Find your way here

CSC 250: Foundations of Computer Science I

Fall 2023 - Lecture 8

Amitabha Dey

Department of Computer Science
University of North Carolina at Greensboro

How to measure algorithm efficiency?

$$a + b + c = n$$

Find all sets of nonnegative integers (a, b, c) that sum to integer n $(n \geq 0)$

Bob's Solution

1. Try all combinations of (a, b, c)
2. If $a + b + c = n$, print(a, b, c)

Example: $n = 3$

$a \lfloor 3$ $b \lfloor 3$ $c \lfloor 3$

$(0,0,3)$ $(1,1,1)$
$(0,1,2)$ $(1,2,0)$
$(0,2,1)$ $(2,0,1)$
$(0,3,0)$ $(2,1,0)$
$(1,0,2)$ $(3,0,0)$

Alice's Solution

1. For all (a, b) set $c = n - (a + b)$
2. If $c \geq 0$, print(a, b, c)

Example: $n = 3$

$a \lfloor 3$ $b \lfloor 3$

$c = n - (a + b) = -3$

$(0,0,3)$ $(1,1,1)$
$(0,1,2)$ $(1,2,0)$
$(0,2,1)$ $(2,0,1)$
$(0,3,0)$ $(2,1,0)$
$(1,0,2)$ $(3,0,0)$

Image credit - Reducible @ YouTube

Both Bob and Alice try to solve the problem and present their solution.

Bob tries every combination of a, b, and c and whenever their sum is n, he prints it out.

Alice tries to solve the problem in another way. Since the value of n is given to us, she tries every combination of a and b up to n, and then for every combination set c as n - (a+b).

If c value is non-negative, she finds all the valid combination.

How to measure algorithm efficiency?

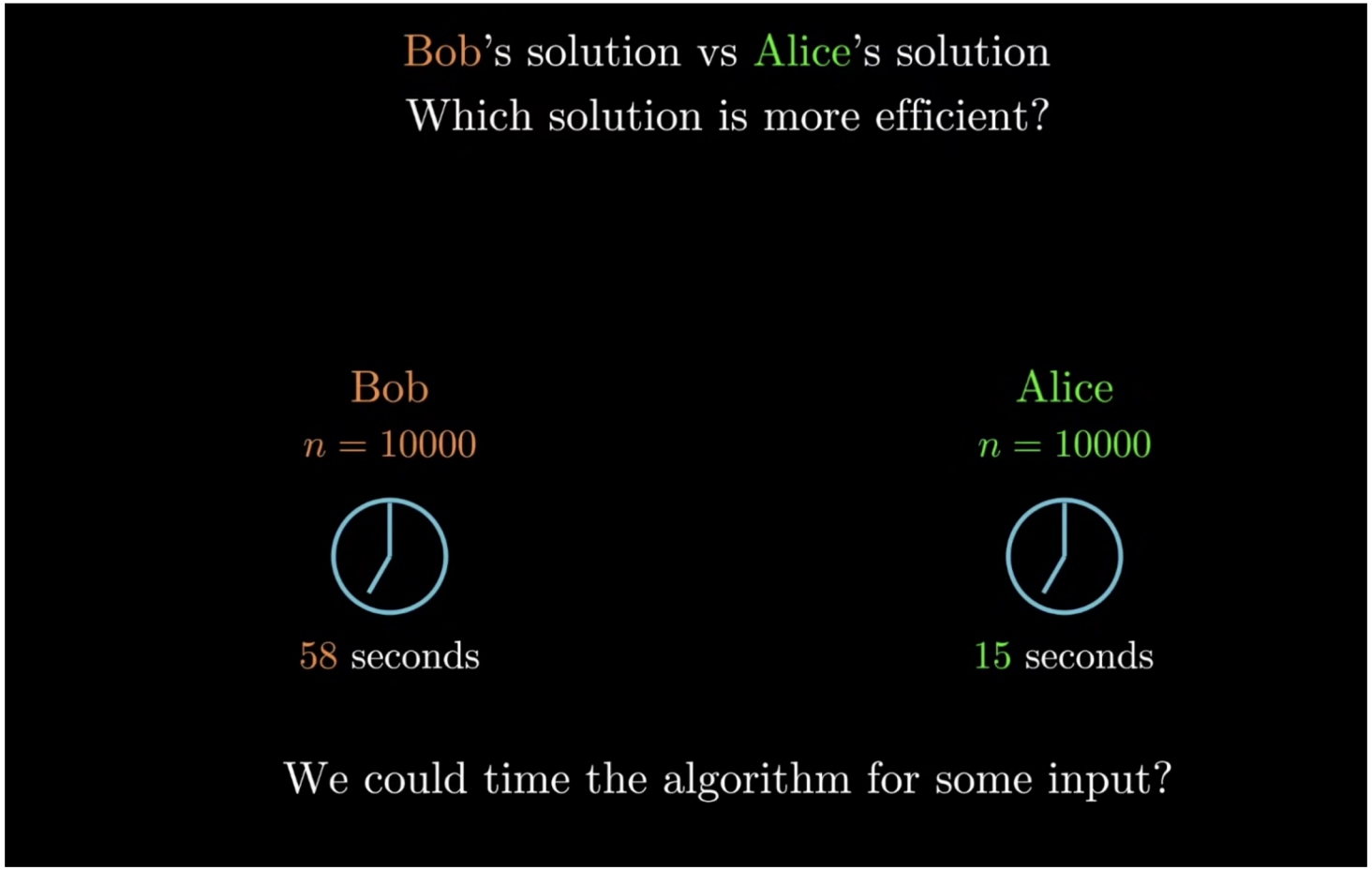


Image credit - Reducible @ YouTube

How do we objectively define which student has the better algorithm?

The simplest idea would be to physically time each of the algorithm for a particular input.

Let's say in a particular computer, Bob's algorithm ran for 58 seconds, while Alice's algorithm ran for 15 seconds.

How to measure algorithm efficiency?

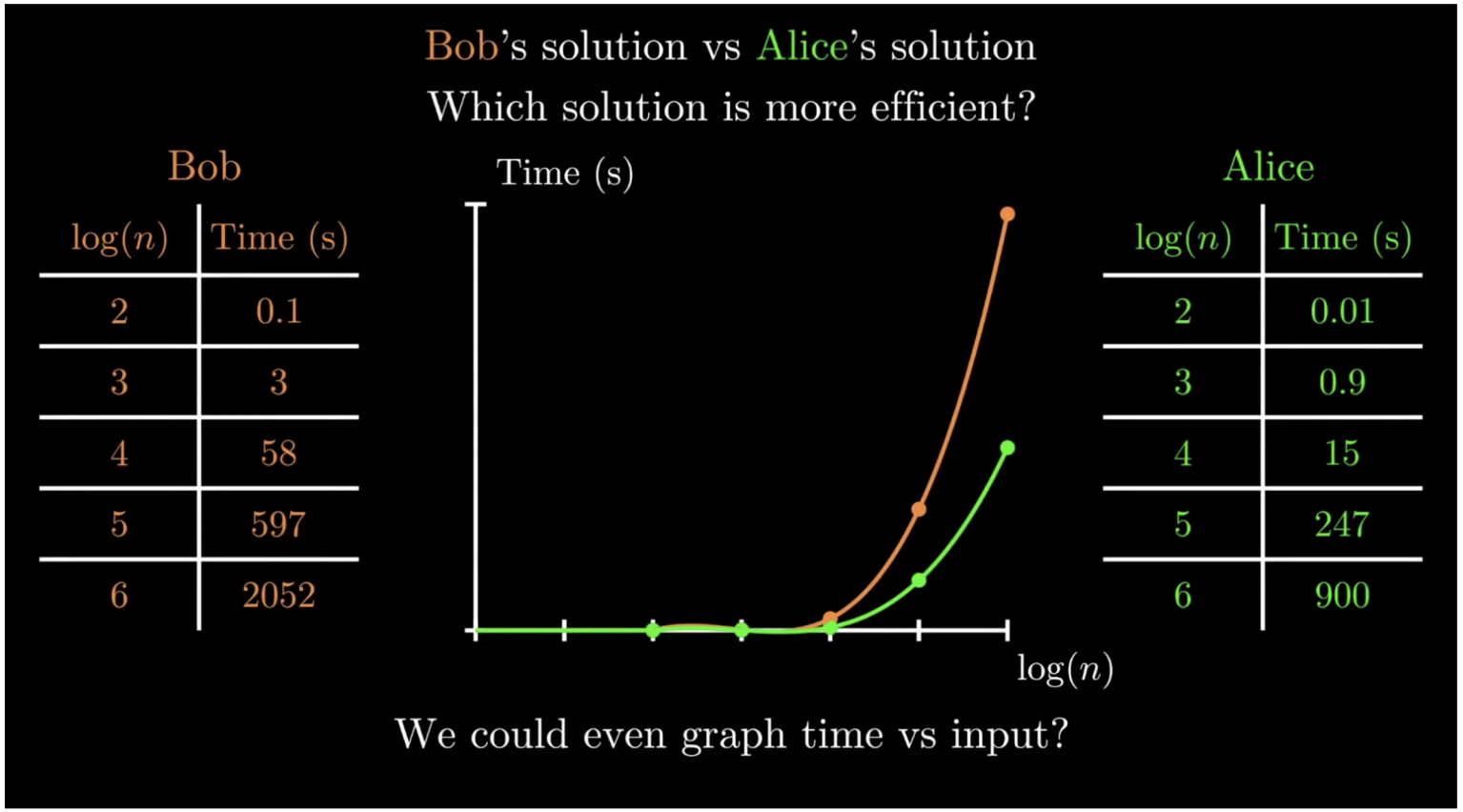


Image credit - Reducible @ YouTube

We could put these values in a graph and see approximately how the time grows as a function of the input.

We could standardize x-axis on a logarithmic scale.

It's visible that Alice's solution is better than Bob's.

How to measure algorithm efficiency?

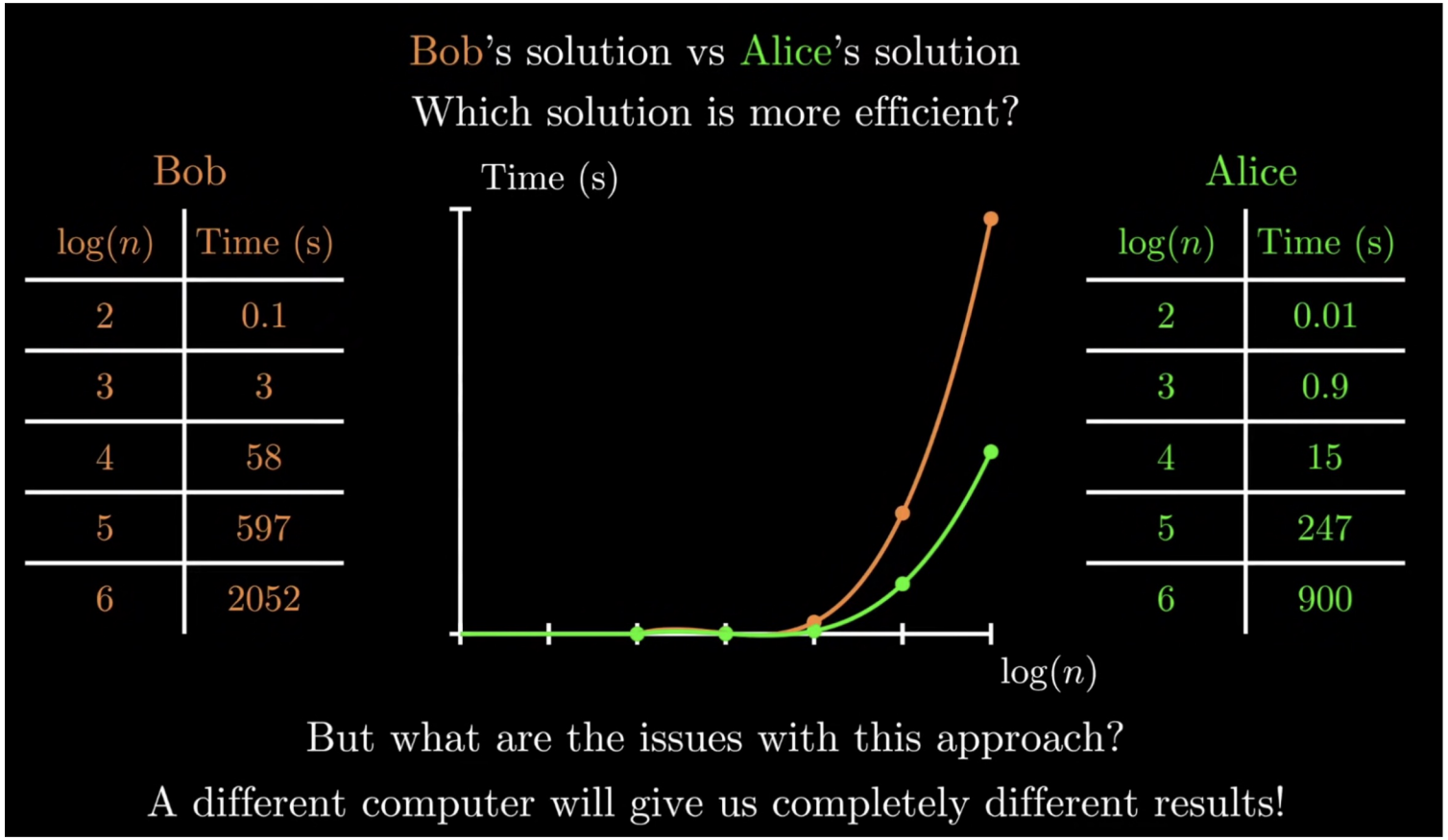


Image credit - Reducible @ YouTube

All these time measurements are taken on a specific computer. On a different computer, all the data points could look totally different.

⇨ **Machine dependence**

A true and pure efficiency measurement should be **Machine independent**! We want to be able to tell how fast the algorithm is regardless of what machine it runs on.

How to m

Bob

log(n)	Time (s
2	0.1
3	3
4	58
5	597
6	2052

A differe

Machine Independent

- The evaluation of efficiency should be as machine independent as possible.
- It is not useful to measure how fast the algorithm runs as this depends on which particular computer, OS, programming language, compiler, and kind of inputs are used in testing
- Instead,
  - we count the number of *basic operations* the algorithm performs.
  - we calculate how this number depends on the size of the input.
- A basic operation is an operation which takes a constant amount of time to execute.
- Hence, the efficiency of an algorithm is the number of basic operations it performs. This number is a function of the input size **n**.

are taken on a
ent computer, all
ally different.

pendence

easurement should
We want to be able
is regardless of

Image credit - Reducible @ YouTube

How to measure algorithm efficiency?

