Sorting and Searching

We sort many things in our everyday lives: A handful of cards when playing Bridge; bills and other piles of paper; jars of spices; and so on. And we have many intuitive strategies that we can use to do the sorting, depending on how many objects we have to sort and how hard they are to move around. Sorting is also one of the most frequently performed computing tasks. We might sort the records in a database so that we can search the collection efficiently. We might sort the records by zip code so that we can print and mail them more cheaply. . We might use sorting as an intrinsic part of an algorithm to solve some other problem, such as when computing the minimum-cost spanning tree.

1.1 Definition of Sorting

Given a set of records $r_1, r_2, \ldots, r_n$ with key values $k_1,\ k_2, \ldots, k_n$, the Sorting Problem is to arrange the records into any order s such that records $r_{s1}, r_{s2}, \ldots, r_{sn}$ have keys obeying the property $k_{s1} \le k_{s2} \le \ldots \le\ k_{sn}$. In other words, the sorting problem is to arrange a set of records so that the values of their key fields are in non-decreasing order.

When comparing two sorting algorithms, the most straightforward approach would seem to be simply program both and measure their running times. When analyzing sorting algorithms, it is traditional to measure the number of comparisons made between keys.

1.2 Applications of Sorting

Commercial computing: Organizations organize their data by sorting it. Accounts to be sorted by name or number, transactions to be sorted by time or place, mail to be sorted by postal code or address, files to be sorted by name or date, or whatever, processing such data is sure to involve a sorting algorithm somewhere along the way.

Search for information: Keeping data in sorted order makes it possible to efficiently search through it using the classic binary search algorithm.

Operations research: Suppose that we have N jobs to complete. We want to maximize customer satisfaction by minimizing the average completion time of the jobs. The shortest processing time first rule, where we schedule jobs in increasing order of processing time, is known to accomplish this goal.

Combinatorial search: A classic paradigm in artificial intelligence is to define a set of configurations with well-defined moves from one configuration to the next and a priority associated with each move.

Prim's algorithm, Kruskal's algorithm and Dijkstra's algorithm are classical algorithms that process graphs. These use sorting.

Huffman compression is a classic data compression algorithm that depends upon processing a set of items with integer weights by combining the two smallest to produce a new one whose weight is the sum of its two constituents.

String processing algorithms are often based on sorting.

1.3 Bubble Sort

Bubble Sort is often taught to novice programmers in introductory computer science courses. It is a relatively slow sort. It has a poor best-case running time.

Bubble Sort consists of a simple double for loop. The first iteration of the inner for loop moves through the record array from left to right, comparing adjacent keys. If the lower-indexed key's value is greater than its higher-indexed neighbor, then the two values are swapped. Once the largest value is encountered, this process will cause it to "bubble" up to the right of the array. The second pass through the array repeats this process. However, because we know that the largest value reached the right of the array on the first pass, there is no need to compare the right two elements on the second pass. Likewise, each succeeding pass through the array compares adjacent elements, looking at one less value than the preceding pass. The following figure[1] shows an example of Bubble Sort:

