SORTING

Today

- This unit looks at a simple forms of sorting and searching.
- We illustrate sorting by putting some numbers that have been read into an array into order.
- We will start to look at this kind of thing today.
- Later in the unit we will look at mre forms of sorting and how we search through a list of numbers efficiently.
- Both sorting and searching are tasks that we often need to do.
- They also provide a nice basis of thinking about what makes a *good* program, rather than just a program that works.
- First, though, we will take a quick look at the `switch` control structure.

The `switch` statement

- `switch` is a form of conditional control structure.
- In some ways it is more powerful than `if` and `if/else`
  - One control structure allows multiple comparisons.
- On other ways it is less powerful than `if` and `if/else`
  - It is more restricted in what it can compare.
- Let's look at an example (taken from the program `sort.cpp` which you can download from the page for Unit V.


```
switch(choice){

case 0:
  selectSort(numbers, aux, SIZE);
  break;

case 1:
  selectionSort(numbers, aux, SIZE);
  break;

case 2:
  insertionSort(numbers, aux, SIZE);
  break;

case 3:
  bubbleSort(numbers, SIZE);
  break;

default:
  // don't sort the array
  ;
}
```

- `switch` works by testing the value of a variable, `choice` in this case.
- `choice` has to be an `int`.
- (It is also possible to use a `char` and exploit the fact that a `char` is really just a small integer.).
- Here the value of `choice` is tested against the values that follow the keyword `case`.
- If there is a match, the following lines of code are executed.

- The `break` forces execution to skip to the end of the `switch`.
  - Without the `break`, every line of code between the matching `case` and the end of the `switch` would be executed.
- The `default` is a `case` that always matches. Here it does nothing but often we use it to make sure that the `switch` chooses something.
- So, the advantage of the `switch` is that just one control structure neatly allows us to choose between many alternatives.
- The disadvantage is that you can't make the condition a complex logical expression.
- In fact it can't even be a simple logical expression — it has to be a number.

Sorting

- We'll look at four different methods for sorting.
- These are:
  - blort sort
  - selection sort
  - insertion sort
  - bubble sort
- There are many other kinds of sorting...

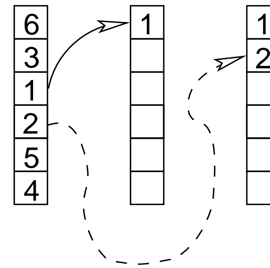
Blort sort

- "Fun but stupid"
- Works like this:
  1. Check to see if the array is ordered.
  2. If it is, then sort is over.
  3. Otherwise, shuffle the elements in the array and start over.
- This *will* work (eventually).
- However, it is not, generally, a good way to go.

Selection sort

- Selection sort uses an *auxilliary* array
  - Another array that collects the sorted numbers.
  - After sorting these values are copied back into the original array.
- The basic algorithm to order the array from *lowest* to *highest* is
  1. Select the smallest element still left in the array.
  2. Add it to the end of the auxilliary array.
- The next slide shows this in operation.

- Here are the first couple of steps in sorting.

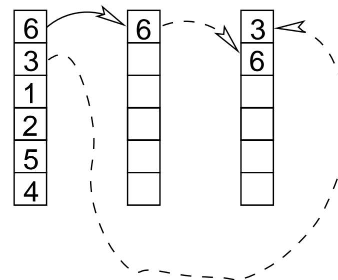


- In the first, 1 is found to be the smallest element and copied to the end (in this case its also the first spot in the array since the auxilliary array starts out empty) of the auxilliary array.
- In the second step, 2 is the smallest remaining element in the array, and is copied to the current end position in the auxilliary.

Insertion sort

- In insertion sort we do the following:
  1. We take elements from the array that is neing sorted and we *insert* them into the correct place in the auxilliary array. (This typically requires moving other elements to make room).
  2. One all the elements have been inserted into the auxilliary array they are copied back into the original array.
- The next slide shows this in operation.

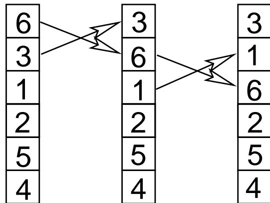
- First 6 is put into the auxilliary array in order — this is easy since it is the first element.



- The we put 3 into the array. To do this we need to push 6 down.
- Next we will insert 1 into the array — to do that we'll need to move both 3 and 6 down.

Bubble sort

- Bubble sort repeatedly compares adjacent members of the array, swapping elements if they aren't in order.



- In the first step, 3 and 6 are swapped.
- Next 1 and 6 are swapped.
- Lower values "bubble up" through the array.

Implementing insertion sort

- For insertion sort we take each element from the list and put it into the auxilliary.
- We put it in place in order.
- To find the right place we have to look through the auxilliary list.
- Once we have found the right place, we have to move all the remaining items in the auxilliary list down to make room.
- Before we run the code on the next slide, we have to set up the auxilliary array so that every item has a large value.
- After the code on the next slide has run, we have to copy the values from the auxilliary array back into the original array.


```
for(i = 0; i < size; ++i){

// Look through the auxilliary

  for(j = 0; j < size; ++j){

    // until the element of the array is smaller than an element of the auxilliary.

    if(a[i] < aux[j]){

      // then move all the remaining elements in the auxilliary down to make room

      for(k = size-1; k >= j; --k){
        aux[k] = aux[k-1];
      }

       // and copy the element of the original array across

       aux[j] = a[i];

      // At this point we don't need to look through the auxilliary array any more.

      j = size;
    }
  }
}
```


- You can try out insertion sort (and the other kinds of sort that we are talking about) — grab the file `sort.cpp` from the webpage for Unit V.

Summary

- This lecture discussed two things.
- First it considered the `switch` statement.
- Then we considered how to sort things.
  - In particular, we looked at blort sort, selection sort, insertion sort and bubble sort.
- We looked at the implementation of insertion sort.
- Next time we'll look in detail at how to program the other sorts.