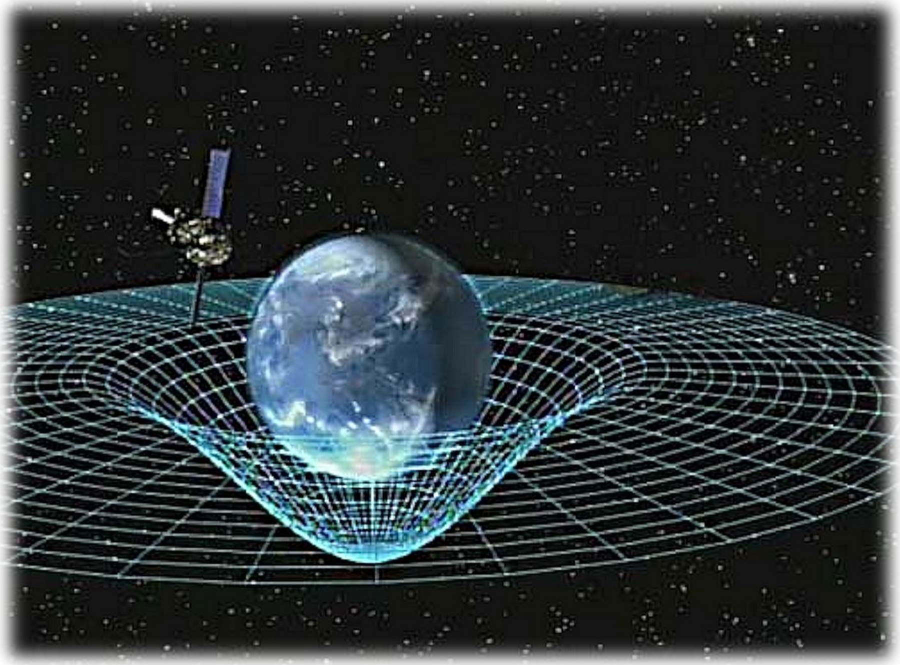
Order of Growth

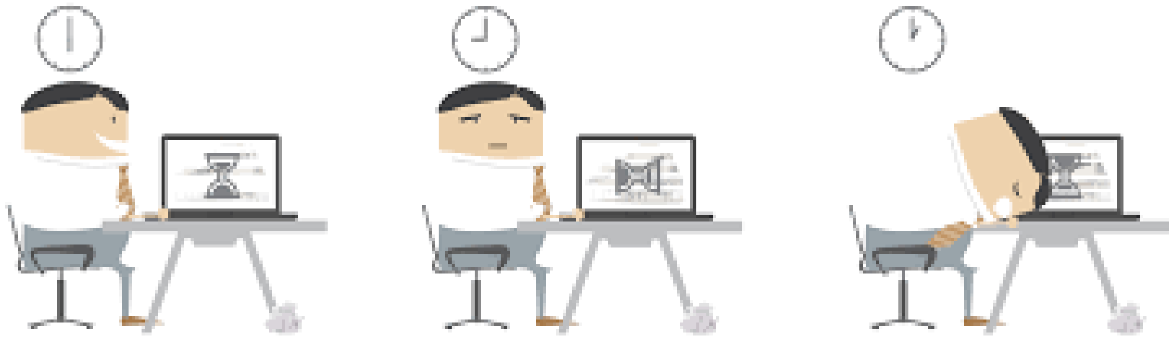
In Physics, We consider

- Time

- Space

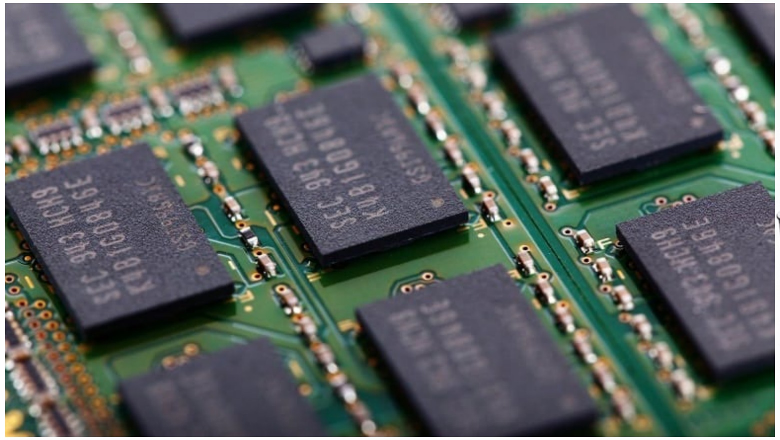


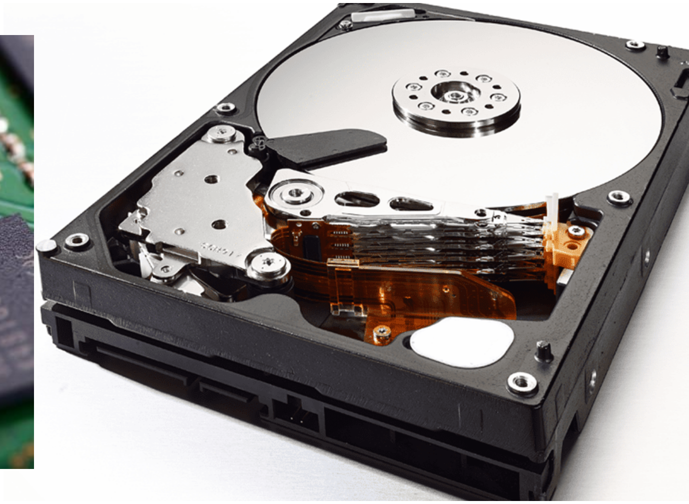
In CS, we consider



- 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?

