Recap

- Linked List
    - Single
    - Double
    - Circular
- Data and Address/Pointer of Next node/element
- data of 5th/ 5 and None
- Head & Tail

Single Linked List

In [20]:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next_node = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.num_of_nodes = 0
    def append_start(self, data):

        self.num_of_nodes += 1

        # pointer of every node
        new_node = Node(data)

        # no elements in linked list
        if not self.head:
            self.head = new_node

        # add 1 ele with none and self.head 1st ele address
        # add 2 ele add 1 ele address 2nd ele then 2 nd none and data of
        # atleast 1 item in linked list
        else:
            new_node.next_node = self.head
            self.head = new_node

    def size(self):
        return self.num_of_nodes

    def traverse(self):
        present_node = self.head
        while present_node is not None:
            print(present_node.data)
            present_node = present_node.next_node

    def append_end(self, data):
        self.num_of_nodes += 1
        new_node = Node(data)
        present_node = self.head
#       1 2 3 ....6
        while present_node.next_node is not None:
            present_node = present_node.next_node
        present_node.next_node = new_node
```

In [26]:

```
ll = LinkedList()
```

In [13]:
```
ll.append_start(1)
ll.append_start(2)
ll.append_start(3)
ll.append_start(4)
ll.append_start(5)
ll.append_start(6)

ll.size()
```

Out[13]: 6

In [14]:
```
ll.traverse()
```

```
6
5
4
3
2
1
```

In [ ]:
```
[6 addr5 4 addr3 3 addr2 2 addr1 1 None]
```

In [ ]:
```
li = [4 add6, 6 add5, 5 none]
```

In [ ]:
```
self.head = addr1

[1 addr3, 3 addr4, 4 addr5, 5 add6, 6 None]

next add to previous ele
break
2
2 addr3

1
2
3
4
5
6
stop
```

In [ ]:

```
self.head = addr1
7 addr7
[1 addr3, 3 addr4, 4 addr5, 5 add6, 6 None]

1
2
3
4
5
6

addr7 to 6th element
7th element none

[1 addr2, 2 addr3, 3 addr4, 4 addr5, 5 add6, 6 addr7, 7]
```

In [27]:

```python
ll.append_start(1)
ll.append_start(2)
ll.append_start(3)
ll.append_start(4)
ll.append_start(5)
ll.append_start(6)

print(ll.size())
ll.traverse()
print('---------------------')
ll.append_end(1)
ll.append_end(2)
ll.append_end(3)
ll.append_end(4)
ll.append_end(5)
ll.append_end(6)

print(ll.size())
ll.traverse()
```

```
6
6
5
4
3
2
1
---------------------
12
6
5
4
3
2
1
1
2
3
4
5
6
```

Time Complexity

```
A person Application 5min
B person Application 3min
C person Application <0.5 min

