

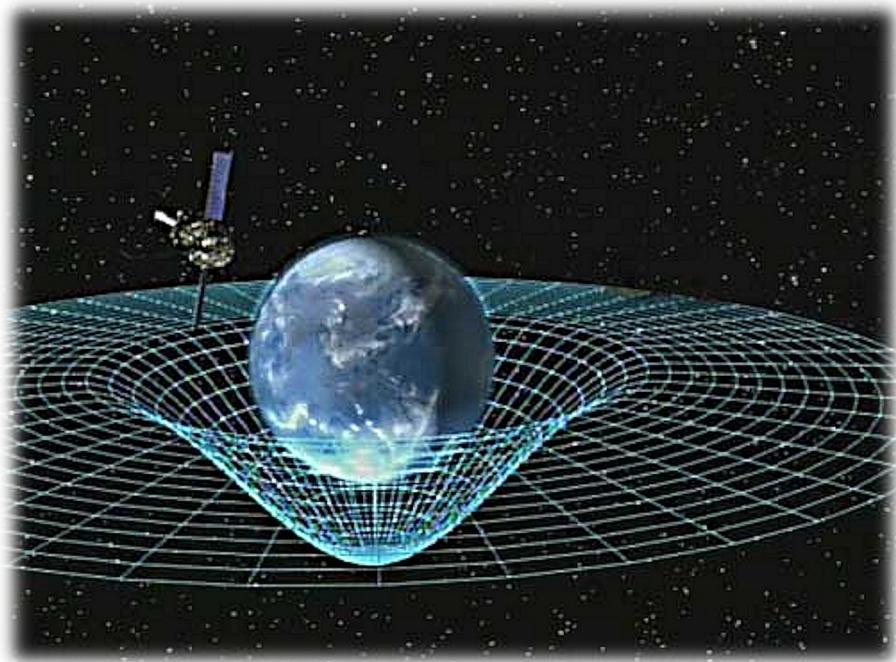
Order of Growth

In Physics, We consider

- Time

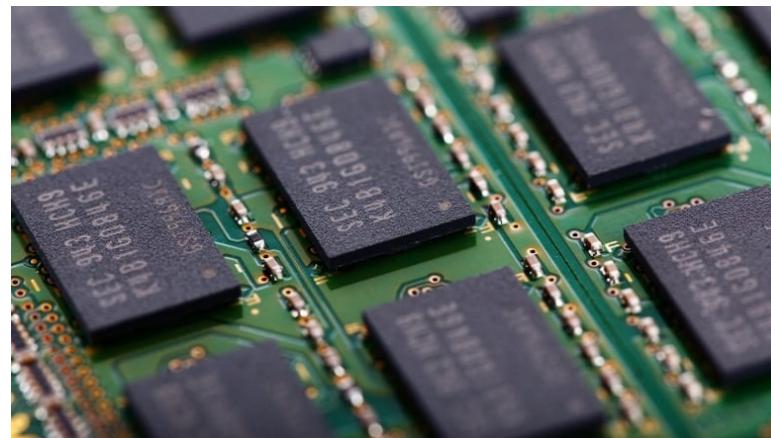


- Space



In CS, we consider

- Time
 - how long it takes to run a program
- Space
 - how much memory do we need to run the program



Order of Growth Analogy

- Suppose you want to buy a Blu-ray movie from Amazon (~40GB)
- Two options:
 - Download
 - 2-day Prime Shipping
- Which is faster?



The Infinity Saga Box Set



Order of Growth Analogy

- Buy the full set?
 - 23 movies
- Two options:
 - Download
 - 2-day Prime Shipping
- Which is faster?

