# Unit 7 Chapter 10 Algorithm

**Algorithm: a process or set of rules to be followed in calculations or other problem-solving operations.**

**Searching** and **sorting** are two of the most fundamental and widely encountered problems in computer science.

The goal of search is to **ﬁnd a particular object in this collection or to recognize that the object does not exist in the collection.**

Often the **objects have key values on which one searches and data values which correspond to the information one wishes to retrieve once an object is found**.

The collection of objects is often **stored in a list or an array**. Given a collection of n objects in an array A[1..n], the i-th element A[i] corresponds to the key value of the i-th object in the collection.

**The objects are sorted by key value.**

Diﬀerent algorithms for search are required if the data is sorted or not.

## Unordered Linear Search

In order to determine that an object does not exist in the collection, one needs to search through the entire collection.

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | 34 | 16 | 12 | 11 | 54 | 10 | 65 | 37 |

Consider executing the Unordered-Linear-Search algorithm on this array while searching for the number 11. The **ﬁrst four elements would need to be examined until the fourth element containing the value 11 is found.**

In analyzing the performance of search algorithms, we will consider these “**examination counts” as a measure of the performance of such algorithms.**

## Ordered Linear Search

Now suppose that the **given array is sorted**. In this case, **one need not necessarily search through the entire list to ﬁnd a particular object or determine that it does not exist in the collection**.

A simple modiﬁcation of the above algorithm yields the **ordered linear search algorithm**. Note that while scanning the array from **left-to-right** (**smallest-to-largest values**), **a search can now be terminated early if and when it is determined that the number being sought (and as yet not found) is less than the element currently being examined.**

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | 10 | 11 | 12 | 16 | 34 | 37 | 54 | 65 |

Consider executing the **Ordered-Linear-Search** on this array while searching for the number 11. In this case, the **ﬁrst two elements would need to be examined until the second element containing the value 11 is found.**

Now consider executing the Ordered-Linear-Search on this array while searching for the number 13. Note that only the ﬁrst four elements need be examined until the value 16 is encountered and one can deﬁnitively return “13 not found.”

## Chunk Search

**Chunk search**: **The collection is divided into “chunk” or subsets, C, which is equal to √i, where i is the number of element in the original set. The search key is compared to the maximum value of the first chunk. If the search key is greater than the maximum of the first chunk, the second chunk will be evaluated, and so on.**

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | 10 | 11 | 12 | 16 | 34 | 37 | 54 | 65 |

Consider executing the Chunk-Search on the above array while searching for the **number 34** and using c **= 3** (i.e., chunks of size 3).

determine if 34 is in the ﬁrst chunk, the third element (at the end of the ﬁrst “chunk”) with value 12 must be examined.

Since **34 > 12**, we next examine the sixth element (at the end of the second “chunk”) with value 37.

**Since 34 < 37, we conclude that the value 34, if it exists in the array, must be contained in the second chunk**. We then simply execute Ordered-Linear-Search on the subarray A[4..6] consisting of the three elements in the second chunk, eventually ﬁnding 34 in the ﬁfth position.

Now consider executing Chunk-Search on the above array while searching for the number 33. Chunk-Search would behave exactly as described above, except that the call to Ordered-Linear-Search would return “33 not found” when searching the subarray.

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | 10 | 11 | 12 | 16 | 34 | 37 | 54 | 65 |

Consider executing the Binary-Search on this array while searching for the number 34.

To determine if 34 is in the ﬁrst or second half of the array, we split the array in half, considering the fourth element (at the end of the “ﬁrst half”) whose value is 16. Since 34 > 16, we conclude that the number 34, if it exists in the array, must be contained in the second half, i.e., the subarray A[5..8].

We then repeat on the second half of the array, splitting it in half and considering the sixth element whose value is 37. Since 34 < 37, we continue with the subarray A[5..6], ﬁnding the element whose value is 34 in the next iteration.

## Analysis of Algorithms

In computer science, the **time complexity** is the computational complexity that **describes the amount of time it takes to run an algorithm.** Time complexity is commonly **estimated by counting the number of elementary operations performed by the algorithm**, supposing that each elementary operation takes a fixed amount of time to perform.

