What is efficiency in programming?

Efficiency in programming refers to how effectively a program uses computational resources—primarily **time** and **memory**—to perform its tasks.

Only working of code as expected doesnot makes program efficient. It must executed faster and in less space.

Efficiency of two program are compared based on time and space.
Time complexity => how much time program takes to execute in RAM.
Space Complexity => how much space program takes in RAM during execution.

Why efficiency is important?

1. Efficient programs run faster while a slow program can lead to poor user experience or system failure.
2. Efficient programs use less memory, CPU power due to lesser time in code execution, and battery
3. Efficient code can handle larger data and more users without breaking or slowing down.

   📌 Example: A search engine like google must efficiently handle millions of queries per second. E.g user search kathmandu in google, then billions of webpage that contains kathmandu are filtered out and sorted based on user preferences. It uses pagerank algorithm.

4. Faster and more efficient software gives companies a competitive edge by delivering better performance, reducing latency, and improving customer satisfaction.
5. Efficient programs are less likely to:
   a. Crash due to memory overuse
   b. Time out or freeze
   c. Overload the system

Types of efficiency

1. Space Efficiency
2. Time Efficiency

Techniques to measure time efficiency

1. Measuring **time** to execute
2. **Counting** operations involved
3. Abstract notion of **order of growth**

Note : Approach 1 and 2 is not best approach and not used in industry.

1. Measuring Time

```
import time
start = time.time()
# creating Delay
for i in range(1,100000000) :
  pass
end = time.time()
print("Time required to execute for loop : {}
second".format(end-start))
```

`time.time()` returns the current time in seconds (as a float) since January 1, 1970 (the Unix Epoch).

Problems with this approach

Different time for different algorithm	✔
Time varies if implementation changes(eg: using same algorithm but used while loop instead of food loop.)	✖
Different machines different time	✖
Does not work for extremely small input (e.g : measuring time of assigning value to variables, print statement) i.e Required less time and become difficult for us to understand.	✖
Time varies for different inputs, but can't establish a relationship between input and time taken by program for different algorithm.	✖

2. Counting Operations

COUNTING OPERATIONS

- assume these steps take **constant time**:
  - mathematical operations
  - comparisons
  - assignments
  - accessing objects in memory
- then count the number of operations executed as function of size of input

```
def c_to_f(c):
    return c*9.0/5 + 32

def mysum(x):
    total = 0
    for i in range(x+1):
        total += i
    return total
```

3 ops

1 op

loop x times

1 op

2 ops

mysum → 1+3x ops

X represent input.

Problems with this approach

Different time for different algorithm(different algorithm => different no. of operation)	✔
Time varies if implementation changes(using different loop for same algorithm changes no. of operations)	✖
Different machines different time(different machine => same no. of operation and same time)	✔
No clear definition of which operation to count(e.g : we have not count for,return,in,def etc in previous examples)	✖
Time varies for different inputs, but can't establish a relationship.(Relationship between input(x) and output i.e time to execute is established.)	✔

What do we want

1. We want to evaluate the algorithm
2. We want to evaluate scalability. Scalability means how the program behave for large input.
3. We want to evaluate in terms of input size i.e mathematical relationship between input and output i.e time taken by program to execute.

DIFFERENT INPUTS CHANGE HOW THE PROGRAM RUNS

- a function that searches for an element in a list

```
def search_for_elmt(L, e):
    for i in L:
        if i == e:
            return True
    return False
```

- when `e` is **first element** in the list → BEST CASE
- when `e` is **not in list** → WORST CASE
- when **look through about half** of the elements in list → AVERAGE CASE

3. Orders of Growth

Best Approach used in industry.

ORDERS OF GROWTH

Goals:

- want to evaluate program's efficiency when **input is very big**
- want to express the **growth of program's run time** as input size grows i.e establish mathematical relationship between input and time taken to execute.
- want to put an **upper bound** on growth – as tight as possible
  i.e want to work in worth case scenarios.
- do not need to be precise: **"order of" not "exact"** growth
- we will look at **largest factors** in run time (which section of the program will take the longest to run?)

