1. Efficiency of Algorithms

Programming and Algorithms II

Degree in Bioinformatics

Fall 2018

What is a Good Program?

External:

- Correct – does what it’s supposed to do
- Efficient – uses few resources
  - Time, memory, bandwidth, disk…
- Robust – does not explote if something not quite right
- Easy to install, to use, to learn
- Goes along with other programs
- Works in many environments

- Internal
  - Easy to understand its code
  - Easy to modify - to do something different
  - Parts can be reused

Efficient – uses few resources

What is “to run fast”?

1 sec? 1 minute? 1 hour? 1 week?

That depends on machine, interpreter, O.S.….

Also, we expect larger inputs to take more time

Efficient – uses few resources

Proposal:

Efficient = “it scales well to larger data”

Orders of magnitude

Let f be a function from N to N.

"Big-Oh" notation

"g is in the order of f" if

$$g \in O(f) \quad \text{iff} \quad \lim_{n \to \infty} \left(\frac{g(n)}{f(n)} \right) < \infty$$

Equivalently:

$$O(f) = \{ g : \exists c \ \exists m \ \forall n > m \ \ g(n) < c * f(n) \}$$

Orders of magnitude

$3n^2 \in O(0.01\, n^2)$

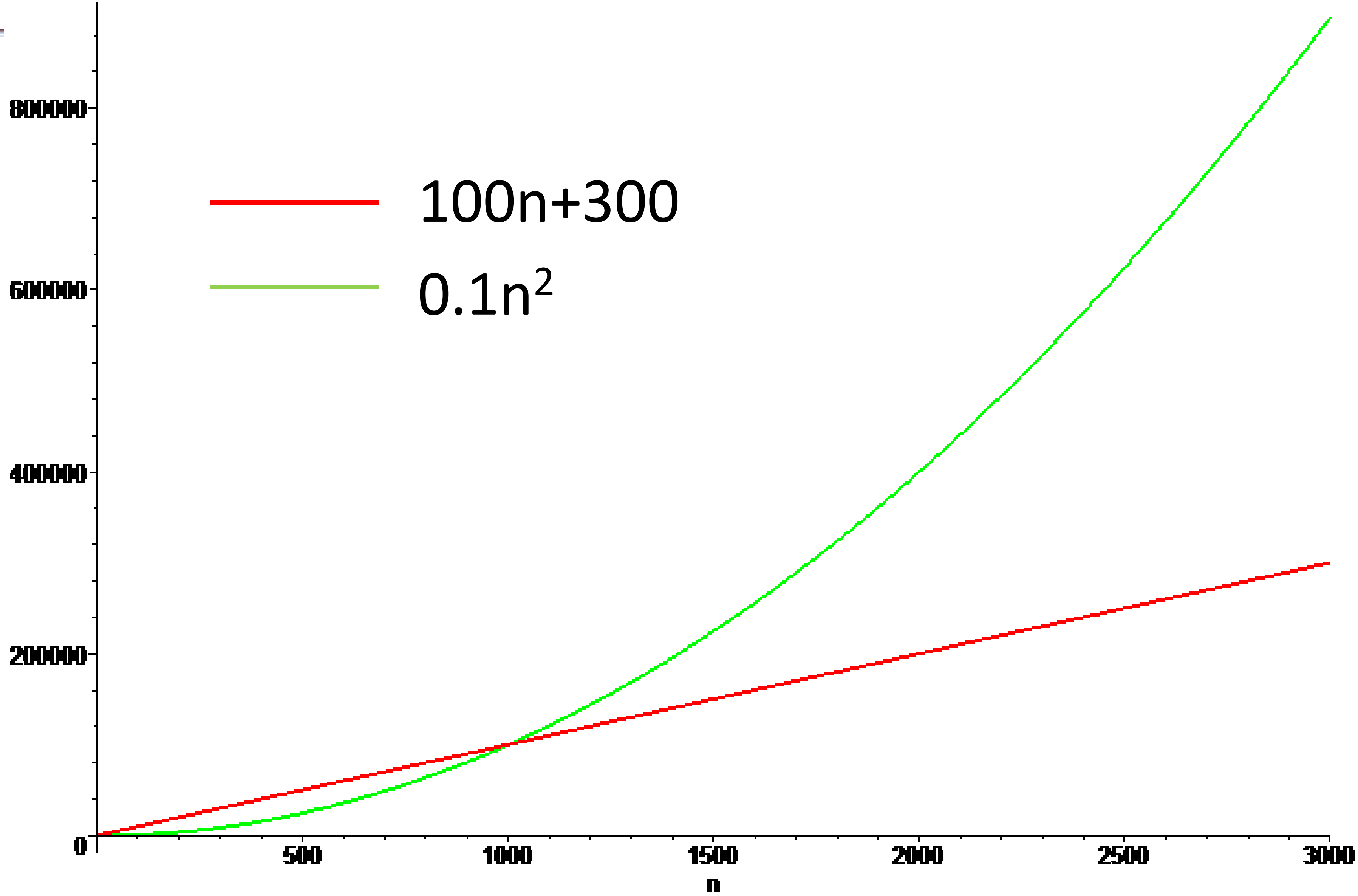
$1000\, n \in O(0.01\, n^2)$

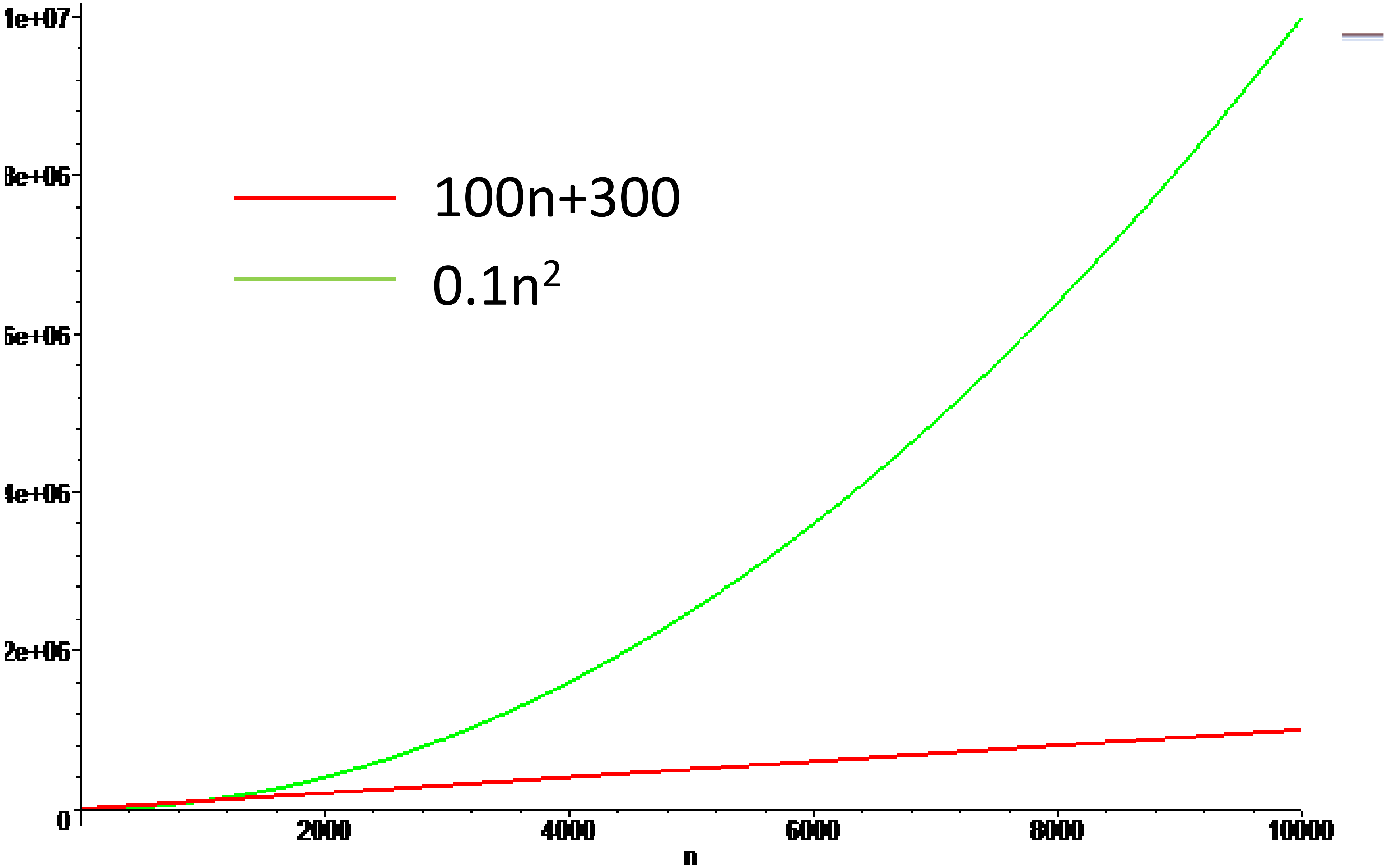
$3n^2 + 100\, n + 1000 \in O(n^2)$

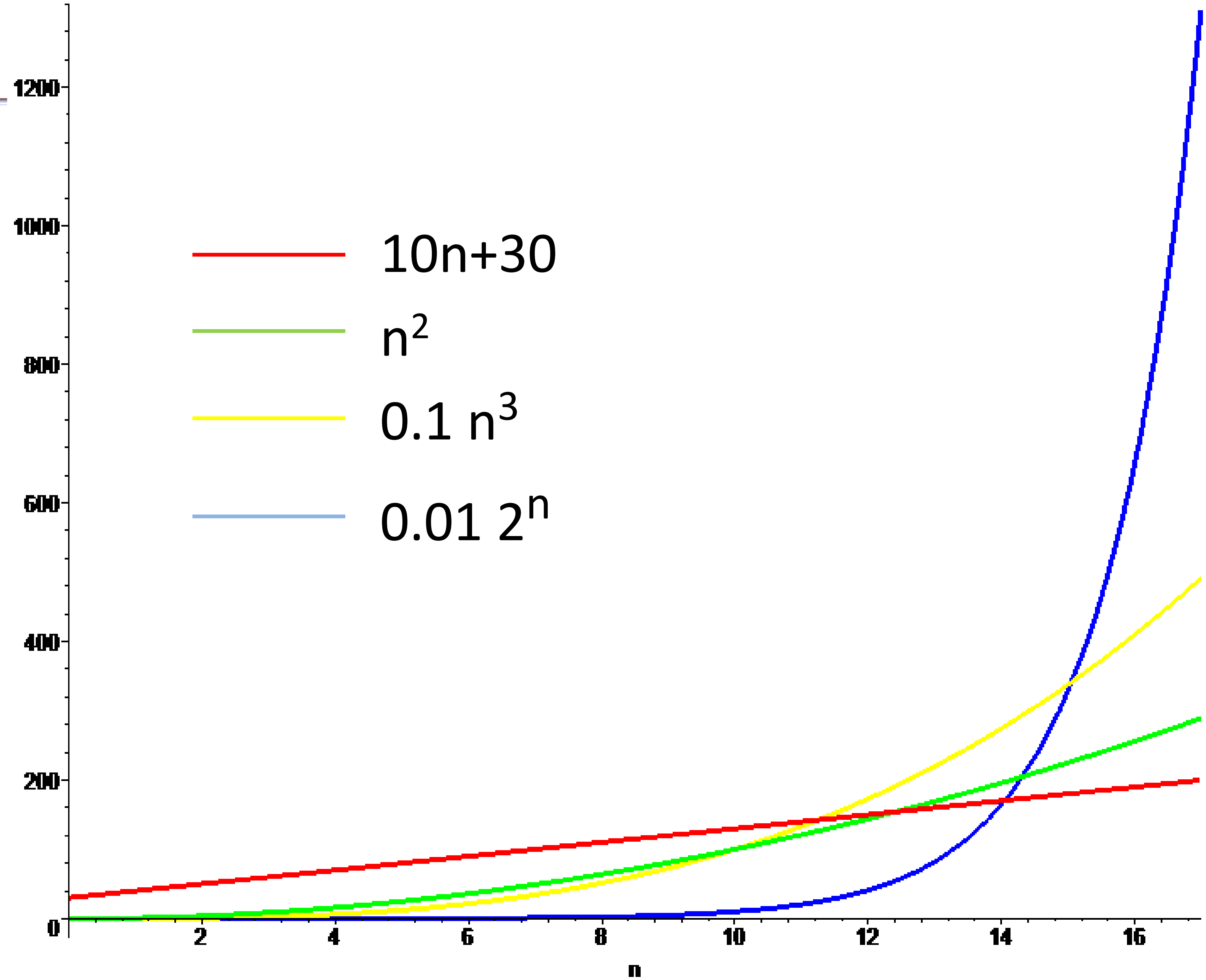
$20 \in O(1)$

It is wrong, but we write

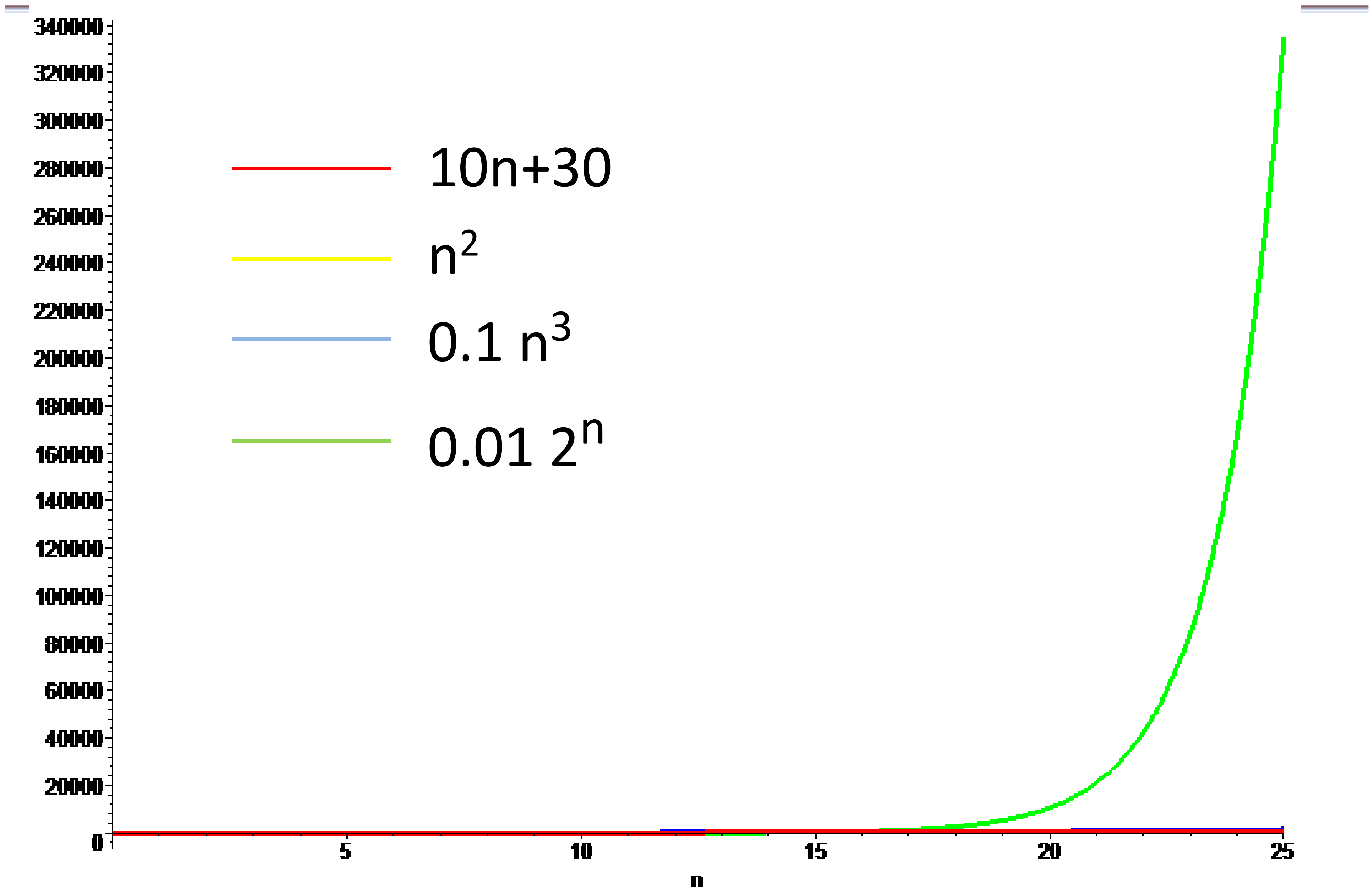
$$3n^2 + 10\, n + 100 = O(n^2)$$


100n+300
0.1n^2
800000
600000
400000
200000
0
500
1000
1500
2000
2500
3000
n


100n+300
0.1n2
1e+07
8e+06
6e+06
4e+06
2e+06
0
2000
4000
6000
8000
10000
n



1200
1000
800
600
400
200
0
2
4
6
8
10
12
14
16
n
10n+30
n^2
$0.1\ n^3$
$0.01\ 2^n$


340000
320000
300000
280000
260000
240000
220000
200000
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
5
10
15
20
25
n
10n+30
n2
0.1 n3
0.01 2n

Common orders of magnitude

1

$\log(n)$

$n^{1/2}$

n

$n \log(n)$

n^2

n^3

2^n

3^n

$n!$ Stirling's approximation $\approx (2\pi n)^{1/2} (n/e)^n$

Orders of magnitude: Scaling

Scaling

Linear:

$f = O(n)$: $f(2n) \approx 2\ f(n)$ $f(10n) \approx 10\ f(n)$

Quadratic

$f = O(n^2)$: $f(2n) \approx 4\ f(n)$ $f(10n) \approx 100\ f(n)$

Exponential

$f = O(2^n)$: $f(n+1) \approx 2\ f(n)$ $f(n+10) \approx 1000\ f(n)$

Worst-case analysis

We take the worst-case input of every size

We say

“the running time of algorithm A is O(n)”

if the function
  n -> max{ running time of A(x) : x input of size n }
is O(n)

Bounding running time of programs

time(simple expression) =
    "number of elementary instructions executed"

time(`I1 then I2`) = time(`I1`) + time(`I2`)

time(`if(cond): I1 else: I2`) <=
    time(`cond`) + max(time(`I1`),time(`I2`))

time(`f(arg1,arg2,arg3)`) =
    time(`arg1`) + time(`arg2`) + time(`arg3`) +
    + time(`f(val1,val2,val3)`)

Bounding running time of programs

time(while cond: body) =

$$\sum_i \text{time(cond\_i)} + \text{time(body\_i)}$$

where i ranges over the iteration number and cond_i and body_i are the executions of cond and body in the ith iterations

If all iterations take roughly the same time, a good bound is:

time(while cond: body) <=

(number of iterations) * (time(cond) + time(body))

Example 1

```
a = int(input("a: "))
b = int(input("b: "))
x = 3*a + 10
y = 9*b - 5
if (x > y):
     theMax = x
else:
     theMax = y
print(theMax)
```

O(1)

Example 2

- Given a natural number n, print all natural numbers that divide n exactly
- E.g., for 12 print 1, 2, 3, 4, 6, 12

```
for i in range(1,n+1):
     if (n % i == 0): print(i)
```

- Value vs. number of digits
- Time ≈ $10^{\text{size of the input}}$

Example 3

```
s = 0
for x in lst:
    s += x
```

O(len(lst))

```
s=0
for i in range(0,len(lst)):
    s += lst[i]
```

O(len(lst))

```
print(sum(lst))
```

O(len(lst))

```
print(sum(lst[i:j]))
```

O(j-i)

Python lists (n=len(lst))

Operation	Call	Cost
creating	lst = [v1,v2,…,vn]	O(n)
access by position	lst[i]	O(1)
append at end	lst.append(x) or lst.extend(lst2)	O(1) (on average…) O(len(lst2))
remove from end	lst.pop()	O(1) (on average…)
Insert at position	lst.insert(pos,x)	O(n)
delete at position	lst.pop(pos)	O(n)
count occurrences	l.count(x)	O(n)
make empty	l.clear()	O(1)
reverse	lst.reverse()	O(n)
copy	lst.copy()	O(n)
sort	lst.sort()	O(n log n)
Iterate / membershp	for x in lst x in/not in lst	O(n)
copy slice	lst[a:b]	O(b-a)
check equality	lst1 == lst2	O(n)

Sorting a list

Given:

- a binary comparison operator “<“

(with what we expect from “<“… details omitted)

- a list whose elements are comparable with <

Do: permute the elements in L so that they are increasingly ordered w.r.t. <

[5,2,9,0,4] → [0,2,4,5,9]

[“dog”,”fish”,”ant”,”frog”] → [“ant”,”dog”,”fish”,”frog”]