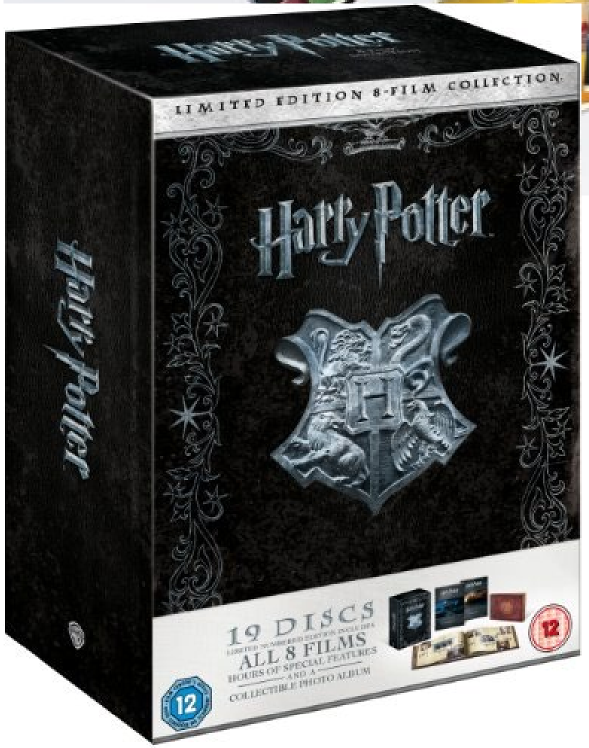
The Infinity Saga Box Set

Order of Growth Analogy

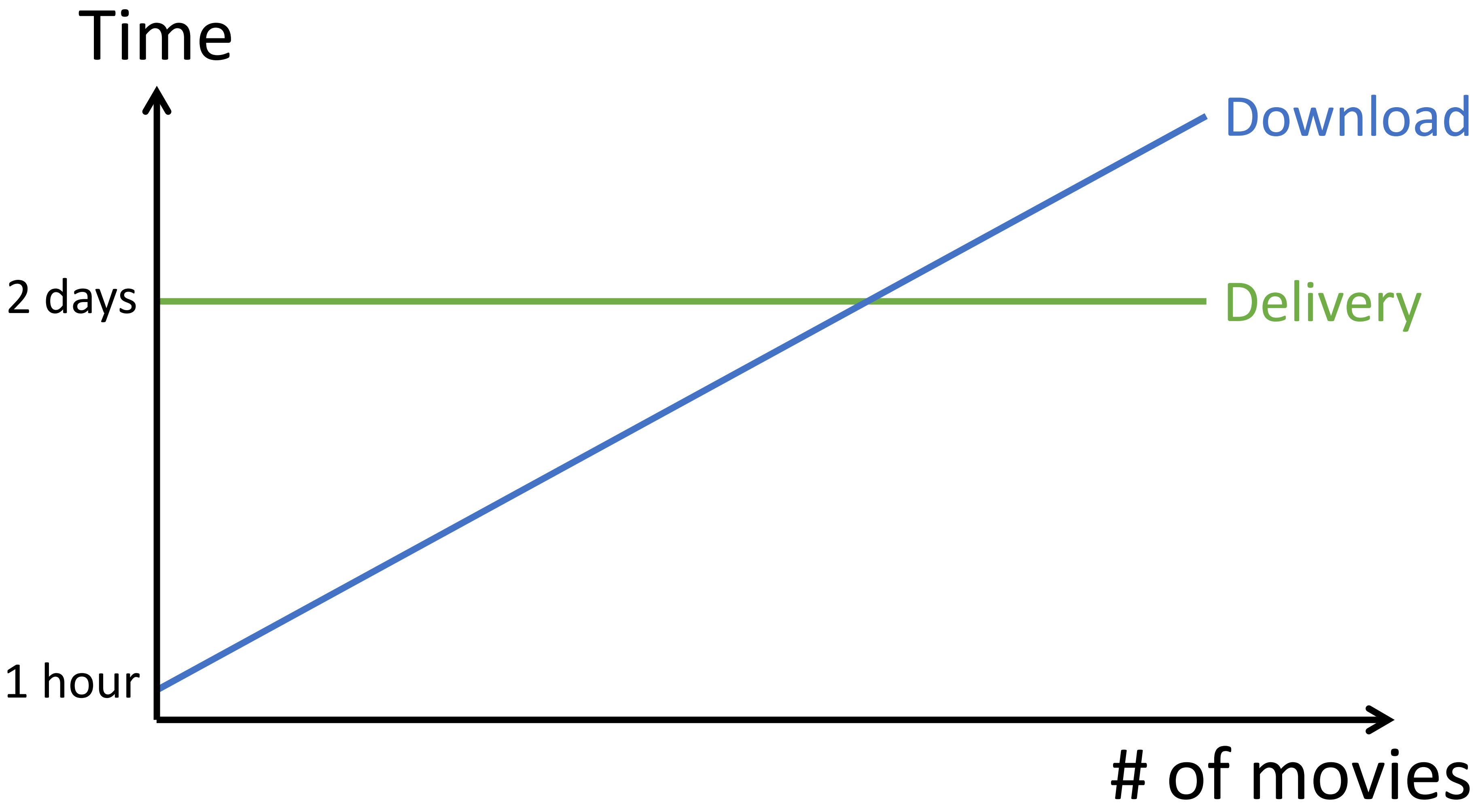
- Buy the full set?
  - 23 movies
- Two options:
  - Download
  - 2-day Prime Shipping
- Which is faster?

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 <u>proportional to n</u>

factorial(5)
↓
factorial(4)
↓
factorial(3)
↓
factorial(2)
↓
factorial(1)

