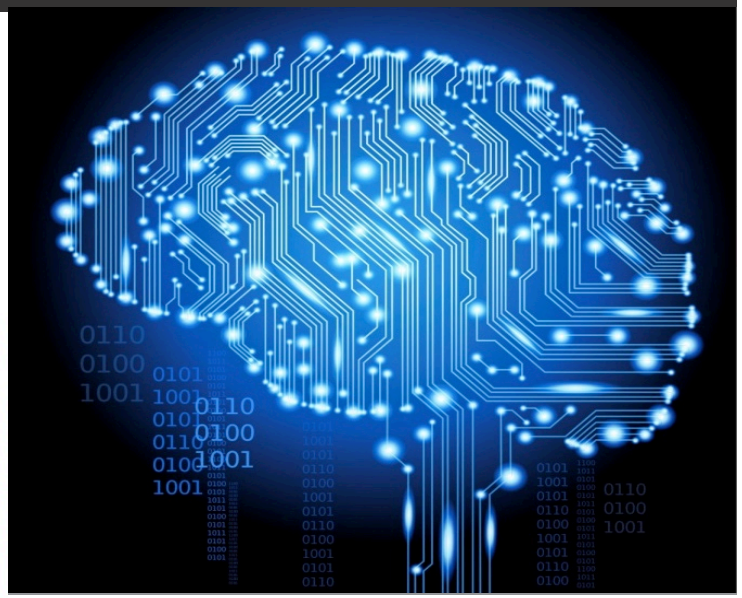
MONASH University

Information Technology

FIT1008/2085
Time Complexity

Prepared by:
Maria Garcia de la Banda
Revised by D. Albrecht, J. Garcia

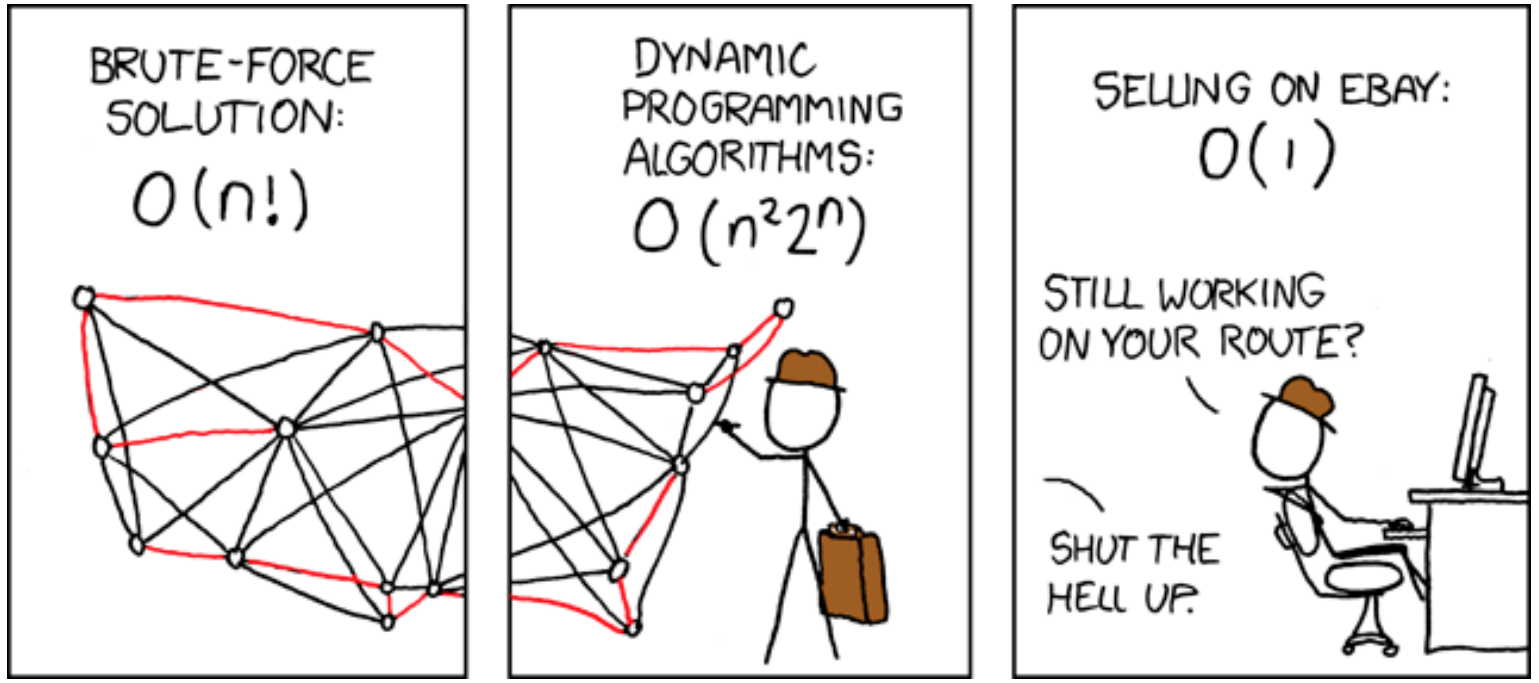


Where are we up to?