Since an algorithm's running time may vary among different inputs of the same size, **one commonly considers the worst-case time complexity, which is the maximum amount of time required for inputs of a given size.** Less common, and usually specified explicitly, is **the average-case complexity, which is the average of the time taken on inputs of a given size** (this makes sense because there are only a finite number of possible inputs of a given size). In both cases, the time complexity is generally expressed as a **function of the size of the input**. Therefore, the time complexity is commonly expressed using **big O notation, typically O(n), O(log2n), O(n2).**

Developing a solution to some problem typically involves at least four steps:

1. **designing an algorithm or step-by-step procedure for solving the problem,**
2. **analyzing the correctness and eﬃciency of the procedure,**
3. **implementing the procedure in some programming language, and**
4. **Testing the implementation.**

The absolute execution time of an algorithm depends on many factors:

* **the size of the input (searching through a list of length 1,000 takes longer than searching through a list of length 10),**
* **the algorithm used to solve the problem (Unordered-Linear-Search is inherently slower than Binary-Search),**
* **programming language used to implement the algorithm (interpreted languages such as Basic are typically slower than compiled languages such as C++),**
* **the quality of the actual implementation (good, tight code can be much faster than poor, sloppy code), and**
* **the machine on which the code is run**

**When analyzing the eﬃciency of an algorithm, one often performs a worst case and/or an average case analysis**. A worst case analysis aims to determine the slowest possible execution time for an algorithm.

A worst case analysis is useful because it tells you that no matter what, the running time of the algorithm cannot be slower than the bound derived. An algorithm with a “good” worst case running time will always be “fast.” On the other hand, an average case analysis aims to determine how fast an algorithm is “on average” for a “typical” input. It may be the case that the worst case running time of an algorithm is quite slow, but in reality, for “typical” inputs, the algorithm is much faster.

Since worst case analyses are

1. interesting in their own right,
2. easier to perform than average case analyses, and
3. often indicative of average case performance, worst case analyses tend to be performed most often.

### Linear Search

This algorithm **simply iterates through the array**, **examining its elements one-by-one**. Note that each iteration of this algorithm takes some constant amount of time to execute, dependent on the programming language used, the actual implementation, the machine on which the code is run, etc.

For searching and sorting algorithms, **we shall consider the number of array elements which must be examined as an indicator of the performance of the algorithm**.

On an input of size **n**, t**he number being sought will be compared to each of the n elements in the array for a total of n array examinations**. Let **T(n) be the function of n which describes the running time of an algorithm.**

**T(n) = n**

### Chunk or Jump Search

One element must be **examined for each chunk considered**, for a maximum of **n/c** such examinations on **an array of size n using chunks of size c**. Ordered-Linear-Search will then be performed on **a chunk of size c (at most)**, engendering c further element examinations.

**T(n) = n/c + c**

In order to determine the optimal chunk size c; i.e., **the chunk size which would minimize the overall running time of Chunk-Search (in the worst case)**.

Suppose that one were to run Chunk-Search using a **very small value of c**. Our chunks would be small, so there would **be lots of chunks**. Much of the time would be spent trying to ﬁnd the right chunk, and very little time would be spent searching for the element in question within a chunk. Consider the extreme case of c = 1: in the worst case, n/c = n/1 = n element examinations would be spent trying to ﬁnd the right chunk while only c = 1 examinations would be spent searching within a chunk for a total of n + 1 examinations.

Now consider using a **very large value of c.** Our chunks would be big, so there would be few of them, and very few element examinations would be spent ﬁnding the right chunk. However, searching for the element in question within a very large chunk would require many such examinations. Consider the extreme case of c = n: in the worst case, **n/c = n/n = 1** element examinations would be spent “ﬁnding” the right chunk (our chunk is the entire list) while c = n examinations would be spent searching within a chunk for a total of n + 1 compares (in the worst case).

Our goal is then to ﬁnd a **c such that n/c (the time spent searching for a chunk) is equal to c (**the time spent searching within a chunk).

**c = n/c**

**c2 = n**

**c = √n**

Thus, the desired chunk size is c = √n, and using this chunk size

**T(n) = n/c + c**

**= (n/√n) + √n**

**= √n +√n**

**= 2√n**

**If n = 1,000,000**, Ordered-Linear-Search would require **1,000,000 element examinations in the worst case**, while **Chunk-Search would require approximately 2,000 examinations in the worst case—Chunk-Search would be 500 times faster.**

### Binary Search

Consider: **Cutting an array of size n in half** yields **n/2**. Cutting this array in half again yields **(n/2)/2 = n/22**= n/4. Cutting the array in half a third time yields ((n/2)/2)/2 = (n/22)/2 = n/23 = n/8. In general, cutting an array in half k times yields an array of size n/2k. How large can k be until n/2k is one? We have

**n/2k = 1**

**n = 2k**

**log2n = k**

**At most log2 n iterations will be performed until the array can no longer be cut in half**, and thus the worst case running time of Binary-Search is

**T(n) = log2n**

**= ln(n)/ln(2)**

In mathematics, the binary logarithm **(log2 n)** is the **power to which the number 2 must be raised to obtain the value n.**

For example, the binary logarithm of 1 is 0, the binary logarithm of 2 is 1, the binary logarithm of 4 is 2, and the binary logarithm of 32 is 5.

The binary logarithm is the logarithm to the base 2. The binary logarithm function is the inverse function of the power of two function.

