three-dimensional arrays can be visualized will be given in class.)

Analyzing the Efficiency of Algorithms: Case Study–Searching and Sorting

Introduction

Often, many ***different*** algorithms can be used to solve a particular problem. Therefore, to select the best algorithm for a given situation, it is important to be able to measure precisely the efficiency of algorithms. Computer scientists use ***complexity theory*** to perform such analyses. Complexity theory helps them to group algorithms into various ***complexity classes***. The problems of searching and sorting, the most widely studied problems in computer science, will be used to illustrate the main ideas of complexity theory.

Precise Statement of the Problems of Searching and Sorting

Given *n* records stored in an array of *n* elements, how can the records be sorted (i.e. arranged in "alphabetic" order) efficiently? Once the records are sorted, how can a certain record be located in the least time possible?

Exactly what do we mean by Efficiency?

Space and time are the most important quantities to consider in the analysis of an algorithm.

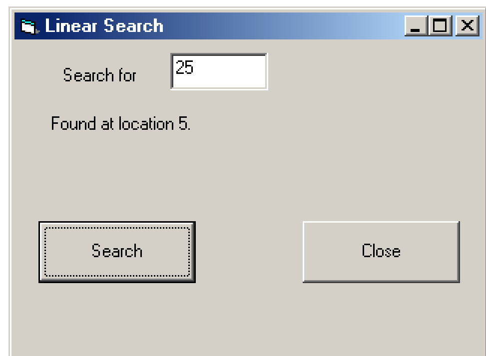
- **Space:** The amount of memory required during the execution of a program.
- **Time:** The amount of time required for a program to complete a certain task.

These two quantities ***tend to be inversely related.*** Fast programs ***tend to use a lot of memory*** while programs that use memory efficiently ***tend to be slow***.

Example – Linear Search

Consider the following Visual Basic program that uses a function procedure to perform a ***linear search*** (sequential search) of an array of *n* elements.

```
Dim SomeArray(1 To 20) As Integer

Private Sub Form_Load()
    Dim I As Integer
    'Store random integers between 1 and 100 in the array.
    For I = 1 To 20
        SomeArray(I) = Int(Rnd*100+1)
    Next I
End Sub

Private Sub cmdClose_Click()
    End
End Sub

Private Sub cmdSearch_Click()
    Dim Location As Integer
    Location = LinearSearch(SomeArray(), Val(txtSearchFor.Text), 20)
    If Location <> 0 Then
        lblFoundAt.Caption = "Found at location " & CStr(Location)&"."
    Else
        lblFoundAt.Caption = "Not found"
    End If
End Sub

' This function performs a linear search of the array passed to the
' array parameter "A" for the value passed to the parameter "Item."
' If the item is found, its location within the array is returned.
' Otherwise, zero is returned.  It is assumed in this function that
' the array is declared with indices running from 1 to "N."

Function LinearSearch(A() As Integer, ByVal Item As Integer, _
                                      ByVal N As Integer) As Integer
    Dim I As Integer
    For I = 1 To N
        If A(I) = Item Then
            LinearSearch = I
            Exit Function
        End If
    Next I
    LinearSearch = 0 'Return 0 if required value was not found
End Function
```



Let $f(n)$ represent the ***growth function*** of the "LinearSearch" VB function procedure shown at the left. That is, $f(n)$ represents the maximum number of statements that need to be executed by "LinearSearch." If we exclude the first and last lines of the function, it's easy to verify that (*n* represents the number of elements in the array being searched)

$$f(n) = 4n + 2.$$

In this function (which represents the performance of the given linear search algorithm), as the data size *n* increases, the "4*n*" term will dominate. Therefore, we say that this is an $O(n)$ algorithm (read "order *n*" or "big O of *n*").

To determine the *O* value of an algorithm,

1. Ignore the constants since we are only interested in the growth characteristic of the algorithm.
2. Choose the fastest growing term since it will account for the majority of the growth.

Binary Search

While linear search is easy to program and is reasonably fast when used to search small arrays, it is excruciatingly slow if used to search an array with a large number of elements. For instance, consider an array of one million strings. ***On average***, the linear search requires 500000 comparisons before a required value is found. ***In the worst case***, one million comparisons are needed. Obviously, this method wastes a great deal of CPU time. Fortunately, there are much faster algorithms that can be used to search very large data sets. ***Binary search***, for instance, can find any value in an array of 1000000 elements using ***10 or fewer comparisons***. In order for binary search to work, however, the array must be ***sorted***.

Example

Suppose that the following sorted array is being searched for the value "80."

Index	Data
1	6
2	14
3	14
4	21
5	29
6	36
7	42
8	43
9	56
10	56
11	63
12	69
13	71
14	76
15	77
16	80
17	85
18	89
19	97
20	100

Step 1

The search begins at the ***middle of the array***. The value being sought is "80" and the value stored at the middle of the array is "56." Since 80 > 56, the first half of the list is ignored and the search continues at the middle of the second half of the array.

