Searching and sorting

- it is very common to store and search large amounts of data
- if the data is unsorted then in a search we might have to look at every stored value to verify if a specific value is present/not
- if the data is already sorted then we can apply more efficient search techniques
- for now we'll assume our data is stored in an array
- we'll look at two search techniques (linear and binary) and one sorting techniques (bubblesort), as well as a routine to check if our data values are sorted or not

linear searches

- a linear search looks in each position of the array, going from first to last (or from last to first)
- it works whether the data is sorted or not, but can be slow for large arrays since
  - if the value is in the array we still might have to look in every position before we finally find it
  - if the value isn't in the array we have to check every position before we can be sure it isn't anywhere in the array

Example: linear search

- here going from front to back, looking for a target value and returning the first array position where we find it
- if we never find it then we return -1 (something that isn't a valid position, allows caller to recognize it wasn't found)

```
int search(float arr[], int size, float target)
{
   for (int p=0; p<size; p++) {
      if (arr[p] == target) {
         return p;  // found it!
      }
   }
   return -1; // never found it
}
```

Calling search

- after calling search, we check the return value to see if it found something

```
int pos = search(myArray, myArraySize, 17.5);
if (pos == -1) {
   cout << “17.5 was not in the array” << endl;
} else {
   cout << “found 17.5 in position “ << pos << endl;
}
```

binary search on sorted data

- if we know our data is in sorted order (and know if it's in increasing order or decreasing order) we can use (more efficient) binary search
- use low and high to keep track of the section of the array we are still searching, start with 0 and size-1
- repeat until target found or low > high:
  - compute the middle position between low and high
  - if we find the target in the middle position then we're done
  - else if the value in the middle is bigger than what the target then we can ignore everything from middle to high, so change high to middle-1
  - otherwise we can ignore everything from low to middle, set low to middle+1

recursive binary search

- assumes sorted in increasing order

```
int binarySearch(float arr[], int low, int high, float target)
{
   if (low > high) {
      return -1;  // impossible range
   }
   int mid = (low + high) / 2;  // int division, drops fractions
   if (arr[mid] == target) {
      return mid;  // found it!
   } else if (arr[mid] > target) {
      high = mid - 1;  // value must be in lower half of this section
      return binarySearch(arr, low, high, target);
   } else {
      low = mid + 1; // value must be in upper half of this section
      return binarySearch(arr, low, high, target);
   }
}
```

Sample call

- as with linear search, except we're passing 0 and size-1 as the lower/upper portions of the array to search
- if the array finds duplicates then binarySearch doesn't necessarily find the first one, just guaranteed to find one

```
int pos = BinarySearch(myArray, 0, myArraySize-1, 17.5);
if (pos == -1) {
   cout << “17.5 was not in the array” << endl;
} else {
   cout << “found 17.5 in position “ << pos << endl;
}
```

Efficiency

- binary search “discards” half of remaining elements with each call
  - after 1 call the elements left are divided by 2, after 2 calls by 2*2, after three calls by 2*2*2, ... after i calls by 2^i ...
- suppose we start with 2^N elements
  - after N calls the space left to search is just $2^N/2^N$, i.e. a single element!
  - thus 10 calls can search roughly a thousand elements (1024 = 2^{10})
  - 20 calls can search roughly a million elements (2^{20})
  - 30 calls can search roughly a billion elements (2^{30})
  - etc
- much more efficient than linear search, which had to look at all 2^N

Iterative (loop) version

```
int binarySearch(float arr[], int size, float target)
{
   int low = 0;
   int high = size-1;
   do {
        int mid = (low + high) / 2;
        if (arr[mid] == target) {
           return mid;
        } else if (arr[mid] > target) {
           high = mid - 1;
        } else {
           low = mid + 1;
        }
   } while (low <= high);
   return -1;
}
```

same logic, just using a loop to keep updating low and high until we're done

Bubblesort

- first sorting algorithm, for an array of size N (in this example we assume sorting in increasing order)
- go through the entire array over and over
  - each time we go through the array, we compare all the pairs of adjacent elements
    - if a pair of elements is out of order then we swap them
- at the end of each pass through the array it will be closer to being sorted
- also after each pass, the next biggest value will have reached it's correct position in the array

Example: bubblesort

- initial array content 3 17 -1 8
- in first pass
  - 3,17 are in ok order so move on
  - 17,-1 out of order so swap them, now 3 -1 17 8
  - 17,8 out of order so swap them, now 3 -1 8 17
- in second pass
  - 3,-1 out of order so swap them, now -1 3 8 17
  - 3,8 are in ok order, so move on
  - 8,17 are in ok order so move on
- keep repeating passes until eventually all sorted

swap revisited

- we'll be swapping array elements frequently, so would help to have a swap function

```
void swap(float &x, float &y)
{
    float originalX = x;
    x = y;
    y = originalX;
}
```

bubblesort version 1

- we can guaranteed that after size-1 passes all values will have reached their correct position

```
void bubblesort(float arr[], int size)
{
   // pass tracks how many passes we've made through array
   for (int pass=0; pass<size-1; pass++) {
      // pair is position of the second of the two elements we're comparing now
      for (int pair=1; pair<size; pair++) {
         if (arr[pair-1] > arr[pair]) {
            swap(arr[pair-1], arr[pair]); // swap the out-of-order pair
         }
      }
   }
}
```

bubblesort version 2

```
// stops after a pass finds nothing out of order
void bubblesort(float arr[], int size)
{
   bool sorted;
   int pN = 1; // which pass we're on
   do {
        sorted = true;
        // don't have to go into last pN positions, they're already ok
        for (int p=0; p<size-pN; p++) {
             if (arr[p] > arr[p+1]) {
                sorted=false;  // found something still out of order
                swap(arr[p], arr[p+1]);
             }
        }
        pN++;
   } while (!sorted);
}
```

checking if sorted

- we may not know if data in an array is already sorted, so can write a routine to check

```
bool isSorted(float arr[], int size)
{
   for (int p=0; p<size-1; p++) {
       if (arr[p] > arr[p+1]) {
          return false; // found out of order pair
       }
   }
   return true; // everything was in order
}
```