Unit 1

Computer Science as Applied Mathematics

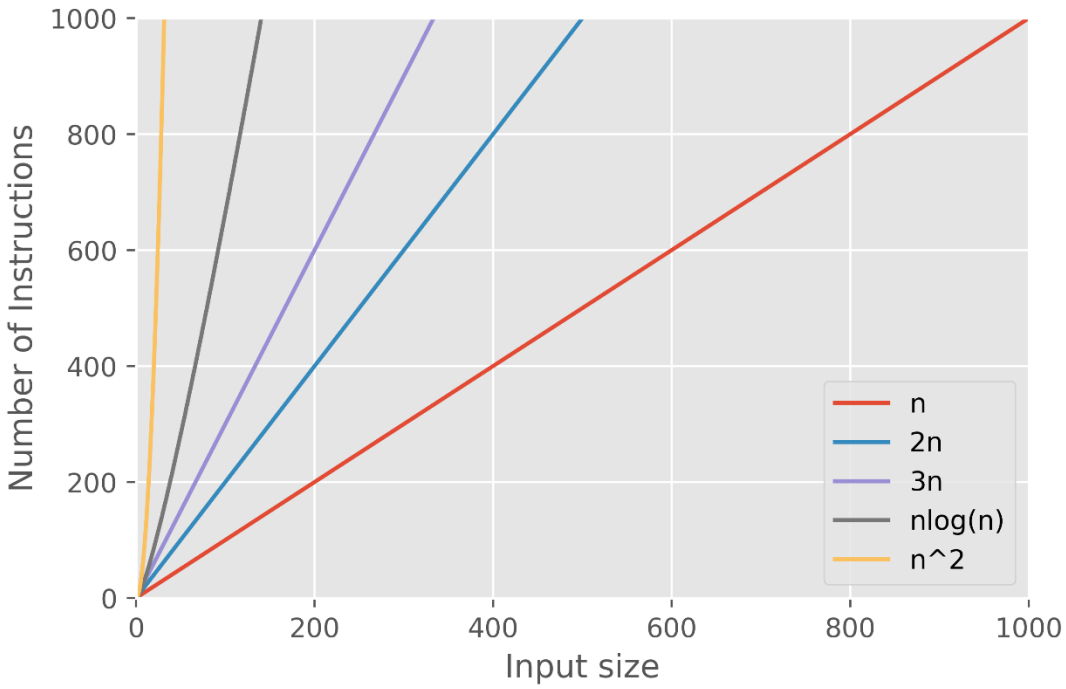
Efficiency of an Algorithm

There are multiple algorithms that can solve the same problem. Whenever this is the case, it is usually desired to use the most efficient solution. In application, computers using inefficient algorithms will cause us frustration as the make our computers slow down!

Example:
Suppose you are in a group that wants to solve the problem of walking to the other side of the room. One algorithm may involve walking in a straight line towards the other side of the room. Another algorithm may involve going to the nearest wall and traveling along the perimeter of the room to get to the other side. Both methods solved the problem, but the first algorithm—traveling in a straight line—was more efficient. A large part of computer science is looking at algorithms to determine how efficient they are.

The efficiency of an algorithm is based on the size of the input. We will always assume an algorithm is given an input of size n and represent its runtime in terms of n. To analyze the efficiency of an algorithm, we analyze how fast the algorithm's runtime grows in relation to its input size. We can classify algorithms into different categories based on this metric.

Usually we speak about the efficiency of an algorithm by classifying based on their *asymptotic complexity.* An algorithm that executes $2n$ instructions will be very similar to an algorithm that needs $3n$ instructions. Even though the latter algorithm technically is a little bit less efficient, you can see that they are still more closely related to each other than they are to an algorithm that requires n^2 instructions.



We say that "*2n* is Big O of n" and *"3n* is Big O of n."
This is often written as *2n* = ***O****(n)* and *3n* = ***O****(n).*

2n and *3n* are different but they are in same category of complexity.
Similarly, $3n^2$=***O****(n^2)* and $100n^2$=***O****(n^2)*.

The important idea is that ***O****(n)* algorithms don't grow as fast as ***O****(n^2)* algorithms, ***O****(n^2)* algorithms don't grow as fast as ***O****(n!)* algorithms, etc…

Constant Time Algorithms: ***O****(1)*
Algorithms that run in Constant time do the same number of steps no matter how large the input grows. These are usually the most efficient algorithms, and they always solve a problem in the same amount of time no matter how big the problem gets!

