



15-110 PRINCIPLES OF COMPUTING – S19

LECTURE 26:

EFFICIENCY AND COMPLEXITY OF ALGORITHMS

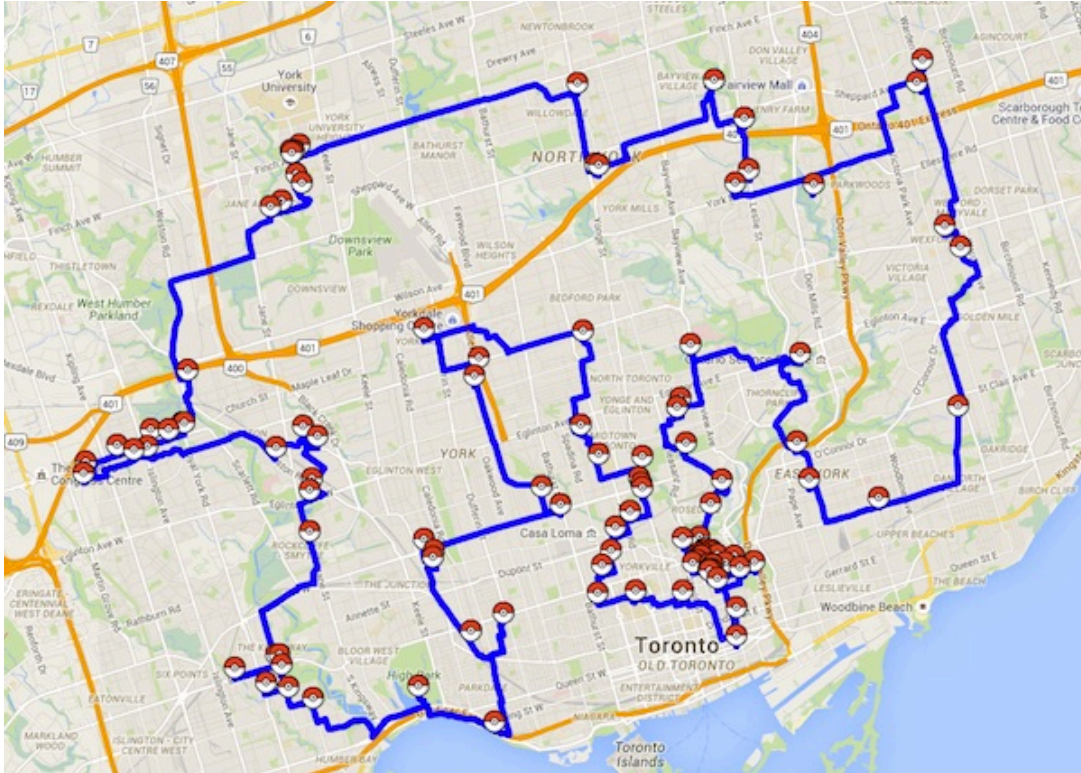
TEACHER:

GIANNI A. DI CARO

Designing an algorithm: what are the main criteria?

- ✓ **Correctness:** the algorithm / program does what we expect it to do, without errors
- ✓ **Efficiency:** the results are obtained within a time frame which is compatible with the time restrictions of the scenario of use of the program (and the shorter the time the better)
 - ✓ Efficiency is not *optional*, since a program might result unusable in practice if it takes too long to produce its output
- ❖ **Portability, modularity, maintainability, ...** always useful, but in a sense also *optional*

Example: an algorithm for defining *routes*



- Given a set of sites to visit, find the **shortest route** that visits all sites (and never passes to the same location twice), and goes back to the starting point
- **Traveling Salesman Problem (TSP)**
- Locations can model:
 - bus stops
 - customers
 - goods delivery
 - sites in electronic printed boards,
 - Pokemon Go locations,
 - DNA sequences (shortest common superstring)
 - ...

