Computer Science 252
Problem Solving with Java
The College of Saint Rose
Spring 2016

Topic Notes: Searching and Sorting

Searching

We all know what *searching* is – looking for something. In a computer program, the search could be:

- Looking in a collection of values for some specific value (where is the 17 in this array of `int`?).
- Looking for a value with a specific property (which object on the canvas contains the location where I clicked the mouse?).
- Looking for a record in a database (what is the tax history for the last four years for the taxpayer with SSN 101-11-1009?).
- Searching for text in some document or collection of documents (what web pages contain the text “searching and sorting algorithms?”).
- What known amino acid sequences best match this sequence gathered from proteins in a given virus?

We have done some searching this semester. Remember the test to see which image was selected in the “DragStudents” example.

```
for (int imageNum = 0; imageNum<heads.length; imageNum++) {
  if (heads[imageNum].contains(point)) {
    selectedHead = heads[imageNum];
    dragging = true;
    selectedHead.sendToFront();
  }
}
```

We have to search through our collection of objects (`VisibleImages`) to see which one, if any, contains the point.

How do we know that we’re done searching? In this case, we keep looking until we get to the end of our array. But in many cases, we need only search until we find the first matching entry.

Let's try to get some idea of how much "work" it takes for us to get an answer. As a rough estimate of work, we will count how many times we call the `contains` method of a `VisibleImage`.

If we have n `VisibleImages`, how many calls to the `VisibleImage contains` method will we have to make before we know the answer? In this case, it's once per entry in the array, so n times.

In some other cases, it depends on how quickly we find the answer. If none of the `VisibleImages` contains the point at all, we need to check all n before we know the answer. If one does contain the point, we can stop as soon as we find the first one that contains it. It might be the first, it might be the last – we just don't know. Assuming that there's an equal probability that the `VisibleImage` that contains the point is at any of the n positions, we have to examine, on average, $\frac{n}{2}$ `VisibleImages`.

In this case, we can't do any better. Perhaps if we decided to check in some other order rather than always examining the first, then the second, and so on.

We are searching in an array, where we have the option to look at any element directly. We will consider an array of `int`, though most of what we discuss applies to a wider range of "searchable" items.

A method to do this:

```
/*
 * Search for num in array.  Return the index of the number, or
 * -1 if it is not found.
 */
int getIndexOfNum(int[] array, int num) {
  for (int index = 0; index < array.length; index++) {
    if (array[index] == num) {
      return index;
    }
  }
  return -1;
}
```

The procedure here is a lot like the searches we have seen. We have no way of knowing that we're done until we either find the number we're looking for, or until we get to the end of the array. So again, if the array contains n numbers, we have to examine all n in an unsuccessful search, and, on average, $\frac{n}{2}$ for a successful search. We could instead search from the end to the front, and we would have no reason to believe that we'd do any better or worse, on average.

Now, suppose the array has been sorted in ascending order.

Well, we can do the same type of search – start at the beginning and keep looking for the number. In the case of a successful search, we still stop when we find it. But now, we can also determine that a search is unsuccessful as soon as we encouter any number larger than our search number. Assuming that our search number is, on average, is going to be found near the median value of the array, our unsuccessful search is now going to require that we examine, on average, $\frac{n}{2}$ items. This sounds great, but in fact is not a really significant gain, as we will see. These are all examples of a *linear search* – we examine items one at a time in some linear order until we find the search item or until we can determine that we will not find it.

But there is a better way. To get the intuition for the next way to search for a number, think back to your favorite number guessing game. I pick a number between 1 and 100 and you have to guess what it is. The game usually goes something like this:

```
Me: Guess my number.
You: 50.
Me: Too High.
You: 25.
Me: Too Low.
You 37.
Me: Too High.
You 31.
Me: That's right.
```

If you know that there is an order – where do you start your search? In the middle, since then even if you don't find it, you can look at the value you found and see if the search item is smaller or larger. From that, you can decide to look only in the bottom half of the array or in the top half of the array. You could then do a linear search on the appropriate half – or better yet – repeat the procedure and cut the half in half, and so on. This is a *binary search*. It is an example of a *divide and conquer* algorithm, because at each step, it divides the problem in half.

A Java method to do this:

```
/*
 * Binary Search for num in array.
 */
int getIndexOfNum(int[] array, int num) {
  int mid;
  int left = 0;
  int right = array.length - 1;
  while (left < right) {
    mid = (low + high) / 2;
    if (array[mid] == num) {
      // num is same as middle number
      return mid;
    } else if (num < array[mid]) {
      // num is smaller than middle number
      right = mid - 1;
    } else {
      // num is larger than middle number
```

```
        left = mid + 1;
      }
    }
    return -1;
  }
```

How many steps are needed for this?

