INTRODUCTION

In this week we cover the following topics

7.1 ALGORITHMS

- An *algorithm* is any well-defined computational procedure that takes a set of values as input and produces a set of values as an output within a finite amount of time.

- Computing scientists have developed many algorithms to solve numerous problems that have arisen. In this lecture we will start with basic searching algorithms, as they give us a good feel for what algorithms are about and lead us into understanding efficiency issues.

7.2 SEARCHING

- One of the most common things we want to do with arrays is search them for a particular value. There are two well-known methods for doing this, depending if the array contains elements in sorted or non-sorted order.

7.2.1 Linear Search

- If the array we are searching is unordered then we are forced to check each elements in the array, one at a time, until we either find the element or prove it's not in the array. This is called *Linear Search*. The following code demonstrates the search.

```
int linearSearch(int array[], int size,
                 int element)
{
   /*************************************\
      Use linear search to find the
      position of element in array.
      If element can't be found return -1
   \*************************************/

   int index;

   for (index = 0; index < size; index++)
   {
      if (array[index] == element)
         return index;
   }
   return -1;
}
```

7.2.2 Binary Search

- If the array is sorted we can use this knowledge to dramatically speed up the search. The searching algorithm is called *Binary Search* and we use similar techniques to allow us to find a telephone number in the telephone book. The following code demonstrates the search.

```
int binarySearch(int array[], int size
                 int element)
{
   /*************************************\
      Use binary search to find the
      position of element in array.
      Array must be in ascending
      sorted order.
      If element can't be found return -1
   \*************************************/

   int left = 0, right = size-1, mid;

   while (left <= right)
   {
      mid = (left + right) / 2;
      if (array[mid] == element)
         return mid;
      else if (array[mid] < element)
         left = mid+1;
      else
         right = mid-1;
   }
   return -1;
}
```

- An essential feature of this search technique is that it halves the number of elements it has to check with each pass of the loop.

- The following example illustrates how the algorithm works. In this example we are searching for the position of the element containing the value 15.

Pass 1 through the loop

index	0	1	2	3	4	5	6	7
value	5	8	9	12	15	16	18	21
	left			mid				right

Pass 2 through the loop

index	0	1	2	3	4	5	6	7
value	5	8	9	12	15	16	18	21
					left	mid		right

Pass 3 through the loop

index	0	1	2	3	4	5	6	7
value	5	8	9	12	15	16	18	21
					left mid right			

- On the third pass through the loop the array at index 4 contains the required value so the function would return 4.

7.3 ALGORITHM EFFICIENCY

- It is interesting to compare the efficiency of these search techniques. On average we have to check $\frac{1}{2}n$ items in an array with Linear Search but only $\lg_2(n)$ items using Binary Search. The following table compares the differences.

Items in table	Number of searches	
	Linear Search	Binary Search
100	~50	7
10,000	~5,000	14
1,000,000	~500,000	20

- As the number of items in the array grows the performance difference between Linear Search and Binary Search becomes very significant.

- As computing science has developed, a considerable amount of research time has been spent finding better algorithms to solve certain problems. The performance increases that may accrue from a better algorithm can be far more significant than the increase in machine speed that has been occurring over the last 30 years.

7.4 BIG 'O' NOTATION

Ideas of algorithm performance can be summarised by the use of Big 'O' notation.

7.4.1 Definition

If $f(n)$ and $g(n)$ are functions defined for positive integers of *n*, then we write

$$f(n)\ is\ O(g(n))$$

if there is a constant *c* such that $|f(n)| \le c|g(n)|$ for all sufficiently large integers of *n*. In this case we say $f(n)$ is *Order* of $g(n)$.

By 'sufficiently large *n*' we mean there is some value *n* beyond which the inequality is always true.

For example:
if $f(n) = 3n^2$ then $f(n)$ is $O(n^2)$

7.4.2 Keeping the Dominant Term

The order of any expression can be determined by keeping the dominant term within the expression.