A & B try to application <0.1 min
```

- Big O Notation -> 5 sec Worst Case Scenario/ rare scenario -> less than worsty case runtime
- Omega Notation -> 1 sec Best Case scenario/ > 1 sec
- Theta Notation -> average-case complexity -> Between 1 and 5

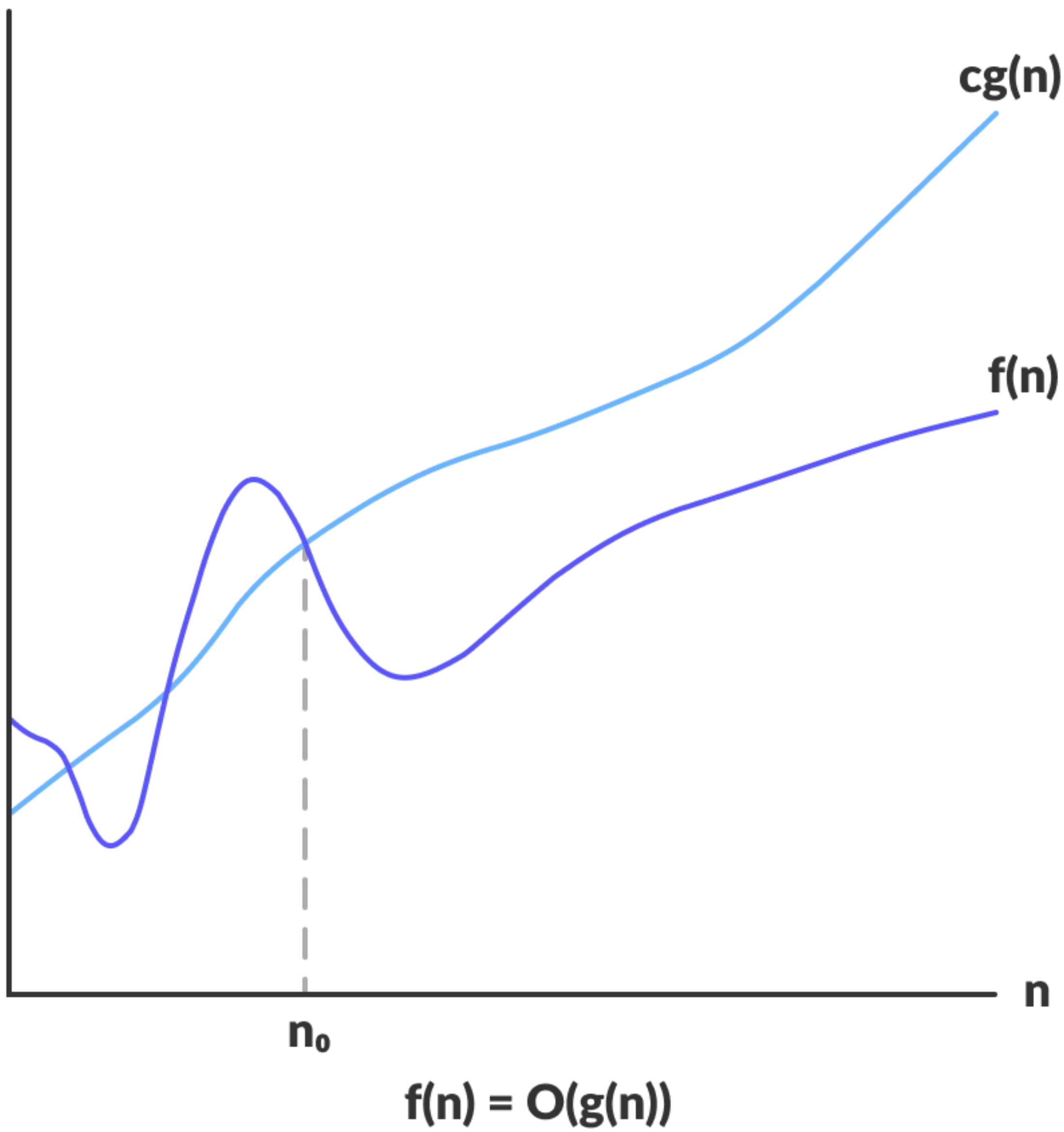
c1*g(n) < f(n)

c2 * g(n) > f(n)

c1g(n) < f(n) < c2f(n)

Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.



Big-O gives the upper bound of a function