Index	Data
1	6
2	14
3	14
4	21
5	29
6	36
7	42
8	43
9	56
10	56
11	63
12	69
13	71
14	76
15	77
16	80
17	85
18	89
19	97
20	100

Step 2

The search continues at the ***middle of the second half of the array***, where "77" is stored. Since 80 > 77, the first half of the second half of the array is ignored and the search continues at the lowest quarter of the array.

Index	Data
1	6
2	14
3	14
4	21
5	29
6	36
7	42
8	43
9	56
10	56
11	63
12	69
13	71
14	76
15	77
16	80
17	85
18	89
19	97
20	100

Step 3

The search continues at the ***middle of the lowest quarter of the array***, where "89" is stored. Since 80 < 89, the second half of the lowest quarter of the array is ignored and the search continues.

Index	Data
1	6
2	14
3	14
4	21
5	29
6	36
7	42
8	43
9	56
10	56
11	63
12	69
13	71
14	76
15	77
16	80
17	85
18	89
19	97
20	100

Step 4

The search ends at element 16 of the array, where the required value is found. Notice that half of the elements remaining are eliminated after each comparison, which means that no more than five comparisons are required to search 20 elements.

Important Programming Exercises

1. Translate the VB “LinearSearch” function procedure on page into a Java method. (Please remember that in Java, array indices always run from 0 to *arraySize-1*. Therefore, you will need to modify the strategy used in the VB function procedure shown on the previous page.)
2. Write two different Java versions of binary search, one that works with numeric data and another that works with strings.

Formal Definition of Complexity Classes

Let $f(n)$ represent the number of statements that need to be performed by an algorithm to complete a task given a data size of n. Then $f(n)$ is said to belong to the ***complexity class O(g(n))*** (read “order g of n” or “big O of g of n”) if there exist positive constants $k \in \mathbb{N}$ and $c \in \mathbb{R}$ such that for all $n \geq k$,

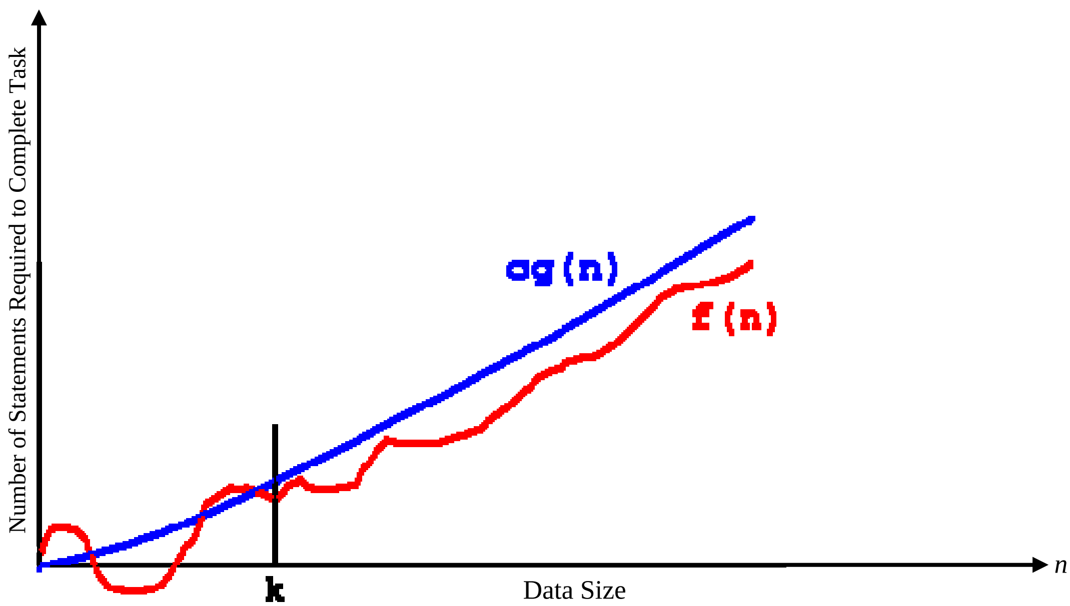
$$f(n) \leq cg(n).$$

We can state this definition more concisely symbolically:

$$f(n) \in O(\ g(n)\ ) \text{ if } \exists\ k \in \mathbb{N} \text{ and } c \in \mathbb{R} \ni \forall\ n \geq k,\ f(n) \leq cg(n).$$

Intuitive Translation of this Definition

When the data size n is large enough, then there will be a function $cg(n)$ which is larger than $f(n)$ ($cg(n)$ is an upper bound for $f(n)$). We choose to use $cg(n)$ because it is ***a simpler, more well behaved function than f(n)***. This makes it much easier to analyze than $f(n)$. In addition, when we choose $g(n)$ we ignore all terms except for the ***fastest growing term*** (dominant term) because it accounts for most of the growth. In addition, we can also ignore all the statements except for the dominant operation. For example, in any sorting algorithm, ***comparisons are performed more often than any other operation***. Therefore, in order to determine the efficiency of any sorting algorithm, it is enough to count the number of ***comparisons***.