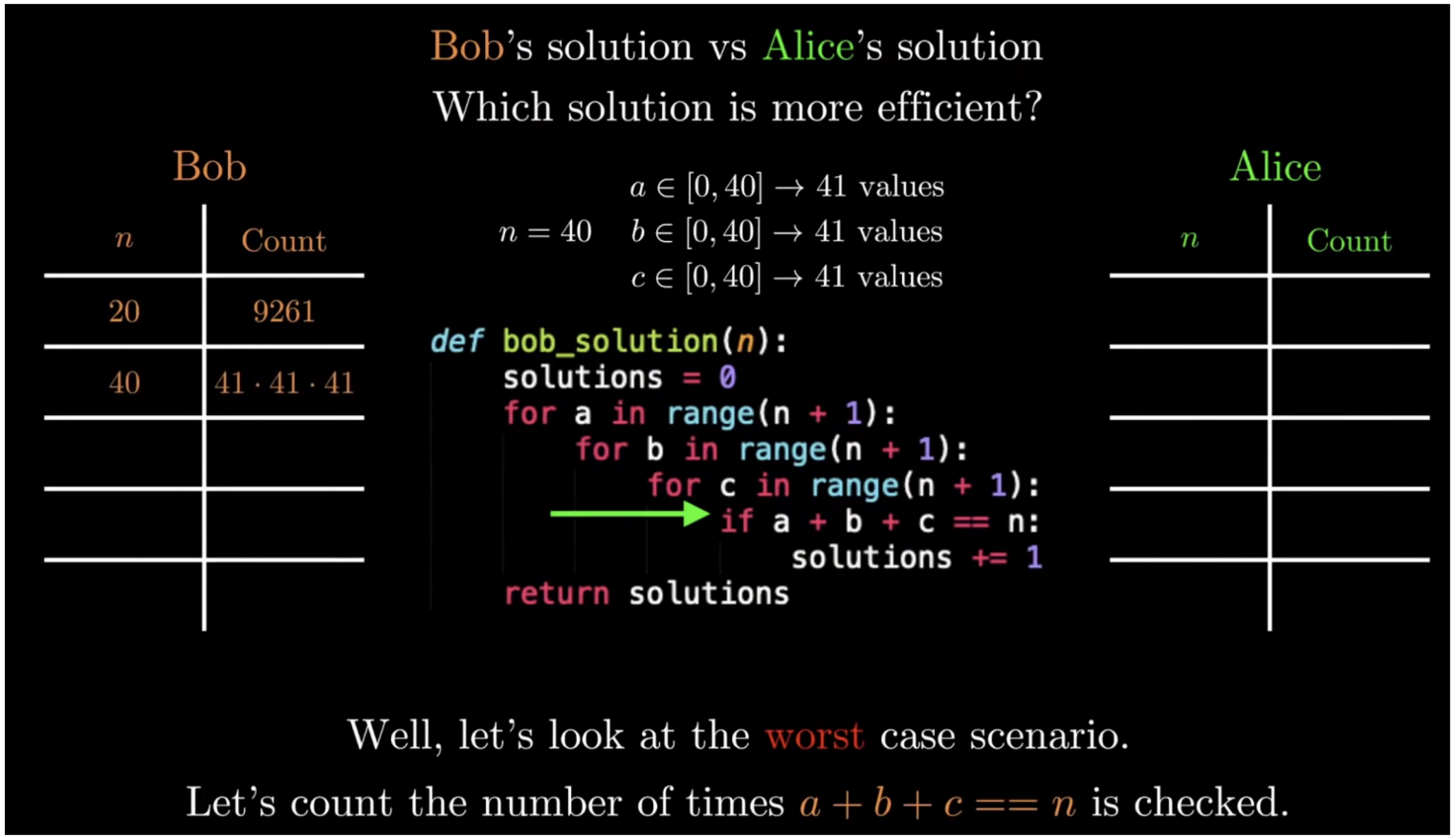


Image credit - Reducible @ YouTube

One machine independent way is to count the operations in the program and see how it changes as the input increases.

We want to focus on the **worst-case scenario** for good representation.

Let's say n = 20.

In Bob's program, we can look at the if statements, and three for loops and inspect how many times it runs.

How to measure algorithm efficiency?

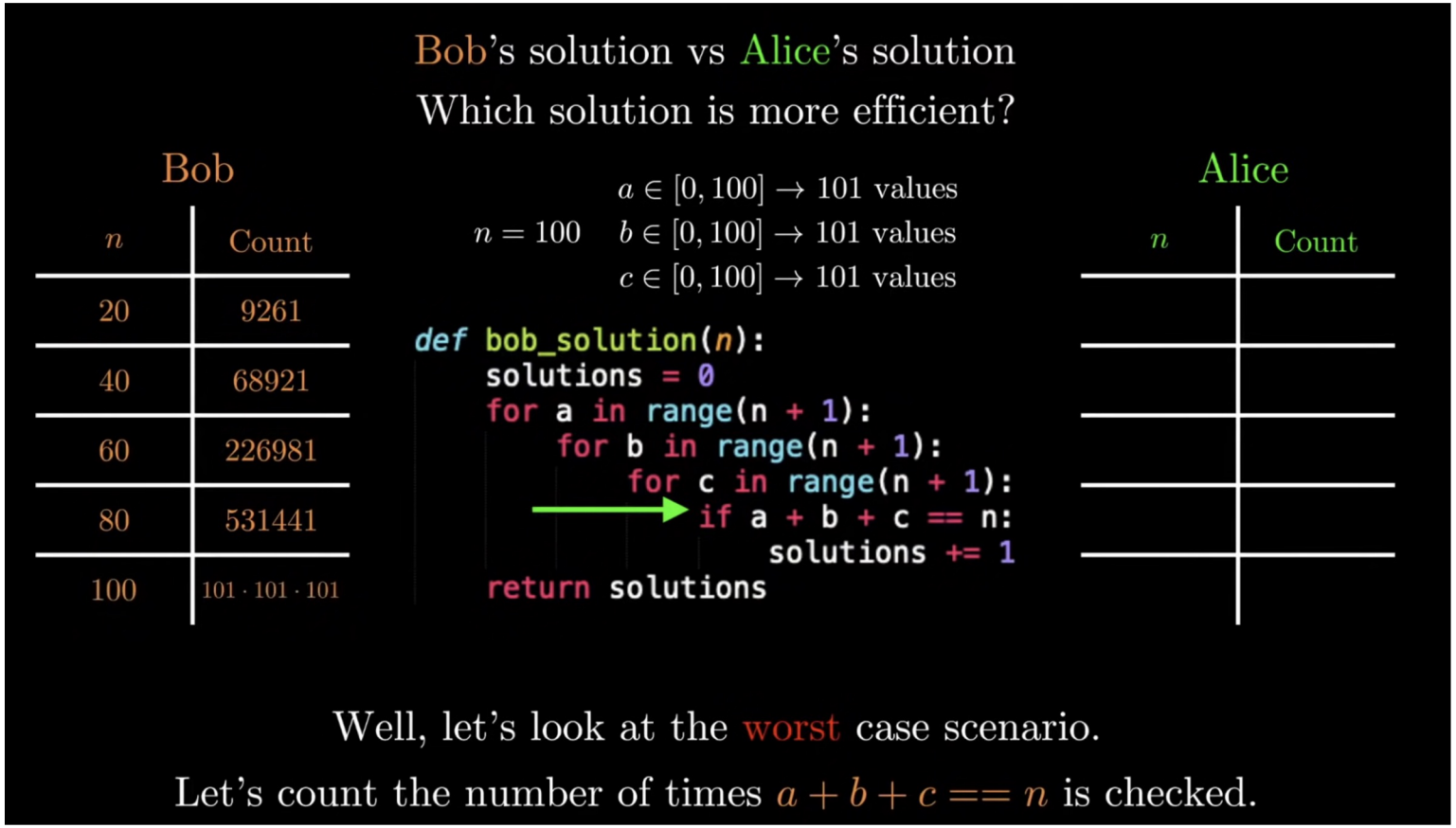


Image credit - Reducible @ YouTube

One machine independent way is to count the operations in the program and see how it changes as the input increases.

We want to focus on the **worst-case scenario** for good representation.

Let's say n = 20.

In Bob's program, we can look at the if statements, and three for loops and inspect how many times it runs.

How to measure algorithm efficiency?

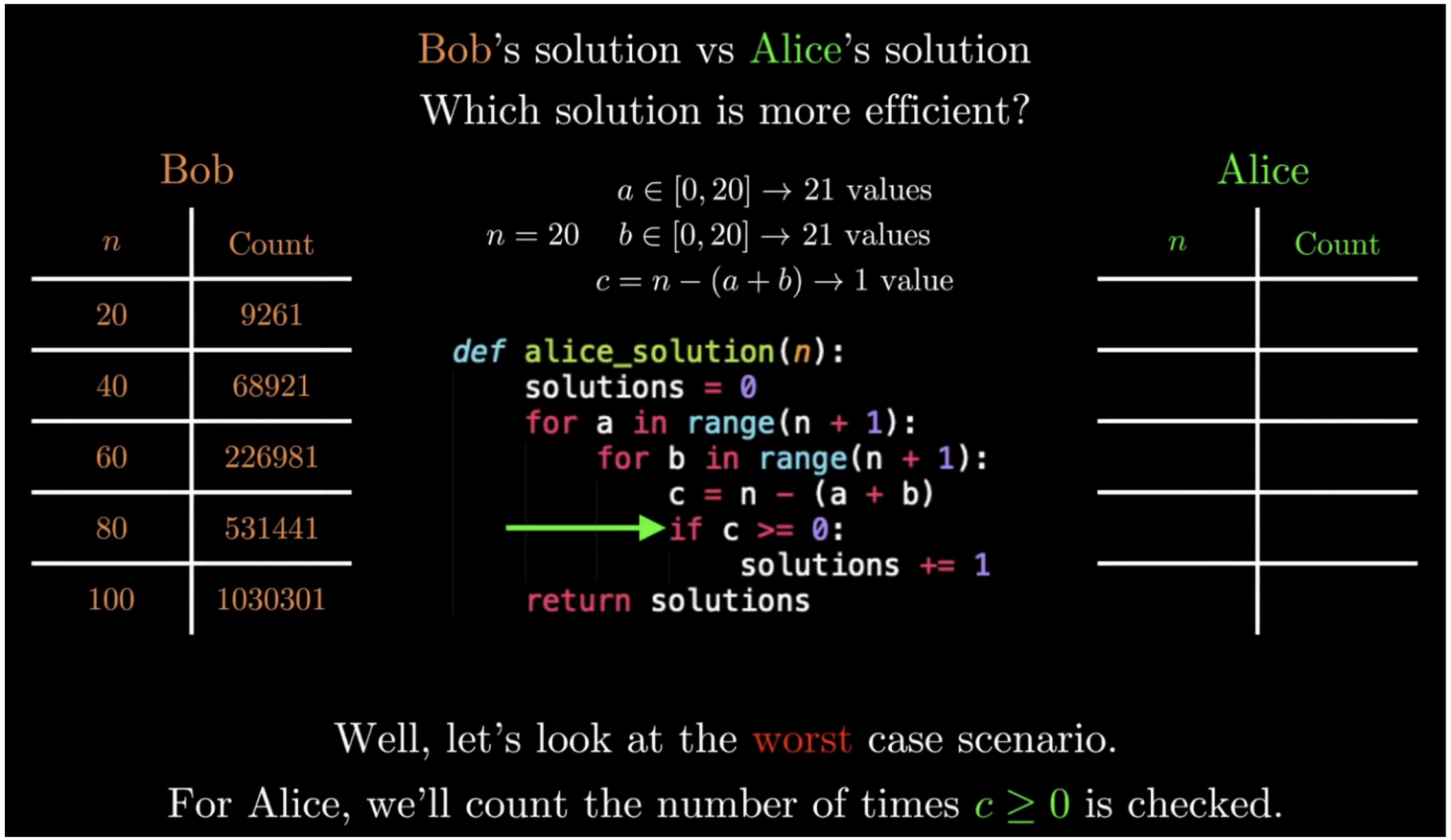
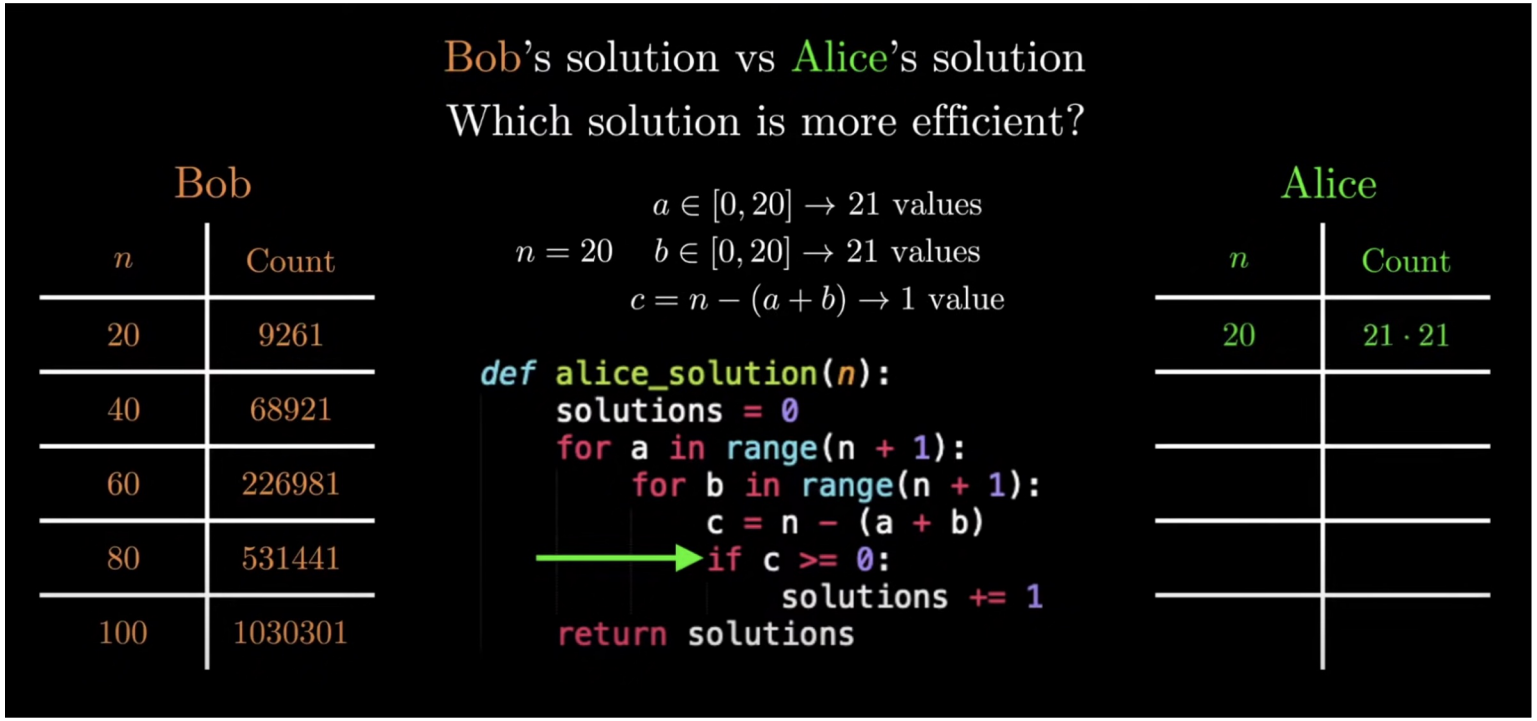


Image credit - Reducible @ YouTube

The defining characteristic about Alice's program is that for each pair of a and b, we have exactly one c value.

How to measure algorithm efficiency?



The defining characteristic about Alice's program is that for each pair of a and b, we have exactly one c value.

Image credit - Reducible @ YouTube

How to measure algorithm efficiency?

Bob's solution vs Alice's solution

Which solution is more efficient?

Bob

n	Count
20	9261
40	68921
60	226981
80	531441
100	1030301

$n = 100$

$a \in [0, 100] \rightarrow 101$ values

$b \in [0, 100] \rightarrow 101$ values

$c = n - (a + b) \rightarrow 1$ value

```
def alice_solution(n):
    solutions = 0
    for a in range(n + 1):
        for b in range(n + 1):
            c = n - (a + b)
            if c >= 0:
                solutions += 1
    return solutions
```

Alice

n	Count
20	441
40	1681
60	3721
80	6561
100	10201

What are some issues with this scheme?

