Computer Science 211
Data Structures
Mount Holyoke College
Fall 2009

Topic Notes: Sorting

Searching and sorting are very common operations and are also important examples to demonstrate complexity analysis.

Searching

Before we deal with sorting, we briefly consider searching.

Linear Search

As you certainly know, a search is the method we use to locate an instance of a data item with a particular property within a collection of data items. The method used for searching depends on the organization of the data in which we are searching.

To start, we will assume we are searching for a particular value in an array of `int`.

The *linear search* is very straightforward. We simply compare the element we're looking for with successive elements of the array until we either find it or run out of elements.

```
public static int search (int[] elts, int findElt) {
     int index = 0;
     while (index < elts.length) {
         if (findElt == elts[index])
             return index;
         index++;
     }
     return -1;                      // Didn't find elt.
 }
```

Some properties of this linear search for an array of size n:

- On average, this will require $\frac{n}{2}$ compares if element is in the array.
- It required n compares if element not in array (worst case).
- Both are $O(n)$.

Note that we can very easily modify the search method to work on any array of `Object`s:

```
public static int search (Object[] elts, int findElt) {
     int index = 0;
     while (index < elts.length) {
         if (findElt.equals(elts[index]))
             return index;
         index++;
     }
     return -1;                      // Didn’t find elt.
 }
```

We can get away with this because all `Object`s are required to have an `equals` method, and this is the only comparison needed for a linear search.

Binary search

The linear search is the best we can do if we have no information about the ordering of the data in our array. However, if we have *ordered* data, we can use a *binary search*.

Here, we start by considering the middle element in the array:

- If the middle element is the search element, then we’re done.
- If the middle element smaller than search element, then we know the element, if it is in our array, can be found by a binary search of the bigger elements.
- If the middle element larger than search element, then we do a binary search of the smaller elements.

See Example:
`/home/jteresco/shared/cs211/examples/BinSearch`

Notice that we had to write a protected helper method to do the search recursively, since a user of this search shouldn’t need to specify a start and end in their method call. From their point of view, they should need only specify the array and the element to be located.

This is a classic example of a *divide and conquer* approach.

Each recursive call will lead to at most two compares.

What is maximum number of recursive calls?

- Each time we make a recursive call, we divide size of array to be searched in half.
- How many times can we divide a number in half before there is only 1 element left?
- If you start with 2^k then divide to 2^{k-1}, 2^{k-2}, 2^{k-3}, ..., 2^0 = 1; divide k times by 2.

- In general can divide n by 2 at most $\log n$ times to get down to 1. In this course, we will write $\log n$ and understand that we mean $\log_2 n$.

There are at most $(\log n) + 1$ invocations of the method and therefore at most $2 \cdot ((\log n) + 1)$ comparisons. This is $O(\log n)$ comparisons.

`Comparable` `Objects`

If we are going to deal with `Object`s for a binary search, we need a way to compare them. 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.

See Example:
`/home/jteresco/shared/cs211/examples/BinSearch`

Note the weird syntax. In this case, we don't have a generic type for the class, we have it just for these methods.

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` and `Double`.

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

Sorting

Computers spend a lot of time sorting data. Some have claimed that anywhere from $\frac{1}{4}$ to $\frac{1}{3}$ of all computation time is spent doing sorting. We already saw that sorting data makes searching much more efficient. Now we consider how to approach sorting.

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 arrays of items to be sorted. The text starts with arrays of ints for simplicity, but we will consider `Comparable`s, as we saw in our generic binary search.

An extremely inefficient (both in time and space) but correct way to sort would be to construct all possible permutations of the array (there are $n!$ of them) and then look at each one in a linear time search to see if all pairs of adjacent objects are in the right order (each of these searches is potentially $O(n)$). We can do better.

We will build sorting procedures out of two main operations:

- compare two elements
- swap two elements

We know how to compare base types, and we saw the idea of `Comparable`s for comparing objects that provide a `compareTo()` method.

A swap is very easy to write in Java. If we have an array of some base type, we can write:

```
public static void swap(int data[], int i, int j) {

  int temp = data[i];
  data[i] = data[j];
  data[j] = temp;
}
```

If we have an array of `Object` references, we can easily just change the types of the array and the temp variable.

```
public static void swap(Object data[], int i, int j) {

  Object temp = data[i];
  data[i] = data[j];
  data[j] = temp;
}
```

Or, using generics:

```
public static <T> void swap(T[] data, int a, int b) {

  T temp = data[a];
  data[a] = data[b];
  data[b] = temp;
}
```

In this case, there is no great benefit to the generic version. We don't really care what the types of the elements of the array actually are. We are not treating them as anything more specific than `Object`s.