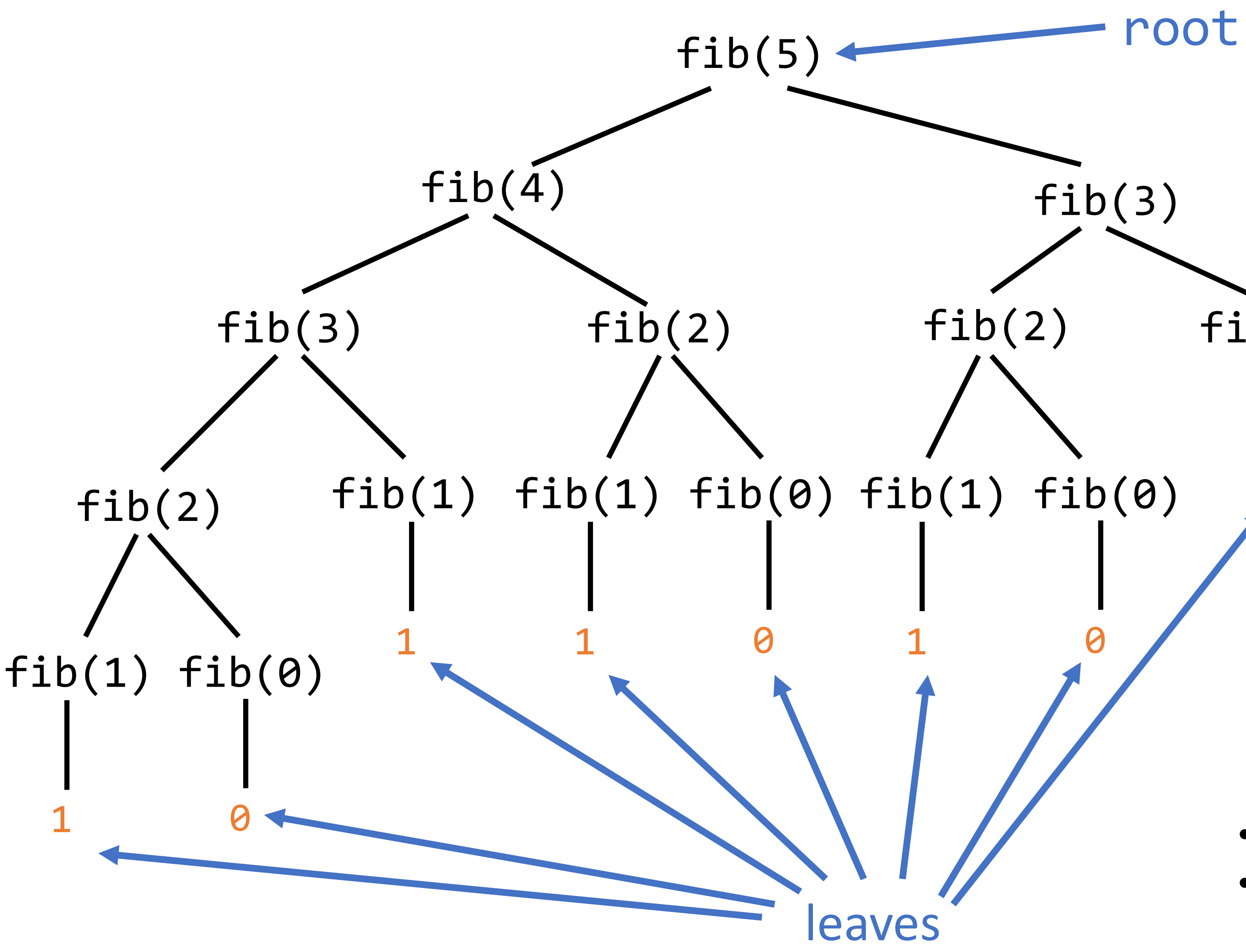
Fib

- More complicated

Why?

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

Fibonacci: 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
```

No of calls proportional to n

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

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

Bubble Sort

8	4	5	9	2	3	7	1	6	0
4	8	5	9	2	3	7	1	6	0
