[](https://devopedia.org/)

[**DEVOPEDIA**](https://devopedia.org/)

®

**for developers. by developers.**

Top of Form

-

Bottom of Form

Algorithmic Complexity

* [Summary](https://devopedia.org/algorithmic-complexity#summary)
* [Discussion](https://devopedia.org/algorithmic-complexity#discussion)
  + [Can you give real-world examples of various algorithmic complexities?](https://devopedia.org/algorithmic-complexity#qst-ans-0)
  + [What notations are used to represent algorithmic complexity?](https://devopedia.org/algorithmic-complexity#qst-ans-1)
  + [What does it mean to state best-case, worst-case and average time complexity of algorithms?](https://devopedia.org/algorithmic-complexity#qst-ans-2)
  + [Why should we care about an algorithm's performance when processors are getting faster and memories are getting cheaper?](https://devopedia.org/algorithmic-complexity#qst-ans-3)
  + [Are there techniques to figure out the complexity of algorithms?](https://devopedia.org/algorithmic-complexity#qst-ans-4)
  + [If an algorithm is inefficient, does that mean that we can't use it?](https://devopedia.org/algorithmic-complexity#qst-ans-5)
  + [Since algorithmic complexity is about algorithms, is it relevant to talk about data structures?](https://devopedia.org/algorithmic-complexity#qst-ans-6)
  + [Can you state the complexity of well-known sorting algorithms?](https://devopedia.org/algorithmic-complexity#qst-ans-7)
  + [Can you give the complexity of some important algorithms?](https://devopedia.org/algorithmic-complexity#qst-ans-8)
* [Milestones](https://devopedia.org/algorithmic-complexity#milestones)
* [Sample Code](https://devopedia.org/algorithmic-complexity#sample-code)
* [References](https://devopedia.org/algorithmic-complexity#references)
* [Further Reading](https://devopedia.org/algorithmic-complexity#further-reading)
* [Article Stats](https://devopedia.org/algorithmic-complexity#article-stats)
* [Cite As](https://devopedia.org/algorithmic-complexity#cite-as)

Algorithmic complexity is a measure of how long an algorithm would take to complete given an input of size n. If an algorithm has to scale, it should compute the result within a finite and practical time bound even for large values of n. For this reason, complexity is calculated asymptotically as n approaches infinity. While complexity is usually in terms of time, sometimes complexity is also analyzed in terms of space, which translates to the algorithm's memory requirements.

Analysis of an algorithm's complexity is helpful when comparing algorithms or seeking improvements. Algorithmic complexity falls within a branch of theoretical computer science called computational complexity theory. It's important to note that we're concerned about the order of an algorithm's complexity, not the actual execution time in terms of milliseconds.

Algorithmic complexity is also called *complexity* or *running time*.