However, if you need to write a swap method inside a generic class, you will need to use the generic type.

Bubble Sort

We begin with a very intuitive sort. We just go through our array, looking at pairs of values and swapping them if they are out of order.

It takes $n-1$ "bubble-ups", each of which can stop sooner than the last, since we know we bubble up one more value to its correct position in each iteration. Hence the name *bubble sort*.

So we do $(n-1)+(n-2)+...+1 = O(n^2)$ comparisons. We swap, potentially, after each one of these, for $O(n^2)$ swaps.

Remember that a swap involves three assignments, which would be more expensive than the individual comparisons.

The text has code for an iterative bubble sort of `int`s. You can easily change this to a sort of `Comparable`s the same way we changed our binary search example from `int`s to `Comparable`s.

Think about how you'd write a recursive bubble sort.

Selection Sort

Our first improvement on the bubble sort is based on the observation that one pass of the bubble sort gets us closer to the answer by moving the largest unsorted element into its final position. Other elements are moved "closer" to their final position, but all we can really say for sure after a single pass is that we have positioned one more element.

So why bother with all of those intermediate swaps? We can just search through the unsorted part of the array, remembering the index of (and hence, the value of) the largest element we've seen so far, and when we get to the end, we swap the element in the last position with the largest element we found. This is the *selection sort*.

Here, we do the same number of comparisons, but at most $n-1 = O(n)$ swaps.

The text has an iterative selection sort on `int`s. Let's look at a recursive selection sort method on objects that implement `Comparable`.

See Example:
`/home/jteresco/shared/cs211/examples/SortingComparisons/SelectionSort`

Insertion Sort

Consider applying selection sort to an already-sorted array. We still need to make all $O(n^2)$ comparisons (but no swaps). This is unfortunate. There's a good chance that sorting routines could be called frequently on already-sorted or nearly-sorted data.

Our next procedure does better in those situations.

The idea is that we build up the sorted portion of the array, one item at a time, by inserting the next unsorted element into its final location. Everything else is cascaded up to make room. This is the *insertion sort*.

See Example:
`/home/jteresco/shared/cs211/examples/SortingComparisons/InsertionSort`

The complexity here is $O(n^2)$ again. The call to `recInsSort(n-1,elts)` takes $\leq n*(n-1)/2$ comparisons.

Because our `while` loop might quit early, an insertion sort only uses half as many comparisons (on average) than selection sort. Thus, it's usually twice as fast (but still $O(n^2)$).

Insertion sort also has much better behavior on sorted or nearly-sorted data. Each insertion might stop after just one comparison, leading to $O(n)$ behavior in this best case circumstance.

Merge sort

Each procedure we have considered so far is an "in-place" sort. They require only $O(1)$ extra space for temporary storage.

Next, we consider a procedure that uses $O(n)$ extra space in the form of a second array.

It's based on the idea that if you're given two sorted arrays, you can merge them into a third in $O(n)$ time. Each comparison will lead to one more item being placed into its final location, limiting the number of comparisons to $n - 1$.

In the general case, however, this doesn't do anything for our efforts to sort the original array. We have completely unsorted data, not two sorted arrays to merge.

But we can create two arrays to merge if we split the array in half, sort each half independently, and then merge them together (hence the need for the extra $O(n)$ space).

If we keep doing this recursively, we can reduce the "sort half of the array" problem to the trivial cases.

This approach, the *merge sort*, was invented by John von Neumann in 1945.

How many splits will it take? $O(\log n)$

Then we will have $O(\log n)$ merge steps, each of which involves sub-arrays totaling in size to n, so each merge (which will be k independent merges into $\frac{n}{k}$-element arrays) step has $O(n)$ operations.

This suggests an overall complexity of $O(n \log n)$.

The text has some example code for this. Again, you can easily convert it from its current functionality, sorting `int`s to sort `Comparable`s.

See Example:
`/home/jteresco/shared/cs211/examples/SortingComparisons/MergeSort`

Note that this implementation uses a clever way to allow copying only half of the array into the temp array at each step.