- **We can implement, use and modify the following sorting algorithms:**
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
- **We are able to determine important invariants of sorting algorithms and use them to improve their algorithms**
- **In particular, we can reason about the stability and incrementality of sorting algorithms**

Objectives for this lesson

- **To learn about running time, computational time complexity, and asymptotic analysis (Big-O notation) of time complexity**
  - To be able to compute the Big-O time complexity of simple functions
- **To reason about the Big-O complexity of three sorting algorithms:**
  - Bubble Sort
  - Selection Sort
  - Insertion Sort



https://xkcd.com/399/

MONASH University

Running Time and Computational Time Complexity

Running time

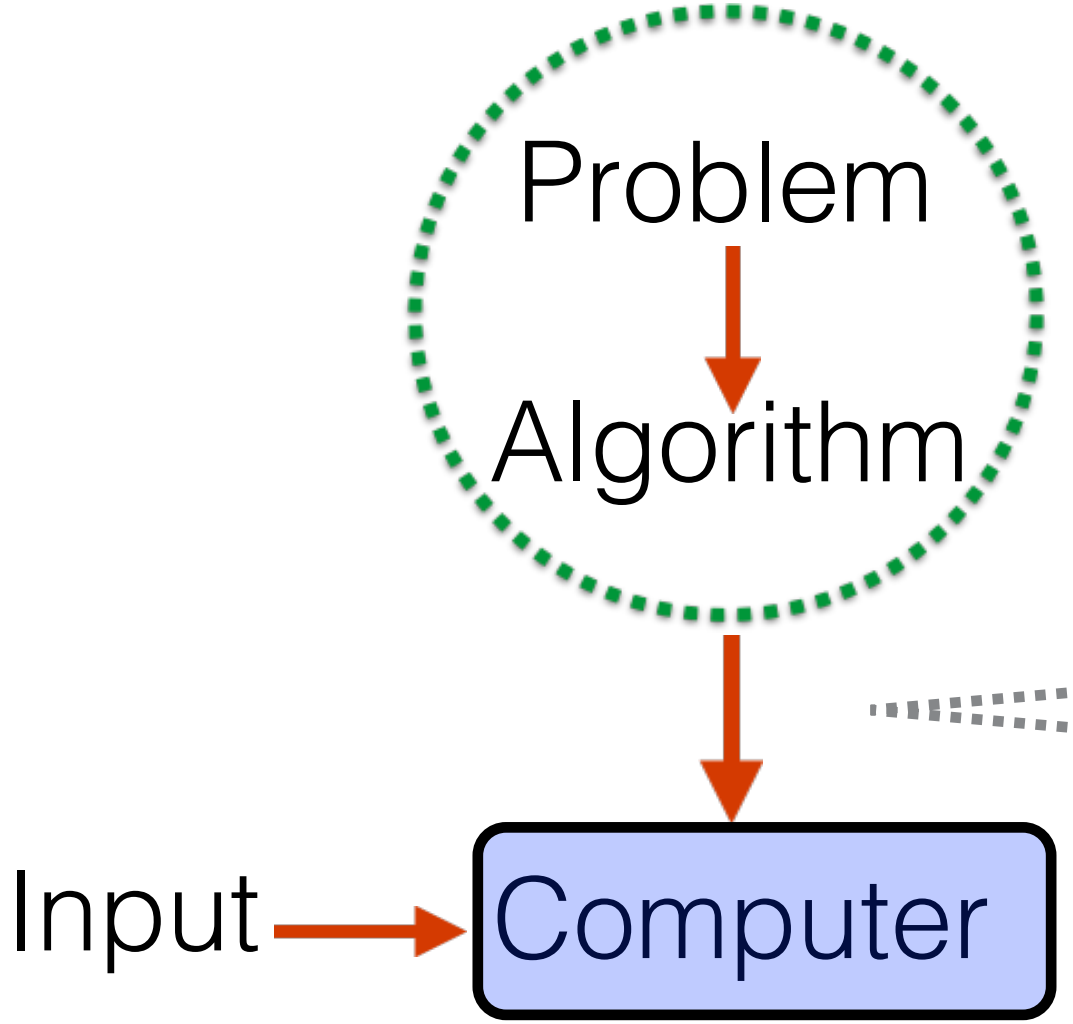
Depends on a number of factors including:

- **The input**
- **The quality of the code generated by the compiler**
- **The nature and speed of the instructions on the machine executing the program**
- **The time complexity of the algorithm**

Taking these things into account is difficult and not generalisable

Jamaica's Usain Bolt celebrating after winning the final of the men's 100 metres athletics event at the 2015 IAAF World Championships in Beijing. AFP PHOTO / PEDRO UGARTE

We are going to focus on this one



time complexity of the algorithm

We need some kind of abstraction, a "theoretical computer": **Computational model**

Simple Computational Model of Time Complexity

This number is the same for all compilers and machines

- **The computational (time) complexity of an algorithm is the number of elementary steps T(n) needed to compute its output for an input *of* size n**
- **Each simple statement/operation takes one "elementary step":**
  - Read, print and comparisons of numbers and Booleans
  - Python list/string/set access (e.g., list[i])
  - Assignments and basic numerical operations (e.g., x//2, x = i)
  - Return statements (e.g., return x)

Can think of an elementary step as a particular definition of a MIPS "block"

- **Sequence of statements: sum of their steps (e.g., x = list[i]//2 is 3 steps)**
- **If-then-else: sum of the test plus the steps of its branch(es)**
- **For a loop: sum of its statements times the number of iterations**
  - Careful with nested loops (multiply inner and outer loop's iterations)
- **Function calls: computed from its statements**

Example with Bubble Sort

The first assignment is outside the loop

```
def bubble_sort(the_list):
    n = len(the_list)          1 access and 1 assignment
    for _ in range(n-1):       2 assignments, 1 comparison, 1 increment
        for i in range(n-1):   2 assignments, 1 comparison, 1 increment
            if (the_list[i] > the_list[i+1]):   2 accesses, 1 comparison
                swap(the_list, i, i+1)          7 steps
```

n-1 times

n-1 times

Remember your for to while translation!

$1+1+1+(n-1)*(1+1+1+1+(n-1)*(1+1+1+2+1+7))=3+(n-1)*(4+(n-1)*13)) = 3+(n-1)*(13n-9)$
$= 3+(13n^2-22n+9) = 13n^2-22n+12$ "steps" for bubble sort ….

Wow!!! Is all this detail accurate? Useful?

```
def swap(the_list,i,j):
    tmp = the_list[i]          1 access and 1 assignment
    the_list[i] = the_list[j]  2 accesses and 1 assignment
    the_list[j] = tmp          1 access and 1 assignment
```

1+1+2+1+1+1=7 "steps" for swap

Don't worry, we will not be computing the exact number of steps of our algorithms!!

MONASH University

Big O Time Complexity

Big O notation for time complexity

- **The exact computational time function T(n), where n is the size of the input data, can be difficult to compute and understand**
- **Instead: compute an upper bound f(n) to T(n) that**
  - Ignores parts of **T(n)** that do not add significantly to the total running time
  - Bounds the error made when ignoring these small parts: simple but formal
- **Gives us a way of describing the growth rate of a method**
  - Behaviour when its input arguments grow towards infinity

Answers the question: what happens when my input gets really big?