Discussion

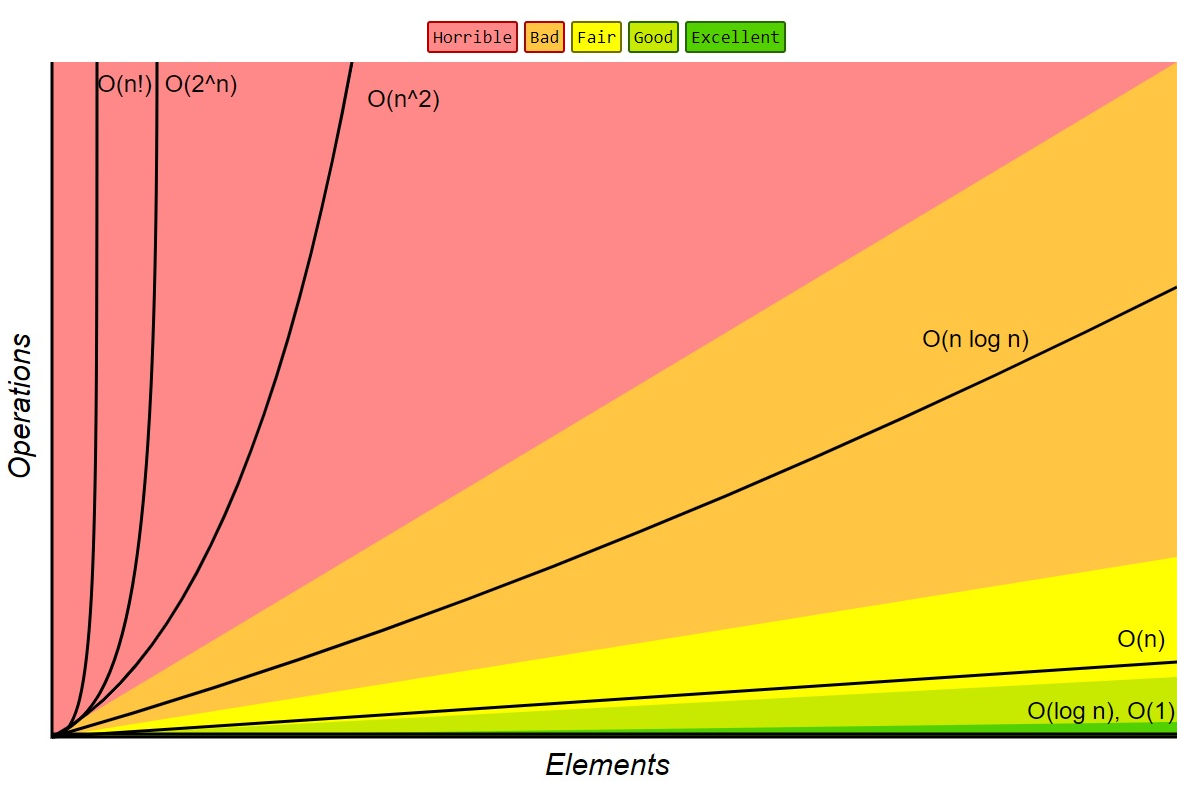
* Can you give real-world examples of various algorithmic complexities?

Suppose you're looking for a specific item in a long unsorted list, you'll probably compare with each item. Search time is proportional to the list size. Here complexity is said to be *linear*.

On the other hand, if you search for a word in a dictionary, the search will be faster because the words are in sorted order, you know the order and can quickly decide if you need to turn to earlier pages or later pages. This is an example of *logarithmic* complexity.

If you're asked to pick out the first word in a dictionary, this operation is of *constant* time complexity, regardless of number of words in the dictionary. Likewise, joining the end of a queue in a bank is of constant complexity regardless of how long the queue is.

Suppose you are given an unsorted list and asked to find all duplicates, then the complexity becomes *quadratic*. Checking for duplicates for one item is of linear complexity. If we do this for all items, complexity becomes quadratic. Similarly, if all people in a room are asked to shake hands with every other person, the complexity is quadratic.

* What notations are used to represent algorithmic complexity?

**Order of growth of algorithms specified in Big-O notation. Source: Big-O Cheat Sheet, 2016.**

*Big-O notation* is the prevalent notation to represent algorithmic complexity. It gives an upper bound on complexity and hence it signifies the worst-case performance of the algorithm. With such a notation, it's easy to compare different algorithms because the notation tells clearly how the algorithm scales when input size increases. This is often called the *order of growth*.

Constant runtime is represented by O(1)O(1); linear growth is O(n)O(n); logarithmic growth is O(logn)O(logn); log-linear growth is O(nlogn)O(nlogn); quadratic growth is O(n2)O(n2); exponential growth is O(2n)O(2n); factorial growth is O(n!)O(n!). Their orders of growth can also be compared from best to worst:

O(1)<O(logn)<O(√n)<O(n)<O(nlogn)<O(n2)<O(n3)<O(2n)<O(10n)<O(n!)O(1)<O(logn)<O(n)<O(n)<O(nlogn)<O(n2)<O(n3)<O(2n)<O(10n)<O(n!)

In complexity analysis, only the dominant term is retained. For example, if an algorithm requires 2n3+logn+42n3+logn+4 operations, its order is said to be O(n3)O(n3) since 2n32n3 is the dominant term. Constants and scaling factors are ignored since we are concerned only about asymptotic.

Audrey Nasar gives [formal definitions of Big-O](https://scholarworks.umt.edu/cgi/viewcontent.cgi?article=1375&context=tme). Wikipedia [lists orders of common functions](https://en.wikipedia.org/wiki/Big_O_notation).

* What does it mean to state best-case, worst-case and average time complexity of algorithms?

