Part A: Stack vs. Heap

Before we get into discussing various computer science algorithms such as recursion, we need to understand more about how a computer organizes memory for a running program. When a program loads into memory, it is organized into various segments. Two of these segments are called the **stack** and the **heap**. Both of these segments are areas in memory responsible for data storage of a program.

Stack

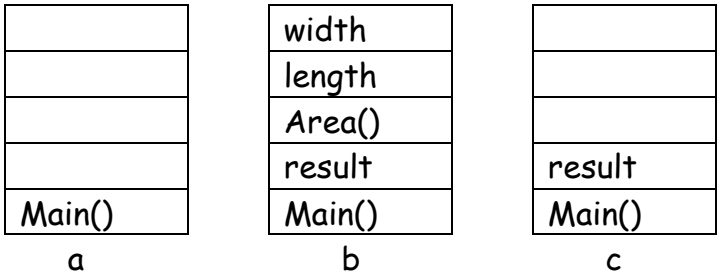
In general terms, a stack is a data structure that stores items in a **last-in**, **first-out** manner (**LIFO**). In terms of memory for a running program, a stack is a chunk of memory allocated to a running program that contains both **local variables** and the **call stack** in a LIFO manner. Local variables are variables declared inside a method and can either be a **value** type (primitive type) or a **reference** type (object). Value type variables are stored directly on the stack and are very quick to access. Reference types are stored on the heap, which is described below.

When a method is called in a program, it is also added to the top of the stack. When the method returns back to the user, it is removed from the stack. Execution can continue with the next method that is on the stack. When a variable is created within a method, it is added to the stack. The variable is popped off the stack when the method has completed execution.

Heap

The **heap** is a data structure in memory created to store reference types (objects). When an object is created (using the **new** keyword in C#), the object is stored on the heap as a block of data containing its data members. The address of the object is stored as a reference variable typically on the stack. This means that the value stored in the stack is the memory address number that points to the object structure on the heap. Accessing data on the heap is not as fast as accessing data on the stack but variables on the heap will exist for the entire duration of the program.

The following illustrates the stack for a C# program. In the program the Main() method calls a method named Area(). Area() creates two variables named length and width and returns the result.



Part B: Recursion

Up to now we have not called a method within another method. It is also possible for a method to call itself. When a method calls itself, this is called **recursion**. Clearly you must include some logic in a recursive method so that it will eventually stop calling itself. When a recursive method is called to solve a problem, the method actually is capable of solving only the simplest case(s), or **base case(s)**. If the method is called with a base case, the method returns a result. If the method is called with a more complex problem, the method typically divides the problem into two conceptual pieces, a piece that the method knows how to do and a piece that the method does not know how to do. To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or a smaller version of it. Since this new problem looks like the original problem, the method calls a fresh copy of itself to work on the smaller problem. This is referred to as a **recursive call** and is also called the **recursion step**. The recursion step normally includes a return statement, because its result will be combined with the portion of the problem the method knew how to solve to form a result that will be passed back to the original caller.

The recursion step executes while the original call to the method is still active (i.e., while it has not finished executing). The recursion step can result in many more recursive calls as the method divides each new sub-problem into two conceptual pieces. For the recursion to eventually terminate, each time the method calls itself with a simpler version of the original problem, the sequence of smaller and smaller problems must converge on a base case. At that point, the method recognizes the base case and returns a result to the previous copy of the method. A sequence of returns ensues until the original method call returns the final result to the caller.

To better understand recursion, we will look at creating a program that displays the factorial (n!) of a positive integer n. We can break down the factorial calculation as:

n! = n * (n-1) * (n-2) * (n-3) * … 1 (1! and 0! both equal 1)

From the above we can write n! as:

n! = n * (n-1)!