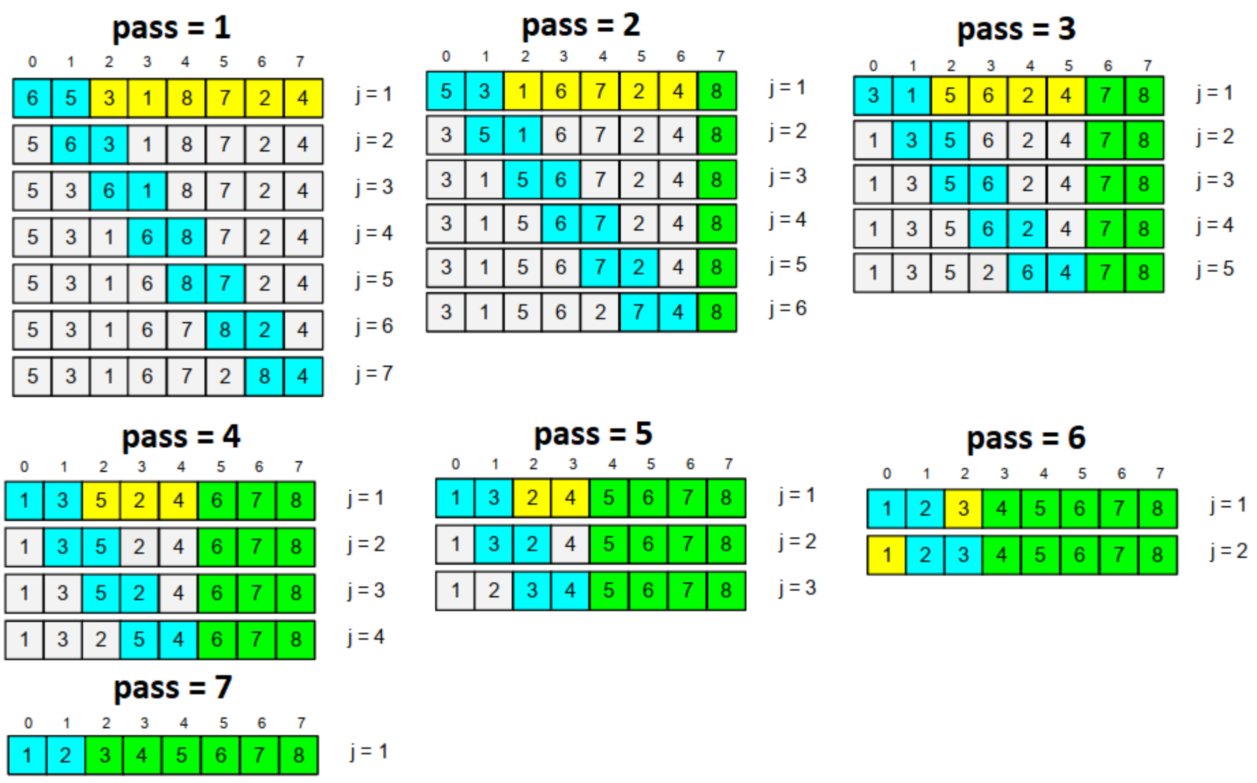

Figure 5. Error! No text of specified style in document.**.1: An illustration of Bubble Sort to sort an array in increasing order**

```
Algorithm (Bubble Sort):
```

[1] https://eleni.blog/2019/06/09/sorting-in-go-using-bubble-sort/

```
Input: A (array), N (#elements)
$pass$ =1
Step 1: $u$ = $N-$ $pass$
Compare the pairs ($A$[0], $A$[1]), ($A$[1], $A$[2]), ($A$[2], $A$[3]), … ,
($A$[$u$−1], $A$[$u$]) and Exchange pair of elements if they are not in
order.
Step 2: $pass$ = $pass$ + 1. If the array is unsorted and $pass$ < $N$−1
then go to step 1
```

It is to notice that if in a pass no swapping occurs then next passes are redundant. In the above illustration, it is seen that pass 7 is redundant since there was no swapping in pass 6.

Not much use in the real world, but it's easy to understand and fast to implement. It is used when a fast algorithm is needed to sort: 1) an extremely small set of data (Ex. Trying to get the books on a library shelf back in order.) or 2) a nearly sorted set of data. (Ex. Trying to decide which laptop to buy, because it is easier to compare pairs of laptops one at a time and decide which you prefer, than to look at them all at once and decide which was best.)

1.4 Selection Sort

Consider the problem of sorting a pile of phone bills for the past year. An intuitive approach might be to look through the pile until you find the bill for January, and pull that out. Then look through the remaining pile until you find the bill for February, and add that behind January. Proceed through the ever-shrinking pile of bills to select the next one in order until you are done. The ith pass of Selection Sort "selects" the ith smallest key in the array, placing that record into position i. In other words, Selection Sort first finds the smallest key in an unsorted list, then the second smallest, and so on. Its unique feature is that there are few record swaps. To find the next smallest key value requires searching through the entire unsorted portion of the array, but only one swap is required to put the record in place. Thus, the total number of swaps required will be $n-1$ (we get the last record in place "for free").

Selection Sort (as written here) is essentially a Bubble Sort, except that rather than repeatedly swapping adjacent values to get the next smallest record into place, we instead remember the position of the element to be selected and do one swap at the end. Selection sort is particularly advantageous when the cost to do a swap is high, for example, when the elements are long strings or other large records. Selection Sort is more efficient than Bubble Sort (by a constant factor) in most other situations as well.

There is another approach to keeping the cost of swapping records low that can be used by any sorting algorithm even when the records are large. This is to have each element of the array store a pointer to a record rather than store the record itself. In this implementation, a swap operation need only exchange the pointer values; the records themselves do not move. The following figure shows an example of Selection Sort:

1	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***Remarks***
65	70	75	80	50	60	55	85	45	find the first smallest element
i								j	swap a[i] & a[j]
45	70	75	80	50	60	55	85	65	find the second smallest element
	i			j					swap a[i] and a[j]
45	**50**	75	80	70	60	55	85	65	Find the third smallest element
		i				j			swap a[i] and a[j]
45	**50**	**55**	80	70	60	75	85	65	Find the fourth smallest element
			i		j				swap a[i] and a[j]
45	**50**	**55**	**60**	70	80	75	85	65	Find the fifth smallest element
				i				j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	80	75	85	70	Find the sixth smallest element
					i			j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	75	85	80	Find the seventh smallest element
						i j			swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	85	80	Find the eighth smallest element
							i	j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	**80**	**85**	The outer loop ends.

