## cis1.5 spring2009 lecture V.2

today we are going to talk about...
sorting algorithms

- algorithms
- sorting
  - blort sort
  - selection sort
  - insertion sort
  - bubble sort

## what is an **algorithm**?

- “a step-by-step sequence of instructions for carrying out some task”
- examples of algorithms outside of computing:
  - cooking recipes
  - dance steps
  - proofs (mathematical or logical)
  - solutions to mathematical problems
- in computing, algorithms are synonymous with *problem solving*
- *How to Solve It*, by George Polya
  1. understand the problem
  2. devise a plan
  3. carry out your plan
  4. examine the solution
- example: find the tallest person in the class

## analysis of algorithms

- often, there is more than one way to solve a problem, i.e., there exists more than one algorithm for addressing any task
- some algorithms are better than others
- which *features* of the algorithm are important?
  - speed (number of steps)
  - memory (size of work space; how much scrap paper do you need?)
  - complexity (can others understand it?)
  - parallelism (can you do more than one step at once?)
- **Big-Oh** notation
  - $O(N)$ means solution time is proportional to the size of the problem $(N)$
  - $O(log_2 N)$ means solution time is proportional to $log_2 N$

## classic algorithm examples: searching and sorting

- *sequential* search
- *binary* search
- search the Manhattan phone book for “Al Pacino”:
  - how many *comparisons* do you have to make in order to find the entry you are looking for?
  - *equality* versus *relativity*—which will tell you more? which will help you solve the problem more efficiently?
  - can you take advantage of the fact that the phone book is in *sorted* order? (i.e., an “ordered list”)
  - what would happen to your algorithm if the phone book were in random order?

## sorting

- 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
  - selection sort
  - insertion sort
  - bubble sort
- 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 are this type

- 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
  - blort and bubble sorts are this type

## swap

- most sorts use a utility function called `swap()` to swap two elements in an array
- 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 (this should look familiar—it is similar to the `swap()` function we looked at earlier in the term when studying reference parameters):

```
int myArray[LENGTH];
.
.
.
// declare function swap(), to swap two entries in array
void swap( int a, int b ) {
  int tmp;
  tmp = myArray[a];
  myArray[a] = myArray[b];
  myArray[b] = tmp;
  return;
} // end of swap()
```

## blort sort

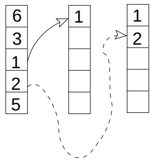
- blort sort is the “fun but stupid” sort
- here is the basic *algorithm*:
  1. check to see if the array is in sorted order
  2. if it is, then blort sort is done
  3. if it is not, then randomly permute the elements being sorted (i.e., mix them up) and loop back to the first step

## selection sort

- selection sort uses an *auxiliary* array to store the sorted elements temporarily
  1. first, 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. then, it copies the content of the auxiliary array back into the original array, in sorted order
- below is an illustration of the first two steps:

  in the first step, $\boxed{1}$ is selected as the smallest element in the array and is copied to the auxiliary array in the “last” spot (in this case, it’s also the first spot since the auxiliary array starts out empty)

  in the second step, $\boxed{2}$ is selected as the smallest (remaining, i.e., unsorted) element in the array and is copied to the auxiliary array in the last spot

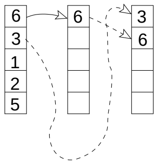


## insertion sort

- insertion sort uses an *auxiliary* array to store the sorted elements temporarily
  1. first, 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. then, it copies the content of the auxiliary array back into the original array, in sorted order
- below is an illustration of the first two steps:

  in the first step, $\boxed{6}$ is selected as the first unsorted element in the array and is inserted to the auxiliary array in “sorted” order (in this case, it’s just the first spot since the auxiliary array starts out empty)

  in the second step, $\boxed{3}$ is selected as the next unsorted element in the array and is inserted to the auxiliary array in “sorted” order (in this case, that means moving the $\boxed{6}$ down in the auxiliary array to make room for the $\boxed{3}$)

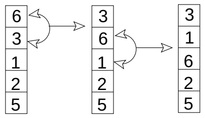


## bubble sort

- 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
- below is an illustration of the first two steps:

  in the first step, $\boxed{6}$ is swapped with $\boxed{3}$

  in the second step, $\boxed{1}$ is swapped with $\boxed{6}$



## run time

- one of the things we do when comparing algorithms in computer science is examine the *run time*
- this is the relative amount of time it takes for the program to run
- it is usually calculated based somehow on the amount of data the program is dealing with
- in the case of sorting algorithms, the *run time analysis* is based on the number of elements in the array being sorted
- if we call the number of elements in the array being sorted $N$, then we want to determine, for each algorithm, the amount of run time — also called *order* — relative to $N$

## run time analysis of sorting algorithms

- blort sort — cannot compute, since the number of passes made is not predictable
  in the *best case*, only one pass through the array is made, in the case where the array is in sorted order to begin with
  in the *worst case*, the number of passes is infinite...
- selection sort — order $N^2 = O(N^2)$
  because there is one pass made for each element in the array, i.e., as each element is shifted from the array to be sorted into the auxiliary array, and for each pass, the algorithm looks through the array to find the smallest element to select (which takes $O(N)$)
- insertion sort — order $N^2 = O(N^2)$
  because there is one pass made for each element in the array, i.e., as each element is shifted from the array to be sorted into the auxiliary array (same as selection sort), and for each pass, the algorithm looks through the auxiliary array to find a position for the new element (which takes $O(N)$)
- bubble sort – order $(N-1)^2 = O((N-1)^2)$
  because there is one pass made for each element in the array minus 1, and for each pass, the algorithm compares each element in the array to its neighbor, starting with the first

element in the array and ending with the second to last element (which takes $O(N-1)$)