INTRODUCTION

This week we cover the following subjects

10.1 SORTING

- Sorting is an interesting problem for a number of reasons.
  - There are many occasions when it needs to be done
  - There are many algorithms for doing it
  - These algorithms are fun to tinker with.

- This week we will look at a number of basic algorithms and analyse their performance. We will see that different algorithms have significantly different performance.

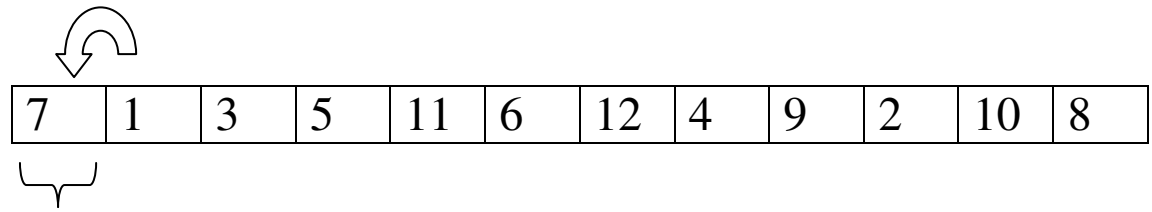
- In analysing the performance of a sorting algorithm we need to study two things
  - The number of comparisons the algorithm makes
  - The number of swaps the algorithms makes

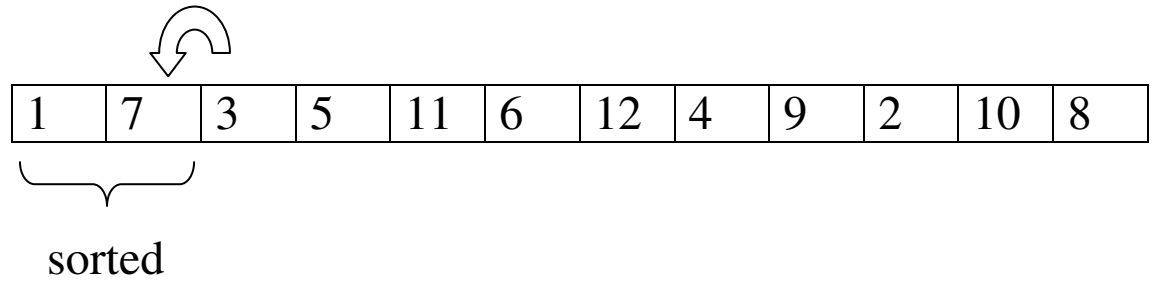
10.1.1 Insertion Sort

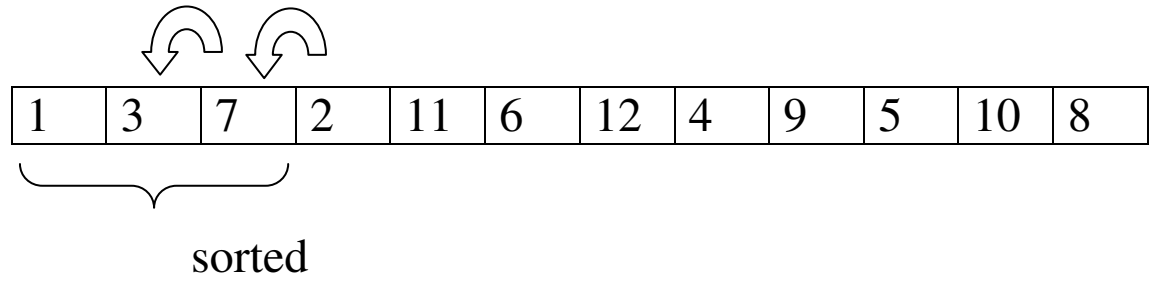
- The algorithm starts with the 2nd item in the container and compares it with the 1st item. If they are out of order then they are swapped.