Shortest common superstring in sets of DNA sequences

$S = \{ \text{ATC}, \text{CCA}, \text{CAG}, \text{TCC}, \text{AGT} \}$

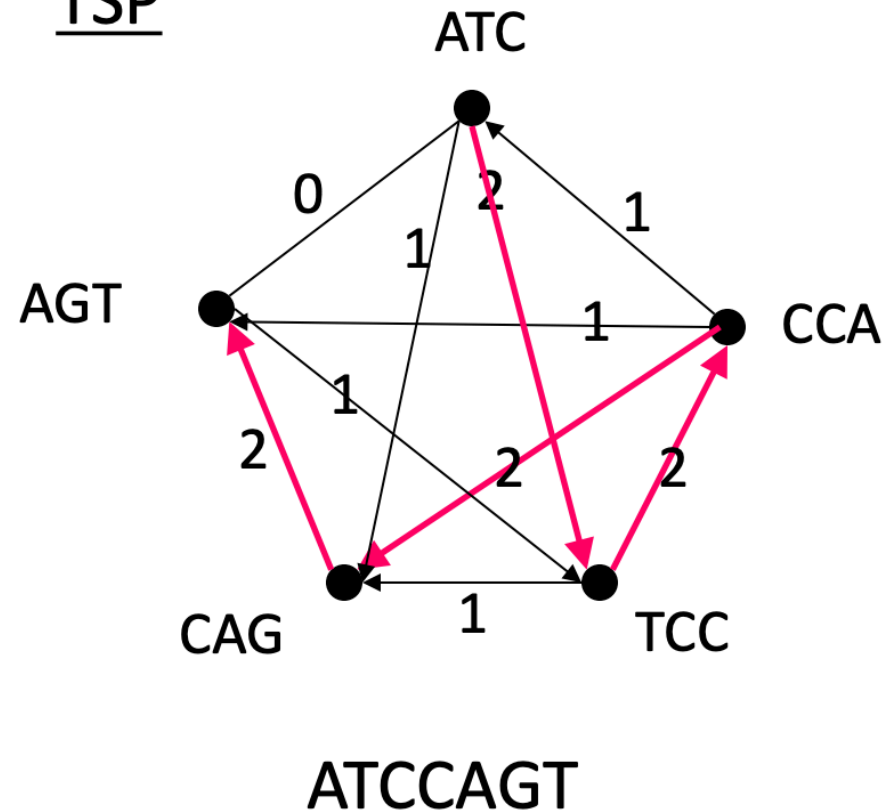
SSP

AGT
CCA
ATC
ATCCAGT
TCC
CAG

Distances / costs from s_i to s_j are defined in terms of *overlaps*: the length of the longest suffix of s_i that matches a prefix of s_j

→ Maximize overlaps (or minimize $1/\text{Distance}$)

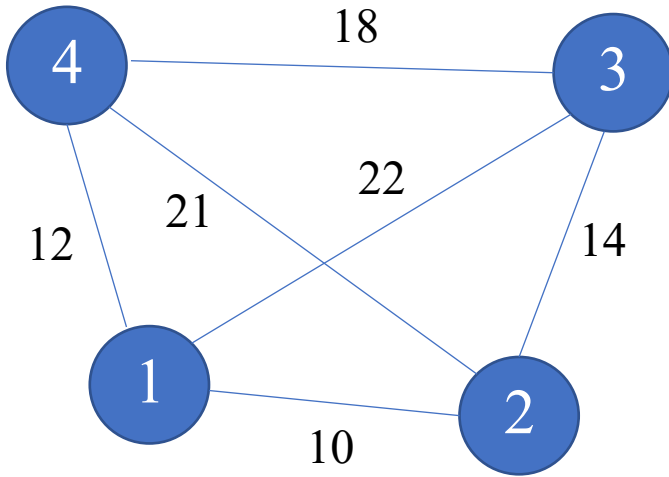
TSP



Naïve algorithm for TSP

- Given a set of sites to visit, find the **shortest route** that visits all sites (and never passes to the same location twice), and goes back to the starting point

➤ A TSP with $n = 4$ sites to visit



12341	21342	31243	41234
12431	21432	31423	41324
13241	23142	32143	42134
13421	23412	32413	42314
14231	24132	34123	43124
14321	24312	34213	43214

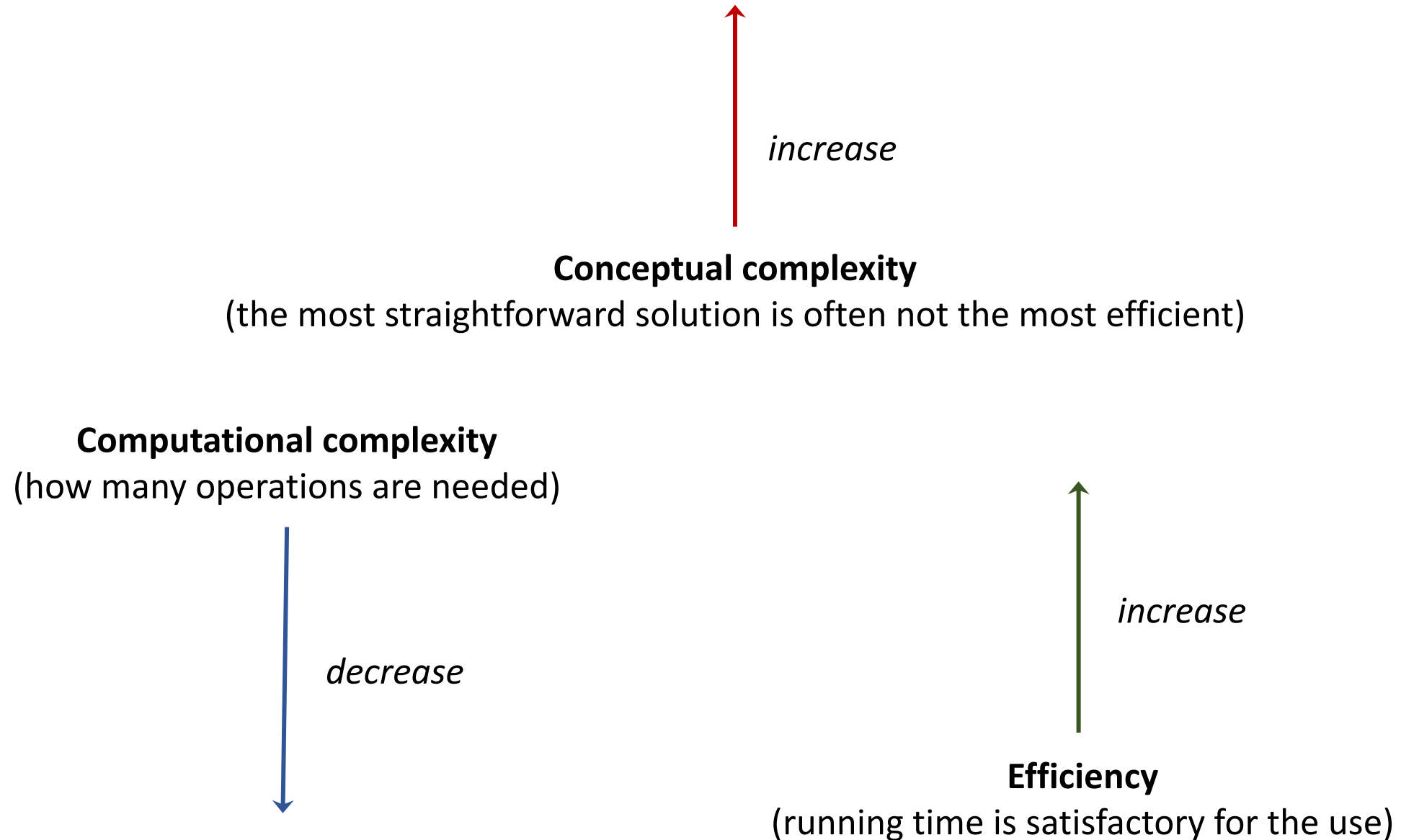
1. List all possible routes that visit each site once and only once and go back to the starting site
2. Compute the travel time (cost for *moving*) for each route
3. Choose the shortest one

How many possible routes?