Sorting a list

General rule: Don't write your sorting algorithm

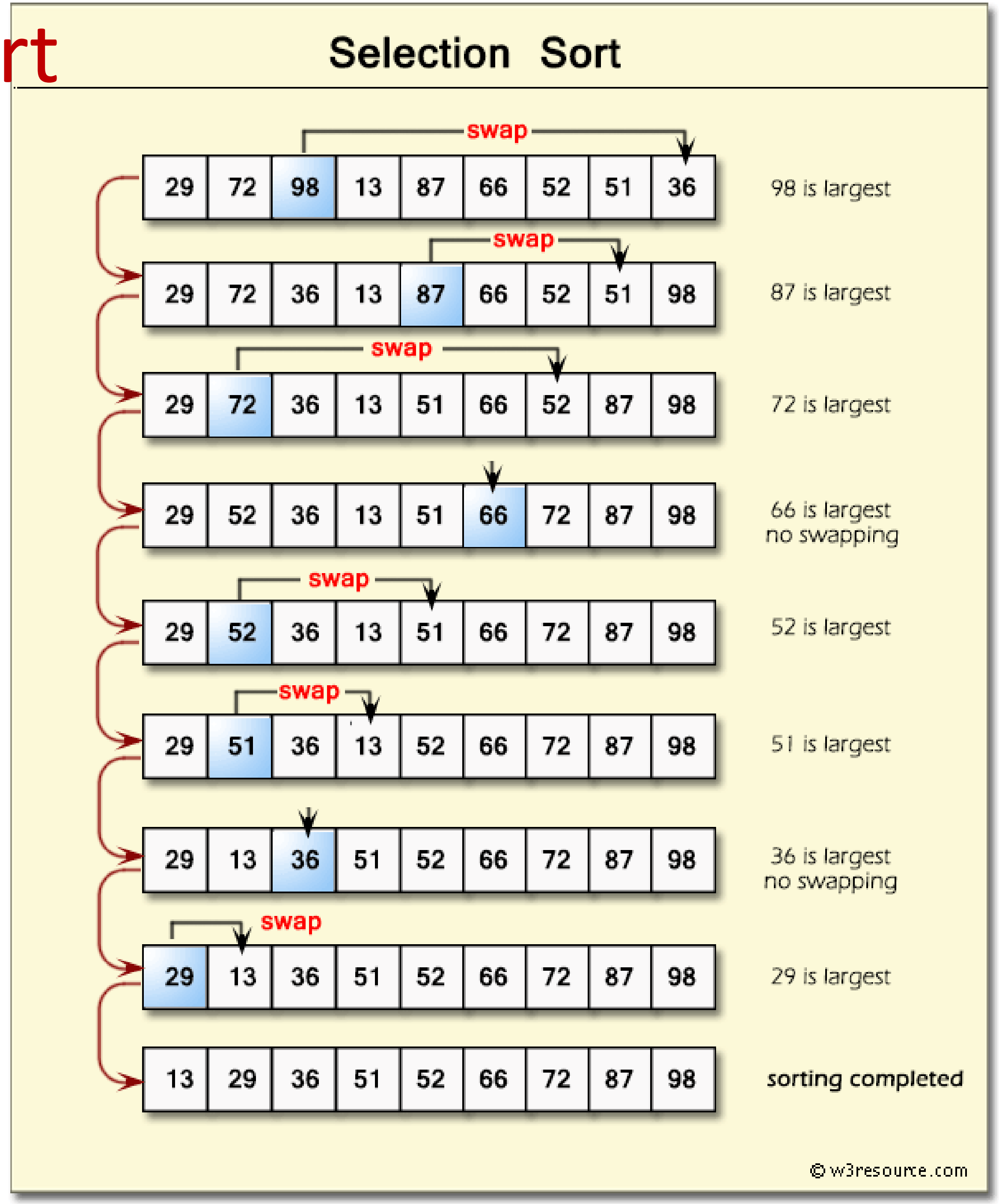
Almost always: list.sort()

But we'll see some sorting algorithms:

- Selection
- Insertion
- Mergesort
- Quicksort
- ~~Bubblesort~~ just don't!

Selection sort

https://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-search-and-sorting-exercise-5.phpA LIST



Selection sort

```
def selection_sort(lst):
   for i in range(0,len(lst)-1):
       pos_min = i
       for pos in range(i+1,len(lst)):
           if lst[pos]<lst[pos_min]:
               pos_min = pos
       lst[i],lst[pos_min] = lst[pos_min],lst[i]
```

Selection sort

```
def selection_sort(lst):
    for i in range(0,len(lst)-1):
        pos_min = i
        for pos in range(i+1,len(lst)):
            if lst[pos]<lst[pos_min]:
                pos_min = pos
        lst[i],lst[pos_min] = lst[pos_min],lst[i]
```

Cost O(len(lst)-i)