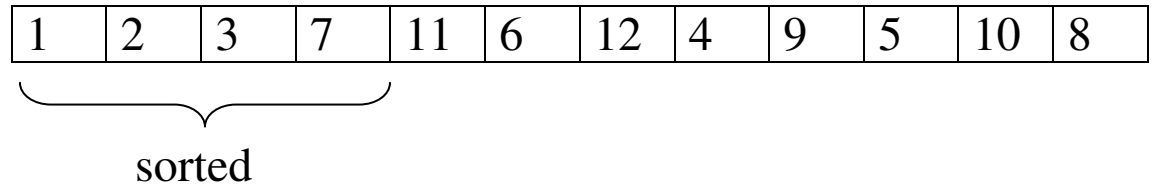
- The sort then looks at the 3rd item in the container and compares it with the previous items to find the correct position for it.

- This is continued for each subsequent item in the container. In order to do this two loops are needed.









- After each pass through the container the number of sorted items increases by 1.

Analysis:

- Assuming there are *n* items in the container then, with the 1st pass, we make possibly 1 comparison and swap. With the 2nd pass we make up to 2 comparisons and 2 swaps. We continue making passes with the

number of comparisons and swaps increasing by somewhere between 0 to n-1.

$$1+2+...+n-2+n-1=\frac{n(n-1)}{2}=\frac{1}{2}n^2-\frac{1}{2}n=O\left(n^2\right)$$

- Insertion sort is $O\left(n^2\right)$ for both comparisons and swaps made.

```
sally% cat makefile

CC = g++
prog: testinsert.o
    $(CC) testinsert.o -Wall -o testsort
testinsert.o: testinsert.cpp dataobject.h
    $(CC) -Wall -c testinsert.cpp

sally% cat testinsert.cpp

/*********************************************\
   Test program for demonstrating sorting
algorithms
\*********************************************/
#include <sys/time.h>
#include <time.h>
#include <stdlib.h>

#include <iostream>
#include <algorithm>
#include <vector>

#include "dataobject.h"

using namespace std;

double difUtime(struct timeval *first,
                struct timeval *second);
int randNum();
template <typename dataType>
   bool isSorted(const vector<dataType> &array);
template <typename dataType>
   void sortArray(vector<dataType> &array);
```

```
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 difusec =
      second->tv_usec - first->tv_usec;

   return (difsec + difusec / 1000000.0);
}

int randNum()
{
   // rand() produces pseudo-random number in
   // range 0 to RAND_MAX
   // for most compilers RAND_MAX is 2^15-1
   // randNum() produces pseudo-random number
   // in range 0 to 2^31-1

   static bool initialised = false;

   if (!initialised) {
      srand(time(NULL));
      initialised  = true;
   }

   return ((rand() << 16) + rand());
}

template <typename dataType>
   bool isSorted(const vector<dataType> &array)
{
   // return true or false if vector
   // is sorted in ascending order

   for (unsigned int i=0;
        i<array.size()-1; i++) {
      if (array[i] > array[i+1]) return false;
   }
   return true;
```

```
}

template <typename dataType>
   void sortArray(vector<dataType> &array)
{
   // sort vector using insertion sort
   unsigned int inner, outer, inner2;
   dataType temp;

   for (outer=1; outer<size; outer++) {
      inner2 = inner = outer;
      inner2--;
      temp = array[inner];
      while (inner > 0 && temp < array[inner2]) {
         array[inner] = array[inner2];
         inner--;
         inner2--;
      }
      array[inner] = temp;
   }
}

int main()
{
   const unsigned int MAXDATA = 10000;
   vector<dataObject> array;
   dataObject *doPtr;
   unsigned int i;

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

   try {
      /*****************************************\
         Initialise things to demonstrate
         the sort
         - fill vector with dataObjects with
           unique keyvals
         - scramble the vector
      \*****************************************/
```

```
      for (i=0; i<MAXDATA; i++) {
         doPtr = new dataObject(i);
         array.push_back(*doPtr);
         delete doPtr;
      }

      for (i=0; i<MAXDATA; i++)
         swap(array[i],
              array[randNum() % MAXDATA]);
      cout << MAXDATA << " Items in vector\n";

      /*****************************************\
         sort the vector and show timing of sort
      \*****************************************/

      gettimeofday(&first, NULL);
      sortArray(array);
      gettimeofday(&second, NULL);
      usecs = difUtime(&first, &second);
      cout << "Time to sort vector = "
          << usecs << " seconds\n";

      if (isSorted(array)) {
         cout << "vector confirmed sorted\n";
      } else {
         cout << "Sort failed\n";
      }

   } catch(...) {
      cout <<
        "\nERROR - undefined Exception thrown\n";
      exit(1);
   }

   return 0;
}

sally% ./testsort
10000 Items in vector
Time to sort vector = 710.1618 seconds
Vector confirmed sorted
```

