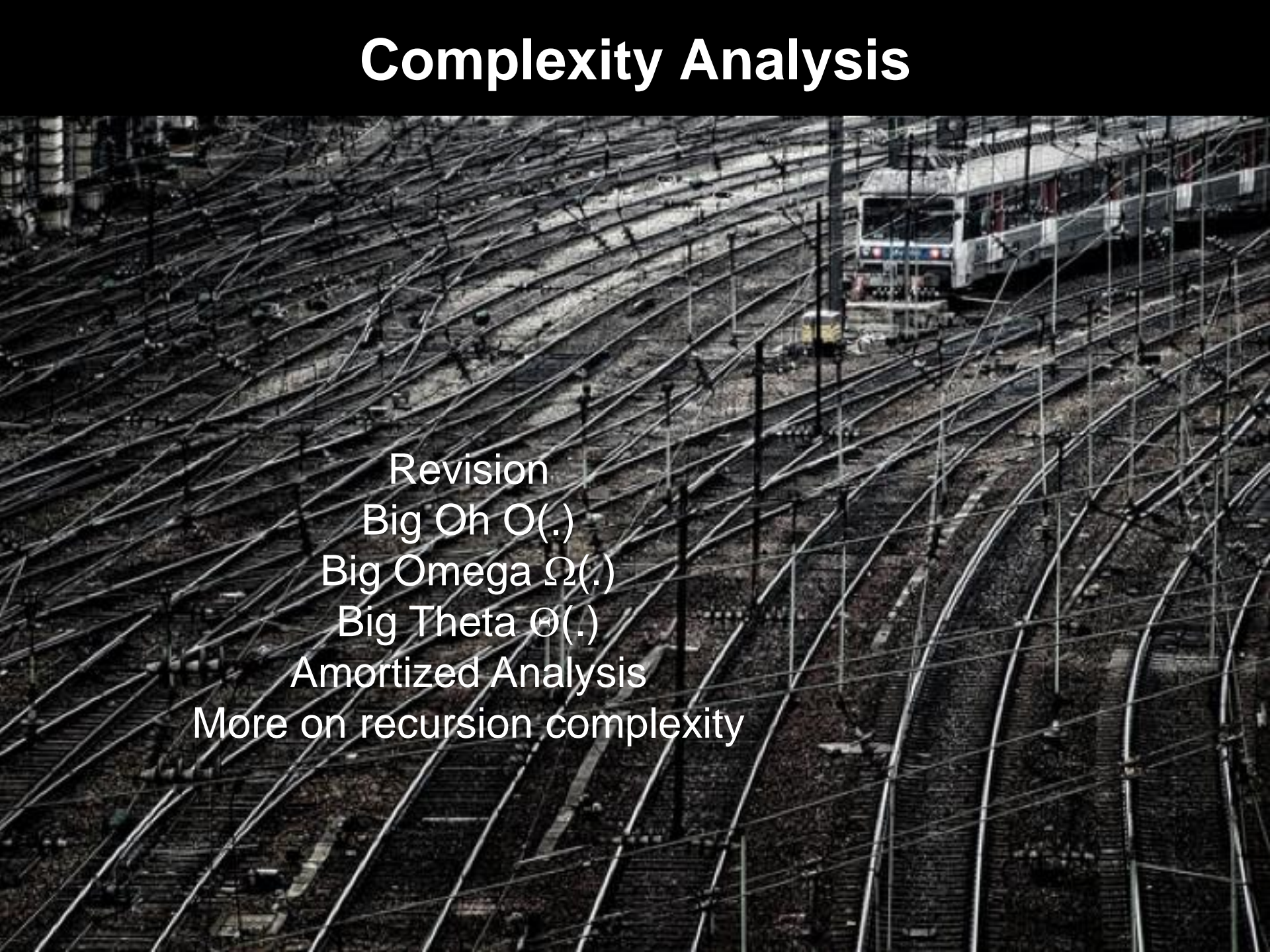


# Complexity Analysis



Revision  
Big Oh  $O(\cdot)$   
Big Omega  $\Omega(\cdot)$   
Big Theta  $\Theta(\cdot)$   
Amortized Analysis  
More on recursion complexity

# Why worry about algorithm analysis?

programs that take too long to finish are useless

good programming skills will not compensate for poor algorithms

we need to understand how much work our algorithms will require for different inputs