```
O(g(n)) = { f(n): there exist positive constants c and n0
            such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }
```

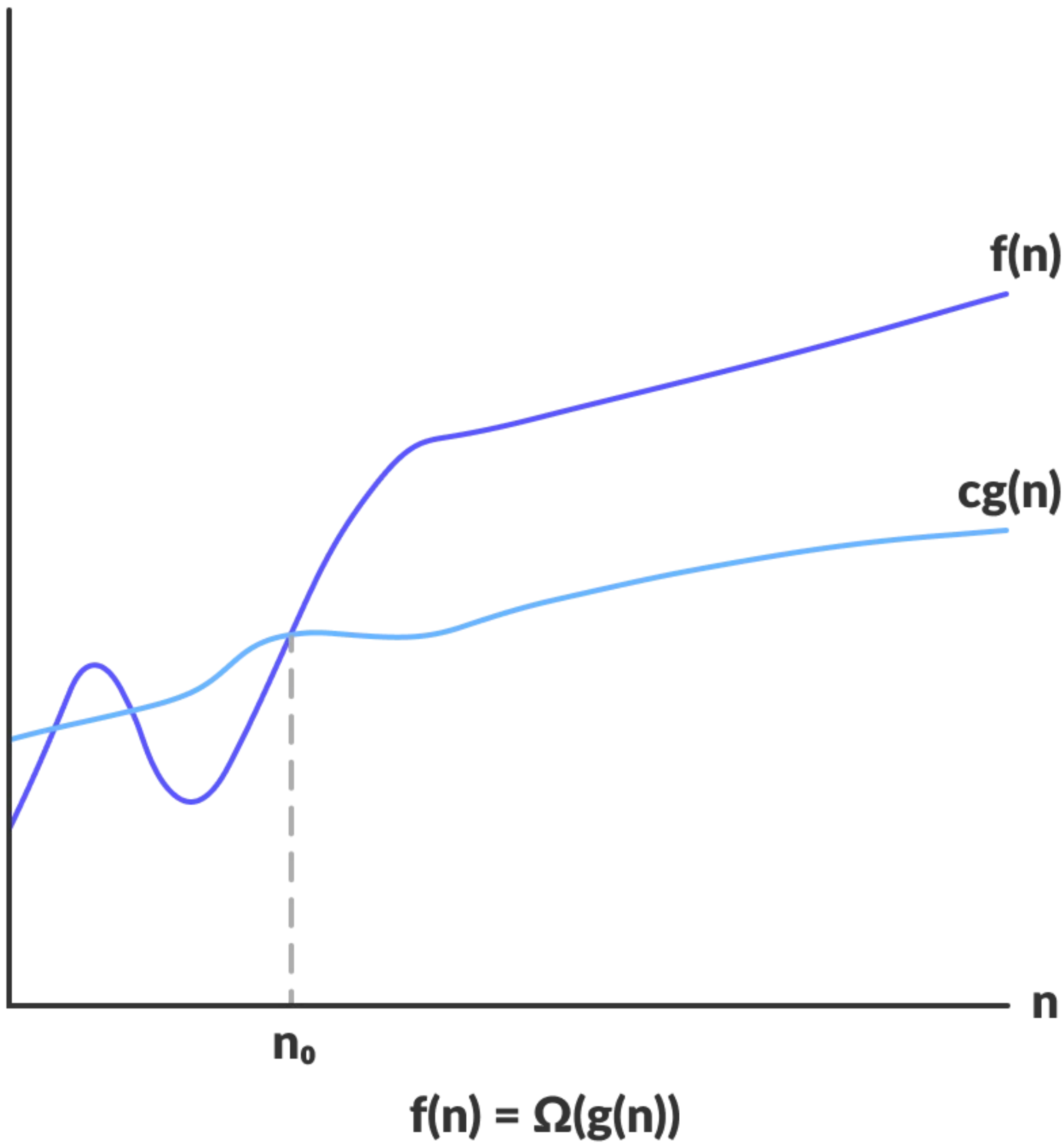
The above expression can be described as a function `f(n)` belongs to the set `O(g(n))` if there exists a positive constant `c` such that it lies between `0` and `cg(n)`, for sufficiently large `n`.

For any value of `n`, the running time of an algorithm does not cross the time provided by `O(g(n))`.

Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

Omega Notation (Ω-notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.



Omega gives the lower bound of a function

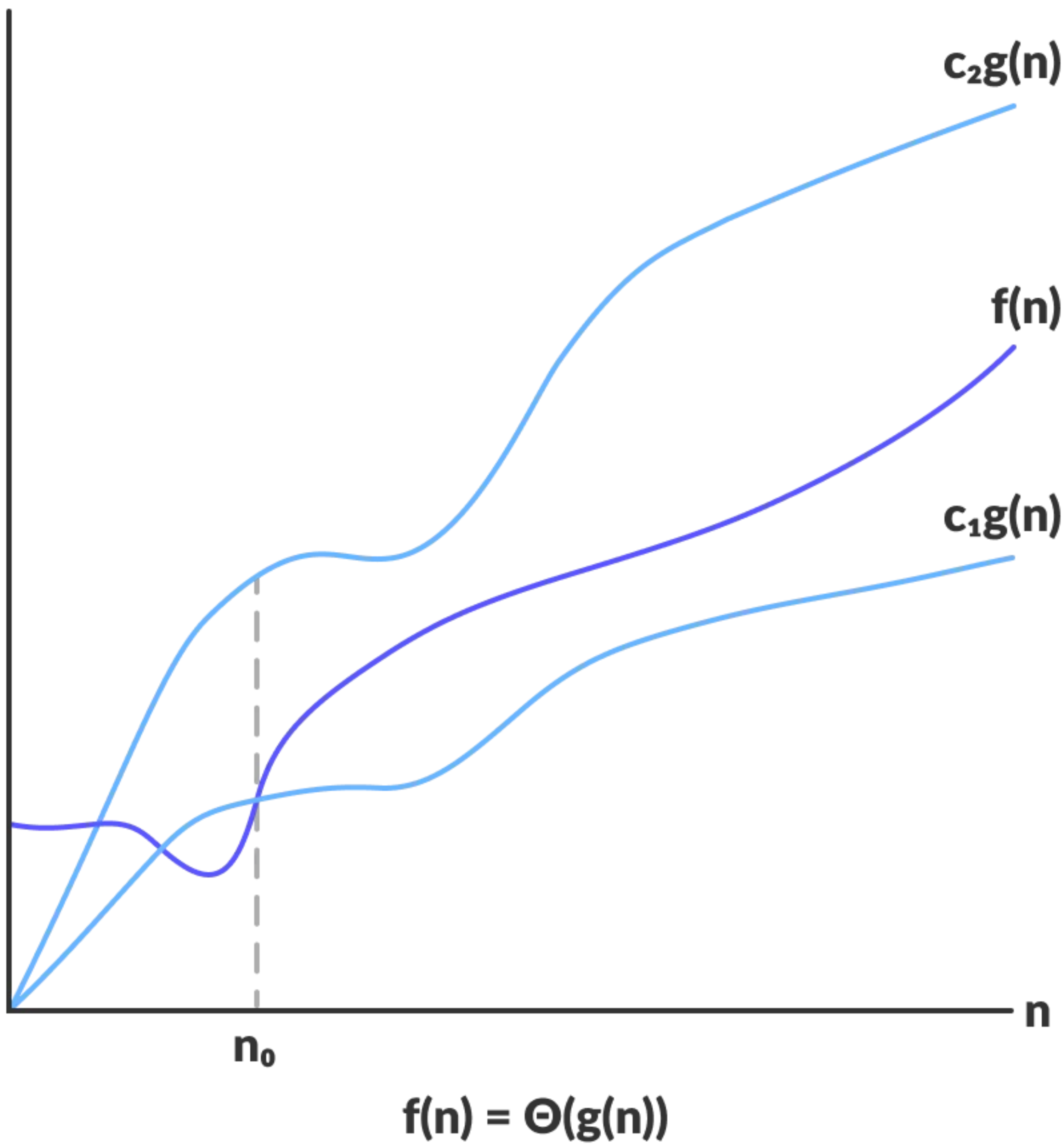
```
Ω(g(n)) = { f(n): there exist positive constants c and n0
            such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }
```

The above expression can be described as a function `f(n)` belongs to the set `Ω(g(n))` if there exists a positive constant `c` such that it lies above `cg(n)`, for sufficiently large `n`.

For any value of `n`, the minimum time required by the algorithm is given by Omega `Ω(g(n))`.

Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.



Theta bounds the function within constants factors

For a function `g(n)` , `Θ(g(n))` is given by the relation:

```
Θ(g(n)) = { f(n): there exist positive constants c1, c2 and n0
            such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0 }
```

The above expression can be described as a function `f(n)` belongs to the set `Θ(g(n))` if there exist positive constants `c1` and `c2` such that it can be sandwiched between `c1g(n)` and `c2g(n)` , for sufficiently large n.

If a function `f(n)` lies anywhere in between `c1g(n)` and `c2g(n)` for all `n ≥ n0` , then `f(n)` is said to be asymptotically tight bound.

Types of Time Complexity Functions

- Constant Time Complexity: O(1)
- Linear Time Complexity: O(n)
- Logarithmic Time Complexity: O(log n)
- Quadratic Time Complexity: O(n²)
- Exponential Time Complexity: O(2^n)

memory/processor is constant

In [29]:
```
li = [1, 2, 3, 4, 5]

# 1st time -> 1 sec
print(li[2]) #O(1)
```

```
3
```

In [ ]:
```
for i in li:
    print(i) # O(N)
```

In [30]:
```
# Logarithmic Time Complexity: O(log n)
li = [1, 2, 3, 4, 5]

for i in range(0, len(li), 2): # O(log n)
    print(li[i], i)
```

```
1 0
3 2
5 4
```

In [35]:
```
# Quadratic Time Complexity: O(n²)
mat = [[1,2,3],
       [4,5,6]]

#  [5, 6, 7], [6, 7, 8], ...
# O(N^2)
for i in range(len(mat)):
    for j in range(len(mat[i])):
        print(mat[i][j] + mat[i][j], end = '\t')
    print()
```

```
2	4	6
8	10	12
```

In [ ]:
```
# Exponential Time Complexity: O(2^n)

2 sec
4 sec
8 sec
16 sec
```

In [ ]:

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


print(li[2]) -> o(1)
5
li.index(5) O(N)

```

5 Rules

No	Description	Complexity
Rule 1	Any assignment statements and if statements that are executed once regardless of the size of the problem	O(1)
Rule 2	A simple “for” loop from 0 to n (with no internal loops)	O(n)
Rule 3	A nested loop of the same type takes quadratic time complexity	O(n^2)
Rule 4	A loop, in which the controlling parameter is divided by two at each step	O(log n)
Rule 5	When dealing with multiple statements, just add them up	

In [ ]:

```
print(5) --> O(1)
for i range(5):
    print(i)   --> o(n)

for i in range(5):
    for j in range(5):
        print(i,  j)   --> o(n^2)
```