If
$$f(n) = c \times g(n) + h(n)$$
with
$$c|g(n)| \geq |h(n)| \text{ for sufficiently large } n$$
then
$$f(n) \; is \; O(g(n))$$

7.4.3 Example 1

If
$$f(n) = \tfrac{1}{2}n^3 + n\lg_2 n + 4$$
then
$$c = \tfrac{1}{2}$$
$$g(n) = n^3$$
$$h(n) = n\lg_2 n + 4$$
Since
$$\tfrac{1}{2}n^3 \geq n\lg_2 n + 4 \text{ for all } n \geq 2$$
then
$$f(n) \text{ is } O(n^3)$$

7.4.4 Example 2

If

$$f(n) = n\left(\frac{n+10}{2}\right) = \tfrac{1}{2}n^2 + 5n$$

then

$$c = \tfrac{1}{2}$$
$$g(n) = n^2$$
$$h(n) = 5n$$

Since

$$\tfrac{1}{2}n^2 \geq 5n \text{ for all } n \geq 10$$

then

$$f(n) \text{ is } \mathrm{O}(n^2)$$

7.4.5 Big-O and Searching

Applying our knowledge of Big-O to the previous search algorithms we examined, it follows that
Linear Search speed is $O(n)$
Binary Search speed is $O(\lg_2 n)$

7.4.6 Common Orders

Analysis of many common algorithms in computing science often generates the following orders.

NAME	ORDER	COMMENT
Logarithmic	$O(\lg_2 n)$	very quick
Linear	$O(n)$	quick
Linear-logarithmic	$O(n\lg_2 n)$	acceptable
Quadratic	$O(n^2)$	slow
Exponential	$O(2^n)$	intractable
Factorial	$O(n!)$	intractable

The following table demonstrates their relative speeds

	n=10	100	1000	10000
$\lg_2 n$	4	7	10	14
n	10	100	1000	10000
$n\lg_2 n$	34	665	9996	132880
n^2	100	10000	1×10^6	1×10^8
2^n	1024	1.3×10^{30}	1.1×10^{301}	huge
$n!$	3.6×10^6	9.3×10^{157}	huge	huge

- From the above table the last two orders become impossible to work with once n becomes greater than about 60 for exponential and 20 for factorial.

- Unfortunately, there are a large number of important real world problems for which the best known solution is exponential or factorial.

7.5 STANDARD TEMPLATE LIBRARY

- Because you can build generic data structures in C++ using templates there is a library, called the *Standard*

Template Library (STL), that contains many standard data structures or containers as they are called.

- The containers fall into two broad categories
  1. Sequential
  2. Associative

- The following list is the sequential containers
  - vector
  - list
  - dequeue

- The following list is the associative containers
  - set
  - multiset
  - map
  - multimap

- We will explore some of these containers, and their properties, throughout this subject.

- Along with the STL there is another library called `algorithms` which contains many function template implementations of the algorithms needed to manipulate the STL containers.

- The following sites contain a reference to the STL and the algorithms libraries.

  http://www.cplusplus.com/reference/stl/
  http://www.cplusplus.com/reference/algorithm/

