Introduction to Data Structures and Algorithms

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have some data which has, player's **name** "Virat" and **age** 26. Here "Virat" is of **String** data type and 26 is of **integer** data type.

We can organize this data as a record like **Player** record, which will have both player's name and age in it. Now we can collect and store player's records in a file or database as a data structure. **For example**: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33

If you are aware of Object Oriented programming concepts, then a `class` also does the same thing, it collects different type of data under one single entity. The only difference being, data structures provides for techniques to access and manipulate data efficiently.

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily. It represents the knowledge of data to be organized in memory. It should be designed and implemented in such a way that it reduces the complexity and increases the efficiency.

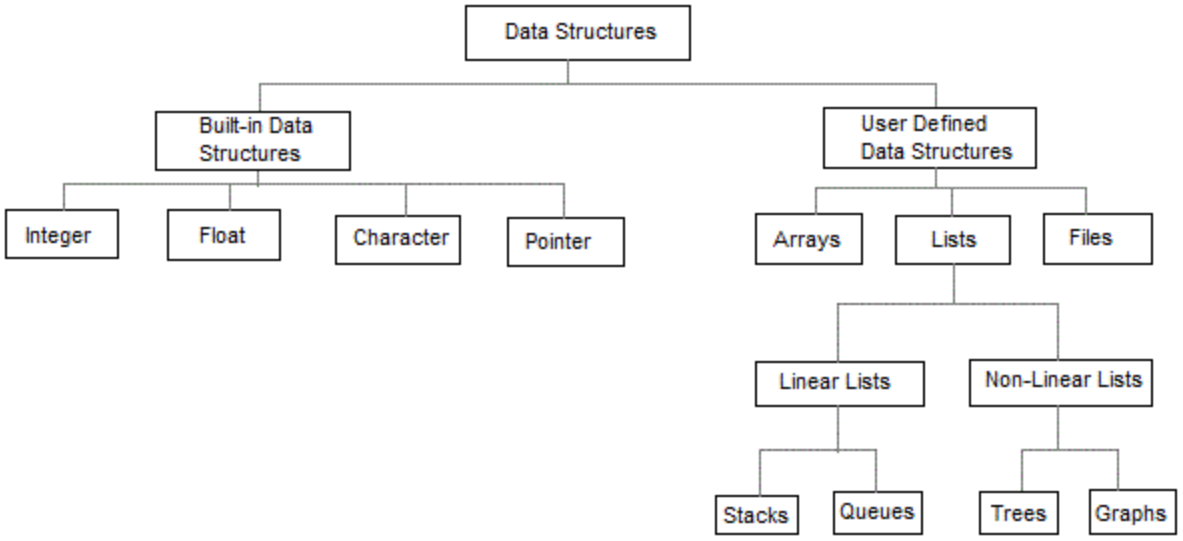
Basic types of Data Structures

As we have discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of **Abstract Data Structure** are :

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.



INTRODUCTION TO DATA STRUCTURES

The data structures can also be classified on the basis of the following characteristics:

Characterstic	**Description**
Linear	In Linear data structures,the data items are arranged in a linear sequence. Example: **Array**
Non-Linear	In Non-Linear data structures,the data items are not in sequence. Example: **Tree**, **Graph**
Homogeneous	In homogeneous data structures,all the elements are of same type. Example: **Array**
Non-Homogeneous	In Non-Homogeneous data structure, the elements may or may not be of the same type. Example: **Structures**
Static	Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: **Array**
Dynamic	Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations changes. Example: **Linked List created using pointers**

What is an Algorithm ?

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as **pseudocode** or using a **flowchart**.

Every Algorithm must satisfy the following properties:

1. **Input**- There should be 0 or more inputs supplied externally to the algorithm.
2. **Output**- There should be atleast 1 output obtained.
3. **Definiteness**- Every step of the algorithm should be clear and well defined.
4. **Finiteness**- The algorithm should have finite number of steps.
5. **Correctness**- Every step of the algorithm must generate a correct output.

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties :

1. Time Complexity
2. Space Complexity

Space Complexity

Its the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

An algorithm generally requires space for following components :

- **Instruction Space:** Its the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
- **Data Space:** Its the space required to store all the constants and variables(including temporary variables) value.
- **Environment Space:** Its the space required to store the environment information needed to resume the suspended function.

To learn about Space Complexity in detail, jump to the Space Complexity (space-complexity-of-algorithms) tutorial.

