CS112 Lecture: Searching and Sorting

Revised 4/29/08

Objectives:

1. To introduce the linear search algorithm.
2. To introduce the binary search algorithm.
3. To show superiority of O(log n) vs O(n)
4. To take note of sorting methods discussed in book (selection, insertion, merge) and "preview coming attractions"..

I. Introduction/Motivation

A. Many computer applications require the ability to search a list of items to find some particular item.

EXAMPLES:

1. When students register for courses, the software used by the registrar must locate the appropriate course record (identified by course ID) and then the appropriate student record (identified by student ID) to ensure tht there is room in the course and then to record that the student is registered in these courses.

2. Entering a URL in a web browser results in several searches - e.g.:

   a) One or more name servers perform searches to translate the server's name into an IP number

   b) One or more routers perform searches to find a physical path from the requesting computer to the server.

   c) The server performs one or more searches in various directories to locate the file containing the requested page.

3. Others ?

   ASK

B. Likewise, many computer applications require the ability to sort a list of items into some order.

1. We shall see shortly that one motivation for this is improved search performance - it is faster to search for an item in an appropriately-ordered list than in a list that is not ordered.

Example: Which is easier - to look up a person in a conventional phone book given their last name, or to look up a person in a conventional phone book given their street address? Why?

ASK

2. A second motivation is human convenience - in many cases, humans prefer information to be presented in some rational order.

   Example: The course schedule booklets used at registration time contains two lists of courses:

   a) One sorted by course ID (department + number)

   b) One sorted by time

3. A third motivation for sorting is that there are some operations which depend on working with sorted lists - e.g. computer matching.

   a) Consider the process used by state tax authorities to match information you supply on your state tax return with information supplied on your federal return. Basically what they do is sort the information they get from taxpayers by social security number, and likewise sort the information they get on Massachusetts residents from the federal government by social security number. Now they repeat the following process as long as the lists are not empty.

      (1) If the first entry on both lists has the same ssn, they compare the information for consistency and move on the the next item on both lists.

      (2) If the first entry on the federal list has a smaller ssn, then some Massachusetts taxpayer may not have filed a state tax return. (In this case, they take appropriate action and then move on to the next item on the federal list, keeping the same item on the state list.)

      (3) If the first entry on the state list has a smaller ssn, then some Massachusetts taxpayer may not have filed a federal tax return. (In this case, they take appropriate action and then move on to the next item on the state list, keeping the same item on the federal list.)

   b) While people who pay their taxes conscientiously should appreciate the use of matching in this case as a tool for catching people who fail to carry their fair share of the load, matching can also be used in less socially-desirable ways - e.g. to match (say) a list of subscribers to some "subversive" periodical against a list of employees in a company that doesn't want their employees reading such material.

II. Searching Algorithms

A. The simplest and most generally applicable search strategy is a strategy called *linear* or *sequential search.*

1. Basic algorithm:

```
Start searching with the first item
while there are still items to search and we
        haven't found the one we want
    move on to the next item
if we have found what we want
    return it
else
    return some failure code (e.g. null or -1)
```

2. Example: Do a sequential search of the class as seated for a person with a particular name.

3. Time complexity: takes time proportional to the number of items in the list. We call this O(n). Note that average case for item found examines half the items in the list; item not found or worst case for item found examines all of them.

B. An alternative, and much better, searching strategy becomes possible if the entries are kept in order on the key we are searching for - e.g. here alphabetical order of name. This strategy is called *binary search.*

1. Demo using searching for a page in a book

2. Basic algorithm

```
lo = index of start of list (e.g. 0);
hi = index of last item (e.g. length - 1);
Calculate position of middle item: mid = (lo+hi)/2;
while (lo <= hi && middle item is not what we want)
    if middle item is bigger than what we want
        hi = mid - 1;
    else
        lo = mid + 1;
    mid = (lo + hi) / 2;
if (lo > hi)
    item we want is not in list;
else
    item at position mid is what we want;
```

