Course Notes 9

Unrelated Facts Worth Remembering

- Real change in any organization usually happens due to an outside impetus
- Own your career

Table of Contents

1 Introduction

Two common task which permeate computer science are sorting—putting into order, and searching—finding from a bunch. The following document describes some of the techniques used for these tasks.

2 Searching and Sorting

Two array processing techniques that are particularly common are searching and sorting. Searching here refers to finding an item in the array that meets some specified criterion. Sorting refers to rearranging all the items in the array into increasing or decreasing order (where the meaning of increasing and decreasing can depend on the context).

Sorting and searching are often discussed, in a theoretical sort of way, using a list of numbers as an example. In practical situations, though, more interesting types of data are usually involved. For example, the array might be a mailing list, and each element of the array might be an object containing a name and address. Given the name of a person, you might want to look up that person's address. This is an example of searching, since you want to find the object in the array that contains the given name. It would also be useful

to be able to sort the array according to various criteria. One example of sorting would be ordering the elements of the array so that the names are in alphabetical order. Another example would be to order the elements of the array according to zip code before printing a set of mailing labels. (This kind of sorting can get you a cheaper postage rate on a large mailing.)

This example can be generalized to a more abstract situation in which we have an array of objects, and we want to search or sort the array based on the value of one of the instance variables in that array. We can use some terminology here that originated in work with "databases," which are just large, organized collections of data. We refer to each of the objects in the array as a record. The instance variables in an object are then called fields of the record. In the mailing list example, each record would contain a name and address. The fields of the record might be the first name, last name, street address, state, city and zip code. For the purpose of searching or sorting, one of the fields is designated to be the key field. Searching then means finding a record in the array that has a specified value in its key field. And sorting means moving the records around in the array so that the key fields of the record are in increasing (or decreasing) order.

In this section, most of my examples follow the tradition of using arrays of numbers. But I'll also give a few examples using records and keys, to remind you of the more practical applications.

3 Searching

There is an obvious algorithm for searching for a particular item in an array: Look at each item in the array in turn, and check whether that item is the one you are looking for. If so, the search is finished. If you look at every item without finding the one you want, then of course you can say that the item is not in the array. It's easy to write a subroutine to implement this algorithm. Let's say the array that you want to search is an array of ints. Here is a method that will search the array for a specified integer. If the integer is found, the method returns the index of the location in the array where it is found. If the integer is not in the array, the method returns the value -1 as a signal that the integer could not be found:

```
static int find(int[] A, int N)
{
      // Searches the array A for the integer N.
      for (int index = 0; index < A.length; index++)
      {
            if ( A[index] == N )
            return index;  // N has been found at this index!
      }
      // If we get this far, then N has not been found
      // anywhere in the array.  Return a value of -1 to
      // indicate this.
      return -1;
}
```

This method of searching an array by looking at each item in turn is called linear search. If nothing is known about the order of the items in the array, then there is really no better alternative algorithm. But if the elements in the array are known to be in increasing or decreasing order, then a much faster search algorithm can be used. An array in which the elements are in order is said to be sorted. Of course, it takes some work to sort an array, but if the array is to be searched many times, then the work done in sorting it can really pay off.

Binary search is a method for searching for a given item in a sorted array. The idea is simple: If you are searching for an item in a sorted list, then it is possible to eliminate half of the items in the list by inspecting a single item. For example, suppose that you are looking for the number 42 in a sorted array of 1000 integers. Let's assume that the array is sorted into increasing order. Suppose you check item number 500 in the array, and find that item is 93. Since 42 is less than 93, and since the elements in the array are in increasing order, we can conclude that if 42 occurs in the array at all, then it must occur somewhere before location 500. All the locations numbered 500 or above can be eliminated as possible locations of the number 42.

The next obvious step would be to check location 250. If the number at that location is, say, 21, then you can eliminate locations before 250 and limit further search to locations between 251 and 499. The next test will limit the search to about 125 locations, and the one after that to about 62. After just 10 steps, there is only one location left. This is a whole lot better than looking through every element in the array. If there were a million items, it would still take only 20 steps for this method to search the array! (Mathematically, the number of steps is the logarithm, in the base 2, of the number of items in the array.)