Our base case will return 1 when n <= 1 and our recursion step is n*(n-1)!. Notice that the recursion step will converge to the base case (eventual n-1 will equal 1). Our code will look something like the following.

```
public class Factorial
{

 public static void main(String[] args)
 {
   long value = calcFact(5);
   Console.WriteLine(value);
 }

 public static long calcFact(long n)
 {
   if (n<=1)
   {
     return 1;
   }
   else
   {
     return n * calcFact(n-1);
   }
 }
}
```

If we were trying to find 4!, how would the program process this?

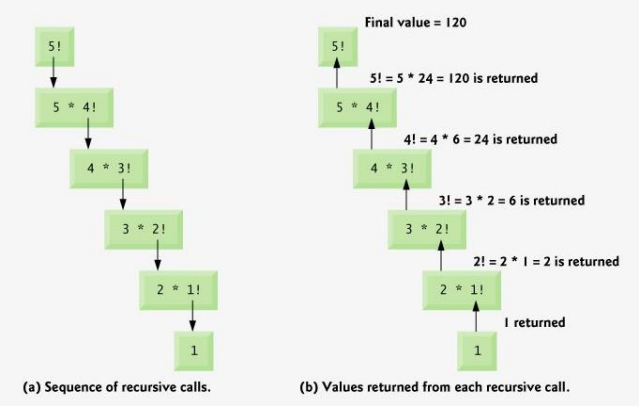
calcFact(5) : not the base case so 5 * calcFact(4) is returned.
calcFact(4) cannot be resolved, a call to calcFact(4) is made.

calcFact(4) : not the base case so 4 * calcFact(3) is returned.
calcFact(3) cannot be resolved, a call to calcFact(3) is made.

calcFact(3): not the base case so 3 * calcFact(2) is returned.
calcFact(2) cannot be resolved, a call to calcFact(2) is made.

calcFact(2) : not the base case so 2 * calcFact(1) is returned.
calcFact(1) cannot be resolved, a call to calcFact(2) is made.

calcFact(1): is the base case so 1 is returned. Now that we have our base case, the result is bubbled back up. This is illustrated in the following diagram.



Lab

- create a program that uses recursion to display the Fibonacci series of a given number.
- the series can be defined as

$$F(n) = \begin{cases} 0 \text{ if } n=0 \\ 1 \text{ if } n=1 \\ F(n-1) + F(n-2) \text{ if } n>1 \end{cases}$$

Recursion vs. Iteration

Both iteration and recursion are based on a control statement: Iteration uses a repetition statement (e.g., for, while or do...while), whereas recursion uses a selection statement (e.g., if, if...else or switch). Both iteration and recursion involve repetition. Iteration explicitly uses a repetition statement, whereas recursion achieves repetition through repeated method calls. Iteration and recursion each involve a termination test. Iteration terminates when the loop condition fails, whereas recursion terminates when a base case is reached. Iteration with counter-controlled repetition and recursion each gradually approach termination. Iteration keeps modifying a counter until the counter assumes a value that makes the loop condition fail, whereas recursion keeps producing simpler versions of the original problem until the base case is reached. Both iteration and recursion can occur infinitely. An infinite loop occurs with iteration if the loop condition never becomes false, whereas **infinite recursion** occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case, or if the base case is not tested.

Basically, any method written using iteration can be written using recursion and any method written using recursion can be written using iteration. So which programming technique should be used? That may depend on the problem. Some problems can be naturally recursive so recursion should be used or a problem may be simpler to implement using recursion than using iteration. However, one should understand that there is more overhead using recursion. Each recursive call causes another copy of the method and all its method variables to be created. This repetition can be expensive in terms of both processor time and memory space. Doing this can consume considerable memory space and effect algorithm performance.

Part C: Sorting

Sorting data (i.e., placing the data into some particular order, such as ascending or descending) is one of the most important computing applications. A bank sorts all checks by account number so that it can prepare individual bank statements at the end of each month. Telephone companies sort their lists of accounts by last name and, further, by first name to make it easy to find phone numbers. Virtually every organization must sort some data, and often, massive amounts of it. Sorting data is an intriguing, computer-intensive problem that has attracted intense research efforts.