Let's take the example of searching for an item sequentially within a list of unsorted items. If we're lucky, the item may occur at the start of the list. If we're unlucky, it may be the last item in the list. The former is called *best-case complexity* and the latter is called *worst-case complexity*. If the searched item is always the first one, then complexity is O(1)O(1); if it's always the last one, then complexity is O(n)O(n). We can also calculate the *average complexity*, which will turn out to be O(n)O(n). The term "complexity" normally refers to worst-case complexity.

Mathematically, different notations are defined (example is for linear complexity):

* + Worst-case or upper bound: Big-O: O(n)O(n)
  + Best-case or lower bound: Big-Omega: Ω(n)Ω(n)
  + Average-case: Big-Theta: Θ(n)Θ(n)
  + Non-tight upper bound: o(n)o(n)
  + Non-tight lower bound: ω(n)ω(n)

When upper or lower bounds don't coincide with average complexity, we can call them non-tight bounds.

As an example, Quicksort's complexity is Ω(nlogn)Ω(nlogn), Θ(nlogn)Θ(nlogn) and O(n2)O(n2).

There's also *amortized complexity* in which complexity is calculated by averaging over a sequence of operations.

* Why should we care about an algorithm's performance when processors are getting faster and memories are getting cheaper?

Complexity analysis doesn't concern itself with actual execution time, which depends on processor speed, instruction set, disk speed, compiler, etc. Likewise, the same algorithm written in assembly will run faster than in Python. Programming languages, hardware and memories are external factors. Complexity is about the algorithm itself, the way it processes the data to solve a given problem. It's a software design concern at the "idea level".

It's possible to have an inefficient algorithm that's executed on high-end hardware to give a result quickly. However, with large input datasets, the limitations of the hardware will become apparent. Thus, it's desirable to optimize the algorithm first before thinking about hardware upgrades.

Suppose your computer can process 10,000 operations/sec. An algorithm of order O(n4)O(n4) would take 1 sec to process 10 items but more than 3 years to process 1,000 items. Comparatively, a more efficient algorithm of order O(n2)O(n2) would take only 100 secs for 1,000 items. With even larger inputs, better hardware cannot compensate for algorithmic inefficiency. It's for this reason algorithmic complexity is defined in terms of asymptotic behaviour.

* Are there techniques to figure out the complexity of algorithms?

Instead of looking for exact execution times, we should evaluate the number of high-level instructions in relation to the input size.

A single loop that iterates through the input is linear. If there's a loop within a loop, with each loop iterating through the input, then the algorithm is quadratic. It doesn't matter if the loops process only alternative items or skip a fixed number of items. Let's recall that complexity ignores constants and scaling factors. Likewise, a loop within a loop, followed by another loop, is quadratic, since we need to consider only the dominant term.

A recursive function that calls itself n times is linear, provided other operations within the function don't depend on input size. However, a recursive implementation of Fibonacci series is exponential.

A search algorithm that partitions the input into two parts and discards one of them at each iteration, is logarithmic. An algorithm such as Mergesort that partitions the input into halves at each iteration, plus does a merge operation in linear time at each iteration, has a log-linear complexity.

* If an algorithm is inefficient, does that mean that we can't use it?

Polynomial complexity algorithms of order O(nc)O(nc), for c > 1, may be acceptable. They can be used for inputs up to thousands of items. Anything exponential can probably work for only inputs less than 20.

Algorithms such as Quicksort that have complexity of O(n2)O(n2) rarely experience worst-case inputs and often obey Θ(nlogn)Θ(nlogn) in practice. In some case, we can preprocess the input so that worst-case scenarios don't occur. Likewise, we can go with sub-optimal solutions so that complexity is reduced to polynomial time.