4	5	8	9	2	3	7	1	6	0
4	5	8	9	2	3	7	1	6	0
4	5	8	2	9	3	7	1	6	0
4	5	8	2	3	9	7	1	6	0
4	5	8	2	3	7	9	1	6	0
4	5	8	2	3	7	1	9	6	0
4	5	8	2	3	7	1	6	9	0
4	5	8	2	3	7	1	6	0	9

Bubble Sort

4	5	8	2	3	7	1	6	0	9
4	5	8	2	3	7	1	6	0	9
4	5	8	2	3	7	1	6	0	9
4	5	2	8	3	7	1	6	0	9
4	5	2	3	8	7	1	6	0	9
4	5	2	3	7	8	1	6	0	9
4	5	2	3	7	1	8	6	0	9
4	5	2	3	7	1	6	8	0	9
4	5	2	3	7	1	6	0	8	9

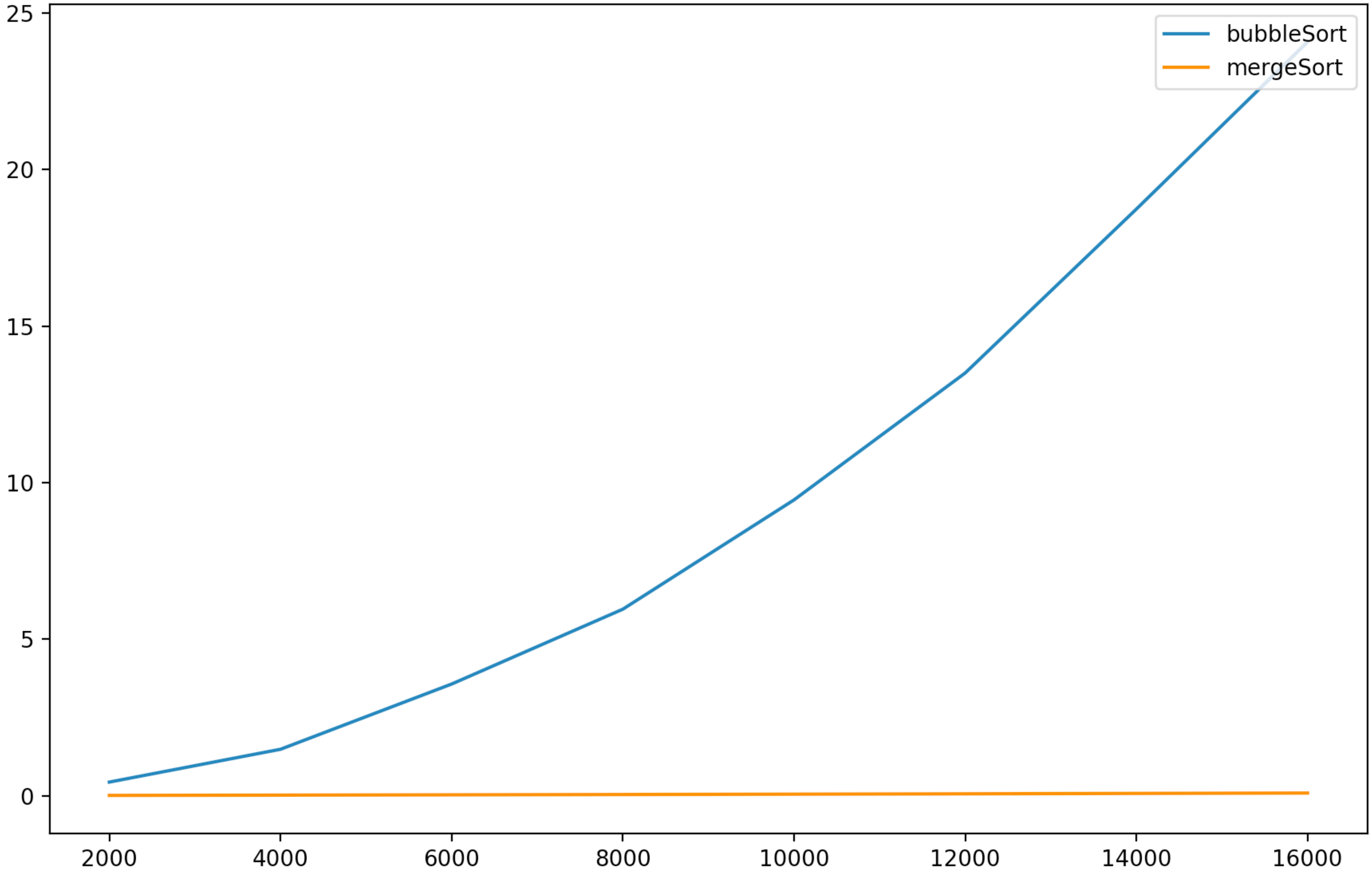
Bubble Sort

