Divide and conquer (i.e. - recursion)

https://xkcd.com/1270/
Subtitled: “Functional programming combines the flexibility and power of abstract mathematics with the intuitive clarity of abstract mathematics”.

Richard Lobb
Department of Computer Science and Software Engineering
University of Canterbury

Divide and Conquer (D&C)

(a.k.a. recursive programming)

D&C/recursion is a fundamental algorithm design technique:

1. Divide the problem into 2 or more sub-problems, each smaller instances of the original.
2. Solve ("conquer") the subproblems recursively.
   - If small enough, use a direct solution. The *base case*.
3. Combine the sub-solutions to get a solution to the original problem.

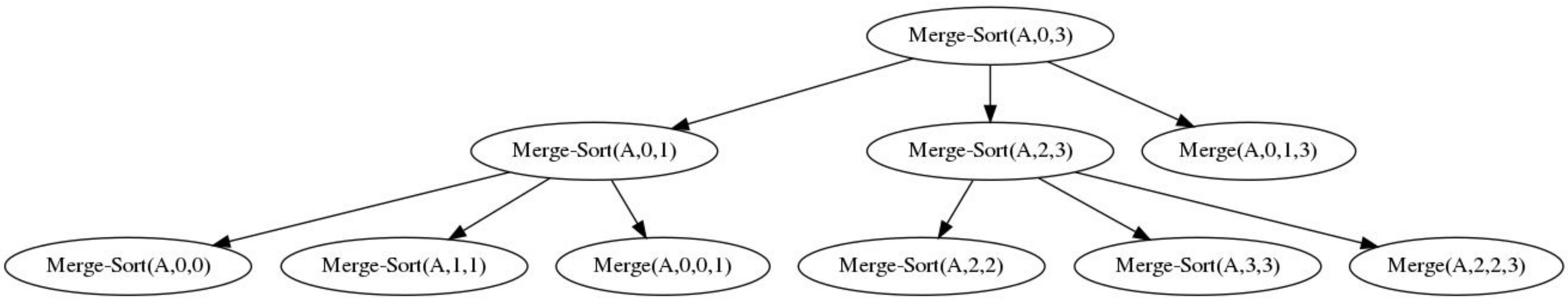
Example: Merge Sort

procedure Merge-Sort(A, left, right)
    if left < right
        mid $\leftarrow \left\lfloor \frac{left + right}{2} \right\rfloor$
        Merge-Sort(A, left, mid)
        Merge-Sort(A, mid + 1, right)
        Merge(A, left, mid, right)

Q1: What are three steps of D&C?

Q2: What is the base case?

Invocation tree ("call tree")



Q1: in what order are the calls initiated?

Q2: in what order are the calls completed?

Analysis of D&C algorithms

Time to solve base case = $\Theta(1)$

Time to solve all other cases =

Time-to-divide + Time-to-conquer + Time-to-combine

This leads to a recurrence equation (next slide).

Example: Merge Sort

Let $T(n)$ be the time to sort an array with n elements.

Time-to-divide $= k = \Theta(1)$ where k is some constant

Time-to-merge $= c\, n = \Theta(n)$ where c is another constant

So

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ \Theta(1) + 2T(\frac{n}{2}) + \Theta(n) & \text{if } n > 1. \end{cases}$$

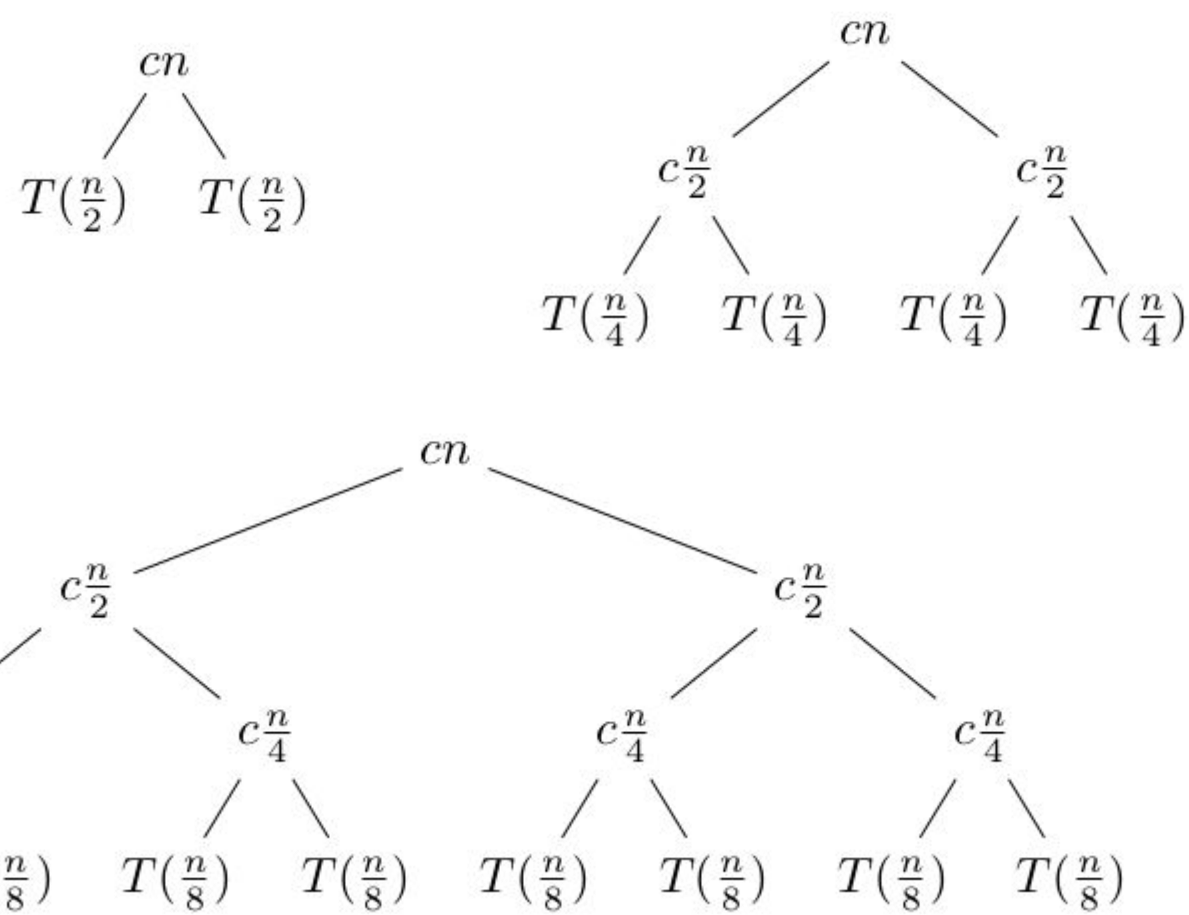
$$= \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(\frac{n}{2}) + \Theta(n) & \text{if } n > 1. \end{cases}$$

How to solve this recurrence equation?

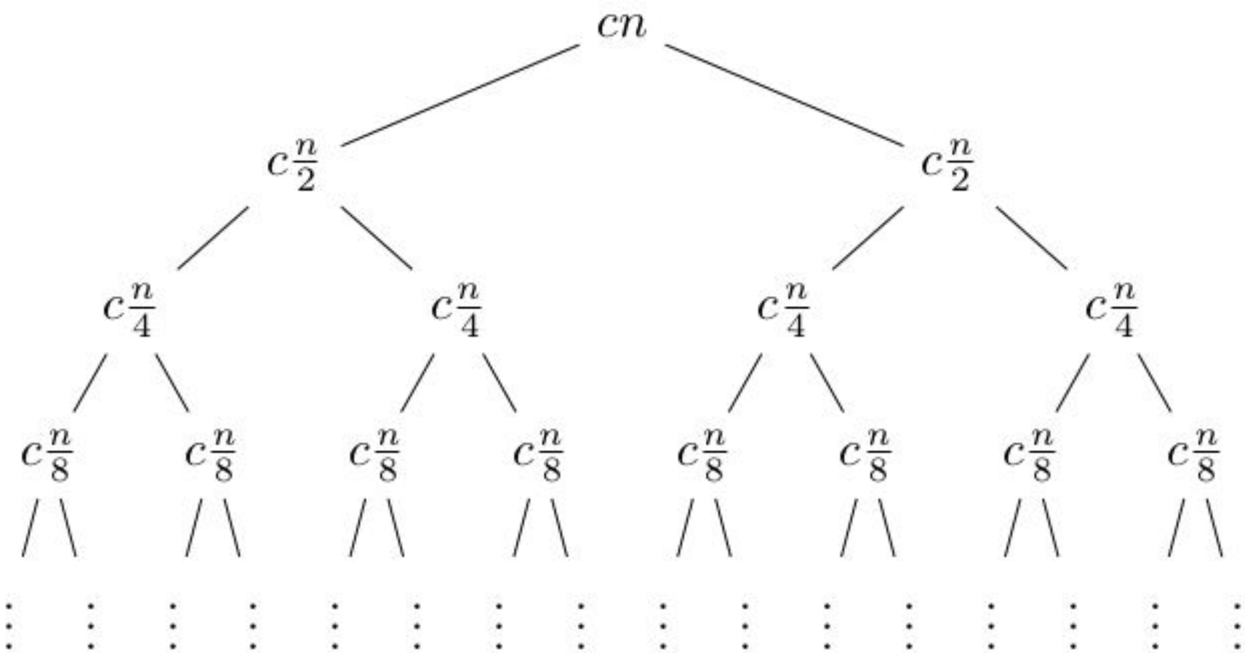
Recurrence tree

Repeatedly expand the recurrence, using $c\,n$ for $\Theta\,(n)$:

$T(n)$



Recurrence tree (cont'd)



2^d nodes at depth d, each of cost $c\frac{n}{2^d}$

Total cost at depth $d = c\ n$

Base case reached when $\frac{n}{2^d} = 1$ i.e. when $d = \log n$

Therefore total cost = $T(n) = c\ n\ d_{max} = \text{cn} \log n$ = $\Theta(\text{n} \log \text{n})$

Example 2: Towers of Hanoi

```
procedure Move-Tower(height, source, destination, auxiliary)
  if height ≥ 1
    Move-Tower(height - 1, source, auxiliary, destination)
    Move-Disk(source, destination)
    Move-Tower(height - 1, auxiliary, destination, source)
```

Recurrence equation (using *n* for *height*) is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n-1) + \Theta(1) & \text{if } n > 1. \end{cases}$$

Exercise: draw the recurrence tree.

Analysis

2^d nodes at depth d, each of cost c

Total cost at depth $d = c\ 2^d$

Base case reached when $d = n$ - 1

Therefore total cost = $$T(n) = \sum_{d=0}^{n-1} c2^d = c(2^n - 1) = \Theta(2^n)$$

Exercise: derive an exact expression for the number of moves required to solve a height *n* Towers of Hanoi problem.

Example 3: binary search

procedure Binary-Search(A, x, left, right)

  if left > right

    return NULL

  mid ← $\left\lfloor \frac{left + right}{2} \right\rfloor$

  if x < A[mid]

    return Binary-Search(A, x, left, mid - 1)

  else if x = A[mid]

    return mid

  else

    return Binary-Search(A, x, mid + 1, right)

Exercise: give a tight bound on the worst-case complexity using a recurrence tree

Answer

Recurrence equation is: $$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T(\frac{n}{2}) + \Theta(1) & \text{if } n > 1. \end{cases}$$

Recurrence tree is … ?

Nodes at depth d: 1

Cost of each node at depth d is c

Base case is reached when $\frac{n}{2^d} = 1$ i.e. when $d = \log n$

Total cost is $\mathrm{T}(n) = c \log n = \Theta(\log n)$

Example 4: Fast exponentiation

Algorithm to compute a^n where a and n are integers and n is large and an exact result is required (an arbitrarily-long integer):

procedure Power(a, n)
  if $n = 0$
    return 1
  else
    $p \leftarrow$ Power(a, $\lfloor n/2 \rfloor$)
    if n is even
      return $p \times p$
    else
      return $a \times p \times p$

Question: what is its time complexity?

Example 5: factorial

1. Give a D&C factorial algorithm.
2. Is this better or worse than an iterative approach? Why?
3. Is there a difference between worst-case and best-case complexity?
4. Give the recurrence equation for the time complexity.
5. Use a recurrence tree to find a tight bound on the time complexity.
6. Is the time complexity better than the iterative version?

Example 6: Fibonacci numbers

Fibonacci numbers: $F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n > 1$.

1. Use induction to prove that

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n .$$

2. Use this to design a D&C algorithm to compute F_n in time $O(\log n)$

Some notes on recursion

- Python recursion limit is by default 1000.

  To increase:

  ```
  import sys
  sys.setrecursionlimit(30000)
  ```

  But can still get hardware stack overflow ("segfault"), so might also need

  ```
  import resource
  resource.setrlimit(resource.RLIMIT_STACK, (0x10000000, resource.RLIM_INFINITY))
  ```

  (soft, hard) limits in bytes. See docs.

  But (rule of thumb): avoid *setrlimit*. Find a different algorithm (iterative?). Code is too OS dependent and likely flaky.

- Iteration is usually much faster than recursion, if practicable.
- Many subtle effects can dramatically influence behaviour
  - Hardware cache, virtual memory, chosen data types, memory allocator, ...

Example: consider ...

```
def recursive_count1(items, target):
    if len(items) == 0:
        return 0
    elif items[0] == target:
        return 1 + recursive_count1(items[1:], target)
    else:
        return recursive_count1(items[1:], target)

def recursive_count2(items, target, start=0):
    if start >= len(items):
        return 0
    elif items[start] == target:
        return 1 + recursive_count2(items, target, start + 1)
    else:
        return recursive_count2(items, target, start + 1)

def iterative_count(items, target):
    count = 0
    for item in items:
        if item == target:
            count += 1
    return count
```

What are their orders of complexity?

What runtimes would you predict with 1,000, 10,000, 100,000,000 and 1,000,0000 items?

Analysis

- Iterative version is clearly (?) Θ(n)
- For both (?) recursive versions, recurrence is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T(n-1) + \Theta(1) & \text{if } n > 1. \end{cases}$$

  - Recurrence tree has n levels, 1 node at each level of cost c
  - Hence also O(n)
- Or is it?

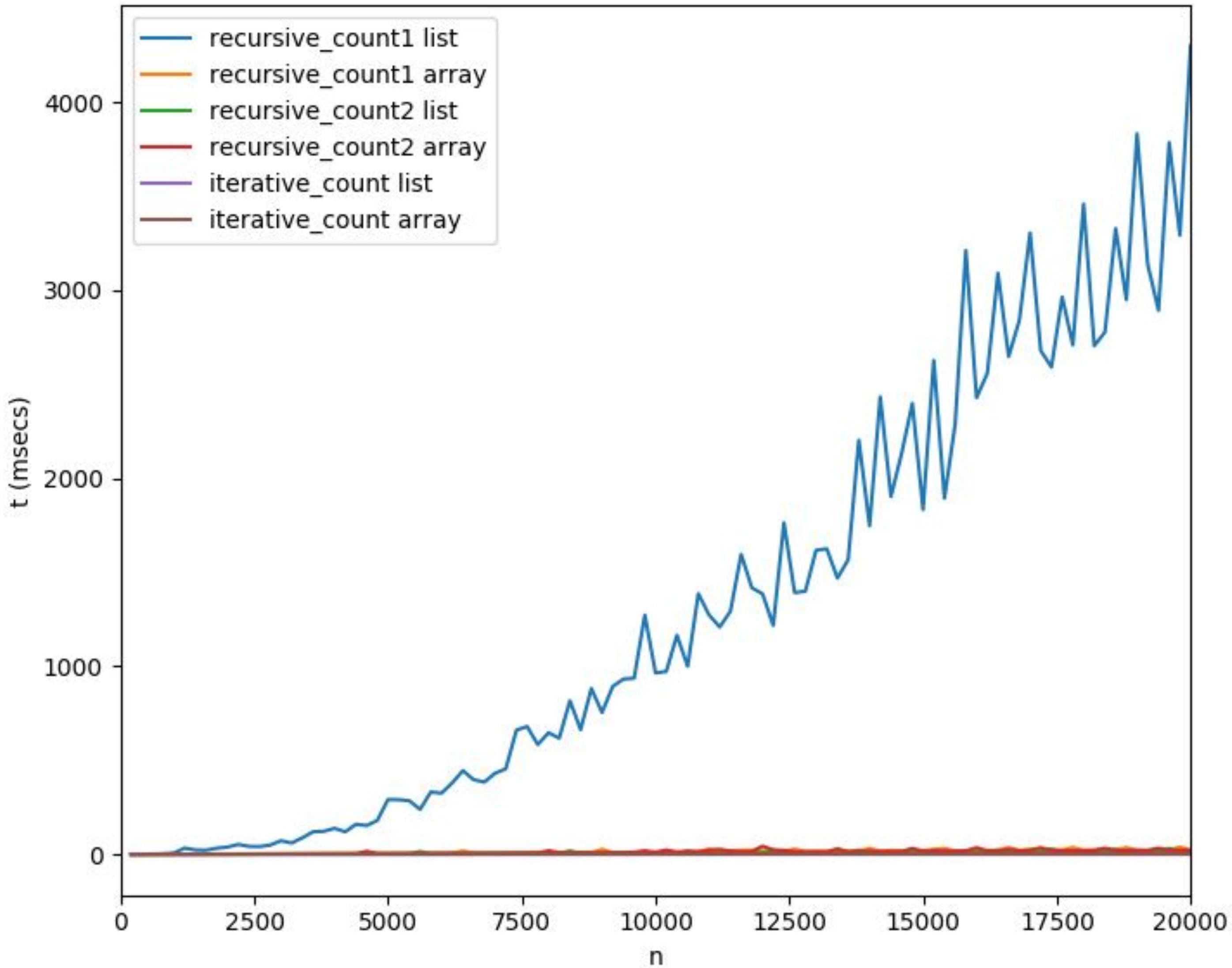
Let’s measure it!

Note use of numpy array as an alternative to a list. Don’t worry about that for now.

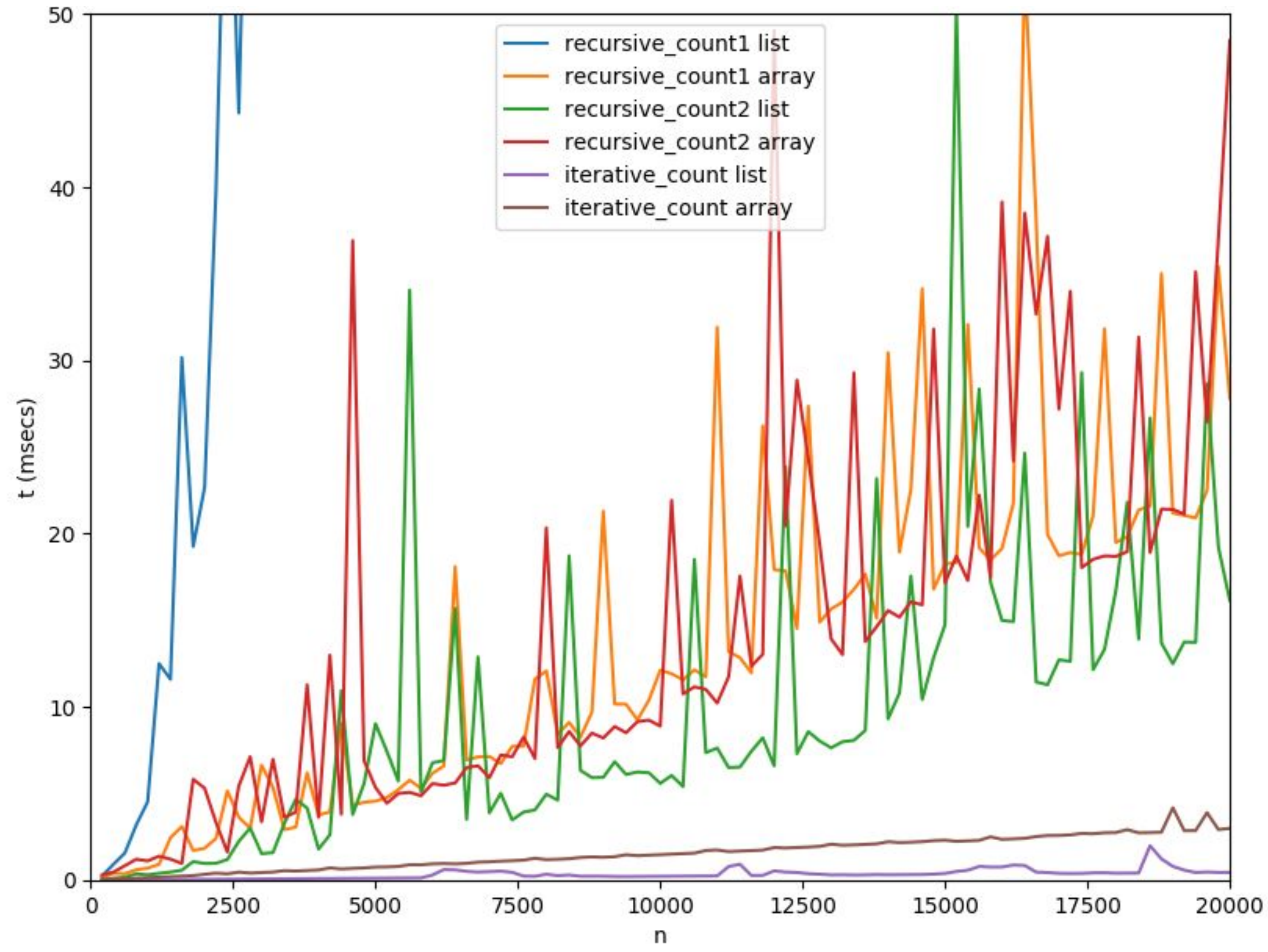
```
import sys
import numpy as np
import matplotlib.pyplot as plt
from time import perf_counter
import resource
sys.setrecursionlimit(30000)
resource.setrlimit(resource.RLIMIT_STACK,
        (0x10000000, resource.RLIM_INFINITY))

NMIN = 200
NMAX = 2000 # Vary to suit
DN = 200

# Different count functions go here

axes = plt.axes()
funcs = [recursive_count1, recursive_count2,
         iterative_count]
```

```
for fun in funcs:
    for seq_type in [list, np.array]:
        ns = []
        ts = []
        for n in range(NMIN, NMAX + 1, DN):
            ns.append(n)
            items = seq_type(range(n))
            t0 = perf_counter()  # Time in secs
            cnt = fun(items, n // 2)
            ts.append(perf_counter() - t0) * 1000  # msecs
        axes.plot(ns, ts,
         label=f"{fun.__name__} {seq_type.__name__}")

axes.set_xlim(0, NMAX)
axes.set_xlabel('n')
axes.set_ylabel('t (msecs)')
axes.legend()
plt.show()
```

Results


recursive_count1 list
recursive_count1 array
recursive_count2 list
recursive_count2 array
iterative_count list
iterative_count array
t (msecs)
4000
3000
2000
1000
0
0
2500
5000
7500
10000
12500
15000
17500
20000
n

Results with different vertical scale



Observations

- Noisy/erratic measurements with recursive algorithms
  - Due to garbage collection/memory allocation (see next 2 slides)
  - Every function call has to dynamically allocate a new stack frame.
  - See ENCE260.
- recursive_count1 is quadratic on lists
  - Because slicing (items[1:n]) is linear in *n*
  - But not with numpy arrays, where a slice is a different "view" into an array
    - Just alters the indexing
    - No copying involved - O(1)
- Iteration is 30 - 100 times faster than even the O(*n*) recursions

Memory management

- Computer memory is a huge linear array of numbered "slots"
    - Each slot holds 8-bits (1 byte)
    - The slot number is called its 'address'
- In COSC121 we talked about "the object store"
- This is implemented by layers of memory management.
- A statement like data = [100, -301, 11] might involve:
    - Allocate a dictionary entry in __locals__ for 'data'
        - Resize the allocated space for __locals__ if necessary
    - Allocate space for a list object, and set __locals__['data'] to its address
    - Allocate space for three new ints (100, -301, 11) and insert their addresses into the list object

Memory management (cont'd)

- And when adding items to the list object it might be necessary to allocate a new large block of memory and copy across all the values from the old.
- In addition it might be necessary to 'garbage collect' any disused memory blocks.
- These operations can sometimes be expensive and dramatically affect performance.
- Do ENCE260 if you want to understand this better.

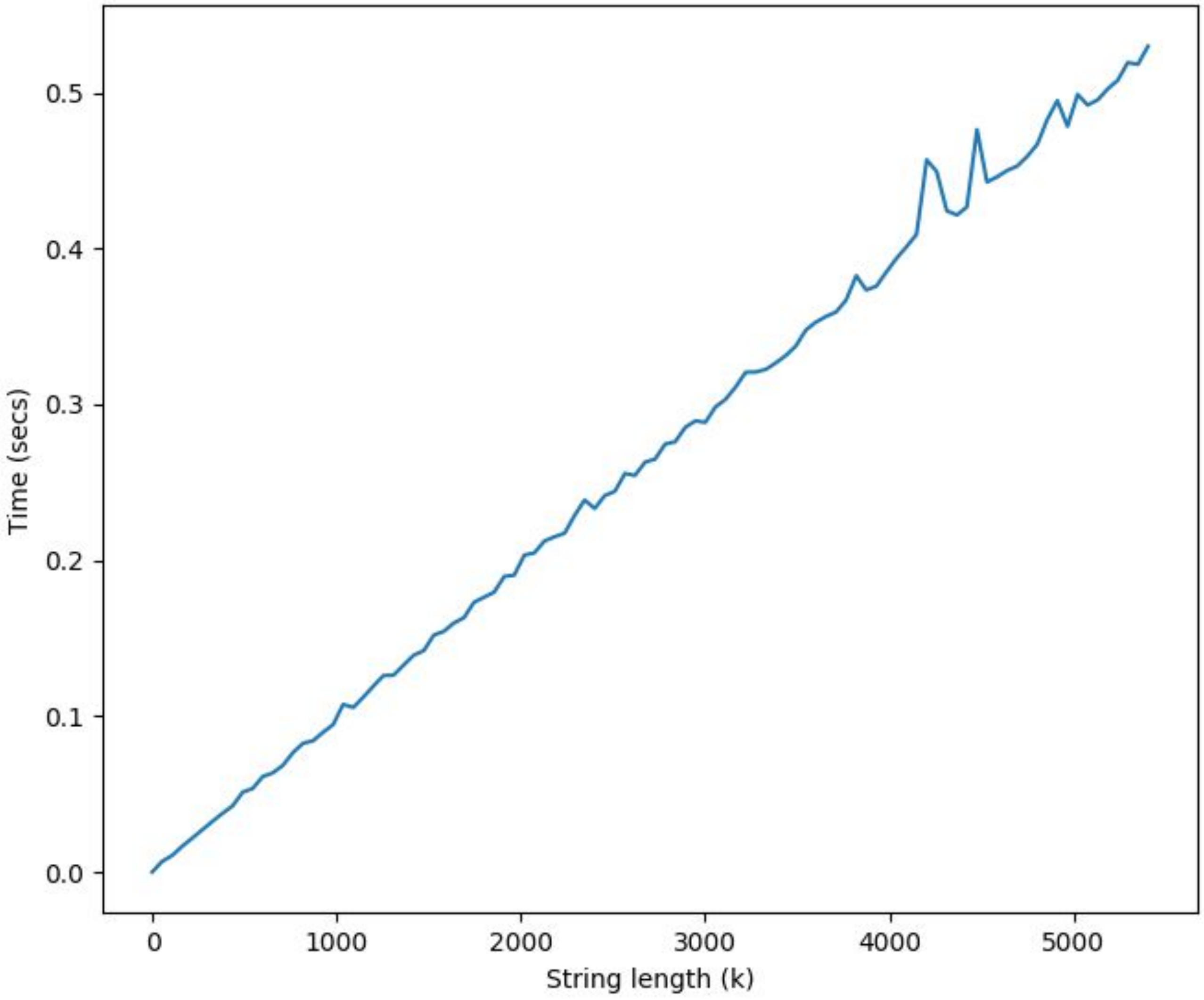
Another Python example

Q: What’s the O complexity of the following code?

```
def remove_white_space(filename):
    data = open(filename).read()
    result = ''
    for char in data:
        if char not in [' \t\n']:
            result += char  # Or result = result + char
    return result
```

A: If you answered $O(n^2)$, congratulations. That’s how your lecturer answered, too. But it’s wrong. If you answered $O(n)$, congratulations, too. You’re right. Did you know more than your lecturer, or less?

Results



Takes ~450 msecs for a 5 million character string.

Huh? How come?

The Python engine recognises both

```
s = s + delta_s
```

and

```
s += delta_s
```

as growing a string. It (mostly) avoids copying the string by growing it in place.

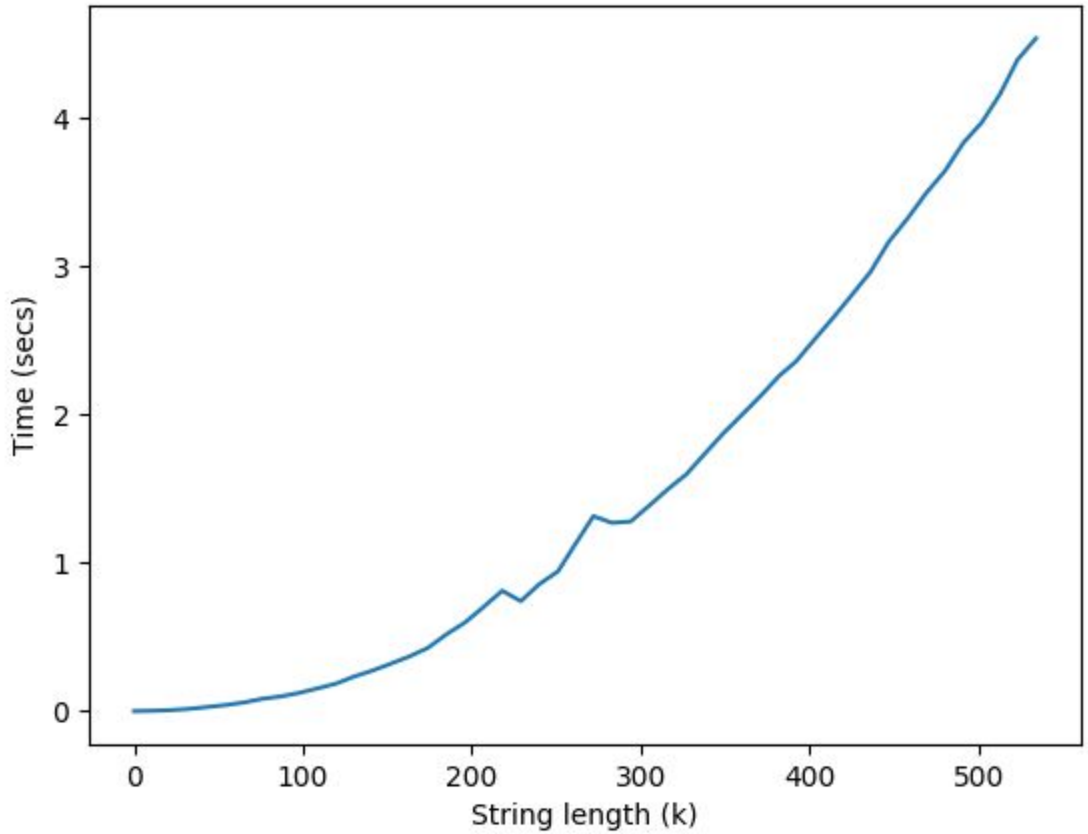
Yet another Python example

Q: What about this version, then?

```
def remove_white_space_reversed(filename):
    data = open(filename).read()
    result = ''
    for char in data:
        if char not in [' \t\n']:
            result = char + result
    return result
```

A: If you answered O(n^2), congratulations. This time, you're right. The entire string gets copied each time through the loop.

Results



Quadratic growth. It took ~22 minutes for a 5 million character string.

Q: how could you make it linear?

Tail recursion

- “Functional languages” (e.g. Haskell, Scheme) don’t support iteration but rely on recursion.
- They and a very few other non-functional languages (e.g. Scala, Lua) perform “Tail Call Optimisation” (TCO) to convert tail recursion into iteration.
  - With such languages tail recursive functions are just as fast as iterative ones (because they *are* iterative!)
- Tail recursion is when the return value of a function is just a recursive call to the function
  - Recursion can be converted to a jump back to the start (after tweaking parameters)
  - No extra computation is allowed in the return value
  - Although clever TCO can do transformations as in the following slide
- But Python, Java, C++ etc do *not* support TCO.

Tail recursion example

Our recursive count function can be made tail recursive (it wasn't before):

```
def tail_recursive_count(items, target, start=0, count_so_far=0):
    if start >= len(items):
        return count_so_far
    elif items[start] == target:
        return tail_recursive_count(items, target, start + 1, count_so_far + 1)
    else:
        return tail_recursive_count(items, target, start + 1, count_so_far)
```

But because Python doesn't support TCO, this doesn't gain you anything.

Moral: if you really want to recurse efficiently (and never iterate again), use a functional language (or maybe Scala).

Big-O isn’t everything

If algorithm A is $O(n^2)$ and algorithm B is $O(n)$ you should always use algorithm B. True or false?

False. Need to consider:

- The proportionality constants in both.
- The size (n) of the problem(s) to be solved.
- The complicatedness of the two algorithms.
  - Are you going to have to time to code B?
  - Will you get it right?
- How often the code needs to be run.
- A myriad of underlying complexities (memory management, caching, etc).

Measure the performance of your code.
Don’t assume you know how it will perform.

Example: Python sorted containers

See http://www.grantjenks.com/docs/sortedcontainers

- Python does not have a standard sorted list class
  - i.e. a list class that remains sorted when you insert items
  - (Re)sorting on every insertion is O(n log n).
  - Or (better) binary searching for insertion point then making space by shifting is O(n).
- We want, at worst, O(log n) insertion, deletion, indexing and searching.
- Q: what COSC122 data structure(s) would provide such a capability?
  - And for each, what are the complexities of insert, delete, index, search?

Conventional wisdom

and its limitations

- Some sort of tree structure is called for (e.g. AVL, Red-Black)
  - O(log n) insertion, deletion, indexing
- But a (binary) tree node requires at least 3 pointers:
  - Node value, left child, right child.
  - 24 bytes (on 64-byte machine).
  - These overheads limit the maximum size (n_max) compared to a simple list with a single pointer, 8 bytes, per item.
  - And memory sizes limit data sizes in any case
- Per-node memory allocation can be relatively expensive
- Many O(n) operations on built-in Python lists are very fast
  - Highly optimised machine code

SortedContainers module

- **Written in Python.**
  - Provides *SortedList*, *SortedDict* and *SortedSet* classes
- Partitions the sorted list into sublists, each up to 1,000 elements.
- An additional sorted list of *max_element_in_list* can be binary searched to find what list a particular element is in.
  - Then that list is binary searched.
- A binary tree of cumulative indices is used to index into the list of lists.
- All very ad hoc and “unprofessional” (?)
- O(n) insertion and deletion complexity.
- But generally outperforms even C/C++ tree and skip-list versions with O(log n) insertion and deletion complexity.
  - See http://www.grantjenks.com/docs/sortedcontainers/performance.html
- But probably has some poor worst-case scenarios

My point being ... ?

- Big-O analysis is extremely important but:
  - The constants of proportionality matter too.
  - The RAM model on which it is based does not account for many performance complexities, e.g. hardware caches.
  - In dynamic languages like Python, performance is further complicated by memory-management and other implementation details.
  - For small problems, simplicity and maintainability matter more.
- Guidelines:
  - Use tested reliable off-the-shelf algorithms and data structures where possible.
  - Big-O analysis is the best starting point but if performance becomes an issue, instrument and measure your code. Don't guess.