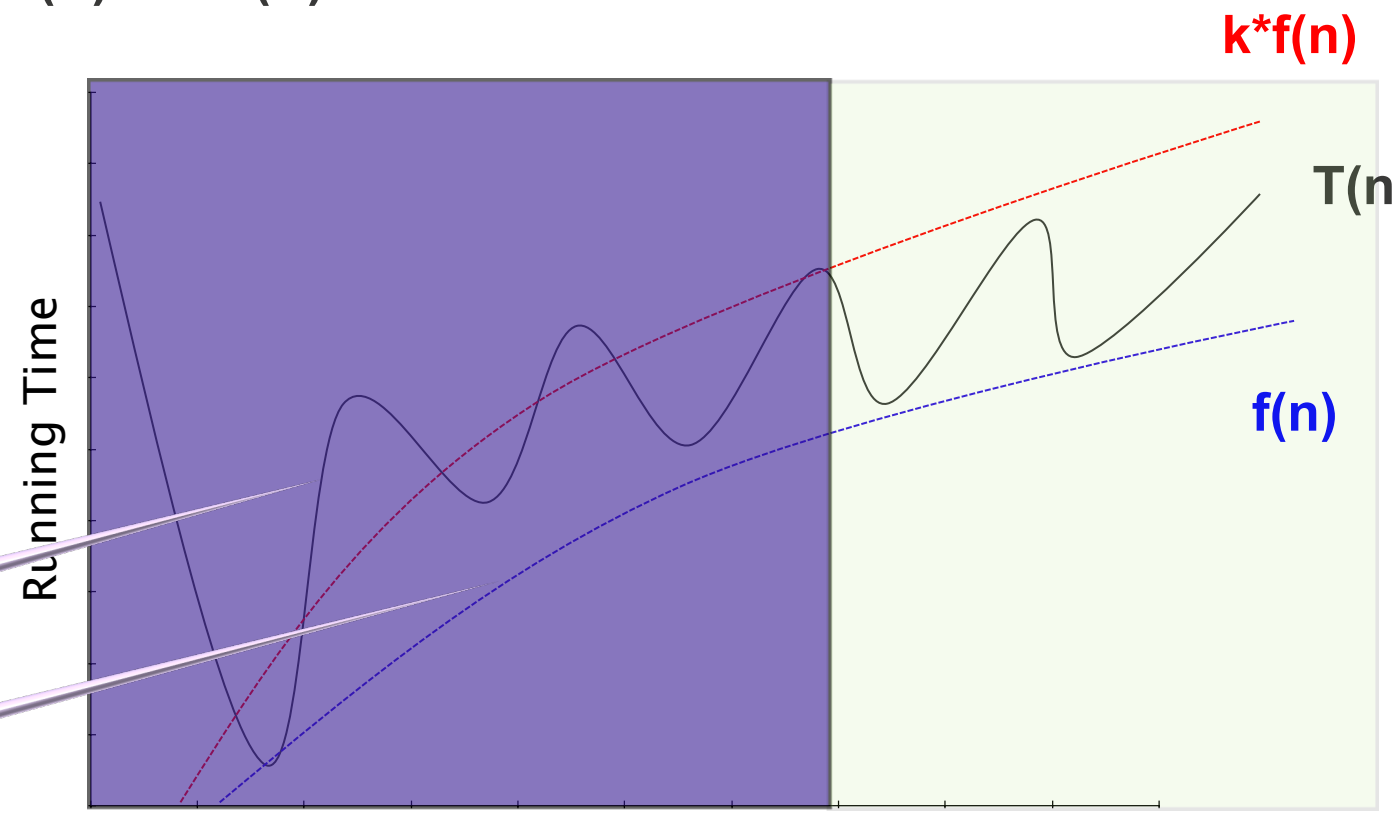
- **Formally: function T(n) is said to be O(f(n)) if there exist constants k and L such that**

$$T(n) \leq k*f(n) \text{ for all } n > L$$

Big O notation for time complexity

- **Function T(n) is said to be O(f(n)) if there exist constants k and L such that**

$$T(n) \leq k*f(n) \text{ for all } n > L$$



Big O gives us an idea of T(n)'s growth behaviour for large inputs. Simple but formal.

T(n) is not simple

f(n) is simple

Intuition: once n is large enough (greater than L), our much simpler function f(n) captures the growth of T(n), and when multiplied by k, is always greater than T(n)

How to compute Big O?

- **Transform all T(n) constants into 1**
  - It is not O(10n), it is just O(n)
- **Ignore parts that do not contribute significantly (always biggest factor)**
  - It is not O(n^3 + n^2 + n), just O(n^3)
- **Always assume an unknown input size n for each argument**
  - n will be large (measuring growth towards infinity)
- **Makes things much easier!**
  - Don't need to worry about the exact number of steps
- **Why can you do this?**
  - It is an upper bound

First time that 1+1 = 1 ☺

Back to Bubble Sort

```
def bubble_sort(the_list):
    n = len(the_list)                    constant
    for _ in range(n-1):                 constant
        for i in range(n-1):             constant
            if (the_list[i] > the_list[i+1]):   constant
                swap(the_list, i, i+1)          constant
```

n-1 times

n-1 times

So what is the Big-O complexity of `bubble_sort`?

Before we said $T(n) = 13n^2-22n+12$, now we say? $O(n^2)$

Do we need to first compute T(n) and then O(n)?