7.6 ORDERED VECTOR

As an example of how the STL works we can create a new data structure called an *ordered vector* that uses the STL vector as its base container.

```
sally% cat ovector.h

#ifndef VECTOR_H_
#define VECTOR_H_

#include <stdexcept>
#include <vector>

/*********************************************\
   template class for vector
\*********************************************/

template<typename dataType> class orderedVector
{
   private:
      std::vector<dataType> aVector;

      typedef typename
        std::vector<dataType>::iterator iterator;

      typedef typename
         std::vector<dataType>::const_iterator
         const_iterator;

   public:

      size_t size() const {
         return aVector.size();
      }

      bool empty() const {
         return aVector.empty();
      }
```

```
/****************************************\
   insertion and push functions
\****************************************/

void insert(const dataType &newData) {
   // set iterator to correct position
   // with binary search
   iterator position =
      lower_bound(begin(), end(), newData);

   // insert data into vector just
   // before position
   aVector.insert(position, newData);
}

/****************************************\
   iterator functions
\****************************************/

const_iterator begin() const {
   return aVector.begin();
}

iterator begin() {
   return aVector.begin();
}

const_iterator end() const {
   return aVector.end();
}

iterator end() {
   return aVector.end();
}

/****************************************\
   erase, remove and pop functions
\****************************************/

void erase(const dataType &data) {
   // set iterator to correct position
   // with binary search
```

```
      iterator itr =
         lower_bound(begin(), end(), data);


      // if data found then erase it
      if (data == *itr) aVector.erase(itr);
   }

   void pop_back() {
      aVector.pop_back();
   }

   void pop_front() {
      aVector.erase(begin());
   }

   /****************************************\
      misc functions
   \****************************************/

   bool findData(const dataType &data,
                 dataType &result)
   {
      iterator first =
         lower_bound(begin(), end(), data);

      if (data == *first) {
         result = *first;
         return true;
      } else {
         return false;
      }
   }

   /****************************************\
      overloaded operators
   \****************************************/

   // overloaded [] operator
   dataType& operator [] (int index) {
      return aVector[index];
   }
```

```
      // overloaded const [] operator
      const dataType&
         operator [] (int index) const {
         return aVector[index];
      }

      // overloaded assignment operator
      orderedVector<dataType>&
         operator =
         (const orderedVector<dataType> &other) {
         aVector = other.aVector;
         return *this;
      }
};

#endif

// #############################################

sally% cat testmain.cpp

/*********************************************\
   Test program for demonstrating container
   types
\*********************************************/

#include <sys/time.h>
#include <time.h>
#include <stdlib.h>

#include <iostream>
#include <algorithm>

#include "dataobject.h"
#include "ovector.h"

using namespace std;

double difUtime(struct timeval *first,
                struct timeval *second);
```

```
double difUtime(struct timeval *first,
                struct timeval *second)
{
   // return the difference in seconds,
   // including milli seconds


   double difsec =
      second->tv_sec - first->tv_sec;

   double udifsec =
      second->tv_usec - first->tv_usec;

   return (difsec + udifsec / 1000000.0);
}

int main()
{
   const int MAXDATA = 10000;
   dataObject *doPtr, data;
   int i, keyvals[MAXDATA];
   orderedVector<dataObject> testContainer;

   // data for calculating timing
   struct timeval first, second;
   double usecs;

   try {
      /****************************************\
         Initialise things to demonstrate the
         container
         - fill keyvals and scramble it
      \****************************************/

      for (i=0; i<MAXDATA; i++) keyvals[i] = i;
      srand(time(NULL));
      for (i=0; i<MAXDATA; i++)
         swap(keyvals[i],
              keyvals[random() % MAXDATA]);
      int middle = keyvals[MAXDATA/2];
```

```
/****************************************\
   test inserting MAXDATA data pieces
   into the container with keyval 0 to
   MAXDATA-1 in random order
\****************************************/

gettimeofday(&first, NULL);
for (i=0; i<MAXDATA; i++) {
   doPtr = new dataObject(keyvals[i]);
   testContainer.insert(*doPtr);
}

gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << MAXDATA << " items in container
   in random order\n";
cout << "time taken to push data into
   container = " << usecs <<
   " seconds\n\n";

/****************************************\
   test finding data in the container
\****************************************/

gettimeofday(&first, NULL);
testContainer.findData(
                  dataObject(0), data);
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "time taken to find first item in
   container = " << usecs << " seconds\n";

gettimeofday(&first, NULL);
testContainer.findData(
        dataObject(keyvals[middle]), data);
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "time taken to find item in middle
 of container = " << usecs << " seconds\n";

gettimeofday(&first, NULL);
testContainer.findData(
              dataObject(MAXDATA), data);
gettimeofday(&second, NULL);
```

```
      usecs = difUtime(&first, &second);
      cout << "time taken to find item doesn't
         exit in container = " << usecs <<
         " seconds\n\n";

      /****************************************\
         test removing data from the container
      \****************************************/

      gettimeofday(&first, NULL);
      testContainer.pop_front();
      gettimeofday(&second, NULL);
      usecs = difUtime(&first, &second);
      cout << "time taken to erase first item in
         container = " << usecs << " seconds\n";

      gettimeofday(&first, NULL);
      dataObject temp(keyvals[middle]);
      testContainer.erase(temp);
      gettimeofday(&second, NULL);
      usecs = difUtime(&first, &second);
      cout << "time taken to erase middle item in
         container = " << usecs << " seconds\n";

      gettimeofday(&first, NULL);
      testContainer.pop_back();
      gettimeofday(&second, NULL);
      usecs = difUtime(&first, &second);
      cout << "time taken to erase last item in
         container = " << usecs << " seconds\n";
   } catch (out_of_range &ex) {
      cout << "\nERROR - Out of Range Exception
         thrown\n" << ex.what() << "\n";
      exit(1);
   } catch(...) {
      cout << "\nERROR - undefined Exception
         thrown\n";
      exit(1);
   }

   return 0;
}
//################################################
```

```
sally% testmain

10000 items in container in random order
time taken to push data into container = 10.5353
seconds

time taken to find first item in container =
1.6e-05 seconds
time taken to find item in middle of container =
1.5e-05 seconds
time taken to find item doesn't exit in container
= 1.4e-05 seconds

time taken to erase first item in container =
0.004152 seconds
time taken to erase middle item in container =
0.000506 seconds
time taken to erase last item in container = 1e-
06 seconds
```

- Note in `testmain.cpp` I have only inserted 10,000 items in the ordered vector rather than 1,000,000.