Quicksort

Another very popular divide and conquer sorting algorithm is the *quicksort*. This was developed by C. A. R. Hoare in 1962.

Unlike merge sort, quicksort is an in-place sort.

While merge sort divided the array in half at each step, sorted each half, and then merged (where all work is in the merge), quicksort works in the opposite order.

That is, quicksort splits the array (which takes lots of work) into parts consisting of the "smaller" elements and of the "larger" elements, sorts each part, and then puts them back together (trivially).

It proceeds by picking a *pivot* element, moving all elements to the correct side of the pivot, resulting in the pivot being in its final location, and two subproblems remaining that can be solved recursively.

See Example:
`/home/jteresco/shared/cs211/examples/SortingComparisons/QuickSort`

In this case the leftmost element is chosen as the pivot. Put it into its correct position and put all elements on their correct side of the pivot.

If `partition` works then `quickSort` clearly works.

Note: we always make a recursive call on a smaller array (but it's easy to make a coding mistake where it doesn't, and then the sort never terminates).

The complexity of quicksort is harder to evaluate than merge sort because the pivot will not always wind up in the middle of the array (in the worst case, the pivot is the largest or smallest element).

The `partition` method is clearly $O(n)$ because every comparison results in `left` or `right` moving toward the other and quit when they cross.

In the best case, the pivot element is always in the middle and the analysis results in $O(n \log n)$, exactly like merge sort.

In the worst case the pivot is at one of the ends and quicksort behaves like a selection sort, giving $O(n^2)$.

A careful analysis can show that quicksort is $O(n \log n)$ in the average case (under reasonable assumptions on distribution of elements of array).

Correctness and Complexity Proofs

Just as we used the principle of mathematical induction to prove properties of mathematical formulas, we can use it to prove statements about the correctness and complexity of recursive algorithms.

To prove correctness of a procedure like our recursive selection sort, our proof by mathematical induction will take the form:

1. Prove the base case(s). (Usually this is trivial – an empty or one-element instance)
2. Show that if the algorithm works correctly for all simpler (i.e., smaller) input, then it will

work for current input.

A Simple Correctness Proof: FastPower

Consider this method:

```
    // pre:  exp >= 0
    // post - Returns base raised to exp
public static int fastPower(int base, int exp) {
    if (exp == 0)
        return 1;
    if (exp%2 == 1)       // exp is odd
        return base * fastPower(base, exp-1);
    return fastPower(base * base, exp / 2);
}
```

We wish to show that the method is correct, that `fastPower(base,exp)` = $base^{exp}$.

Proof: We will proceed by mathematical induction on the value of `exp`.

Base case: `exp = 0, fastPower(base,0) = 1 =` $base^0$ – correct

Inductive hypothesis: Assume that `fastPower(base,exp) =` $base^{exp}$ for all $exp < n$.

Inductive step: Show that `fastPower(base,n) =` $base^n$

There are two cases to consider:

- n is odd: `fastPower(base,n) =` $base$ `* fastPower(base, n-1) =` $base * base^{n-1}$ (by induction) $= base^n$
- n is even: `fastPower(base,n) = fastPower(base*base,n / 2) =` $(base^2)(n/2)$ (by induction) $= base^n$ (since n is even)

Therefore by mathematical induction, the method works correctly for all values of `exp` $<= 0$. $\diamond$

Correctness Proof for Recursive Selection Sort

We wish to prove the correctness of the `recSelSort` algorithm given earlier.

Proof: We will proceed by mathematical induction on the size of the unsorted portion of the array `elts`. This is determined by the parameter `lastIndex`.

Base case: If `lastIndex == 0` then the single entry at `elts[0]` is the only part of the array we are considering. This is trivially sorted.

Inductive Hypothesis: Suppose the algorithm is correct when `lastIndex` $<$ `n`.

Inductive step: Show that it is correct for `lastIndex = n (`> 0`)`.

The `for` loop finds largest element in the unsorted part of the array and then swaps it with `elts[lastIndex]`.

Thus, at the end of the loop, `elts[lastIndex]` holds the largest element of the array.

Since `lastIndex - 1` $<$ `lastIndex`, we know (by the inductive hypothesis) that `recSelSort(lastIndex-1,elts)` sorts `elts[0..lastIndex-1]` correctly.

So at the end of the method, `elts[0..lastIndex-1]` are in order and `elts[lastIndex]` is $\geq$ all of them, so all of `elts[0..lastIndex]` is sorted.

