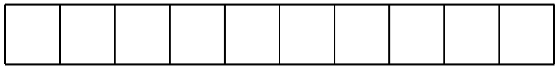
CS1007 lecture #10 notes

tue 8 oct 2002

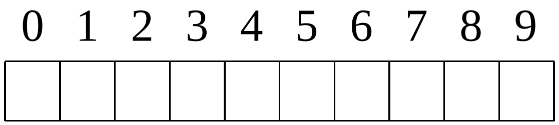
- NEWS
  - error quiz # 1, question 1d
  - if you got the question WRONG, bring the quiz to class on thu 10 oct for a one-time regrade
- arrays (one-dimensional)
- finding array minimum and maximum
- sorting
- big-Oh
- 2-dimensional arrays (preview)
- reading: *ch 5.1-5.7*

arrays (1).

- used to associate multiple instances of the same type of variable
- the "`[]`" indicates it's an *array*
- we can have arrays of anything (i.e., other data types)
- one example we've already used is `String[]`, which is an array of `String`...
- visualize an array as a sequence of boxes, contiguous in the computer's memory, where each box stores one instance of the type of data associated with that array:



- the boxes are numbered, starting with 0 and ending with the length of the array less one; each number is called an *index*
- the *indices* for an array of 10 items can be visualized like this:



arrays (2).

- to use an array, first you must declare it:

  ```
  int[] A;
  ```

- then you must instantiate it:

  ```
  A = new int[10];
  ```

- or you can do both of these in one step:

  ```
  int[] A = new int[10];
  ```

- then you can access its elements:

  ```
  A[4]
  ```

  (index=4, which is the 5th item in the array...)

- you can use this accessed item just like any single data element of that type, in this case an `int`
- the number of items in the array is the variable `A.length`

arrays (3).

- here’s an example that stores in an array 5 random numbers between 0 and 100:

```
public class ex10a {
  public static void main( String[] args ) {
    int[] A = new int[5];
    for ( int i=0; i<A.length; i++ ) {
      A[i] = (int)(Math.random()*100);
    }
    for ( int i=0; i<A.length; i++ ) {
      System.out.println( "i["+i+"]="+A[i] );
    } // end for i
  } // end of main()
} // end of class ex10a
```

arrays — finding the minimum.

```
public class ex10b {
  public static void main( String[] args ) {
    int[] A = new int[5];
    for ( int i=0; i<A.length; i++ ) {
      A[i] = (int)(Math.random()*100);
    }
    int min = A[0];
    for ( int i=1; i<A.length; i++ ) {
      if ( A[i] < min ) {
        min = A[i];
      }
    } // end for i
    System.out.println( "the minimum is: " + min );
  } // end of main()
} // end of class ex10b
```

arrays — finding the maximum.

```
public class ex10c {
  public static void main( String[] args ) {
    int[] A = new int[5];
    for ( int i=0; i<A.length; i++ ) {
      A[i] = (int)(Math.random()*100);
    }
    int max = A[0];
    for ( int i=1; i<A.length; i++ ) {
      if ( A[i] > max ) {
        max = A[i];
      }
    } // end for i
    System.out.println( "the maximum is: " + max );
  } // end of main()
} // end of class ex10c
```

sorting (1).

- sorting is one of the classic tasks done in computer programming
- the basic idea with sorting is to rearrange the elements in an array so that they are in a specific order — usually ascending or descending, in numeric or alphabetic order
- we will discuss 4 sorting algorithms (i.e., methods for sorting):
  - blort sort
  - insertion sort
  - selection sort
  - bubble sort

sorting (2).

- some sorts require an extra “auxiliary” array during sorting
  - the elements are moved from the original array into the auxiliary array, one at a time
  - at the end of the sort, the auxiliary array contains all the elements in sorted order
  - the final step is to copy the elements from the auxiliary array back into the original array
  - insertion and selection sorts can be done this way
- some sorts do not use an auxiliary array during sorting, but just move the elements around within the original array
  - these sorts involve the use of a swap() function, to switch the locations of two entries in the array
  - insertion and selection sorts can be done this way too
  - blort and bubble sorts are always done this way

swap.

- most sorts use a utility method called `swap()` to swap two elements in an array or Vector
- the methodology works like this
  - given two variables A and B, you want to switch the values so that the value of A gets the value of B and vice versa
  - you can't just simply copy one to the other and then vice versa because you'll lose the first value you copy to, so you need a temporary variable
  - here's the steps:
    1. $\boxed{\text{temp}} \leftarrow \boxed{\text{A}}$
    2. $\boxed{\text{A}} \leftarrow \boxed{\text{B}}$
    3. $\boxed{\text{B}} \leftarrow \boxed{\text{temp}}$
- example code: in `sorts.java`, the `swap()` method

blort sort.

- blort sort is the “fun but stupid” sort:
  1. check to see if the Vector is in sorted order
  2. if it is, then blort sort is done
  3. if it isn’t, then randomly permute the elements being sorted and loop back to the first step
- example code: three methods, in `sorts.java`:
  - `isSorted()`
  - `permute()`
  - `blortSort()`

insertion sort.

- insertion sort can use an *auxiliary* array to store the sorted elements temporarily
  1. it takes elements one at a time from the front the array being sorted and *inserts* them in sorted order into the auxiliary array
  2. it copies the content of the auxiliary array back into the original array, in sorted order
- insertion sort can also use an auxiliary array *conceptually*, by partitioning the array into a "sorted" section (i.e., the auxiliary array) and an "unsorted" section
- the second version is shown in `sorts.java`:
  – `insertionSort()`
- it uses a utility method for finding the smallest element:
  – `findMin()`

selection sort.

- selection sort can also use an *auxiliary* array to store the sorted elements temporarily
    1. it *selects* elements one at a time in sorted order from the array being sorted and appends them to the end of the auxiliary array
    2. it copies the content of the auxiliary array back into the original array, in sorted order
- selection sort can also use the auxiliary array *conceptually*, by partitioning the array into a “sorted” section (i.e., the auxiliary array) and an “unsorted” section
- the second version is shown in `sorts.java`:
    – `selectionSort()`
- it also uses the utility method for finding the smallest element:
    – `findMin()`

bubble sort.

- bubble sort repeatedly performs pairwise comparisons with neighboring elements in the array
- bubble sort always performs the number of passes equal to the size of the array minus 1
- one version of the method is shown in `sorts.java`:
  - `bubbleSort()`

big-oh notation.

- how do we choose the best sorting algorithm?
- we look at the number of comparisons made
- let's examine `findMin()`
- the number of comparisons made is equal to the length of the array
- this is referred to as $O(n)$, where:
  - n is the length of the array
  - O is called "big-oh" and means *on the order of n*
- as in, the run-time of the `findMin()` is on the order n
- in other words, as n gets bigger, `findMin()` takes longer to run
- and, as n gets smaller, `findMin()` takes less time to run

two-dimensional arrays (preview).

- arrays of arrays
- also called a two-dimensional array
- two-dimensional arrays are declared like this:
  `char[][] a2;`
- and instantiated like this (for example for a 5x5 array):
  `a2 = new char[5][5];`
- the first dimension is called *row*
- the second dimension is called *column*
- so the element in the i-th row and the j-th column is accessed like this:
  a2[i][j]
- example code is in `arr.java`