SIENAcollege
Computer Science

Computer Science 210
Data Structures
Siena College
Fall 2017

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 (*e.g.*, where is the 17 in this array of `int`?).
- Looking for a value with a specific property (*e.g.*, which object in a graphics window contains the location where I clicked the mouse?).
- Looking for a record in a database (*e.g.*, what is the tax history for the last four years for the taxpayer with SSN 101-11-1001?).
- 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 spell was specified in the "SpellsArray" example.

```
spellnum = -1;
for (int spellIndex = 0; spellIndex < spells.length; spellIndex++) {
  if (spellName.equals(spells[spellIndex].getKey())) {
    spellnum = spellIndex;
    break;
  }
}
if (spellnum >= 0) {
  System.out.println(spells[spellnum].getValue());
}
else {
  System.out.println("Your wand doesn't know that one. It explodes. Bye
}
```

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

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

Let's consider 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 `equals` method of a `String` to compare the key.

If we have n `Associations` in the array, how many calls to the `String` `equals` method will we have to make before we know the answer?

It depends on how quickly we find the answer. If none of the `Associations` contains the matching key at all, we need to check all n before we know the answer. If one does contain a match for the key, we can stop as soon as we find the first one that matches it (the `break` statement in the loop achieves this). It might be the first, it might be the last – we just don't know. Assuming that there's an equal probability that the `Association` that contains the matching key is at any of the n positions, we have to compare, `Associations`, on average, $\frac{n}{2}$ times.

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.

We could still 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.

To understand a better way to search for a number, think back to the children's 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.
```

Why? The person trying to guess the number is trying to narrow down the possible range of values as quickly as possible. By guessing in the middle at each step, half of the range of values can be eliminated. This idea leads directly to a better way to search.

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;
  }
```

This algorithm is expressed very naturally in recursive form, which we will see in an upcoming example.

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...

Maximum Number of Comparisons				
array size	10	100	1000	1,000,000
linear search	10	100	1000	1,000,000
binary search	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.

See Example: BinSearch

`Comparable` Objects

If we are going to deal with `Objects` rather than primitive types for a binary search, we need a way to compare them. We need a more general analog of the `equals` method. We can write a method that compares an `Object` to another, like the `compareTo()` method of `String`s. However, there is no `compareTo` method in `Object`.

Fortunately, Java provides an interface that does exactly this, the `Comparable` interface. Any object that implements `Comparable` will have a `compareTo` method, so if we write our search (and next up, sorting) routines to operate on `Comparable`s, we will be all set.

We return to our previous example and consider the generic binary search methods.

See Example: BinSearch

Note the weird syntax in the headers of the generic methods:

```
public static <T extends Comparable> int search (T[] a, T val)
```

Unlike when we considered generic classes like `GenericPair` and `Vector`, this program does not specify a generic type for the class, just for the methods that use the generic type.

The `<T extends Comparable>` means that any class can be used for the type of the array and search element, as long as the array was declared and constructed as some type that implements the `Comparable` interface.

Several standard Java classes implement the `Comparable` interface, including things like `Integer`, `Double`, and `String`.

So we can write methods that expect objects that extend `Comparable`, and be guaranteed that an appropriate `compareTo` method will be provided.

Sorting

We'll now look at *sorting*. One of the many motivations for sorting data is that we need sorted data to be able to use a binary search. As we will see, sorting is a fairly expensive process, 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 that 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, but that precision is necessary if we're going to write code to do sorting.

We will consider a few algorithms:

- selection sort
- insertion sort
- merge sort

Selection Sort

Let's think about the first procedure as it would work to sort a shuffled stack of 100 cards with unique numbers in ascending order.

- Search for the smallest card among all 100, and move it to the front of the deck.
- Next, ignoring the that smallest card in the first position, search for the next smallest card among the 99 remaining cards, and move it to the second position in the deck.
- Next, ignoring the those two smallest cards in the first and second positions, search for the next smallest card among the 98 remaining, and move it to the third position in the deck.
- ...
- Next, ignoring the those 98 smallest cards in the first 98 positions, search for the next smallest card of the 2 remaining cards, and move it to the 99th position in the deck.
- The remaining card goes into the 100th position (it's the largest).

The procedure 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; j++) {
       if (array[j] < array[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 *comparison counting* – determining how many comparisons need to be done in sorting an array.

For an array with $n > 1$ elements, 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 an array of n elements. In case you aren't already familiar with the summation formula that can give us that value, let's see how 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, as you might already have known, $S = \frac{n^2-n}{2}$. The graph of this as n increases looks like n^2 – a parabola. Therefore, selection sort takes $O(n^2)$ time, which is much worse than the cost of the searching algorithms we saw earlier.

A recursive implementation of selection sort, but where each pass finds the largest element and places it at the end of the unsorted portion of the array, is found in this example:

See Example: SortingComparisons

Insertion Sort

The selection sort builds up the sorted array by finding the smallest (or largest) element and putting it into the first (or last) position, then the second smallest and putting it into the second (or second to last) position, etc., until the entire array is sorted.

Insertion sort takes a different approach. It builds up a sorted array using the fact that we can build a sorted array of size $n + 1$ by taking a sorted array 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 $O(n^2)$ time in most cases, though it has a best case behavior of $O(n)$ when the input array is sorted or nearly sorted.

An example recursive implementation is in the same example:

See Example: SortingComparisons

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.

A recursive implementation of this is also in the sorting comparisons example.

See Example: SortingComparisons

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

Again we'd like to count the number of comparisons necessary in order to sort an array of n elements with merge sort. This is complicated by the fact that the `mergeSortRecursive` method doesn't include any element comparisons at all – all of the comparisons are in the `merge` method.

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

Suppose we start with an array of n elements. Let $T(n)$ be a function telling us the number of comparisons necessary to perform merge sort on an array with n elements. As we noted above, we break the array in half, merge sort each half, and then merge the two (now sorted) pieces. Thus the total amount of comparisons needed are the number of comparisons to perform merge sort on 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:

n	$T(n)$
$1 = 2^0$	0
$2 = 2^1$	$2 \cdot T(1) + 2 = 2 = 2 \cdot 1$
$4 = 2^2$	$2 \cdot T(2) + 4 = 8 = 4 \cdot 2$
$8 = 2^3$	$2 \cdot T(4) + 8 = 24 = 8 \cdot 3$
$16 = 2^4$	$2 \cdot T(8) + 16 = 64 = 16 \cdot 4$
$32 = 2^5$	$2 \cdot T(16) + 32 = 160 = 32 \cdot 5$
...	...
$n = 2^k$	$2 \cdot T(\frac{n}{2}) + n = n \cdot k$

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 merge sort 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 on larger values of n, the time for merge sort is far smaller than that of selection sort.