Time Complexity

Time Complexity is a way to represent the amount of time required by the program to run till its completion. It's generally a good practice to try to keep the time required minimum, so that our algorithm completes it's execution in the minimum time possible. We will study about Time Complexity (time-complexity-of-algorithms) in details in later sections.

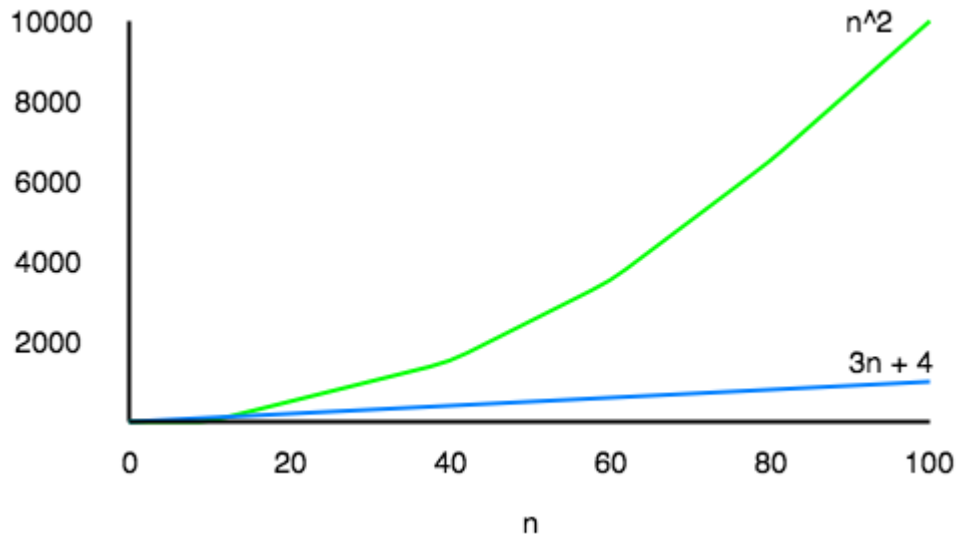
NOTE: Before going deep into data structure, you should have a good knowledge of programming either in C or in C++ or Java or Python etc.

Asymptotic Notations

When it comes to analysing the complexity of any algorithm in terms of time and space, we can never provide an exact number to define the time required and the space required by the algorithm, instead we express it using some standard notations, also known as **Asymptotic Notations**.

When we analyse any algorithm, we generally get a formula to represent the amount of time required for execution or the time required by the computer to run the lines of code of the algorithm, number of memory accesses, number of comparisons, temporary variables occupying memory space etc. This formula often contains unimportant details that don't really tell us anything about the running time.

Let us take an example, if some algorithm has a time complexity of $T(n) = (n^2 + 3n + 4)$, which is a quadratic equation. For large values of `n`, the `3n + 4` part will become insignificant compared to the `n^2` part.



For n = 1000, `n^2` will be `1000000` while `3n + 4` will be `3004`.

Also, When we compare the execution times of two algorithms the constant coefficients of higher order terms are also neglected.

An algorithm that takes a time of `200n^2` will be faster than some other algorithm that takes `n^3` time, for any value of `n` larger than `200`. Since we're only interested in the asymptotic behavior of the growth of the function, the constant factor can be ignored too.

What is Asymptotic Behaviour

The word **Asymptotic** means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken).

Remember studying about **Limits** in High School, this is the same.

The only difference being, here we do not have to find the value of any expression where `n` is approaching any finite number or infinity, but in case of Asymptotic notations, we use the same model to ignore the constant factors and insignificant parts of an expression, to device a better way of representing complexities of algorithms, in a single coefficient, so that comparison between algorithms can be done easily.

Let's take an example to understand this:

If we have two algorithms with the following expressions representing the time required by them for execution, then:

Expression 1: $(20n^2 + 3n - 4)$

Expression 2: $(n^3 + 100n - 2)$

Now, as per asymptotic notations, we should just worry about how the function will grow as the value of `n` (input) will grow, and that will entirely depend on n^2 for the Expression 1, and on n^3 for Expression 2. Hence, we can clearly say that the algorithm for which running time is represented by the Expression 2, will grow faster than the other one, simply by analysing the highest power coeeficient and ignoring the other constants(20 in $20n^2$) and insignificant parts of the expression(`3n - 4` and `100n - 2`).

The main idea behind casting aside the less important part is to make things **manageable**.