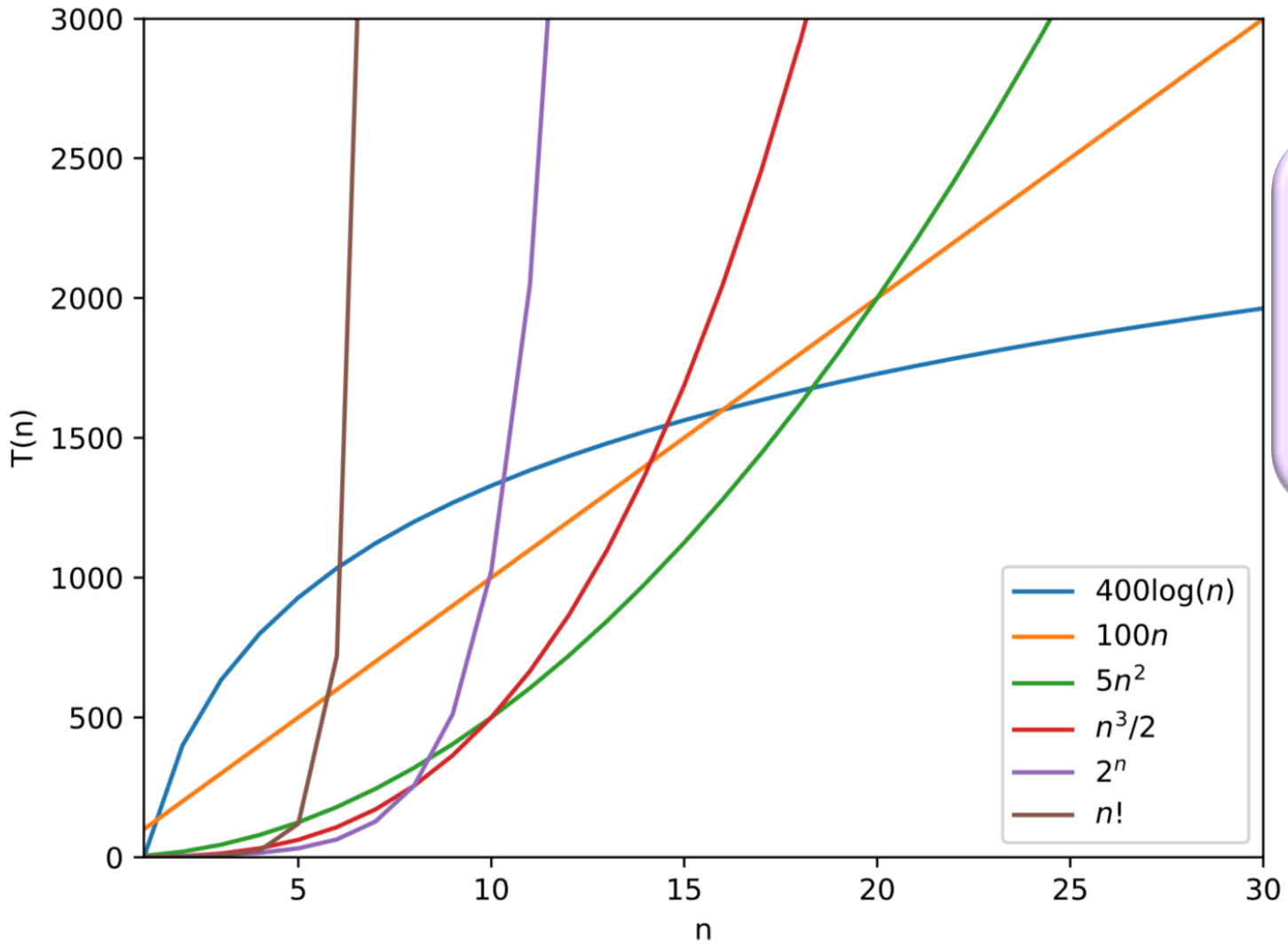
```
def swap(the_list,i,j):
    tmp = the_list[i]                    constant
    the_list[i]=the_list[j]              constant
    the_list[j] = tmp                    constant
```

So what is the Big-O complexity of `swap`?

Constant run time, so O(1)

No! We will see a bit later in more detail how to compute O(n) directly

Growth of computational complexity T(n)/O(n) functions



Multiplied by constants to make them easier to see. So they are technically T(n), rather than Big-O functions. But they are just multiplied by constants, so they look pretty similar.

Common Big O classes

Constant	O(1)	Running time does not depend on N	Find the first word in a dictionary	N doubles, T remains constant
Logarithmic	O(log N)	Problem is broken up into smaller problems and solved independently. Each step cuts the size by a constant factor.	Finding a word in a dictionary by looking in the mid-page, and then looking left or right until found	If N doubles, running time T gets slightly slower (not much!)
Linear	O(N)	Each element requires a certain (fixed) amount of processing	Multiplying every element in a list by a certain amount.	If N doubles, running time T doubles (2*T)
Superlinear	O(N log N)	Problem is broken up in sub-problems. Each step cuts the size by a constant factor and the final solution is obtained by combining the solutions.	Sorting a list by dividing in half, recursively sorting each half, and then merging (called merge sort)	If N doubles, running time T gets slightly bigger than double (2*T and a bit)
Quadratic	$O(N^2)$	Processes pairs of data items. Often occurs when you have double nested loop	Comparing every element of one list to every element of another	If N doubles, running time T increases four times (4*T)
Exponential	$O(2^N)$	Think about a tree where each node has two children.	Contagion rate when every person transmits to 2 people	If N doubles, running time T squares (T*T)
Factorial	O(N!)	A tree where each node at level n has n+1 children.	Find all permutations of N items	

Since 1 does not depend on N

If log N = T, then log 2*N= log 2 + log N = log2 + T

If N = T, then 2*N=2*T

If N*logN = T, then (2*N)*(log 2*N) =2*N*(logN+log2)=2*N*logN + 2N*log2 = 2*T +2Nlog2

If $N^2 = T$, then $(2N)^2=4N^2 = 4*T$

If $2^N = T$, then $2^{2N}= 2^{(N+N)} = 2^N * 2^N = T*T$

A reasonably modern CPU (2014). For 2020, 10^{-12}

Growth rates (every "step" takes a nanosecond 10^{-9}s)

N	log(N)	N	Nlog(N)	N^2	2^N	N!
10	0.003 µs	0.01 µs	0.033 µs	0.1 µs	1 µs	3.63 ms
20	0.004 µs	0.02 µs	0.086 µs	0.4 µs	1 ms	77.1 years
30	0.005 µs	0.03 µs	0.147 µs	0.9 µs	1 sec	8.4×10^{15} years
40	0.005 µs	0.04 µs	0.213 µs	1.6 µs	18.3 min	
50	0.006 µs	0.05 µs	0.282 µs	2.5 µs	13 days	
100	0.007 µs	0.1 µs	0.644 µs	10 µs	4×10^{13} years	
1,000	0.010 µs	1 µs	9.966 µs	1 ms		
10,000	0.013 µs	10 µs	130 µs	100 ms		
100,000	0.017 µs	100 µs	1.67 ms	10 sec		
1,000,000	0.020 µs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 µs	10 ms	0.23 sec	1.16 days		
100,000,000	0.027 µs	0.1 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 µs	1 sec	29.90 sec	31.7 years		

That is a lot of years… ☺

A million

A billion

Microseconds, 10^{-6}

Best/worst/average case complexity

- **Running time can depend on things OTHER than size: properties of DATA**
  - Like the list being sorted, or the word we look for being right at the beginning
- **Worst case: gives a guarantee, that is, it is correct for all inputs of size n**
  - Most often-quoted
- **Best case: correct for at least one input of size n**
  - Less useful. “Lucky” inputs (word being in the first place we look at) may be rare
- **Average: describes “usual” behaviour, not extremes …**
  - Often tricky to work out, so not discussed in our unit
- **If run time depends only on input size: best = worst**
- **Together, worst & best give an idea of the range of possibilities**
- **In unspecified, “time complexity” means worst case**

MONASH University

Time complexity of Bubble Sort

We left Bubble Sort here…

```
def bubble_sort(the_list):
    n = len(the_list)                          constant
    for _ in range(n-1):                       constant
        for i in range(n-1):                   constant
            if (the_list[i] > the_list[i+1]):  constant
                swap(the_list, i, i+1)         constant
```

n-1 times

n-1 times

So what is the Big-O complexity of `bubble_sort`?

Before we said $T(n) = 13n^2-22n+12$, now we say? $O(n^2)$

Do we need to first compute T(n) and then O(n)?

No! We will see a bit later in more detail how to compute O(n) directly

```
def swap(the_list,i,j):
    tmp = the_list[i]                  constant
    the_list[i]=the_list[j]            constant
    the_list[j] = tmp                  constant
```

So what is the Big-O complexity of `swap`?

Constant run time, so O(1)

Back to Bubble Sort: details

```
def bubble_sort(the_list):
    n = len(the_list)
    for _ in range(n-1):
        for i in range(n-1):
            if (the_list[i]>the_list[i+1]):
                swap(the_list, i, i+1)
```

- **The inner loop**
  - Runs (n-1)*(n-1) times
  - The comparison is always performed
    - For now we will assume constant time comparison
    - This is often NOT true (e.g., if the elements are strings)
  - The swap might be performed
    - Always (list in reverse order)
    - Never (list already sorted)
    - Sometimes (common case)
  - As both are constant, whether they are performed or not only affects the constant:
    - Smallest if already sorted
    - Biggest if reversed

Bubble Sort: Time complexity

- **Approximating inner loop operations by a constant (1), we get:**

$$(n-1)*(n-1)= (n^2-2n+1) \rightarrow O(n^2)$$

- **That is the worst time complexity:**
  - Both loops run for the maximum number of iterations
- **What is the best time complexity?**
  - Any properties of the elements that reduce big O?
  - In this case: that stop any of the two loops early? (e.g., does it stop if sorted?)
  - IMPORTANT: Being empty is NOT a property of the elements!
    - We are considering the scalability of the algorithm
    - So we must always assume a big n
- **No such property for the algorithm. This tells you what?**
  - best = worst

Properties of sorting algorithms

Algorithm	Best case	Worst case
Bubble Sort	$O(n^2)$	$O(n^2)$

Improved Bubble Sort : Time Complexity

```
def bubble_sort(the_list):
    n = len(the_list)                    constant
    for mark in range(n-1,0,-1):         constant
        for i in range(mark):            constant
            if (the_list[i] > the_list[i+1]):    constant
                swap(the_list, i, i+1)           constant
```

n-1 times

mark times

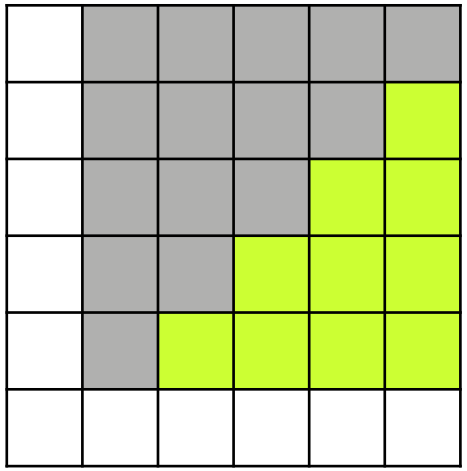
Main difference

First time mark is n-1, then n-2, n-3, …, until 1

Intuition: nested loops, both dependent on n, every operation on inner loop performed a fixed number of times: the worst case is going to be $O(n^2)$

Bubble Sort II: Time complexity

When sorting a list of n=6 elements, mark iterates 5 times, and for each mark the inner loop i iterates 5, 4, 3, 2, and 1 times (each represented by a grey cell). This avoids the work performed by the naïve algorithm (which includes green cells). Similarly for n.



- **The inner loop runs for:**

$$(n-1) + (n-2) + (n-3) + \ldots + 1 = n*(n-1)/2 = (n^2 - n)/2$$

  - The rest is as before
- **Ignoring small factors and constants, we have:**

$$(n^2-n)/2 \rightarrow O(n^2)$$

- **Any properties of the list elements that affect big O?**
  - In this case: that stop any of the two loops early?
- **No! This tells you what?**
  - best = worst
- **Same complexity than the previous version! So why is it better?**
  - Not better scalability BUT
  - Better efficiency (almost half the number of inner iterations, as green cells avoided)

Bubble Sort II

```
def bubble_sort(the_list):
    n = len(the_list)
    for mark in range(n-1,0,-1):
        swapped = False
        for i in range(mark):
            if (the_list[i] > the_list[i+1]):
                swap(the_list, i, i+1)
                swapped = True
        if not swapped:
            break
```

? times

mark times

Does this change BigO complexity?

Yes! We can now leave the outer loop if the list is sorted. So, best case is now O(n) when the list is sorted

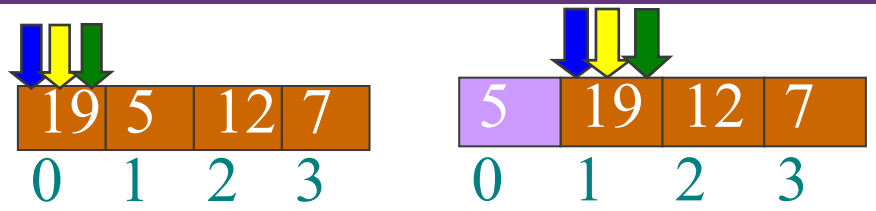
Properties of sorting algorithms

Algorithm	Best case	Worst case
Bubble Sort	$O(n^2)$	$O(n^2)$
Bubble Sort II	$O(n)$	$O(n^2)$

MONASH University

Complexity of Selection Sort and Insertion Sort

Selection Sort: Time complexity




```
def selection_sort(the_list):
    n = len(the_list)                                   constant
    for mark in range(n-1):                             constant
        min_index = find_index_min(the_list,mark)       n-mark-1 times
        swap(the_list, mark, index_min)                 constant
```

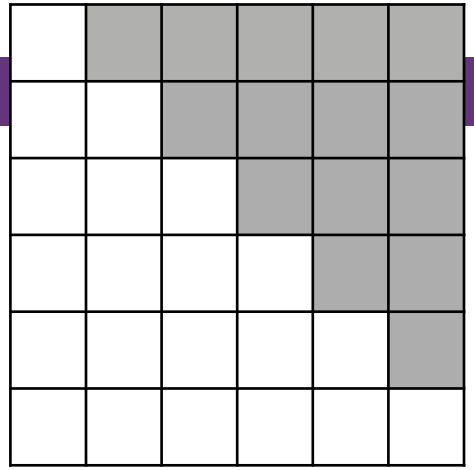
n-1 times

```
def find_index_min(the_list,mark):
    pos_min = mark                                      constant
    n = len(the_list)                                   constant
    for i in range(mark+1,n):                           constant
        if(the_list[i]<the_list[index_min]):            constant
            pos_min = i                                 constant
    return pos_min                                      constant
```

n-mark-1 times each

First time function is called, mark is 0, then 1, 2, …, until n-2

First time the loop runs n-1 times, then n-2, n-3, …, until 1 time



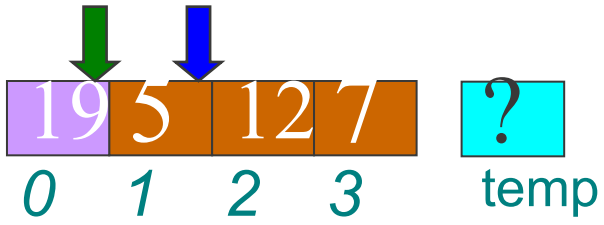
Selection Sort: Time complexity

- **The inner loop**
  - Always runs for:
    $$(n-1) + (n-2) + (n-3) + \ldots + 1 = n*(n-1)/2 = (n^2 - n)/2$$
  - The comparison is always performed
  - The swap is always performed once per iteration
  - This only affects the constants, so $O(n^2)$
- **Any properties of the list elements that affect big O?**
  - In this case: that stop any of the two loops early?
- **No! This tells you what?**
  - best = worst
- **Same time complexity as naïve & improved Bubble Sort (not BubbleSort II) BUT usually faster:**
  - Fewer swaps in average translate in a smaller k

Properties of sorting algorithms

Algorithm	Best case	Worst case
Bubble Sort	$O(n^2)$	$O(n^2)$
Bubble Sort II	$O(n)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$

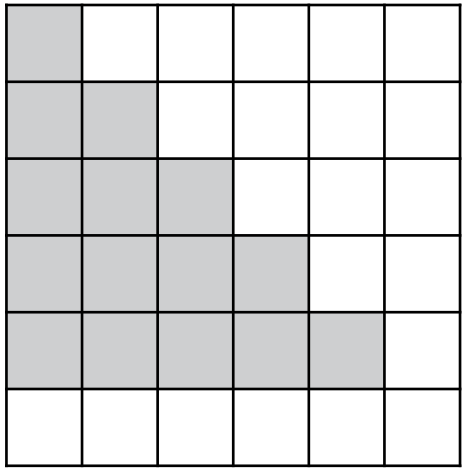
Insertion Sort: Time Complexity




```
def insertion_sort(the_list):
    n = len(the_list)                    constant
    for mark in range(1,n):              constant
        temp = the_list[mark]            constant
        i = mark - 1                     constant
        while i>=0 and the_list[i] > temp:   constant
            the_list[i+1] = the_list[i]  constant
            i-= 1                        constant
        the_list[i+1] = temp             constant
```

n-1 times

??



Inner loop might not run (if `the_list[i] <= temp`)

At most it runs mark times (grey cells in the matrix)

Insertion Sort: Time complexity

- **Can we stop any of the two loops early?**
  - Yes, the second one, when the element is already bigger
- **This already tells you what?**
  - best ≠ worst
- **Worst case?**
  - Every element needs to be shuffled to the left when inserting: the list is sorted in reverse order
  - This means $O(n^2)$ – two nested loops both dependent on n, with the inner one performing a fixed amount of steps
- **Best case?**
  - No element needs to be shuffled when inserting: the list is already sorted
  - This means O(n) – one loop dependent on n and performing a fixed amount of steps

Properties of sorting algorithms

Algorithm	Best case	Worst case
Bubble Sort	$O(n^2)$	$O(n^2)$
Bubble Sort II	$O(n)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$

Insertion Sort versus the other sorts

- **Usually faster than bubble and selection sort, especially for "almost sorted" lists**
- **Can you figure out why?**
  - What do you avoid in that case?

- **It is however slower than selection sort if our write access to memory is slow**
- **Can you figure out why?**
  - What do you do in one and not in the other?

Think about MIPS memory access

Points to keep in mind

- Big-O gives an upper bound. May be much larger than the actual one
- The input that gives the worst case may be very unlikely
- Big-O ignores constants. In practice they may be very large
- If a program is used only a few times, then the actual running time may not be a big factor in the overall costs
- If a program is only used on small inputs, the growth rate of the running time may be less important than other factors
- A complex but efficient algorithm can be less desirable than a simpler one
- Efficiency is not everything: In numerical algorithms, other properties (like stability and incrementally) can be as important as efficiency
- The average case is always between the best and the worst cases

Summary of this lesson

- **You now know about running time, computational time complexity, and asymptotic analysis (Big-O notation) of time complexity**
- **You are able to compute the Big-O time complexity of simple functions**
  - Both best and worst case
- **In particular, you are now able to compute the Big O time complexity of:**
  - Bubble Sort
  - Selection Sort
  - Insertion Sort