10.1.2 Shell Sort

- A useful property of insertion sort is that it is much faster if the container it is sorting is partially ordered. Shell sort is a modification of insertion sort that partially orders the list with each pass through it.

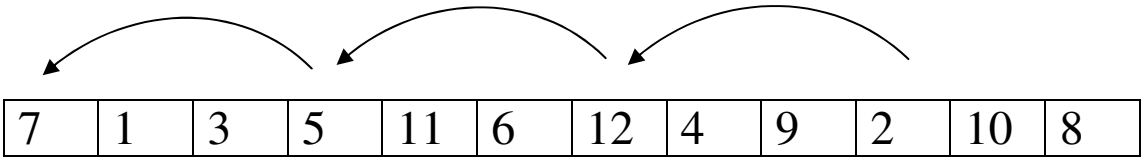
- Instead of comparing items next to each other, we compare items that are far apart in the container. This is done using a *gap* variable to determine how far apart the variables are when being compared. This will have the effect of partially ordering the list, though done so fairly quickly.

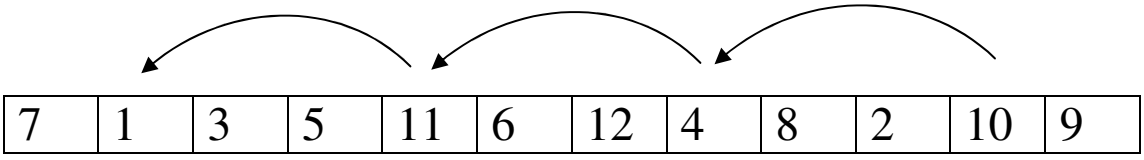
- Once the data is sorted using the current gap, the gap is reduced.

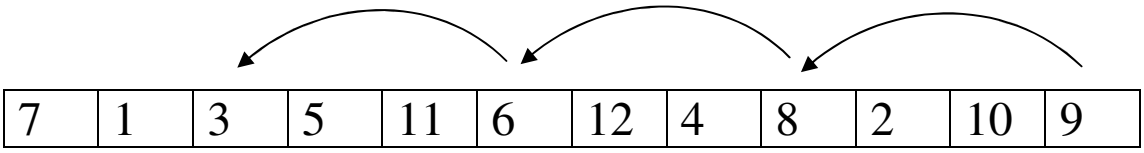
- We keep doing this until the gap becomes equal to 1 and we are reduced to a standard insertion sort - though on a list that is almost completely sorted.