Example:

> Consider an algorithm that prints the last ten items from a list of n items. No matter how large n is, only the last 10 items are printed. This means the algorithm runs in constant time. (We will assume the list has at least 10 items.)

Linear Time Algorithms: ***O****(n)*
The runtime of linear time algorithms grows linearly as the input grows. That is to say, if the input is of size n, then the algorithm's runtime will be *C*n* for some number *C*. (*4n, 5n, 5.5n, … , 100n, 1000n, etc…)*

Example:

> Consider an algorithm that prints all the items in a list of size n. Printing n items requires at least n steps. That means that as the size of the input, n, grows, the runtime is at least of size n.

Logarithmic Time Algorithms: *O**(log(n))***

Note that the base of the logarithm is 2, NOT 10.

These are very efficient algorithms. You have already seen one example in the Binary Search Algorithm, in which we halved the size of the problem in every iteration. If the original problem was a list of 256 elements, then in the worst case we would halve it to 128 elements, then halve it again to 64 elements, then 32, then 16, then 8, then 4, then 2, and then finally 1. This took *log(256) = 8* steps.

Quadratic Time Algorithms: *O**(n^2)**

Algorithms that run in quadratic time are much less efficient than constant and linear time algorithms. As the input size, n, grows, the runtime is $C*n^2$ for some number *C.*

Example:

> Consider an algorithm that prints a list of n integers n times. It will print the list n times, which requires n operations, but each time it prints the list, it will require n operations to print each of the n integers. This gives us n * n = n^2 operations. This means, as the input size, n, grows, the runtime is at least n^2 operations.

Linear Time Algorithms: O(nlog(n))

This particular class of algorithms shows up a lot in algorithms for sorting lists. We can't make a general purpose sorting algorithm that performs any better than this.

Order them from most to least efficient.

Assuming a single step takes 1 nanosecond (ns) (1 billionth of a second):

Running time	n = 10	n = 100	n = 1,000	n = 100,000	n = 1 Million	n = 100 Million	n = 1 Billion	n = 100 Billion
O(1)	constant	constant	constant	constant	constant	constant	constant	constant
O(LogN)	4 ns	7 ns	10 ns	17 ns	20 ns	27 ns	30 ns	37 ns
O(N)	10 ns	0.1 µs	1 µs	0.1 ms	1 ms	0.1 s	1 s	1 m 40 s
O(NLogN)	34 ns	0.7 µs	10 µs	2 ms	20 ms	2.7 s	30 s	1 h
O(N^2)	0.1 µs	10 µs	1 ms	10 s	17 m	116 d	31 y 7 mo.	3.1 millennia
O(N^3)	1 µs	1 ms	1 s	12 d	32 y	3.17 x 10^7 y	3.17 x 10^10 y	3.17 x 10^16 y
O(2^n)	1 µs	4.02 x 10^13 y	Too large for most calculators					

As you might be able to tell from this table, choosing an inefficient algorithm for solving your problem could make your program take too much time to be useful. These large values for n aren't exaggerated at all: plenty of computers around the world are solving problems at that size and beyond. Nobody wants to wait an entire day for their Google search to finish, so it's important that Google uses an efficient algorithm for giving you your search results.

Classical Algorithms

Linear Search

A very easy-to-understand algorithm is *Linear Search*.
Suppose you were given a list of numbers and you could only look at one number at a time. Someone asks you whether the number "21" was in that list. The *Linear Search* algorithm is simply checking the 1st number, then the 2nd number, and so on, all the way to the very last number. If you ever see the number "21" while you're checking each number, then you can say "yes!"

A computer is not able to simply glace at the list like we can to find 21. In a list, each element has what is called an index. We will say that the index of the first element in a list is 1. The index of the second element is 2, and so on. (In some cases, people start the indexing at 0, but we will use 1.)