$\approx n!$

$$n = 100 \rightarrow 100! \approx 9 \cdot 10^{157}$$

We need *efficient* algorithms



Computational performance of algorithms

- Core question 1 (**time**): how *long* will it take to the program to produce the desired output?
- Core question 2 (**space**): how much *memory* does the program need to produce the desired output?

☹ **Answer is *not* obvious at all. It will depend, among others, upon:**

- **Speed of the computer** on which the program will run
(e.g., a smartphone vs. a Google server)
- The **programming language** used to implement the algorithm
(e.g., C vs. Python)
- The **efficiency of the implementation**, for the specific machine
- The **value of the input**
(e.g., n , how many sites have to be visited in the TSP example)

We need *efficient* algorithms

- The **programming language** used to implement the algorithm

```
dim = 50000
x = [0] * dim
y = [0] * dim

for i in range(dim):
    y[i] = i * 1.0

for i in range(dim):
    for j in range(dim):
        x[i] += y[j]
```

≈ 7 minutes (on my machine)


≈ 7 seconds (on my machine)

```
int main()
{
    int dim = 50000;
    float x[dim];
    float y[dim];

    int i;
    for (i = 0; i < dim; i++)
    {
        y[i] = i * 1.0;
    }

    for (i = 1; i < dim; i++)
    {
        for (int j = 1; j < dim; j++)
        {
            x[i] += y[j];
        }
    }
}
```


Machine-independent analysis of algorithms

- **Speed of the computer** on which the program will run
(e.g., a smartphone vs. a Google server)
 - The **programming language** used to implement the algorithm
(e.g., C vs. Python)
 - The **efficiency of the implementation**, for the specific machine
- 
- These are based on measuring performance in ***running time*** (e.g., in seconds)
 - This measure of time is *machine-dependent* and *implementation-dependent*
- ✓ Let's define a more **abstract measure of time** that allows to score the performance of an algorithm in a way which is machine-independent and implementation-independent

Machine-independent analysis of algorithms

- ❖ Instead of cpu time (of a specific implementation on a specific machine), an algorithm is *scored* based on the number of **steps** / **basic operations** that it requires to accomplish its task
- We assume that **every basic operation takes constant time**
- Example Basic Operations:
Addition, Subtraction, Multiplication, Division, Memory Access, Indexing
- Non-basic Operations:
Sorting, Searching
- ❖ **Efficiency of an algorithm is the *number of basic operations* it performs**
 - We do not distinguish between the basic operations, each basic operation counts one step
 - The abstract running time measure is the number of basic operations → ***Time complexity***
 - This allows us to compare two algorithms on a more general terms than using cpu time

Basic operations

❖ Examples of basic operations

- **Assignment:** `x = 1, a = True, s = 'Hello', L = [], d = {0:1, 2:4}`
- **Arithmetic operations:** `x + 2, 3 * y, x / 2, y // 6, x % 3`
- **Relational operations:** `x > 0, y == 2, z <= y, not z`
- **Indexing, memory access:** `x[2], L[i], y[0][3]`

❖ How many basic operations?

```
1  def factorial(n):  
2      res = 1  
3      for i in range(n):  
4          res *= i+1  
5      return res
```

Basic operations and time complexity: input size

❖ How many basic operations?

```
1 def factorial(n):  
2     res = 1  
3     for i in range(n):  
4         res *= i+1  
5     return res
```

$n = 1000$

- One assignment in line 2
- One assignment (to i) and one arithmetic operation / indexing for each one of the n values generated by `range()` in line 3
- One assignment and two arithmetic operations in line 4, repeated n times
- One memory operation in 5

$n(2 + 3)$

$$1 + 5n + 1 = 5n + 2$$

5002 for the specific input

Time complexity depends on size of the input!

Basic operations and time complexity: input value

❖ How many basic operations?

```
1 def linear_search(L, item):  
2     for i in L:  
3         if i == item:  
4             return True  
5     return False
```

- Indexing plus assignment in line 2
 - One relational operation in line 3
 - One memory operation in either 4 or 5
- } Repeated at most n times, where $n = \text{len}(L)$

`L = [1, 3, -2, 0, 100, 18, 45, 32, -9, 5]`

`#operations for linear_search(L, 99):` $n = 10$

`#operations for linear_search(L, 1):` $n = 1$

`#operations for linear_search(L, 100):` $n = 5$

Time complexity depends on value of the input!

