In [1]:

```
%%html
<style>
    p,ul {

          font-size: 18px;
          font-family: "Segoe Print","Georgia", "Verdana", "Lucida Console", "Courier N
ew";
          line-height: 150%;
    }

    .text_cell_render p {
    text-align: justify;
    text-justify: inter-word;
    }
</style>
```

In [2]:

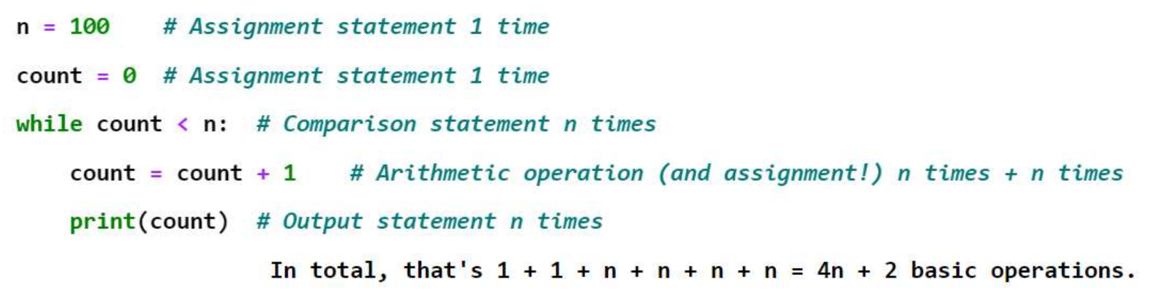
```
from IPython.display import Image
# Image(filename = "tsc00.png", width = "100%")
```

- There are two measures of efficiency:
  - **Time Complexity**: the time taken by an algorithm to execute.
  - **Space Complexity**: the amount of memory used by an algorithm while executing.
- Time complexity is often considered more important, but space considerations are sometimes relevant too.
- The technique for calculating time complexity is to add up how many basic operations an algorithm will execute as a function of the size of its input, and then simplify this expression. Basic operations include things like
  - Declarations
  - Assignments
  - Arithmetic Operations
  - Comparison statements
  - Calling a function
  - Return statements
- One way to count the basic operations is:

In [3]:

```
Image(filename = "tsc00.png", width = "100%")
```

Out[3]:



```
n = 100     # Assignment statement 1 time
count = 0  # Assignment statement 1 time
while count < n:  # Comparison statement n times
    count = count + 1    # Arithmetic operation (and assignment!) n times + n times
    print(count)  # Output statement n times
```

In total, that's 1 + 1 + n + n + n + n = 4n + 2 basic operations.

In [2]:

```
def sumOfN(n):

    theSum = 0

    for i in range(1,n+1):

        theSum = theSum + i

    return theSum
```

In [5]:

```
%timeit sumOfN(10000)
```

```
393 µs ± 2.68 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [4]:

```
%timeit sumOfN(100000)
```

```
4.15 ms ± 18.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [7]:

```
def sumOfN2(n):

    theSum = (n*(n+1))/2

    return theSum
```

In [8]:

```
%timeit sumOfN2(10000)
```

```
140 ns ± 0.384 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops eac
h)
```

In [9]:

```
%timeit sumOfN2(100000)
```

```
142 ns ± 0.652 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops eac
h)
```

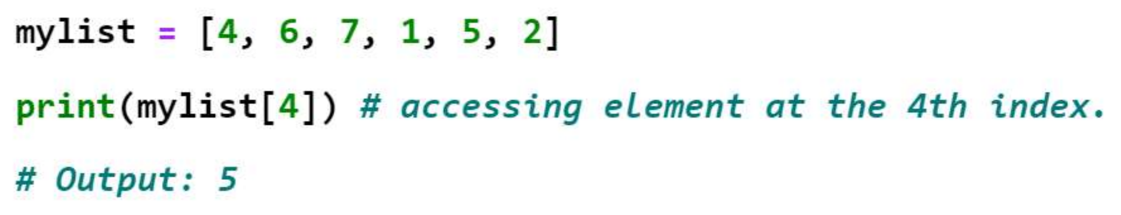
## Time Complexity Examples

### O(1) – Constant Time Complexity

In [4]:

```
Image(filename = "tsc0a.png", width = "100%")
```

Out[4]:



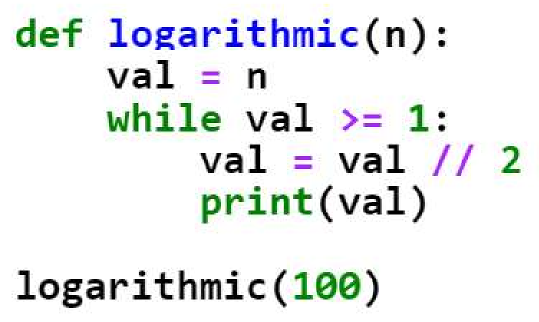
```
mylist = [4, 6, 7, 1, 5, 2]

print(mylist[4]) # accessing element at the 4th index.

# Output: 5
```