Figure 5. Error! No text of specified style in document.**.2: An illustration of Selection Sort to sort an array in increasing order**

```
Algorithm (Selection Sort):
Input: A (array), N (#elements)
                    $i = 0$
Step 1: Find the smallest element from the positions starting at index $i$ to $N-1$. Swap the smallest element with the element at index $i$.
Step 2: $i = i+1$. If $i < N-1$ go to step 1
```

1.5 Insertion Sort

Consider again the problem of sorting a pile of phone bills for the past year. A fairly natural way to do this might be to look at the first two bills and put them in order. Then take the third bill and put it into the right order with respect to the first two, and so on. As you take each bill, you would add it to the sorted pile that you have already made. This naturally intuitive process is the inspiration for our first sorting algorithm, called **Insertion Sort**. Insertion Sort iterates through a list of records. Each record is inserted in turn at the correct position within a sorted list composed of those records already processed.

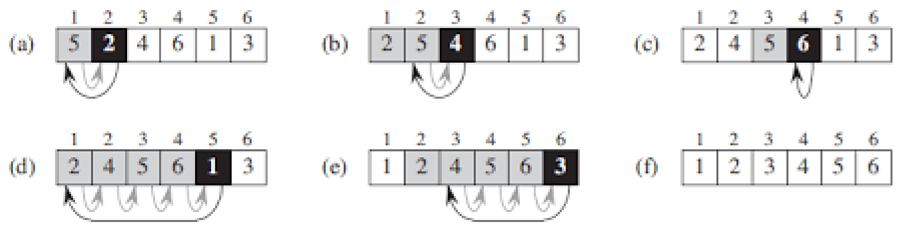
Figure 5. Error! No text of specified style in document.**.3: An illustration of Insertion Sort**

It is to notice that a list is correctly sorted as quickly as can be done. Furthermore, insertion sort remains an excellent method whenever a list is nearly in the correct order and few entries are many positions away from their correct locations.

```
Algorithm (Insertion Sort):
Input: A (array), N (#elements)
$i = 1$
Step 1: $v = A[i]$
Compare $v$ backwards with all previous (down to 0 index) elements.
If a previous element is larger shift it forward otherwise stop
comparing.
Step 2: $i = i + 1$. If $i < N$ go to step 1.
```

1.6 Searching

Organizing and retrieving information is at the heart of most computer applications, and searching is surely the most frequently performed of all computing tasks. Search can be viewed abstractly as a process to determine if an element with a particular value is a member of a particular set. The more common view of searching is an attempt to find the record within a collection of records that has a particular key value, or those records in a collection whose key values meet some criterion such as falling within a range of values.

We can define searching formally as follows. Suppose that we have a collection **L** of *n* records of the form

$$(k_1, I_1), (k_2, I_2), \ldots, (k_n, I_n)$$

Where I_j is information associated with key k_j from record j for $1 \leq j \leq n$. Given a particular key value K, the **search problem** is to locate a record (k_j, I_j) in **L** such that $k_j = K$ (if one exists). **Searching** is a systematic method for locating the record (or records) with key value $k_j = K$.

A **successful** search is one in which a record with key $k_j = K$ is found. An **unsuccessful** search is one in which no record with $k_j = K$ is found (and no such record exists).

Searching in array are basically two types name sequential or linear search and binary search. Sequential or linear search is applied when a given array is unsorted but it works for sorted array also. On the other hand, the binary search is applicable if the array is sorted. Both the sorting algorithms are described details in the following sections.

1.7 Sequential or Linear Search

Beyond doubt, the simplest way to do a search is to begin at one end of the list and scan down it until the desired key is found or the other end is reached. The simplest form of search is sequential or linear search which has already been presented in **Chapter 2**. In linear search, the major consideration is whether K is in list **L** at all. Thus, we have $n + 1$ distinct possible events: That K is in one of positions 0 to $n - 1$ in **L** (each position having its own probability), or that it is not in **L** at all.

```
Algorithm (Linear Search):
Input: Array, #elements, item (to search)
Start with the element at index = 0
Step 1: Compare the element at index with item. If it is equal to
item then return index with status "Found" otherwise go to step 2.
Step 2: Increase index by 1. If index is less than #elements go to
step 1 otherwise return -1 with status "Not found".
```

1.8 Binary Search

Sequential search is easy to write and efficient for short lists, but a disaster for long ones. Imagine trying to find the name “Amanda Thompson” in a large telephone book by reading one name at a time starting at the front of the book! To find any entry in a long list, there are far more efficient methods, provided that the keys in the list are already **sorted** into order.

