Sorting and Searching Algorithms

The following provides an introduction to some of the terminology and common algorithms for sorting and searching the values found in a data structure. Algorithms are presented that implement the *selection* and *bubble* sort algorithms, and the *linear* and *binary* search algorithms. In addition, the relative efficiency of each of the sorting and searching algorithms is briefly discussed.

Sorting Algorithms

Key field - a value occurring in a data structure that uniquely identifies a particular value of the structure. The key field is used for sorting or searching the data structure's values.

Sorting - The process of arranging data into ascending (increasing) or descending (decreasing) order by some key field or fields that determine the data's sorting order (e.g., numerical, ordinal, or chronological).

Sorting lists of values is one of the most thoroughly studied areas of computer science. The primary reason for keeping lists sorted is to make searching them easier. But, having a list sorted can also make interpreting the data stored in the list easier. There are two categories of sorting algorithms:

Internal sort - a sorting process by which all items to be sorted must be in the computer's main memory (RAM) while the sorting is being done. This memory requirement is in addition to the memory required for the sort program and any temporary storage areas it requires during the sorting process. The advantage of an internal sort is its speed of execution.

External sort - a sorting process accomplished by repeatedly reading into the computer's main memory part of the data located in a file, sorting the elements read, and then storing the sorted portions on an external storage device. The advantage of an external sort is that it uses relatively little main memory and hence can sort large amounts of data.

We will cover two sorting algorithms. Each algorithm will sort lists of data into ascending order based on some key field, and each will implement internal sorts on array data structures.

Selection Sort

Given a list of names, how might you sort them?

Repeat: For each name in the list from the first to the last
    Find the name that is lowest in alphabetical order (that has not been crossed off the list)
    Cross that name off the original list
    Write that name to the end of a second list

This process is simple but very inefficient. It wastes memory because it requires space to store two complete lists. We can do away with the need for a second list by making one small change: when you have found the next lowest element, move it to its new location in the list and use its old space as the location to move the element to that it is displacing.

Pseudocode: Selection Sort

For each element in the list from the first to the next-to-last
    Find the smallest value in the list from the current element to the last element
    If the smallest value is not equal to the current value
        Swap the current value with the minimum value in the list

Selection Sort Example – Sort the following list using the selection sort algorithm:

Original list: 69 97 32 55 84

Pass 1: current element = 69; the smallest value in the list from 69 to the last (84) is 32; swap

New list: 32 97 69 55 84

Pass 2: current element = 97; the smallest value in the list from 97 to the last is 55; swap

New list: 32 55 69 97 84
Pass 3: current element = 69; the smallest value in the list from 69 to the last is 69; no swap
New list: 32 55 69 97 84
Pass 4: current element = 97; the smallest value in the list from 97 to the last is 84; swap
Sorted list: 32 55 69 84 97

Bubble Sort

The *Bubble Sort*, also called the *Exchange Sort*, is one in which adjacent elements of the list are exchanged with one another in such a manner that the list becomes sorted. It works by making multiple passes through the list such that during each pass, the largest unsorted element in the list is placed into its correct place (i.e., the largest value will "bubble up" on each pass through the list to its proper position).

Pseudocode: Bubble Sort

Assume that a list of *n* elements is stored in an array *A*:

```
For (pass = 0; pass < n-1; pass++)
    For (j=1; j<(n-pass); j++)
            If A[j]<A[j-1]
               Swap A[j] with A[j-1]
    EndFor
Endfor
```

Bubble Sort Example – Sort the following list using the bubble sort algorithm:
Note: The inner loop comparisons are designated by bold typeface.

Original list:	97	84	69	55	32	
First pass:	**84**	**97**	69	55	32	
	84	**69**	**97**	55	32	
	84	69	**55**	**97**	32	
	84	69	55	**32**	**97**	done with first pass
Second pass:	**69**	**84**	55	32	97	
	69	**55**	**84**	32	97	
	69	55	**32**	**84**	97	done with second pass
Third pass:	**55**	**69**	32	84	97	
	55	**32**	**69**	84	97	done with third pass
Final pass:	**32**	**55**	69	84	97	done – the list is sorted.

Efficiency of the Sorting Algorithms

Big O Notation - used to express the running time of an algorithm in terms of *N*, where *N* is the number of items being processed. Saying that an algorithm is *O(N)* means that it takes approximately or "on the order" of *N* steps to complete (ignoring constant multiples, additions, or subtractions). Some of the most common running times encountered when examining the performance of sorting algorithms:

O(1) - *constant time*: The running time of the algorithm is not affected by the specific data being processed; this includes any algorithm that works by evaluating a formula, such as a quadratic equation, the distance formula, the Pythagorean theorem, or computing the square of a number.

