cis1.5 spring2009 lecture V.3

today we are going to talk about...

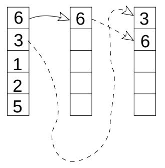
- finish up sorting algorithms from last time
  - insertion sort
  - bubble sort
- run time analysis
- searching algorithms

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 $\boxed{3}$)

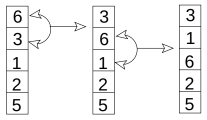


bubble sort

- 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 times the algorithm runs is not predictable
  in the *best case*, the algorithm runs only once, in the case where the array is in sorted order to begin with
  in the *worst case*, the algorithm runs infinitely...
- selection sort — order $N = O(N)$
  because the algorithm runs once for each element in the array, i.e., as each element is shifted from the array to be sorted into the auxiliary array
- insertion sort — order $N = O(N)$
  because the algorithm runs once 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)
- bubble sort – order $N - 1 = O(N - 1)$
  because the algorithm runs once for each element in the array minus 1 — we are always comparing each element to the one next to it, starting with the first element in the array and ending with the second to last element (which is compared to the last one); thus we only need to run the algorithm one less time than the number of elements in the array

searching algorithms

- often, when you have data stored in an array, you need to locate an element within that array
- this is called **searching**
- typically, you search for a *key* value (simply the value you are looking for) and return its *index* (the location of the value in the array)
- as with sorting, there are many searching algorithms...
- we'll study the following:
  - sequential or linear search
    * linear search on unsorted data
    * linear search on sorted data
  - binary search

linear search on UNSORTED DATA

- linear search simply looks through all the elements in the array, one at a time, and stops when it finds the key value
- this is inefficient, but if the array you are searching is not sorted, then it may be the only practical method
- example code:

```
int linearSearch( int key ) {
  for ( int i=0; i<NUM_DICE; i++ ) {
    if ( key == dice[i] ) {
      return( i );
    }
  } // end for i
  return( -1 );
} // end of linearSearch()
```

linear search on SORTED data

- if the array you are searching IS sorted, then you can modify the linear search to stop searching if you have looked past the place where the key would be stored if it were in the array
- example code:

```
int linearSearch2( int key ) {
  for ( int i=0; i<NUM_DICE; i++ ) {
    if ( key == dice[i] ) {
      return( i );
    }
    else if ( key < dice[i] ) {
      return( -1 );
    }
  } // end for i
  return( -1 );
} // end of linearSearch2()
```

binary search

- binary search is much more efficient than linear search, ON A SORTED ARRAY
- *binary search CANNOT be used on an UNSORTED array!*
- it takes the strategy of continually dividing the search space into two halves, hence the name *binary*
- say you are searching something very large, like the phone book... if you are looking for one name (e.g., "Gilligan"), it is extremely slow and inefficient to start with the A's and look at each name one at a time, stopping only when you find "Gilligan". but this is what linear search does.
- binary search acts much like you'd act if you were looking up "Gilligan" in the phone book
  - you'd open the book somewhere in the middle, then determine if "Gilligan" appears before or after the page you have opened to
  - if "Gilligan" appears after the page you've selected, then you'd open the book to a later page
  - If "Gilligan" appears before the page you've selected, then you'd open the book to an earlier page

- you'd repeat this process until you found the entry you are looking for or until you realized that the entry wasn't in the array (phone book)
- example code:

```
int binarySearch( int key ) {
  int lo = 0, hi = NUM_DICE-1, mid;
  while ( lo <= hi ) {
    mid = ( lo + hi ) / 2;
    if ( key == dice[mid] ) {
      return( mid );
    }
    else if ( key < dice[mid] ) {
      hi = mid - 1;
    }
    else {
      lo = mid + 1;
    }
  } // end while
  return( -1 );
} // end of binarySearch()
```