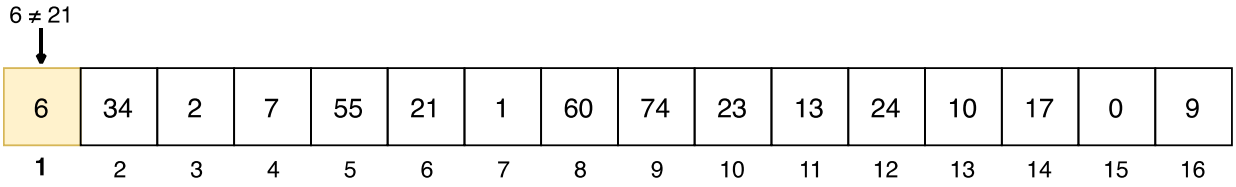
Example:
Suppose we want to see where 21 is the following list, *L*:

6	34	2	7	55	21	1	60	74	23	13	24	10	17	0	9

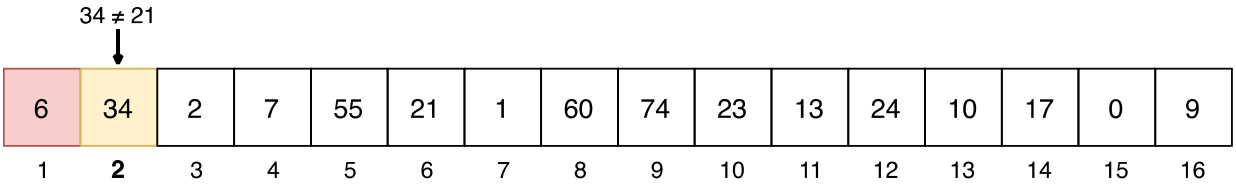
We will give the first number the index “1,” the second number the index “2,” and so on.

6	34	2	7	55	21	1	60	74	23	13	24	10	17	0	9
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

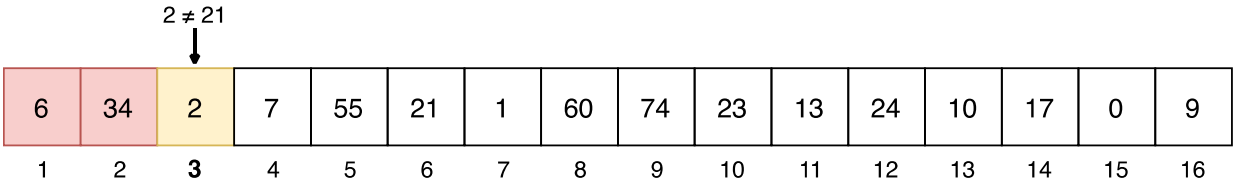
Remember, our problem specified that we can only look at one number at a time. This limitation is imposed because computers can usually only “think about” one number in a list at a time. The first item is 6.



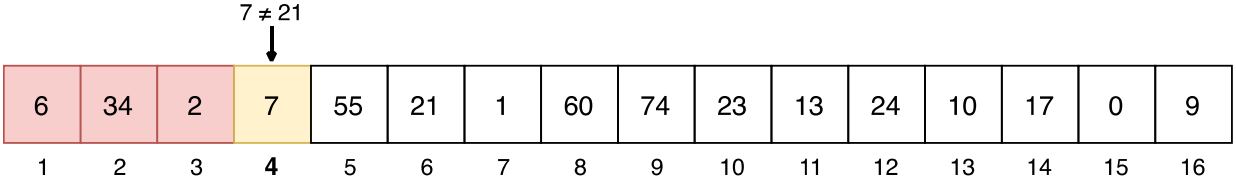
We compare this 1st item, 6, with the number we are looking for: 21. They are not equal. So, we move on to index 2.



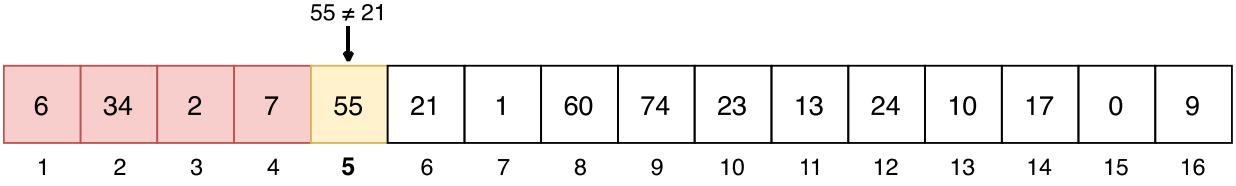
The 2nd number is 34, but this does not equal 21, so we move on again.



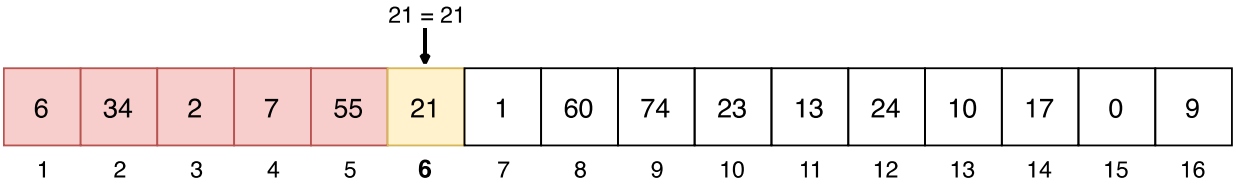
The 3rd number is 2, which does not equal 21. We check the next index.