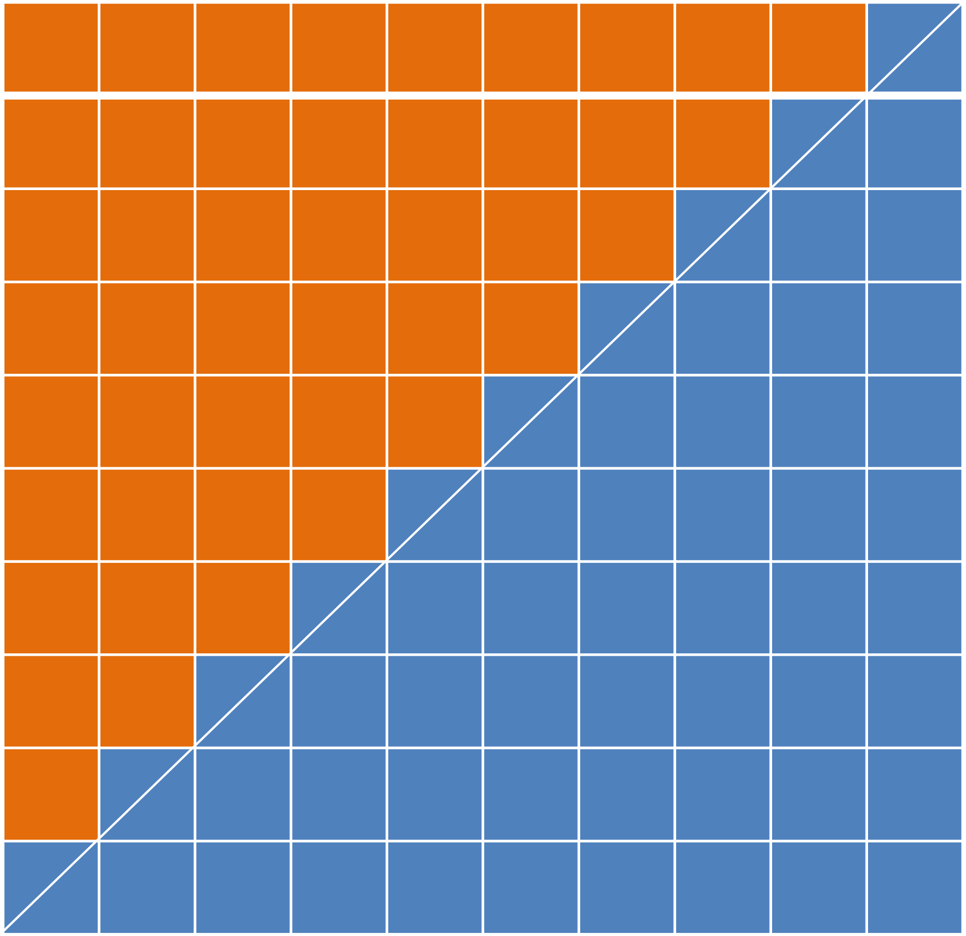
If we call N=len(lst), cost is

$$O((N-2)+(N-3)+(N-4)+\ldots+3+2+1)$$

Selection sort: Running time

$$1 + 2 + \ldots + (N-1) + N \quad = N(N+1)/2$$

- By induction
- Or geometrically



Selection sort: Running time

Selection sort on a list of length N

- Running time $O(N^2)$

- 3(N-1) element <u>assignments</u>

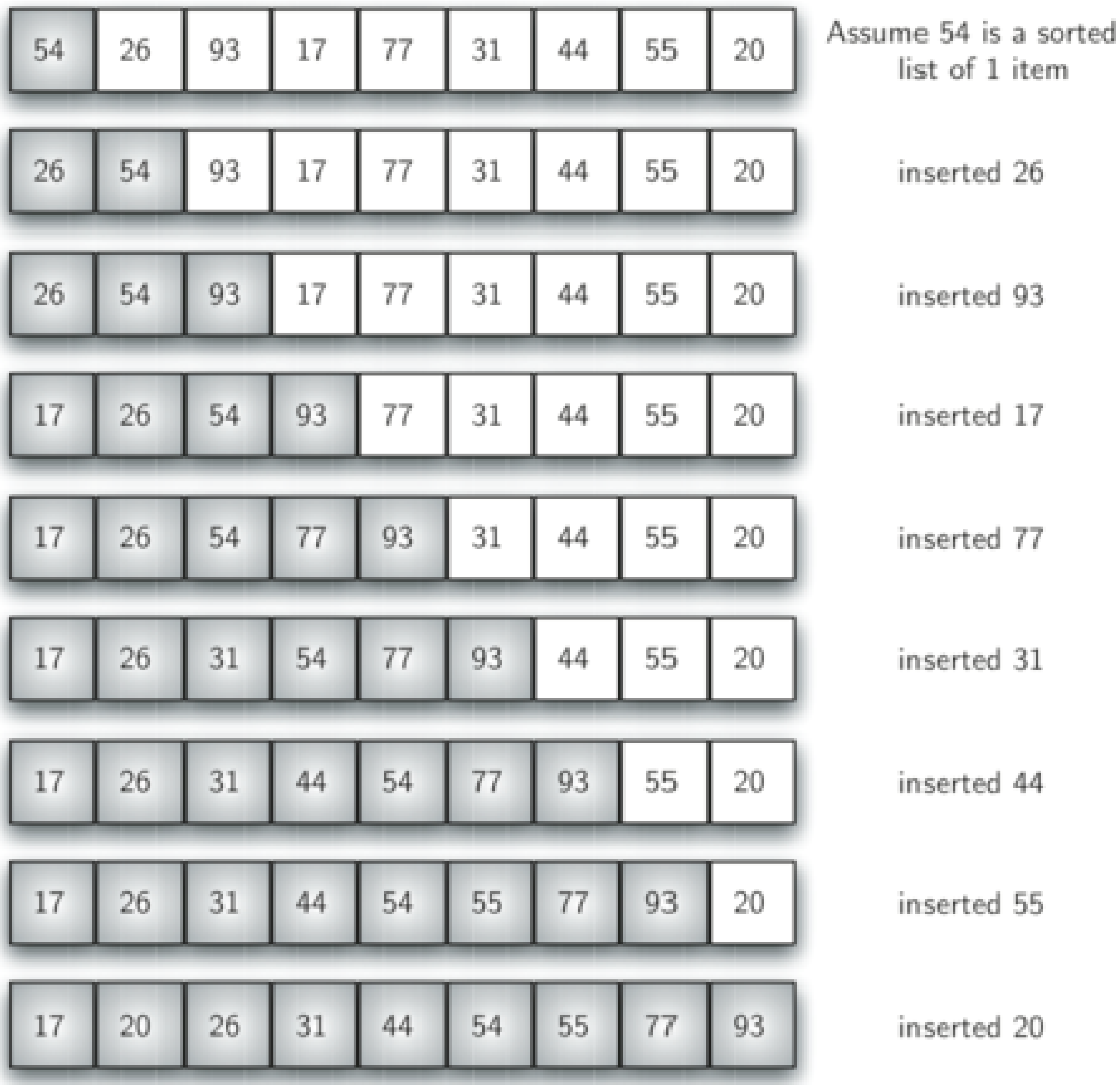
- N(N-1)/2 element <u>comparisons</u>

Selection sort

- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Insertion sort

http://interactivepython.org/courselib/static/pythonds/SortSearch/TheInsertionSort.html



Insertion sort

```
def insertion_sort(lst):
   for i in range(1,len(lst)):
      x = lst[i]
      pos = i
      while pos > 0 and lst[pos-1] > x:
         lst[pos] = lst[pos-1]
         pos = pos-1
      lst[pos] = x
```

If we call N=len(lst), cost is

$$O(1+2+3+\ldots+(N-1)+(N-2)+(N-1)) = O(N^2)$$

Insertion sort

Cost O(i) – at most

```
def insertion_sort(lst):
    for i in range(1,len(lst)):
      x = lst[i]
      pos = i
      while pos > 0 and lst[pos-1] > x:
          lst[pos] = lst[pos-1]
          pos = pos-1
      lst[pos] = x
```

If we call N=len(lst), cost is

$$O(1+2+3+\ldots+(N-1)+(N-2)+(N-1)) = O(N^2)$$

Insertion sort

- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Insertion sort: Running time

Potentially $O(N^2)$ movements and comparisons

- If in reverse order

But faster when elements near their place (almost sorted): short inner loop!

Stable: “Equal” elements (neither < nor >) remain in the same order as they were

To remember

Selection sort:

- + Easy to remember
- + O(N) movements
- - Not stable

Insertion sort:

- - Potentially $O(N^2)$ movements
- + Faster if most elements near their place
- + But usually faster than selection sort
- + Stable

(note: in Python, “movement” = reference change, cheap;
In othr languages, “moving” an element can mean copy, expensive)

To remember

O(…) useful to discuss the running time of algorithms as data gets large

If two algorithms differ in O(…), one is faster than the other for large enough inputs

We know two algorithms that sort lists in time $O(n^2)$
Later we will see faster algorithms with time $O(n \log n)$

Some links

- Sorting algorithms

https://www.youtube.com/watch?v=h-QYzgTmgVI

- Selection sort folk dance:

https://www.youtube.com/watch?v=Ns4TPTC8whw

- Insertion sort folk dance:

https://www.youtube.com/watch?v=ROalU379l3U