Best, worst, and average cases depending on input value

```
L = [1, 3, -2, 0, 100, 18, 45, 32, -9, 5]
```

```
1 def linear_search(L, item):  
2     for i in L:  
3         if i == item:  
4             return True  
5     return False
```

```
#operations for linear_search(L, 1):    n = 1
```

- ✓ **Best-case running time:** The minimum running time over all possible inputs of a given size
 - For `linear_search()` the best-case running time is **constant**: it's independent of the sizes of the inputs

```
#operations for linear_search(L, 99):    n = 10
```

- ✓ **Worst-case running time:** The maximum running time over all possible inputs of a given size
 - For `linear_search()` the worst-case running time is **linear** in the size of the input list

```
#operations for linear_search(L, 100):    n = 5
```

- ✓ **Average-case running time:** The average running time over all possible inputs of a given size
 - For `linear_search()` the average-case running time is **linear** ($n/2$) in the size of the input list

Worst-case machine-independent time complexity analysis

Summarizing what you have achieved so far:

- ❖ Using the number of basic operations as an abstract *time* measure, we can carry out a machine- and implementation-independent analysis of the running time performance of an algorithm
- ❖ Running time performance defines the **time complexity** of the algorithm (*how long* it will take to get the desired results), and it depends on both **size** and **value** of the input
- ❖ In terms of dependence on input values, **best**, **worst** and **average** cases can happen for running time (for the same input size)

Worst-case machine-independent time complexity analysis

What we would like to have more:

- A function $f(\textit{algorithm}, n)$ that, depending on the size of the input n , quantifies the time complexity of an algorithm
- Unfortunately, the time complexity depends on both size and value of input, we can't define such a function accounting for best, worst, and average cases ☹️
- But, we can define such a function if we focus on the **worst-case scenario**, that makes f *independent* of the values of the input since only the worst case for the values is considered

Worst-case analysis: it provides an **upper bound** on the running time

By being as *pessimistic* as possible, we are safe that the program will not take longer to run, and if it runs faster, all the better.

Size of the input(s)

- An algorithm can have *multiple inputs*, some inputs may affect the time complexity, others may not, in general hereafter the size of the input refers to the specific **combination of inputs** that affects the running time of the algorithm

```
1 def linear_search(L, item):
2     for i in L:
3         if i == item:
4             return True
5     return False
```

Under the assumption of worst-case analysis, the input argument `item` doesn't contribute to the time complexity of the function, only the length of the list `L` does

```
1 def sumMtoN(m, n):
2     sum = 0
3     for i in range(m, n+1):
4         sum += i
5     return sum
```

Time complexity is determined by both `m` and `n`, more specifically, by their difference `n-m`, that quantifies the *size* of the input

Worst-case time complexity function and additive constants

```
1 def factorial(n):  
2     res = 1  
3     for i in range(n):  
4         res *= i+1  
5     return res
```

What is the form of the function $f(\text{factorial}, n)$?

$$f(\text{factorial}, n) = \text{\#operations}(n) = 5n + 2$$

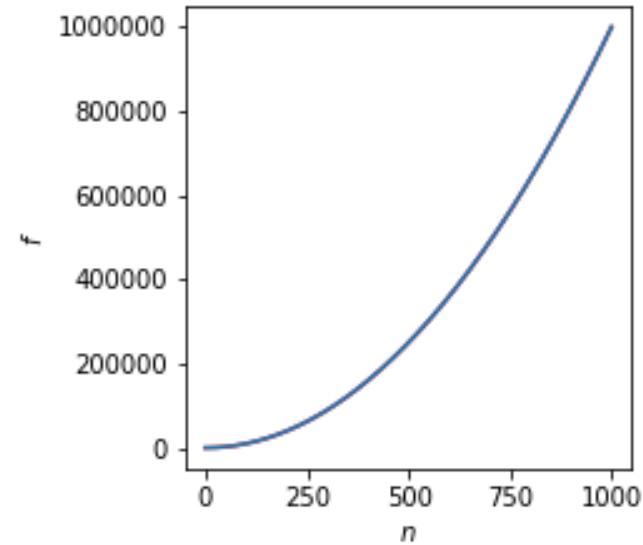
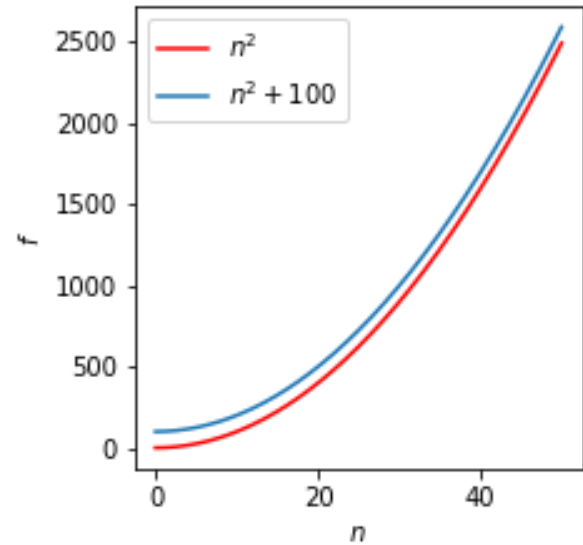
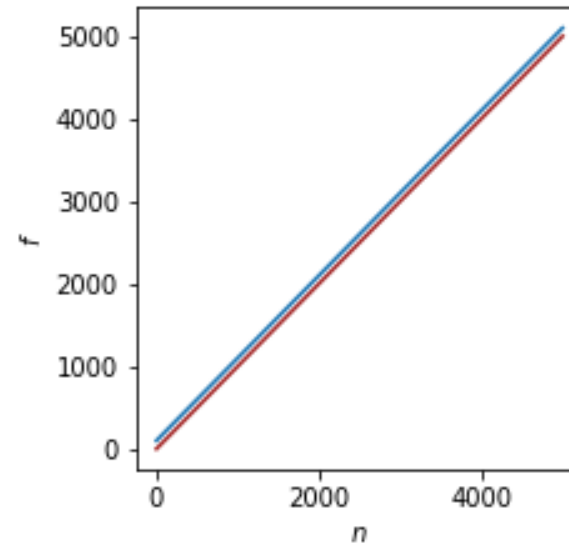
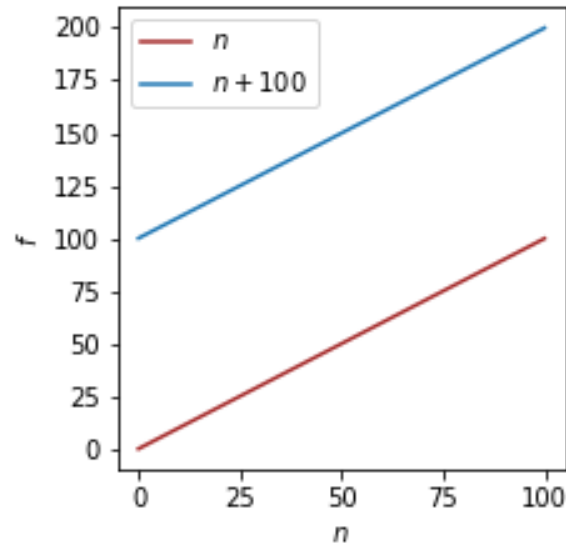
- $f(10) = 50 + 2$
- $f(100) = 500 + 2$
- $f(1000) = 5000 + 2$
- $f(1000000) = 5000000 + 2$