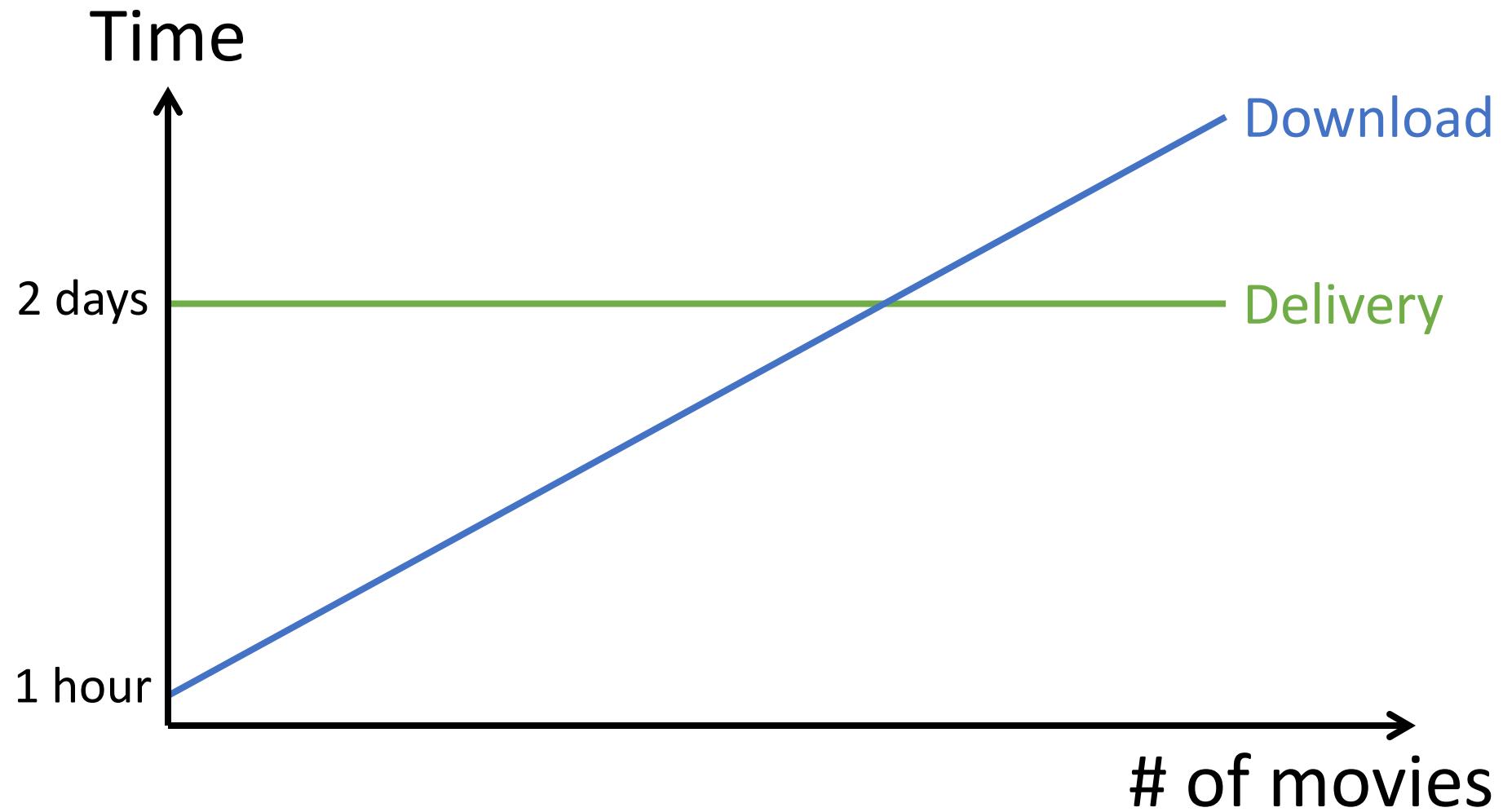


Order of Growth Analogy

- Or even more movies?



Download vs Delivery



Ultimate Question

- If the “volume” increased
- How much more resources, namely **time** and **space**, grow?

Will they grow in the same manner?

- From
 - factorial(10)
- To
 - factorial(20)
- To
 - factorial(100)
- To
 - factorial(10000)
- From
 - fib(10)
- To
 - fib(20)
- To
 - fib(100)
- To
 - fib(10000)

Order of Growth

- is NOT...
 - The **absolute** time or space a program takes to run
- is
 - the **proportion of growth** of the time/space of a program **w.r.t.** the growth of the input

Let's try it on something we know

```
def factorial(n):  
  
    if n <= 1:  
        return 1  
    else:  
        return n * factorial(n - 1)  
  
def fib(n):  
  
    if (n == 0):  
        return 0  
    elif (n == 1):  
        return 1  
    else:  
        return fib(n - 1) + fib(n - 2)
```

Let's try it on something we know

```
nfact, nfib = 0, 0
def factorial(n):
    global nfact
    nfact +=1
    if n <= 1:
        return 1
    else:
        return n * factorial(n - 1)

def fib(n):
    global nfib
    nfib +=1
    if (n == 0):
        return 0
    elif (n == 1):
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Compare