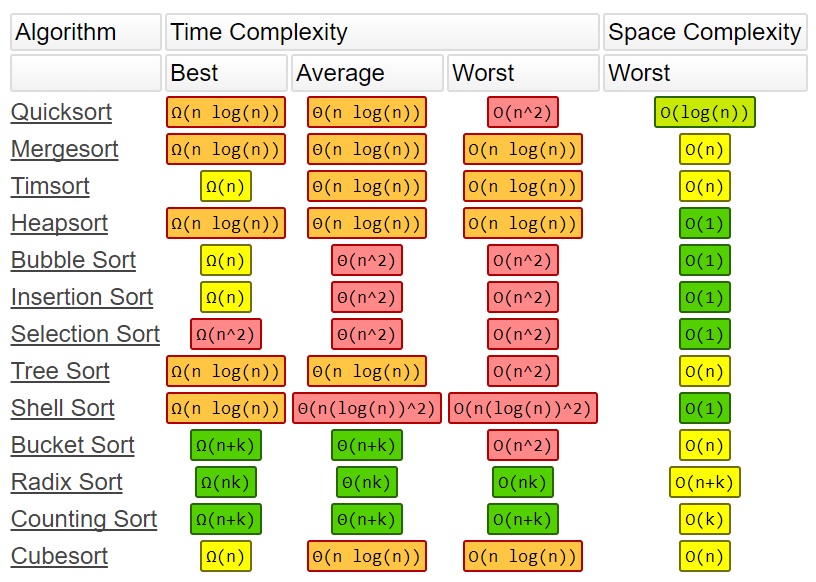
In practice, a linear algorithm can perform worse than a quadratic one if large constants are involved and n is comparable to these constants. It's also important to analyze every operation of an algorithm to ensure that non-trivial operations are not hidden or abstracted away within libraries.

* Since algorithmic complexity is about algorithms, is it relevant to talk about data structures?

**Complexity of operations on data structures. Source: Big-O Cheat Sheet, 2016.**

Data structures only store data but the algorithmic complexity comes into consideration when we operate on them. Operations such as insertion, deletion, searching and indexing need to be analyzed. The intent is to choose the right data structures so that complexity is reduced.

For example, accessing an array by index has constant complexity whereas the same operation with a linked list has linear complexity. Searching by key in a hash table incurs constant average complexity but this operation is linear with stacks, queues, arrays and linked list. A more detailed discussion on [what data structures to use](http://www.introprogramming.info/english-intro-csharp-book/read-online/chapter-19-data-structures-and-algorithm-complexity/) is given by Svetlin Nokav and others.

* Can you state the complexity of well-known sorting algorithms?

**Complexity of sorting algorithms. Source: Big-O Cheat Sheet, 2016.**

**Hash Table Explained: What it Is and How to Implement It**



A hash table, also known as a hash map, is a data structure that maps keys to values. It is one part of a technique called hashing, the other of which is a hash function. A hash function is an algorithm that produces an index of where a value can be found or stored in the hash table.

Some important notes about hash tables:

1. Values are not stored in a sorted order.
2. You mush account for potential collisions. This is usually done with a technique called chaining. Chaining means to create a linked list of values, the keys of which map to a certain index.

**Implementation of a hash table**

The basic idea behind hashing is to distribute key/value pairs across an array of placeholders or "buckets" in the hash table.

A hash table is typically an array of linked lists. When you want to insert a key/value pair, you first need to use the hash function to map the key to an index in the hash table. Given a key, the hash function can suggest an index where the value can be found or stored:

index = f(key, array\_size)

This is often done in two steps:

hash = hashfunc(key)

index = hash % array\_size

Using this method, hash is independent of the size of the hash table. hash is reduced to an index – a number between 0, the start of the array, and array\_size - 1, the end of the array – using the modulo (%) operator.

Consider the following string, S:

string S = “ababcd”

You need to count the frequency of all the characters in S. The easiest way to do this is to iterate through all the possible characters and count the frequency of each, one by one.

This works, but it's slow – the time complexity of such an approach is O(26\*N), with N being the size of the string S multiplied by 26 possible characters from A-Z.

void countFre(string S)

{

for(char c = ‘a’;c <= ‘z’;++c)

{

int frequency = 0;

for(int i = 0;i < S.length();++i)

if(S[i] == c)

frequency++;

cout << c << ‘ ‘ << frequency << endl;

}

}

**Output:**

a 2

b 2

c 1

d 1

e 0

f 0

…

z 0

Let's take a look at a solution that uses hashing.

Take an array and use the hash function to hash the 26 possible characters with indices of the array. Then iterate over S and increase the value of the current character of the string with the corresponding index for each character.

The complexity of this hashing approach is O(N), where N is the size of the string.

int Frequency[26];

int hashFunc(char c)

{

return (c - ‘a’);

}

void countFre(string S)

{

for(int i = 0;i < S.length();++i)

{

int index = hashFunc(S[i]);

Frequency[index]++;

}

for(int i = 0;i < 26;++i)

cout << (char)(i+’a’) << ‘ ‘ << Frequency[i] << endl;

}

Output

a 2

b 2

c 1

d 1

e 0

f 0

…

z 0

**Hash Collisions**

Since your hash map will probably be significantly smaller than the amount of data you're processing, hash collisions are unavoidable. There are two main approaches to handling collisions: *chaining* and *open addressing*.