In order to make binary search into a Java subroutine that searches an array A for an item N, we just have to keep track of the range of possible locations that could possibly contain N. At each step, as we eliminate possibilities, we reduce the size of this range. The basic operation is to look at the item in the middle of the range. If this item is greater than N, then the second half of the range can be eliminated. If it is less than N, then the first half of the range can be eliminated. (If the number in the middle just happens to be N exactly, then the search is finished.) Here is a subroutine that returns the location of N in a sorted array A. If N cannot be found in the array, then a value of -1 is returned instead:

```
static int binarySearch(int[] A, int N)
{
      // Searches the array A for the integer N.
      // A is assumed to be sorted into increasing order.
      int lowestPossibleLoc = 0;
      int highestPossibleLoc = A.length - 1;
      while (highestPossibleLoc >= lowestPossibleLoc)
      {
            int middle = (lowestPossibleLoc + highestPossibleLoc) / 2;
```

```
            if (A[middle] == N)
                  return middle;
                  // N has been found at this index!
            else if (A[middle] > N)
                  highestPossibleLoc = middle - 1;
                  // eliminate locations >= middle
            else
                  lowestPossibleLoc = middle + 1;
                  // eliminate locations <= middle
      }
      // At this point, highestPossibleLoc < LowestPossibleLoc,
      // which means that N is known to be not in the array.  Return
      // a -1 to indicate that N could not be found in the array.
      return -1;
}
```

4 Association Lists

One particularly common application of searching is with association lists. The basic example of an association list is a dictionary. A dictionary associates definitions with words. Given a word, you can use the dictionary to look up its definition. We can think of the dictionary as being a list of pairs of the form (w,d), where w is a word and d is its definition. A general association list is a list of pairs (k,v), where k is some "key" value, and v is a value associated to that key. In general, we want to assume that no two pairs in the list have the same key. The basic operation on association lists is this: Given a key, k, find the value v associated with k, if any.

Association lists are very widely used in computer science. For example, a compiler has to keep track of the location in memory where each variable is stored. It can do this with an association list in which each key is a variable name and the associated value is the location of that variable. Another example would be a mailing list, if we think of it as associating an address to each name on the list. As a related example, consider a phone directory that associates a phone number to each name. The items in the list could be objects belonging to the class:

```
      class PhoneEntry
      {
            String name;
            String phoneNum;
      }
```

The phone directory could be an array of PhoneEntry objects. To keep things neat, a phone directory could be an instance of the class:

```
class PhoneDirectory
{
      PhoneEntry[] info = new PhoneEntry[100];
      // space for 100 entries
      int entries = 0;
      // actual number of entries in the array
```

```
	void addEntry(String name, String phoneNum)
	{
		// add a new item at the end of the array
		info[entries] = new PhoneEntry();
		info[entries].name = name;
		info[entries].phoneNum = phoneNum;
		entries++;
	}
	String getNumber(String name)
	{
		// Return phone number associated with name,
		// or return null if the name does not occur
		// in the array.
		for (int index = 0; index < entries; index++)
		{
			if (name.equals(info[index].name))  // found it!
			return info[index].phoneNum;
		}
		return null;
		// name wasn't found
	}
}
```

Note that the search method, getNumber, only looks through the locations in the array that have actually been filled with PhoneEntries. Also note that unlike the search routines given earlier, this routine does not return the location of the item in the array. Instead, it returns the value that it finds associated with the key value, name. This is often done with association lists.

This class could use a lot of improvement. For one thing, it would be nice to use binary search instead of simple linear search in the getNumber method. However, we could only do that if the list of PhoneEntries were sorted into alphabetical order according to name. In fact, it's really not all that hard to keep the list of entries in sorted order, as you'll see in just a second.

5 Insertion Sort

We've seen that there are good reasons for sorting arrays. There are many algorithms available for doing so. One of the easiest to understand is the insertion sort algorithm. This method is also applicable to the problem of keeping a list in sorted order as you add new items to the list. Let's consider that case first:

Suppose you have a sorted list and you want to add an item to that list. If you want to make sure that the modified list is still sorted, then the item must be inserted into the right location, with all the smaller items coming before it and all the bigger items after it. This will mean moving each of the bigger items up one space to make room for the new item.