### O(log n) – Logarithm Time Complexity

In [8]:

```
Image(filename = "tsc0b.png", width = "50%")
```

Out[8]:



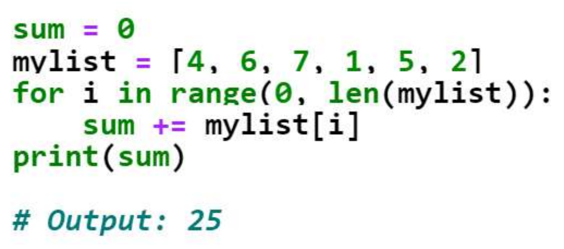
```
def logarithmic(n):
    val = n
    while val >= 1:
        val = val // 2
        print(val)

logarithmic(100)
```

### O(n) – Linear Time Complexity

In [9]:

```
Image(filename = "tsc0c.png", width = "50%")
```

Out[9]:



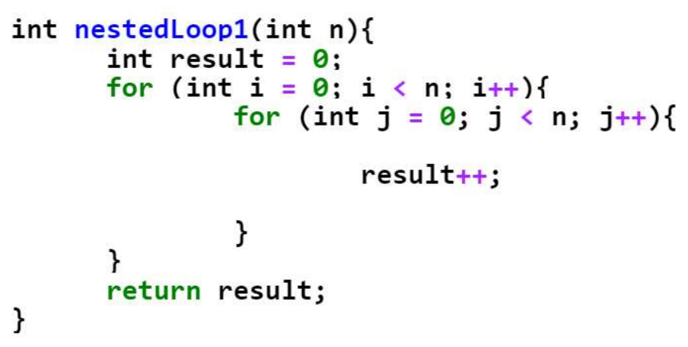
```
sum = 0
mylist = [4, 6, 7, 1, 5, 2]
for i in range(0, len(mylist)):
    sum += mylist[i]
print(sum)

# Output: 25
```

## O($n^2$) – Quadratic Time Complexity

In [11]:

```
Image(filename = "tsc1.png", width = "60%")
```

Out[11]:



```
int nestedLoop1(int n){
        int result = 0;
        for (int i = 0; i < n; i++){
                for (int j = 0; j < n; j++){

                        result++;

                }
        }
        return result;
}
```

In [6]:

```
mat = [[1, 2, 3], [1, 1, 1], [5, 7, 8]]
sum = 0
for i in range(len(mat)):
    for j in range(len(mat[0])):
        sum += mat[i][j]
print(sum)
```

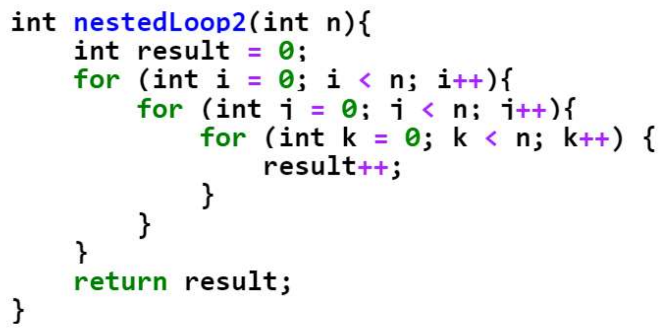
```
29
```

## O($n^k$) – Polynomial Time Complexity (k >=3)

In [13]:

```
Image(filename = "tsc2.png", width = "60%")
```

Out[13]:



```
int nestedLoop2(int n){
    int result = 0;
    for (int i = 0; i < n; i++){
        for (int j = 0; j < n; j++){
            for (int k = 0; k < n; k++) {
                result++;
            }
        }
    }
    return result;
}
```

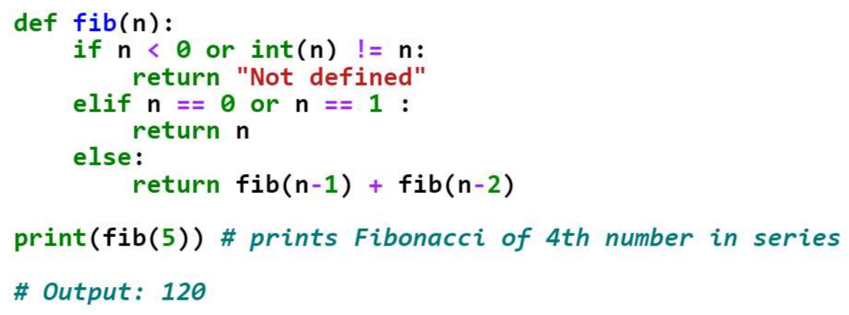
## O($2^n$) – Exponential Time Complexity