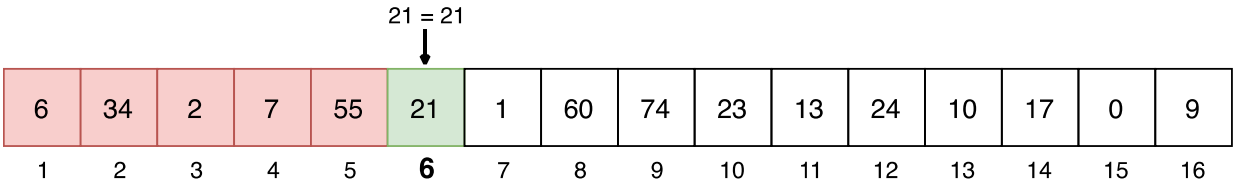
The element with index 4 is 7, but 7 isn't 21, so we still haven't found the answer.



Again, we increment our index to 5. The number here still isn't what we are looking for.



Now, when we increment the index again, we see that the number at this index is 21. We have found our target, so we can say "yes, the number was in the list."



This is a very simple way to look for something in a list. We set some index *i=1*. As long as *i* is less than the length of the list, we check if L_i is equal to 21. If it is, then the answer to the problem is "yes" and we stop the algorithm. If not, we increase *i by 1.*Then at the end, if we never found 21, the answer is "no."

In this example, we found our number quickly. However, if we had been looking for 9, we would have had to search through the entire list. In real-world applications, computers can deal with lists with billions of items, so this can be a problem sometimes.

Binary Search

Binary search, also known as half-interval search or logarithmic search, is a searching algorithm that finds the position of a target value within a sorted list of data. (This algorithm will not work if the list is not sorted.)

We compare the target value to the middle item in the list, *k*. If the target value is larger than k, the half of the list which contains the larger values is eliminated, and we compare our target with the middle item in the other half of the list. This continues until we either find an item that matches our target or one of the halves being empty, meaning that no match can be found.

Example:
Suppose we want to see where 17 is in the following sorted list:

0	2	3	5	6	8	11	12	16	17	20	24	26	27	28	30

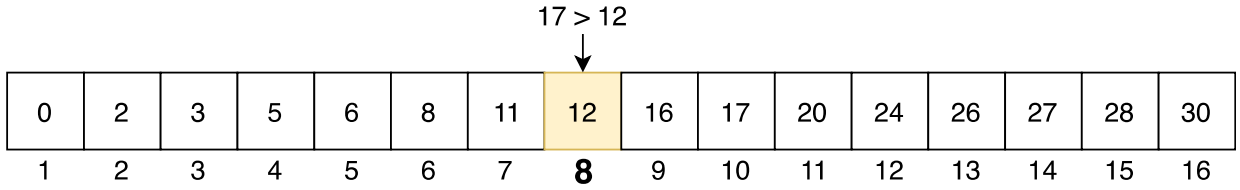
We will again assign an index (starting at 1) to each element of the list.

0	2	3	5	6	8	11	12	16	17	20	24	26	27	28	30
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

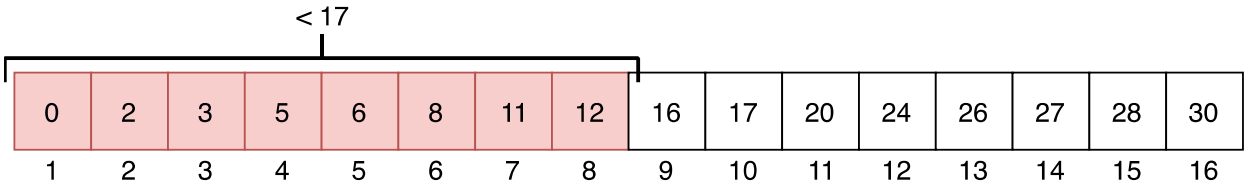
First, we compare our target value, 17, to the middle element of the list. To find the middle element, we have to find the midpoint between the first element and the last element. The index of our first element is 1, and the index of our last element is 16. We divide the sum of those two numbers by 2 and take the floor of that number.

$$\lfloor (1 + 16)/2 \rfloor = \lfloor 17/2 \rfloor = \lfloor 8.5 \rfloor = 8$$

We will compare our target value, 17, to the 8th element in the list.