As n grows the additive constant 2 becomes *irrelevant*

- Since in time complexity analysis we are interested in what happens for **large n** , in the limit of $n \rightarrow \infty$, additive constants can be ignored in the expression of f

$$f(\text{factorial}, n) = 5n$$

Irrelevance of additive constants for large n



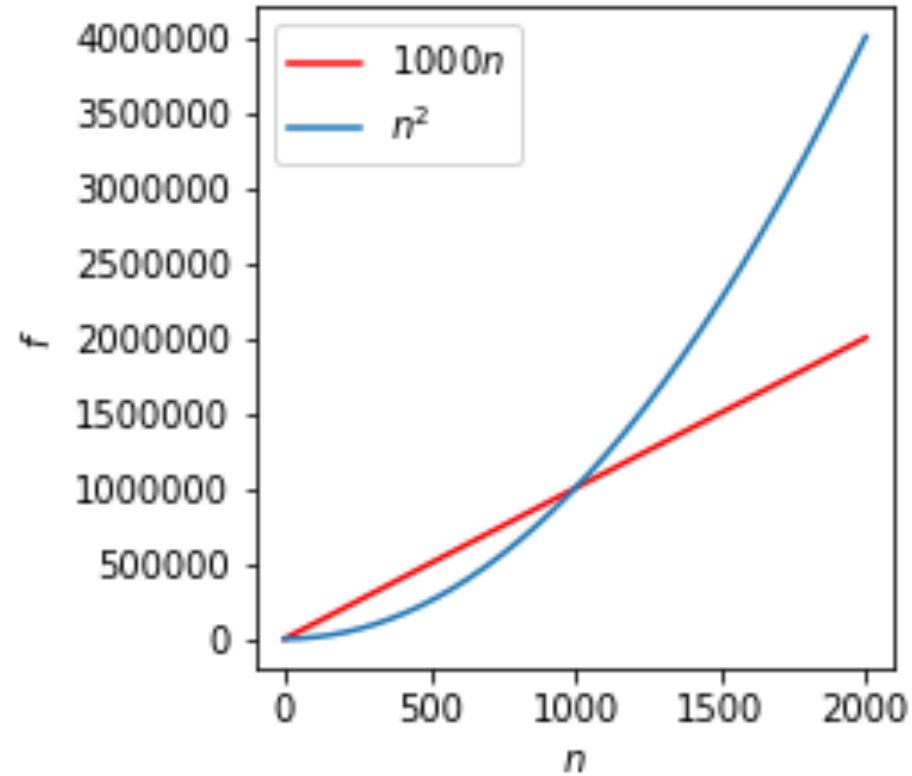
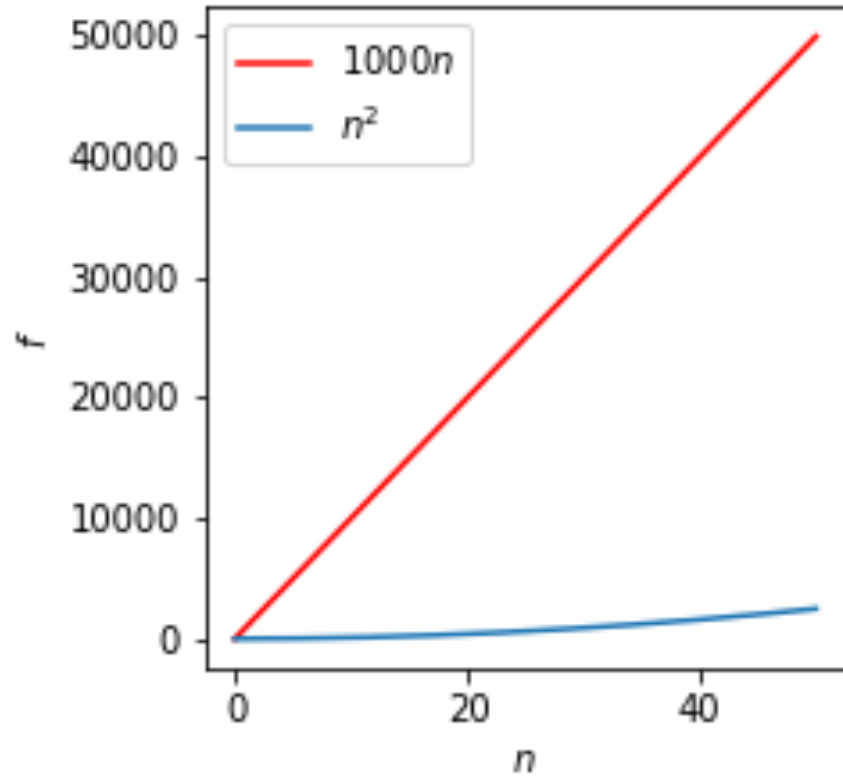
Worst-case time complexity function and multiplicative factors

$$f(\text{factorial}, n) = 5n$$

Is that multiplicative factor 5 important?