Hard to discern the growth of the algorithm from a table

Image credit - Reducible @ YouTube

What we like about measuring the efficiency of algorithm this way is that it is completely machine independent.

Even in a supercomputer, this counts are not going to change.

One issues with this way is that we don't have a sense of how the algorithm grows as the input increases.

$n = 1000?$

$n = 10000?$

It's hard to make an estimate by looking at the table of counts.

How to measure algorithm efficiency?

Bob's solution vs Alice's solution

Which solution is more efficient?

$$n = 100 \quad \begin{aligned} a &\in [0, 100] \rightarrow 101 \text{ values} \\ b &\in [0, 100] \rightarrow 101 \text{ values} \\ c &= n - (a + b) \rightarrow 1 \text{ value} \end{aligned}$$

```
def alice_solution(n):
    solutions = 0
    for a in range(n + 1):
        for b in range(n + 1):
            c = n - (a + b)
            if c >= 0:
                solutions += 1
    return solutions
```

Is there a way we can simplify this counting scheme?

Let's try to generalize for any n

Image credit - Reducible @ YouTube

Instead of figuring out the count for specific n values, let's generalize to any n.

Looking at Bob's algorithm -

$$\forall n \quad \begin{aligned} a &\in [0, n] \rightarrow (\mathrm{n} + 1) \text{ values} \\ b &\in [0, n] \rightarrow (\mathrm{n} + 1) \text{ values} \\ c &\in [0, n] \rightarrow (\mathrm{n} + 1) \text{ values} \end{aligned}$$

```
def bob_solution(n):
    solutions = 0
    for a in range(n + 1):
        for b in range(n + 1):
            for c in range(n + 1):
                if a + b + c == n:
                    solutions += 1
    return solutions
```

How to measure algorithm efficiency?

Bob's solution vs Alice's solution

Which solution is more efficient?

$$n = 100 \quad \begin{aligned} a &\in [0, 100] \rightarrow 101 \text{ values} \\ b &\in [0, 100] \rightarrow 101 \text{ values} \\ c &= n - (a + b) \rightarrow 1 \text{ value} \end{aligned}$$

```
def alice_solution(n):
    solutions = 0
    for a in range(n + 1):
        for b in range(n + 1):
            c = n - (a + b)
            if c >= 0:
                solutions += 1
    return solutions
```

Is there a way we can simplify this counting scheme?

Let's try to generalize for any n

Image credit - Reducible @ YouTube

Instead of figuring out the count for specific n values, let's generalize to any n.

Looking at Bob's algorithm -

Bob's Count

$(n+1)^3$

$$\forall n \quad \begin{aligned} a &\in [0, n] \rightarrow (\text{n} + 1) \text{ values} \\ b &\in [0, n] \rightarrow (\text{n} + 1) \text{ values} \\ c &\in [0, n] \rightarrow (\text{n} + 1) \text{ values} \end{aligned}$$

```
def bob_solution(n):
    solutions = 0
    for a in range(n + 1):
        for b in range(n + 1):
            for c in range(n + 1):
                if a + b + c == n:
                    solutions += 1
    return solutions
```

How to measure algorithm efficiency?

Bob's solution vs Alice's solution

Which solution is more efficient?

$$n = 100 \quad \begin{aligned} a &\in [0, 100] \rightarrow 101 \text{ values} \\ b &\in [0, 100] \rightarrow 101 \text{ values} \\ c &= n - (a + b) \rightarrow 1 \text{ value} \end{aligned}$$

```
def alice_solution(n):
    solutions = 0
    for a in range(n + 1):
        for b in range(n + 1):
            c = n - (a + b)
            if c >= 0:
                solutions += 1
    return solutions
```

Is there a way we can simplify this counting scheme?

Let's try to generalize for any n

Image credit - Reducible @ YouTube

Alice's algorithm has a similar generalized n count -

Bob's solution vs Alice's solution

Which solution is more efficient?

$$\forall n \quad \begin{aligned} a &\in [0, n] \rightarrow (n + 1) \text{ values} \\ b &\in [0, n] \rightarrow (n + 1) \text{ values} \\ c &= n - (a + b) \rightarrow 1 \text{ value} \end{aligned}$$

Bob's Count

$(n+1)^3$

```
def alice_solution(n):
    solutions = 0
    for a in range(n + 1):
        for b in range(n + 1):
            c = n - (a + b)
            if c >= 0:
                solutions += 1
    return solutions
```

Alice's Count

$(n+1)^2$

Is there a way we can simplify this counting scheme?

Let's try to generalize for any n

How to measure algorithm efficiency?

Bob's solution vs Alice's solution

Which solution is more efficient?

$$\forall n \quad \begin{aligned} a &\in [0, n] \rightarrow (\mathrm{n} + 1) \text{ values} \\ b &\in [0, n] \rightarrow (\mathrm{n} + 1) \text{ values} \\ c &= n - (a + b) \rightarrow 1 \text{ value} \end{aligned}$$

Bob's Count

$n^3 + 3n^2 + 3n + 1$

```
def alice_solution(n):
    solutions = 0
    for a in range(n + 1):
        for b in range(n + 1):
            c = n - (a + b)
            if c >= 0:
                solutions += 1
    return solutions
```

Alice's Count

$n^2 + 2n + 1$

Let's now try to simplify more!

What happens when n becomes very large?

Image credit - Reducible @ YouTube

Now we have a general expression for how each algorithm operates for any n value.

If we are really concerned about how the program performs as the input grows, what we should really focus on is the worst-case performance when n is a very large number.

When n is very large, the *lower order terms* becomes so irrelevant that we can ignore them.

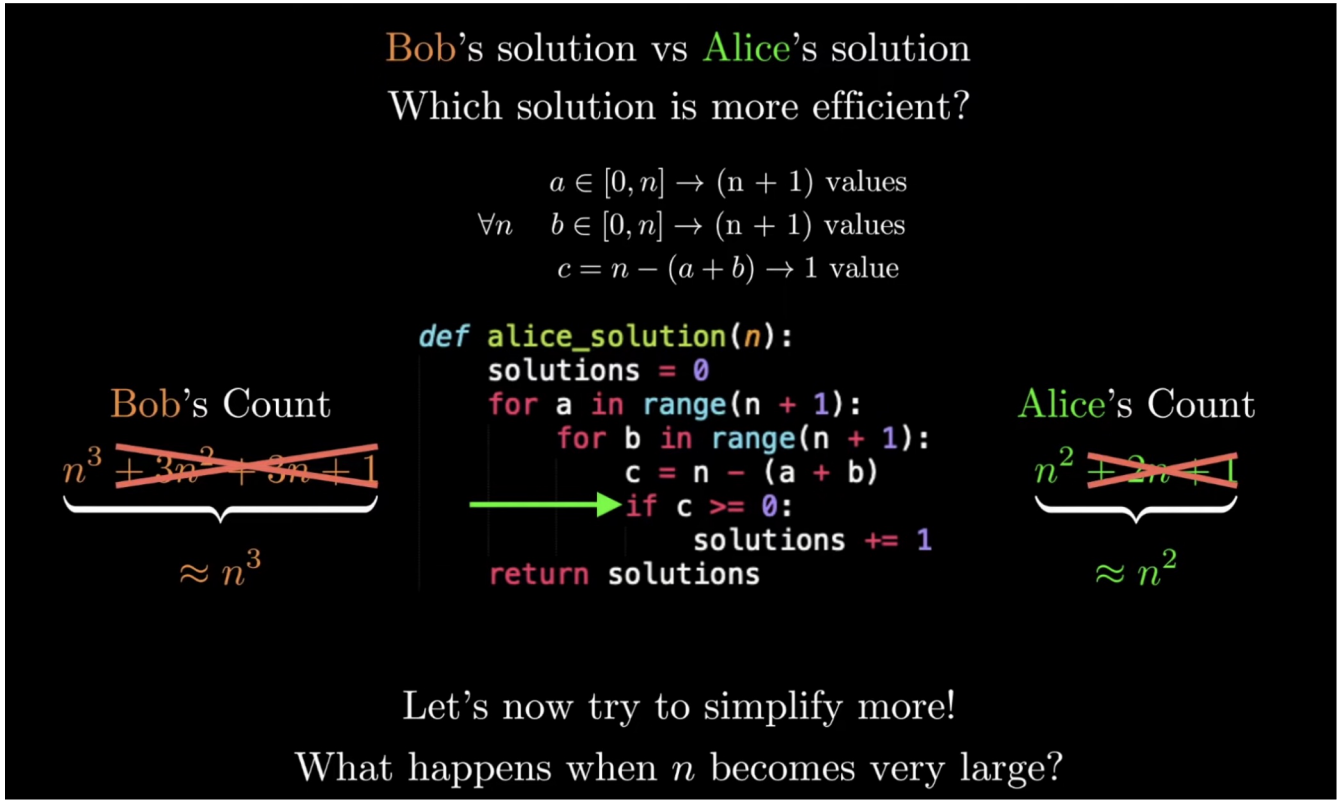
Bob's Count

$n^3 + 3n^2 + 3n + 1$

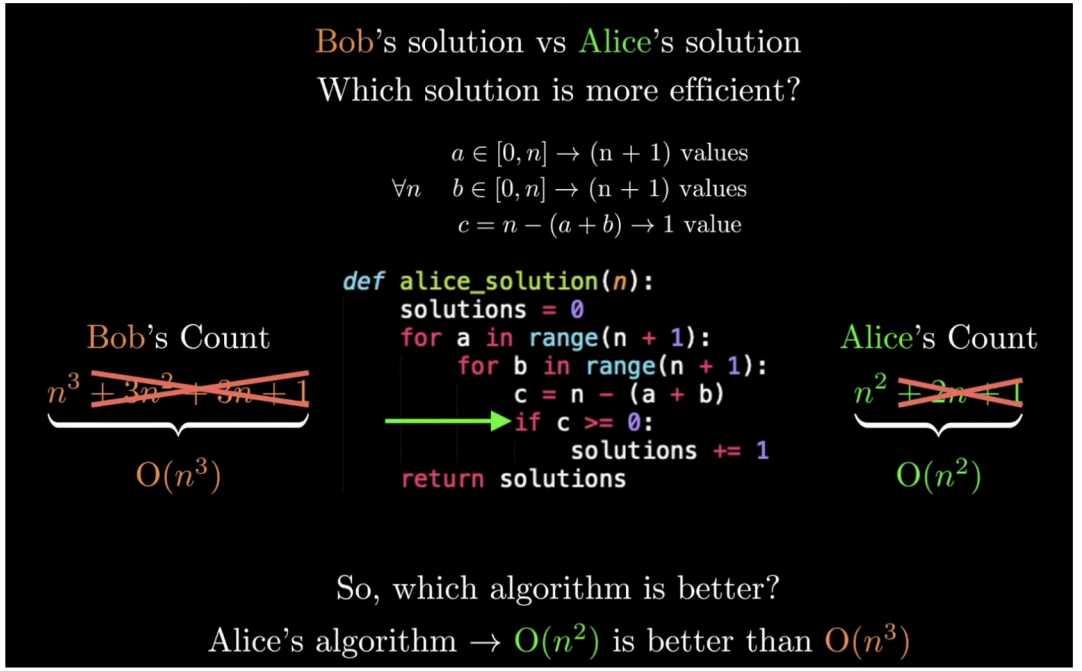
Alice's Count

$n^2 + 2n + 1$

How to measure algorithm efficiency?



The final single term for each of this algorithm encapsulates the **running time of an algorithm**.

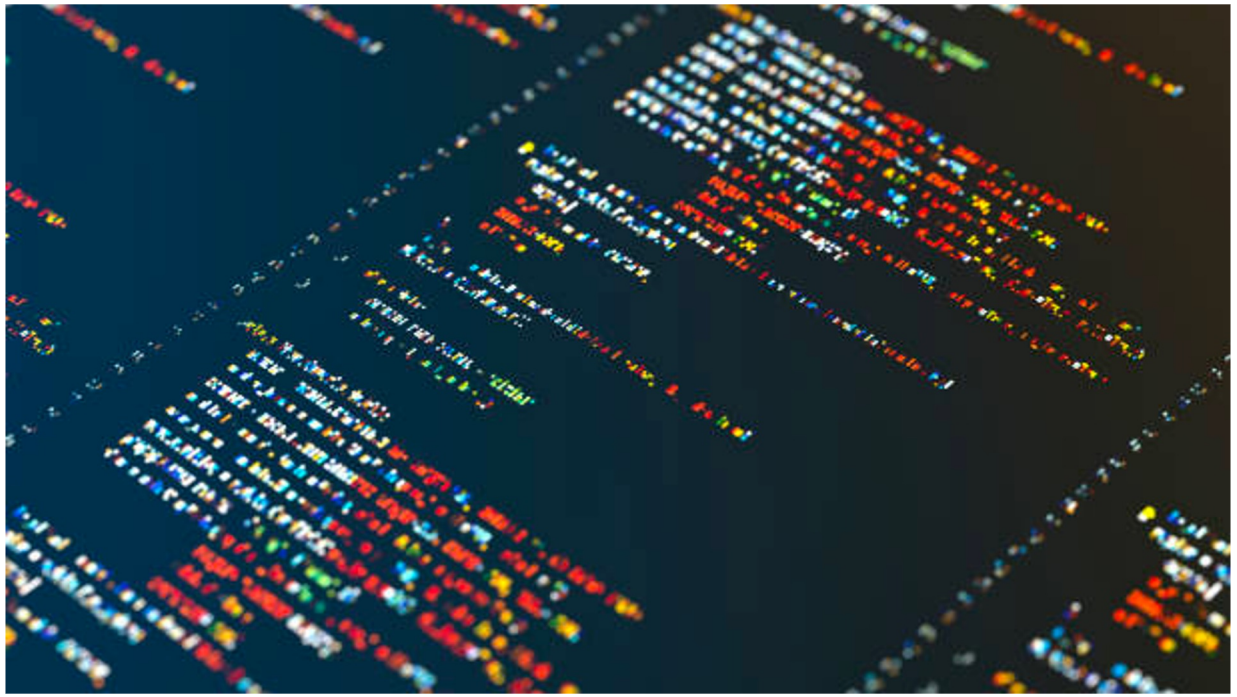


Now, we can definitively say that Alice's program is better than Bob's because it grows at a much slower pace compared to Bob's.

Image credit - Reducible @ YouTube

Motivations for Complexity Analysis

- There are often many different algorithms which can be used to solve the same problem. Thus, it makes sense to develop techniques that allow us to:
  - compare different algorithms with respect to their “efficiency”
  - choose the most efficient algorithm for the problem
- The efficiency of any algorithmic solution to a problem is a measure of the:
  - Time efficiency: the time it takes to execute.
  - Space efficiency: the space (primary or secondary memory) it uses.
- We will focus on an algorithm’s efficiency with respect to time.



Time and Space Complexity

Time complexity is defined as the amount of time taken by an algorithm to run, as a function of the length of the input.

It measures the time taken to execute each statement of code in an algorithm. It is not going to examine the total execution time of an algorithm. Rather, it is going to give information about the variation (increase or decrease) in execution time when the number of operations (increase or decrease) in an algorithm.

Yes, as the definition says,
the amount of time taken is a function of the length of input only.

When an algorithm is run on a computer, it necessitates a certain amount of memory space.
Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.
One major underlying factor affecting your program's performance and efficiency is the hardware, OS, and CPU you use.

But you don't consider this when you analyze an algorithm's performance. Instead, the time and space complexity as **a function of the input's size** are what matters.