**Chaining**

As mentioned earlier, chaining means that each key/value pair in the hash table, the value is a linked list of data rather than a single cell.

For example, imagine that the key 152 holds the value "John Smith". If the value "Sandra Dee" is added to the same key, "Sandra Dee" is added as another element to key 152, just after "John Smith".

152: [["John Smith", "p01"]]

...

152: [["John Smith", "p01"] ["Sandra Dee", "p02"]]

The main drawback of chaining is the increase in time complexity. Instead of 0(1) as with a regular hash table, each lookup will take more time since we need to traverse each linked list to find the correct value.

**Open addressing**

Open addressing means that, once a value is mapped to a key that's already occupied, you move along the keys of the hash table until you find one that's empty. For example, if "John Smith" was mapped to 152, "Sandra Dee" will be mapped to the next open index:

152: ["John Smith", "p01"]

...

152: ["John Smith", "p01"],

153: ["Sandra Dee", "p02"]

The main drawback to open addressing is that, when you look up values, they might not be at the key map you expect them at. Instead, you have to traverse different parts of the hash table to find the value you're looking for.

The simplest sorting algorithm is probably Bubblesort but it's quadratic in the average case and hence not efficient. Better alternatives are those with log-linear complexity: Quicksort, Mergesort, Heapsort, etc. If the list is already sorted, the best-case complexity occurs with Bubblesort, Timsort, Insertionsort and Cubesort, all completing in linear time.

It's been noted that best and worst cases rarely occur. The average case is based on an input distribution model that could be a random sample as well. Analysis of these averages or abstract basic operations can help us pick the best suited algorithm for a specific problem.

* Can you give the complexity of some important algorithms?

Here's a selection:

* + Fast Fourier Transform: O(nlogn)O(nlogn)
  + Multiply two n-digit numbers using Karatsuba algorithm: O(n1.59)O(n1.59)
  + Matrix multiplication due to Coppersmith and Winograd: O(n2.496)O(n2.496)
  + Prime recognition of an n-digit integer due to Adleman, Pomerance and Rumley: nO(loglogn)nO(loglogn)
  + Gaussian elimination: O(n3)O(n3) but O(x2n)O(x2n) bit complexity of its operands
  + GCD(a, b) by Euclid's algorithm: O(log(a+b))O(log(a+b)) but O((log(a+b))2)O((log(a+b))2) bit complexity when integers a and b are large

Bit complexity matters when integers exceed the 64-bit machine capability. In these cases, when arbitrary precision is used, operations such as division and modulo arithmetic are no longer trivial and their complexity must be accounted for. From a theoretical perspective, this doesn't matter to computer scientists but it matters to programmers who have to work within machine limits.