```
def bubble(my_list):
    for i in range(len(my_list)-1):
        if my_list[i] > my_list[i+1]:
            if my_list[i] > my_list[i+1]:
                my_list[i], my_list[i+1] = my_list[i+1], my_list[i]


def bubblesort(my_list):
    for i in range(len(my_list)-1):
        bubble(my_list)

my_list_1 = [38,2,10,3,1]
bubblesort(my_list_1)
print(my_list_1)
```

Sorting a list of n Items

- Selection/Bubble 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

```python
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)
```



bubbleSort
mergeSort
25
20
15
10
5
0
2000
4000
6000
8000
10000
12000
14000
16000

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?

Bogo Sort

- Randomly shuffle the list till the list is sorted

```
import random
def is_not_sorted(shuffled_list):
    for i in range(len(shuffled_list)-1):
        if shuffled_list[i] > shuffled_list[i+1]:
            return True
    return False

def bogosort(my_list):
    while is_not_sorted(my_list):
        random.shuffle(my_list)

my_list = [38,2,10,3,1]
bogosort(my_list)
print(my_list)
```

Can we do better?

Hill-Climbing for Sorting

- Optimization algorithm
  - Require an evaluation function

- Which metric is better for evaluation?
  - Let my_list be our list
  - Number of index pairs i, j such that such that $my\_list[i] > my\_list[j]$

- Example:

```
>>> my_list = [38,2,10,3,1]
>>> my_list
[38, 2, 10, 3, 1]
```

$Value(my\_list) = 4 + 1 + 2 + 1 = 8$