All we need to do is, first analyse the algorithm to find out an expression to define it's time requirements and then analyse how that expression will grow as the input(n) will grow.

Types of Asymptotic Notations

We use three types of asymptotic notations to represent the growth of any algorithm, as input increases:

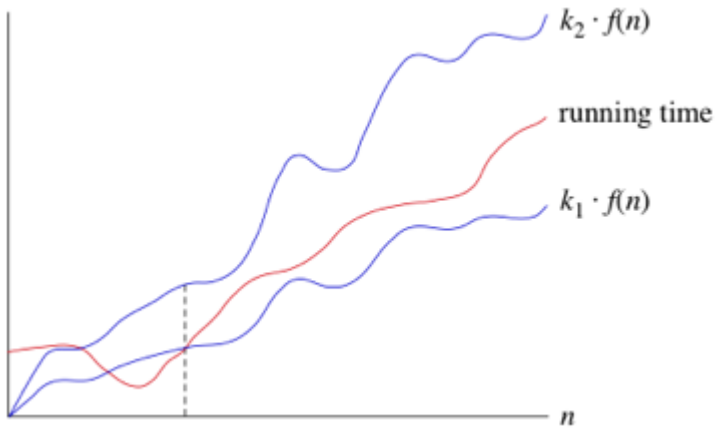
1. Big Theta (Θ)
2. Big Oh(O)
3. Big Omega (Ω)

Tight Bounds: Theta

When we say tight bounds, we mean that the time compexity represented by the Big-Θ notation is like the average value or range within which the actual time of execution of the algorithm will be.

For example, if for some algorithm the time complexity is represented by the expression $3n^2 + 5n$, and we use the Big-Θ notation to represent this, then the time complexity would be $\Theta(n^2)$, ignoring the constant coefficient and removing the insignificant part, which is 5n.

Here, in the example above, complexity of $\Theta(n^2)$ means, that the avaerage time for any input `n` will remain in between, `k1 * n^2` and `k2 * n^2`, where k1, k2 are two constants, therby tightly binding the expression rpresenting the growth of the algorithm.



Upper Bounds: Big-O

This notation is known as the upper bound of the algorithm, or a Worst Case of an algorithm.

It tells us that a certain function will never exceed a specified time for any value of input `n`.

The question is why we need this representation when we already have the big-Θ notation, which represents the tightly bound running time for any algorithm. Let's take a small example to understand this.

Consider Linear Search algorithm, in which we traverse an array elements, one by one to search a given number.

In **Worst case**, starting from the front of the array, we find the element or number we are searching for at the end, which will lead to a time complexity of `n`, where `n` represents the number of total elements.

But it can happen, that the element that we are searching for is the first element of the array, in which case the time complexity will be `1`.

Now in this case, saying that the big-Θ or tight bound time complexity for Linear search is Θ(n), will mean that the time required will always be related to `n`, as this is the right way to represent the average time complexity, but when we use the big-O notation, we mean to say that the time complexity is O(n), which means that the time complexity will never exceed `n`, defining the upper bound, hence saying that it can be less than or equal to `n`, which is the correct representation.

This is the reason, most of the time you will see Big-O notation being used to represent the time complexity of any algorithm, because it makes more sense.

Lower Bounds: Omega

Big Omega notation is used to define the **lower bound** of any algorithm or we can say **the best case** of any algorithm.

This always indicates the minimum time required for any algorithm for all input values, therefore the best case of any algorithm.

In simple words, when we represent a time complexity for any algorithm in the form of big-Ω, we mean that the algorithm will take atleast this much time to cmplete it's execution. It can definitely take more time than this too.

Induction Proofs: Introduction *(page 1 of 3)*

If you're reading this module, then you've probably just covered induction in your algebra class, and you're feeling somewhat uneasy about the whole thing. If you're anything like me, you're having the same thoughts that I had: "Can't you prove *anything* is true, if you *assume* it to be true in the first place? What's left to prove, if you've already assumed? Isn't that cheating somehow?"

Well...

First off, don't get mad at your instructor. Induction makes perfect sense to him, and he honestly thinks he's explained it clearly. When I took this stuff, I know I had a good professor, so I know I got a "good" lesson. But I still didn't trust induction. I'll try to pick up where your instructor left off, and fill in some of the holes.