An important item to understand about sorting is that the end result, the sorted array, will be the same no matter which algorithm you use to sort the array. The choice of algorithm affects only the run time and memory use of the program. We will investigate a number of sorting algorithms. The first three algorithms; **bubble sort**, **selection sort** and **insertion sort**, are simple algorithms to program, but are not very efficient. The sort algorithm, **quick sort**, is a much faster algorithm than **selection sort** and **insertion sort**, but is a more complex algorithm. We will focus on sorting arrays of primitive data types, namely integers. It is possible to sort arrays of objects of classes as well.

Big-O Notation

Big-O (pronounced Big-Oh) notation is a mathematical representation to estimate the CPU or memory resources of an algorithm. This helps a programmer to analyse the relationship between the number of data elements and resource usage by using a simple formula approximation. This is useful to predict how an algorithm may perform on large data sets. Many programmers may test their algorithms with small amounts of data and are surprised by the poor performance they achieve when moved to a large data set.

Big-O notation uses a polynomial express with N as the variable that represents the number of data elements in an algorithm. For example, an algorithm may have $O(N^3)$. With this expression, a programmer can analyse the time it will take to process N items and determine if this will be acceptable or not.

The following chart illustrates how many operations will be performed for various values of N.

	constant	*logarithmic*	*linear*		*quadratic*	*cubic*
N	**O(1)**	**O(log N)**	**O(N)**	**O(N log N)**	**$O(N^2)$**	**$O(N^3)$**
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4,096
1,024	1	10	1,024	10,240	1,048,576	1,073,741,824
1,048,576	1	20	1,048,576	20,971,520	10^{12}	10^{16}

Is it really necessary to evaluate an algorithm this way? For instance, you have a 1 Mega Pixel digital camera (and we know that this isn't a very good camera). If you created an algorithm that processed the pixels to be displayed on the camera's LCD display with a Big-O notation of $O(N^2)$, would this be acceptable? If it took 1 microsecond (10^{-6} seconds) to process one pixel, it would take this algorithm over 1 week (approx 11 days) to display the image. In this case it would be time to redesign your algorithm.

Bubble Sort

Bubble sort is probably the most simple and most inefficient sorting algorithm. It starts by comparing the first two elements. If the second element is smaller than the first element, it swaps them. The sort then proceeds to compare the second element to the third element and swaps them if required. This continues for the full length of the array and repeats until all the elements are in order.

As an example, consider the array

34 56 4 10 77 51 93 30 5 52

A program that implements bubble will begin by comparing the first and second elements. Since 34 is less than 56 there will be no swap. The second element and third element compared next. The element 4 is less than 56 so a swap occurs resulting in

34 4 56 10 77 51 93 30 5 52

The sort will then compare the third element and the fourth element. In this case 10 is less than 56 so a swap occurs result in

34 4 10 56 77 51 93 30 5 52

The sort will continue swapping values until it reaches the last element which will store the largest value. The sort will iterate through the elements again. To improve efficiency, the algorithm should take into account that it does not need to check the last element from the previous iteration.

The bubble sort has a complexity of $O(N^2)$ where the O is referred to as Big O notation (pronounced Big-Oh). Big O notation is often used in computing to classify algorithms according to the upper bounds of their total running time.

Pseudo Code

```
set n equal to the array length
for i= n-1 to 0
   for j=1 to i
      if a[j] > a[j+1]
         swap(a[j] , a[j+1])
      end if
    end for
end for
```

Selection Sort

Selection sort is a simple, but inefficient, sorting algorithm. The first iteration of the algorithm selects the smallest element in the array and swaps it with the first element. The second iteration selects the second-smallest item (which is the smallest item of the remaining elements) and swaps it with the second element. The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last index, leaving the largest element in the last index.