- Every time data is inserted into the vector potentially every item in the `orderedVector` will have to be moved out of the way to make room for the new data.

- As the number of items in the vector grows it will slow down the rate at which it can insert data into order. At first it will be 1 unit of time, then 2, then 3 to *n* units of time for the number of items in the vector.

  $1 + 2 + 3 + \cdots + n = n(n+1)/2 = \frac{1}{2}n^2 + \frac{1}{2}n = O(n^2)$

  To add 100 times as many items would take almost 10000 times as long to store.

7.7 RECURSION

- So far, every program we have seen has been written with a logic called *iteration*. There is another form of logic that can be used in programming and this is called *recursion*.

- Recursive functions are functions that call themselves. Functions that don't call themselves are iterative. In this light iteration is most commonly associated with loops.

- Recursion is based on an idea called '*divide and conquer*'. In this we take a problem and divide it into smaller problems of the same nature. We keep dividing our problems until we get to base cases that can be solved simply.

- There is a proof that anything you can code using iteration you can do with recursion and anything you can do with recursion you can do with iteration. However, it doesn't mean they can be done just as easily as the other. Some problems are very easy to do using recursion while hard with iteration and visa-versa.

7.7.1 Recursion Example - Factorial

- Consider the following code

```
#include <iostream>

using namestape std;
```

```
int factorial(num);

void main()
{
   int fact, num=4;

   fact = factorial(num);
   cout << num << “ factorial = " <<
               fact << "\n”;
}

int factorial(int num)
{
   /* num factorial. Assumes num >= 1 */

   int i, fact=1;

   for (i=2; i<=num; i++) fact *= i;
   return fact;
}
```

This code will print

```
4 factorial = 24
```

- The code for `factorial` uses iteration. We can also write it recursively

```
int factorial(int num)
{
   /* num factorial. Assumes num >= 1 */

   if (num == 1) return 1;
   else return (num * factorial(num-1));
}
```

- Note how we are calling `factorial` from within `factorial`. We are chopping up the factorial problem into smaller and smaller ones. This relies on the fact that

$$n! = n \times (n-1)!$$

Which can become

$$n! = n \times ((n-1) \times (n-2)!)$$

Which can become

$$n! = n \times ((n-1) \times ((n-2) \times (n-3)!))$$

And so on until we can't decrease *n* by any more.

- Also note how we have a terminating condition that eventually stops this from happening. This is the line

```
if (num == 1) return 1;
```

In other words, the case when we get to 1!. At this point we stop chopping since 1! = 1. Without this terminating condition the program will attempt to recurse forever and quickly crash.

7.7.2 Recursion Example - Binary Search

- Consider the following code

```
#include <iostream>

using namespace std;

int binarySearch(int list[], int num,
                 int  size);

void main()
{
   int list[] = { 1, 3, 5, 7, 12, 15, 18, 20 };
   int found;
   found = binarySearch(list, 15, 8);
   if (found) printf("15 found in list\n");
   else printf("15 not found in list\n");
}

int binarySearch(int list[], int num, int size)
{
   /*****************************************\
     Binary search using iteration.
      Return location in list or -1 if not
      found.
   \*****************************************/

   int start=0, end=size-1, mid;

   while (end > start)
   {
      mid = (start + end) / 2;
      if (list[mid] >= num) end = mid;
      else start = mid+1;
   }

   if (list[start] == num) return start;
   else return -1;
}
```

- The interesting thing is that binary search naturally lends itself to being done recursively.

```
int binarySearch(int list[], int num,
                 int start, int end)
{
   /* Binary search using recursion */
   int mid;

   if (end == start) {
      if (return (list[start] == num))
         return start;
      else
         return -1;
   }
```

```
      mid = (start + end) / 2;

      if (list[mid] >= num)
         return
            binarySearch(list, num, start, mid);
      else
         return
            binarySearch(list, num, mid+1, end);
   }
```

- In the `main` function, you would call `binarySearch` in this way

```
position = binarySearch(list, 15, 0, 7);
```

- As you can see, it takes about the same amount of code to do binary search using iteration or recursion. There are many problems where it is much easier, and the code much shorter, to code using recursion. On the other hand, code written using iteration generally runs faster than code using recursion. Calling functions takes time and this slows recursion down.