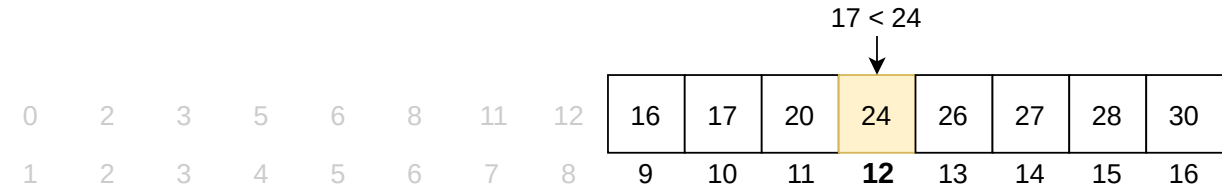
Since 17 is greater than 12, we know we only need to search in the right-hand half of the list (remember: this algorithm only applies to lists that are sorted. If this list were unsorted, we wouldn't be able to guarantee that the number we are looking for was on the right-hand side).



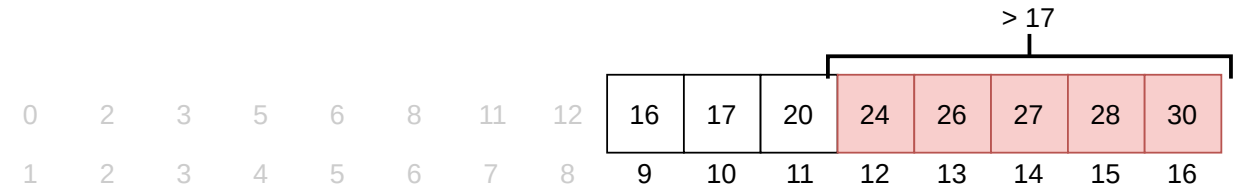
Now, we compare 17 to the middle element in the remaining list of elements. We find the floor of the midpoint between index 9 and index 16.

$$\lfloor (9 + 16)/2 \rfloor = \lfloor 25/2 \rfloor = \lfloor 12.5 \rfloor = 12$$

We will compare our target value to the value at the 12th index of the list.



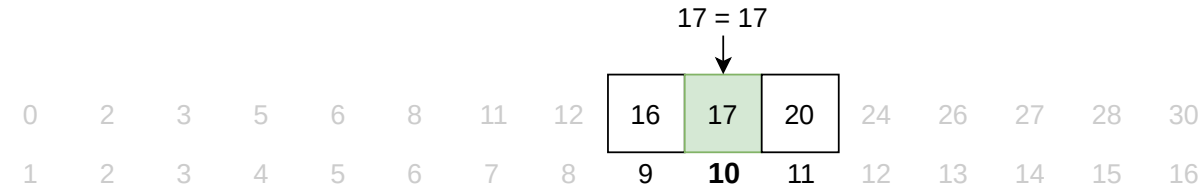
We can see that 17 is less than 24, so the half of the list that contains values greater than 17 is eliminated.



We compare 17 to the middle element in the remaining list. We find this by finding the floor of the midpoint between index 9 and index 11.

$$\lfloor (9 + 11)/2 \rfloor = \lfloor 20/2 \rfloor = \lfloor 10 \rfloor = 10$$

We compare 17 to the 10th element in the list.



Since 17 = 17, we know that we found our target value, which is the element at the 10th index of the list.

Binary search will return the number 10, the position of our target value.

In summary, binary search is like how you might look up someone's number in a phone book: you look in the middle, and if the name you're looking for comes after the names in the middle, then you look halfway between the middle and the end. You keep getting closer and closer to where their name is by comparing it to the names on the page you're looking at.
Nobody would try to find a name in a phonebook by checking every single page. They would take advantage of the names being sorted and do something similar to this binary search!

Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path between two nodes in a graph. A **graph** is an abstract way that data is stored. It consists of **nodes**, which hold the data, and **edges**, which connect the nodes together. Sometimes, edges have values associated with them. We call this the **weight** or **cost** of the edge. Dijkstra's algorithm finds the shortest path, the least costly path, from one given start node, *s*, to an ending node, *t*.

Given the following graph, Dijkstra's algorithm would tell us that the shortest path from s to t is s -> d -> f -> e -> t with a cost of 10: