COMP1730/COMP6730
Programming for Scientists

(Algorithm and problem)
Computational complexity

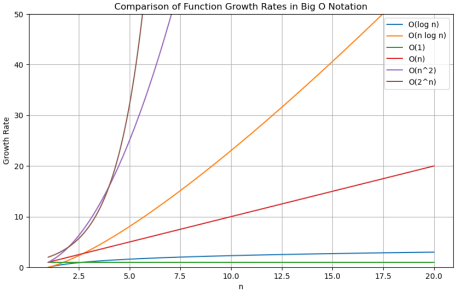
Algorithm complexity

- The time (or memory) consumed by a particular algorithm that solves a computational problem:
  - Counting “elementary operations” (not μs).
  - Expressed as a function of the size of its input arguments.
  - In the worst case.

- Complexity describes scaling behaviour: How much does runtime grow if the size of the arguments grow by a certain factor?
  - Understanding algorithm complexity is especially important when dealing with large problems.

Big-O notation

- $O(f(n))$ means roughly "a function that grows at (or below) the rate of $f(n)$ for large enough n".
- Note that we do not care about constants, only the overall growth curve type.
- For example,
  - $n^2 + 2n + 1$ is $O(n^2)$ (quadratic time)
  - $100n$ is $O(n)$ (linear time)
  - 10^{12} is $O(1)$ (constant time).



Example

- Find the greatest element $\leq x$ in an *unsorted* sequence of *n* elements. (For simplicity, assume some element $\leq x$ is in the sequence.)
- Two approaches:

a) Search through the sequence; or

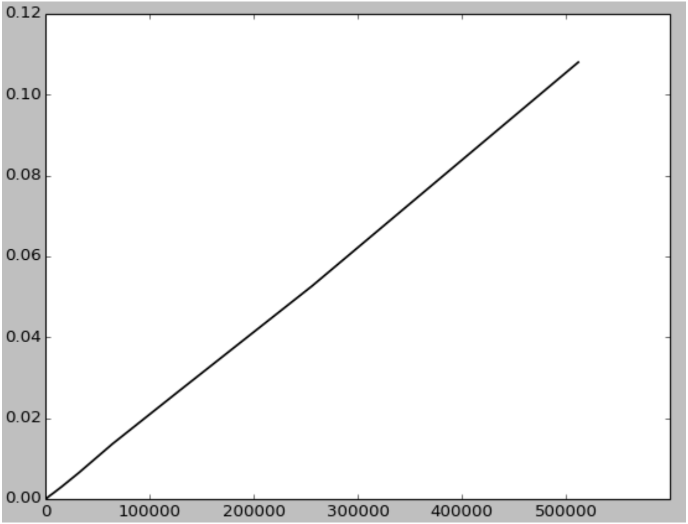
b) First sort the sequence, then find the greatest element $\leq x$ in a *sorted* sequence.

Searching an unsorted sequence

```
def unsorted_find(x, ulist):
    """
    search unsorted list (ulist) for largest element <= x
    """
    best = min(ulist)
    for elem in ulist:
        if elem == x:
            return elem  # elem found
        elif elem < x:
            if elem > best:
                best = elem  # update if larger
    return best
```

Analysis

- Elementary operation: comparison.
  - Can be arbitrarily complex.
- If we're lucky, `ulist[0] == x`.
- Worst case?
  - `ulist = [0, 1, 2, ..., x - 1]`
  - Compare each element with `x` and current value of `best`
- What about `min(ulist)`?
- $f(n) = 2n$, so $O(n)$



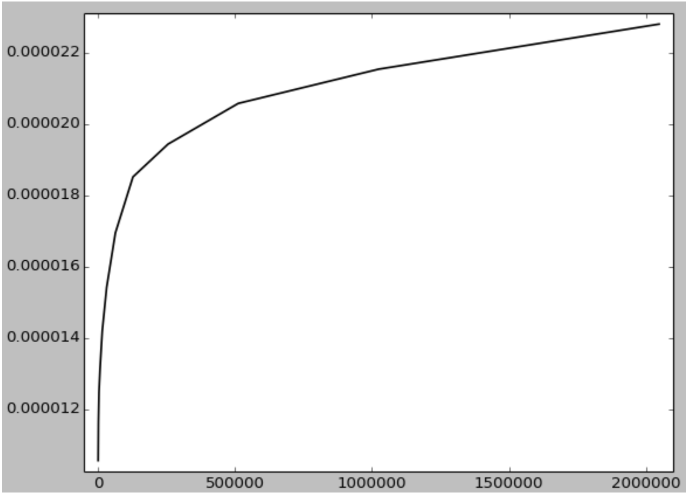
Measured runtime

Searching a sorted sequence

```
def sorted_find(x, slist):
    """
    search the sorted list for the largest element <= x.
    """
    if slist[-1] <= x:
        return slist[-1]
    lower = 0
    upper = len(slist) - 1
    # search by interval halving
    while (upper - lower) > 1:
        middle = (lower + upper) // 2
        if slist[middle] <= x:
            lower = middle
        else:
            upper = middle
    return slist[lower]
```

Analysis

- Loop invariant: `slist[lower] <= x` and `x < slist[upper]`.
- How many iterations of the loop?
  - Initially, `upper - lower` $= n - 1$.
  - The difference is halved in every iteration.
  - Can halve it at most $\log_2(n)$ times before it becomes 1.
- $f(n) = \log_2(n) + 1$, so $O(\log(n))$.



