

Week 2: Algorithmic complexity

Algorithmic Data Science

2025-26

```

1 def function1(alist):
2     pos=0
3     sofar=alist[0]
4     for (index,thing) in enumerate(alist):
5         if thing>sofar:
6             pos=index
7             sofar=thing
8
9     return pos,sofar
10

```

```

1 def function2(alist):
2     if len(alist)<2:
3         return alist
4     else:
5         index,element=function1(alist)
6         if index<len(alist)-1:
7             rest=alist[:index]+alist[index+1:]
8         else:
9             rest=alist[:index]
10
11         new_rest=function2(rest)
12         new_rest.append(element)
13     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?

Main topics per week

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

Today:

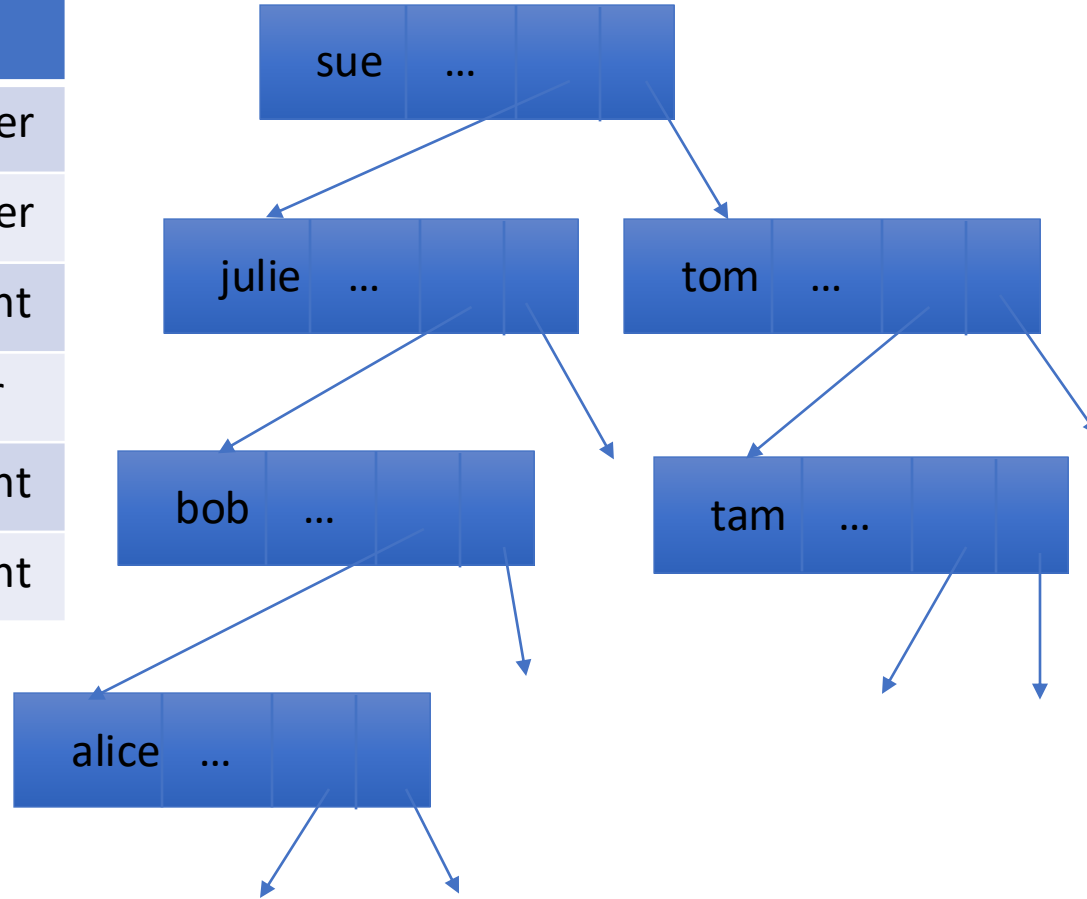
1. Finishing up Topic 1: Data Structures and Data formats
2. Algorithmic Complexity. Sorting algorithms.

1. Finishing up Topic 1: Data Structures and Data formats

Binary Search Trees

occupations = { }

key	value
sue	lecturer
julie	lecturer
tom	student
bob	porter
tam	student
alice	student



height	nodes	capacity
0	1	1
1	2	3
2	4	7
3	8	15
h	2^h	$2^{h+1}-1$

Binary Search Trees

- balanced tree -> number of nodes at height h is 2^h
- capacity $n = \sum_{i=0}^h 2^i = 2^{h+1} - 1$
- query runs in $O(\log_2 n)$ time
- unbalanced tree will result if the keys are pre-sorted -> linked list and query runs in $O(n)$ time
- useful data structure when there is a strict order over the keys but keys will be presented in random order
- can be used to implement dictionaries

Example

- Suppose a librarian wants to know how many books in their library have an author that starts with A.
- Would it be quicker if the dictionary were stored in a hash table or a binary search tree?

Topic 2: Algorithmic complexity. Sorting.

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** interchangeably 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

```
1 def find_max_element(alist):  
2     sofar=alist[0]  
3     for thing in alist[1:]:  
4         if thing>sofar:  
5             sofar=thing  
6     return sofar  
7
```

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 $\forall x \in X, \forall y \in Y$:

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

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

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

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} , $c_{\text{if-then-else}}$, ... such that
 - Executing an addition always takes c_{add} seconds
 - Executing a subtraction always takes c_{subtract} seconds
 - Executing a conditional always takes $c_{\text{if-then-else}}$ seconds
- These constants rely heavily on the chosen programming language, CPU, compiler, clockspeed etc

Constant-time

Problem size $n = \text{len}(\text{alist})$

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

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

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: **$O(1)$**

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

Iterations: Linear-time

Problem size $n = \text{len}(\text{alist})$

```
1 def myfunc1(alist):  
2     for thing in alist:  
3         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 $n = \text{len}(\text{alist})$

This internal loop is carried out n times

```
: 1 def myfunc2(alist):  
  2     for thing1 in alist:  
  3         for thing2 in alist:  
  4             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 $n = \text{len}(\text{alist})$

```
1 def myfunc1b(alist, something):  
2     for thing in alist:  
3         print(thing)  
4         if thing==something:  
5             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

Assume an algorithm has run-time

$$f(n) = 3 \times 2^n + 2345n^7 + 8934n^5 + 163 \sin(90435 + n)$$

What is a good approximation to f for big data?

(A)

$$f(n) \approx \text{const} \cdot 2^n$$

0%

(B)

$$f(n) \approx \text{const} \cdot n^7$$

0%

(C)

$$f(n) \approx \text{const} \cdot n^5$$

0%

(D)

$$f(n) \approx \text{const} \cdot (2^n + n^7 + n^5)$$

0%

Asymptotic order of growth

Assume an algorithm has run-time

- $f(n) = 3 \times 2^n + 2345n^7 + 8934n^5 + 163 \sin(90435 + n)$

What is a good approximation to f for big data?

$$f(n) \approx \text{const} * 2^n$$

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	$n \log n$	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.

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:

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

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.

Time complexity O Notation: Loose definition.

A run-time is $O(g(n))$ if:

For sufficiently large data, $\text{const} \times g(n)$ approximates the run-time well,

and the approximation generally gets better and better the bigger the data.

Example: $17n^6 + 15n^4$ is $O(n^6)$

Here $g(n)$ is n^6 and the constant is 17:

The bigger n gets,
the better $17n^6 = 17g(n)$ approximates
the run-time.

O Notation

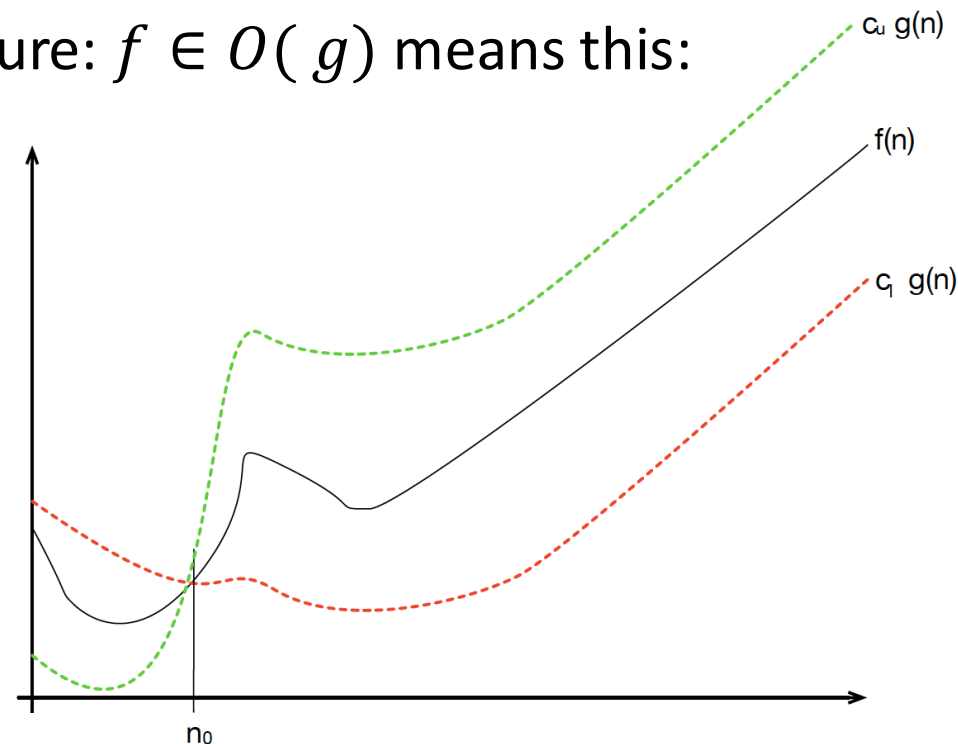
O notation asymptotically bounds a function from above and below.

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

$$c_1(g(n)) \leq f(n) \leq c_2(g(n))$$

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

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 .

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 $n > n_0$, $f(n) \leq c(g(n))$

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

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

What is the time complexity of find_max_element?

```
1 def find_max_element(alist):
2     pos=0
3     sofar=alist[0]
4     for (index,thing) in enumerate(alist):
5         if thing>sofar:
6             pos=index
7             sofar=thing
8
9     return pos,sofar
10
```

O(1)

0%

O(n)

0%

O(n²)

0%

O(2ⁿ)

0%

The Sorting Problem

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

Function: `max-sort(alist)`

1. If the length of `alist` < 2 , it is sorted so `return alist`
2. Else,
 - 2.1 search `alist` from beginning to end and find the maximum element and its position (call these `element` and `i`).
 - 2.2 Remove `element` from `alist`
 - 2.3 `sortedlist = max-sort(alist with max element removed)`
 - 2.4 `sortedlist.append(element)`
 - 2.5 `return sortedlist`

What's the upper bound on its run time?

Can we do better?

Exercise: Go back to the beginning of the lecture notes and understand the Python code that implements this pseudo code.

Insertion-sort

```
: 1 def insertion_sort(alist):
  2     for index in range(1, len(alist)):
  3         item = alist[index]
  4         sofar = index - 1
  5         while sofar > -1 and alist[sofar] > item:
  6             alist[sofar + 1] = alist[sofar]
  7             sofar -= 1
  8         alist[sofar + 1] = item
  9     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

```
: 1 def insertion_sort(alist):
  2     for index in range(1, len(alist)):
  3         item = alist[index]
  4         sofar = index - 1
  5         while sofar > -1 and alist[sofar] > item:
  6             alist[sofar + 1] = alist[sofar]
  7             sofar -= 1
  8         alist[sofar + 1] = item
  9     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

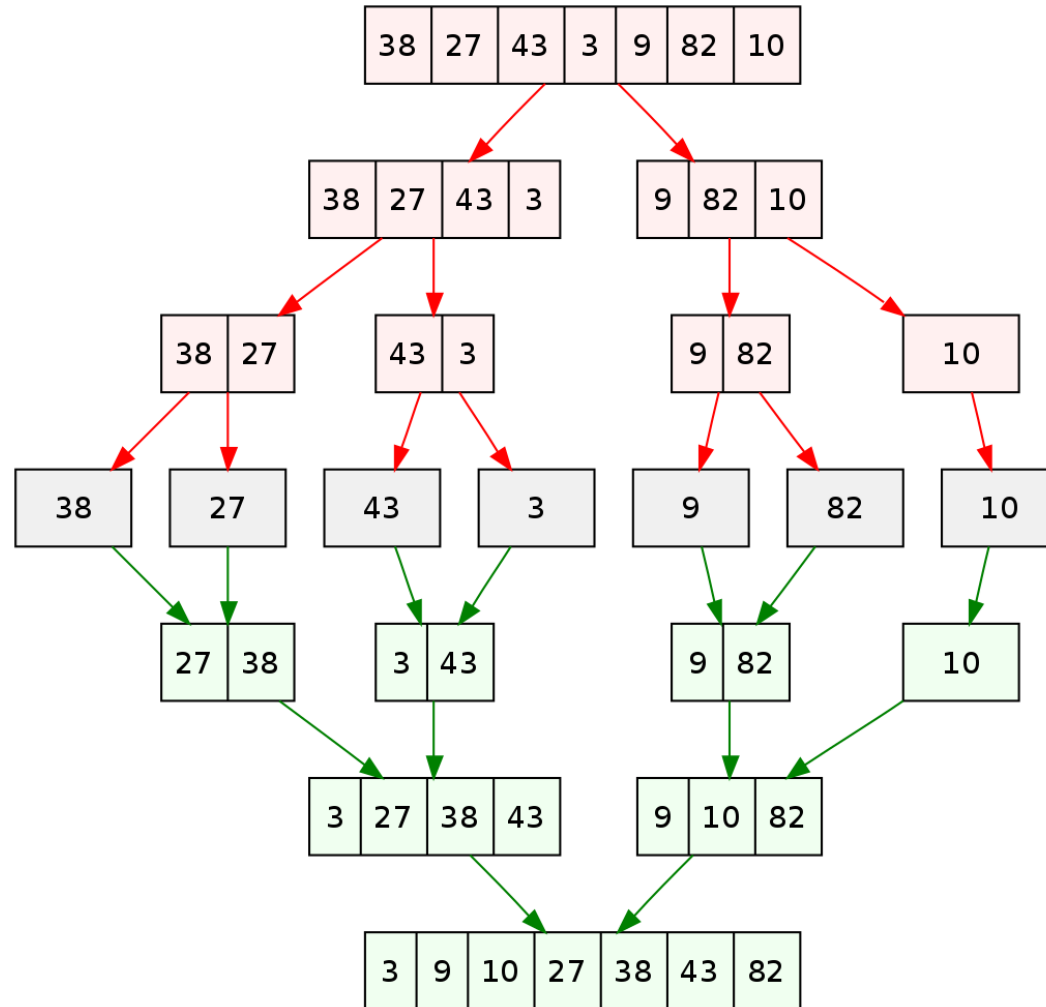
Merge-sort

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 mergesort)



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

Merging 2 sorted lists

`newlist=[]`

`counter1=0`

`counter2=0`

While `counter1 < length(list1)` and `counter2 < length(list2)`

Compare `list1[counter1]` and `list2[counter2]`

Append the smaller of these to `newlist`.

Add 1 to the corresponding counter.

One list will have some stuff left over - append this to `newlist`.

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

```
1 def merge(sortedlist1,sortedlist2):
2     index1=0
3     index2=0
4     newlist=[]
5     while index1<len(sortedlist1) and index2<len(sortedlist2):
6         if sortedlist1[index1]<sortedlist2[index2]:
7             newlist.append(sortedlist1[index1])
8             index1+=1
9         else:
10            newlist.append(sortedlist2[index2])
11            index2+=1
12    newlist+=sortedlist1[index1:]
13    newlist+=sortedlist2[index2:]
14    return newlist
```

Exercise: Go through this Python code and understand it.

Analysis of merge-sort

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

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

Using mathematical induction we can show:

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

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

Other sorting algorithms

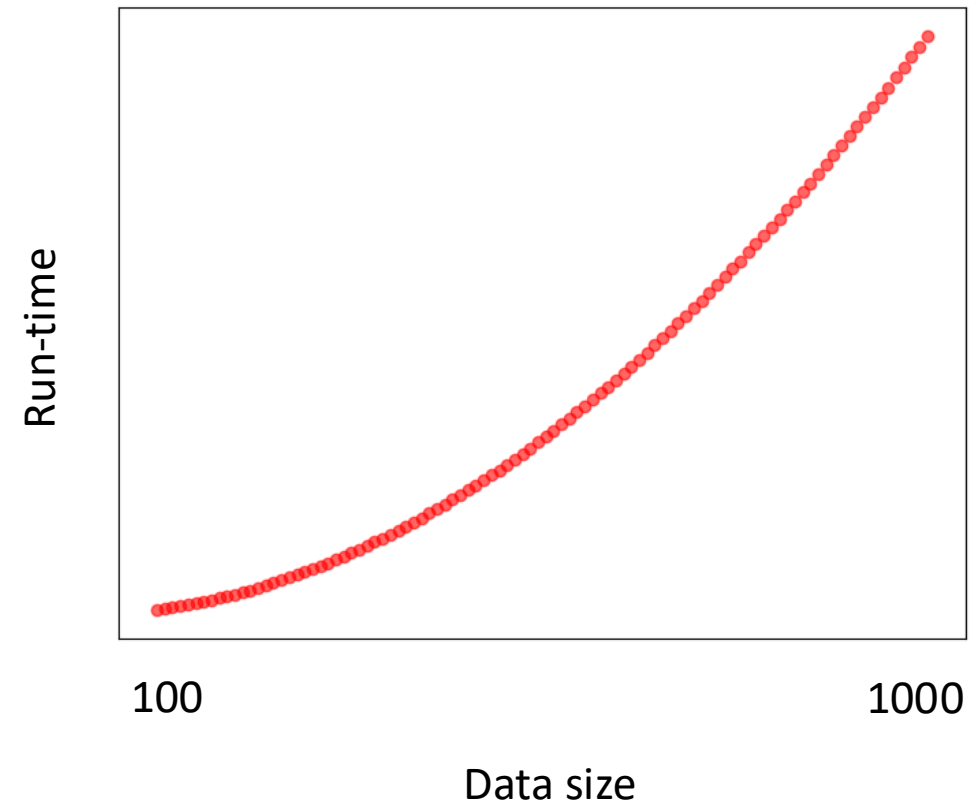
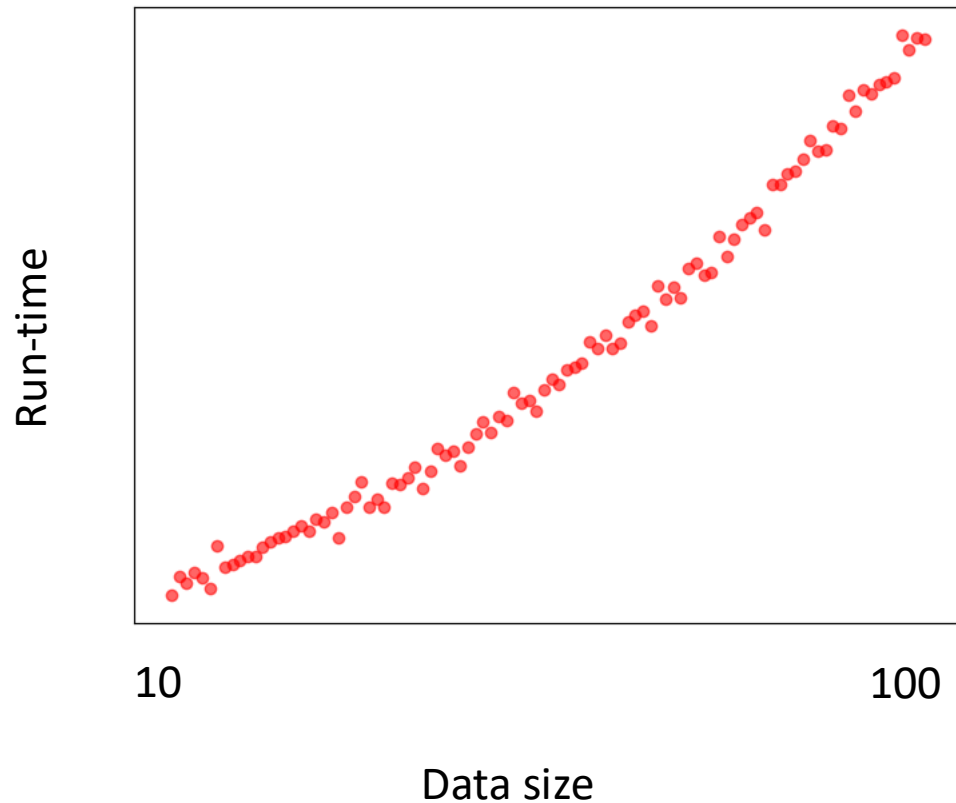
- Python uses *Timsort*, a hybrid of something similar to *mergesort* and something similar to *insertion sort*.
- It can be shown that the lower bound for comparison sorting is $O(n \log n)$. *Mergesort* and *Heapsort* (which we'll meet in a couple of weeks) are **asymptotically optimal**.
- Sorting 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.

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	

Empirically testing the run-time

- In the labs you will plot run-time against data size.
- Here is one that for small n looks maybe linear, but for larger n , we see that it is not-linear.



- If we assume this is $O(n^\alpha)$, how do we find α ?

Empirically testing the run-time

If t is $O(n^\alpha)$ then the run-time is the following, for some constant c :

$$t = cn^\alpha$$

Take logs of both sides of the equation:

$$\log(t) = \log(cn^\alpha)$$

Using $\log(xy) = \log(x) + \log(y)$

And $\log(a^b) = b \log(a)$

$$\log(t) = \log(c) + \alpha \log(n)$$

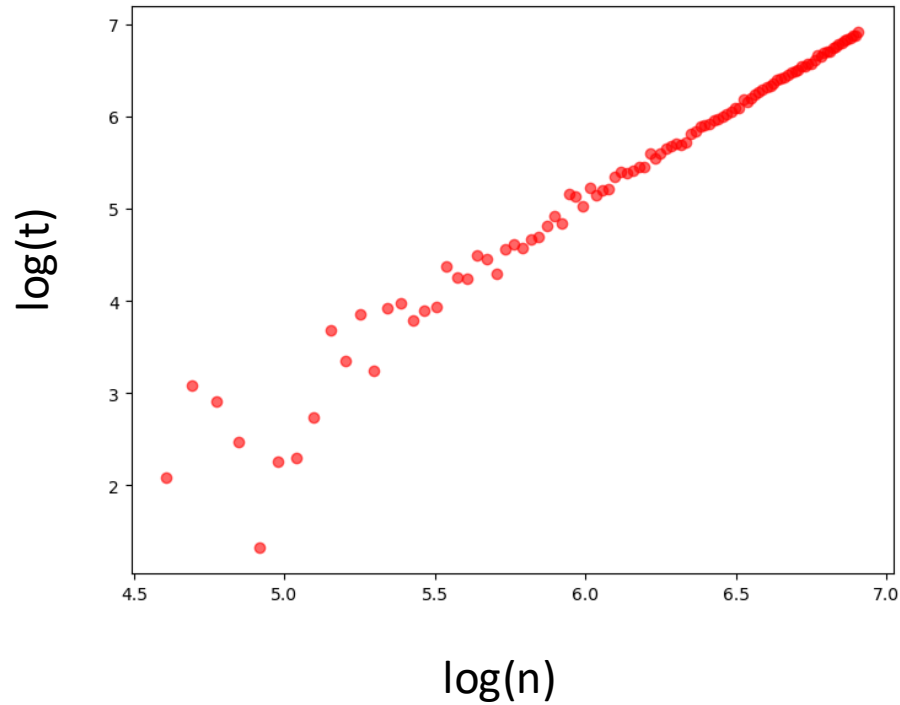
This means that if I plot $y = \log(t)$ against $x = \log(n)$, I will get a straight line, with α being the gradient (slope) and $\log(c)$ being the intercept:

$$y = \log(c) + \alpha x$$

You can then use a stats library to find the gradient and intercept and hence α and the constant.

Example

Doing this for the example:



Use a stats library to find the slope and the intercept.

Can tell by eye that the slope is roughly 2 for large n , which is what we're interested in. So this is $O(n^2)$.

Summary

- What have you learnt today?
 - Time complexity and O notation
 - Analysed sorting algorithms, recursion as a tool.
 - Insertion sort
 - Mergesort
 - Heapsort