Searching and Sorting

What do we do with arrays? They are particularly useful for sorting and searching.

Linear Search

The simplest way to search a linear array is by using the "linear search algorithm." The algorithm examines each array element, one at a time, using a loop, and compares it to the "target," the data element it is searching for.

```
//linearsearch.cpp
//modified from Hubbard ex 5.11 p. 134

#include <iostream>

void search (int&, int&, int[], int, int);

int main(){
   int data[] = {55, 22, 99, 66, 44, 88, 33, 77};
   int target, found, location, n=8;
   cout << "Target:  ";
   cin >> target;
   while (target !=0) {
      search(found, location, data, n, target);
      if (found) cout << target << " is at a[" << location << "].\n";
      else cout << target << " was not found.\n";
      cout << "Target:  ";
      cin >> target;
   }
   cout << "Press any key to close console window."; char c; cin >> c;
}

//Linear search algorithm
void search (int &found, int &loc, int x[], int n, int target){
   found = loc = 0;
   while (!found && loc < n) found = (x[loc++] == target);
   --loc;
}
```

Output:

```
Target:  44
44 is at a[4].
Target:  88
88 is at a[5].
Target:  50
50 was not found.
Target:  0
Press any key to close console window.
```

Searching is more efficient if the array is in some sort of order. Then we may not have to exhaustively examine each and every element in the array in order to find the target.

Sorting

We will study three basic types of sorting algorithms:
Insertion sort
Selection sort
Exchange Sort

Insertion Sort: Take the 'next' key of the unsorted list and insert it in its proper relative position in a growing sorted list of data.

The entire list of sorted keys must be available in main memory. But the unsorted list can be input "as you go along."

Selection sort: Repeatedly select the smallest key remaining in the unsorted list as the next key in a growing sorted list of data. With this algorithm, there are two arrays, the original unsorted array and the to-be-created sorted array. The entire list of unsorted keys must be available in main memory. The sorted list may be output "as you go along," i.e., one key (probably tied to a record or an object) at a time.

This type of algorithm requires N passes over the unsorted list. In each pass, (N-1) comparisons are made. Thus, the total number of comparisons is N(N-1).
e.g.,

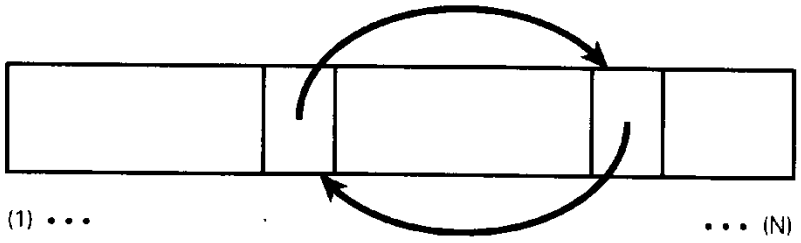
```
for (int next=0; next<SIZE; next++)
{
   min=0;
   for (int j=1; j<SIZE; j++)
      if (x[j] < x[min]) min=j;
   xsorted[next] = x[min]
   x[min] = INT_MAX;
}
```

Try (play computer) with example:
x = [2, 4, 8, 5, 1, 3, 7, 6]

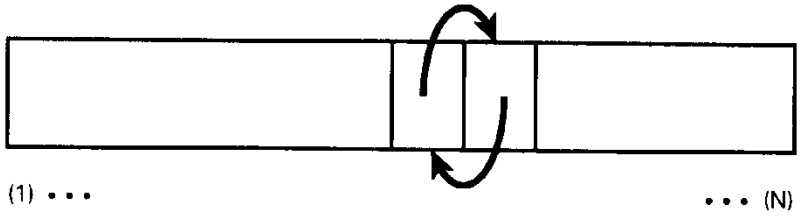
Selection and insertion sorts are both relatively inefficient but are easy to understand and to program. Both are widely used.

Exchange sort: In an exchange sort, we sort the data *in place*, without copying them to a different array. We compare pairs of keys and swap (exchange) them in they are not in the correct relative positions.

The **Bubble Sort** is a commonly used type of Exchange Sort. There are others, some of which are more clever in their efficiency. The bubble sort allows each key to float to its proper position through a series of pairwise comparisons, and exchanges with adjacent key values. Each pass results in bubbling one key to its final position in the sorted list.



(a) General exchange sort



(b) Bubble sort

The bubble sort algorithm sorts an array without copying it to a second empty array. At each iteration, the next largest data item is moved into its correct position.

With the bubble sort

- there are at most n-1 passes, and
- at the end of each pass, 1 more record is in its final sorted position.