- With each pass through, it gets closer and closer to being fully sorted.

sort with gap = 3







result after sorting using gap of 3

2	1	3	5	4	6	7	10	8	12	11	9

result after sorting using gap of 2

2	1	3	5	4	6	7	9	8	10	11	12

Analysis:

Shell sort has so far defied analysis. It is certainly much faster than insertion sort. Empirical studies indicate it is approximately $O\left(n^{1.25}\right)$.
The following table illustrates the difference in speed

	n=10	100	1000	10000
$O\left(n^{1.25}\right)$	18	316	5623	1×10^{5}
$O\left(n^{2}\right)$	100	10000	1×10^{6}	1×10^{8}

```
template <typename dataType>
   void sortArray(vector<dataType> array)
{
   // sort vector using simple shell sort
   unsigned int gap, inner, outer;
   dataType temp;

   for (gap = size; gap > 0;
        gap = (gap > 1 && gap < 5) ?
              1 : gap*5/11) {
      // insertion sort using gap
      for (outer=gap; outer<size; outer++) {
         inner = outer;
         temp = array[inner];
         while (inner >= gap &&
                 temp < array[inner-gap]) {
            array[inner] = array[inner-gap];
            inner -= gap;
         }
         array[inner] = temp;
      }
   }
}
```

```
sally% ./testsort
10000 Items in vector
Time to sort vector = 0.561978 seconds
Vector confirmed sorted
```

10.1.3 Quick sort

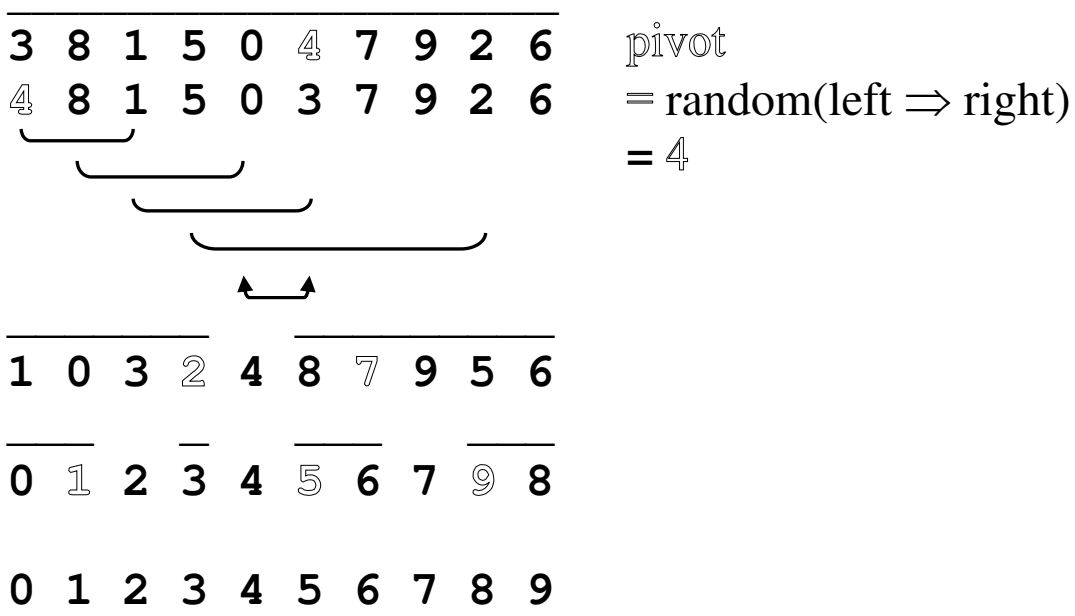
- One of the most efficient sorting algorithms around is the Quick sort algorithm. Because of this it is the most commonly used sorting algorithm in the world. While very difficult to implement using iteration, this algorithm is a 'natural' for recursion.

  1. The first thing Quick sort does is find a *pivot* value that roughly represents a value somewhere in the middle of the items being sorted. One common way

this is done is to randomly pick one of the values being sorted

2. Once this value is calculated, Quick sort then re-orders the container so one half of the list only contains values less than the pivot value and the other half only contains values greater than the pivot. This can be done in one pass though the container.

3. Quick sort is then recursively applied to each half of the container.



Analysis:

- Re-ordering the list takes one pass through the list. If the list has n items then this is $O(n)$.

