SORTING AND SEARCHING

Today

- Last time we considered a simple approach to sorting a list of objects.
- This lecture will look at another approach to sorting.
- We will also consider how one *searches* through a list to see if it contains something we are looking for.
- This leads us to think about the *efficiency* of different approaches, and the *analysis of algorithms* in general.

Recap

- Let's recall the sorting algorithm from last lecture:

```
for(counter = 0; counter < 6; counter++)
  {
    for(counter2 = counter; counter2 < 6; counter2++)
      {
         if(numbers[counter2] < numbers[counter])
          {
            temp              = numbers[counter];
            numbers[counter]  = numbers[counter2];
            numbers[counter2] = temp;
          }
      }
  }
```

- We'll call this *linear sort*.

Variants of linear sort

- This sorting algorithm will sort integers into ascending order (so that the smallest is first).
- We can easily change the algorithm so that it sorts largest first.
- We replace:

```
if(numbers[counter2] < numbers[counter])
```

with

```
if(numbers[counter2] > numbers[counter])
```

- The same algorithms will work on any kinds of data that we can compare using > and <.
- Thus we can also sort `char`, `double` and even `string`.

Bubblesort

- Linear sort can often be inefficient.
- If we ask it to sort a list of numbers that is already in order, it will take as long to do this as it will to sort a list of numbers that is randomly ordered.
- Other kinds of sorting can be more efficient.
- One such approach is *bubblesort*
- Bubbblesort works like this:

  Compare each pair of numbers in the list. If the first is bigger than the second, swap them. When you get to the end of the list, if you made any swaps, go through the list again. Repeat until you don't make any swaps.

- Here's bubblesort in C++:

```
while(swapped)
   {
      swapped = false;

      for(count = 0; count < 5; count++)
         if(numbers[count] > numbers[count+1])
            {
               temp             = numbers[count];
               numbers[count]   = numbers[count+1];
               numbers[count+1] = temp;

               swapped = true;
            }
    }
```

- This sorts smallest first. As before we can easily modify it to sort largest first.

Genetic algorithms

- The program `ga.cpp` which you can download from the course website is an example of the use of bubblesort.
- It is also an example of *biologically inspired computing*, where ideas from biology are used to make computer programs more efficient.
- In *genetic algorithms* we breed solutions to a computing problem, and allow them to evolve until we have the best solution.
- Genetic algorithms can be a very efficient way to find solutions to some problems.

Searching

- Sorting a list of things is one very common thing to want to do.
- Another common operation is searching to see if something is in a list.
- The simplest way to do this is to look through the list, item by item:

```
for(counter = 0; counter < 6; counter++)
  {
     if(numbers[counter] == numberWeWant)
        {
          cout << ''We found it'' << endl;
        }
  }
```

- We'll call this *linear search*.

- Again this isn't very efficient.
- In the worst case, it means we have to look through every element of the list to find if the number is in there.
- That is fine if the list is 6 elements long, but not so fine if it is a million elements long.
- Much more efficient is *binary search*, though binary search only works if the list is sorted.

Binary search

- Binary search works as follows, assuming the list is sorted smallest first.
- Look at the middle element of the list.
- If it is the one we are looking for, we are done.
- If the middle element is larger than the one we are looking for, then the one we are looking for must be in the first half of the list (if it is in the list).
- If the middle element is smaller than the one we are looking for, then the one we are looking for must be in the second half of the list (if it is in the list).
- Repeat in the relevant half of the list

Analysis of algorithms

- If you try binary and linear search out on some examples, you will see that binary search usually finds the result (that the thing we want is in or out of the list) quicker than linear search.
- However, we can say more precisely what the advantage of binary search is.
- We consider how many comparisons we will have to do for an arbitrary list that holds N elements,
- For linear search, the worst thing that can happen is that we have to look at all N elements.
- Sometimes we will look at less, and on average we will end up looking at $\frac{N}{2}$ elements.
- In binary search we will have to look at $\log_2(n+1)$ elements at most.

- We can look at the worst case number of comparisons for different values of N.

N	Linear	Binary
100	100	7
1,000	1,000	10
1,000,000	1,000,000	20

- So we can see that binary search is a lot more efficient than linear search as the size of the list increases.
- However, to use binary search, we need a sorted list.

- How efficient is sorting?
- Well if we use linear sorting on a list with N elements, we have to do N comparisons followed by $N-1$ comparisons, followed by $N-2$ comparisons, followed by ... all the way down to $N-N$ comparisons.
- In total, that is:

$$\frac{N(N+1)}{2}$$

  comparisons.
- The most significant part of this number is the N^2, and we write the number of comparisons as $O(N^2)$.
- This is known as "Big O" notation.

- Linear searching is $O(N)$, and so will be more efficient than linear sorting since N is always smaller than N^2.
- Binary searching is $O(\log N)$, and so is more efficient than either linear searching or linear sorting since $\log N$ is smaller than N and N^2.
- However, if we sort with linear sort and and then search using binary search, overall that will be less efficient than using linear search.
- Note that there are other algorithms that sort more efficiently than either linear sort and bubblesort.
- There are also slgorithms that search unsorted lists more efficiently than linear search.

Summary

- This lecture continued to look at sorting, introducing bubblesort.
- This lecture also looked at searching, considering linear search and binary search.
- Finally, this lecture considered the analysis of these simple sorting and searching algorithms and introduced Big O notation.