Examples: for loop takes more time to execute than assignment, arithmetic operation etc. In this case : largest factors => loop

- **thus, generally we want tight upper bound on growth, as function of size of input, in worst case**

EXACT STEPS vs O()

```
def fact_iter(n):
    """assumes n an int >= 0"""
    answer = 1
    while n > 1:
        answer *= n
        n -= 1
    return answer
```

Execute only one time.

This while loop execute 5 operation for each iteration.

Execute only one time.

answer = answer * n

temp = n-1

n = temp

Count total no. of operation.

n = input size

- computes factorial
- number of steps: $1 + 5n + 1$
- worst case asymptotic complexity: $O(n)$
  - ignore additive constants
  - ignore multiplicative constants

Step 1 : count no. of operation to find out expression.

So the idea is simple

$n^2 + 2n + 2$

$n^2 + 100000n + 3^{1000}$

$\log(n) + n + 4$

$0.0001*n*\log(n) + 300n$

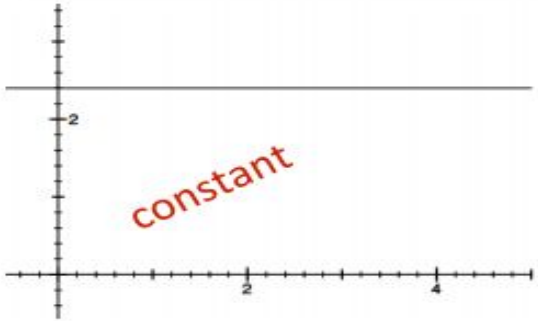
$2n^{30} + 3^n$

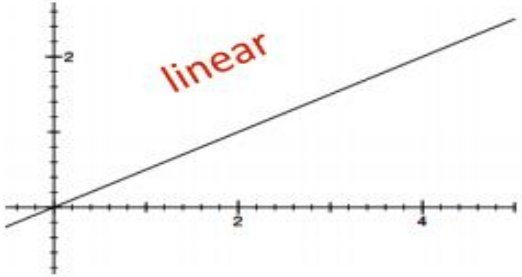
n2 + 2n + 2

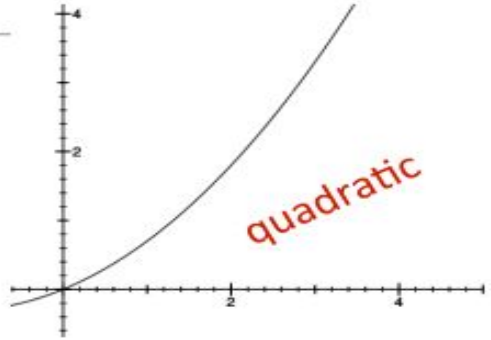
1. n2 + 2n # remove additive
2. n2 + n # remove multiplicative
3. # multiple factors => look at largest factors in runtime
   n2
4. Time complexity = O(n2)

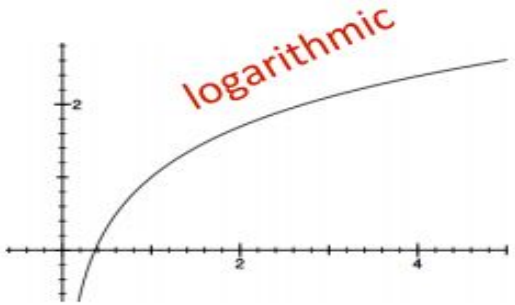
a. O(n2)
b. O(n2)
c. O(n)
d. O(nlog(n))
e. O(3^n)