As an example, consider the array

34 56 4 10 77 51 93 30 5 52

A program that implements selection sort first determines the smallest element (4) of this array which is contained in index 2. The program swaps 4 with 34, resulting in

4 56 34 10 77 51 93 30 5 52

The program then determines the smallest value of the remaining elements (all elements except 4), which is 5, contained in index 8. The program swaps 5 with 56, resulting in

4 5 34 10 77 51 93 30 56 52

On the third iteration, the program determines the next smallest value (10) and swaps it with 34.

4 5 10 34 77 51 93 30 56 52

The process continues until the array is fully sorted.

4 5 10 30 34 51 52 56 77 93

Note that after the first iteration, the smallest element is in the first position. After the second iteration, the two smallest elements are in order in the first two positions. After the third iteration, the three smallest elements are in order in the first three positions.

Similar to the bubble sort, the selection sort is a $O(n^2)$ algorithm. However, the selection sort is typically more efficient as it will execute less swaps than the bubble sort.

Pseudo Code

```
for i=0 to n-1
  set min to a[i]
   for j=i+1 to n
      if a[j]<min
         set min to a[j]
         set min index to j
      end if
   end for
   swap min element with a[i] element
end for
```

Insertion Sort

Insertion sort is another simple, but inefficient, sorting algorithm but it has been proven to outperform both bubble and selection sorts. The insertion sort is also $O(n^2)$. The first iteration of this algorithm takes the second element in the array and, if it is less than the first element, swaps it with the first element. The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order. At the n^{th} iteration of this algorithm, the first n elements in the original array will be sorted.

Consider as an example the following array.

34 56 4 10 77 51 93 30 5 52

A program that implements the insertion sort algorithm will first look at the first two elements of the array, 34 and 56. These two elements are already in order, so the program continues (if they were out of order, the program would swap them).

In the next iteration, the program looks at the third value, 4. This value is less than 56, so the program stores 4 in a temporary variable and moves 56 one element to the right. The program then checks and determines that 4 is less than 34, so it moves 34 one element to the right. The program has now reached the beginning of the array, so it places 4 in the 0^{th} element. The array now is

4 34 56 10 77 51 93 30 5 52

In the next iteration, the program stores the value 10 in a temporary variable. Then the program compares 10 to 56 and moves 56 one element to the right because it is larger than 10. The program then compares 10 to 34, moving 34 right one element. When the program compares 10 to 4, it observes that 10 is larger than 4 and places 10 in element 1. The array now is

4 10 34 56 77 51 93 30 5 52

Pseudo Code

```
for j=1 to n
  set i to index on left
  set temp to a[j]
   loop while i >=0 and if temp < a[i]
      swap(a[i+1], a[i])
      decrement i
   end loop
   set a[i+1] to temp
end for
```

Quick Sort

Quick sort is an efficient sorting algorithm, but it is conceptually more complex than the sorting algorithms we have looked at until now. Quick sort is of O(n logn) order. The quick sort is a divide-and-conquer method for sorting. It works by **partitioning** an array into two parts, then sorting the parts independently.

For this strategy to be effective, the **partition** phase must ensure that all the elements in lower part are less than all those in the upper part. To do this, we choose a **pivot** element and arrange to have all the items in the lower part to be less than the pivot and all those in the upper part greater than the pivot element. We typically do not know anything about the items to be sorted so we can choose one in the middle. In the portioning phase we will exchange the first element in the lower part that is not less than the pivot with the first element in the upper part that is not greater than the pivot. After the partition sequence, recursion is used to sort each partition. The recursion will end when a part consists of one element only.

Code

```
void quicksort (int[] a, int lo, int hi)
{
//  lo is the lower index, hi is the upper index
//  of the region of array a that is to be sorted
    int i=lo, j=hi, h;
    int x=a[(lo+hi)/2];

    //  partition
    do
    {
        while (a[i]<x) i++;
        while (a[j]>x) j--;
        if (i<=j)
        {
            h=a[i]; a[i]=a[j]; a[j]=h;
            i++; j--;
        }
    } while (i<=j);

    //  recursion
    if (lo<j) quicksort(a, lo, j);
    if (i<hi) quicksort(a, i, hi);
}
```

Big-O Revisited

When analysing an algorithm using Big-O notation, the expression usually refers to the average case using random data. Depending on how the data is provided, there may be a best case and a worst case scenario in addition to the average case which can have an effect on some algorithms. It will be up to the programmer to analyse how the data may be provided and determine which algorithm would be most suitable. The following table compares the performance of the algorithms we have just discussed.