- Each time, we cut the part of the array we still need to search in half.
- How many times can divide number in half before you get to 1?
- If you start with n, you divide to get $\frac{n}{2}$ then $\frac{n}{4}$, $\frac{n}{8}$, ... and eventually get 1.
- Let's suppose that $n = 2^k$, then divide to 2^{k-1}, 2^{k-2}, 2^{k-3}, ..., 2^0 = 1; divide k times by 2.
- In general, we can divide n by 2 at most $\log_2 n$ times to get down to 1.

So how much better is this, really? In the case of a small array, the difference is not really significant. But as the size grows...

Search# elts	10	100	1000	1,000,000
linear	10	100	1000	1,000,000
binary	8	14	20	40

That's pretty huge. Even if you think about the search really needing on average $\frac{n}{2}$ steps, for the 1000-element case, the binary search is still winning 500 to 20. The logarithmic factor is really important.

We can see this better by looking at graphs of n *vs.*$\log n$ and n. The difference is large, and gets larger and larger as n gets larger. Even if we multiply by constant factors in an attempt to make the $\log n$ graph as large as the n graph, there will always be a value of n large enough that the scaled function for n will be larger than the scaled function for $\log n$. More on this later.

Sorting

We'll now look at *sorting*. This is useful in many contexts, one of which is to be able to use binary search. As we will see, sorting takes a fair amount of time, but if we are going to be searching a large array a lot, the savings obtained by using binary search over linear will more than make up for the cost of sorting the array once.

Suppose our goal is to take a shuffled deck of cards and to sort it in ascending order. We'll ignore suits, so there is a four-way tie at each rank.

Describing a sorting algorithm precisely can be difficult. Let's consider a couple.

1. selection sort

2. insertion sort

Selection Sort

First, we will look at this procedure:

- Search for the smallest card, and move it to the front of the deck.
- Search for the next smallest card, and move it to the second position in the deck.
- ...

What I have described is a form of a *selection sort* – at each step, we select the item that goes into the next position of the array, and put it there. This gets us one step closer to a solution.

```
public void selectionSort(int[] array) {
  for (int i = 0; i < array.length - 1; i++) {
    int smallestPos = i;
    for (int j = i+1; j < array.length - 1; j++) {
       if (a[j] < a[smallestPos) {
         smallestPos = j;
       }
    }
    int temp = array[smallestPos];
    array[smallestPos] = array[i]
    array[i] = temp;
  }
}
```

How long does this algorithm take? As we did with searching, we won't try to calculate an exact time, but we will estimate the cost by computing the number of comparisons done in sorting an array. We could alternately choose the to count the total number of "visits" to an array element, but the "shape" of the answer will be the same no matter which of these we compute.

Suppose the original array has n elements, where $n > 1$. Then it takes $n - 1$ comparisons to find the smallest element of the array (compare the first with the second, the largest of those with the third, etc.). In general, the number of comparisons needed to find the smallest element is one less than the number of elements to be sorted. Once this element has been put into the first slot of the array, we need to sort the remaining $n - 1$ elements of the array. By the argument above, it takes $n - 2$ comparisons to find the largest of these. We continue with successive stages taking $n - 3$, $n - 4$, all the way down to the last pass through when there are only two elements and it takes only 1 comparison. (Once we get down to 1 element there is nothing to be done.)

Thus it takes $S = (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1$ comparisons to sort a list of n elements. We can compute this sum by writing the list forwards and backwards, and then adding the columns:

```
   S = (n-1) + (n-2) + (n-3) + ... +   3   +   2   +   1
   S =   1   +   2   +   3   + ... + (n-3) + (n-2) + (n-1)
   -------------------------------------------------------
  2S =   n   +   n   +   n   + ... +   n   +   n   +   n   = (n-1)*n
```

Therefore $S = \frac{n^2-n}{2}$. The graph of this as n increases looks like n^2 – a parabola. Therefore, selection sort takes n^2 time, which is much worse than the behavior for the searching algorithms we saw last time.

Insertion Sort

The selection sort builds up the sorted list by finding the smallest and putting it into the first position, the sthe second smallest and putting it into the second position, etc., until the entire list is sorted.

Insertion sort takes a different approach. It builds up a sorted list by noticing that we can build a sorted list of size $n+1$ by taking a sorted list of size n and inserting the $n+1^{st}$ element in its correct position.

