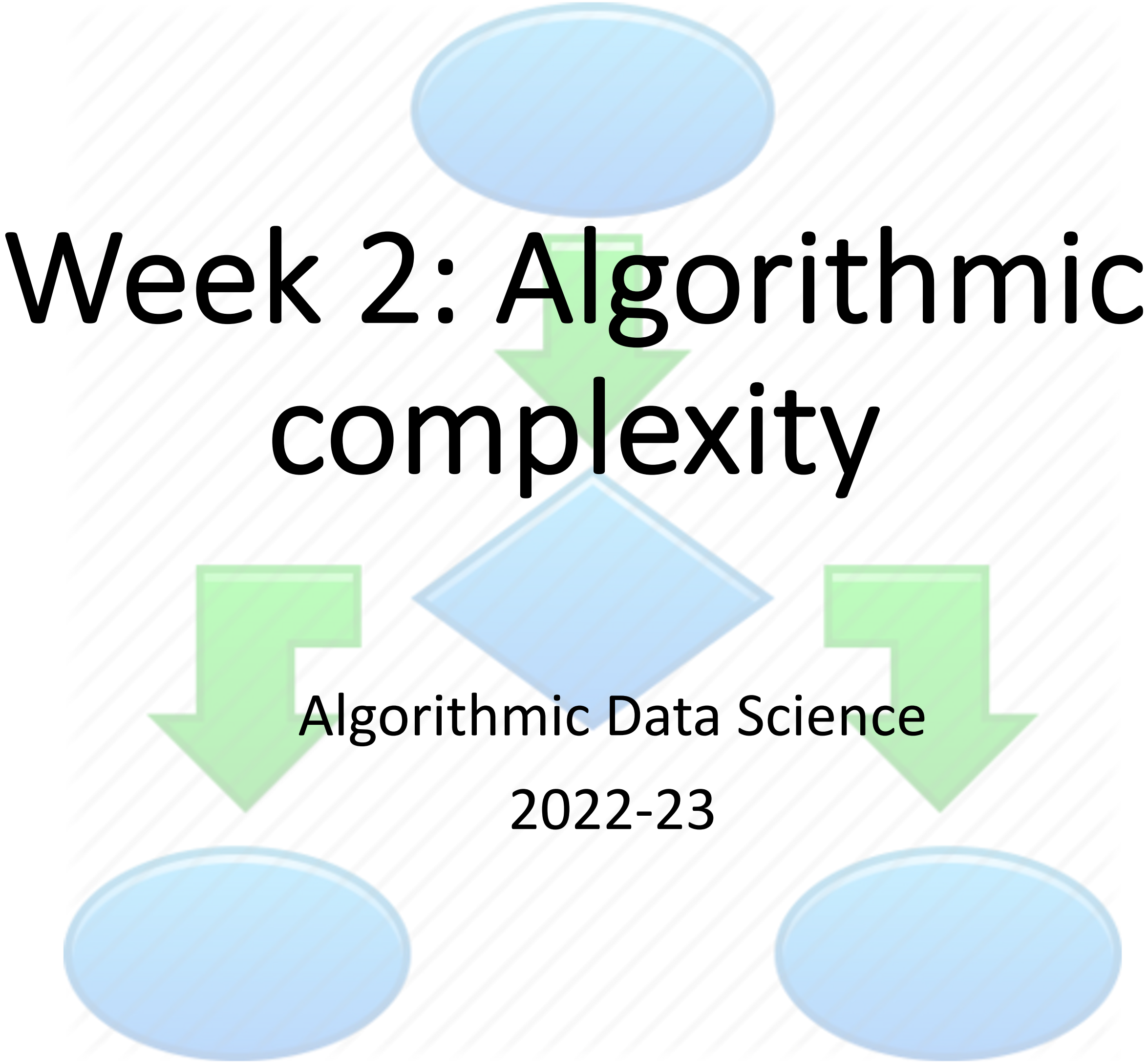


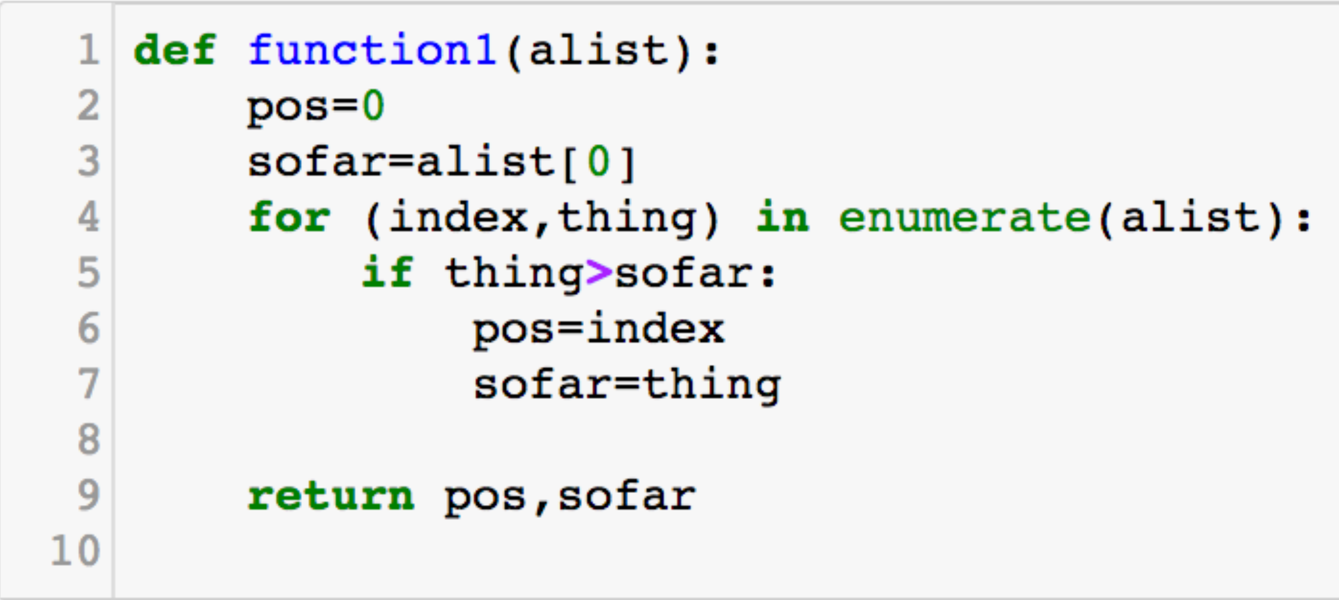
Week 2: Algorithmic complexity

Algorithmic Data Science

2022-23

UNIVERSITY OF SUSSEX

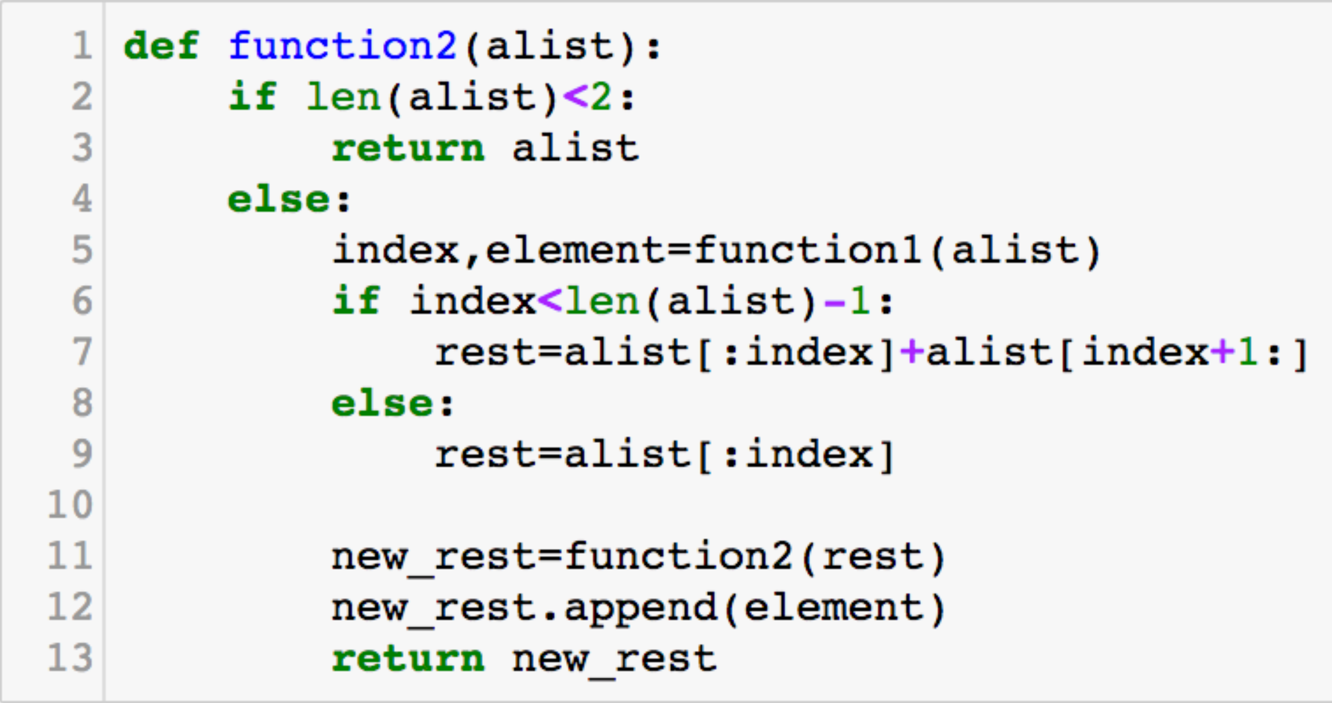
Dr Adam Barrett



```python
def function1(alist):
    pos=0
    sofar=alist[0]
    for (index,thing) in enumerate(alist):
        if thing>sofar:
            pos=index
            sofar=thing

    return pos,sofar

```



```python
def function2(alist):
    if len(alist)<2:
        return alist
    else:
        index,element=function1(alist)
        if index<len(alist)-1:
            rest=alist[:index]+alist[index+1:]
        else:
            rest=alist[:index]

        new_rest=function2(rest)
        new_rest.append(element)
        return new_rest
```

Warm-up:

Look at the code for function1() and function2()

1. What would be the result of running each of them on the list [5,3,7]?
2. What problem is each of them trying to solve?
3. Do you think each of them uses the best algorithm to solve the problem they are trying to solve?

Week	Who	Topic
1	Barrett	Data structures and data formats
2	**Barrett**	**Algorithmic complexity. Sorting.**
3	Barrett	Matrices: Manipulation and computation
4	Barrett	Similarity analysis
5	Rosas	Processes and concurrency
6	Rosas	Distributed computation
7	Barrett	Map/reduce
8	Barrett	Clustering, graphs/networks
9	Barrett	Graphs/networks, PageRank algorithm
10	Barrett	Databases
11		*independent study*

Recap: algorithms vs problems

- An **algorithm** is a well-defined *procedure* that takes a value or set of values as *input* and produces some value or set of values as *output*.

- We distinguish **algorithms** from **problems** (or functions), because intuitively we know that sometimes very different algorithms do the same thing i.e., solve the same problem

Aside: algorithms vs programs

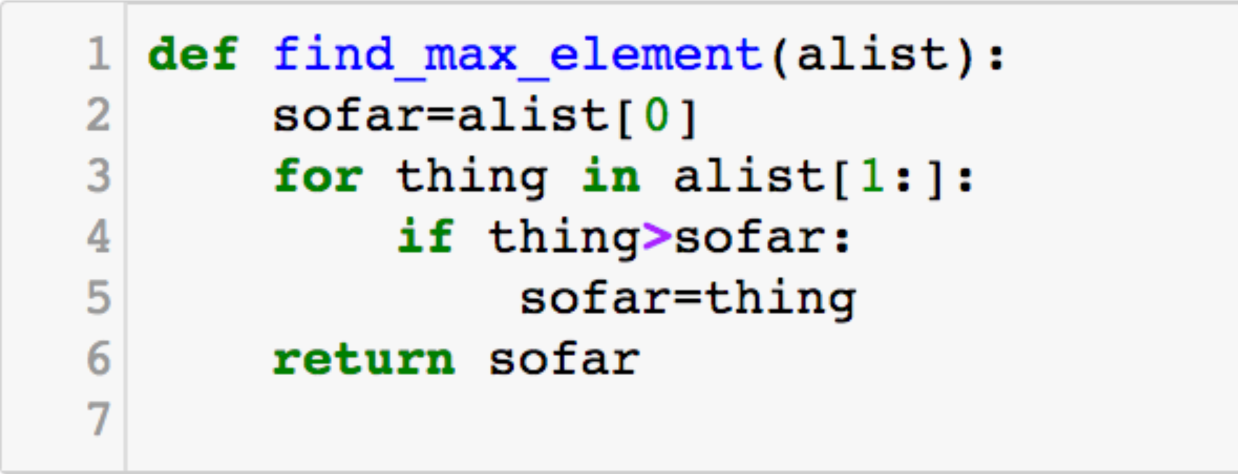
- We use the term algorithm interchangabley with program (e.g., in Python). The emphasis is different though:

An algorithm

```
Finding biggest element in a list:

Set biggest_sofar to be first element of list.
Then, check each element in list in turn
If It is bigger than biggest_sofar,
update biggest_sofar
```

A program



```python
def find_max_element(alist):
    sofar=alist[0]
    for thing in alist[1:]:
        if thing>sofar:
            sofar=thing
    return sofar

```

Algorithms focus on **ideas**, not syntax

Programs should be **syntactically correct**

Problems

- More formally, a problem is a mapping or a relation from inputs to outputs.
- Many problems are also functions. For functions, the mapping is many-to-one or one-to-one. In other words there is only a single correct output for a given input.

$$f: X \rightarrow Y$$

We say that an algorithm A implements a function problem f, if $\forall x \in X, \forall y \in Y$:

$$f(x) = y \quad \text{iff. } A(\text{x}) \text{ terminates and yields } y$$

Rate my algorithm ….

Its not enough to write down an algorithm and say ‘Behold. Look at this marvellous algorithm I have written.’

We need to convince ourselves that:

- The algorithm **terminates (for every possible input)**
- The algorithm is **correct (for every possible input)**
- The algorithm is **better than other possible algorithms.**

Comparing algorithms

What does it mean for algorithm A to be better than algorithm B?

TIME COMPLEXITY

How long will my program take to run?
Is algorithm A faster than algorithm B?

SPACE COMPLEXITY

How much memory will my program need?
Does algorithm A use less memory than algorithm B?

COMMUNICATION COMPLEXITY

How many packets of data will my program send over the network?
Does algorithm A send less packets over the network than algorithm B?

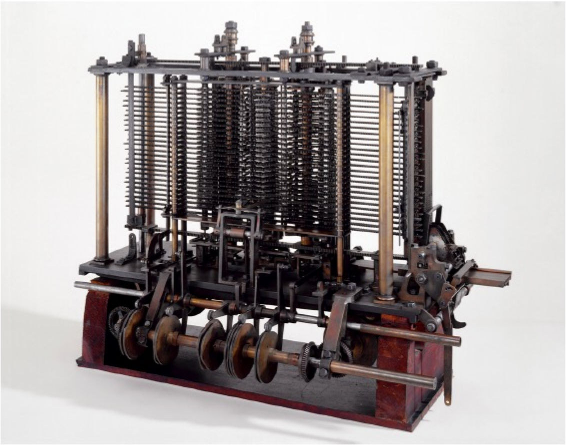
What does A is faster than B mean?

- A runs faster than B on **some** inputs
- A runs faster than B on **all** inputs
- A runs faster than B on the **best case**
- A runs faster than B on the **worst case**
- A runs faster than B on **average**
- A runs faster than B on **average relative to some probability distribution over the inputs**
- A runs faster than B on **typical** inputs

History of algorithmic analysis

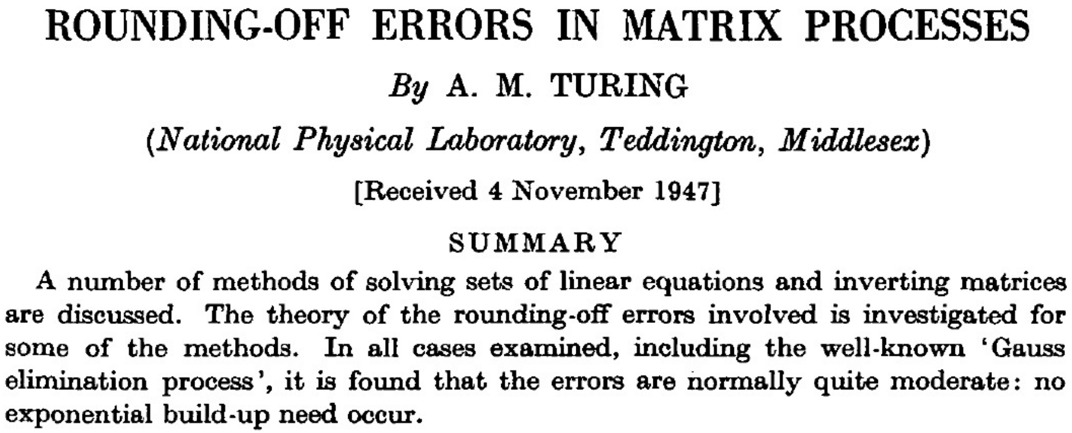
As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise – By what course of calculation can these results be arrived at by the machine in the shortest time? **Charles Babbage 1864.**

The Analytic Engine:



History of algorithmic analysis

It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then give them various weights. **A. Turing, 1947.**



ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING

(*National Physical Laboratory, Teddington, Middlesex*)

[Received 4 November 1947]

SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known 'Gauss elimination process', it is found that the errors are normally quite moderate: no exponential build-up need occur.

History of algorithmic analysis

In the 1960s Donald Knuth and others began building a scientific foundation for the analysis of algorithms. The goal:

Predict performance of algorithms in a *lightweight* way and enable their comparison

The tools for understanding the performance of algorithms are drawn from 19th century mathematics:

- Asymptotic analysis
- Recurrence relations
- Generating functions

Time Complexity

- How does the run-time of an algorithm depend on the size of the problem instance / input?

Size of the problem instance : *n*

We notice that data objects $x \in X$ come with a natural notion of size, which we refer generally to as *n*:

- If x is a number, size(x) = x
- If x is a string, size(x) = number of characters in x
- If x is a list, size(x) = number of elements in the list
- If x is a tree, size(x) = number of nodes in the tree
- If x is a tree, size(x) = height of tree
- For any data structure x, size(x) is the number of bits it takes to represent it in memory

Time Bounds

We can define a time bound for algorithm A: $\mathrm{X} \rightarrow Y$ as a function $time$

that for all $x \in X$:

the run-time of A(x) $\leq time(size(x))$

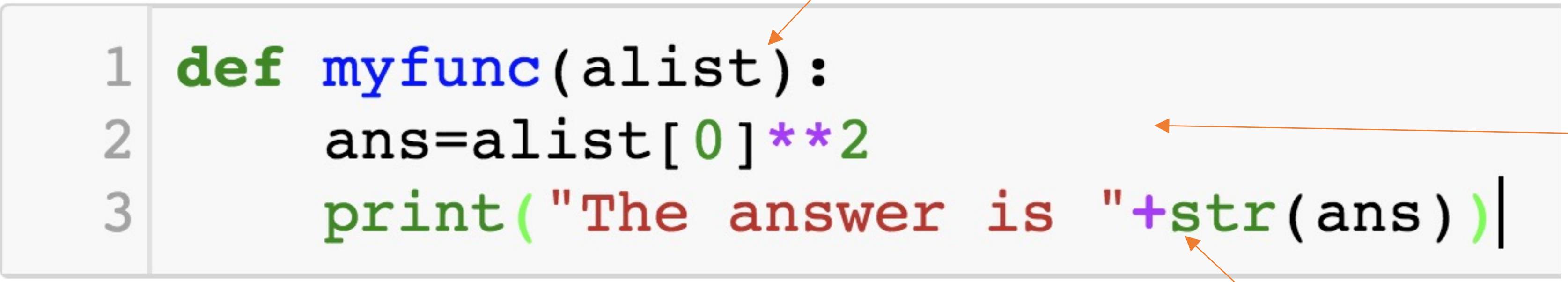
- This is an upper bound on run-time
- It is highly sensitive to the choice of size()

Constant-time Operations

- Clearly, taking more information into account yields more accurate predictions. But, it also makes the analysis much more complicated.
- An important insight is that addition, subtraction, if-then-else, …, are (to a good approximation) **constant-time operations.**
- That means there are constants c_{add}, $c_{subtract}$, $c_{if\text{-}then\text{-}else}$, … such that
  - Executing an addition always takes c_{add} seconds
  - Executing a subtraction always takes $c_{subtract}$ seconds
  - Executing a conditional always takes $c_{if\text{-}then\text{-}else}$ seconds
- These constants rely heavily on the chosen programming language, CPU, compiler, clockspeed etc

Constant-time

Problem size n = len(alist)



```
def myfunc(alist):
    ans=alist[0]**2
    print("The answer is "+str(ans))
```

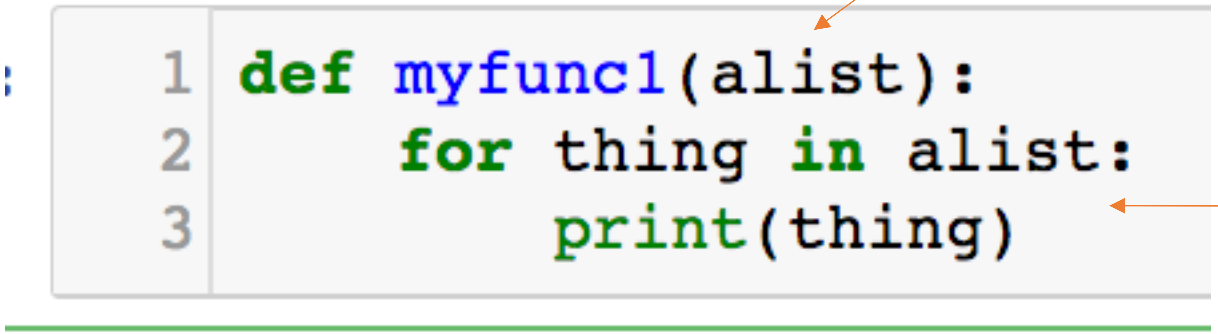
3 operations which are all constant-time: access to list component, exponentiation and assignment

3 constant-time operations: conversion to string, string concatenation, printing

The length of time to complete does NOT depend on the size of the input. This is a constant-time function. The run-time is of the order of 1: **O(1)**

Iterations: Linear-time

Problem size *n* = len(alist)



```python
def myfunc1(alist):
    for thing in alist:
        print(thing)
```

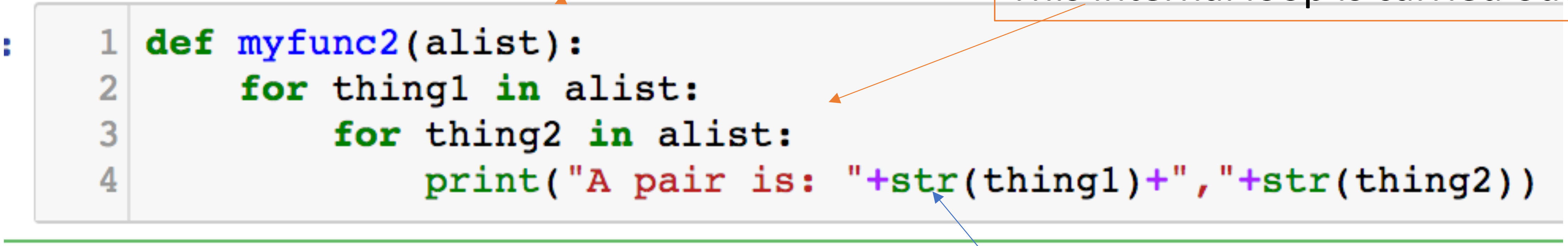
A constant time operation (print) is carried out *n* times

The length of time to complete depends linearly on the size of the input. If the length of the list is doubled, the run-time will be doubled. The run time is some function of n i.e., it is of the order of n. We call this **O(n)**

Nested iterations: quadratic time

Problem size *n* = len(alist)

This internal loop is carried out *n* times



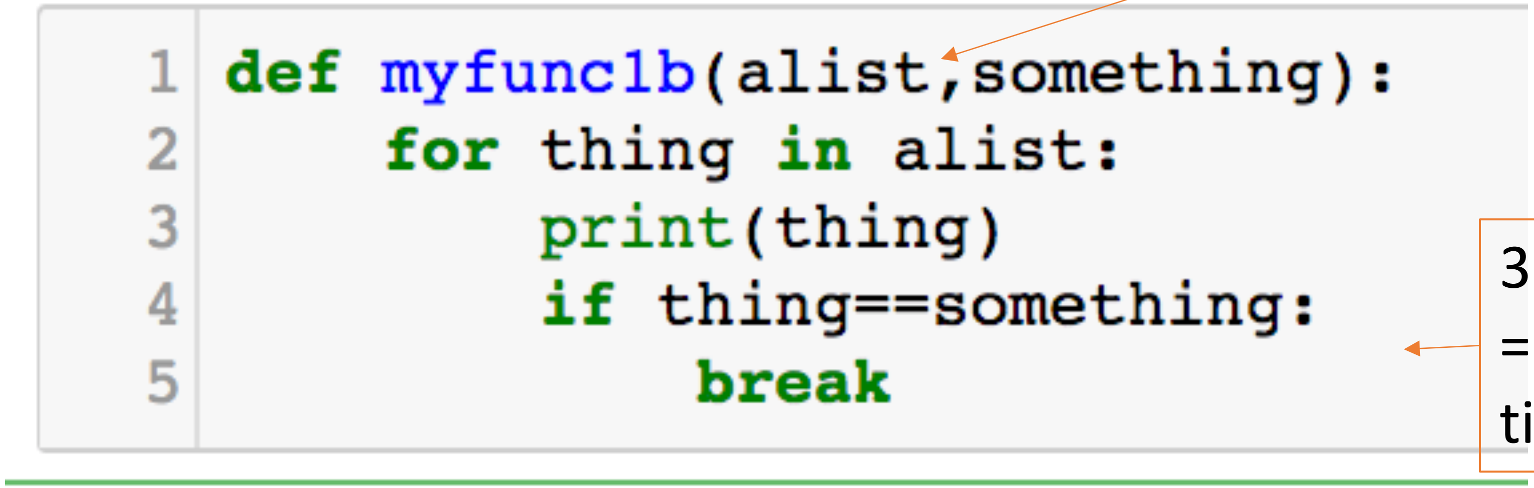
```
def myfunc2(alist):
    for thing1 in alist:
        for thing2 in alist:
            print("A pair is: "+str(thing1)+","+str(thing2))
```

This instruction is carried out **n * n** times.

The run-time is quadratic in the size of the input. If the length of the list doubles, the run-time will quadruple. We call this **O(n^2)**

Broken Iterations: Linear-time

Problem size *n* = len(alist)



```python
def myfunc1b(alist,something):
    for thing in alist:
        print(thing)
        if thing==something:
            break
```

3 constant time operations (print, == and if) is carried out up to *n* times

The UPPER BOUND on the length of time to complete depends linearly on the size of the input. If the length of the list is doubled, the run-time will be potentially doubled. The upper-bound on run time is some linear function of n i.e., it is of the order of n. This is **O(n)**

Asymptotic order of growth

Let's assume we have two algorithms A_1 and A_2 for a given problem, and their run-time is given by f_1 for A_1 and f_2 for A_2, where:

- $f_1(n) = 0.00001 \times 2^n$
- $f_2(n) = 289346724587138654380 \times n^2$

Which algorithm would you choose and why?

Asymptotic order of growth

Let's assume we have two algorithms A_1 and A_2 for a given problem, and their run-time is given by f_1 for A_1 and f_2 for A_2, where:

- $f_1(n) = 0.00001 \times 2^n$
- $f_2(n) = 2893467245871386543 80 \times n^2$

For small data, A_1 is faster, but eventually, for big enough data A_2 is faster. We are interested in the behaviour of algorithms as the input data grows large.

Asymptotic order of growth

Assume an algorithm has run-time

- $f(n) = 0.2456 \times 2^n + 2345n^7 + 8934n^5 + 163\sin(90435 + n)$

What is a good approximation to f for big data?

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

Why? Because 2^n dominates everything else in the definition of *f* for large enough n. The function *f* asymptotically behaves like a simpler function *g*. Therefore we use *g* instead of *f* to think about f for large inputs.

Do constants matter?

- Assume 1,000,000 operations per second

	n	nlogn	n^2	n^3	1.5^n	2^n	n!
n=10	<1s	<1s	<1s	<1s	<1s	<1s	4s
n=30	<1s	<1s	<1s	<1s	<1s	18m	10^{25}y
n=50	<1s	<1	<1s	<1s	11m	36y	long
$n=10^2$	<1s	<1s	<1s	1s	12892y	long	long
$n=10^3$	<1s	<1s	1s	18m	long	long	long
$n=10^4$	<1s	<1s	2m	12d	long	long	long
$n=10^5$	<1s	2s	3h	32y	long	long	long
$n=10^6$	1s	20s	12d	31710y	long	long	long

s=seconds
m=minutes
h=hours
d=days
y=years
long> 10^{25} years

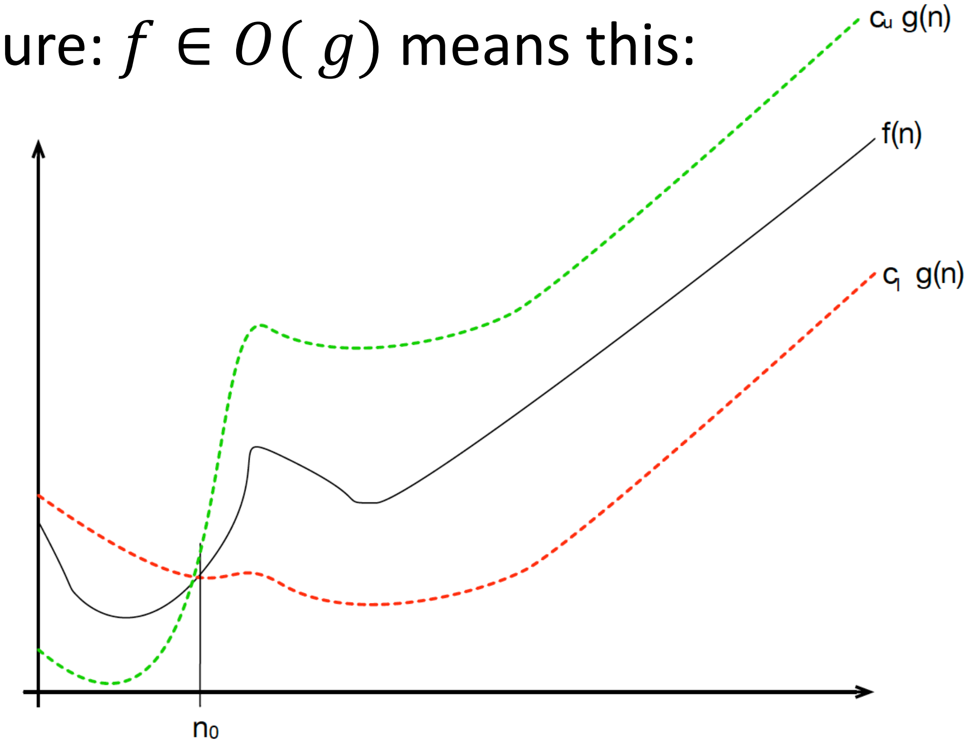
- For large enough *n*, the specific constants often fade into insignificance in comparison to the number of operations.
- In practice, you can find the constant by running algorithm on a subset of data, and then extrapolate to estimate the time needed for the big data task you want to do.

O Notation

O notation asymptotically bounds a function from above and below.

We write $f(n) \in O(g(n))$ if there are positive constants n_0, c_1 and c_2 such that for $\mathrm{n} > n_0$,
$c_1(g(n)) \leq f(n) \leq c_2(g(n))$

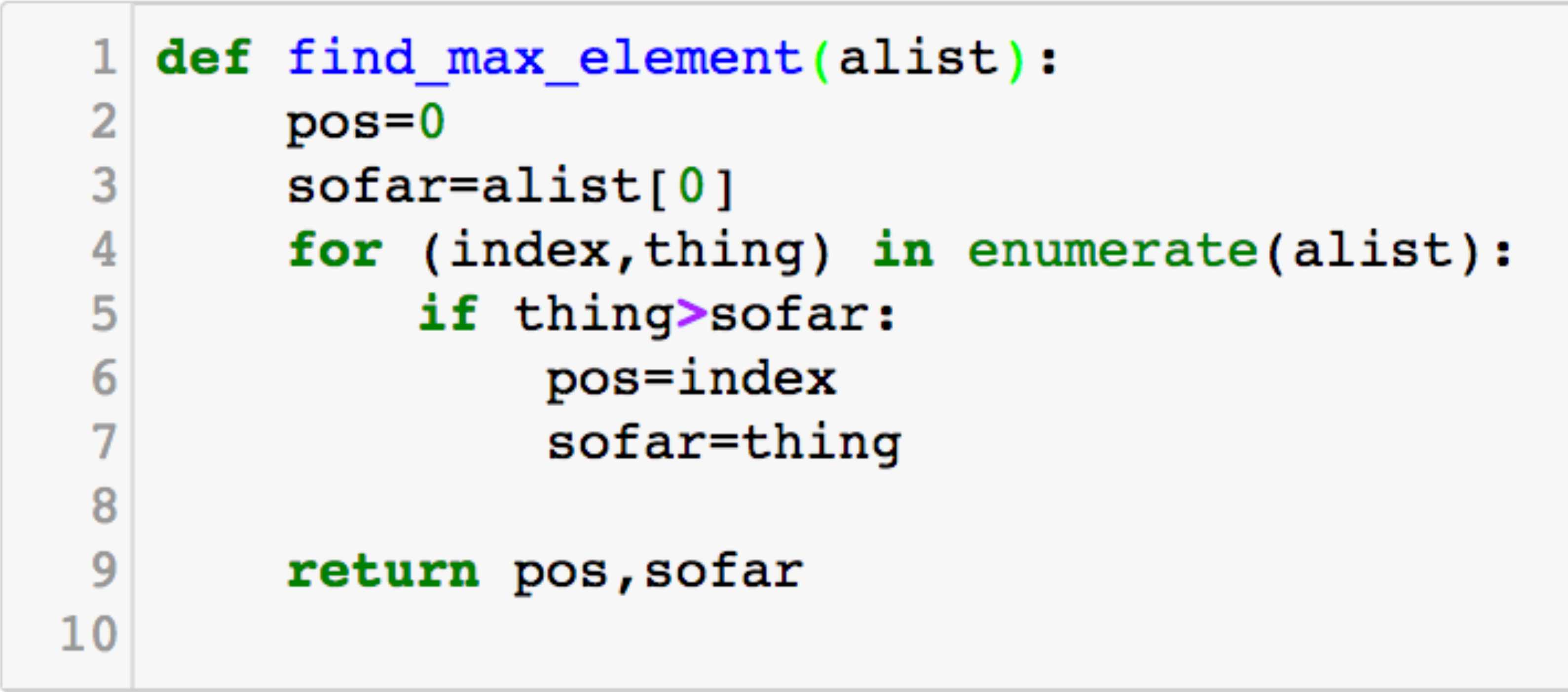
In a picture: $f \in O(g)$ means this:



For almost all *n, f*(*n*) is equal to g(n), to within a constant factor. We say that *g* is an asymptotically tight bound for *f*.

NB: In some textbooks, Θ is used instead of O. And then O means something slightly different- see 2 slides down.

What is the time complexity of find_max_element()?



```python
def find_max_element(alist):
    pos=0
    sofar=alist[0]
    for (index,thing) in enumerate(alist):
        if thing>sofar:
            pos=index
            sofar=thing

    return pos,sofar
```

- (a) O(1)
- [✓ (b) O(n)]
- (c) O(n^2)
- (d) O(2^n)

O Notation Examples: True or False?

• $17x^6 + 15x^4 \in O(x^6)$	TRUE
• $782x^2 + x^3 \in O(x^2)$	FALSE
• $128x^2 + 564x - 76 \in O(x^3)$	FALSE
• $2^n \in O(2^{n+1})$	TRUE
• $2^{n+1} \in O(2^n)$	TRUE
• $2^{2n} \in O(2^n)$	FALSE

NB: In some textbooks, Θ is used instead of O. And then O means something slightly different

A sometimes encountered alternative definition of O:

This alternative O, that you might encounter elsewhere, provides an asymptotic upper bound on a function.

$f(n) \in O(\,g(n))$ if there are positive constants n_0 and c such that for $\mathrm{n} > n_0$, $f(n) \leq c(g(n))$

With this alternative definition, $128x^2 + 564x - 76 \in O(x^3)$ becomes true.

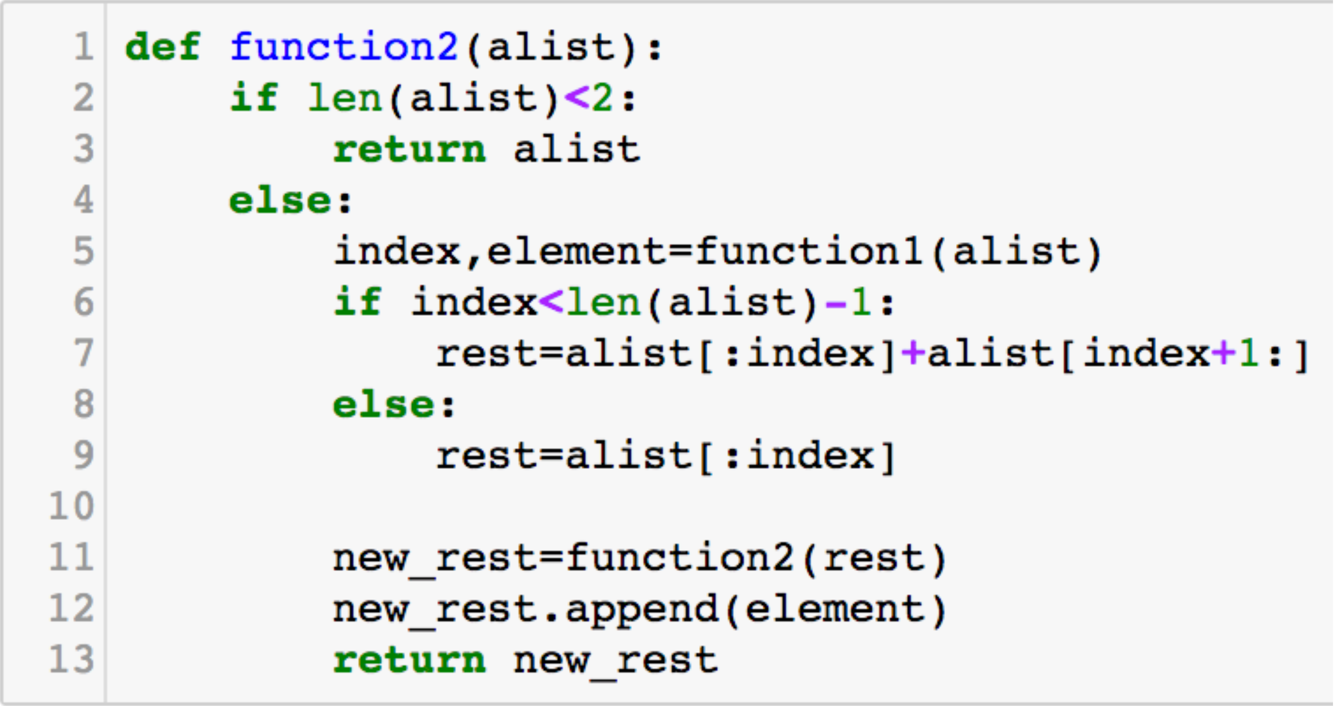
DO NOT use this definition in any of your work in this module!

Ω Notation

Ω notation bounds a function from below

- $32n^2 + 3n \in \Omega(n^2)$
- $32n^2 + 3n \in \Omega(n)$
- $32n^2 + 3n \in \Omega(1)$

The Sorting Problem



```
def function2(alist):
    if len(alist)<2:
        return alist
    else:
        index,element=function1(alist)
        if index<len(alist)-1:
            rest=alist[:index]+alist[index+1:]
        else:
            rest=alist[:index]

        new_rest=function2(rest)
        new_rest.append(element)
        return new_rest
```

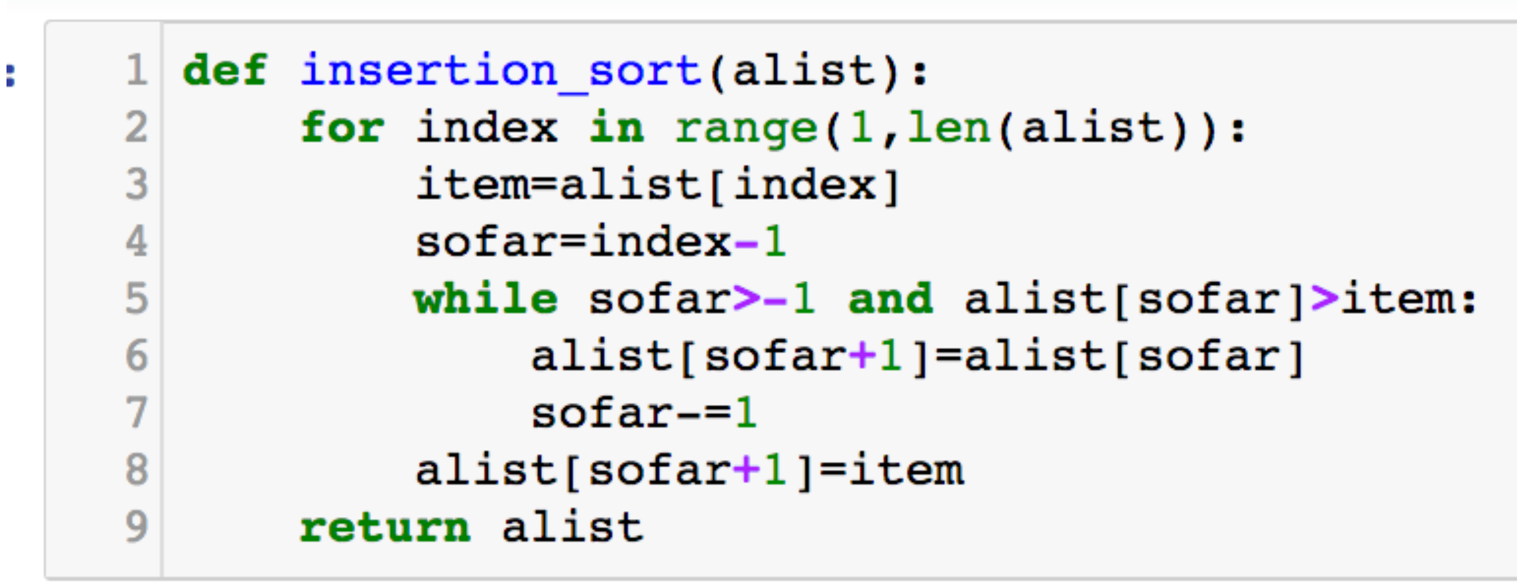
function1 here extracts the maximum element from the list

We saw this algorithm for sorting a list at the beginning of the lecture.

What’s the upper bound on its run time?

Can we do better?

Insertion-sort



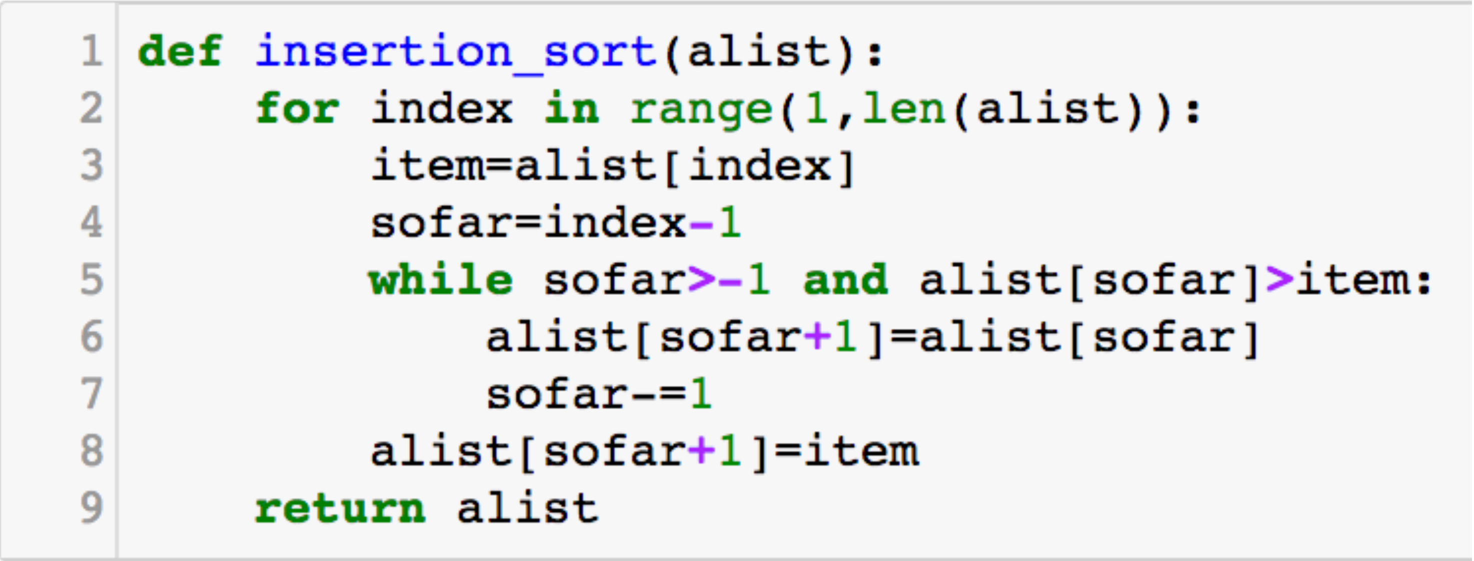
```python
def insertion_sort(alist):
    for index in range(1,len(alist)):
        item=alist[index]
        sofar=index-1
        while sofar>-1 and alist[sofar]>item:
            alist[sofar+1]=alist[sofar]
            sofar-=1
        alist[sofar+1]=item
    return alist
```

For each element in the list, find its correct place in the already sorted list to the left. Insert it (by shifting everything up) and proceed to next element.

	alist =
index=1	5, 2, 1, 6
index=2	2, 5, 1, 6
index=3	1, 2, 5, 6
	1, 2, 5, 6

What's the upper bound on running time for this algorithm?

Insertion-sort



```
def insertion_sort(alist):
    for index in range(1,len(alist)):
        item=alist[index]
        sofar=index-1
        while sofar>-1 and alist[sofar]>item:
            alist[sofar+1]=alist[sofar]
            sofar-=1
        alist[sofar+1]=item
    return alist
```

For each element in the list, find its correct place in the already sorted list to the left. Insert it (by shifting everything up) and proceed to next element.

alist =

index=1 5, 2, 1, 6

index=2 2, 5, 1, 6

index=3 1, 2, 5, 6

1, 2, 5, 6

What's the upper bound on running time for this algorithm? $O(n^2)$

But on average, faster than max_sort – why?

Divide-and-Conquer

Many useful algorithms are recursive in nature. They use the divide-and-conquer paradigm

1. **Divide** the problem into a number of subproblems which are similar to the original problem but smaller in size
2. **Conquer** the subproblems by solving them recursively. If the subproblems are small enough, solve them in an obvious way (the base case!)
3. **Combine** the solutions to the subproblems into the solution for the original problem

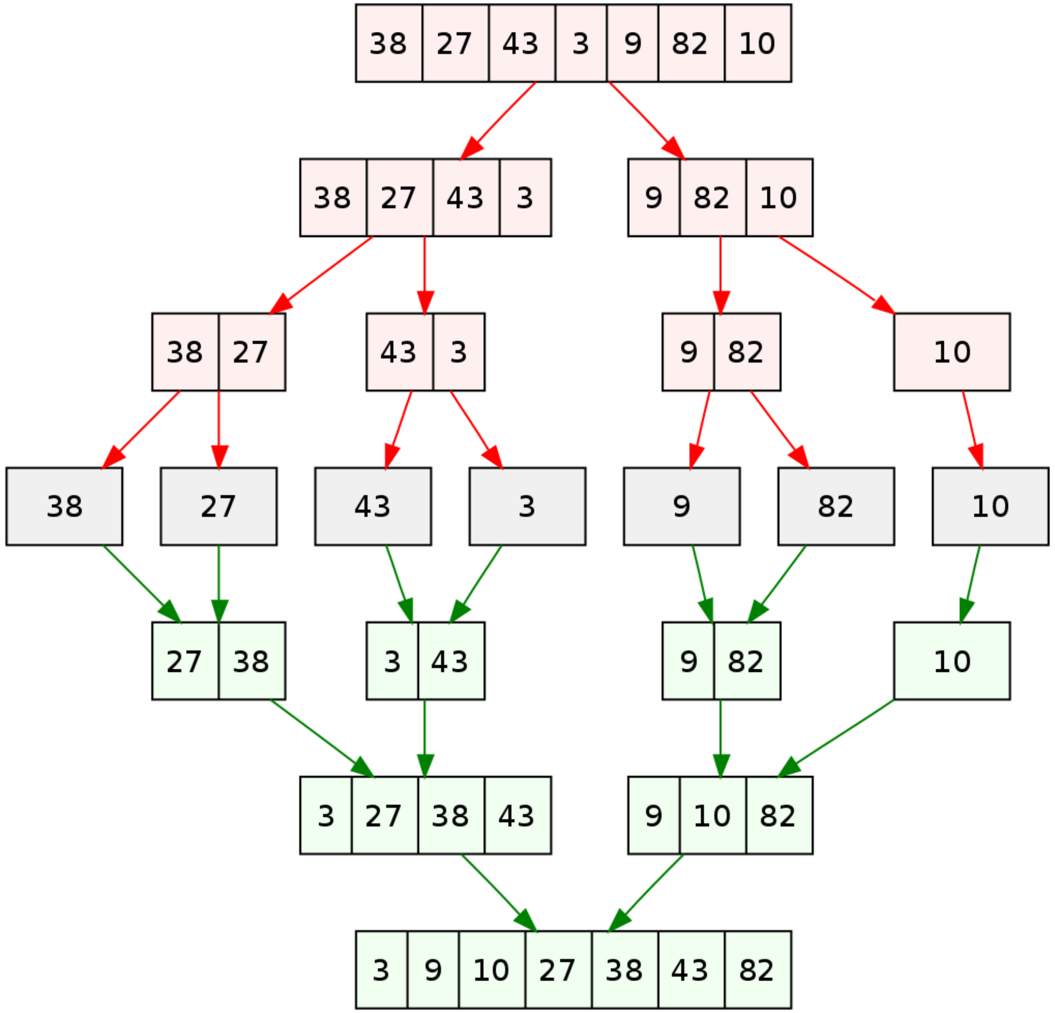
Merge-sort

1. If the length of the list < 2, it is sorted so return
2. Else, divide the list into 2 halves
3. Sort each list use merge-sort
4. **MERGE** the two sorted lists

What's the time complexity of this?

$T(n) = 2T(n/2) + T(\text{MERGE}(n))$

Example (from Wikipedia page on mergsort)



38 | 27 | 43 | 3 | 9 | 82 | 10

38 | 27 | 43 | 3   9 | 82 | 10

38 | 27   43 | 3   9 | 82   10

38   27   43   3   9   82   10

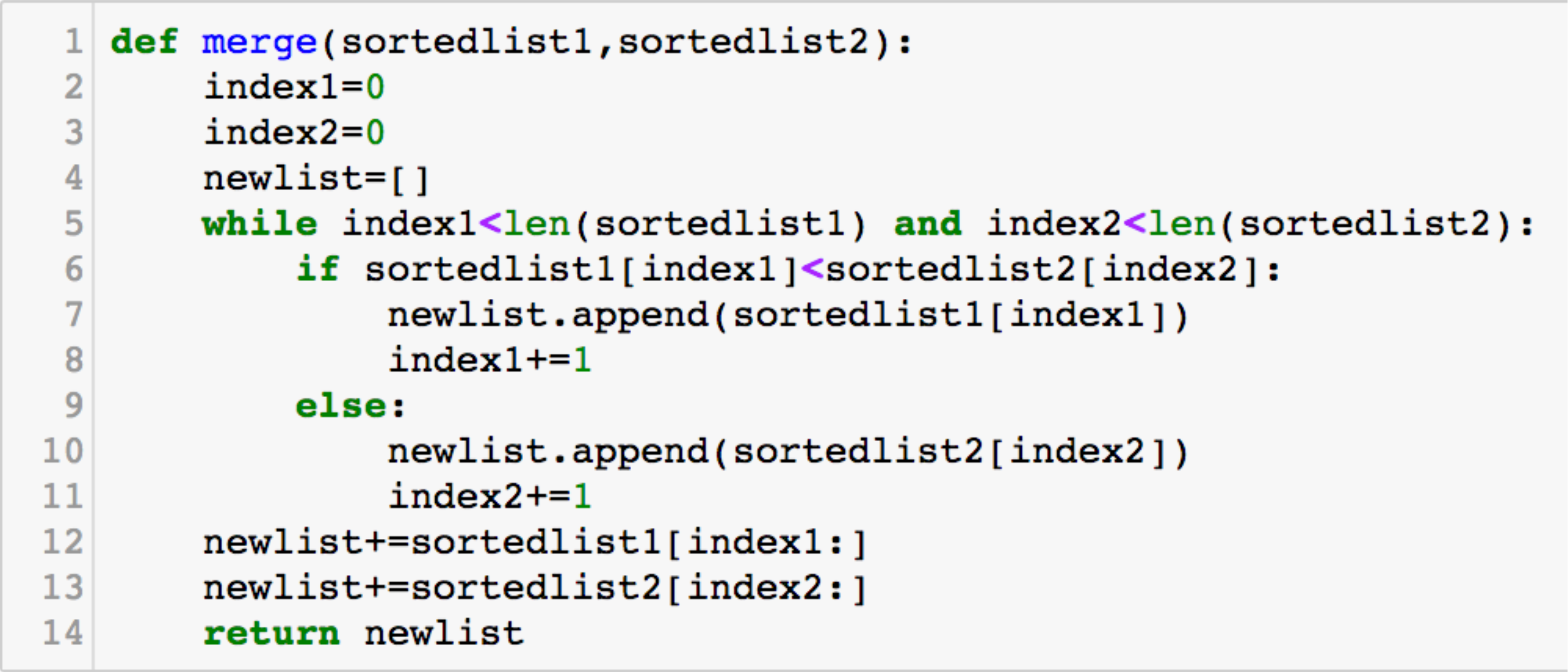
27 | 38   3 | 43   9 | 82   10

3 | 27 | 38 | 43   9 | 10 | 82

3 | 9 | 10 | 27 | 38 | 43 | 82

By Vineet Kumar at English Wikipedia - Transferred from en.wikipedia to Commons by Eric Bauman using CommonsHelper.

Merging 2 sorted lists



```python
def merge(sortedlist1,sortedlist2):
    index1=0
    index2=0
    newlist=[]
    while index1<len(sortedlist1) and index2<len(sortedlist2):
        if sortedlist1[index1]<sortedlist2[index2]:
            newlist.append(sortedlist1[index1])
            index1+=1
        else:
            newlist.append(sortedlist2[index2])
            index2+=1
    newlist+=sortedlist1[index1:]
    newlist+=sortedlist2[index2:]
    return newlist
```

At most, we check and append each element in each list once. So merging 2 sorted lists is **O(n)**

Analysis of merge-sort

Worst case running time for n elements (if n=2^k and k>1):

$$T(n) = 2T\left(\frac{n}{2}\right) + const \times n$$

Using mathematical induction we can show:

$$T(n) = O(\,n \log n\,)$$

[Get log n copies of const x n term, and other term is $2^{log_2 n}T(1) = O(n)$]

Other sorting algorithms

- Other comparison-sorting algorithms exist e.g.,:
    - Quicksort : $O(n^2)$ but on average faster, provided the input is in a random order
    - Heapsort : $O(n \log n)$
- It can be shown that the lower bound for comparison sorting is $O(n \log n)$. Mergesort and Heapsort are asymptotically optimal
- Sorting can be achieved in linear time if it can be assumed that the input elements are drawn from a limited domain or have a certain distribution.

Optional exercise: (a) Study heapsort, and convince yourself it is $O(n \log n)$
(b) How much better is it to start from a binary search tree than a hash table data structure if you're going to run heapsort on a dictionary?

Common Run-times

$O(n)$	Computing the maximum of a list of numbers	Merging 2 sorted lists
$O(n \log n)$	Mergesort	Many binary divide-and-conquer algorithms
$O(n^2)$	Find all-pairs distances between all points in a list	Quicksort
$O(n^3)$	Matrix multiplication	Enumerate all possible sets of size 3 from a list of objects
$O(n^k)$	Enumerate all possible sets of size k from a list of objects	
$O(10^n)$	Trying to break a numerical password of length n by trying every possible combination	

Space Complexity (further reading, to be covered later)

- The space complexity of an algorithm A:X → Y for input *x* is the number of memory cells it takes to execute A(x)
- For a fixed notion of size, then a space bound of A:X→ Y is a function $space$ that for all $x \in X$,

$$\text{the memory consumption of A(x)} \leq space(size(x))$$

We can define complexity classes for space in the same way as for time. We can refer to **linear-space** algorithms and **polynomial-space** algorithms

Relationship between Time and Space

- You cannot use more space than time (think about why).
- Therefore, for all A:X→Y the **set of all possible time bounds** is a **subset** of the **set of all possible space bounds**.
- Your *O* notation complexity for space will be something that grows no quicker than your *O* notation complexity for time.

Time-space-tradeoff

- Ideally, we want to minimise time complexity and space complexity
- Sorting can be performed in place so lower bound on space complexity for sorting = O(n)
- In practice, there is usually a tradeoff between time and space complexity. We may use more space to get a faster run-time.
- Insertion-sort sorts in place so space complexity = $O(n)$ and time complexity = $O(n^2)$
- Merge-sort creates new lists at each level of list division so a naïve implementation has space complexity = $O(n \log n)$ and time complexity = $O(n \log n)$

Communication Complexity

- How many packets of data need to be exchanged if processors jointly solve a problem in parallel?
- How does this depend on the size of the problem instance?

Alice and Bob

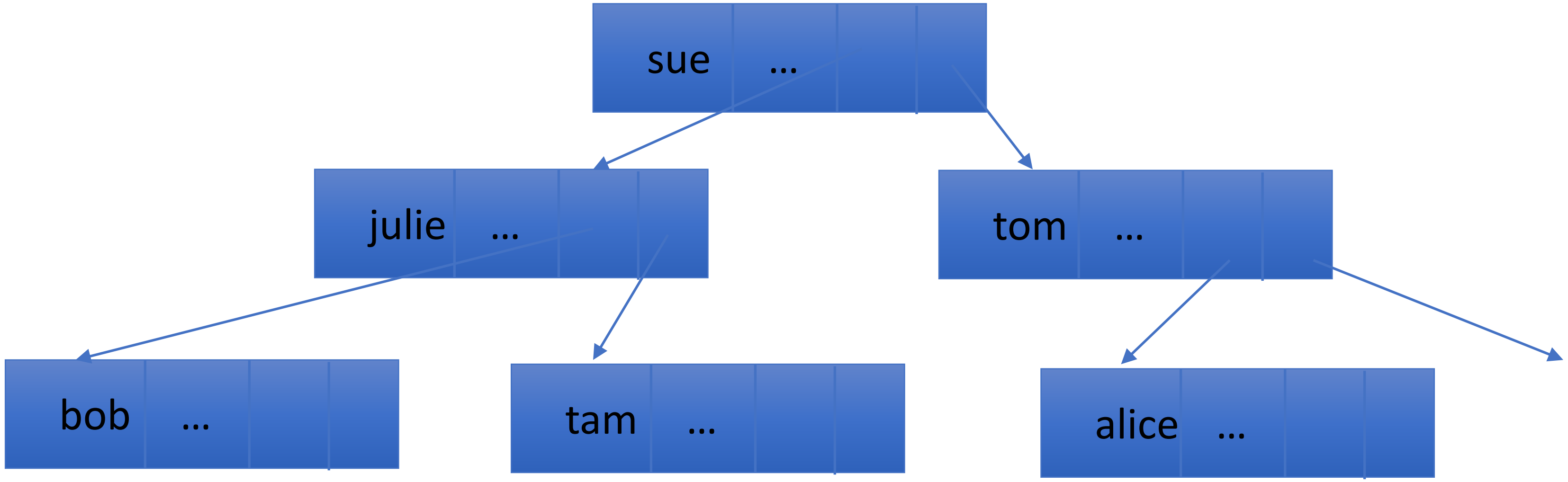
- Alice and Bob are two separated parties.
- Alice receives an n-bit string x and Bob receives an n-bit string y.
- The goal is for one of them to compute a certain function f(x,y).
- How many bits of data need to be communicated between them?
- Obviously, they will succeed if Alice sends Bob n bits (or Bob sends Alice n bits) who then computes f(x,y)
- Is it possible to calculate f with fewer than n bits of communication?
- Sometimes yes – see future lecture on Map/Reduce

Summary

- What have you learnt today?
    - Time complexity and O notation
    - Analysed sorting algorithms, recursion as a tool.
    - Space complexity and communication complexity.

Heaps and the heapsort algorithm

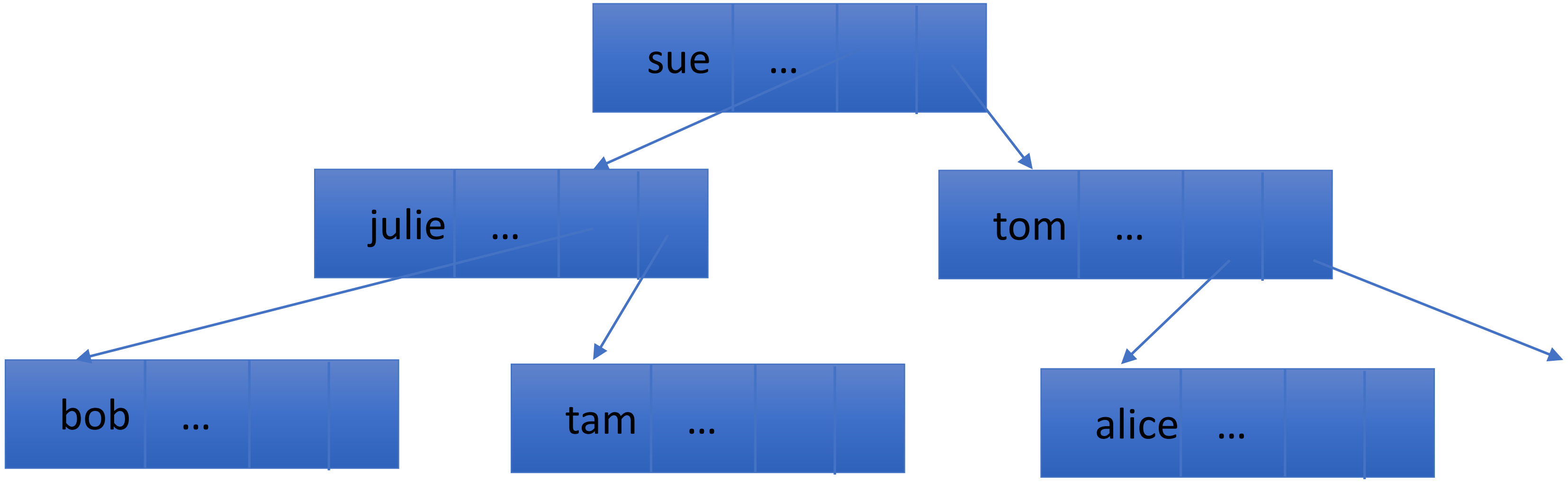
- Heaps are **complete binary trees** which satisfy the **heap property**: Value[parent(i)] ≥ Value[i]



- This is a complete binary tree. Does it satisfy the heap property?

Heaps

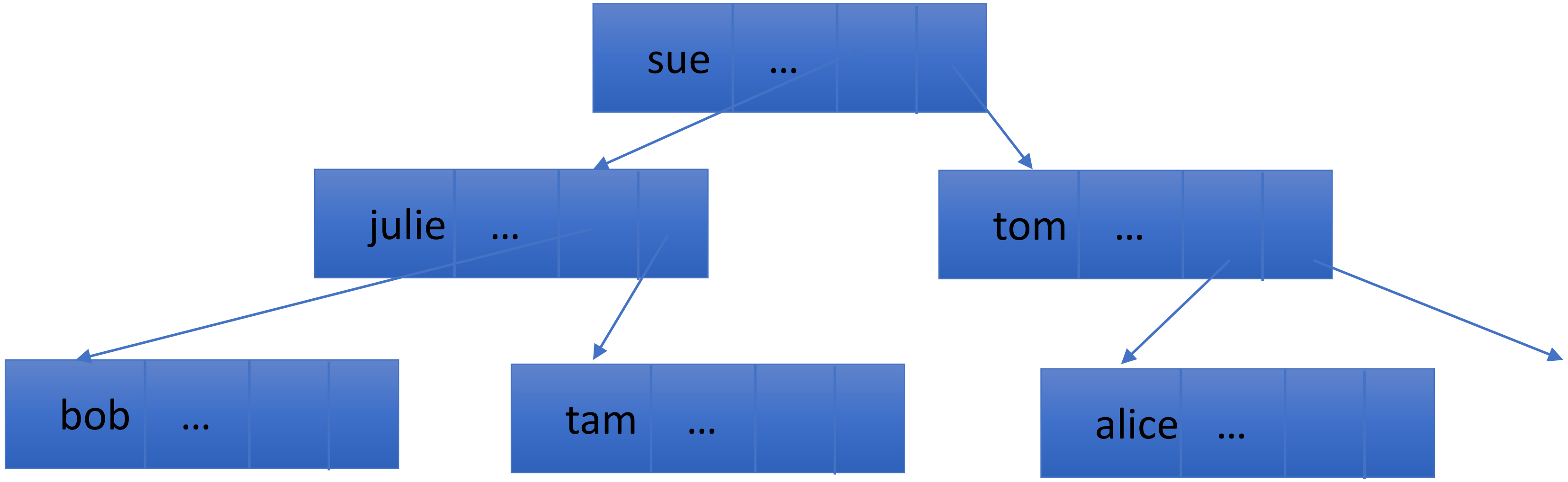
- Heaps are **complete binary trees** which satisfy the **heap property**: Value[parent(i)] ≥ Value[i]



- Run heapify on all non-leaf nodes in a bottom-up fashion i.e.
  - ➢ heapify([tom, julie, sue])

Heaps

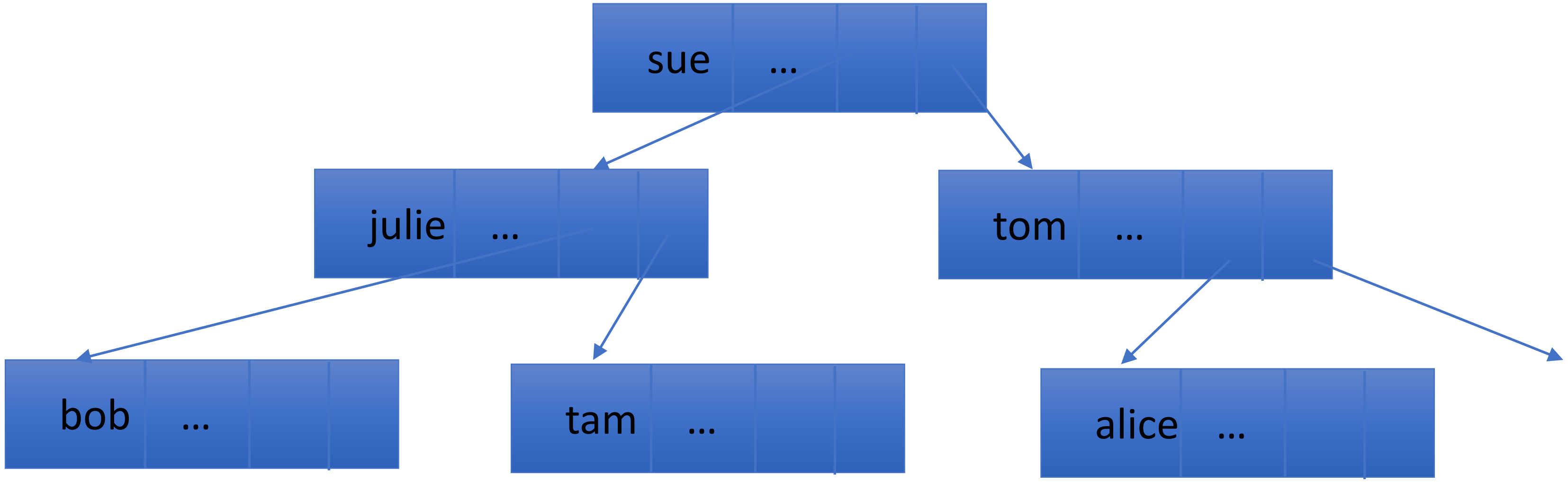
- Heaps are **complete binary trees** which satisfy the **heap property**: Value[parent(i)] ≥ Value[i]



- heapify(tom) :- tom > alice so heap property satisfied.

Heaps

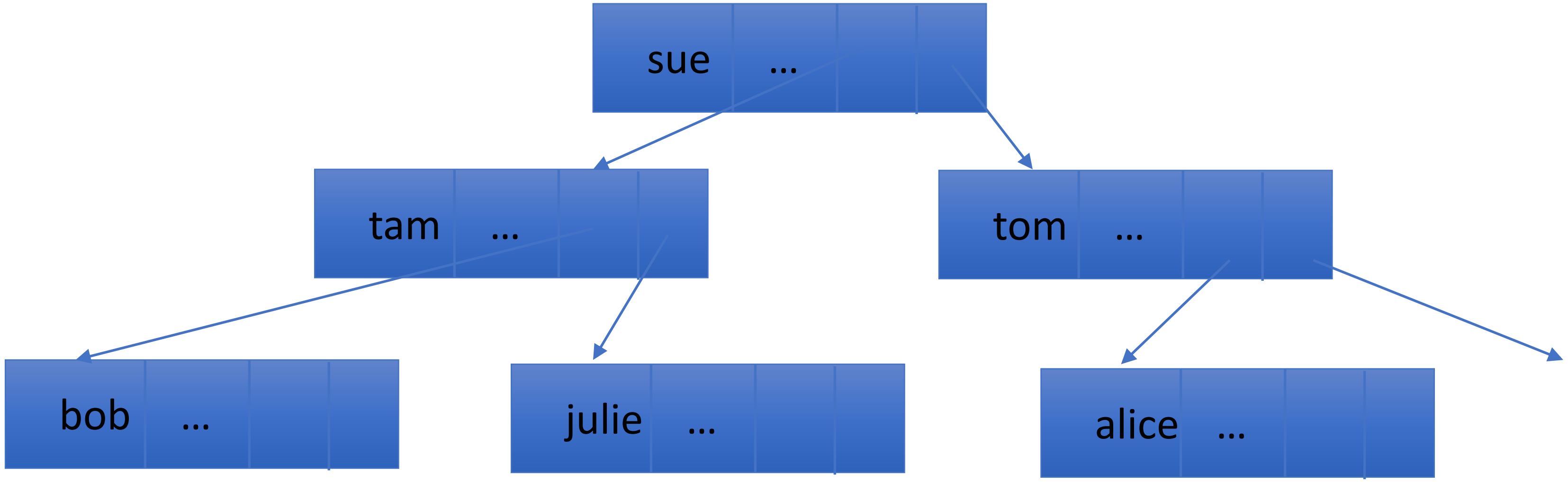
- Heaps are **complete binary trees** which satisfy the **heap property**: Value[parent(i)] ≥ Value[i]



- heapify(julie)
  1. **tam > julie so swap these nodes**
  2. heapify(julie)

Heaps

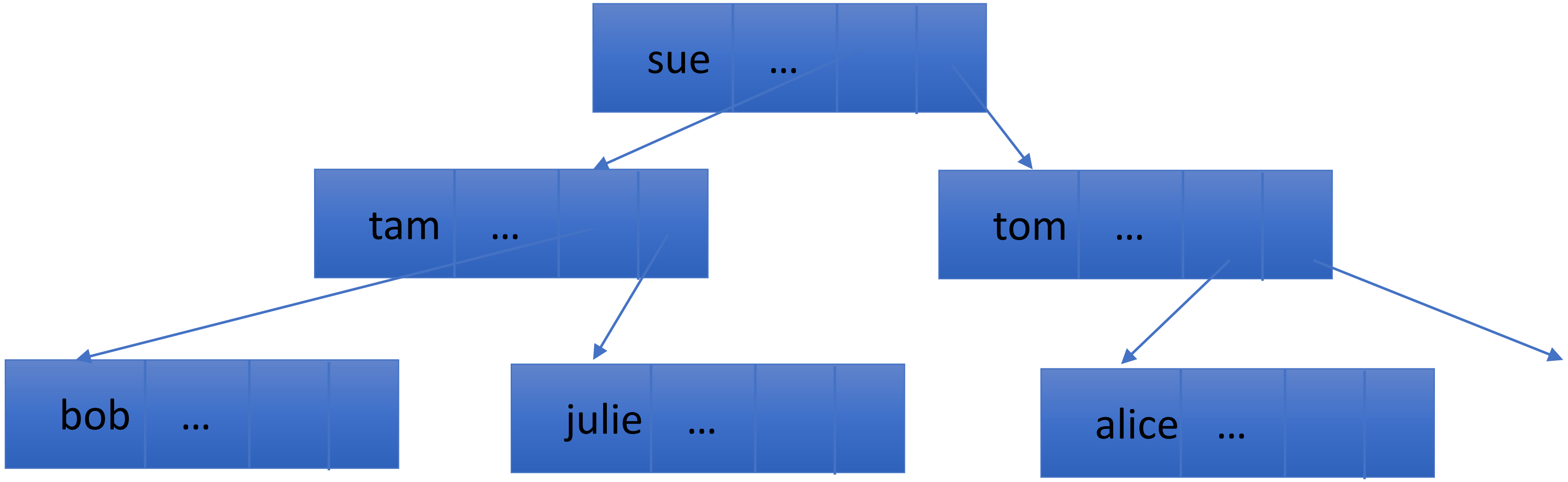
- Heaps are **complete binary trees** which satisfy the **heap property**: Value[parent(i)] ≥ Value[i]



- heapify(julie)
  1. tam > julie so swap these nodes
  2. **heapify(julie)**

Heaps

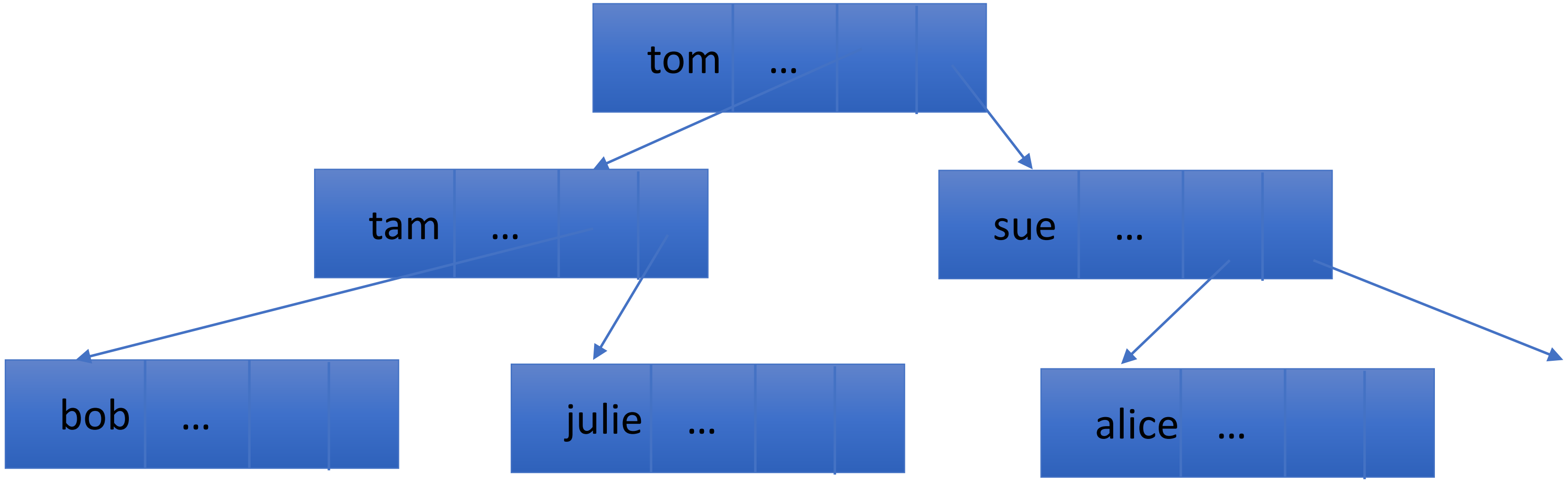
- Heaps are **complete binary trees** which satisfy the **heap property**: Value[parent(i)] ≥ Value[i]



- heapify(sue) :-
    1. **tom > sue & tam so swap sue and tom**
    2. heapify(sue)

Heaps

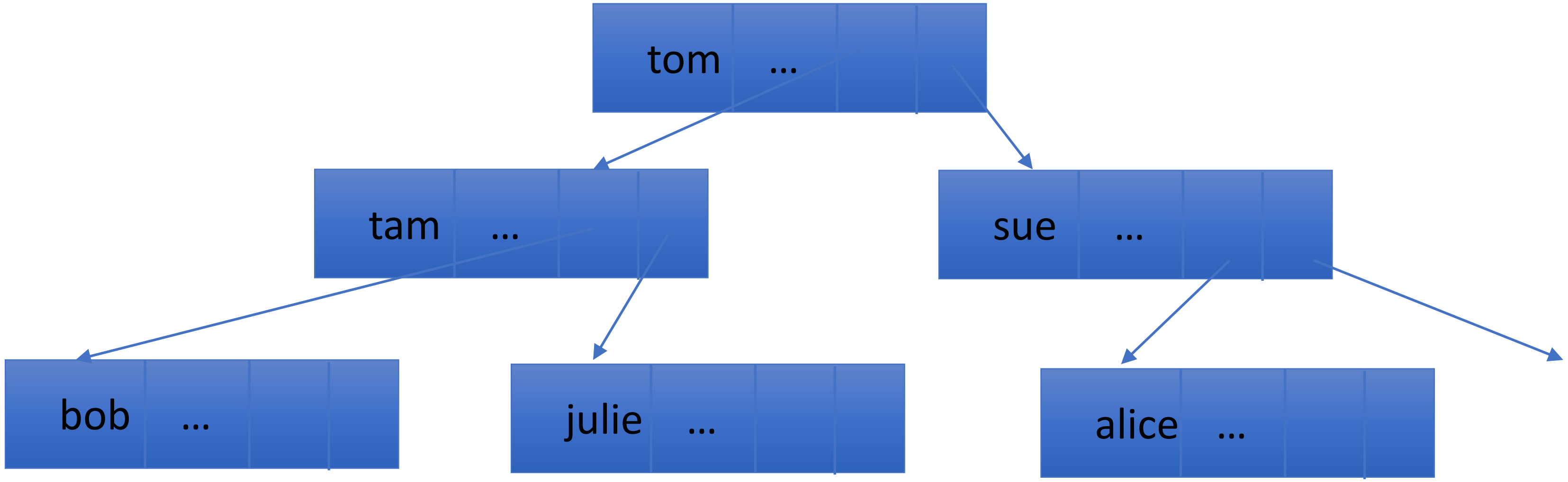
- Heaps are **complete binary trees** which satisfy the **heap property**: Value[parent(i)] ≥ Value[i]



- heapify(sue) :-
  1. tom > sue & tam so swap sue and tom
  2. **heapify(sue)**

Heaps

- Heaps are **complete binary trees** which satisfy the **heap property**: Value[parent(i)] ≥ Value[i]



- heapify(sue) :- sue > alice so heap property satisfied

Why are heaps useful?

- Can be used for sorting (the Heapsort algorithm)
  - Remove maximum element at root of heap
  - Take last element, place it at the root and then heapify
  - Repeat
- Efficient implementation of a ‘priority queue’
- O(*n* log *n*) to build a heap