Induction proofs have four components: (1) the thing you want to prove, (2) the beginning step (usually "let $n = 1$"), (3) the assumption step ("let $n = k$"), and (4) the induction step ("let $n = k + 1$"). Here's the thinking:

You have been working with some numbers, and have noticed something that seems to be a pattern. To use the standard example, let's say that you have noticed that, when you add up all the numbers from 1 to n (that is, $1 + 2 + 3 + 4 + ... + n$), you get a total that equals $^{(n)(n+1)}/_{2}$. That is, for every number that you've checked so far, you get

$$1 + 2 + 3 + 4 + ... + n = {}^{(n)(n+1)}/_{2}$$

For convenience, we'll call this formula "$(*)$", or "star". You'd like to know if $(*)$ is true for all whole numbers, but you obviously can't check every single whole number, since the whole numbers never end. You only know that it's true for the relatively few numbers that you've actually checked. Of course, adding up all those numbers quickly becomes tedious, so you'd *really* like for $(*)$ to work! But to use $(*)$ for whatever number you'd like, you somehow have to prove that $(*)$ works everywhere. Induction proofs allow you to prove that the formula works "everywhere" without your having to actually *show* that it works everywhere (by doing the infinitely-many additions).

So you have the first part of an induction proof, the formula that you'd like to prove:

$(*)$ For all natural numbers n, $1 + 2 + 3 + 4 + ... + n = {}^{(n)(n+1)}/_{2}$

But is $(*)$ *ever* true *anywhere?* If you can't find any place, any particular number, for which $(*)$ is known to be true, then there's really no point in continuing. So you do the second part, which is to show that $(*)$ is indeed true in some particular place:

Let $n = 1$. Then $1 + 2 + 3 + 4 + ... + n$ is actually just 1; that is:

$${}^{(n)(n+1)}/_{2} = {}^{(1)(1+1)}/_{2} = {}^{(1)(2)}/_{2} = 1.$$

Then we get $1 = 1$, so $(*)$ is true at $n = 1$.

There; we've shown that $(*)$ works in some particular place. But does it work anywhere else? Well, that was our problem in the first place: we know that it works in a few places, but we need to prove that $(*)$ works *everywhere*, and we'll never do that by proving one case at a time. So we need the third and fourth parts of the induction proof. The third part is the assumption part:

Let $n = k$.

Assume that, for $n = k$, $(*)$ works; that is, assume that:

$$1 + 2 + 3 + 4 + ... + k = {}^{(k)(k+1)}/_{2}$$

This is not the same as the second part, where we named the actual place where $(*)$ worked. This part just says "Let's assume that $(*)$ works, somewhere, out there in space, I'm not saying where, I don't maybe even *know* where; just *somewhere*." And you think "So what?" Here's what: the fourth step. If we can prove, *assuming (*) works at* $n = k$, that $(*)$ then works at $n = k + 1$ (that is, if it works *some* place, then it must also work at the *next* place), then, since we know of a *certain* place ($n = 1$) where $(*)$ works, we will have proved that $(*)$ works everywhere:

Let $n = k + 1$.	
Then $1 + 2 + 3 + 4 + ... + k + (k + 1)$	[the left-hand side of $(*)$]

$= [1 + 2 + 3 + 4 + ... + k] + k + 1$	[from part three]
$= [{}^{(k)(k+1)}/_{2}] + k + 1$	[our assumption]
$= [{}^{(k)(k+1)}/_{2}] + {}^{2(k+1)}/_{2}$	[common denominator]
$= {}^{(k)(k+1)+2(k+1)}/_{2}$	[adding fractions]
$= {}^{(k+2)(k+1)}/_{2}$	[simplifying]
$= {}^{((k+1)+1)(k+1)}/_{2}$	[restating in "$k + 1$" form]

This last line is the right-hand side of $(*)$. In other words, if we assume that $(*)$ works as some unnamed faceless number k, then we can show (by using that assumption) that $(*)$ works at the next number, $k + 1$. And we already know of a number where $(*)$ works! Since we showed that $(*)$ works at $n = 1$, the assumption and induction steps tell us that $(*)$ then works at $n = 2$, and then *by induction* $(*)$ works at $n = 3$, and then *by induction*, $(*)$ works at $n = 4$, and so on. The assumption and induction steps allow us to make the jump from "It works here and there" to "It works everywhere!" It's kinda like dominoes: instead of knocking each one down individually, you say "If I can knock down one of them, then that one will knock down the next one, and then the next one, and eventually all of them; and – look! – I can knock down this first one right here."