The “smallest” function which is an upper limit is chosen for $g(n)$. For example, $3n+2 \in O(n^2)$ is true, but there is a “smaller” big-O value which is a better fit, $O(n)$.

True	Better	Intuitive Meaning
$n^2 + 7 \in O(n^3)$	$n^2 + 7 \in O(n^2)$	The growth rate of $n^2 + 7$ is no larger than that of cn^2 for some $c \in \mathbb{R}$ and for large enough values of n.
$n^2 + n^3 \in O(2^n)$	$n^2 + n^3 \in O(n^3)$	The growth rate of $n^2 + n^3$ is no larger than that of cn^3 for some $c \in \mathbb{R}$ and for large enough values of n.
$10^n + n^2 + n^3 \in O(n^n)$	$10^n + n^2 + n^3 \in O(2^n)$	The growth rate of $10^n + n^2 + n^3$ is no larger than that of $c(2^n)$ for some $c \in \mathbb{R}$ and for large enough values of n.

Listed below are some common complexity classes and some well known algorithms.

Some Common Complexity Classes		
Complexity Class Name	**Complexity**	**Common Algorithms with The Given Complexity**
Constant	$O(1)$	Any program that executes in a constant time regardless of input. Very few practical algorithms belong to this class.
Logarithmic	$O(\log n)$	Binary Search of a Sorted array, Search of an Approximately Balanced Binary Tree
Linear	$O(n)$	Linear Search
Quadratic	$O(n^2)$	Bubble Sort, Selection Sort, Insertion Sort
Polynomial	$O(n^k)$, $k \in \mathbb{N}$	Multiplying Two $n \times n$ Matrices ($O(n^3)$).
Exponential	$O(2^n)$	Travelling Salesperson Program

Using Sorting Algorithms to Gain a Different Perspective on Complexity Classes

Measuring the efficiency of any algorithm is a matter of ***counting*** the number of statements that need to be executed. Usually, a single type of statement tends to require the bulk of the processing time. For instance, sorting methods spend most of their time ***comparing*** and ***swapping*** (exchanging) data. Since a swap can only occur after a comparison, the number of swaps will always be less than or equal to the number of comparisons. Thus, the ***dominant operation*** for sorting is the comparing of data; to study the performance of sorting algorithms, it is only necessary to count the number of comparisons. (The above paragraph should help you to understand why we ignore the constants and all terms except for the dominant one.)

Exercises

Comparison of Some Well Known Sorting Algorithms						
Algorithm	**Best Case**		**Average Case**		**Worst-Case**	
	Comparisons	**Exchanges**	**Comparisons**	**Exchanges**	**Comparisons**	**Exchanges**
Bubble Sort (most efficient version)	$O(n)$	0	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	0	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	0	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n)$
Shell Sort	$O(n^{1.25})$	0	$O(n^{1.25})$	$O(n^{1.25})$	$O(n^{1.5})$	$O(n^{1.5})$
Quicksort	?	?	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)$

1. Use a graphing calculator (or a graphing program) to sketch the graphs of $y = x$, $y = x^2$, $y = x^{1.25}$, $y = \log x$, $y = 2^x$ and $y = x \log x$ on a single set of axes. Use the graphs to rank the complexity classes $O(n)$, $O(n^2)$, $O(n^{1.25})$, $O(\log n)$, $O(2^n)$ and $O(n \log n)$ from most efficient to least efficient. Then use the graphs to rank the sorting algorithms shown above from fastest to slowest (average case).
Suggestion for Graphing Calculator Window: Set XMin=0, XMax=100, XScl=10, YMin=0, YMax=300, YScl=25

2. Explain why it is not possible to choose a sorting algorithm that is best in all cases.

3. Visit the Web site http://www.cs.smith.edu/~thiebaut/java/sort/demo.html and try out the sorting algorithm demo. Use it to complete the following table (use the phrases "random array," "array sorted in increasing order" and "array sorted in decreasing order."

Algorithm	What Produces Best Case Behaviour	What Produces Average Case Behaviour	What Produces Worst Case Behaviour
Standard Quicksort	?		
Shell Sort			
Insertion Sort			
Selection Sort			
Quicksort with Random Pivot	?		
Quicksort with Median Pivot	?		

4. Explain why you should never use an $O(2^n)$ algorithm (exponential time algorithm).