```
>>> factorial(5)
```

```
120
```

```
>>> nfact
```

```
5
```

```
>>> nfact = 0
```

```
>>> factorial(10)
```

```
3628800
```

```
>>> nfact
```

```
10
```

```
>>> nfact = 0
```

```
>>> factorial(20)
```

```
2432902008176640000
```

```
>>> nfact
```

```
20
```

```
>>> fib(5)
```

```
5
```

```
>>> nfib
```

```
15
```

```
>>> nfib = 0
```

```
>>> fib(10)
```

```
55
```

```
>>> nfib
```

```
177
```

```
>>> nfib = 0
```

```
>>> fib(20)
```

```
6765
```

```
>>> nfib
```

```
21891
```

Order of Growth of Factorial

```
>>> factorial(5)
120
>>> nfact
5
>>> nfact = 0
>>> factorial(10)
3628800
>>> nfact
10
>>> nfact = 0
>>> factorial(20)
2432902008176640000
>>> nfact
20
```

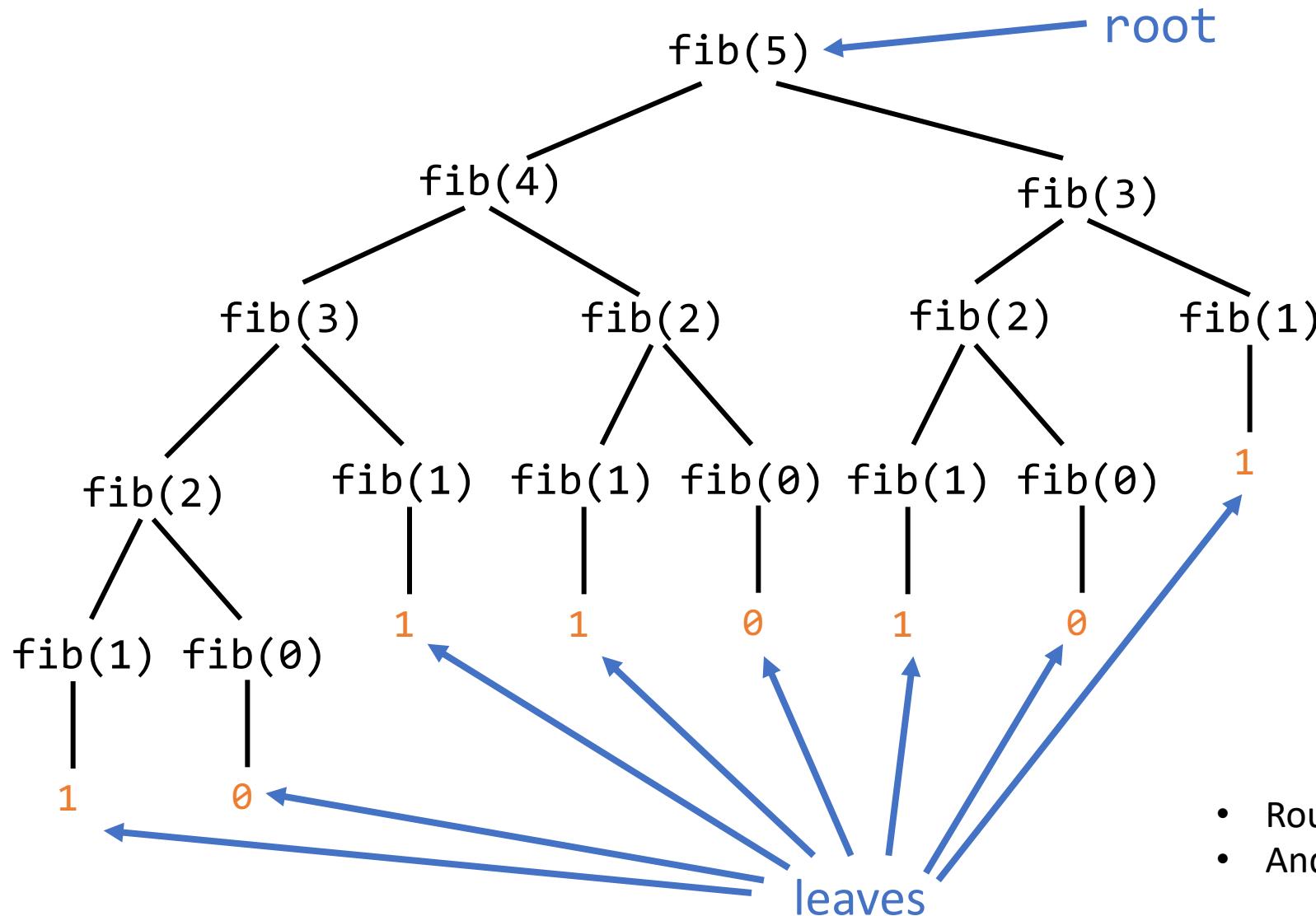
- Factorial is simple
 - If the input is n , then the function is called n times
 - Because each time n reduced by 1
- So the number of times of calling the function is proportional to n

Fib

- More compliated

