CS 211
Search & Sort

Searching & Sorting

Searching and Sorting

Searching means that we have some collection of data, and we seek a particular value that *might* be contained within our collection.

- We provide a **key**, some piece of information that will let us identify the desired piece of information.

→ example: we have an array of Person objects; please find the person named "Bob". ("Bob" is the key; the array is the collection; a reference to the person, or an index for later use with the array, could be returned).

Searching and Sorting

Sorting means that we will take some collection of data, and re-organize it so that we contain the same pieces of data, but in a predictable ordering.

→ usually some numeric or alphabetic ordering, considering a specific piece of the values.

→ examples:

sorting an array of Person objects by name
sorting an array of Person objects by age
sorting an ArrayList of Books by ISBN #.

Basic Search: Linear

Assume we've got some linear data structure, like a list:

A linear search starts at the beginning, and checks each element in turn, until a match is found.

→ if no match is found, all elements were inspected – slow!

→ doesn't require any ordering of the data (unsorted is ok).

```
public static int getIndex (int[] xs, int key) {
    for (int i = 0; i<xs.length; i++) {
        if (xs[i] == key) return i;
    }
    return -1;
 }
```

Better Searching...

In order to search through data better, we need to have more information about the listing of data.

We want the data to be sorted by some property (such as increasing numeric value)
→ we can know more about "the rest of the list"
→ we know about elements before/after this one

→ so let's learn how to sort data. (We'll stick with arrays of ints for now).

Basic Sort: Bubble

compare consecutive pairs
→ indexes 0&1, 1&2, 2&3, etc.

swap, 'bubbling' larger values towards top.

repeat N times, done.
→ each complete pass, one more 'largest' number is guaranteed to be bubbled all the way to the top where it belongs.

Basic Sort: Bubble

```
public static void bubbleSort(int[]xs){
   for (int n=0; n<xs.length; n++){
      for (int i=1; i<xs.length; i++){
         if (xs[i-1]>xs[i]){
            int temp = xs[i-1];
            xs[i-1] = xs[i];
            xs[i] = temp;
} } } }
```

Bubble Sort is Horrible!

Bubble sort is painfully inefficient!

It only moves things towards their correct place one step at a time

For a list with n elements in it, we can expect the time taken to be on the order of n^2. (As n increases linearly (one by one), the execution time increases quadratically).

Its "average case" running time is equivalent to its "worst case" running time! (It rarely does better than the worst it could do)

About the only thing going for bubble sort is that it is easy to understand.
→ Other than that, at least it identifies sorted lists quickly.

Basic Sort: Bubble (slightly faster)

```
public static void bubbleSort(int[]xs){
    boolean tryAgain = true;
    while (tryAgain) {
        tryAgain = false;
        for (int i=1; i<xs.length; i++){
            if (xs[i-1]>xs[i]){
                tryAgain = true;
                int temp = xs[i-1];
                xs[i-1] = xs[i];
                xs[i] = temp;
}   }   }   }
```

quits as soon as no values needed relocations for an entire pass.

Better Searching: Binary Search

Binary Search is the "phonebook" search, or "dictionary" search

If you are looking for a word in the dictionary, you know the words are sorted alphabetically.

- So you open up to the middle page, figure out which half has your word, and then pretend that's the whole dictionary.
- You then split that part in half, again figuring out which half contains your word.
  → If you find your word, success!
  → If you find that your word isn't between two that it had to be, then it's not present. Failure! (value isn't in your list).
- Each attempt removes ½ the search space – efficient!
- This can be a recursive approach.

Binary Search

```
public static int binarySearch(int[]xs, int key)
    {  return bs(xs,key, 0, xs.length-1);  }


private static int bs(int[] xs, int key, int low, int high){
    int mid = low + ((high-low)/2);

    if (low>high) return -1;          // key not found.
    if (xs[mid]==key) return mid;       // key found.

    //key in lower portion.
    if (key<xs[mid])  return bs(xs,key, low,   mid-1);
    // key in upper portion: (xs[mid]<key)
    else              return bs(xs,key, mid+1, high );
}
```

Binary Search Performance

Binary Search can run much faster than linear search. For a list of n numbers, binary search will perform no more than $\log_2(n)$ calls. Each method call is at most a few comparisons.

examples of scale:

n=10:	$\log_2(n) = 3.3$
n=100:	$\log_2(n) = 6.6$
n=10,000:	$\log_2(n) = 13.2$
n=1,000,000:	$\log_2(n) = 19.9$

So having sorted data is really, really worth the effort!

One More Sort: MergeSort

MergeSort is a **divide and conquer** algorithm.

- It successively splits the list in half, until each sublist is size 1.
- Then the two halves are mergeSorted (they will be individually sorted now).
- The two sublists are then merged together, picking the smallest elements from the start of each list until all elements have been added back to our full (sorted) list.
- MergeSort can take significant space if implemented naively (we create many sublists).

MergeSort Code – beginning

```
public static void mergeSort (ArrayList<Integer> xs){

        // size 1 - already sorted.
        if (xs.size()<=1) { return ;}

        //split into left and right
        int mid = xs.size()/2;
        List<Integer> left  = new ArrayList<Integer>();
        List<Integer> right = new ArrayList<Integer>();

        //put the first half in left.
        for (int i=0; i<mid; i++){  left.add(xs.remove(0)); }

        //put the rest in right.
        while (xs.size()>0) { right.add(xs.remove(0)); }

        …
```

MergeSort Code - continued

```
   …
   //recursively sort left and right.
   mergeSort(left);
   mergeSort(right);


   //merge the two sorted lists together, keeping the result sorted.
   while (left.size()>0 || right.size()>0){

      //if either list is empty, just add from other.
      if      (left .size()==0) xs.add(right.remove(0));
      else if (right.size()==0) xs.add(left .remove(0));

      //if neither is empty, choose lowest elt,  add.
      else if (left.get(0) <= right.get(0)) xs.add(left .remove(0));
      else                                  xs.add(right.remove(0));
}    }
```

Other Sorts

Selection Sort: very basic sort that successively finds the smallest value not yet sorted, and swaps it with the leftmost unsorted location.

QuickSort: choose an arbitrary value (the 'pivot'); put all smaller values to the left, all larger values to the right, and the pivot in the middle. Call quickSort on each half. Done!

even more sorting algorithms: **Shell Sort**, **BogoSort**, **TreeSort**, …

Summary

Searching allows us to find specific pieces of data in our collections.

Sorting organizes our data, also making searching must faster.

We only considered linear structures, but searching and sorting is also possible/desirable on non-linear structures (e.g., trees).

There are simpler search/sort algorithms, but they tend to be inefficient. So we have many more algorithms that are more efficient by doing more complicated/ intelligent manipulations on our data.
→ we compare algorithms by talking about how long basic operations take, relative to the # of elements (n).

→ CS 310 will cover these "computational complexities" in much greater detail, for each new data structure.