Induction Proofs:
Examples of where induction fails *(page 2 of 3)*

If you're anything like I was, you're probably feeling a bit queasy about that assumption step. I mean, if we *assume* something is true, can't we prove *anything?*

Well, actually, no. But I'll bet your text gives you a list of formulas, and asks you to "try to prove..." or "determine whether true...", and, whaddya know, they all turn out to be true! So, in spite of their good intentions, your teacher and your book have been quite misleading. To help you feel more confident about induction, let's try to prove a couple of statements that we *know* are wrong, so you can see that you can't use induction to prove something that ain't so. Here's the first one:

- **(*) For all $n > 0$, $n^3 \leq n^2$**

  We know perfectly well that this statement just isn't so: cubes of whole numbers are bigger than squares (except for the cube and square of 1, where they're equal). But let's try to prove this false statement, and see what happens.

  Let $n = 1$. Then $n^3 = 1^3 = 1$ and $n^2 = 1^2 = 1$, and $1 \leq 1$.

  Then $(*)$ holds for $n = 1$.

  Hmm...that's weird; the first part worked. Well, let's continue, and see what happens:

  Let $n = k$.

  Assume that, for $n = k$, we have $k^3 \leq k^2$.

  For this next bit, just trust me: everything will become clear at the end:

  Let $n = k + 1$.

  Whatever k is, we know that three of their square is bigger than just one of their square; that is, $3k^2 > k^2$, because $k^2 > 0$.

  Also, we know that three of the value is bigger than two; that is, $3k > 2k$, because $k > 0$.

  Also, we know that $k^3 > 0$, because $k > 0$.

  Also, $1 \geq 1$.

  Adding together all the left-hand sides and all the right-hand sides above, we get:

  $$k^3 + 3k^2 + 3k + 1 > k^2 + 2k + 1$$

  ...and this can be restated as:

  $$(k + 1)^3 > (k + 1)^2$$

  But we needed to prove just the opposite!!

  That is, $(*)$ failed the induction step. Even if $(*)$ is true in some one place, it will *not* be true at the next place. So, even though $(*)$ was true for $n = 1$, it was *not* true for $n = 2$, and **(*) fails**, as we knew it ought to!

Let's try another one. In this one, we'll do the steps out of order:

- **(*) For all n, $n + 1 < n$**

  Any child capable of counting on his fingers knows that this isn't true! But let's see what happens if we assume that it *is* true somewhere:

  Let $n = k$. Assume, for $n = k$, that $(*)$ holds; that is, assume that $k + 1 < k$

  Let $n = k + 1$. Then:

$$(k+1)+1 < (k)+1 = (k+1)$$

...so:

$$(k+1)+1 < (k+1)$$

...and (*) holds for $n = k+1$.

Well, that's a bit disturbing: if $(*)$ is true *anywhere*, then it's true *everywhere*. Ah, but we haven't shown $(*)$ to be true anywhere! And that's where the induction proof fails in this case. *You can't find any number for which this (*) is true*. Since there is no starting point (no first domino, as it were), then **induction fails**, just as we knew it ought to.

I hope these examples, in showing that induction cannot prove things that are not true, have increased your confidence in the technique. Induction is actually quite powerful and clever, and it would be a shame for you not to have caught a glimpse of that.

Induction Proofs: Worked examples *(page 3 of 3)*

- **(*) For $n \geq 1$, $2 + 2^2 + 2^3 + 2^4 + ... + 2^n = 2^{n+1} - 2$**

  Let $n = 1$. Then:

  $$2 + 2^2 + 2^3 + 2^4 + ... + 2^n = 2^1 = 2$$

  ...and:

  $$2^{n+1} - 2 = 2^{1+1} - 2 = 2^2 - 2 = 4 - 2 = 2$$

  So $(*)$ works for $n = 1$.

  Assume, for $n = k$, that $(*)$ holds; that is, that

  $$2 + 2^2 + 2^3 + 2^4 + ... + 2^k = 2^{k+1} - 2$$

  Let $n = k + 1$.

  $$\begin{aligned} &2 + 2^2 + 2^3 + 2^4 + ... + 2^k + 2^{k+1} \\ &\quad = [2 + 2^2 + 2^3 + 2^4 + ... + 2^k] + 2^{k+1} \\ &\quad = [2^{k+1} - 2] + 2^{k+1} \\ &\quad = 2 \times 2^{k+1} - 2 \\ &\quad = 2^1 \times 2^{k+1} - 2 \\ &\quad = 2^{k+1+1} - 2 \\ &\quad = 2^{(k+1)+1} - 2 \end{aligned}$$

  Then **(*) works** for $n = k + 1$.

