CMPSCI 187: Programming With Data Structures

Lecture #24: Introduction to Searching and Sorting
7 November 2011

Introduction to Searching and Sorting

- L&C's `SortingAndSearching` Class
- Static Generic Methods
- Linear Search of an Array
- Binary Search of a Sorted Array
- Comparing Search Algorithms
- A Lower Bound for Searching
- Some O(n^2) Sorting Algorithms

L&C's `SortingAndSearching` Class

- We'll now spend considerable time on two basic operations for a collection. **Searching** is determining whether a particular object is in a collection, and **sorting** is transforming a collection until it is in order (for a given ordering).

- How shall we reuse code when we search and sort on different classes? We don't want to write separate methods to do this for every class, but this is a different situation from the data structures we have studied so far.

- Pre-generic Java had an interface `Comparable` for objects that could be searched or sorted by comparing one object to another. There were algorithms to sort or search arrays of `Comparable` objects, but making an array so general loses some of the advantages of data typing.

- We'd like generic methods that can sort objects of any suitable type, but where should these methods live? In a `SortingAndSearching` class.

Static Generic Methods

- We could have a generic class `SortingAndSearching<T extends Comparable<? super T>`, like the classes we had for ordered lists.

- But it would be awkward (and possibly costly) to instantiate an object of this class every time we wanted it. Instead we want old-fashioned methods that take input and produce output, without being attached to an object at all.

- A **generic method** can live in a class, and be declared **static** so that it does not need an object of that class. Note that we need to define the type variable `T` before we can use it in the types of parameters.

- We call this method as `SortingAndSearching.LinearSearch(...)`

```
public static <T extends Comparable<? super T>> boolean
   linearSearch (T[ ] data, int min, int max, T target)
```

Linear Search of an Array

- The simplest search idea is to check each element of an array in turn, to see whether it is the item we have in mind. Clearly this takes O(n) time in the worst case.

- In this algorithm's favor is that it can easily be adapted to a linear structure, and that it makes no assumptions about the structure it is searching. Against it is that O(n) time is slow.

```
public static <T extends Comparable<? super T>> boolean
      linearSearch (T[ ] data, int min, int max, T target) {
   int index = min;
   boolean found = false;
   while (!found && index <= max) {
      if (data[index].compareTo(target) == 0) found = true;
      index++;}
   return found;}
```

Binary Search of a Sorted Array

- We get a faster search of an array if it is sorted -- first check the middle element. If it is too high or too low, search the half of the array that contains the target if it is there at all.

- This is a simple use of recursion, with the base case being an array of length 1. In the worst case we need log n recursions to go from size n to size 1.

```
public static <T extends comparable<? super T>> boolean
      binarySearch (T[ ] data, int min, int max, T target) {
   boolean found = false;
   int midpoint = (min + max) / 2;
   if (data[midpoint].compareTo (target) == 0)
      found = true;
   else if (data[midpoint].compareTo (target) > 0)
      if (min <= midpoint - 1)
         found = binarySearch (data, min, midpoint - 1, target);
   else if (midpoint + 1 <= max)
      found = binarySearch (data, midpoint + 1, max, target);
   return found;}
```

Comparing Search Algorithms

- Linear search is more generally applicable, since it does not require random access to the inputs, and is certainly simpler.

- Binary search is faster, even for n = 3 with two instead of three comparisons in the worst case. But the advantage is small for small lists and may not matter. In particular it may not justify the overhead effort to keep the list sorted. (For example, note that using our linear search algorithm requires the base type T to have an order on it -- we could always define some kind of order on it but that might be more effort than the savings from binary search would justify.)

- If the type has an order, we can always run some **sorting algorithm** on any linear structure to make it sorted. The more searches we plan to do later, the more likely that sorting the data and enabling binary searches will pay off.

A Lower Bound for Searching

- Sometimes we are able to prove an algorithm to be **optimal**, meaning that no other algorithm can do better under certain assumptions.

- If we are searching a sorted array of length n, binary search requires us to look at about log n array elements in the worst case, comparing each with the target element. In fact if n = 2^k - 1 for some integer k, we can search with exactly k probes. (Examples: k = 1 and n = 2^1 - 1 = 1, one probe is both necessary in the worst case and sufficient. For k = 2 and n = 2^2 - 1 = 3, two probes are necessary and sufficient.)

- To show that we *need* k probes to search n = 2^k - 1 elements, we use an **adversary argument**. If you claim to have an algorithm to do better, I need to show that there *exists* a case where your algorithm either uses k or more probes, or gets the wrong answer. The easiest way to do this is to construct the bad case in response to the questions that the algorithm asks about the data. (I'll describe the example of the **dishonest guessing game**.)

Some Simple Sorting Algorithms

- The problem of **sorting** is a basic one -- we are given n items in an arbitrary order and need to put them in order. We'll later see a number of ways to do this in O(n log n) time, but first we'll see how some basic ideas accomplish the job in O(n^2) time.

- Given an ordered list or other priority queue, we can put elements into the structure in any order we like and then get them out in order. For an ordered list this is called **insertion sort** and takes O(n^2) because each insertion takes O(n). Using an O(log n) priority queue (when we get one) will allow this idea to sort in O(n log n) time.

- Back in Discussion #2 we sorted some containers by repeatedly finding the smallest one in the collection and removing it. This general method is called **selection sort** and also takes O(n^2) time if we need to take O(n) time for each search operation.

More on Simple Sorting Algorithms

- **Bubble sort** is another O(n^2) algorithm whose only virtue it its simplicity. We make repeated passes over the set, exchanging any pair that is out of order. It is not hard to show that n - 1 passes of this kind leave the data sorted.

- Unlike insertion or selection sort, bubble sort leads to an O(n) time **parallel** algorithm -- if many compare-and-swaps can be done at once. For example, suppose n people are standing in a line and we want to sort them by height. We can do this in n - 1 rounds as follows: On even-numbered rounds, each even-numbered person compares heights with the person *after* them and changes places if they are shorter. On odd-numbered rounds, each even-numbered person compares heights with the person *before* them and changes places if they are taller.

- We will argue in Discussion #9 that you need at least O(n log n) time to sort n items by comparisons. Here we use O(n) time, but O(n^2) comparisons.