Image credit - https://www.qapitol.com/

Asymptotic Notations

Asymptotic Notations are programming languages that allow you to analyze an algorithm's running time by identifying its behavior as its input size grows. This is also referred to as an algorithm's growth rate. When the input size increases, does the algorithm become incredibly slow? Is it able to maintain its fast run time as the input size grows?

You can answer these questions thanks to **Asymptotic Notation**.

You can't compare two algorithms head to head. It is heavily influenced by the tools and hardware you use for comparisons, such as the operating system, CPU model, processor generation, and so on. Even if you calculate time and space complexity for two algorithms running on the same system, the subtle changes in the system environment may affect their time and space complexity.

As a result, you compare space and time complexity using asymptotic analysis. It compares two algorithms based on changes in their performance as the input size is increased or decreased.

Asymptotic notations are classified into three types:

- Big-Oh (O) notation
- Big Omega (Ω) notation
- Big Theta (Θ) notation

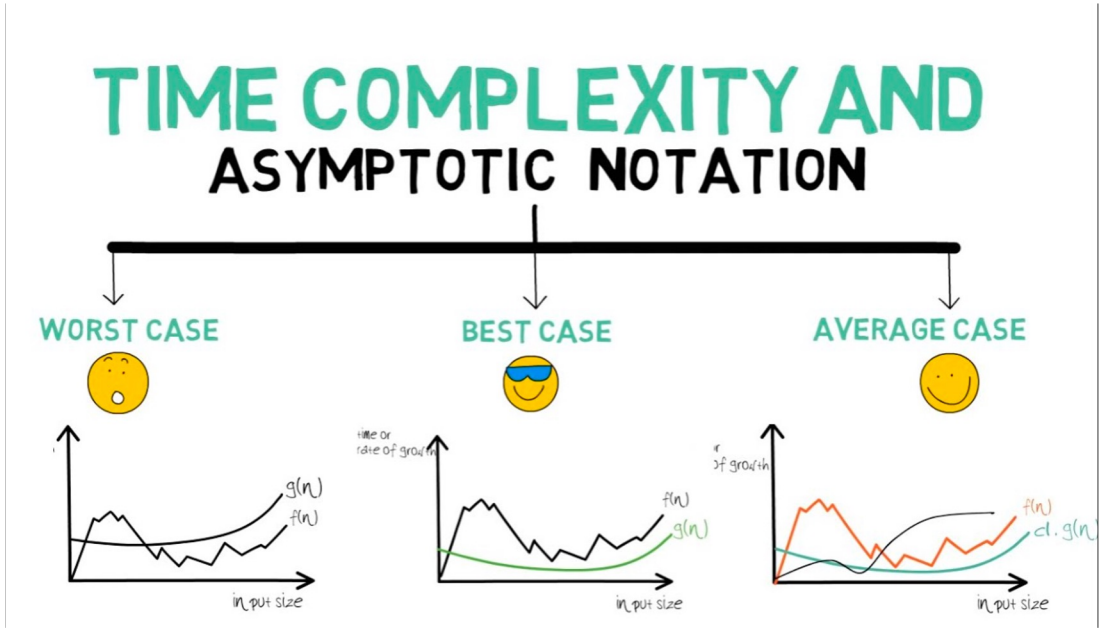


Image credit - https://www.boardinfinity.com/

Asymptotic Notations

Big-O Notation

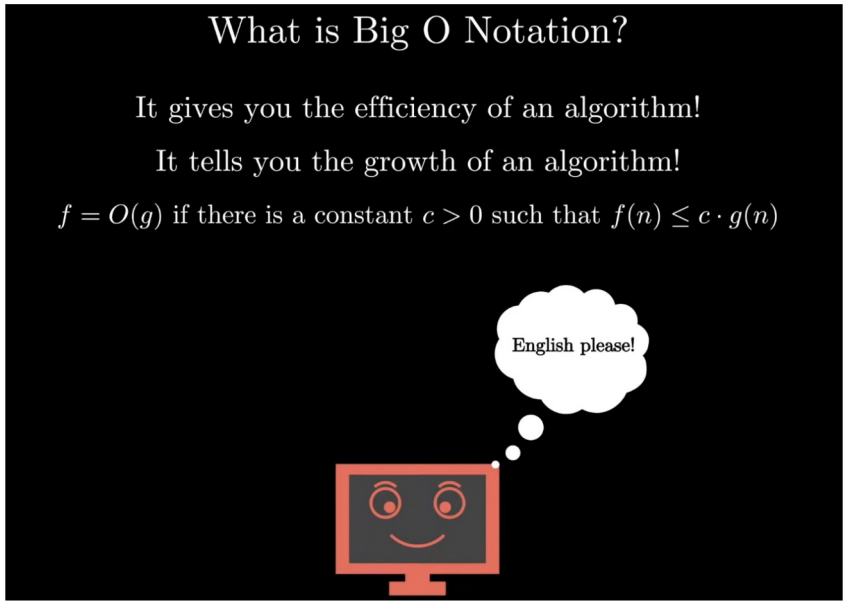
The Big-O notation describes the worst-case running time of a program.

We compute the Big-O of an algorithm by counting how many iterations an algorithm will take in the worst-case scenario with an input of N.

We typically consult the Big-O because we must always plan for the worst case. For example, O(log n) describes the Big-O of a binary search algorithm.

The common algorithmic runtimes from fastest to slowest are:

- **constant**: Θ(1)
- **logarithmic**: Θ(log N)
- **linear**: Θ(N)
- **polynomial**: Θ(N^2)
- **exponential**: Θ(2^N)
- **factorial**: Θ(N!)



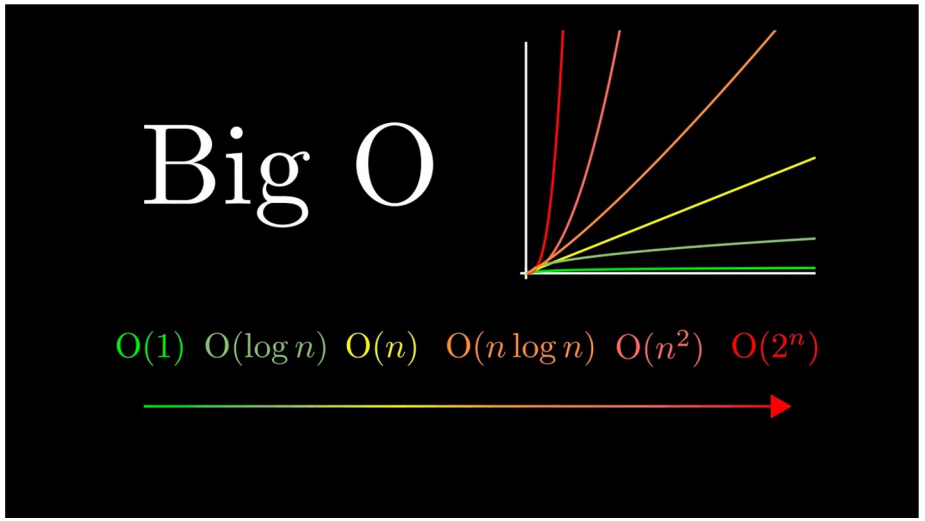


Image credit - Reducible @ YouTube

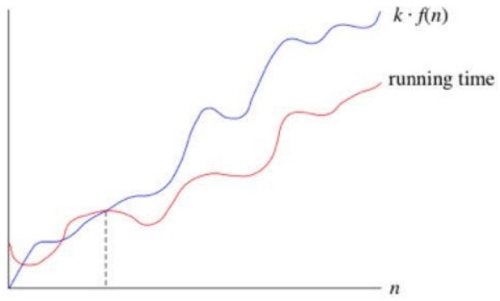
Asymptotic Notations
Big-Ω and Big-Θ Notation

Big-Ω (Omega) describes the best running time of a program.

We compute the big-Ω by counting how many iterations an algorithm will take in the best-case scenario based on an input of N.

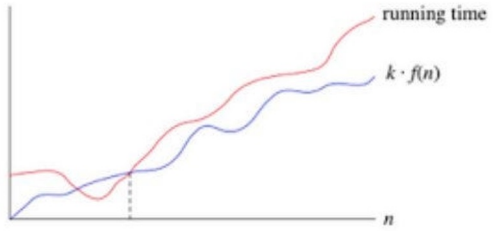
For example, a Bubble Sort algorithm has a running time of Ω(N) because in the best case scenario the list is already sorted, and the bubble sort will terminate after the first iteration.

Big – Theta(Θ) notation specifies asymptotic bounds (both upper and lower) for a function f(n) and provides the average time complexity of an algorithm.



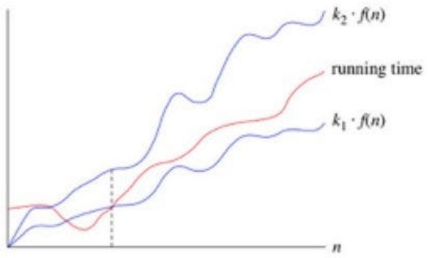
Big O - function that describes 'upper bound'

aka the worst case senario



Big Omega - function that describes 'lower bound'

aka the best case senario



Big Theta- function that describes 'tight bound'

aka when Big O and Big Omega are the same type of function

The lower bound has a smaller constant k_1 and the upper bound has a larger constant k_2

Image credit - https://blog.devgenius.io/

Asympto
Big-Ω and B

Big-Ω (Omega)
program.

We compute the
algorithm will ta
input of N.

For example, a
of Ω(N) because
already sorted,
first iteration.

Big – Theta(Θ)
(both upper and
average time co

Best, Worst, and Average

- We are usually interested in the worst case complexity: what are the most operations that might be performed for a given problem size. We will not discuss the other cases -- best and average case.

- Best case depends on the input
- Average case is difficult to compute
- So we usually focus on worst case analysis
  - Easier to compute
  - Usually close to the actual running time
  - Crucial to real-time systems (e.g. air-traffic control)

es 'upper bound'

escribes 'lower bound'

cribes 'tight bound'

ega are the same

ller constant k_1
larger constant k_2

Image credit - https://blog.devgenius.io/

Big-O Notation

Big O notation is simplified analysis of an algorithm's efficiency.

1. Big-O gives an algorithm's complexity (amount of resources required to run it) in terms of input size, N
2. Machine-independent - we don't care about the stats of the machine
3. We examine basic computer steps
4. Big-O can analyze both time & space complexity

The letter O was chosen by Bachmann to stand for Ordnung, meaning the order of approximation.

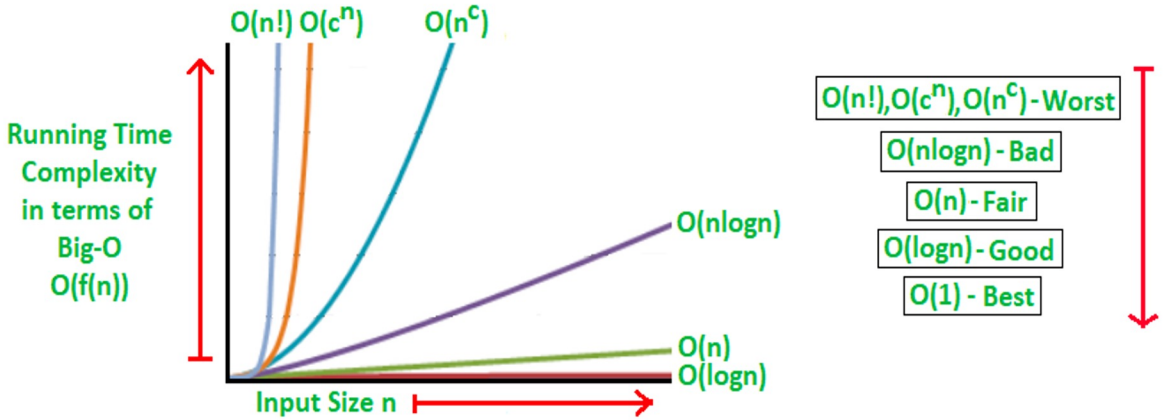


Image credit - https://www.geeksforgeeks.org/

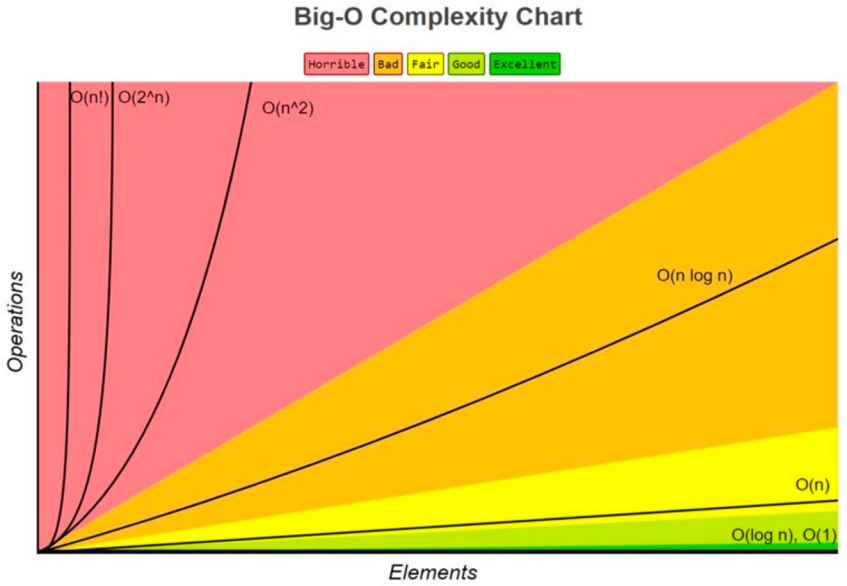


Image credit - https://www.freecodecamp.org/

General rules -

1. Ignore constants
   5n → O(n)
   Because as n gets large, the 5 no longer matters

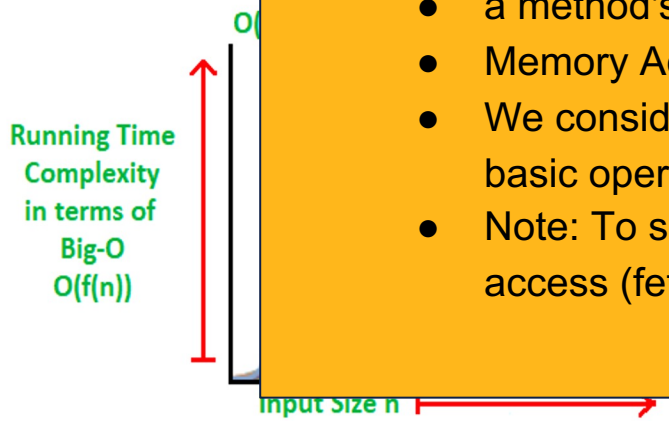
2. As n grows, certain terms "dominate" others
   O(1) < O(log n) < O(n) < O(nlogn) < O (n^2) < O(2^n) < O(n!)
   i.e., ignore low-order terms when they are dominated by high-order ones

Big-O No

Big O notation is

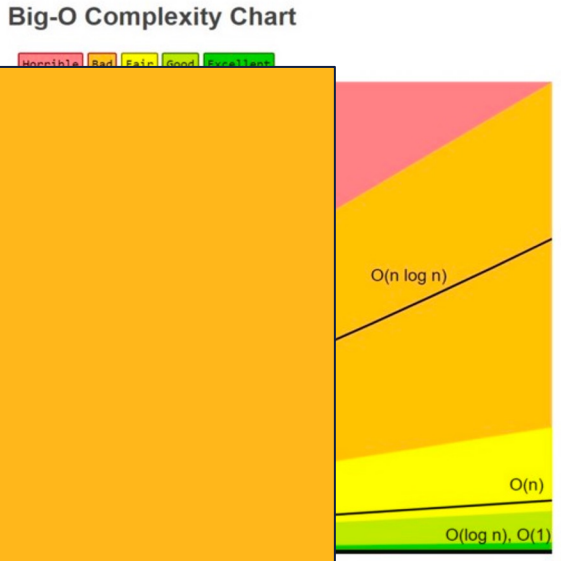
1. Big-O gives
   terms of in
2. Machine-in
3. We examin
4. Big-O can

The letter O was
approximation.