Example:

Unsorted list	Pass 1	After Pass 1	Pass 2	After Pass 2	Pass 3	After Pass 3	Pass 4	After Pass 4	Pass 5
2	2	2	2	2	2	2	2 1	1	1
4	4	4	4	4	4 1	1	2	2	2
8	8 5	5	5 1	1	4 3	3	3	3	3
5	8 1	1	5 3	3	4	4	4	4	4
1	8 3	3	5	5	5	5	5	5	5
3	8 7	7	7 6	6	6	6	6	6	6
7	8 6	6	7	7	7	7	7	7	7
6	8	8	8	8	8	8	8	8	8

Remarks: 1st pass orders: 8 — 2nd pass orders: 7 — 5th pass No exchange

```
//bubblesort.cpp
// modified from Hubbard ex 5.12 p. 135

#include <iostream>
void print (const float[], const int);
void sort (float[], const int);
void swap (float&, float&);

int main(){
   float data[] = {55.5, 22.5, 99.9, 66.6, 44.4, 88.8, 33.3, 77.7};
   int n = 8;
   print(data, n);
   sort(data, n);
   print(data, n);
   cout << "Press any key to close console window."; char c; cin >> c;
   return 0;
}
//*************************************************************//
void print (const float x[], const int n){
   for (int i=0; i<n; i++) cout << '\t' << x[i];
   cout << endl;
}
//*************************************************************//
void swap (float &a, float &b) {
   float temp = a;
   a = b;
   b = temp;
}
//*************************************************************//
//Bubble Sort Algorithm
void sort (float x[], const int n) {
   for (int i=n-1; i>0; i--)
      for (int j=0; j<i; j++)
                     if (x[j] > x[j+1]) swap(x[j],x[j+1]);
}
```

Output:

```
  55.5   22.5   99.9   66.6   44.4   88.8   33.3   77.7
  22.5   33.3   44.4   55.5   66.6   77.7   88.8   99.9
Press any key to close console window.
```

Enhanced version of the Bubble Sort: Take advantage of the fact that the list may be ordered well before the $(n-1)^{th}$ pass. Add a flag set to "true" at the beginning of each pass. Proceed with the comparisons. If an exchange is required, do it and set the flag to "false." Test the flag at the end of each pass: if the value of the flag is "true" then the data is sorted.

Evaluating the sort algorithms

We can evaluate these sorting algorithms on performance, storage requirements, and ease of implementation.

What's the worst case (maximum number of compares) for each?

Selection sort:

- n passes, (n-1) compares / pass
- Total # compares = n(n-1)
- Total # moves = n

Insertion sort:

- n passes
- Max # compares = n(n-1)/2

Bubble sort:

- At most n-1 passes
- n/2 comparisons per pass (approx..)
- Max # compares = n(n-1)/2

For a short list (say, 15-20 records), these are all "good" sorts; for a moderately short list (50-60 records), they are acceptable. For long lists, their performance deteriorates. Butthey have the advantage of ease of implementation.

Shell sort and **Quicksort** are better and faster for long lists.

Binary Search:

The binary search algorithm makes use of the fact that the array being searched is in sorted order.

In a sequential (linear) search, after each comparison, the file is reduced by only 1. With the binary search technique, after each comparison, either the search terminated successfully or the size of the file remaining to be searched is reduced by about ½.

At each pass:
Compare the middle key with the search argument, x. Then based on the comparison, draw a conclusion for the following:

- If x < key [mid], then the record must be in the lower-numbered half of the file (if it is present at all).
- If x = key[mid], this is the record being searched for
- If x > key[mid], then the record must be in the higher-numbered half of the file (if it is present).

```
//binarysearch.cpp
//modified from Hubbard ex 5.13 p. 136

#include <iostream>
void search (int&, int&, int[], int, int);

int main(){
   int data[] = {22, 33, 44, 55, 66, 77, 88, 99};
   int target, found, location, n=8;
   cout << "Target:  ";
   cin >> target;
   while (target != 0) {
      search(found, location, data, n, target);
      if (found) cout << target << " is at data[" << location << "].\n";
      else cout << target << " was not found.\n";
      cout << "Target:  ";
      cin >> target;
   }
   cout << "Press any key to close console window."; char c; cin >> c;
   return 0;
}
//*************************************************************//
//Binary Search Algorithm
void search (int &found, int &loc, int x[], int n, int target){
   int lower = 0, upper = n-1;
   found = 0;
   while (!found && lower <= upper){
      loc = (lower + upper)/2;           //the midpoint
      found = (x[loc] == target);
      if (x[loc] < target) lower = loc + 1;
      else upper = loc - 1;
   }
}
```

Output:

```
Target:  88
88 is at data[6].
Target:  33
33 is at data[1].
Target:  50
50 was not found.
Target:  0
Press any key to close console window.
```