One of the best methods for a list with keys in order is first to compare the target key with one in the center of the list and then restrict our attention to only the first or second half of the list, depending on whether the target key comes before or after the central one. With one comparison of keys we thus reduce the list to half its original size. Continuing in this way, at each step, we reduce the length of the list to be searched by half. In only twenty steps, this method will locate any requested key in a list containing more than a million keys.

The method we are discussing is called binary search. This approach of course requires that the keys in the list be of a scalar or other type that can be regarded as having an order and that the list already be completely in order. The following example (in figure 5.4) illustrates the simulation of Binary Search.

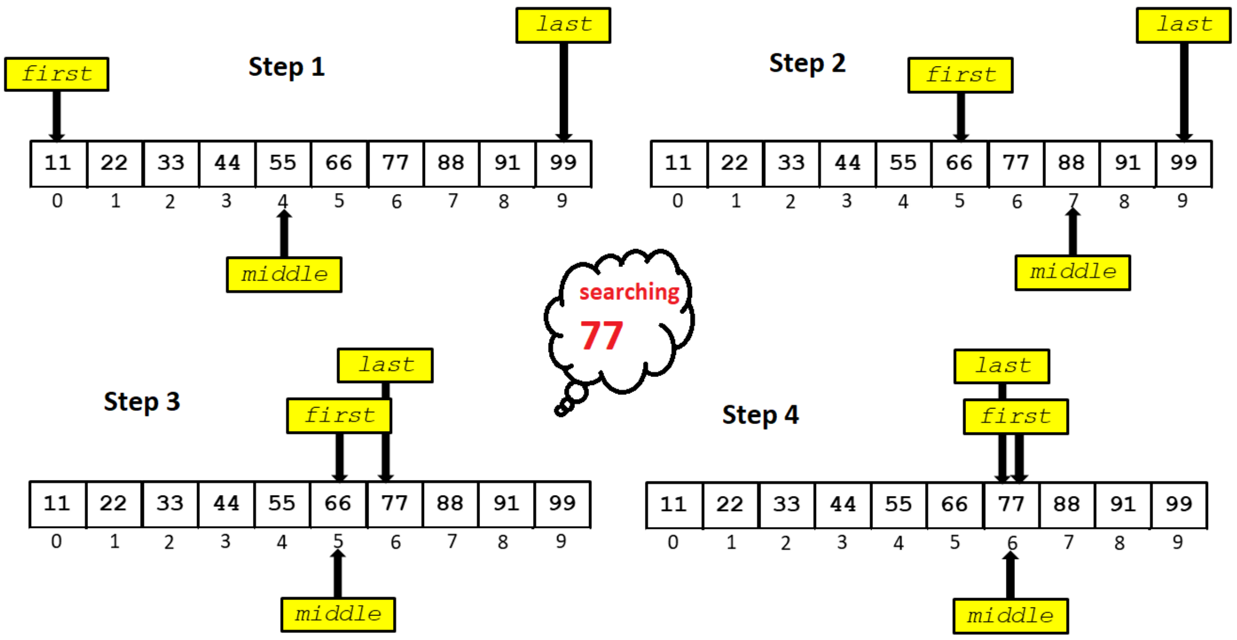

Figure 5.4: An example of Binary Search

The binary search discussed here is appropriate only for list elements stored in a sequential array-based representation. After all, how can you efficiently find the middle point of a linked list? However, you will know of a structure that allows you to perform a binary search on a linked data representation, the binary search tree. The operations used to search in a binary search tree will be discussed in **Chapter 7**.

```
Algorithm (Binary Search):
Input: Array, #elements (N), value(to search)
first= 0 and last= N-1
Step 0:
middle = $\lfloor (first + last)/2 \rfloor$
Step 1:
If low>high exit with status Not Found.
If $Array[middle] = value$ then return $middle$ with status "Found"
If $Array[middle] < value$ then update $first$ by middle+1 and repeat step 0.
If $Array[middle] > value$ then update last by $middle - 1$ and repeat step 0.
```

1.9 References

- Kruse, R. L., Ryba, A. J., & Ryba, A. (1999). Data structures and program design in C++ (p. 280). New York: Prentice Hall.
- Shaffer, C. A. (2012). Data structures and algorithm analysis. Update, 3, 0-3.
- Dale, N. B. (2003). C++ plus data structures. Jones & Bartlett Learning.