Basic Operations

- Arithmetic operations: *, /, %, +, -
- Assignment statements of simple data types.
- Reading of primitive types
- Writing of a primitive types
- Simple conditional tests: if (x < 12) ...
- method call (Note: the execution time of the method itself may depend on the value of parameter and it may not be constant)
- a method's return statement
- Memory Access
- We consider an operation such as ++ , += , and *= as consisting of two basic operations.
- Note: To simplify complexity analysis we will not consider memory access (fetch or store) operations.



amp.org/

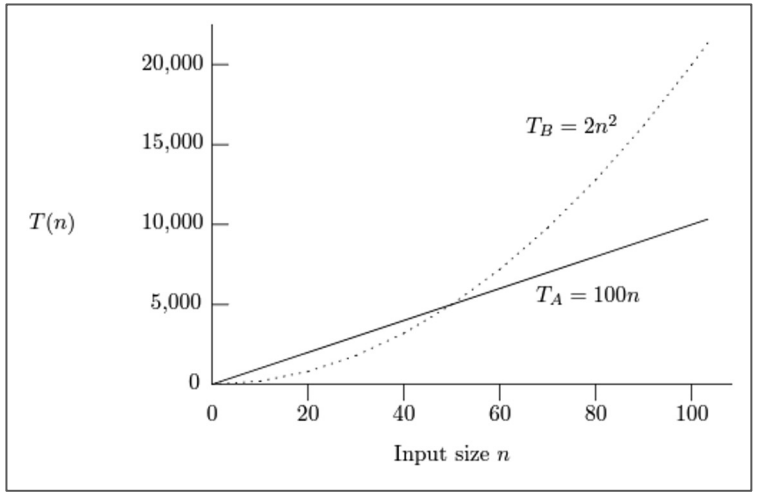
ger matters

others
) (n^2) < O(2^n) < O(n!)
i.e., ignore low-order terms when they are dominated by high-order ones

Image credit - https://www.geeksforgeeks.org/

Comparing Different Running Times

Suppose that for some problem we have the choice of using a linear-time program A whose running time is TA(n) = 100n and a quadratic-time program B whose running time is TB(n) = 2n^2. Let us suppose that both these running times are the number of milliseconds taken on a particular computer on an input of size n.



We see that for inputs of size less than 50, program B is faster than program A. When the input becomes larger than 50, program A becomes faster, and from that point on, the larger the input, the bigger the advantage A has over B. For inputs of size 100, A is twice as fast as B, and for inputs of size 1000, A is 20 times as fast.

The general principles of computing Big-O are:

- **Constant factors don't matter**

Take two algorithms: Bucket Sort, and Bubble Sort. From analysis we can see the Bucket Sort has O(n) wheras Bubble Sort has O(n*n). Let us say, for instance that in practice our Bucket Sorting algorithm has a time constant multiple of 200 wheras our Bubble Sort is just a constant of 2.

So given an input of 30 items our Bucket Sort could be expected to run at something like:

200 * 30 = 6000

and our Bubble Sort at:
30 * 30 * 2 = 1200

Well wait a second Bubble Sort runs quicker here, but its complexity is quadratic as compared to Bucket Sorts linear time with a constant. Ok, now lets see what happens with an input size of 1,000,000.

Bucket Sort:
200 * 1,000,000 = 200,000,00

Bubble Sort:
1,000,000 * 1,000,000 * 2 = 2,000,000,000,000

So as you can see, as the input size gets large enough, the constants aren't really going to mean anything.

- **Low order terms don't matter**

$$T(n) = 3n^5 + 10n^4 - 4n^3 + n + 1$$

The highest-order term is n^5, and we claim that $T(n)$ is $O(n^5)$.

For example, let $T(n) = 2^n + n^3$. It is known that every polynomial, such as n^3, grows more slowly than every exponential, such as 2^n. Since $n^3/2^n$ approaches 0 as n increases, we can throw away the lower-order term and conclude that $T(n)$ is $O(2^n)$.

Big-O Notation: O(1)

Constant Time - Simple Statements

The algorithm performs a constant number of operations regardless of the size of the input.

O(1) "big oh of one"

```
x = 5 + (15 * 20);
```

independent of input size, N

Examples:

- Access a single number in an array by index
- Add 2 numbers together.

We drop constants so it's still big-oh of one

```
x = 5 + (15 * 20);
y = 15 - 2;
print x + y;
```

total time = O(1) + O(1) + O(1) = O(1)

3 * O(1)

Image credit - Michael Sambol

Constant Time Complexity O(1)

Constant Time requires the same amount of time regardless of the input size. In other words, as the input grows, execution time stays the same.

Examples: Accessing an element in an array; Hash Table look-ups; Array methods: .push and .pop

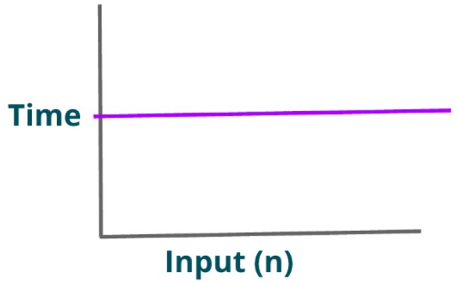


Image credit - Jacob Kagon @Medium

```
int   sumOfList( int A[ ], int n)
{
    int   sum = 0, i;
    for(i = 0; i < n; i++)
        sum = sum + A[i];
    return   sum;
}
```

Cost (Time require for line (Units))	Repeatation (No. of Times Executed)	Total (Total Time required in worst case)
1	1	1
1 + 1 + 1	1 + (n+1) + n	2n + 2
2	n	2n
1	1	1
		4n + 4 (Total Time required)

Image credit - http://www.btechsmartclass.com/

Big-O Notation: O(1)

Constant Time - Simple Statements

Certain simple operations on data can be done in *O*(1) time, that is, in time that is independent of the size of the input. These primitive operations in consist of:

1. Arithmetic operations (e.g. + or %).
2. Logical operations (e.g., &&).
3. Comparison operations (e.g., <=).
4. Structure accessing operations (e.g. array-indexing like A[i], or pointer following with the-> operator).
5. Simple assignment such as copying a value into a variable.
6. Calls to library functions (e.g., scanf, printf).

Simple statements that can be executed in *O(1)* time, that is, in some constant amount of time independent of input:

1. Assignment statements that do not involve function calls in their expressions.
2. Read statements.
3. Write statements that do not require function calls to evaluate arguments.
4. The jump statements break, continue, goto, and return expression, where expression does not contain a function call.

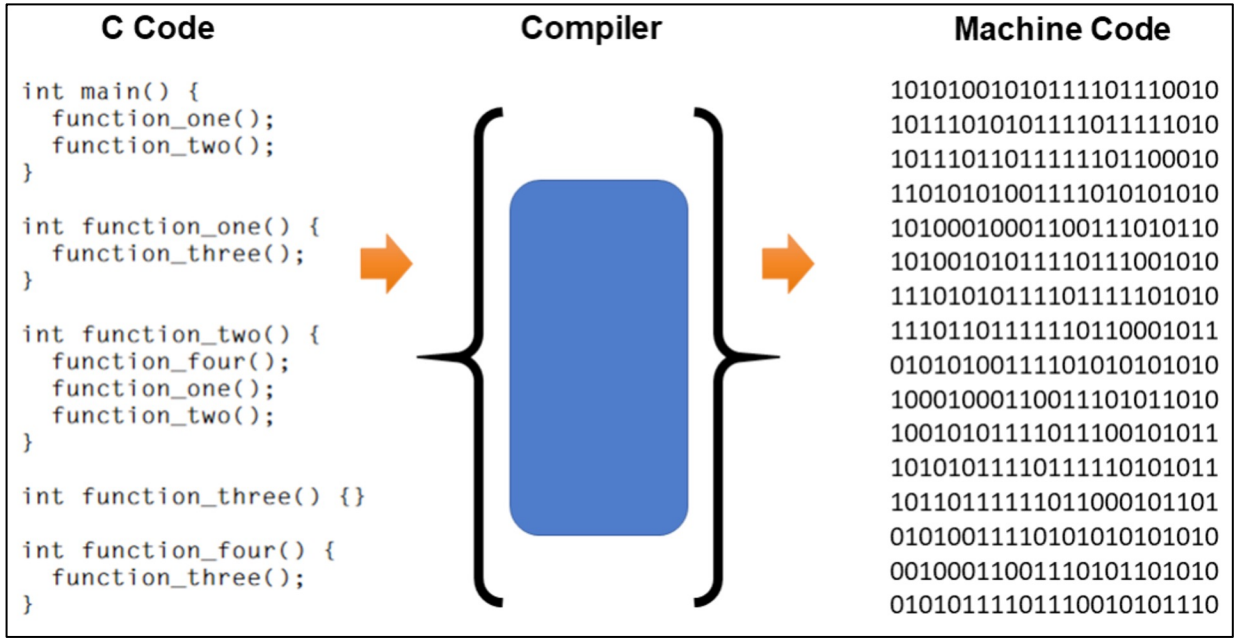


Image credit - https://www.astateofdata.com/

Big-O Notation: O(1)

Constant Time - Selection Statements

An if-else selection statement has the form

```
if (<condition>)
    <if-part>
else
    <else-part>
```

where

1. The condition is an expression to be evaluated,
2. The if-part is a statement that is executed only if the condition is true (the value of the expression is not zero), and
3. The else-part is a statement that is executed if the condition is false (evaluates to 0). The else followed by the <else-part> is optional.

A condition, no matter how complex, requires the computer to perform only a constant number of primitive operations, as long as there are no function calls within the condition.

Thus, the evaluation of the condition will take **O(1) time**.

```
for(int i = 1; i <= n; i++) {
   if(i == 1)
      block1;
   else
     block2;
}
```

- **block1** is executed only once, **block2** is executed **n - 1** times
  - Complexity is **O(T(block1))** if T(block1) > (n – 1)*T(block2)
  - Otherwise block2 is dominant:
    Complexity is **O(n*T(block2))**

Image credit - https://www.astateofdata.com/

```
(1)        if (A[1][1] == 0)
(2)            for (i = 0; i < n; i++)
(3)                for (j = 0; j < n; j++)
(4)                    A[i][j] = 0;
           else
(5)            for (i = 0; i < n; i++)
(6)                A[i][i] = 1;
```

The running time of lines (2) through (4) is *O(n^2)*, while the running time of lines (5) and (6) is *O(n)*. Thus, f(n) is n^2 here, and g(n) is n. Since n is O(n^2), we can neglect the time of the else-part and take O(n^2) as a bound on the running time of the entire fragment. That is to say, we have no idea if or when the condition of line (1) will be true, but the only safe upper bound results from assuming the worst: that the condition is true and the if-part is executed.

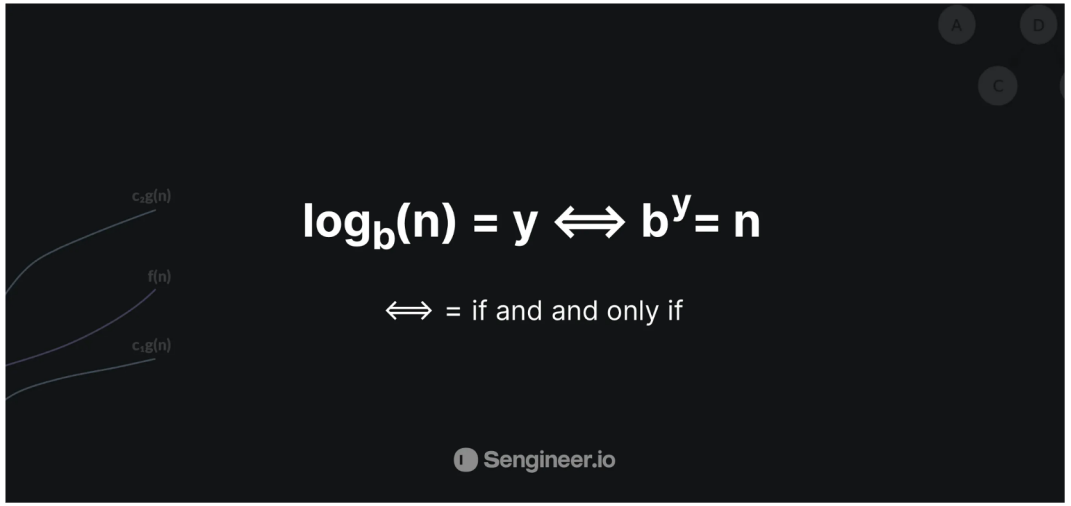
Big-O Notation: O(log n)

Logarithmic Time

How many times must a base of 20 be multiplied by itself to get 8,000? The answer is 3 (8000 = 20 × 20 × 20). So the logarithm base 20 of 8,000 is 3. It's represented using a subscript (small number) to the lower right of the base number. So the statement would be log20(8,000) = 3.

- log20(400) is like asking "How many 20s do we multiply to get 400? which is 2(20 * 20). So log20(400) = 2

Logarithms are defined using the following equation:



Because logarithms mostly occur in computer science by repeatedly dividing some **data input** (e.g. list, array, etc...) in half, which often occurs with algorithms like:

- divide-and-conquer algorithms like binary search, quicksort, Closest Pair of Points, Merge Sort, etc...

In those cases, the number of times you can divide a **data input** (e.g. list, array, etc...) of length **n** in half before you get down to single-element arrays is $\log_2 n$.

$$\log_b(a) = c \iff b^c = a$$

Image credit - Humam Abo Alraja @ Medium

Big-O Notation: O(log n)

Logarithmic Time

As the size of input `n` increases, the algorithm's running time grows by `log(n)`.
This rate of growth is relatively slow, so O(log n) algorithms are usually very fast.
As you can see in the table below, when `n` is 1 billion, `log(n)` is only 30.

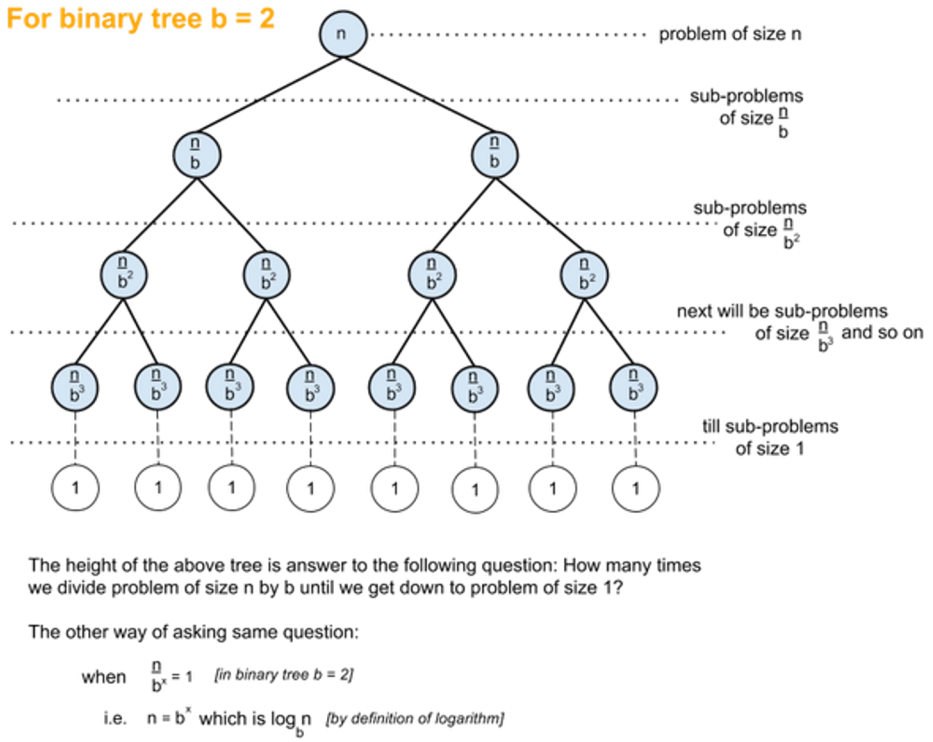
Examples:

Binary search on a sorted list. The size that needs to be searched is split in half each time, so the remaining list goes from size n to n/2 to n/4... until either the element is found or there's just one element left. If there are a billion elements in the list, it will take a maximum of 30 checks to find the element or determine that it is not on the list.