- Per se, it is, since an if, let's say, the actual (average) running time for an operation is 1ms, for $n = 10^6$, it would take about 10^3 seconds to complete for an algorithm with a time complexity function $f(n) = n$, while it would take 10^6 seconds for a function $f(n) = 1000n$
- However, would that multiplicative factor play a role when comparing, let's say, $f(\text{alg1}, n) = 1000n$ with. $f(\text{alg2}, n) = 6n^2$?
- For instance, for $n = 10^6$, $f(\text{alg1}, n) = 10^9$ while $f(\text{alg2}, n) = 6 \cdot 10^{12}$! There's no game, algorithm 1 is way better than algorithm 2
- Multiplicative factors do not really matter for large n when it comes comparing algorithms with different degrees of growth (i.e., different time complexity functions f)

Irrelevance of multiplicative factors when *comparing* algorithms for large n



Asymptotic behavior

- ✓ For small input sizes n almost *any algorithm is sufficiently efficient*, difference aren't macroscopic
- ✓ We are interested comparing the running times of different algorithms for **very large inputs n**
- ✓ ... and we interested comparing the running times under the assumption **of worst-case** scenarios ...
 - Mathematically, very large means: $n \rightarrow \infty$
 - **Asymptotic behavior** of the running time of an algorithm

Asymptotic behavior

```
1 def f(n):
2     '''Input n is an int > 0'''
3     add = 0
4
5     #Loop that takes constant time
6     for i in range(1000):
7         add += 1
8     print('Number of additions operations so far:', add)
9
10    #Loop that takes time n
11    for i in range(n):
12        add += 1
13    print('Number of additions operations so far:', add)
14
15    #Loop that takes time n**2
16    for i in range(n):
17        for j in range(n):
18            add += 1
19            add += 1
20    print('Number of additions operations so far:', add)
```

$$f(n) \approx 1000 + n + 2n^2$$

$$f(n) \approx 2n^2 \quad \text{as } n \rightarrow \infty$$

$$f(n) \approx n^2$$

dropping constants

```
1 f(10)
```

```
Number of additions operations so far: 1000
Number of additions operations so far: 1010
Number of additions operations so far: 1210
```

```
: 1 f(1000)
```

```
Number of additions operations so far: 1000
Number of additions operations so far: 2000
Number of additions operations so far: 2002000
```

```
: 1 f(10000)
```

```
Number of additions operations so far: 1000
Number of additions operations so far: 11000
Number of additions operations so far: 200011000
```

- As we increase n , the relative impact of the first two loops on the running time becomes negligible
- The **dominant term** in the expression of $f(n)$ is the quadratic one!

Asymptotic behavior and Big O notation

➤ **Rules of thumb** in describing the asymptotic complexity of an algorithm:

- ✓ If the running time is the *sum of multiple terms*, keep the one with the **largest growth rate** and drop the others, since they will not have an impact for large n
- ✓ If the remaining term is a product, drop any *multiplicative constants*