```
>>> my_list = [1, 2, 3, 10, 38]
>>> my_list
[1, 2, 3, 10, 38]
```

$Value(my\_list) = 0 + 0 + 0 + 0 = 0$

Hill-Climbing for Sorting

- Repeat the following either till value of the list is zero or a predetermined number of times
  - Shuffle the list
  - Accept the shuffled list if its value is lower than that of current list

Algorithm

Anyone can give some algorithm

But how fast is your algorithm?

How about

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

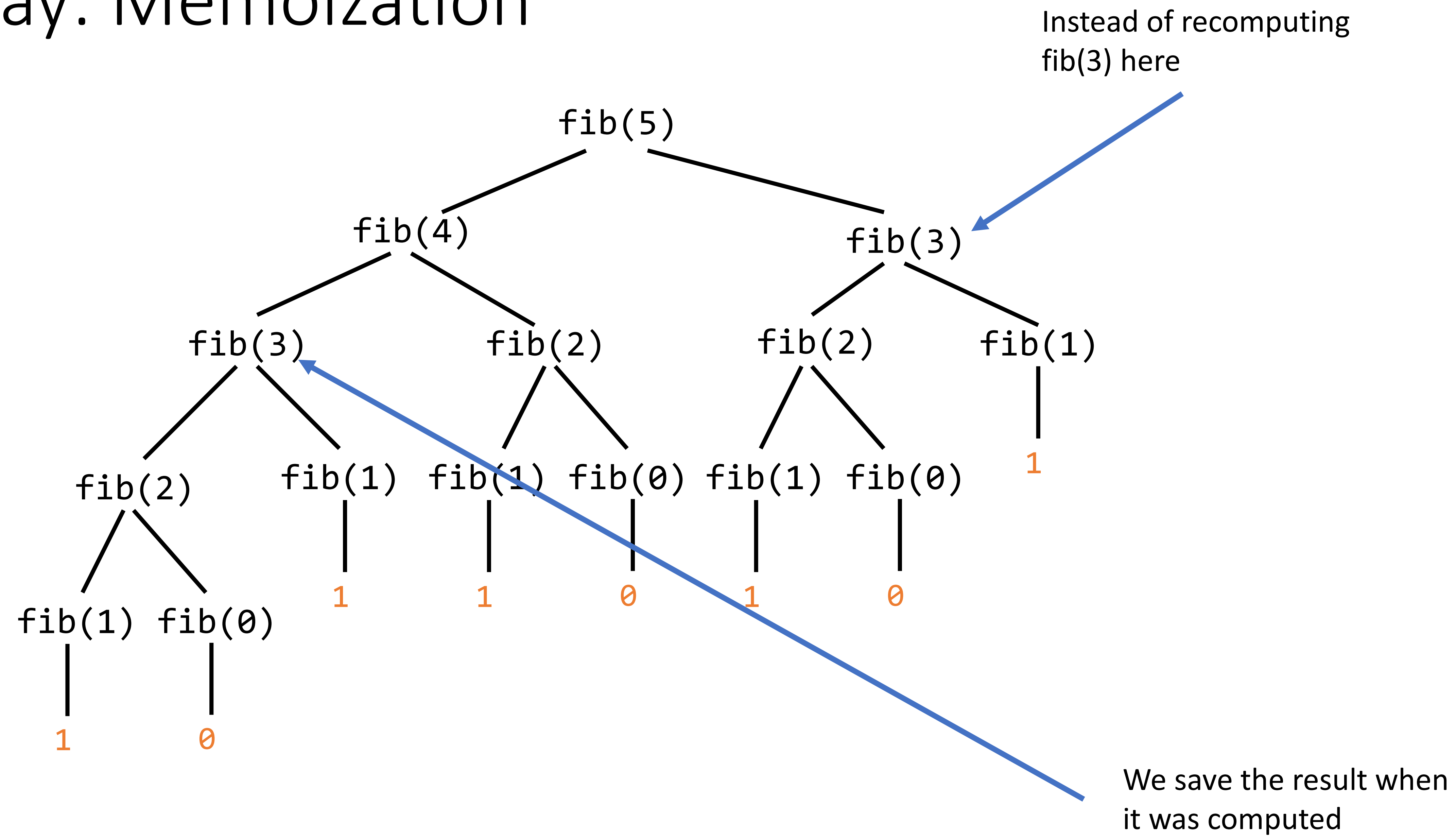
a) Choose a random permutation of the array A.
b) If A is sorted, return A.
c) If A is not sorted, destroy the universe.

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

Improvement?

Let’s try 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 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
```

Can you use Memoization to compute nCk?

Recursion Removal

- Store all the answers in an array
- Add a new fib(i)
  - as fib(i-1) + fib(i-2)

```
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]
```

- Wait a min…
  - Do we need all the past numbers if we only need fib(n)?

Recursion Removal 2

- Add a new fib(i)
  - as fib(i-1) + fib(i-2)
- And I only need to keep fib(i-1) and 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 fib(n) with time proportional to n
  - The fastest algorithm to compute 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