Whenever an algorithm only goes through part of its input, see if it splits the input by at least half on average each time, which would give it a logarithmic running time.

n	log(n)
2	1
1000	10
1M	20
1B	30

Computer scientists generally think of "$\log n$" as meaning $\log_2 n$, rather than $\log_e n$ or $\log_{10} n$. Notice that $\log_2 n$ is the number of times we have to divide n by 2 to get down to 1, or alternatively, the number of 2's we must multiply together to reach n. You may easily check that $n = 2^k$ is the same as saying $\log_2 n = k$; just take logarithms to the base 2 of both sides.



The height of the above tree is answer to the following question: How many times we divide problem of size n by b until we get down to problem of size 1?

The other way of asking same question:

when $\frac{n}{b^x} = 1$ *[in binary tree b = 2]*

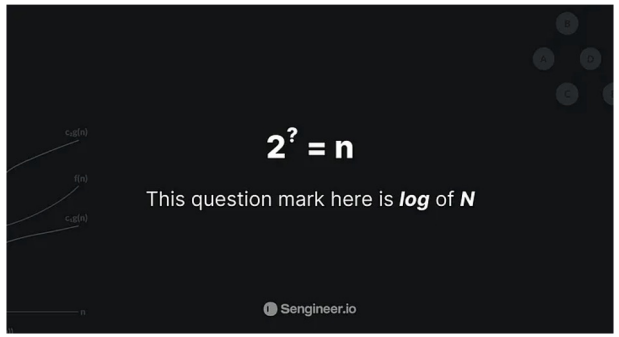
i.e. $n = b^x$ which is $\log_b n$ *[by definition of logarithm]*

Image credit - Sumod Mathilakath @ Quora

Big-O Notation: O(log n)
Logarithmic Time

So logarithm base 2 of n would be equal to y if and only if the number 2 to the power of y were equal to n.

$$\log_2(n) = y \iff 2^y = n$$



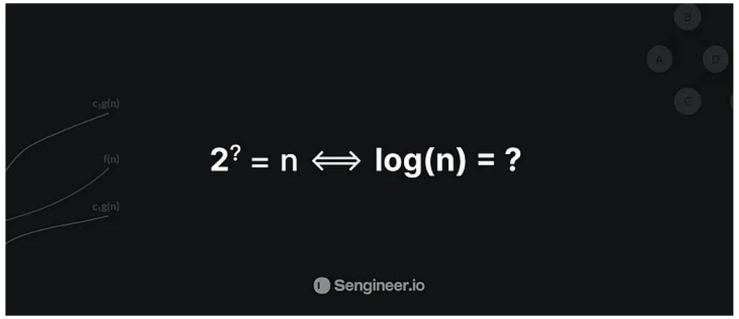


Image credit - Humam Abo Alraja @ Medium

log(n) increases only by a tiny amount as **N** increases. When **N** doubles, *log(n)* only increases by 1. And so this is why, if we tie this back to complexity analysis when we have an algorithm with time complexity of *log(n)*, that is incredibly good because that means as the input increases/doubles, the number of elementary operations that we're performing in the algorithm only increases by one.

> ***Logarithmic time complexity*** **log(n)**: *Represented in Big O notation as* ***O(log n)****, when an algorithm has O(log n) running time, it means that as the input size grows, the number of operations grows very slowly.* ***Example:*** *binary search.*

So I think now it's clear for you that a *log(n)* complexity is extremely better than a **linear complexity O(n)**. Even though **O(n)**, linear time is already pretty good for an algorithm. *log(n)* time is gonna be way better as the size of your input increases.

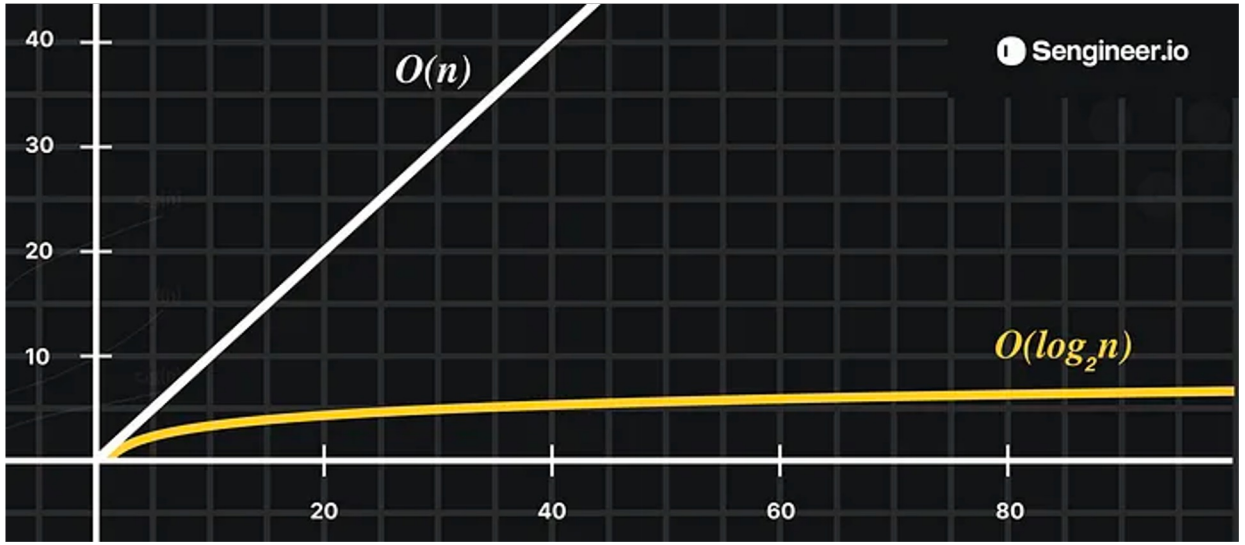
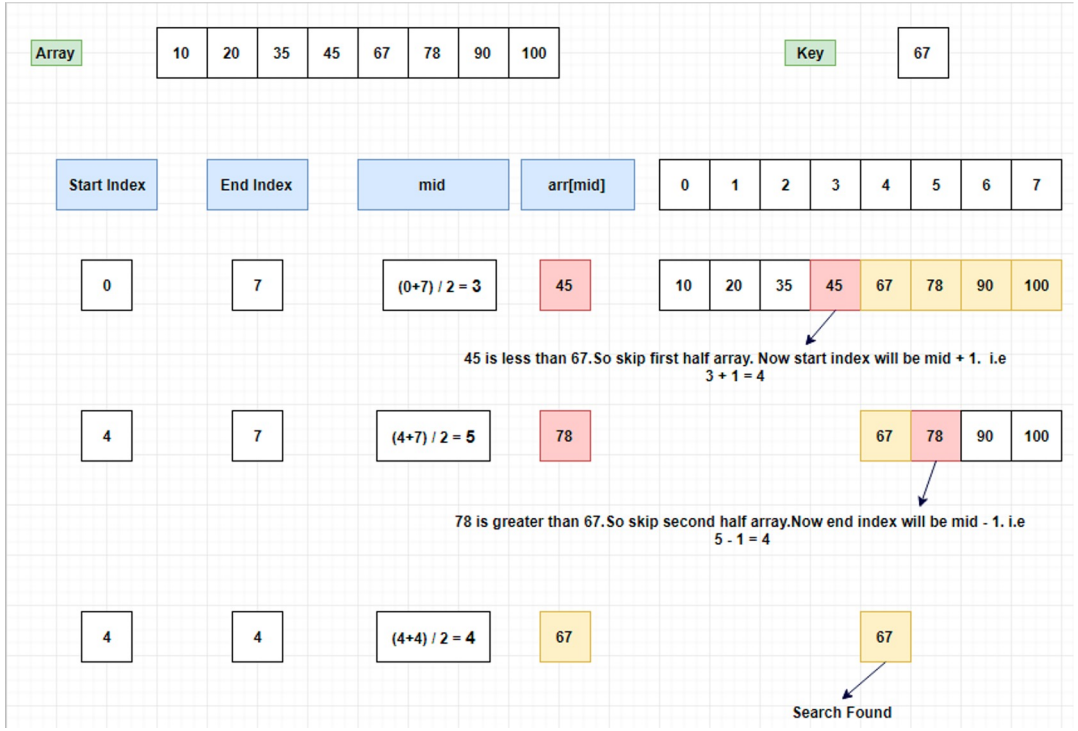
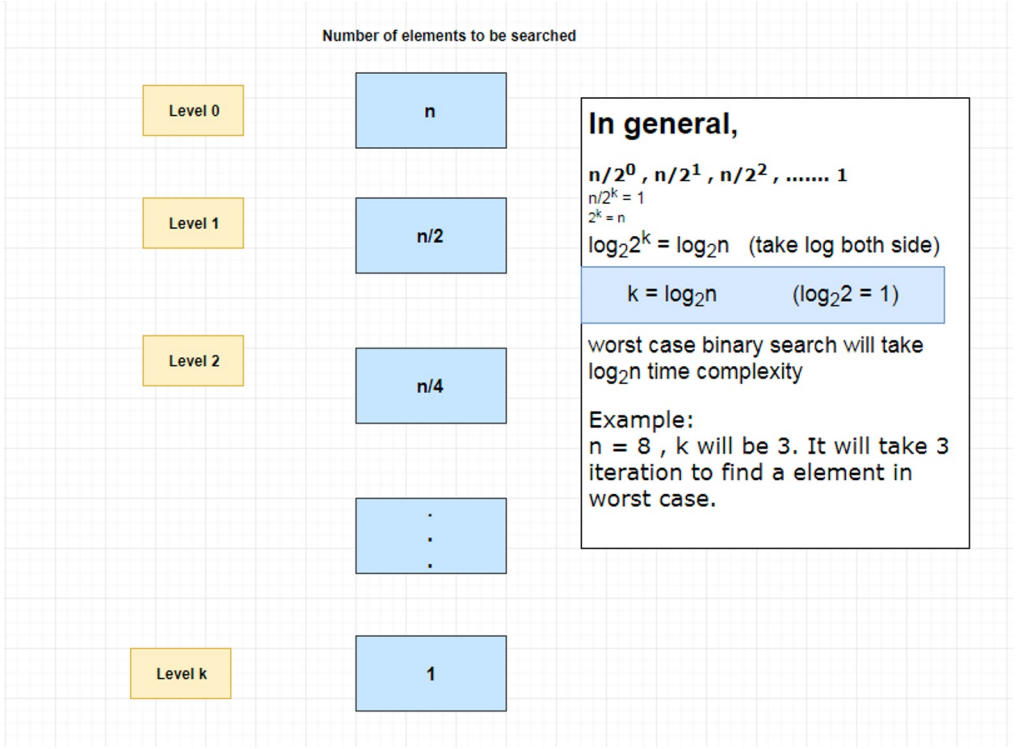


Image credit - https://www.log2base2.com/

Big-O Notation: O(log n)

Logarithmic Time (Binary Search)





O(log N) basically means time goes up linearly while the n goes up exponentially. So if it takes 1 second to compute 10 elements, it will take 2 seconds to compute 100 elements, 3 seconds to compute 1000 elements, and so on.

It is **O(log n)** when we do divide and conquer type of algorithms e.g binary search. Another example is quicksort where each time we divide the array into two parts and each time it takes O(N) time to find a pivot element. Hence it N O(log N)

Big-O Notation: O(log n)

Logarithmic Time

```
for (int i = k; i < n; i = i * m){
        statement1;
        statement2;
 }
```

Step 1: i = k = k * m^0
Step 2: i = k * m = k * m^1
Step 3: i = k * m * m = k * m^2
Step 4: i = k * m * m * m = k * m^3

Let's assume, i = n (when loop terminates) at step s

Step s i = n = k * m^(s-1) => (n/k) = m^(s-1)

Apply log base m on both sides

log (n/k) = (s-1) => s = log (n/k) + 1

Therefore, number of steps ≈ **O (log n/k)** (base m)

```
//Example of Logarithmic growth
while(i <= n){
    //statement
    i = i*2;
}
```

Step 1: i = 1 * 2 = 2 = 2^1
Step 2: i = 2 * 2 = 4 = 2^2
Step 3: i = 4 * 2 = 8 = 2^3
Step 4: i = 8 * 2 = 16 = 2^4

Let's assume, i = n+1 (when loop terminates) at step k

Step k i = n+1 = 2^k

Apply log base 2 on both sides

log (n+1) = log (2^k) => k = log (n+1)

Therefore, number of steps ≈ **O (log n)** (base 2)

Big-O Notation: O(n)

Linear Time - Simple For Loops

The running time of an algorithm grows in proportion to the size of input.Following for loop prints the number from 0 to n:

$$N * O(1) = O(N)$$

```
for x in range (0, n):
    print x; // O(1)
```

```
y = 5 + (15 * 20);          O(1)
for x in range (0, n):      } O(N)
    print x;
```

$$\text{total time} = O(1) + O(N) = O(N)$$

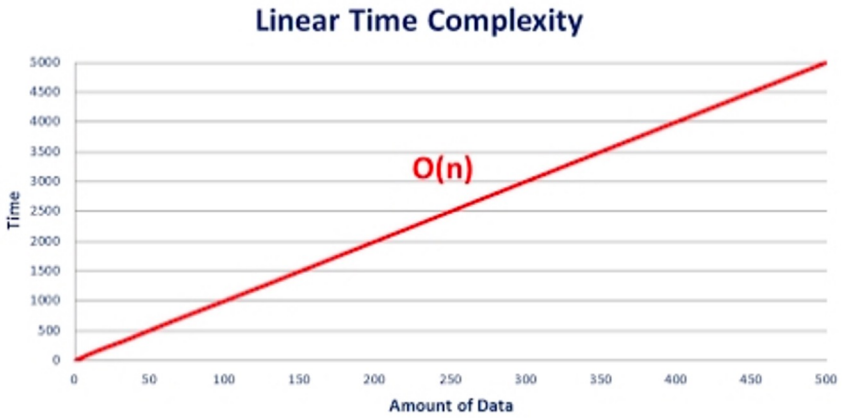
Image credit - Michael Sambol

Data	Time
1	10
2	20
4	40
8	80
16	160
32	320
64	640
128	1280
256	2560
512	5120

Examples:

- Search through an unsorted array for a number
- Sum all the numbers in an array
- Access a single number in a LinkedList by index

During the summation, we drop low-order term. When n gets large, the time it takes to compute the value of y is meaningless.
In this example, the for loop dominates the run time.