$$f(n) \approx 1000 + n + 2n^2 \mapsto f(n) \approx n^2$$

$$f(n) \approx 2^n + 2^{n+1} \mapsto f(n) \approx 2^{n+1}$$

$$f(n) \approx 2 + 400n + 2^n \mapsto f(n) \approx 2^n$$

$$f(n) \approx n + 500 \log n \mapsto f(n) \approx n$$

$$f(n) \approx 2\sqrt{n} + 500 \log n \mapsto f(n) \approx \sqrt{n}$$

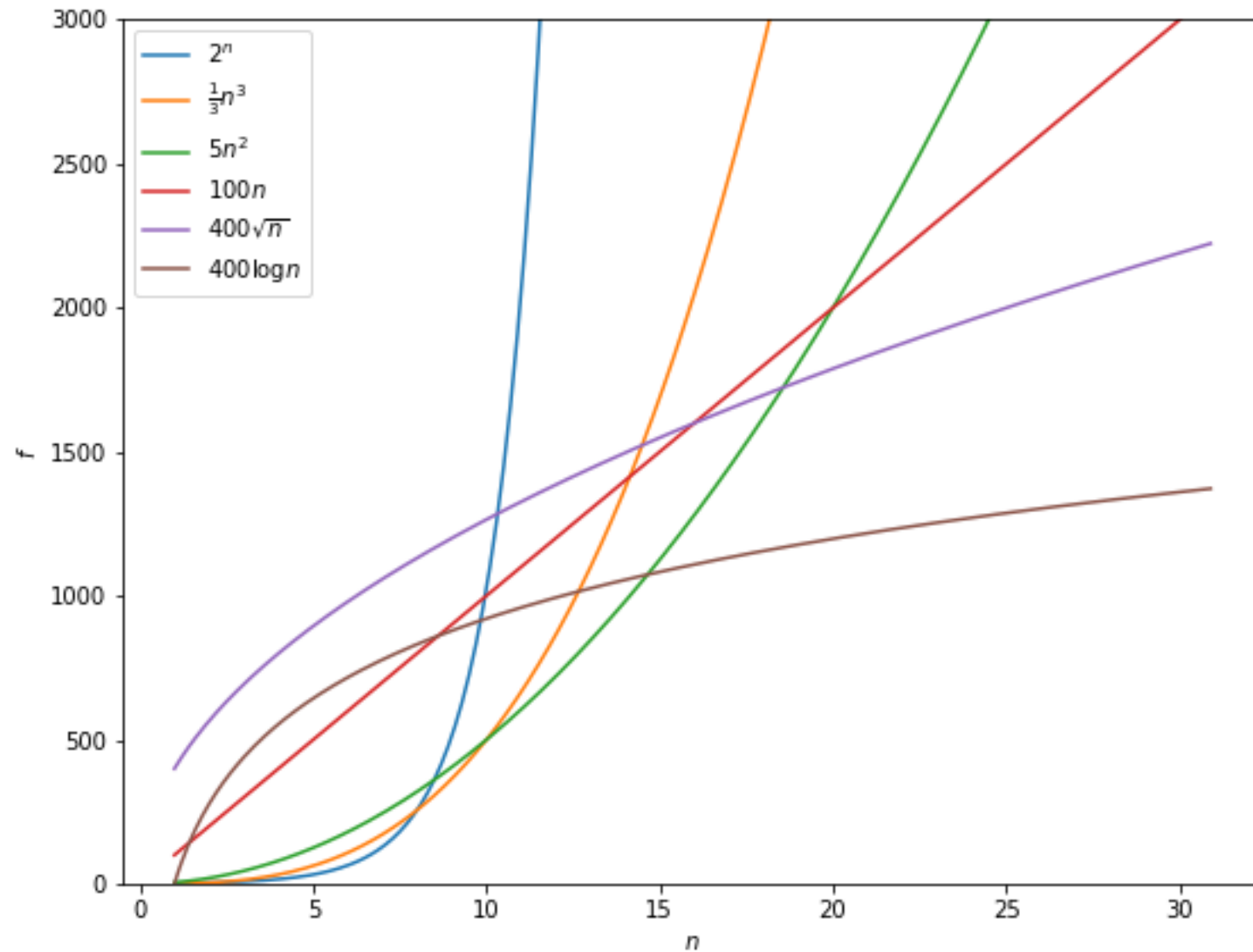
$$f(n) \approx \sqrt{n} + n + n^3 \mapsto f(n) \approx n^3$$

Asymptotic behavior and Big O notation

- The **Big O** notation is used to capture the **asymptotic order of growth** of the running time of an algorithm depending on the size of its input
- More precisely, and more in general, the Big O notation is used to give an **upper bound on the asymptotic growth of a function $f(n)$**
- E.g., $O(n^2)$ means that the running time complexity of the algorithm is **bounded** by a quadratic growth, while $O(2^n)$ means that the running time complexity is bounded by an exponential growth (bad!)
- If the asymptotic time complexity is described by $O(n^2)$, this means that the actual $f(n)$ can take any form such as:
 - $f(n) = 3n^2 + 4$
 - $f(n) = n^2 + 1000$
 - $f(n) = n^2 + 5n + 2000$
 - $f(n) = 900n^2 + 1000n + 5000000$

Any of these will always have an asymptotic growth less than a cubic growth, $O(n^3)$

Comparison among different growth rates



Common classes of complexity

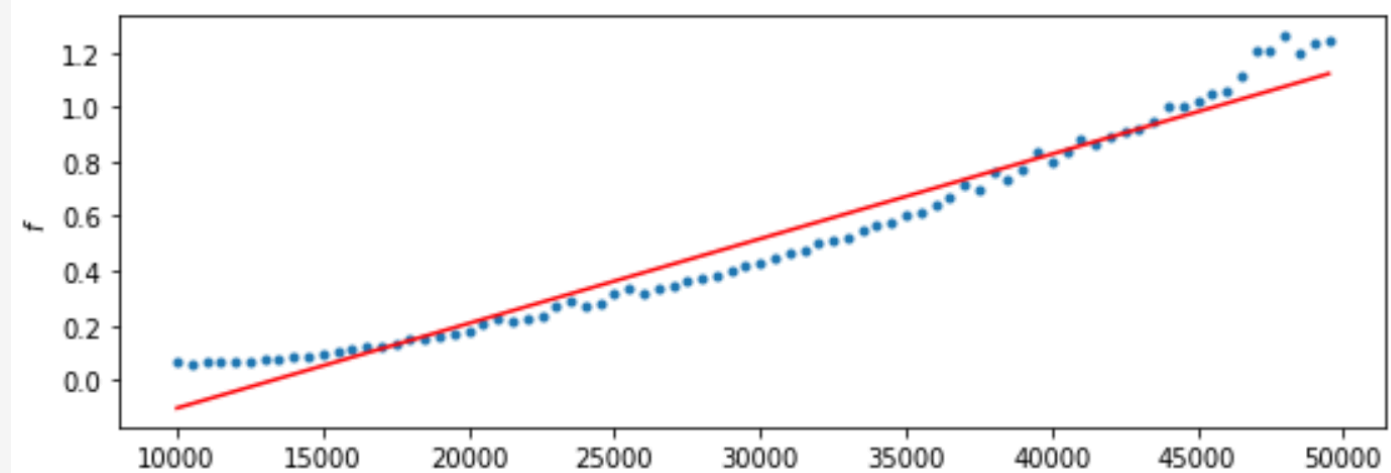
- $O(1)$: **Constant** running time
- $O(n)$: **Linear** running time
- $O(\log n)$: **Logarithmic** running time
- $O(n \log n)$: **Log-Linear** running time
- $O(n^k)$: **Polynomial** running time
- $O(c^n)$: **Exponential** running time
- $O(n!)$: **Factorial** running time