Type of Sort	***Best***	***Worst***	***Average***	***Comments***
BubbleSort	O(N)	$O(N^2)$	$O(N^2)$	Not a good sort, except with ideal data.
Selection sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	Perhaps best of $O(N^2)$ sorts
QuickSort	O(N log N)	$O(N^2)$	O(N log N)	Good, but it worst case is $O(N^2)$
HeapSort	O(N log N)	O(N log N)	O(N log N)	Typically slower than QuickSort, but worst case is much better.

Part D: Searching

Looking up a phone number, accessing a Web site and checking the definition of a word in a dictionary all involve searching large amounts of data. The next two sections discuss two common search algorithms, one that is easy to program yet relatively inefficient and one that is relatively efficient but more complex and difficult to program.

Linear Search

The **linear search** algorithm searches each element in an array sequentially. If the search key does not match an element in the array, the algorithm tests each element, and when the end of the array is reached, informs the user that the search key is not present. If the search key is in the array, the algorithm tests each element until it finds one that matches the search key and returns the index of that element.

As an example, consider an array containing the following values and a program that is searching for 51.

34 56 2 10 77 51 93 30 5 52

Using the linear search algorithm, the program first checks whether 34 matches the search key. It does not, so the algorithm checks whether 56 matches the search key. The program continues moving through the array sequentially, testing 2, then 10, then 77. When the program tests 51, which matches the search key, the program returns the index 5, which is the location of 51 in the array. If, after checking every array element, the program determines that the search key does not match any element in the array, the program returns a sentinel value (for example; -1).

Efficiency of a Linear Search

The worst case in this algorithm is that every element must be checked to determine whether the search item exists in the array. If the size of the array is doubled, the number of comparisons that the algorithm must perform is also doubled. Note that a linear search can provide outstanding performance if the element matching the search key happens to be at or near the front of the array. However, we seek algorithms that perform well, across all searches, including those where the element matching the search key is near the end of the array.

Binary Search

The **binary search** algorithm is more efficient than the linear search algorithm, but it requires that the array be sorted. The first iteration of this algorithm tests the middle element in the array. If this matches the search key, the algorithm ends. Assuming the array is sorted in ascending order, then if the search key is less than the middle element, the search key cannot match any element in the second half of the array and the algorithm continues with only the first half of the array (i.e., the first element up to, but not including the middle element). If the search key is greater than the middle element, the search key cannot match any element in the first half of the array and the algorithm continues with only the second half of the array (i.e., the element after the middle element through the last element). Each iteration tests the middle value of the remaining portion of the array. If the search key does not match the element, the algorithm eliminates half of the remaining elements. The algorithm ends either by finding an element that matches the search key or reducing the sub-array to zero size.

As an example consider the sorted 15-element array and a search key of 65.

2 3 5 10 27 30 34 51 56 65 77 81 82 93 99

A program implementing the binary search algorithm would first check whether 51 is the search key (because 51 is the middle element of the array). The search key (65) is larger than 51, so 51 is discarded along with the first half of the array (all elements smaller than 51). Next, the algorithm checks whether 81 (the middle element of the remainder of the array) matches the search key. The search key (65) is smaller than 81, so 81 is discarded along with the elements larger than 81. After just two tests, the algorithm has narrowed the number of values to check to three (56, 65 and 77). The algorithm then checks 65 (which indeed matches the search key), and returns the index of the array element containing 65. This algorithm required just three comparisons to determine whether the search key matched an element of the array. Using a linear search algorithm would have required 10 comparisons. Note: In this example, we have chosen to use an array with 15 elements so that there will always be an obvious middle element in the array. With an even number of elements, the middle of the array lies between two elements. We implement the algorithm to choose the lower of those two elements.