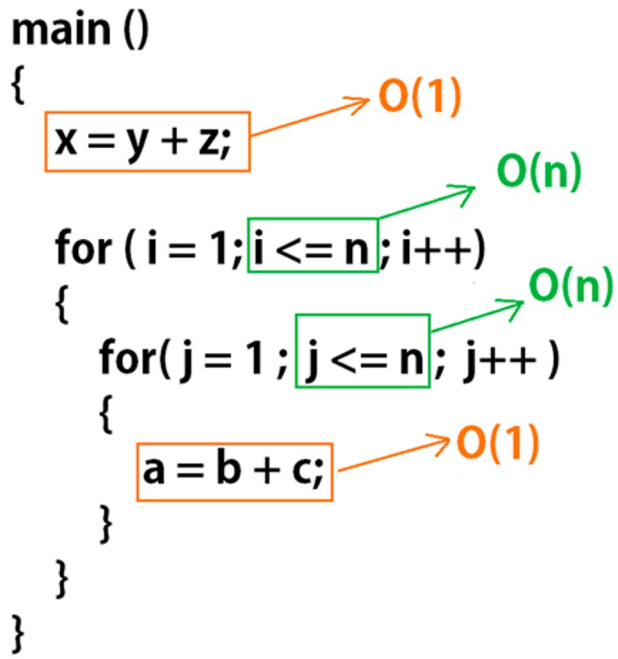


Image credit - https://scienceaid.net/

Big-O Notation: O(n log n)
Linearithmic Time

- Linearithmic runtime refers to an algorithm whose time complexity grows in a near-linear fashion with the size of the input data.
- Algorithms with linearithmic runtime are faster than those with quadratic (O(n^2)) or cubic (O(n^3)) runtimes but slower than linear (O(n)) algorithms.
- Linearithmic algorithms are commonly used in sorting and searching tasks, divide-and-conquer algorithms, and various tree-based data structures like balanced binary search trees.

order of growth description	function	1	10	100	1000
constant	1	1	1	1	1
logarithmic	$\log N$	0	1	2	3
linear	N	1	10	100	1000
linearithmic	$N \log N$	0	10	200	3000
quadratic	N^2	1	100	10,000	1,000,000
cubic	N^3	1	10^3	100^3	1000^3
exponential	2^N	2	2^10	2^100	2^1000
Commonly encountered order-of-growth functions					

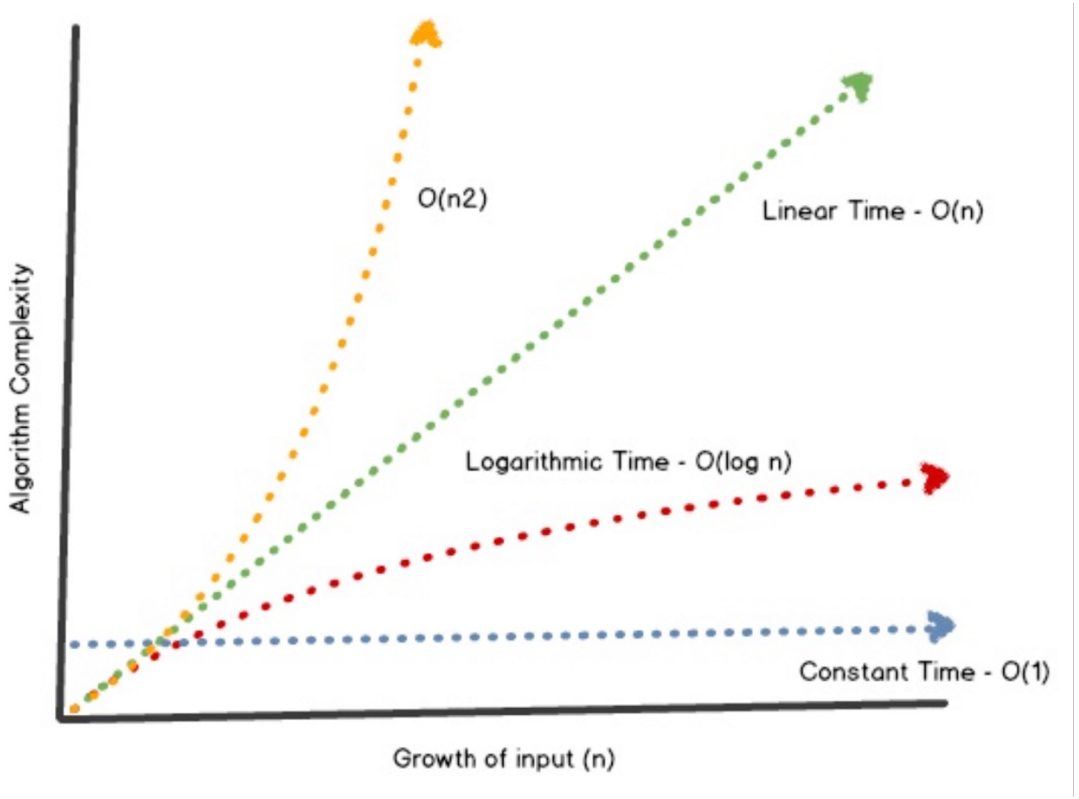


Image credit - https://towardsdatascience.com/

Examples:

Sort an an array with QuickSort or Merge Sort. When recursion is involved, calculating the running time can be complicated. You can often work out a sample case to estimate what the running time will be.

Big-O Notation: O(n^2)

Quadratic Time

The algorithm's running time grows in proportion to the square of the input size, and is common when using nested-loops. Here is a nested loop, which will be executed n x n:

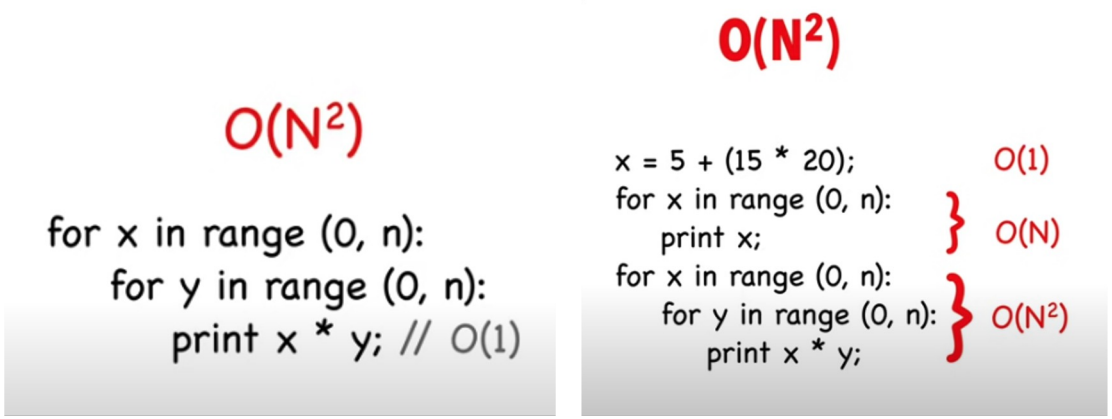
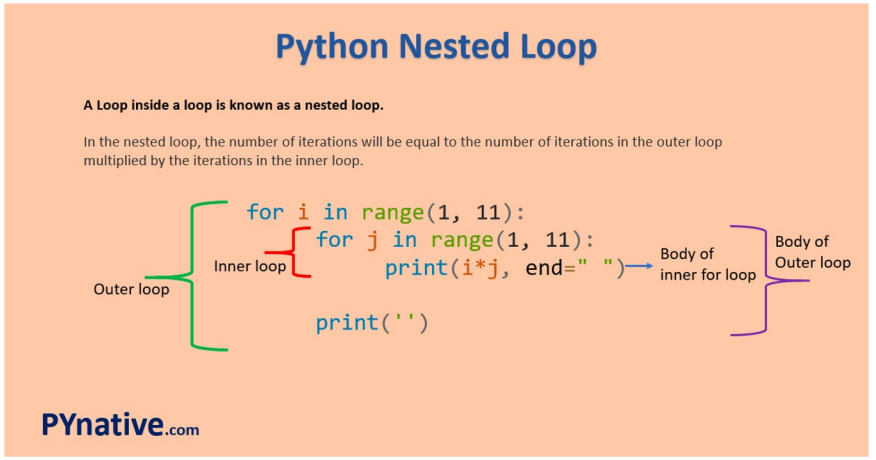


Image credit - Michael Sambol

Some algorithms use nested loops where the outer loop goes through an input *n* while the inner loop goes through a different input *m*. The time complexity in such cases is O(nm). For example, while the above multiplication table compared a list of data to itself, in other cases you may want to compare all of one list *n* with all of another list *m*.

Anatomy of Nested Loops



A Quadratic running time is common when you have nested loops. To calculate the running time, find the maximum number of nested loops that go through a significant portion of the input.

- 1 loop (not nested) = O(n)
- 2 loops = O(n^2)
- 3 loops = O(n^3)

Examples:

- Printing the multiplication table for a list of numbers
- Insertion Sort

Sometimes you'll encounter O(n^3) algorithms or even higher exponents. These usually have considerably worse running times than O(n^2) algorithms even if they don't have a different name!

n	n^2
1	1
10	100
100	10,000
1000	1M

Big-O Notation: O(n^2)

Quadratic Time

```
1   public static void printPairsTwice(int[] numbers) {
2
3           System.out.println("Pairs: ");
4           int n = numbers.length;
5
6           for(int i = 0; i < n; i++) {
7                   for(int j = 0; j < n; j++) {
8                           System.out.println(numbers[i] + " - " + numbers[j]);
9                   }
10          }
11
12          for(int i = 0; i < n; i++) {
13                  for(int j = 0; j < n; j++) {
14                          System.out.println(numbers[i] + " - " + numbers[j]);
15                  }
16          }
17
18  }
```

2 instructions

n * n * 1 instructions

n * n * 1 instructions

Image credit - Nemanja Žunić @ Medium

Big-O Notation: O(n^2)

Nested For Loops

Many for-loops are formed by initializing an index variable to some value and incrementing that variable by 1 each time around the loop. The for-loop ends when the index reaches some limit.
The outer for-loop below uses index variable i. It increments i by 1 each time around the loop, and the iterations stop when i reaches n−1.

```
for (i = 0; i < n-1; i++) {
    small = i;
    for (j = i+1; j < n; j++)
        if (A[j] < A[small])
            small = j;
    temp = A[small];
    A[small] = A[i];
    A[i] = temp;
}
```

Strictly speaking, we must then add O(1) time to initialize the loop index and O(1) time for the first comparison of the loop index with the limit, because we test one more time than we go around the loop. However, unless it is possible to execute the loop zero times, the time to initialize the loop and test the limit once is a low-order term that can be dropped by the summation rule.

For simple for loops, the difference between the final and initial values, divided by the amount by which the index variable is incremented tells us how many times we go around the loop. That count is exact, unless there are ways to exit the loop via a jump statement; it is an upper bound on the number of iterations in any case. For instance, the for-loop in the first line iterates (n − 1) − 0/1 = n − 1 times, since 0 is the initial value of i, n − 1 is the highest value reached by i (i.e., when i reaches n−1, the loop stops and no iteration occurs with i = n−1), and 1 is added to i at each iteration of the loop.

```
(3)     for (j = 0; j < n; j++)
(4)         A[i][j] = 0;
```

We know that line (4) takes *O(1)* time. Clearly, we go around the loop n times, as we can determine by subtracting the lower limit from the upper limit found on line (3) and then adding 1. Since the body, line (4), takes *O(1)* time, we can neglect the time to increment j and the time to compare j with n, both of which are also *O(1)*. Thus, the running time of lines (3) and (4) is the product of n and *O(1)*, which is *O(n)*.

```
(2)     for (i = 0; i < n; i++)
(3)         for (j = 0; j < n; j++)
(4)             A[i][j] = 0;
```

We have already established that the loop of lines (3) and (4) takes *O(n)* time. Thus, we can neglect the *O(1)* time to increment i and to test whether i < n in each iteration, concluding that each iteration of the outer loop takes *O(n)* time. The initialization i = 0 of the outer loop and the (n + 1)st test of the condition i <n likewise take *O(1)* time and can be neglected. Finally, we observe that we go around the outer loop n times, taking *O(n)* time for each iteration, giving a total *O(n^2)* running time.

Big-O Notation: O(2^n)

Exponential Time

An exponential-time algorithm is one whose running time grows as an exponential function of the size of its input.

n	2^n
1	2
10	~1000
20	~1M
30	~1B

Examples:

Trying out every possible binary string of length n.
(E.g. to crack a password).

```
n    = 10    |     100   |      1000
n^2  = 100   |   10000   |   1000000
k^n  = k^10  |   k^100   |    k^1000
```

Exponential is worse than polynomial.

O(n^2) falls into the quadratic category, which is a type of polynomial (the special case of the exponent being equal to 2) and better than exponential.

Exponential is much worse than polynomial.

Example. Suppose the running time of a function is 2^n. Further suppose that each primitive operation can be executed in one micro second (i.e., 10^{-6} seconds). Then solving the problem for n equals 100 will take $10^{-6} \times 2^{100}$ seconds. This equals 1.26765×10^{24} seconds, which is more than 10,000 trillion years, more than the lifetime of the universe by several orders of magnitude. You get the point. The function starts innocuously enough but grows extremely rapidly and reaches astronomical proportions very quickly and for a fairly small value of n. Programs or functions whose running time is exponential can be useful only for tiny inputs.

Big-O Notation: O(n!)