```
>>> fib(5)
5
>>> nfib1
5
>>> nfib = 0
>>> fib(10)
55
>>> nfib
177
>>> nfib = 0
>>> fib(20)
6765
>>> nfib
21891
```

Tree recursion



- Roughly half of a full tree
- And a full tree has $2^n - 1$ nodes

Compare

```
>>> factorial(5)
```

```
120
```

```
>>> nfact
```

```
5
```

```
>>> nfact = 0
```

```
>>> factorial(10)
```

```
3628800
```

```
>>> nfact
```

```
10
```

```
>>> nfact = 0
```

```
>>> factorial(20)
```

```
2432902008176640000
```

```
>>> nfact
```

```
20
```

```
>>> fib(5)
```

```
5
```

```
>>> nfib1
```

```
5
```

```
>>> nfib = 0
```

```
>>> fib(10)
```

```
55
```

```
>>> nfib
```

```
177
```

```
>>> nfib = 0
```

```
>>> fib(20)
```

```
6765
```

```
>>> nfib
```

```
21891
```

No of calls proportional to n

No of calls proportional to 2^n

Searching in a list of n items

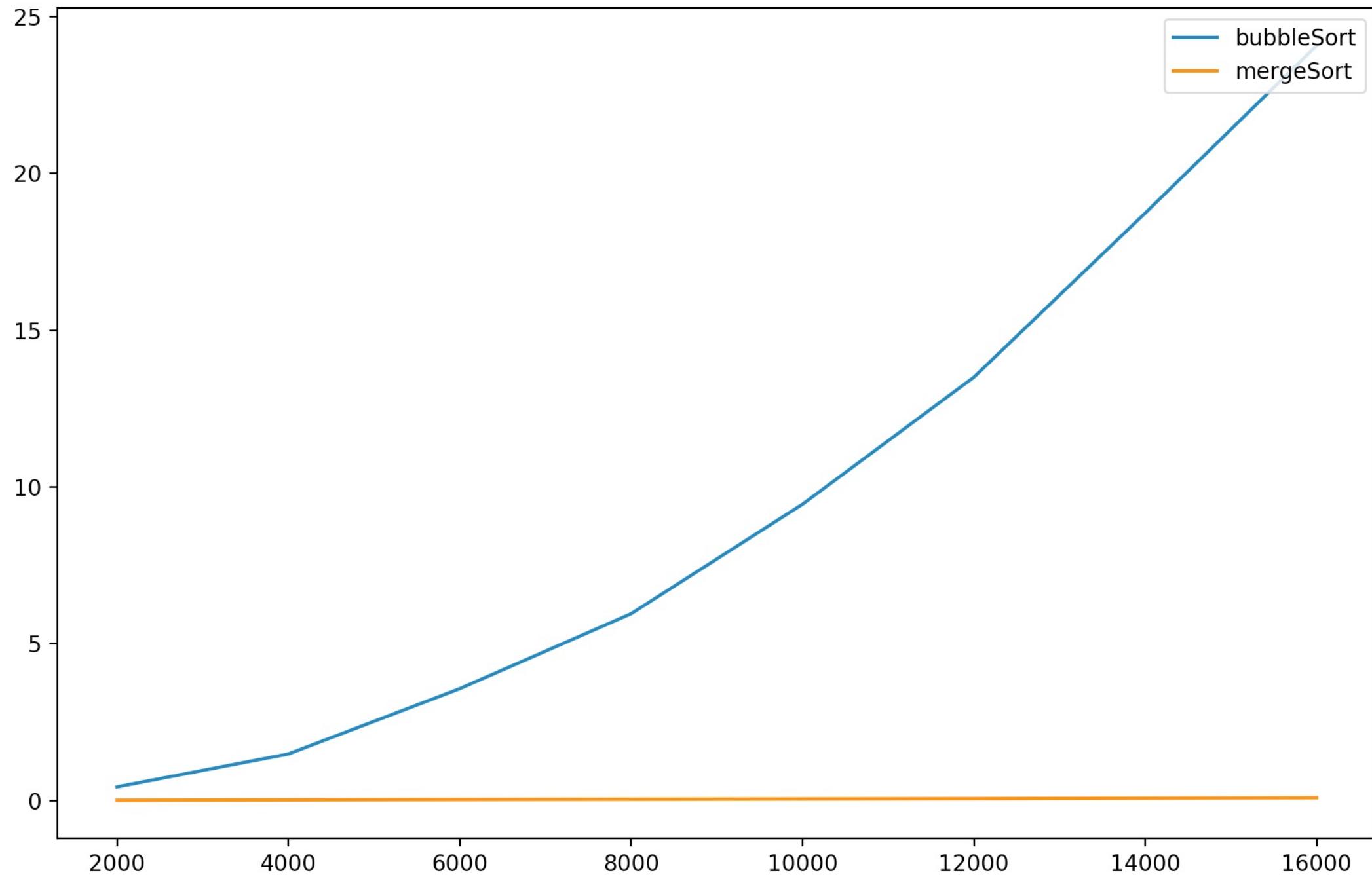
- Linear search
 - # comparisons proportional to n
 - (Because in average, the expected number of search is $n/2$)
- Binary search
 - # comparisons proportional to $\log n$
 - Because, we divide the list into half for at most $\log n$ times

Sorting a list of n Items

- Selection/Bubble/Insertion Sort
 - # comparisons proportional to n^2
 - Because we looped n times, and each time you need to arrange 1 to n items
- Merge sort
 - # comparisons proportional to $n \log n$
 - Because, we divide the list into half for at most $\log n$ times
 - And each time arrange n items