Code (sort of)

```
boolean binarySearch(int key){
   int first,last,mid = 0;
   boolean found = false;
   first = 0;
   last = arraysize-1;

   while((!found) && (first <= last)){
      mid = first + (last - first)/2;
      if(array[mid] == key)
        found = true;
      else if(key < array[mid])
        last = mid-1;
      else if(key > array[mid])
        first = mid+1;
 }
  return found;
}
```

Efficiency of Binary Search

In the worst-case scenario, searching a sorted array of 1,023 elements will take only 10 comparisons when using a binary search. Repeatedly dividing 1,023 by 2 (because after each comparison, we are able to eliminate half of the array) and rounding down (because we also remove the middle element) yields the values 511, 255, 127, 63, 31, 15, 7, 3, 1 and 0. The number 1023 is divided by 2 only 10 times to get the value 0, which indicates that there are no more elements to test. Dividing by 2 is equivalent to one comparison in the binary-search algorithm. Thus, an array of 1,048,575 elements takes a maximum of 20 comparisons to find the key, and an array of over one billion elements takes a maximum of 30 comparisons to find the key. T his is a tremendous improvement in performance over the linear search. For a one-billion-element array, this is a difference between an average of 500 million comparisons for the linear search and a maximum of only 30 comparisons for the binary search! The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 greater than the number of elements in the array which is represented as $\log_2 n$ (2^n = 1,048,575).

The following table lists the Big-O notation for some search algorithms.

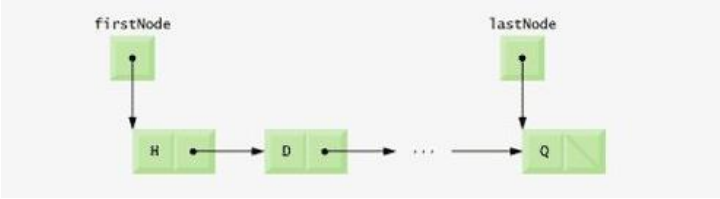
Type of Search	Big-Oh	Comments
Linear search array/ArrayList/LinkedList	O(N)	
Binary search sorted array/ArrayList	O(log N)	Requires sorted data.
Search balanced tree	O(log N)	
Search hash table	O(1)	

Part E: Linked Lists

A linked list is a linear collection (i.e., a sequence) of self-referential-class objects, called **nodes**, connected by reference links, hence, the term "linked" list. Typically, a program accesses a linked list via a reference to the first node in the list. The program accesses each subsequent node via the link reference stored in the previous node. By convention, the link reference in the last node is set to null to mark the end of the list. Data is stored in a linked list dynamically as the program creates each node as necessary. A node can contain data of any type, including references to objects of other classes.

Lists of data can be stored in arrays, but linked lists provide several advantages. A linked list is appropriate when the number of data elements to be represented in the data structure is unpredictable. Linked lists are dynamic, so the length of a list can increase or decrease as necessary. The size of a "conventional" C# array, however, cannot be altered, because the array size is fixed at the time the program creates the array. "Conventional" arrays can become full. Linked lists become full only when the system has insufficient memory to satisfy dynamic storage allocation requests. The .NET library, System.Collections, contains a class named ArrayList for implementing and manipulating linked lists that grow and shrink during program execution. However, we will create our own to use in programs.

Linked lists can be maintained in sorted order simply by inserting each new element at the proper point in the list. (It does, of course, take time to locate the proper insertion point). Existing list elements do not need to be moved. Linked list nodes normally are not stored contiguously in memory. Rather, they are logically contiguous. The following diagram illustrates a linked list with several nodes.



This diagram presents a **singly linked list** which means each node contains one reference to the next node in the list. Often, linked lists are implemented as **doubly linked list** which means each node contains a reference to the next node in the list and a reference to the previous node in the list.

Part F: Stacks

Part G: Queues