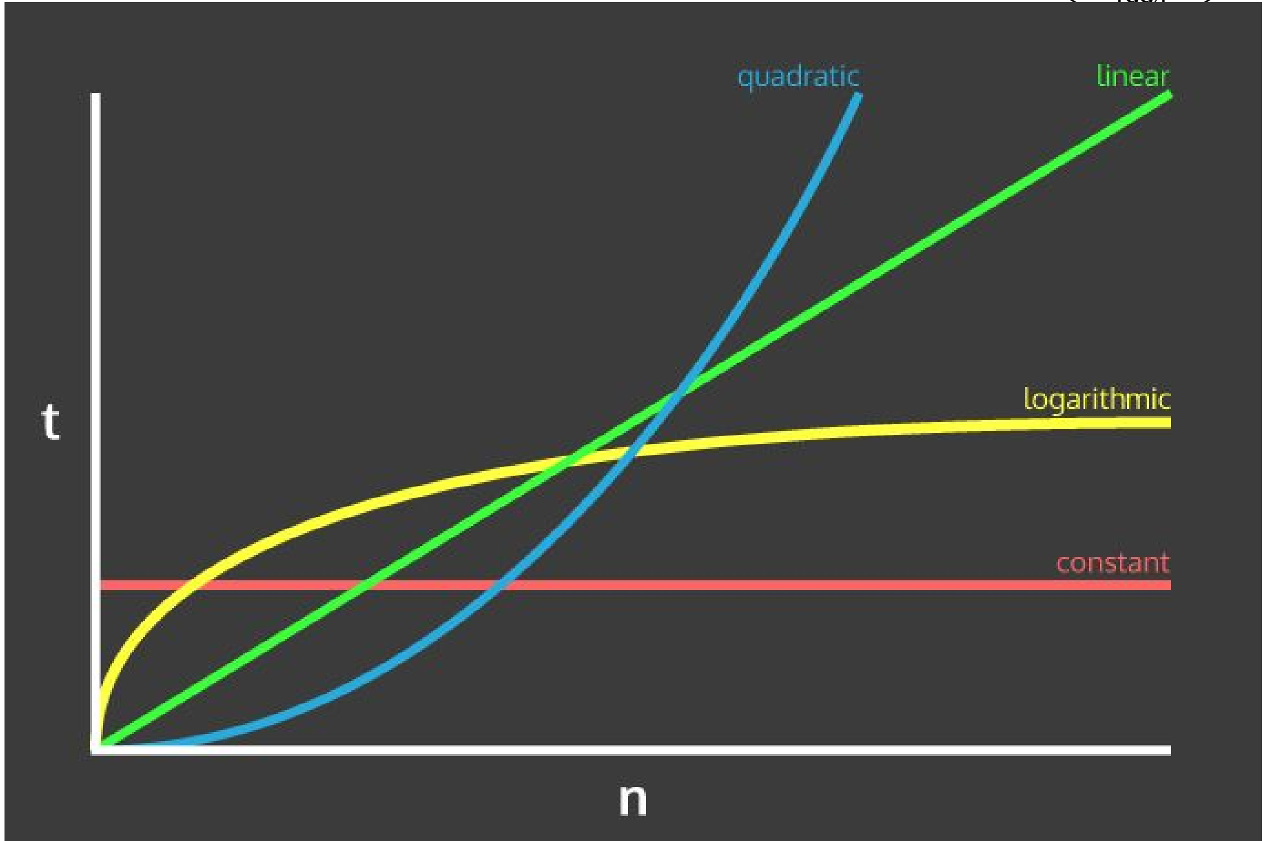
Complexity Analysis

Aula 1

Big O notation

- Constant: $O(1)$
- Linear: $O(n)$
- Logarithmic: $O(\log n)$
- Quadratic: $O(n^2)$
- Exponential: $O(2^n)$
- Factorial: $O(n!)$



O(1) - Constant time

Example 1:

- arr = [1,3,5,7]
- arr[3]

Example 2:

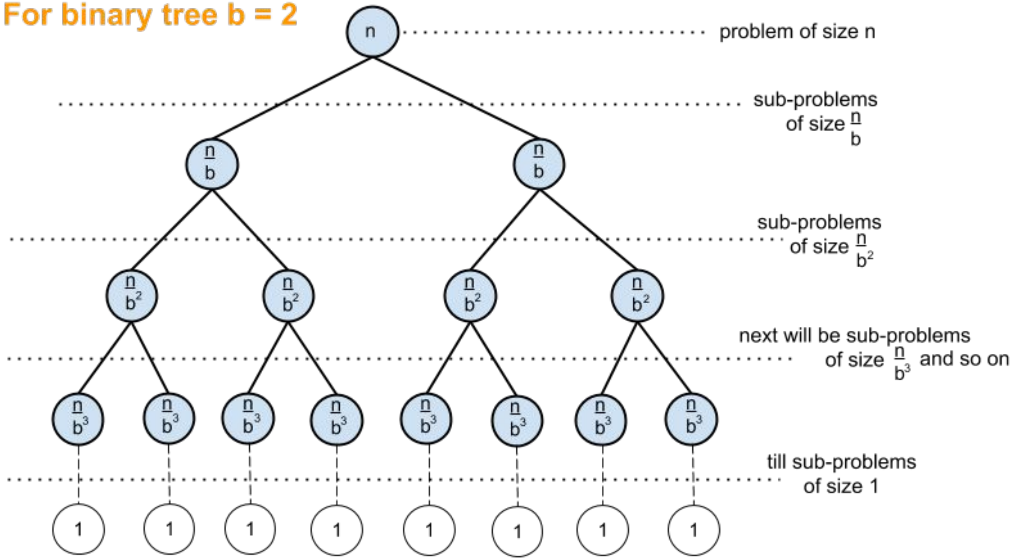
- arr = {"odd":1, "even":2}
- arr["odd"]

O(log n) - Logarithmic time

For each step

The problem is subdivided

For binary tree b = 2


n
problem of size n
sub-problems of size $\frac{n}{b}$
$\frac{n}{b}$
sub-problems of size $\frac{n}{b^2}$
$\frac{n}{b^2}$
next will be sub-problems of size $\frac{n}{b^3}$ and so on
$\frac{n}{b^3}$
till sub-problems of size 1
1

The height of the above tree is answer to the following question: How many times we divide problem of size n by b until we get down to problem of size 1?

The other way of asking same question:

when $\frac{n}{b^x} = 1$ *[in binary tree b = 2]*

i.e. $n = b^x$ which is $\log_b n$ *[by definition of logarithm]*

O(n) - Linear time

Example 1:

```
●  arr = [1,3,5,7]
●  for i in arr:
●      print(i)
```

Example 2:

```
●  def summ(n):
●      if n <=0:
●          return 0
●      return n + summ(n-1)
```

O(n²) - Quadratic

Example:

- `arr = [1,3,5,7]`
- `for i in arr:`
- `    for j in arr:`
- `        print(i+j)`

O(2^n) - Exponential time

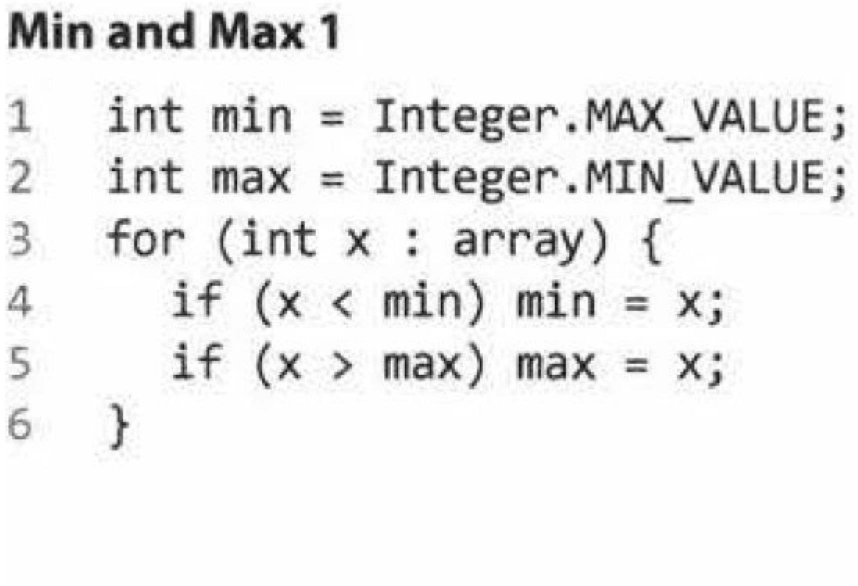
Repeated multiplication:

- $2^4 = 2 \times 2 \times 2 \times 2 = 16$

O($n^!$) - Factorial time

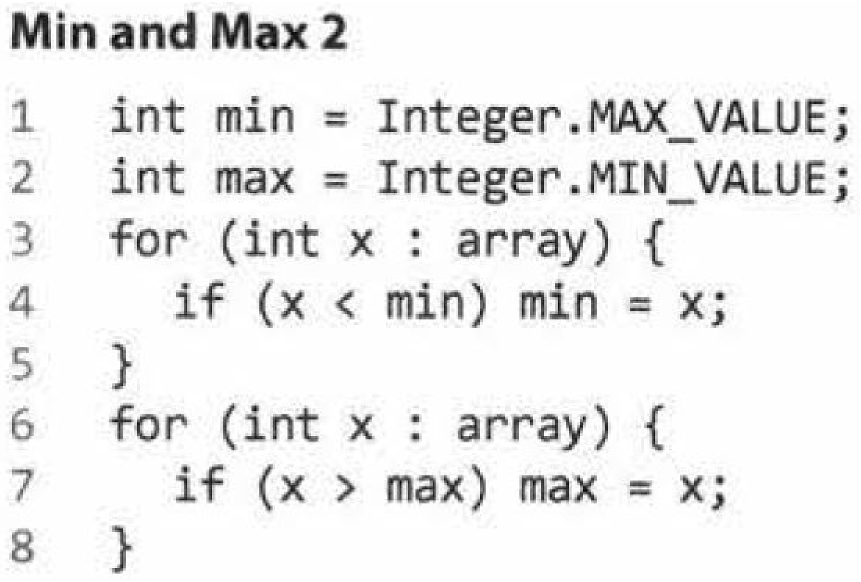
```
def factorial(n):
  if (n == 1):
     return 1
  else:
     return n * factorial(n-1)
```

Which one is faster ?



Min and Max 1

```java
int min = Integer.MAX_VALUE;
int max = Integer.MIN_VALUE;
for (int x : array) {
   if (x < min) min = x;
   if (x > max) max = x;
}
```



Min and Max 2

```java
int min = Integer.MAX_VALUE;
int max = Integer.MIN_VALUE;
for (int x : array) {
   if (x < min) min = x;
}
for (int x : array) {
   if (x > max) max = x;
}
```

O(n)

O(2n)

Both are considered O(n) !

Drop the Non-Dominant Terms

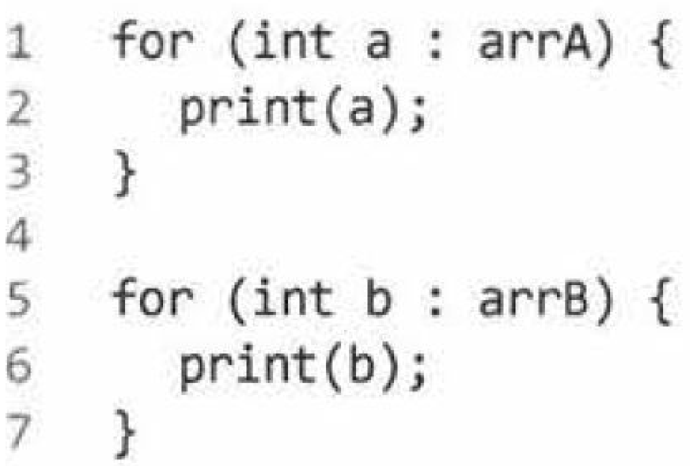
Always considers the worst case for “n”:

- $O(n^2 + n) = O(n^2)$
- $O(n^2 + n^3 + n) = O(n^3)$
- $O(2^{1000} + 2^n) = O(n)$

Adding and multiplying the runtimes

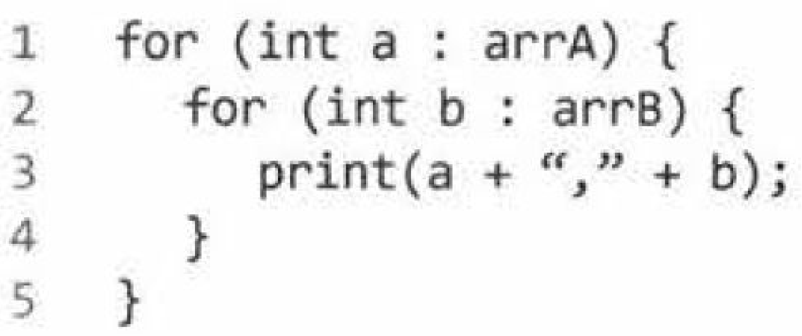
O(A+B) = “do this, then, when you're all done, do that”

O(A*B) = "do this for each time you do that"



```
for (int a : arrA) {
  print(a);
}

for (int b : arrB) {
  print(b);
}
```



```
for (int a : arrA) {
  for (int b : arrB) {
    print(a + “,” + b);
  }
}
```

Recursivity

```
def summ(n):
    if n <=0:
        return 0
    return n + summ(n-1)
```

What is the runtime?

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

What is the runtime ?

- for i in arr:
-     for j in arr:
-         print(i+j)

- for i in arr:
-     for j in arr2:
-         print(i+j)

What is the runtime ?

```
for i in arr:
    for j in arr:
        for k in arr:
            print(i+j+k)
```

```
for i in arr:
    for j in arr:
        for k in range(10000):
            print(i+j+k)
```

A problem-solving flowchart

Brute force -> BUD Optimization (**B**ottlenecks, **U**nnecessary work, **D**uplicated work) -> Code review -> Optimize and write a clean and beautiful code

Python programming

Printing

- print("Hello", "World!")
- # ---------------------------
- hello= "Hello"
- hello+=" World!"
- print(hello)
- # ---------------------------
- w = "World!"
- print("Hello", w)
- # ---------------------------
- h="Hello"
- string = "{} {}".format(h, w)
- print(string)

Array and matrices

- a = [1,3,5,7]
- b = [2,4,6,8]
- c=[a,b]
- print(c)
- # ------------------
- b.append(10)
- a.append(9)

- d = []
- d.append(2)
- c.append(d)
- c.append(1)
- print(c)

Manipulating arrays

```
b = [2,4,6,8]

b.remove(2)

del b[0]

print(b)

b.insert(3, 2)

b.insert(10, 8)

print(b)
```

Tuples and dictionaries

```
a = (1,2,3)

a = list(a)

a.append(5)

a = tuple(a)

print(a)
```

```
b = {}

b["key"] = 45

b.update({"key2":54})

print(b)

b = dict()

b.update({"key3":99})

print(b)
```

Built-in functions

abs(x)

all(iterable)

any(iterable)

bin(x)

hex(x)

dir([object])

enumerate(iterable, start=0)

len(s)

max(iterable, *[, key, default])

min(iterable, *[, key, default])

pow(x, y[, z])

range(start, stop[, step])

set([iterable])

sum(iterable[, start])

Working with files

open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)

- 'r' open for reading (default)
- 'w' open for writing, truncating the file first
- 'x' open for exclusive creation, failing if the file already exists
- 'a' open for writing, appending to the end of the file if it exists
- 'b' binary mode
- 't' text mode (default)
- '+' open a disk file for updating (reading and writing)

Reading from stdin

```
import sys

line = sys.stdin.readline()

while line:

    print line,

    line = sys.stdin.readline()
```

Reading input

Example:

- # Input. Read each line as x, y and print the sum
- 3 2
- 7 8
- 10 15
-
- # Expected output
- 5
- 15
- 25

```
for line in sys.stdin:

    x, y = line.split(' ')

    print int(x) + int(y)
```