- If we manage to split the list evenly then we are halving the size of the list being sorted with each

recursive call. The number of times we can halve the list is $\lg_2(n)$.

- Putting this together the speed of quick sort is $O(n\lg_2 n)$.

- The previous analysis is a best case scenario. In a worst case scenario, each time we split the list it ends up with one value on one side and all the rest on the others. In this case, the speed drops to $O(n^2)$. Therefore, it is critical to select a good pivot value each time we re-order the list. Various refinements of the algorithm exist for this.

```
sally% cat testquick.cpp

/*********************************************\
   Test program for demonstrating sorting
   algorithms
\*********************************************/
#include <sys/time.h>
#include <time.h>
#include <stdlib.h>

#include <iostream>
#include <algorithm>
#include <vector>

#include "dataobject.h"

using namespace std;

double difUtime(struct timeval *first, struct
timeval *second);
int randNum();

template <typename dataType>
   bool isSorted(const vector<dataType> &array);
```

```
template <typename dataType>
   unsigned int median(
      const vector<dataType> &array,
      unsigned int a, unsigned int b,
      unsigned int c);

template <typename dataType>
   void sortArray(vector<dataType> &array,
                  int left, int right);

template <typename dataType>
   bool isSorted(const vector<dataType> &array)
{
   // return true or false if vector is sorted
   // in ascending order

   for (unsigned int i=0;
        i<array.size()-1; i++) {
      if (array[i] > array[i+1]) return false;
   }
   return true;
}

int randNum()
{
   // rand() produces pseudo-random number
   // in range 0 to RAND_MAX
   // for most compilers RAND_MAX is 2^15-1
   // randNum() produces pseudo-random number
   // in range 0 to 2^31-1

   static bool initialised = false;

   if (!initialised) {
      srand(time(NULL));
      initialised  = true;
   }

   return ((rand() << 16) + rand());
}
```

```
template <typename dataType>
   void sortArray(vector<dataType> &array,
                  int left, int right)
{
   // sort vector using simple quick sort
   // assumes there are no duplicated objects

   unsigned int middle, i;
   dataType pivot;

   while (left < right) {
      // deal with simple sort condition
      if (right - left == 1) {
         if (array[left] > array[right])
            swap(array[left], array[right]);
         return;
      }

      // create pivot
      middle = left +
               (randNum() % (right - left + 1));
      swap(array[left], array[middle]);
      middle = left;
      pivot = array[middle];

      // order list around pivot
      for (i=left+1; i<=right; i++) {
         if (array[i] < pivot) {
            middle++;
            swap(array[middle], array[i]);
         }
      }
      swap(array[left], array[middle]);
```

```
      // use tail recursion
      if (middle - left < right - middle) {
         // left side smaller than right side.
         // recurse on left
         if (middle > left+1)
            sortArray(array, left, middle-1);
         left = middle+1;
      }
      else {
         // right side smaller than left side.
         // recurse on right
         if (middle+1 < right)
             sortArray(array, middle+1, right);
         right = middle-1;
      }
   }
}

int main()
{
   const unsigned int MAXDATA = 10000;
   vector<dataObject> array;
   dataObject *doPtr;
   unsigned int i;

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

   try {
      /*****************************************\
         Initialise things to demonstrate the
         sort
         - fill vector with dataObjects with
           unique keyvals
         - scramble the vector
      \*****************************************/

      for (i=0; i<MAXDATA; i++) {
         doPtr = new dataObject(i);
         array.push_back(*doPtr);
         delete doPtr;
      }
```

```
      for (i=0; i<MAXDATA; i++)
         swap(array[i],
         array[randNum() % MAXDATA]);
      cout << MAXDATA << " Items in vector\n";

      /*****************************************\
         sort the vector and show timing of sort
      \*****************************************/

      gettimeofday(&first, NULL);
      sortArray(array, 0, array.size()-1);
      gettimeofday(&second, NULL);
      usecs = difUtime(&first, &second);
      cout << "Time to sort vector = " <<
         usecs << " seconds\n";

      if (isSorted(array)) {
         cout << "Vector confirmed sorted\n";
      } else {
         cout << "Sort failed\n";
      }

   } catch(...) {
      cout <<
        "\nERROR - undefined Exception thrown\n";
      exit(1);
   }

   return 0;
}

sally% ./testsort

10000 Items in vector
Time to sort vector = 0.18031 seconds
vector confirmed sorted
```