- learn to avoid bad or inefficient design patterns

we also need to understand the limits – some problems can't be solved without at least a minimum amount of work

- don't waste time trying to design something that is impossible to achieve

# Complexity basics

we measure complexity in terms of the number of basic steps – i.e. quick constant-time operations:

- reading a value, assigning a value, comparing two values, simple arithmetic operations (+, -, \*, /), calling a function, returning a value, ...

our main concern is the *worst-case* complexity

- we want to know how bad it could get, and use that as a performance guarantee
- complexity is based on the size of the input,  $n$  – how many items in a list, or in a tree, or rows in a file
- we are only concerned about large inputs
  - *asymptotic analysis*: as  $n \rightarrow \infty$ , how many steps as a function of  $n$ ?

# Big Oh notation

For two functions  $f$  and  $g$  operating on positive integers,  $f(n)$  is  $O(g(n))$  if there is an integer constant  $k \geq 0$  and a real constant  $C > 0$  so that for all  $n$  bigger than  $k$ ,

$$f(n) \leq C * g(n)$$

Informally, once  $n$  is big enough,  $f(n)$  is never worse than a constant multiple of  $g(n)$  (and so  $f$  is not significantly worse than  $g$ ).

We can think of  $O(g(n))$  as specifying a set of functions: all those functions that are not significantly worse than  $g$

# Standard function hierarchy

$$O(c) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \dots \subset O(2^n) \subset O(n!)$$

Note: the hierarchy is strict:

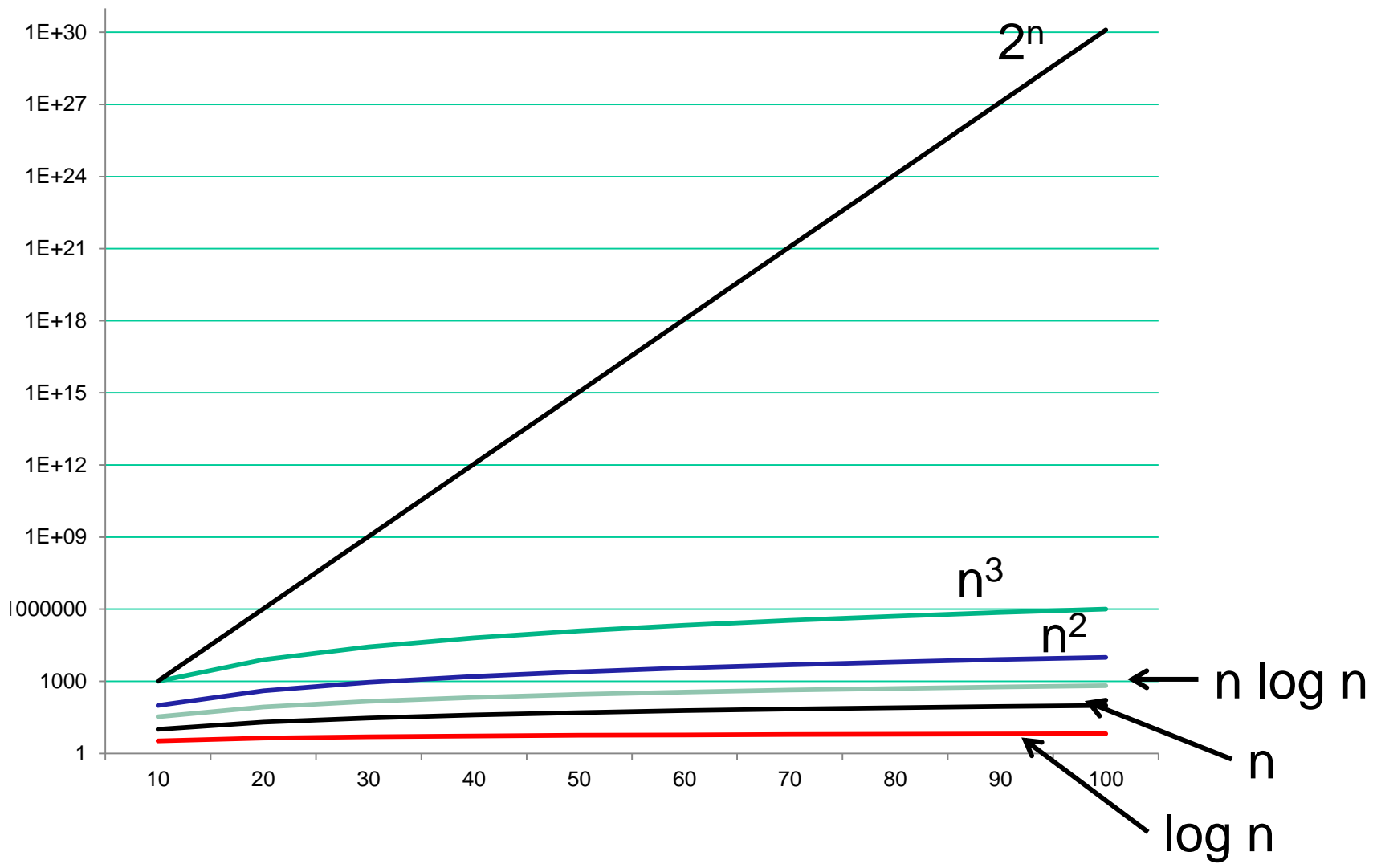
$$O(\log n) \subset O(n), \text{ but } O(n) \not\subset O(\log n)$$

Polynomials are classified by their highest degree:

$$n^2 + 3n + 5 \text{ is } O(n^2)$$

It is correct to say  $n^2 + 3n + 5$  is  $O(n^3)$ , but not as useful  
as saying  $n^2 + 3n + 5$  is  $O(n^2)$

Find the smallest simple function that works for  $O(\cdot)$



# Lower bounds: Big Omega

For two functions  $f$  and  $g$  operating on positive integers,

$$f(n) \text{ is } \Omega(g(n))$$

if there is an integer constant  $k \geq 0$  and a real constant  $C > 0$  so that for all  $n$  bigger than  $k$ ,

$$C * g(n) \leq f(n)$$

Once  $n$  is big enough,  $f(n)$  is never *better* than a constant multiple of  $g(n)$  (or  $f$  is not significantly better than  $g$ ).

# Exercise: Prove fibonacci(n) is $\Omega(3/2)^n$

Note: prove the *value* of fib(n) is  $\Omega(3/2)^n$ , not the work done by an algorithm



# Tight bounds: Big Theta

For two functions  $f$  and  $g$  operating on positive integers,

$$f(n) \text{ is } \Theta(g(n))$$

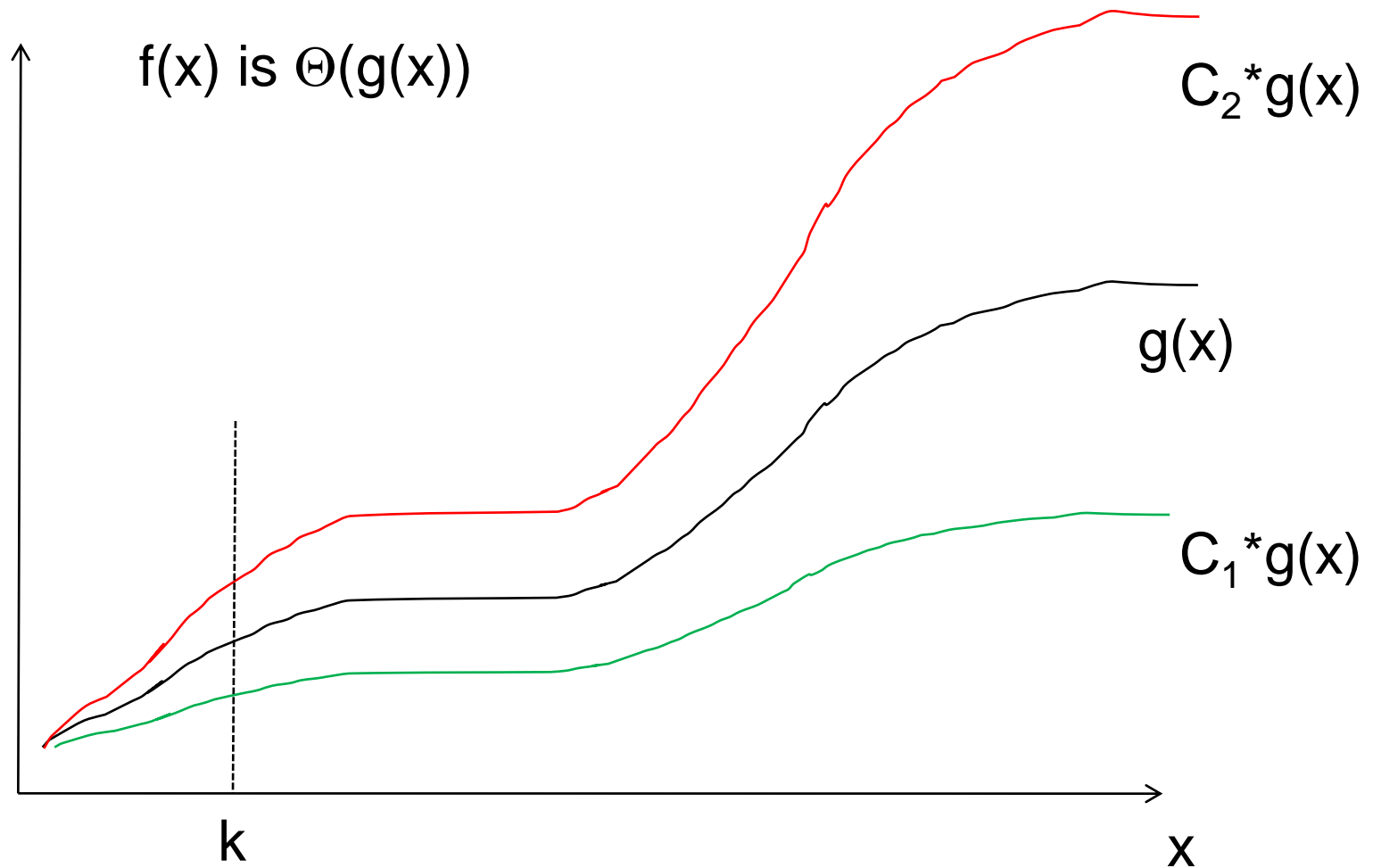
if there is an integer constant  $k \geq 0$  and two real constants  $C_1 > 0$  and  $C_2 > 0$  so that for all  $n$  bigger than  $k$ ,

$$C_1 * g(n) \leq f(n) \leq C_2 * g(n)$$

Once  $n$  is big enough,  $f(n)$  is never worse than some constant multiple of  $g(n)$ , *and also* never better than some constant multiple of  $g(n)$

*i.e*  $f$  is not significantly different from  $g$

# What does big-theta mean for function growth?



For all values of  $x > k$ ,  $f(x)$  will be between the red and green lines

# Example

$n^2 + 3n + 5$  is  $\Theta(n^2)$

# Complexity of list doubling

Policy: each time we must grow the list, double its size.  
 Suppose we are unlucky, and must copy the list each time we grow it.

To create a list that reaches size  $n$ , this will take

1 assignment +

1 copy op + 1 assignment +

2 copy ops + 2 assignments +

4 copy ops + 4 assignments +

...

$n/2$  copy ops +  $n/2$  assignments

$1+2+4+\dots+(n/2)$  copies +  $1+1+2+4+\dots+(n/2)$  assignments

$2^0 + 2^1 + 2^2 + \dots + 2^{\log(n)-1}$  copies +  $2^0 + 2^1 + 2^2 + \dots + 2^{\log(n)-1}$  assignments

$2^{\log(n)}$  copies +  $2^{\log(n)}$  assignments

$n$  copies +  $n$  assignments

Doubling the size each time space is needed takes  $O(n)$  to build a list of size  $n$ .

Average cost of a single append is then  $O(1)$

$$\sum_{i=0}^n 2^i = 2^{n+1} - 1$$

[www.google.ie/search](https://www.google.ie/search)

Amortize

[All](#)[Images](#)[News](#)[Books](#)[Videos](#)[More ▼](#)[Search tools](#)

About 475,000 results (0.43 seconds)

# amortize

/ə'mɔ:tɪz/

*verb*

gradually write off the initial cost of (an asset) over a period.