Common classes of complexity

- $O(1)$: constant (e.g. if $f(n) = 4$ then f is $O(1)$). This is the fastest you can get.
- $O(\log n)$: logarithmic
- $O(\sqrt{n})$: square root
- $O(n)$: linear (e.g. if $f(n) = 10n + 2$ then f is $O(n)$). This is generally still considered good.
- $O(n \log n)$: linearithmic, loglinear or quasilinear
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(n^k)$: polynomial
- $O(k^n)$: exponential (e.g. if $f(n) = 2^{4n}$ then f is $O(2^n)$). This is generally considered slow (although we *can* get much worse!).
- $O(n!)$: factorial. Algorithms falling in this category cannot usually be used in practice for really big n .

Numerical study of the time complexity for a linear growth

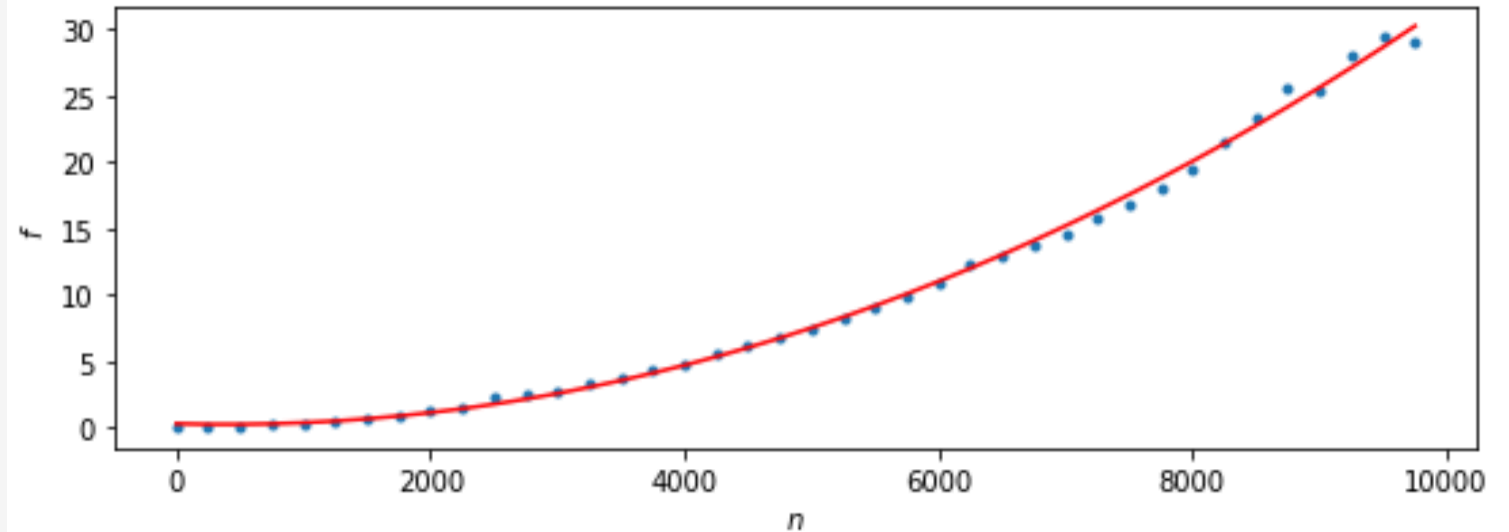
```
1 import time
2 import scipy.optimize as optimization
3
4 n= 50000
5 times = []
6 inputs = []
7 for i in range(10000, n, 500):
8     start = time.process_time()
9     factorial(i)
10    end = time.process_time()
11    times += [end-start]
12    inputs += [i]
13
14 plt.figure(figsize=(9,3))
15 plt.xlabel('$n$')
16 plt.ylabel('$f$')
17
18 plt.scatter(inputs, times, marker='.')
19
20 # fit the data with a linear function
21 params, cov = optimization.curve_fit(linear_func, inputs, times)
22 linear_fit = []
23 for x in inputs:
24     linear_fit += [linear_func(x, params[0], params[1])]
25 plt.plot(inputs, linear_fit, color='red')
```



```
1 def linear_func(x, a, b):
2     return a*x + b
```

Numerical study of the time complexity for a quadratic growth

```
1 import time
2 n= 10000
3 times = []
4 inputs = []
5
6 for n in range(1, n, 250):
7     x = 0
8     start = time.process_time()
9     for i in range(n):
10         for j in range(n):
11             x += i*j
12         end = time.process_time()
13         times += [end-start]
14         inputs += [n]
15
16 plt.figure(figsize=(9,3))
17 plt.scatter(inputs, times, marker='.')
18
19 plt.xlabel('$n$')
20 plt.ylabel('$f$')
21
22 # fit the data with a quadratic function
23 params, cov = optimization.curve_fit(poly2_func, inputs, times)
24 quadratic_fit = []
25 for x in inputs:
26     quadratic_fit += [poly2_func(x, params[0], params[1], params[2])]
27 plt.plot(inputs, quadratic_fit, color='red')
```



```
4 def poly2_func(x, a, b, c):
5     return a*(x**2) + b*x + c
```

Complexity of some common Python functions

Function	Input size	Complexity
<code>L.append(x)</code>	—	$O(1)$
<code>L.sort()</code>	length of L	$O(n \log n)$
<code>min(L)</code>	length of L	$O(n)$
<code>max(L)</code>	length of L	$O(n)$
<code>len(L)</code>	—	$O(1)$