We will not look at this algorithm in great detail here. Like selection sort, insertion sort takes n^2 time.

Merge Sort

Our next sorting algorithm, the *merge sort*, proceeds as follows:

- First, our base case: If the array contains 0 or 1 elements, there is nothing to do. It is already sorted.
- If the array has two or more elements in it, we will break it in half, sort the two halves, and then go through and merge the elements.

The Java method to do it:

```
public void sort(int[] array) {
   // create tempArray for use in merging
   int[] tempArray = new int[array.length];
   mergeSort(array, 0, array.length-1, tempArray);
  }

/*
```

```
   * PRE: left and right are valid indexes of array.
   *      tempArray.length == array.length
   * POST: Sorts array from start to right.
   */
  public void mergeSort(int[] array, int left, int right, int[] tempArray)
    if (left < right) {
      int middle = (right + left) / 2;
      mergeSort(array, left, middle, tempArray);
      mergeSort(array, middle + 1, right, tempArray);
      merge(array, left, middle, right, tempArray);
    }
  }
```

The method `merge` takes the sorted elements in `array[left..middle]` and `array[middle+1..right]` and merges then together using the array `tempArray`, and then copies them back into `array`.

```
  /*
   * PRE: left <= middle <= right and left,middle,right are valid indices f
   *      tempArray.length == array.length
   *      array[left..middle] and array[middle+1..right] must both be sorte
   * POST: Merges the two halves (array[left..middle] and array[middle+1..r
   *       together, and array[left..right] is then sorted.
   */
  private void merge(int []array, int left, int middle, int right, int[] te
    int indexLeft = left;
    int indexRight = middle + 1;
    int target = left;

    // Copy both pieces into tempArray.
    for (int i = left; i <= right; i++) {
      tempArray[i] = array[i];
    }

    // Merge them together back in array while there are
    // elements left in both halves.
    while (indexLeft <= middle && indexRight <= right) {
      if (tempArray[indexLeft] < tempArray[indexRight]) {
        array[target] = tempArray[indexLeft];
        indexLeft++;
      } else {
        array[target] = tempArray[indexRight];
        indexRight++;
      }
      target++;
```

```
    }

    // Move any remaining elements from the left half.
    while (indexLeft <= middle) {
      array[target] = tempArray[indexLeft];
      indexLeft++;
      target++;
    }

    // Move any remaining elements from the right half.
    while (indexRight <= right) {
      array[target] = tempArray[indexRight];
      indexRight++;
      target++;
    }
  }
```

Again we'd like to count the number of comparisons necessary in order to sort an array of n elements. Unfortunately, the code shown above doesn't include any comparisons – all of the comparisons are in the `mergeRuns` method.

Even without looking at the code in `merge` we can estimate the number of comparisons made. If we are trying to merge two sorted lists, every time we compare two elements at the ends of the lists we will put one in its correct position. When we run out of the elements in one of the lists, we put the remaining elements into the last slots of the sorted list. As a result, merging two lists which have a total of n elements requires at most $n-1$ comparisons.

Suppose we start with a list of n elements. Let $T(n)$ be a function telling us the number of comparisons necessary to mergesort an array with n elements. As we noted above, we break the list in half, mergesort each half, and then merge the two pieces. Thus the total amount of comparisons needed are the number of comparisons to mergesort each half plus the number of comparisons necessary to merge the two halves. By the remarks above, the number of comparisons to do the final merge is no more than $n-1$. Thus $T(n) <= T(n/2) + T(n/2) + n - 1$. For simplicity we'll replace the $n-1$ comparisons for the merging by the even larger n in order to make it easier to see how to approximate this result. We have $T(n) = 2 \cdot T(n/2) + n$ and if we find a function that satisfies that equation, then we have an upper bound on the number of comparisons made during a mergesort.

Looking at our algorithm, no comparisons are necessary when the size of the array is 0 or 1. Thus T(0) = T(1) = 0. Let us see if we can solve this for small values of n. Because we are constantly dividing the number of elements in half it will be most convenient to start with values of n which are a power of two. Here we list a table of values:

Table of n vs. $T(n)$ goes here...

Notice that if $n = 2^k$ then $k = \log_2 n$. Thus $T(n) = n \cdot log_2 n$. In fact this works as an upper bound for the number of comparisons for mergesort even if n is not even. If we graph this we see that it grows much, much slower than the graph for a quadratic (for example, the one corresponding to the number of comparison for selection sort).

This explains why, when we run the algorithms, the time for mergesort is almost insignificant compared to that for selection sort.