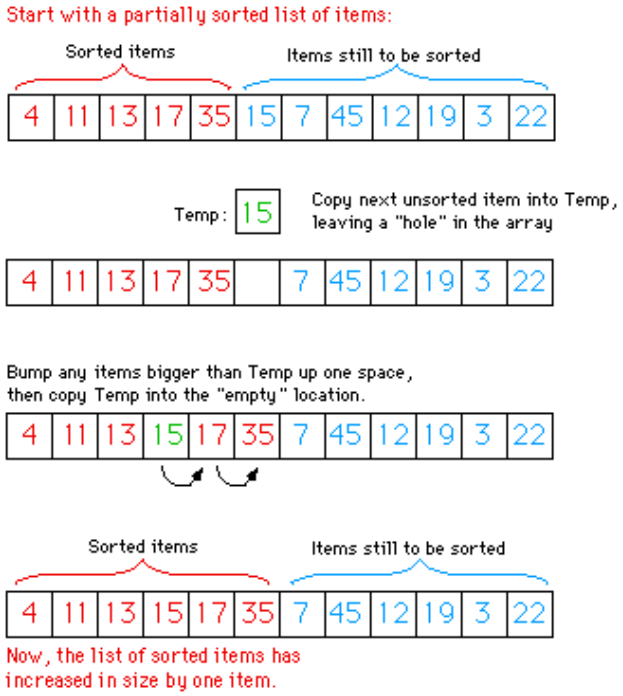
```
static void insert(int[] A, int itemsInArray, int newItem)
{
	// Assume that A contains itemsInArray items in increasing
```

```
        // order (A[0] <= A[1] <= ... <= A[itemsInArray-1]).
        // This routine adds newItem to the array in its proper
        // order.
        int loc = itemsInArray - 1;
        while (loc >= 0 && A[loc] > newItem)
        {
              A[loc + 1] = A[loc];
              // bump item from A[loc] up to loc + 1
              loc = loc - 1;
              // move down to next location
        }
        A[loc + 1] = newItem;
        // put new item in last vacated space
}
```

Conceptually, this could be extended to a sorting method if we were to take all the items out of an unsorted array, and then insert them back into the array one-by-one, keeping the list in sorted order as we do so. Each insertion can be done using the insert routine given above. In the actual algorithm, we don't really take all the items from the array; we just remember what part of the array has been sorted:

```
static void insertionSort(int[] A)
{
        // sort the array A into increasing order
        int itemsSorted;
        // number of items that have been sorted so far
        for (itemsSorted = 1; itemsSorted < A.length; itemsSorted++)
        {
              // assume that items A[0], A[1], ... A[itemsSorted-1] have
              // already been sorted, and insert A[itemsSorted]
              // into the list.
              int temp = A[itemsSorted];
              // the item to be inserted
              int loc = itemsSorted - 1;
              while (loc >= 0 && A[loc] > temp)
              {
                    A[loc + 1] = A[loc];
                    loc = loc - 1;
              }
              A[loc + 1] = temp;
        }
}
```

The following is an illustration of one stage in insertion sort. It shows what happens during one execution of the for loop in the above method, when itemsSorted is 5.



6 Selection Sort

Another typical sorting method uses the idea of finding the biggest item in the list and moving it to the end -- which is where it belongs if the list is to be in increasing order. Once the biggest item is in its correct location, you can then apply the same idea to the remaining items. That is, find the next-biggest item, and move it into the next-to-last space, and so forth. This algorithm is called selection sort. It's easy to write:

```
static void selectionSort(int[] A)
{
	// sort A into increasing order, using selection sort
	for (int lastPlace = A.length-1; lastPlace > 0; lastPlace--)
	{
		// Find the largest item among A[0], A[1], ... A[lastPlace],
		// and move it into position lastPlace by swapping it with
		// the number that is currently in position lastPlace
		int maxLoc = 0;
		// location of largest item seen so far
		for (int j = 1; j <= lastPlace; j++)
		{
			if (A[j] > A[maxLoc])
			maxLoc = j;
			// Now, location j contains the largest item seen
		}
		int temp = A[maxLoc];
		// swap largest item with A[lastPlace]
		A[maxLoc] = A[lastPlace];
		A[lastPlace] = temp;
	}
}
```

7 Merge Sort

Both insertion sort and selection sort work fine for fairly small arrays. However, for large arrays, they both take unreasonable amounts of time. Just as binary search is much faster than linear search for large arrays, there are sorting algorithms that can sort large arrays much more quickly than either selection sort or insertion sort. Unfortunately, most of the fast algorithms are more complicated than I want to get into here. One of the fast sorting methods, merge sort, is reasonably easy to explain -- but in practice it is rarely used because it requires an extra array to use as scratch memory.