```
from random import randint
from time import time
ln = [2000,4000,6000,8000,10000,12000,14000,16000]

bstat = []
mstat = []
for n in ln:
    rl = [randint(1,100000) for i in range(n)]
    st = time()
    bubbleSort(rl)
    btime = time()-st
    st = time()
    mergeSort(rl)
    mtime = time()-st
    print(f'For n = {n}, bubbleSort: {btime}s mergeSort: {mtime}s')
    bstat.append(btime)
    mstat.append(mtime)
```



Algorithm

Anyone can give some algorithms

BogoSort

- BogoSort (L)
 - Repeat:
 - Choose a random permutation of the list L .
 - If L is sorted, return L .
 - If you wait enough time, L is sorted?



Algorithm

Anyone can give some algorithm

But how fast is your algorithm?

How about

`QuantumBogoSort(A[1..n])`

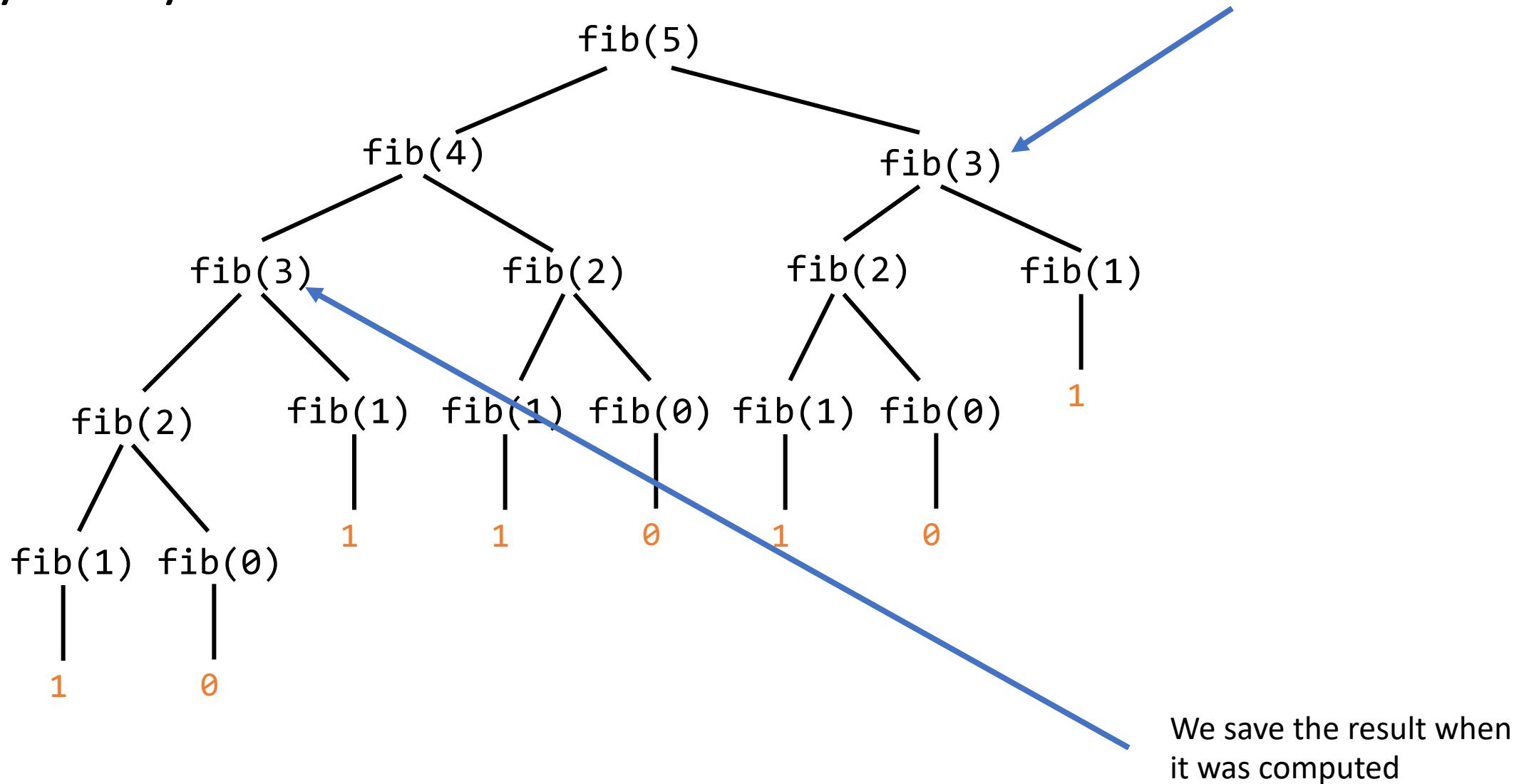
- a) Choose a random permutation of the array A.
- b) If A is sorted, return A.
- c) If A is not sorted, destroy the universe.

- Remember QuantumBogoSort when you learn about non-deterministic Turing Machines

Improvement?

Let's try $\text{fib}(n)$

Easy Way: Memoization



Memoization

- Create a dictionary to remember the answer if `fibm(n)` is computed before
- If the ans was computed before, get the answer direction from the dictionary
- Otherwise, compute the ans and put it into the dictionary for later use

```
fibans = {}

def fibm(n):