- The random selection of the pivot doesn't make much difference to the speed of quick sort when applied to data that is fairly random in distribution.

- It does make a difference when applied to data that is partially sorted, either in reverse or forward order. This makes it very unlikely that quick sort will have $O(n^2)$ performance.

  Quick sort applied to 10000 data items in reverse order and randomly selecting the pivot.

```
10000 Items in array
Time to sort array = 8.83709 seconds
Array confirmed sorted
```

  Quick sort applied to 10000 data items in reverse order and selecting the first value in the array as the pivot.

```
10000 Items in array
Time to sort array = 36.31543 seconds
Array confirmed sorted
```

10.1.4 Summary of Sorts

	Insertion	Shell	Quick
Order	$O(n^2)$	$O(n^{1.25})$	$O(n \lg_2 n)$
10000 items	710.1618	0.561978	.18031

10.2 `<algorithm> - sort()`

- To sort the various containers in the Standard Template Library quick sort has been implemented in the algorithm library as a template function.

- It makes use of iterators to let the function know the first and last item in the container.

- The following code illustrates the use of the sort function.

```
sally% cat testquick.stl.cpp

/*********************************************\
   Test program for demonstrating sorting
   algorithms
\*********************************************/
#include <sys/time.h>
#include <time.h>
#include <stdlib.h>

#include <iostream>
#include <algorithm>
#include <vector>

#include "dataobject.h"

using namespace std;
double difUtime(struct timeval *first,
                struct timeval *second);
int randNum();
template <typename dataType>
   bool isSorted(const vector<dataType> vec);

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

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

```
try {
   /*****************************************\
      Initialise things to demonstrate the
      sort
      - fill vector with dataObjects
        with unique keyvals in scrambled order
   \*****************************************/

   // create keyvals and scramble them
   for (i=0; i<MAXDATA; i++) keyvals[i] = i;
   for (i=0; i<MAXDATA; i++)
      swap(keyvals[i],
           keyvals[randNum() % MAXDATA]);

   // fill vector with scrambled
   // keyval dataObjects
   for (i=0; i<MAXDATA; i++) {
      doPtr = new dataObject(keyvals[i]);
      vec.push_back(*doPtr);
      delete doPtr;
   }
   cout << MAXDATA << " Items in vector\n";

   /*****************************************\
      sort the vector and show timing of sort
   \*****************************************/

   gettimeofday(&first, NULL);
   sort(vec.begin(), vec.end());
   gettimeofday(&second, NULL);
   usecs = difUtime(&first, &second);
   cout << "Time to sort vector = "
        << usecs << " seconds\n";

   if (isSorted(vec)) {
      cout << "Vector confirmed sorted\n";
   } else {
      cout << "Sort failed\n";
   }

} catch(...) {
   cout <<
     "\nERROR - undefined Exception thrown\n";
   exit(1);
```

```
   }

   return 0;
}

sally% testsort
10000 Items in vector
Time to sort vector = 0.070275 seconds
Vector confirmed sorted
```

- The efficient coding of the sort within the algorithm library allows it to run faster than the hand coded sort.

- It maybe that there is no overloaded < operator for the dataType's in the container. If that is so a comparison function will need to be passed to the sort function. E.g.

```
bool compare(const dataObject &d1,
             const dataObject &d2)
{
   return (d1.getKeyval() < d2.getkeyval());
}

sort(vec.begin(), vec.end(), compare);
```