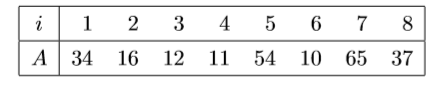
**This is faster still than even Chunk-Search:** on an array of size 1,000,000, Binary-Search would perform 20 comparisons (in the worst case), as compared to 2,000 for Chunk-Search and 1,000,000 for Ordered-Linear-Search!

| Search Algorithm Type | Worst Time | Comment | N = 1000 |
| --- | --- | --- | --- |
| Linear Search | n | Key ≥ Last | 1000 |
| Chunk/Jump Search | √n | When c = n/c | 32 |
| Binary Search | log2n |  | 10 |

**Ordered-Linear-Search, Chunk-Search, and Binary-Search each assume that the underlying array or list which is being searched is already in sorted order.**

## Insertion Sort

Insertion-Sort corresponds to the method that many people use to sort cards as they are being dealt to them. **The ﬁrst card is placed in one’s hand, the second card is compared to the ﬁrst and placed either before or after it**, and so on. In general, one has some i cards, in sorted order, in one’s hand, and the i + 1st card is compared to successive cards (starting from the left or right) until the proper location to “insert” the card is found.



Imagine the numbers 34, 16, 12…, being “dealt” to you, in this order, like cards.

* 1. You would start with the number 34, a trivially sorted list.
  2. Then the number 16 would be inserted into this list, producing the sorted list “q16 34.”
  3. The number 12 would then be inserted into this list, producing “12 16 34,” and so on.

In general**, in phase k of the algorithm, the kth element of the array is inserted into the sorted list formed from the first k − 1 elements of the array**.

Phase in insertion sort:
Processed and unprocessed of values

What is the running time of Insertion-Sort on an array containing n elements?

1. In phase 1 of the algorithm**, the first array element must be examined**.
2. In phase 2 of the algorithm, the **second element must be examined** and **compared to the first (which must therefore be examined).**

In general, in the **kth phase of Insertion-Sort, the kth element must be examined, and in the worst case, it may be compared to each of the previously processed k – 1 elements,** **resulting in k total elements being examined**. Thus, the total number of elements examined (in the worst case) is given by

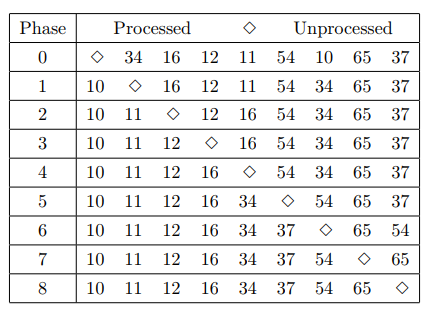
**1 + 2 + 3 + · · · + (n − 2) + (n − 1) + n.**

## Selection Sort

Selection-Sort begins by examining the list of elements, **“selecting” the smallest one**, and **“swapping” it with the first element of the list.**

In phase 2, Selection-Sort selects the smallest element from the remaining n−1 elements and swaps it with the element in position 2. In general, in the kth phase, Selection-Sort selects the smallest element from the remaining.

**n − k + 1** elements and swaps it with the element in position k. The process of selection sort is shown below.



Note that this is the same sum as given by the analysis of Insertion-Sort, only written backwards.

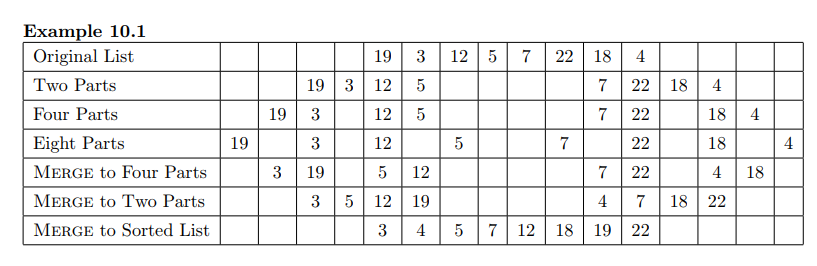
## Merge Sort

Merge-Sort is **a divide-and conquer algorithm**. This is how it works:

Start with an unsorted list of n elements.

* 1. If n = 1, STOP. The list is sorted.
  2. **Break the list into two approximately equal pieces** and apply Merge-Sort to each piece.
  3. **Merge the two sorted sublists to produce the sorted list**.

**Merge takes two sorted lists and combines them by repeatedly comparing the two lowest elements and sending the smaller one off to be the next element in the combined list**.



**Merge-Sort is probably faster than either Insertion-Sort or Selection-Sort**, in the worst case.

| Name | Best | Average | Worst | Average n = 1000 |
| --- | --- | --- | --- | --- |
| Insertion sort | N | N2 | N2 | 106 |
| Selection sort | N2 | N2 | N2 | 106 |
| Merge sort | N log N | N log N | N log N | 9966 |
| Quick sort | N log N | N log N | N2 | 9966 |