*In exponential time complexity, the running time of an algorithm doubles with the addition of each input data.*

In [16]:

```
Image(filename = "tsc3.png", width = "75%")
```

Out[16]:

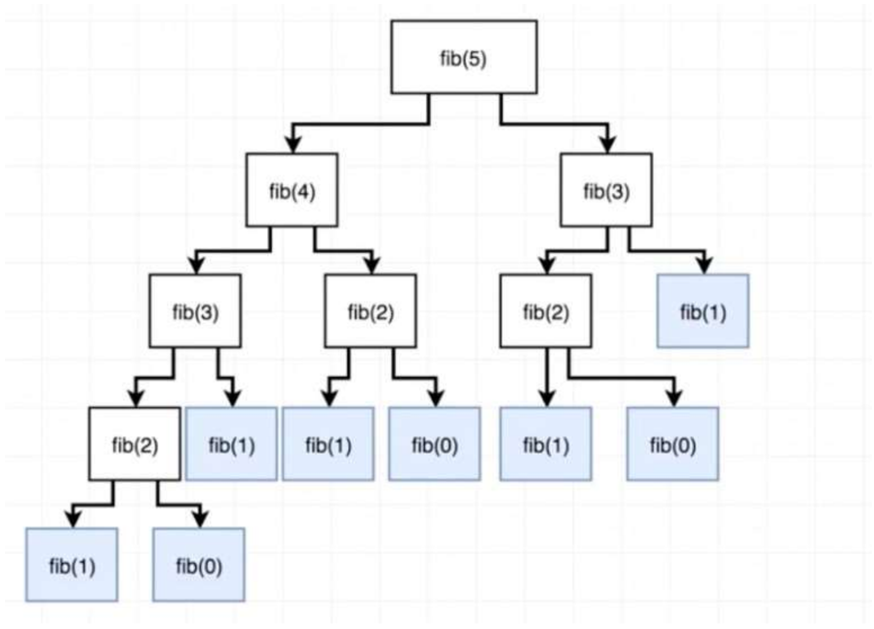


```
def fib(n):
    if n < 0 or int(n) != n:
        return "Not defined"
    elif n == 0 or n == 1 :
        return n
    else:
        return fib(n-1) + fib(n-2)

print(fib(5)) # prints Fibonacci of 4th number in series

# Output: 120
```

In [17]:

```
Image(filename = "exponential.jpg", width = "75%")
```

Out[17]:



## O(n!) – Factorial Time Complexity

### The Traveling Salesman Problem

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? The brute force method would be to check every possible configuration between each city, which would be a factorial, and quickly get crazy!

Say we have 3 cities: A, B and C.

How many permutations are there? Permutations is a maths term meaning how many ways can we order a set of items.

A -> B -> C -> A

A -> C -> B -> A

B -> A -> C -> B

B -> C -> A -> B

C -> A -> B -> C

C -> B -> A -> C

With 3 cities, we have 3! permutations. That's 1 x 2 x 3 = 6 permutations.

Why? If we have three possible starting points, then for each starting point, we have two possible routes to the final destination.

What if our salesman needs to visit 4 cities? A, B, C and D.

We'd have 4! = 4 x 3 x 2 x 1 = 24 permutations.

Why? If we have four possible starting points, then for each starting point, we have three possible routes, and for each of those points we have two possible routes, and the final stop - So:

4 3 2 * 1

In [4]:

```
def perms(a_str):
    stack = list(a_str)
    results = [stack.pop()]
    # print(results)
    while stack:
        current = stack.pop()
        new_results = []
        for partial in results:
            for i in range(len(partial)+1):
                new_results.append(partial[:i] + current + partial[i:])
        results = new_results
    return results

my_str = "ABCD"
print(perms(my_str))
```

```
['ABCD', 'BACD', 'BCAD', 'BCDA', 'ACBD', 'CABD', 'CBAD', 'CBDA', 'ACDB',
'CADB', 'CDAB', 'CDBA', 'ABDC', 'BADC', 'BDAC', 'BDCA', 'ADBC', 'DABC', 'D
BAC', 'DBCA', 'ADCB', 'DACB', 'DCAB', 'DCBA']
```

In [ ]:

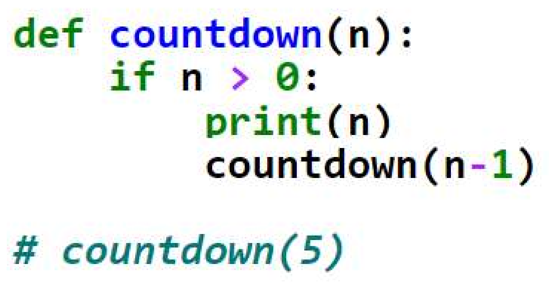
```
n = 3
for i in range(n):
    for j in range(n):
        print(f"i: {i}, j: {j}")
```