Therefore by mathematical induction, this implementation of a recursive selection sort is correct. $\diamond$

Complexity Proof for Recursive Selection Sort

Claim: `recSelSort(n-1,elts)` (i.e, the sort of an n-element array) involves $n * (n - 1)/2$ comparisons of elements of the array.

Proof: We proceed by mathematical induction on the size of the array to sort.

Base case: For n = 1, we require 0 comparisons and $n * (n - 1)/2 = 0$.

Inductive hypothesis: Suppose `recSelSort(k-1,elts)` requires $k * (k - 1)/2$ comparisons for all $k < n$.

Inductive step: Show that `recSelSort(n-1,elts)` requires $n * (n - 1)/2$ comparisons.

In the method, `lastIndex` will be $n - 1$. Since our main loop will execute the comparison `lastIndex` times, we will incur $n - 1$ comparisons.

Any additional comparisons will take place in the recursive call: `recSelSort(last-1,elts)` where $last = n - 1$.

But by our inductive hypothesis (since $last < n$), this takes $last*(last-1)/2 = (n-1)*(n-2)/2$ comparisons.

Therefore, we have a total of $(n-1)+(n-1)*(n-2)/2 = (n-1)*2/2+(n-1)*(n-2)/2 = (n - 1) * (2 + n - 2)/2 = (n - 1) * n/2 = n(n - 1)/2$ comparisons.

By the principle of mathematical induction, we see that the sort of any n-element array will involve $n * (n - 1)/2$ comparisons. $\diamond$

This matches with our previous statements that `recSelSort` takes $O(n^2)$ comparisons. $\diamond$

Correctness of Recursive Merge Sort

It's easy to show that `mergeSortRecursive` is correct if `merge` is correct, as `merge` is where all the work takes place.

But `merge` is not recursive! It is easy to *convince* yourself that `merge` is correct, but a formal *proof* of correctness of iterative algorithms is actually harder than for recursive algorithms, and we will not prove this here.

Complexity of Recursive Merge Sort

It is straightforward to see that if the portion of the array under consideration has k elements (i.e., $k = high - low + 1$), then the complexity of `merge` is $O(k)$:

- If we only look at comparisons, then it's clear that every comparison in the `if` statement in the `while` loop results in an element being copied into `data`.
- In the worst case, you run out of all elements in one run when there is only 1 element left in the other run: $k-1$ comparisons, giving $O(k)$
- If we count copies of elements, then it's also $O(k)$ since $\frac{k}{2}$ copies are made in copying half of `data` into `temp`, and then anywhere from $\frac{k}{2}$ to k more copies in putting elements back (in order) into `data`.

We will prove the complexity by induction.

Claim: The complexity of `mergeSortRecursive`, as measured by the number of comparisons, is $O(n \log n)$ for sort of n elements.

It's easiest to prove this if we assume that $n = 2^m$ for some m. The general case can be extended to 2^m by padding the array with very large values, so this is sufficient.

We proceed by mathematical induction on m to show that a sort of $n = 2^m$ elements takes $\leq n \log n = 2^m \cdot m$ compares.

Base case: $m = 0$, so $n = 1$. No work is needed here, so the 0 needed compares $\leq 2^0 * 0$.

Inductive Hypothesis: Suppose our claim is true for $m-1$, that the sort takes $\leq 2^{m-1} \cdot (m-1)$ compares.

Inductive Step: We show that the claim is true for m.

`mergeSortRecursive` of $n = 2^m$ elements proceeds by doing `mergeSortRecursive` of two lists of size $\frac{n}{2} = 2^{m-1}$, followed by a call of `merge` on list of size $n = 2^m$.

Therefore, $numcompares \leq 2^{m-1} * (m-1) + 2^{m-1} * (m-1) + 2^m = 2 * (2^{m-1} * (m-1)) + 2^m = 2^m * (m-1) + 2^m = 2^m * ((m-1)+1) = 2^m * m$. Therefore by mathematical induction, $numcompares \leq 2^m * m = n \log n$. $\diamond$

Similar arguments can be made to show that the number of copies is $O(n \log n)$.

Radix Sort

We can't do better than $O(n \log n)$ in the average case for a general-purpose sort with no assumptions on the input data and original ordering.

We can actually do a sort in $O(n)$ time if we take advantage of some knowledge of the input array. We will not discuss it in class at this point, but the text describes one such approach, the *radix sort*.