Measured runtime

Nested loops- exam example

- The following function takes as input an integer ‘x‘. Give its computational complexity in big-O notation in terms of ‘x‘.

```python
def func_a(x):
    total = 0
    for i in range(x*2):
        for j in range(x):
            for k in range(x):
                total = total + i * j * k
    return total
```

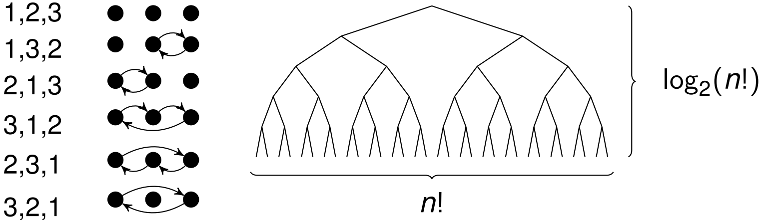
- Answer: O(x^3)
- (Note that the constant in the outer loop is ignored).

Problem complexity

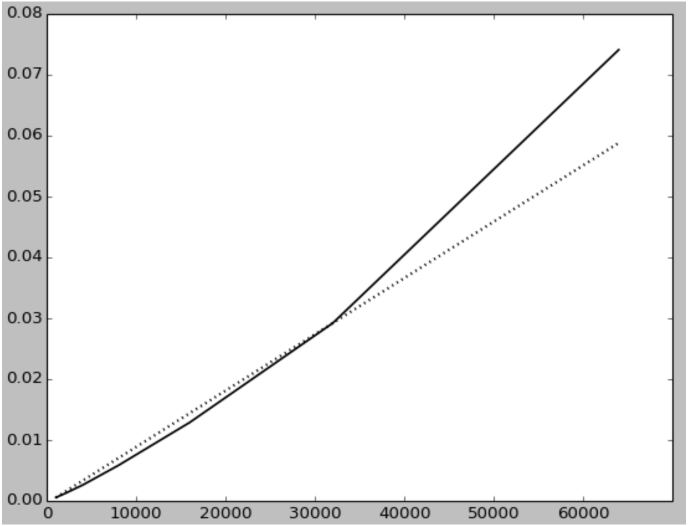
- The complexity of a problem is the time (memory) that **any** algorithm that solves the problem **must** use, in the worst case, as a function of the size of the arguments.
- In other words, the complexity of a problem is the **infimum** of the complexities among all algorithms that solve the problem
- For example, mathematicians have been able to prove that **any sorting algorithm** that uses only pair-wise comparisons **needs** $O(n\log(n))$ **comparisons in the worst case**
- Proving these kind of results is out of the scope of this course and requires advanced arguments in mathematical theory of computation (so will not be tested in exam)

How fast can you sort?

- Any sorting algorithm that uses only pair-wise comparisons needs $n\log(n)$ comparisons in the worst case.



- $\log_2(n!) \leq n\log(n)$ for large enough n.
- So $\log_2(n!)$ is $O(n\log(n))$.



Measured runtime (`list.sort`)

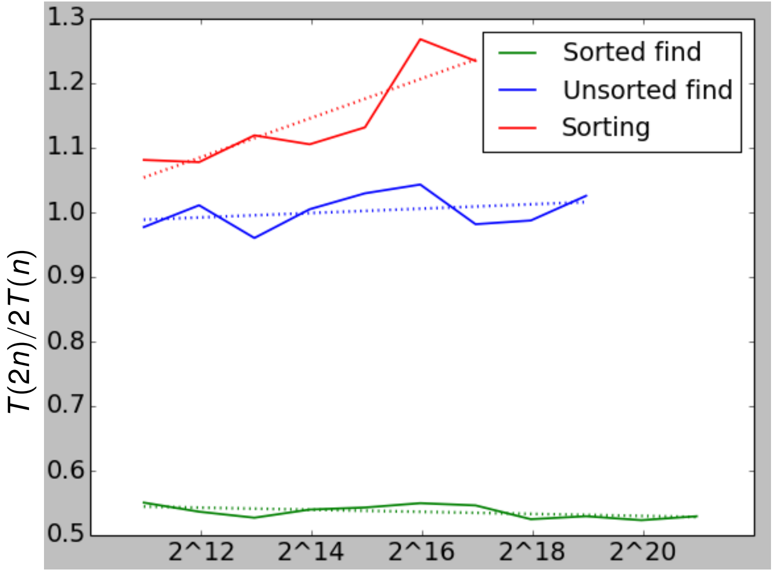
Points of comparison

- Algorithm (a): $O(n)$
- Algorithm (b): $n\log(n) + \log(n) = O(n\log(n))$

	$n = 64$k	$n = 128$k	$n = 512k$
Unsorted find	0.013 s	0.026 s	0.108 s
Sorted find	0.000017s	0.000018s	0.00002 s
Sorting	0.07 s	0.18 s	

Rate of growth

- Algorithm uses $T(n)$ time on input of size n.
- If we double the size of the input, by what factor does the runtime increase?



T(2n)/2T(n)
1.3
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.5
2^12
2^14
2^16
2^18
2^20
Sorted find
Unsorted find
Sorting

Caution

- Remember: Scaling behaviour becomes important when problems become *large*, or when they need to be solved *many times*.
- e.g. an algorithm may work for a small test sample in a scientific pipeline, but by infeasible for a full data analysis.

Takehome message

- Time (or memory) complexity is expressed in big-O notation as a function of the input size.
- The computational (and memory) complexity is a major determinant in choosing a given algorithm or data structure for an application:
- e.g. a dictionary is (average) constant time lookup compared to linear time for an unsorted list and so may be preferred for applications requiring many lookups.
- See, for example, time complexity of operations on Python built-in types available at the Python wiki (wiki.python.org)