$O(\log_2 N)$ - *logarithmic time*: The running time increases very slowly with *N* since $\log_2 N$ merely doubles when *N* is squared.

O(N) - *linear time*: running time is directly proportional to *N*. For example, if *N* doubles, then running time doubles as well.

$O(N \log_2 N)$ (*no name)*: running time is the product of *N* and $\log_2 N$. It increases slightly faster than *N*. The difference between *O(N)* and $O(N \log_2 N)$ becomes smaller as *N* grows large.

$O(N^2)$ - *quadratic time*: running time increases with the square of *N*. An $O(N^2)$ algorithm will require about N^2 steps.

$O(2^N)$ - *exponential time*: running time increases as a power of the base used. Algorithms with exponential running times become impractical as *N* increases due to overhead involved in processing.

Choosing the Appropriate Sorting Algorithm

The selection sort is *O*(*N*) for the number of moves required and $O(N^2)$ in the number of comparisons required to sort a list consisting of *N* elements. The bubble sort is also $O(N^2)$ in the number of comparisons required; the number of moves required depends on the original ordering of the list. For example, in the worst case, if the list is in reverse order, then the bubble sort is $O(N^2)$ in the number of moves required. For randomly arranged lists, the selection sort generally performs equal to or better than the bubble sort.

Searching Algorithms

Sequential (Linear) Search

This is the simplest of all of the searching techniques: Start at the beginning of a list and examine each element until you find the one you want. The list does not have to be in order, but the search is more efficient if it is. For the algorithm given here, we will assume that the list is not ordered.

Pseudocode: Sequential Search

Let *key-value* be the value that is being searched for:

```
Set the found flag to false
Start with the first element of the list
While there are more elements and found flag is false
    If the current element = key-value
        Set found flag to true
    Else
        Get the next element in the list
EndWhile
If found flag is true
    Return the record information requested
Else
    Return not found
```

Binary Search

The binary search algorithm is a more efficient algorithm than the sequential search, but is more complicated and requires the list to be in order. The binary search algorithm is one in which a list to be searched is repeatedly divided in half – each time the list is divided in half, the part of the list that does not contain the key value being sought is ignored. This algorithm is more easily understood when implemented recursively – both a recursive and non-recursive version of this algorithm are presented below.

Pseudocode: Binary Search - Recursive Algorithm

If there are no elements in the list
    Stop searching, the key value is not in the list
Else
    Determine the middle element of the list
If the middle element contains the desired key
    Stop searching, the key value has been located
Else
    If the middle element > the desired key value
        Binary search the 1st half of the list
    Else
        Binary search the 2nd half of the list

Pseudocode: Binary Search - Non-recursive Algorithm

Assume that a sorted list of *n* elements is stored in an array *A*:

Let left = 1 and right = n and found = false
While ((left <= right) and not found)
    Let middle = (left+right)/2
    If A[middle] = key value
        Set found to true
    Else if A[middle] > key value
        Set right = middle-1
    Else
        Set left = middle+1
EndWhile
If found
    Return that element was found at middle
Else
    Return that element was not found

Binary Search Example – Use the non-recursive binary search algorithm to search for 19 in the following list: 1 3 7 12 17 19 23 31 47

Pass 1: left = 1; right = 9; middle = 5; A[5] = 17 < 19, so left = 6 < right, so loop again
Pass 2: left = 6; right = 9; middle = 7; A[7] = 23 > 19, so right = 6 = left, so loop again
Pass 3: left = 6; right = 6; middle = 6; A[6] = 19, so found = true and exit the loop
19 was found at element 6 of the list

Efficiency of Searching Algorithms

The benefits of a sequential search are that it has a simple algorithm and that the list being searched does not need to be sorted in any order for the algorithm to work. But, it is a slow algorithm and is inefficient on searching large lists of data since it is an $O(N)$ search.

The binary search has the benefit of begin fast and efficient on large list of data. But, it is a more complicated algorithm and only works on sorted lists. It also is not guaranteed to be faster for searching very small lists – it requires fewer comparisons, but requires more computation, since arithmetic must be done to find the middle element of the portion of the list being searched. The binary search is an $O(\log_2 N)$ search for a list of N elements. For long lists, the binary search is much better to use than the sequential search.