Milestones

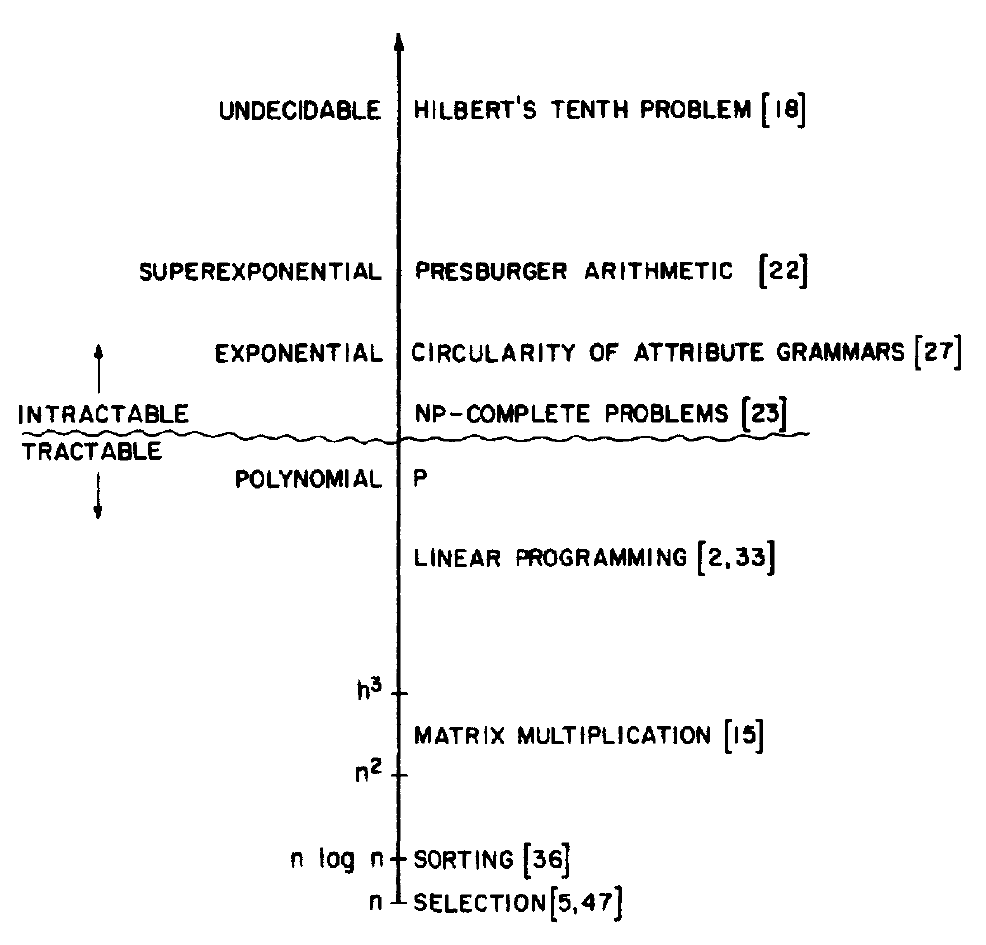
**1864**

Charles Babbage, while building his Analytical Engine, predicts the importance of studying the performance of algorithms.

**1894**

Mathematician Paul Bachmann defines the **Big-O notation**. It originally meant "order of". Decades later, Donald Knuth calls it the capital omicron.

**1960**



**Spectrum of computational complexity. Source: Tarjan 1983, fig. 1.3.**

The **theory of computational complexity** is born in the 1960s. This is also the decade when problems that can't be solved in polynomial time (NP problems) are recognized.

**1975**

Tarjan makes the case that when designing data structures, focusing on worst case running time is less important than **amortized running time**, that is, performance averaged over operations on a long sequence of input. His later contributions include the splay tree (1980) and the Fibonacci heap, both achieving good amortized performance.

**1976**

Donald Knuth introduces new **notations** and clarifies the meaning of OO, ΩΩ and ΘΘ.

**1998**

R. Libeskin-Hadas introduces the term **oblivious algorithm** that's applied to an algorithm whose complexity is independent of the input structure. In other words, the best-case, worst-case and average complexity of the algorithm are all the same.

Sample Code

* [C](https://devopedia.org/algorithmic-complexity)
* *// Source: Nakov et al., 2013, ch. 19.*
* *// Linear complexity: O(N\*N)*
* int FindMaxElement(int[] array)
* {
* int max = int.MinValue;
* for (int i = 1; i < array.Length; i++)
* {
* if (array[i] > max)
* {
* max = array[i];
* }
* }
* return max;
* }

* *// Quadratic complexity: O(N\*N)*
* int FindInversions(int[] array)
* {
* int inversions = 0;
* for (int i = 0; i < array.Length - 1; i++)
* {
* for (int j = i + 1; j < array.Length; j++)
* {
* if (array[i] > array[j])
* {
* inversions++;
* }
* }
* }
* return inversions;
* }

* *// Quadratic complexity: O(N\*M)*
* *// Innermost loop executes N\*min(N,M); thus, this algorithm is not of cubic complexity.*
* long SumMN(int n, int m)
* {
* long sum = 0;
* for (int x = 1; x <= n; x++)
* {
* for (int y = 1; y <= m; y++)
* {
* if (x == y)
* {
* for (int i = 1; i <= n; i++)
* {
* sum += i \* x \* y;
* }
* }
* }
* }
* return sum;
* }

* *// Exponential complexity: O(2^n)*
* long Fibonacci(int n)
* {
* if (n == 0)
* {
* return 1;
* }
* else if (n == 1)
* {
* return 1;
* }
* else
* {
* return Fibonacci(n - 1) + Fibonacci(n - 2);
* }
* }