08.02 Virtual Lecture Notes

For a binary search algorithm to work, the values in the array must be sorted. So that is where we will start. Once sorted, the search can begin by looking at the middle element in the array. If that middle element matches what we want to find, then we are done. If not, we have a choice to make. The next place to look will either be in the first half or the second half of the array, but not both. By comparing the element we want to find to the middle element, we will know if it is equal to, less than, or greater than the middle element. If it is less, we continue the search in the first half of the array and ignore the second half. If it is more, we go to the second half. Once we have picked the half to search, we repeat the process—each time checking the middle element against the value we're looking for and splitting the array in half until we either find the match or run out of elements to search.

In short, for the binary search algorithm you keep splitting the array of items in half, over and over again. It works much faster than sequential search.

Using the event assignment example, we start by adding methods to sort the roster by the person's name and one for location. Look at the `TestAssignment2.java` program and locate the two search methods. Use the print statements for debugging to ensure the values are sorted as you'd expect.

Now we can write a binary search method to find if a particular person is in the list.

```
public static int binarySearchPerson(Assignment[] r, String
toFind)
{
   int high = r.length;
   int low = -1;
   int probe;

   while( high - low > 1 )
   {
      probe = ( high + low ) / 2;
      if( r[probe].getPerson().compareTo(toFind) > 0)
         high = probe;
      else
         low = probe;
```

```
    }
    if( (low >= 0) && (r[low].getPerson().compareTo(toFind) == 0
))
        return low;
    else
        return -1;
}
```

Notice the code splits the problem in half each time through the loop, which means it will work in less time than the sequential search algorithm. The worst case for a sequential search is having to traverse the entire array before determining if a value is present. If the array had 100,000 elements, every one of them may need to be checked. Since the binary search can eliminate half of the array at a time, it is extremely fast. For small arrays, you probably will not notice the difference, but for larger ones, you will.

The binary search is super-fast at locating one instance of a value. But what if you want to find all? You don't know if the one found is the first or last of its kind. You do know they will be all together though, since the array was sorted. To find multiple matches, we'll modify the algorithm a bit. The binary search will be used to find the location of one match. Then a linear search will check both directions from that point, until you find values that do not match, which indicate that you should stop.

For example, imagine you have a sorted array of integers and find a match for the value 5. Then search left until no more fives are found, and then look to the right until the value no longer matches 5. Once the low and high index values for the value 5 are found, you can print or access the records within that range.

For the search by location, let's look for multiple matches. Remember that it is a modification of the binary search algorithm and will not run as fast as the regular binary search, but will still be faster than a sequential search.

```
public static void binarySearchLoc(Assignment[] r, String toFind)
{
    int high = r.length;
    int low = -1;
    int probe;

    while( high - low > 1 )
    {
        probe = ( high + low ) / 2;

        if(r[probe].getLocation().compareTo(toFind) > 0)
```

```
          high = probe;
        else
        {
          low = probe;
          if( r[probe].getLocation().compareTo(toFind) == 0)
          {
            break;
          }
        }
    }

    if( (low >= 0) && (r[low].getLocation().compareTo(toFind) ==
0 ))
    {
      linearPrintLoc(r, low, toFind);
    }
    else
      System.out.println("NOT found: " + toFind);
}
```

In the search method, notice a check was added to the `while` loop to break the loop once a match is found. Then the `if` after the loop calls the `linearPrintLoc` method if a match was found. If a match isn't found, an appropriate message is printed.

Let's explore the `linearPrintLoc` method.

```
public static void linearPrintLoc(Assignment[] r, int low, String
toFind)
{
    int i;
    int start = low;
    int end = low;

    // find starting point of matches
    i = low - 1;
    while((i >= 0) && (r[i].getLocation().compareTo(toFind) ==
0))
    {
      start = i;
      i--;
    }
    // find ending point of matches
    i = low + 1;
    while((i < r.length) && (r[i].getLocation().compareTo(toFind)
== 0))
    {
      end = i;
      i++;
    }
    // now print out the matches
    for(i = start; i <= end; i++)
      System.out.println(r[i]);
}
```

We want to print all of the elements matching the desired location in the order they appear in the array. First, we use two `while` loops to determine the start and end point index values; then the `for` loop will print them up in the order that they are in the array.

To recap, the binary search is great to quickly discover if a value is in an array, but needs modification if you want to find all the matching values in an array. Also, the array must be sorted, otherwise the binary search will not work.

And that is a binary search in action! Be sure to open and run the `TestAssignment2` program. To watch the search work, try adding print statements to the search methods to see the values for the probe, low, and high variables. The binary search is quite effective since half of the list can be eliminated with each comparison.

Print