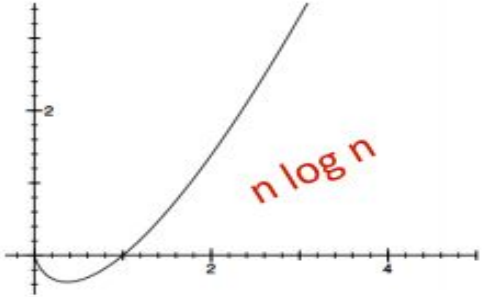
TYPES OF ORDERS OF GROWTH

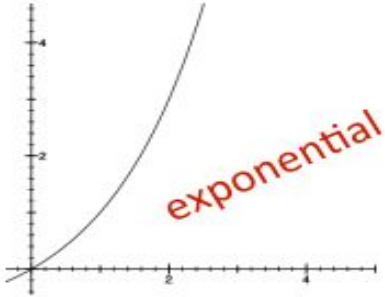












Constant Time Complexity : O(1)

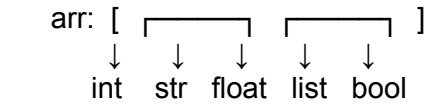
Time taken by program doesn't depend upon input size.

1. Python **lists store references (pointers)** to objects, not the actual objects themselves
2. Accessing element of list have O(1) access time or constant time i.e O(1) time complexity because of how list work at the memory level. [Doesn't depend upon no. of elements in list i.e access time remains same regardless whether it has 50 elements or 500 elements as :
3. arr[i] = base_address + i × size_of_pointer [Therefore, Accessing element doesn't depend upon size]
4. Each slot in the list is a fixed-size pointer (typically 8 bytes on a 64-bit system), regardless of the type or size of the object it points to.

Eg:

arr = [10, "hello", 3.14, [1, 2], True]

But in memory :



Property	Value
List stores	References (pointers)
Pointer size	Fixed (e.g., 8 bytes)

Operation	Time Complexity
Access `arr[i]`	O(1)
Append	Amortized O(1)
Insert/delete	O(n)
Search (unsorted)	O(n)

When working with List, Input size (n) = size of List

Linear Time Complexity : O(n)

Searching element in list is example of linear time access because element in list are searched one by one until search element is found.

In worst case, elements is searched till last element
50 items in list -> search till 49 index
100 items in list -> search till 99 index
i.e depend upon size of array.

So, Time complexity is O(n)

O(n) => When using single loop.

Quadratic Time Complexity : O(n^2)

O(n^2) => when using nested loop.

Logarithmic Time Complexity : O(log(n))

Examples : Binary Search

nlog(n) Time Complexity : O(nlog(n)

Examples : merge Sort

Exponential Time Complexity : O(a^n)

Risky and Dangerous. It need to avoid while writing program - With small rise in input size ,Time taken by program will be very high. In most of cases, for large input size , program cannot calculate due to very large time.

Law of addition

Law of Addition for O():

- used with **sequential** statements
- O(f(n)) + O(g(n)) is O(f(n) + g(n))
- for example,

```
for i in range(n):
    print('a')
for j in range(n*n):
    print('b')
```

O(n)

O(n*n)

O(n) + O(n*n)

is O(n) + O(n*n) = O(n+n^2) = O(n^2) because of dominant term

Law of multiplication

Law of Multiplication for O():

- used with **nested** statements/loops
- O(f(n)) * O(g(n)) is O(f(n) * g(n))
- for example,

```
for i in range(n):
    for j in range(n):
        print('a')
```

O(n)

n loops, each O(n) →
O(n)*O(n)

is O(n)*O(n) = O(n*n) = $O(n^2)$ because the outer loop goes n times and the inner loop goes n times for every outer loop iter.

Complexity Growth

CLASS	n=10	= 100	= 1000	= 1000000
O(1)	1	1	1	1
O(log n)	1	2	3	6
O(n)	10	100	1000	1000000
O(n log n)	10	200	3000	6000000
O(n^2)	100	10000	1000000	1000000000000
O(2^n)	1024	1267650600228229401496703205376	1071508607186267320948425049060001810561404811705533607443750388370351051124936122493198378815695858127594672917553146825187145285692314043598457757469857480393456777482423098542107460506237114187795418215304647498358194126739876755916554394607706291457119647768654216766042983165262438683720566806937 6	Good luck!!

Top to bottom -> bad code
Bottom to Top -> Good code