Factorial Time

This algorithm's running time grows in proportion to n!, a really large number. O(n!) becomes too large for modern computers as soon as n is greater than 15+ (or upper teens). This issue shows up when you try to solve a problem by trying out every possibility, as in the traveling salesman problem.

n	n!
1	1
10	~3.6M
20	~2.43×10^{18}
30	~2.65×10^{32}

Examples:

- Go through all Permutations of a string
- Traveling Salesman Problem (brute-force solution)

  Traveling Salesman has a naive solution that's O(n!), but it has a dynamic programming solution that's O(n^2 * 2^n)

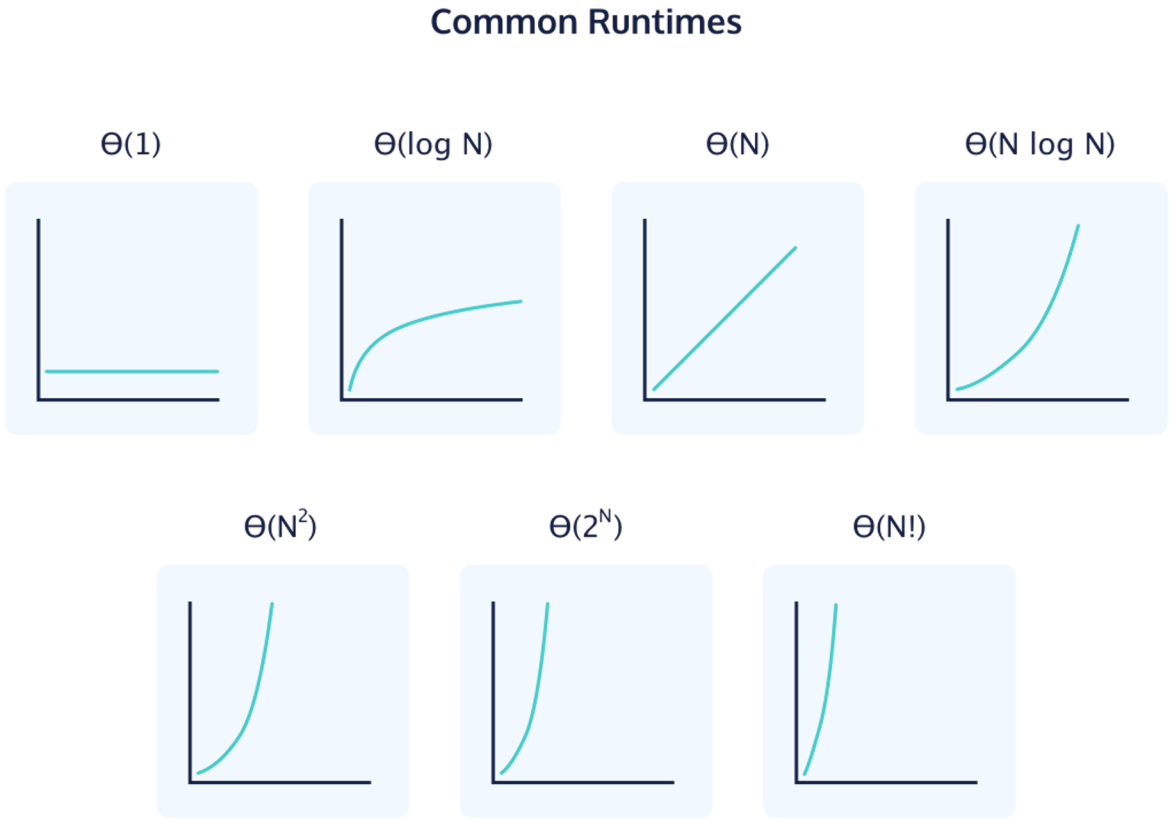


Image credit - https://www.codecademy.com/

Time Complexities

Comparison Table

Name	Complexity class	Running time ($T(n)$)	Examples of running times	Example algorithms
constant time		$O(1)$	10	Finding the median value in a sorted array of numbers. Calculating $(-1)^n$
inverse Ackermann time		$O(\alpha(n))$		Amortized time per operation using a disjoint set
iterated logarithmic time		$O(\log^* n)$		Distributed coloring of cycles
log-logarithmic		$O(\log \log n)$		Amortized time per operation using a bounded priority queue[2]
logarithmic time	DLOGTIME	$O(\log n)$	$\log n$, $\log(n^2)$	Binary search
polylogarithmic time		$\text{poly}(\log n)$	$(\log n)^2$	
fractional power		$O(n^c)$ where $0 < c < 1$	$n^{\frac{1}{2}}$, $n^{\frac{2}{3}}$	Searching in a kd-tree
linear time		$O(n)$	n, $2n + 5$	Finding the smallest or largest item in an unsorted array. Kadane's algorithm. Linear search
"n log-star n" time		$O(n \log^* n)$		Seidel's polygon triangulation algorithm.
linearithmic time		$O(n \log n)$	$n \log n$, $\log n!$	Fastest possible comparison sort Fast Fourier transform.
quasilinear time		$n\text{poly}(\log n)$	$n \log^2 n$	Multipoint polynomial evaluation
quadratic time		$O(n^2)$	n^2	Bubble sort, Insertion sort, Direct convolution
cubic time		$O(n^3)$	n^3	Naive multiplication of two $n \times n$ matrices Calculating partial correlation.
polynomial time	P	$2^{O(\log n)} = \text{poly}(n)$	$n^2 + n$, n^{10}	Karmarkar's algorithm for linear programming AKS primality test[3][4]
quasi-polynomial time	QP	$2^{\text{poly}(\log n)}$	$n^{\log \log n}$, $n^{\log n}$	Best-known $O(\log^2 n)$-approximation algorithm for the directed Steiner tree problem, best known parity game solver,[5] best known graph isomorphism algorithm
sub-exponential time (first definition)	SUBEXP	$O(2^{n^\epsilon})$ for all $\epsilon > 0$		Contains BPP unless EXPTIME (see below) equals MA.[6]
sub-exponential time (second definition)		$2^{o(n)}$	$2^{\sqrt[3]{n}}$	Best classical algorithm for integer factorization formerly-best algorithm for graph isomorphism
exponential time (with linear exponent)	E	$2^{O(n)}$	1.1^n, 10^n	Solving the traveling salesman problem using dynamic programming
exponential time	EXPTIME	$2^{\text{poly}(n)}$	2^n, 2^{n^2}	Solving matrix chain multiplication via brute-force search
factorial time		$O(n!)$	$n!$	Solving the traveling salesman problem via brute-force search
double exponential time	2-EXPTIME	$2^{2^{\text{poly}(n)}}$	2^{2^n}	Deciding the truth of a given statement in Presburger arithmetic

Image credit - https://en.wikipedia.org/

Image credit - Neso Academy

Calculating Time Complexity
More Examples

What is the time complexity of the following program?

```
void fun(int n) {
    int i, j, k, count = 0;
    for(i = n/2; i <= n; i++)
        for(j = 1; j+n/2 <= n; j++)
            for(k = 1; k <= n; k = k*2)
                count++;

}
```

```
void fun(int n) {
    int i, j, k, count = 0;
    for(i = n/2; i <= n; i++)
        for(j = 1; j+n/2 <= n; j++)
            for(k = 1; k <= n; k = k*2)
                count++;

}
```

Iter 1 i = n/2 + 0
Iter 2 i = n/2 + 1
Iter 3 i = n/2 + 2
Iter 4 i = n/2 + 3
⋮
Iter k i = n/2 + k-1 = n

$\frac{n}{2} + k - 1 = n$

$\Rightarrow k - 1 = n - \frac{n}{2} \Rightarrow k - 1 = \frac{2n - n}{2} \Rightarrow k = \frac{n}{2} + 1$

Loop here executes n/2 times, but if we take the least upper bound, we get O(n).

```
void fun(int n) {
    int i, j, k, count = 0;   <-- O(1)
    for(i = n/2; i <= n; i++)
        for(j = 1; j+n/2 <= n; j++)
            for(k = 1; k <= n; k = k*2)
                count++;

}
```

```
void fun(int n) {
    int i, j, k, count = 0;   <-- O(1)
    for(i = n/2; i <= n; i++)
        for(j = 1; j+n/2 <= n; j++)
            for(k = 1; k <= n; k = k*2)
                count++;

}
```

O(n)

Loop executes n/2 times. So we can take the least upper bound ‘n’.

Image credit - Neso Academy

Calculating Time Complexity
More Examples

```
void fun(int n) {
    int i, j, k, count = 0;
    for(i = n/2; i <= n; i++)
        for(j = 1; j+n/2 <= n; j++)
            for(k = 1; k <= n; k = k*2)
                count++;

}
```

$j <= n - \frac{n}{2} = \frac{n}{2}$

$k = \frac{n}{2}$

Iter 1	j = 1
Iter 2	j = 2
Iter 3	j = 3
Iter 4	j = 4
⋮	
Iter k	j = k = n/2

```
void fun(int n) {
    int i, j, k, count = 0;
    for(i = n/2; i <= n; i++)
        for(j = 1; j+n/2 <= n; j++)
            for(k = 1; k <= n; k = k*2)
                count++;

}
```

$n = 2^{p-1}$

$p - 1 = \log_2 n$

$p = \log_2 n + 1$

Iter 1	$k = 1 = 2^0$
Iter 2	$k = 2 = 2^1$
Iter 3	$k = 4 = 2^2$
Iter 4	$k = 8 = 2^3$
⋮	
Iter p	$k = 2^{p-1} = n$

```
void fun(int n) {
    int i, j, k, count = 0;      ◄ O(1)
    for(i = n/2; i <= n; i++)    ◄ O(1)
        for(j = 1; j+n/2 <= n; j++)   ◄ O(n)
            for(k = 1; k <= n; k = k*2)
                count++;                ◄ O(n)

}
```

Loop executes n/2 times. So we can take the least upper bound ‘n’.

```
void fun(int n) {
    int i, j, k, count = 0;      ◄ O(1)
    for(i = n/2; i <= n; i++)    ◄ O(1)
        for(j = 1; j+n/2 <= n; j++)   ◄ O(n)
            for(k = 1; k <= n; k = k*2)   ◄ O(n)
                count++;                  ◄ O(logn)

}
```

$n \times n \times \log n = O(n^2 \log n)$

Calculating Time Complexity

More Examples

Example:

1) Sum of n numbers

Statement	s/e	Frequency	Total steps
Algorithm Sum(list, n)	0	---	0
{	0	---	0
tempsum = 0;	1	1	1
for (i := 1 to n do)	1	n+1	n+1
tempsum := tempsum + list [i];	1	n	n
return tempsum;	1	1	1
}	0	---	0
		Total	2n+3

Image credit - https://academyera.com/

- Ignore constant multiplier.

Example	Step Count
int sum (int a[ ], int n)	0
{	
s=0;	1
for(i=0;i<n;i++)	1+ (n+1)+n
s=s+a[i];	n
return s; }	1
	= 3n+4 So, TC= O(n)

Image credit - EDULINE CSE KNOWLEDGE SHARING PLATFORM

Algorithm 2

```
1  int count_2(int n)
2  {
3    sum = 0 ———————————— 1
4    for i=1 to n { ———————— 2n
5      sum += n+1-i ———————— 3n
6    }
7    return sum ———————————— 1
8  }
```

The running time is $5n+2$

Algorithm 4

```
1  int count_0(int n)
2  {
3    sum = 0 ———————————— O(1)
4    for i=1 to n { ———————— O(n)
5      for j=1 to n { ———— O(n^2)
6        If i<=j then ———— O(n^2)
7          sum++ ———————— O(n^2)
8      }
9    }
10   return sum ———————— O(1)
11 }
```

The running time is $O(n^2)$

Image credit - Dr. Jicheng Fu Department of Computer Science University of Central Oklahoma

Big-O Performance Comparisons

Big - O Notation	Computations for 10 Elements	Computations For 100 Elements	Computations For 1000 Elements
O(1)	1	1	1
O(N)	10	100	1000
O(N^2)	100	10000	1000000
O(log N)	3	6	9
O(N log N)	30	600	9000
O(2^N)	1024	1.26e+29	1.07e+301
O(N!)	3628800	9.33e+157	4.02e+2567

Image credit - https://javaconceptoftheday.com/

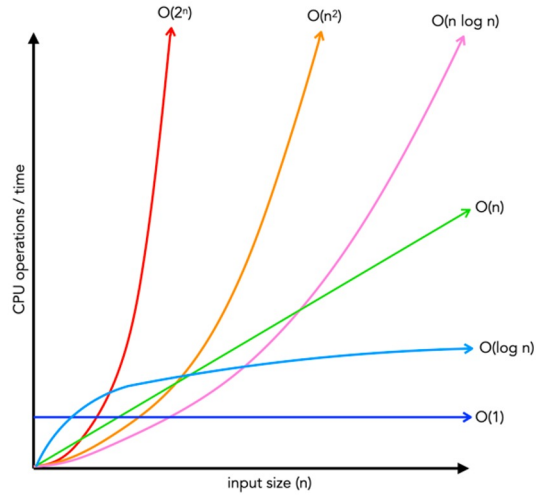


Image credit - https://craftofcoding.wordpress.com/

Big O Notation	Name	Example
O(1)	Constant runtime	Select an item with array index / object key
O(log n)	Logarithmic runtime	Binary search
O(n)	Linear runtime	For loops, Javascript map(), filter(), reduce()
O(n log n)	Linearithmic runtime	Sorting an array with merge sort
O(n^2)	Quadratic runtime	Sorting an array with bubble sort, 2 level nested loops
O(2^n)	Exponential runtime	Recursive calculation of Fibonacci numbers
O(n!)	Factorial runtime	Find all permutations of a given set / string

Image credit - https://www.sahinarslan.tech/

Notation	Type	Examples	Description
O(1)	Constant	Hash table access	Remains constant regardless of the size of the data set
O(log n)	Logarithmic	Binary search of a sorted table	Increases by a constant. If n doubles, the time to perform increases by a constant, smaller than n amount
O(< n)	Sublinear	Search using parallel processing	Performs at less than linear and more than logarithmic levels
O(n)	Linear	Finding an item in an unsorted list	Increases in proportion to n. If n doubles, the time to perform doubles
O(n log(n))	n log(n)	Quicksort, Merge Sort	Increases at a multiple of a constant
$O(n^2)$	Quadratic	Bubble sort	Increases in proportion to the product of n*n
$O(c^n)$	Exponential	Travelling salesman problem solved using dynamic programming	Increases based on the exponent n of a constant c
O(n!)	Factorial	Travelling salesman problem solved using brute force	Increases in proportion to the product of all numbers included (e.g., 1*2*3*4...)

Image credit - https://www.ml-science.com/

Big-O Performance Comparisons

Data Structures

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Indexing	Search	Insertion	Deletion	Indexing	Search	Insertion	Deletion	
Basic Array	O(1)	O(n)	-	-	O(1)	O(n)	-	-	O(n)
Dynamic Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartresian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Image credit - https://www.hackerearth.com/

Big-O Performance Comparisons

Searching

Algorithm	Data Structure	Time Complexity		Space Complexity
		Average	Worst	Worst
Depth First Search (DFS)	Graph of |V| vertices and |E| edges	-	O(|E| + |V|)	O(|V|)
Breadth First Search (BFS)	Graph of |V| vertices and |E| edges	-	O(|E| + |V|)	O(|V|)
Binary search	Sorted array of n elements	O(log(n))	O(log(n))	O(1)
Linear (Brute Force)	Array	O(n)	O(n)	O(1)
Shortest path by Dijkstra, using a Min-heap as priority queue	Graph with |V| vertices and |E| edges	O((|V| + |E|) log |V|)	O((|V| + |E|) log |V|)	O(|V|)
Shortest path by Dijkstra, using an unsorted array as priority queue	Graph with |V| vertices and |E| edges	O(|V|^2)	O(|V|^2)	O(|V|)
Shortest path by Bellman-Ford	Graph with |V| vertices and |E| edges	O(|V||E|)	O(|V||E|)	O(|V|)

Image credit - https://www.hackerearth.com/

Big-O Performance Comparisons

Sorting

Algorithm	Data Structure	Time Complexity			Worst Case Auxiliary Space Complexity
		Best	Average	Worst	Worst
Quicksort	Array	O(n log(n))	O(n log(n))	O(n^2)	O(n)
Mergesort	Array	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Heapsort	Array	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	Array	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	Array	O(n)	O(n^2)	O(n^2)	O(1)
Select Sort	Array	O(n^2)	O(n^2)	O(n^2)	O(1)
Bucket Sort	Array	O(n+k)	O(n+k)	O(n^2)	O(nk)
Radix Sort	Array	O(nk)	O(nk)	O(nk)	O(n+k)

Image credit - https://www.hackerearth.com/

Big-O Performance Comparisons

Heaps

Heaps	Time Complexity						
	Heapify	Find Max	Extract Max	Increase Key	Insert	Delete	Merge
Linked List (sorted)	-	O(1)	O(1)	O(n)	O(n)	O(1)	O(m+n)
Linked List (unsorted)	-	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)
Binary Heap	O(n)	O(1)	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(m+n)
Binomial Heap	-	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Fibonacci Heap	-	O(1)	O(log(n))*	O(1)*	O(1)	O(log(n))*	O(1)

Graphs

Node / Edge Management	Storage	Add Vertex	Add Edge	Remove Vertex	Remove Edge	Query
Adjacency list	O(|V|+|E|)	O(1)	O(1)	O(|V| + |E|)	O(|E|)	O(|V|)
Incidence list	O(|V|+|E|)	O(1)	O(1)	O(|E|)	O(|E|)	O(|E|)
Adjacency matrix	O(|V|^2)	O(|V|^2)	O(1)	O(|V|^2)	O(1)	O(1)
Incidence matrix	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|E|)

Image credit - https://www.hackerearth.com/