Merge sort is based on the idea of merging two sorted lists into one longer sorted list. This is easy to do simply by comparing the items at the head of each list and moving the smaller of those two items into the new list. (You need the extra array for the new list; there is no easy trick for avoiding this requirement, as there was for insertion sort.)

Now, imagine starting with a large unsorted list of items. Pair off the items and sort each pair into increasing order. Each pair can be considered to be a sorted list of length two. The lists of length two can then be paired up, and each pair of lists can be merged into a sorted list of length four. Then lists of length four can be merged into lists of length eight, the lists of length eight into lists of length sixteen, and so on. At each stage, the length of the sorted lists doubles. Soon enough, all the items will be in one long, sorted list. (This requires just a little bit of fudging if the number of items is not a power of two, since in that case when you pair off the lists, you might have an extra list left over.)

Unsorting I can't resist ending this section on sorting with a related problem that is much less common, but is a bit more fun. That is the problem of putting the elements of an array into a random order. The typical case of this problem is shuffling a deck of cards. A good algorithm for shuffling is similar to selection sort, except that instead of moving the biggest element to the end of the list, you pick a random element and move it to the end of the list. Here is a subroutine to shuffle an array of ints:

```
static void shuffle(int[] A)\
{
       for (int lastPlace = A.length-1; lastPlace > 0; lastPlace--)
       {
              // choose a random location from among 0,1,...,lastPlace
              int randLoc = (int)(Math.random()*(lastPlace+1));
              // swap items in locations randLoc and lastPlace
              int temp = A[randLoc];
              A[randLoc] = A[lastPlace];
              A[lastPlace] = temp;
       }
}
```

8 Big O

Big O is a way of characterizing an algorithm in Computer Science by relating the algorithms performance to the size of data it is presented and its worst case performance. The amount of data is known by the variable “n”.

To give a rough and ready example, if you have a loop that does something to each piece of data and stops, then you have an order n algorithm or O(n). We like to throw a lot of math around the idea but a lot of the time the O of an algorithm is governed by the amount of looping through the data (n) that you do. If you have the data being manipulated within a loop that has a loop within it (a nested loop) then you have an $O(n^2)$ algorithm.

8.1 Performance of Linear Search

In computer science, linear search is a search algorithm, also known as sequential search, that is suitable for searching a set of data for a particular value.

It operates by checking every element of a list until a match is found. Linear search runs in O(N). If the data are distributed randomly, on average N/2 comparisons will be needed. The best case is that the value is equal to the first element tested, in which case only 1 comparison is needed. The worst case is that the value is not in the list, in which case N comparisons are needed.

8.2 Binary Search

In computer science, binary search or binary chop is a search algorithm for finding a particular value in a linear array, by "ruling out" half of the data at each step. A binary search finds the median, makes a boolean comparison, the results of which will choose either the upper or lower quartile for further comparison, and so on into absolution. A binary search is an example of a divide and conquer algorithm and a dichotomic search.

8.3 Bubble Sort

The bubble sort is the oldest and simplest sort in use. Unfortunately, it's also the slowest.

The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order). This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list.

The bubble sort is generally considered to be the most inefficient sorting algorithm in common usage. Under best-case conditions (the list is already sorted), the bubble sort can approach a constant O(n) level of complexity. General-case is an abysmal O(n2).

8.4 Selection Sort

The selection sort works by selecting the smallest unsorted item remaining in the list, and then swapping it with the item in the next position to be filled. The selection sort has a complexity of O(n2).

Pros: Simple and easy to implement.
Cons: Inefficient for large lists, so similar to the more efficient insertion sort that the insertion sort should be used in its place.

8.5 Insertion Sort

The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place.

Like the bubble sort, the insertion sort has a complexity of O(n2). Although it has the same complexity, the insertion sort is a little over twice as efficient as the bubble sort.

Pros: Relatively simple and easy to implement.
Cons: Inefficient for large lists.