3. Time complexity:

   ASK

   a) At each time through the loop, we cut the number of items under consideration in half. Thus, after k iterations on a list starting with n items, we are still considering $n / 2^k$. If we get down to the last item, we are done in any case - thus the maximum number of comparisons before this time is given by $n / 2^k = 1$, or $2^k = n$, or $k = \log_2 (n)$. Total comparisons is 1 more (for last item).

   b) "3 AM phone call fact". Note, too, that computer scientists almost always mean base 2 when talking about the log of n.

   c) How much better is this than sequential search? For small n, somewhat better - for large n vastly better:

n	Sequential search		Binary search
	Average	Worst	(both)
8	4	8	4
16	8	16	5
32	16	32	6
...			
1024	512	1024	11
...			
1000000	500000	1000000	21
...			
1000000000	500000000	1000000000	31

4. Note that binary search only works if the list is ordered *on the key we are using for the search.* (E.g. if a list of people were in alphabetical order of name, we wouldn't be able to use binary search to look up a person by zip code.). In all other cases, we still have to use sequential search.

5. The need to maintain the list in order has a performance price tag of its own, of course, which we will consider shortly.

C. We won't discuss these now, but it should be noted that there are searching strategies other than sequential or binary search available to us, which depend on other strategies for ordering the list.

1. One such strategy - which is the one used by the class `java.util.HashMap` - potentially yields O(1) performance (time independent of list size) in the average case, but can degenerate to O(n) - the same as sequential search - in the worst case. You will consider this strategy further in CS212.

2. Another strategy which is the one used by the class `java.util.TreeMap` - makes use of a structure called a balanced binary tree which allows searches in O(log n) time (as with binary search) but also allows the tree structure to be maintained in O(log n) time. Again, you will consider this strategy further in CS212.

III. Sorting

A. We said earlier that there are a number of reasons why we might find it desirable to keep a series of entries in some kind of order.

1. We have just considered one - vastly improved search performance.

2. Another motivation, of perhaps even greater importance, is user convenience.

3. There are other operations whose performance is also improved by keeping one or more lists in order - e.g. computer matching.

B. For a variety of reasons, then, we are interested in algorithms that can be used to sort a list of items.

1. Terminology:

a) Sort key - the value that is used to establish the ordering. (Can be composite - e.g. sort by last name, and by first name within the same last name.).

b) The key must be capable of being compared by a "<" operation, which must be transitive (A < B and B < C implies A < C)

(1) In Java, primitive types (ints etc.) have a < operator which meets this requirement.

(2) In Java, < cannot be used with objects. However, many classes (e.g. String) define an appropriate compareTo() method that can be used to compare two objects of that class - e.g. if s and t are strings, we can ask if s alphabetically precedes t as follows:

```
if (s.compareTo(t) < 0) ...
```

(3) For our discussion now, we will simply use the < operator generically - i.e. to stand for "<" or "compareTo(...) < 0" as appropriate.

c) A list is said to be sorted if, for all i: 0 <= i < list length - 1

the key of item [i] <= the key of item [i + 1]

d) The task of sorting is to find a *sorted permutation* of the original list. Note that, if no two entries have the same key, the sorting of a given list is unique.

2. Over the years, *many* sorting algorithms have been developed. We will consider these in detail in CS212. For now, note that the book discussed three:

ASK - THEN DEMO EACH BY SORTING THE CLASS

a) Selection sort [select alphabetically first, then put in place, alphabetically second, then put in place ...]

b) Insertion sort [build sorted sublists of length 1, 2 ...]

c) Merge sort [Break class in half, recursively sort, merge]

3. The first two have performance that is $O(n^2)$. The third is $O(n \log_2 n)$, which is as good as one can do except in some very unusual cases. Note that both costs are very high - e.g. sorting a list of a million items using an $O(n^2)$ sort will take on the order of 1 trillion comparisons, though most algorithms can cut that by a factor of 2 to only 500 billion!. (The same list, using an $O(n \log_2 n)$ algorithm, can be sorted in only about 20 million comparisons!)

4. The book gave code for each of these sorts. We will not discuss this here - this is a topic in CS212