    if n in fibans.keys():
        return fibans[n]

    if (n == 0):
        ans = 0
    elif (n == 1):
        ans = 1
    else:
        ans = fibm(n - 1) + fibm(n - 2)
    fibans[n] = ans
    return ans
```

Recursion Removal

- Store all the answers in an array

```
def fibi(n):
    ans = [0,1,1]
    if n < 3:
        return ans[n]
    for i in range(3,n+1):
        ans.append(ans[i-1]+ans[i-2])
    return ans[n]
```

- Add a new $\text{fib}(i)$
 - as $\text{fib}(i-1) + \text{fib}(i-2)$
- Wait a min...
 - Do we need all the past numbers if we only need $\text{fib}(n)$?

Recursion Removal 2

- Add a new $\text{fib}(i)$
 - as $\text{fib}(i-1) + \text{fib}(i-2)$
- And I only need to keep $\text{fib}(i-1)$ and $\text{fib}(i-2)$

```
def fibi2(n):
    if n < 3:
        return 1
    fibminus1, fibminus2 = 1, 1
    for i in range(3,n+1):
        fibminus2, fibminus1 = fibminus1, fibminus1 + fibminus2
    return fibminus1
```

Improvement

- For IT5001, you should know how to compute $\text{fib}(n)$ with time proportional to n
 - The fastest algorithm to compute $\text{fib}(n)$ with time proportional to $\log n$
- To know more about this ~~to improve human race and save the world~~
 - CS1231, CS2040, CS3230, etc
- What you learn today is called the Big O notation
 - $O(n)$, $O(\log n)$, $O(n^2)$, $O(n \log n)$, $O(2^n)$, etc