"the vessel's owners could not amortize her high capital costs"

- reduce or pay off (a debt) with regular payments.  
"eighty per cent of the proceeds has been used to amortize the public debt"
- *historical*  
transfer (land) to a corporation in mortmain.  
"lands amortized without licence"



Translations, word origin, and more definitions

# Amortized Analysis

The true cost to the CPU of appending to a Python list of size  $k$ :

- if there is space,  $c$  units of time to assign value to next cell
- if there is not space, then
  - $kc$  units of time to copy  $k$  values across to new list
  - $c$  units of time to assign the value

In the analysis, we will *charge* a fixed cost for each append no matter what space is available or what size the list is.

- for simple appends, this builds up a *profit*
- which we then spend on the complex appends

How much should we charge to ensure we never run at a loss (for big enough lists)?

- this value is the *amortized* complexity of the operation



Assume the fixed  $c$  units of time are worth €1.  
 Simple append is €1, complex append is €(k+1)  
 Let's charge €3 per append

Position	Cost	Charge
1	1+0	3
2	1+1	3
3	1+2	3
4	1	3
5	1+4	3
6	1	3
7	1	3
8	1	3
9	1+8	3

The 'charge' is the *amortized* cost of a single append: 3, which is  $O(1)$

When writing an amortized complexity, use a '\*'  
 i.e. cost of append is  $O(1)^*$

...

$$\begin{aligned}
 \text{Cost} &= n + 1 + 2 + 4 + \dots + (n-1) \\
 &= n + 2^0 + 2^1 + 2^2 + \dots + 2^{\log(n-1)} \\
 &= n + 2^{1+\log(n-1)} \\
 &= n + 2^*(n-1) \\
 &\leq 3n
 \end{aligned}$$

$$\text{Charge} = 3^*n$$



# Examples: Fibonacci

1	2	3	4	5	6	7	8	...
1	1	2	3	5	8	13	21	...

```
def fib1(n):  
    if n < 2:  
        return 1  
    elif n == 2:  
        return 1  
    return fib1(n-1) + fib1(n-2)
```

Binary recursion

2 comparisons

3 arithmetic ops

2 recursive calls per activation, but the 2<sup>nd</sup> is repeated inside the first.



n	fib1(n)	function calls	$2^{n/2}$
1	1	1	$\sqrt{2}$
2	1	1	2
3	2	$1+\#f2+\#f1 = 1+1+1 = 3$	$2\sqrt{2}$
4	3	$1+\#f3+\#f2 = 1+3+1 = 5$	4
5	5	$1+\#f4+\#f3 = 1+5+3 = 9$	$4\sqrt{2}$
6	8	$1+\#f5+\#f4 = 1+9+5 = 15$	8
7	13	$1+\#f6+\#f5 = 1+15+9 = 25$	$8\sqrt{2}$
8	21	$1+\#f7+\#f6 = 1+25+15 = 41$	16
9	34	$1+\#f8+\#f7 = 1+41+25 = 67$	$16\sqrt{2}$
10	55	$1+\#f9+\#f8 = 1+67+41 = 109$	32
...			

so number of function calls for  $f(n)$  is  $> 2^{n/2}$

So the algorithm fib1(n) is  $\Omega(2^{n/2})$

# Examples: efficient Fibonacci

Each call to `fib(n)` computes `fib(n-1)` also.  
We will return them both.

```
def fib(n):  
    (a,b) = _fib(n)  
    return a  
  
def _fib(n):  
    if n == 1:  
        return (1,0)  
    elif n == 2:  
        return (1,1)      # (f(2), f(1))  
    (a,b) = _fib(n-1)      # (f(n-1), f(n-2))  
    return (a + b, a)
```

Linear recursion  
 $n$  recursive calls  
 $O(n)$

# Exercise

$M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$  has the return values of `_fib(2)` and `_fib(1)` as rows

$$M^2 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \text{ which is } \begin{bmatrix} \textit{fib}(3) \\ \textit{fib}(2) \end{bmatrix}$$

$$M^3 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix} \text{ which is } \begin{bmatrix} \textit{fib}(4) \\ \textit{fib}(3) \end{bmatrix}$$

$$M^4 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix} \text{ which is } \begin{bmatrix} \textit{fib}(5) \\ \textit{fib}(4) \end{bmatrix}$$

...

Using this idea, and an example from lecture 2, can you find an algorithm for *fibonacci*(*n*) which takes  $O(\log n)$ ?

# Next Lecture

Sorting