Note this common technique: In the "$n = k + 1$" step, it is usually a good first step to write out the whole formula in terms of $k + 1$, and then break off the "$n = k$" part, so you can replace it with whatever assumption you made about $n = k$ in the previous step. Then you manipulate and simplify, and try to rearrange things to get the right-hand side of the formula in terms of $k + 1$.

- **(*) For $n \geq 1$, $1 \times 2 + 2 \times 3 + 3 \times 4 + ... + (n)(n+1) = (n)(n+1)(n+2)/3$**

  Let $n = 1$.

  Then the left-hand side of $(*)$ is $1 \times 2 = 2$
  and the right-hand side of $(*)$ is $(1)(2)(3)/3 = 2$.

  So $(*)$ holds for $n = 1$. Assume, for $n = k$, that $(*)$ holds; that is, assume that

  $$1 \times 2 + 2 \times 3 + 3 \times 4 + ... + (k)(k+1) = (k)(k+1)(k+2)/3$$

  Let $n = k + 1$. The left-hand side of $(*)$ then gives us:

  $$\begin{aligned} &1 \times 2 + 2 \times 3 + 3 \times 4 + ... + (k)(k+1) + (k+1)((k+1)+1) \\ &\quad = [1 \times 2 + 2 \times 3 + 3 \times 4 + ... + (k)(k+1)] + (k+1)((k+1)+1) \\ &\quad = [(k)(k+1)(k+2)/3] + (k+1)((k+1)+1) \\ &\quad = (k)(k+1)(k+2)/3 + (k+1)(k+2) \\ &\quad = (k)(k+1)(k+2)/3 + 3(k+1)(k+2)/3 \end{aligned}$$

$= (k+3)(k+1)(k+2)/3$

$= (k+1)(k+2)(k+3)/3$

$= (k+1)((k+1)+1)((k+1)+2)/3$

...which is the right-hand side of $(*)$. Then **(*) works** for all $n \geq 1$.

- **(*) For $n \geq 5$, $4n < 2^n$.**

  This one doesn't start at $n = 1$, and involves an inequality instead of an equation. (If you graph $4x$ and 2^x on the same axes, you'll see why we have to start at $n = 5$, instead of the customary $n = 1$.)

  Let $n = 5$.

  Then $4n = 4 \times 5 = 20$, and $2^n = 2^5 = 32$.

  Since $20 < 32$, then $(*)$ holds at $n = 5$.

  Assume, for $n = k$, that $(*)$ holds; that is, assume that $4k < 2^k$

  Let $n = k + 1$.

  The left-hand side of $(*)$ gives us $4(k + 1) = 4k + 4$, and, by assumption,

  $$[4k] + 4 < [2^k] + 4$$

  Since $k \geq 5$, then $4 < 32 \leq 2^k$. Then we get

  $$2^k + 4 < 2^k + 2^k = 2 \times 2^k = 2^1 \times 2^k = 2^{k+1}$$

  Then $4(k+1) < 2^{k+1}$, and $(*)$ holds for $n = k + 1$.

  Then **(*) holds** for all $n \geq 5$.

- **(*) For all $n \geq 1$, $8^n - 3^n$ is divisible by 5.**

  Let $n = 1$.

  Then the expression $8^n - 3^n$ evaluates to $8^1 - 3^1 = 8 - 3 = 5$, which is clearly divisible by 5.

  Assume, for $n = k$, that $(*)$ holds; that is, that $8^k - 3^k$ is divisible by 5.

  Let $n = k + 1$.

  Then:

  $$8^{k+1} - 3^{k+1} = 8^{k+1} - 3 \times 8^k + 3 \times 8^k - 3^{k+1}$$

  $$= 8^k(8 - 3) + 3(8^k - 3^k) = 8^k(5) + 3(8^k - 3^k)$$

  The first term in $8^k(5) + 3(8^k - 3^k)$ has 5 as a factor (explicitly), and the second term is divisible by 5 (by assumption). Since we can factor a 5 out of both terms, then the entire expression, $8^k(5) + 3(8^k - 3^k) = 8^{k+1} - 3^{k+1}$, must be divisible by 5.

  Then **(*) holds** for $n = k + 1$, and thus for all $n \geq 1$.