## Time Complexity of Recursive Algorithms

In [19]:

```
Image(filename = "tsc5.png", width = "50%")
```

Out[19]:

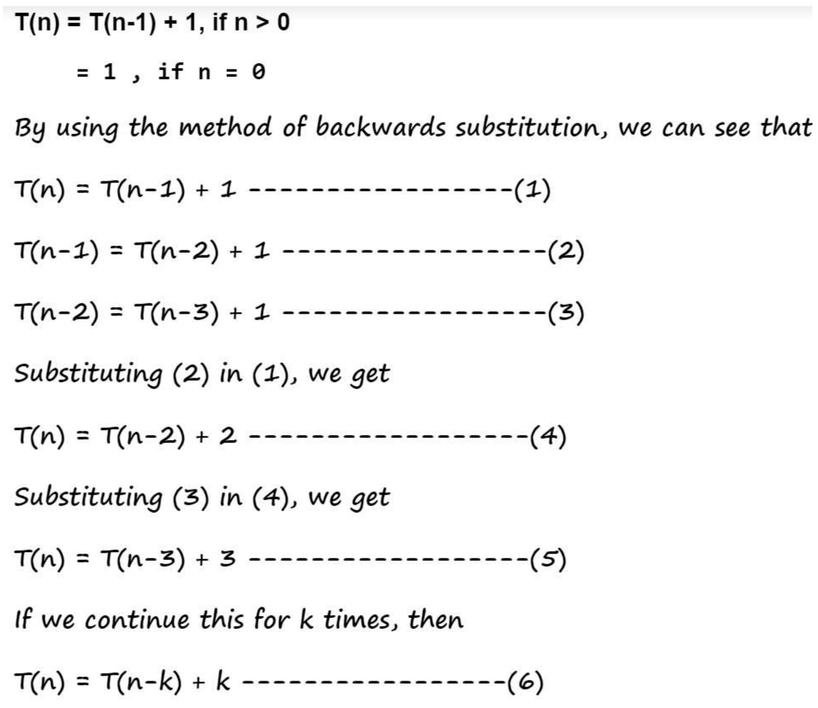


```
def countdown(n):
    if n > 0:
        print(n)
        countdown(n-1)

# countdown(5)
```

In [21]:

```
Image(filename = "tsc6.png", width = "70%")
```

Out[21]:



T(n) = T(n-1) + 1, if n > 0

```
     = 1 , if n = 0
```

By using the method of backwards substitution, we can see that

T(n) = T(n-1) + 1 -----------------(1)

T(n-1) = T(n-2) + 1 -----------------(2)

T(n-2) = T(n-3) + 1 -----------------(3)

Substituting (2) in (1), we get

T(n) = T(n-2) + 2 ------------------(4)

Substituting (3) in (4), we get

T(n) = T(n-3) + 3 ------------------(5)

If we continue this for k times, then

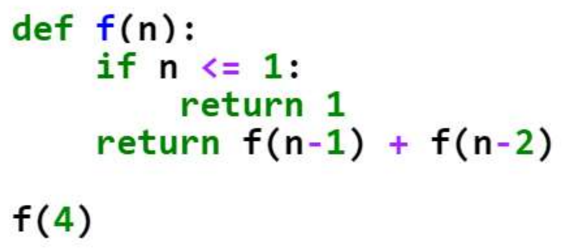
T(n) = T(n-k) + k -----------------(6)

## Measuring the Recursive Algorithm that takes Multiple Calls

In [23]:

```
Image(filename = "tsc7.png", width = "50%")
```

Out[23]:



```
def f(n):
    if n <= 1:
        return 1
    return f(n-1) + f(n-2)

f(4)
```

*Consider each recursive call as a node of a binary tree.*

*When multiple recursive calls are made, we can represent time complexity as $O(branches^{depth})$. Here, branches represents number of children for each node i.e. number of recursive call in each iteration and depth represents parameter in the recursive function.*

## Python Example of O(1) Space Complexity

In [5]:

```
def my_sum(lst):
    total = 0
    for i in range(len(lst)):
        total += lst[i]
    return total

my_list = [5, 4, 3, 2, 1]
```

## Python Example of O(n) Space Complexity

In [6]:

```
def double(lst):
    new_list = []
    for i in range(len(lst)):
        new_list.append(lst[i] * 2)
    return new_list


my_list = [5, 4, 3, 2, 1]
print(double(my_list))
```

```
[10, 8, 6, 4, 2]
```