Australian National University

COMP1730/COMP6730
Programming for Scientists

Algorithm and problem complexity

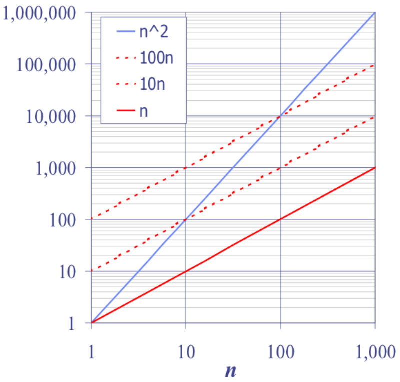
Algorithm complexity

* The time (memory) consumed by an algorithm:
  - Counting “elementary operations” (not μs).
  - Expressed as a function of the size of its 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 important when (but only when) dealing with large problems.

Big-O notation

* $O(f(n))$ means roughly "a function that grows (in the worst-case) at the rate of $f(n)$, for large enough n".
* For example,
  - $n^2 + 2n$ is $O(n^2)$
  - $100n$ is $O(n)$
  - 10^{12} is $O(1)$.



(Image by Lexing Xie)

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:

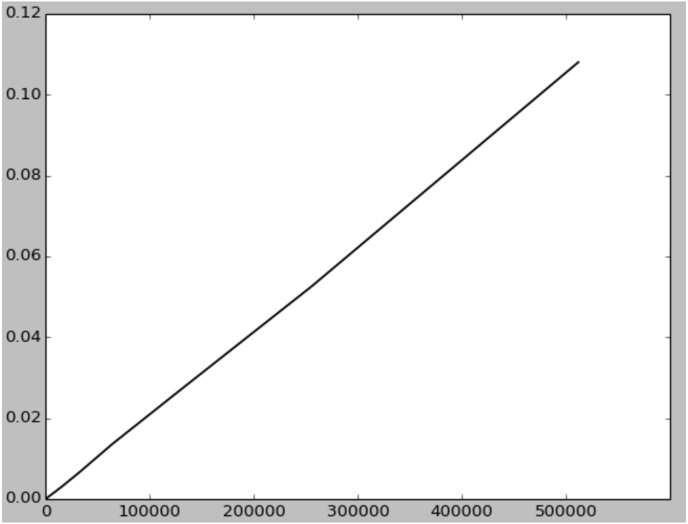
a) Search through the sequence; or
b) First sort the sequence, then find the greatest element $\leq x$ in a *sorted* sequence.

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 $O(n)$



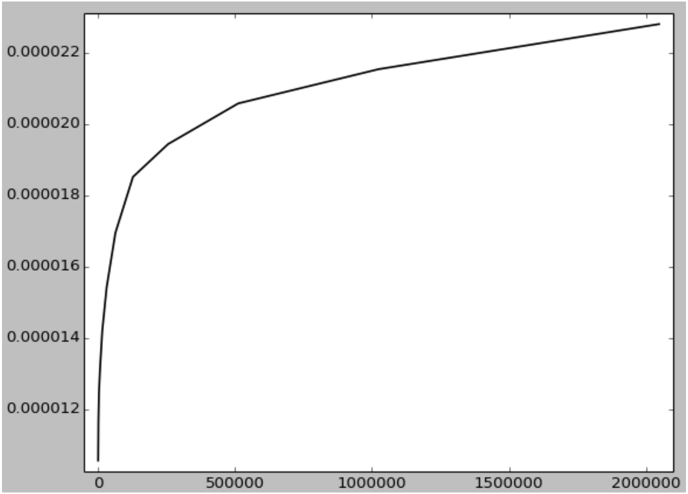
Measured runtime

Searching a sorted sequence

```python
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 (binary search)
    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 $O(\log(n))$.



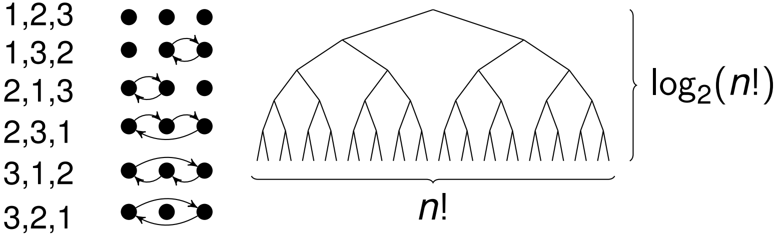
Measured runtime

Problem complexity

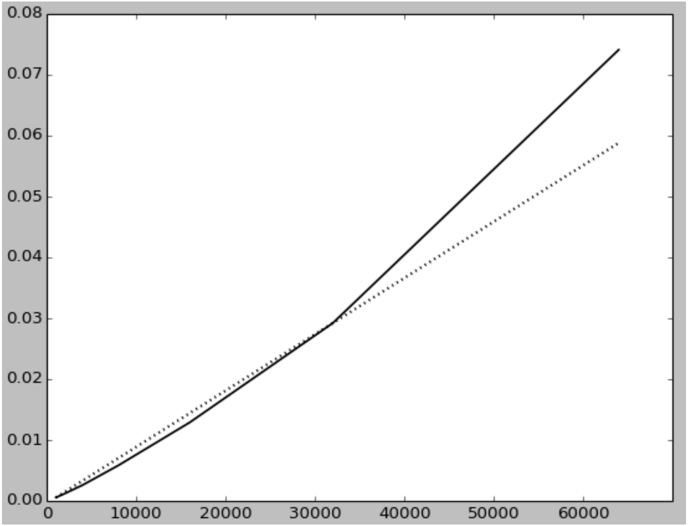
- The complexity of a problem is the time (or memory) that *any* algorithm *must* use, in the worst case, to solve the problem, as a function of the size of the arguments.
- The hierarchy theorem: For any computable function $f(n)$ there is a problem that requires time greater than $f(n)$. (Analogous result for memory.)

How fast can you sort?

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



- $\log_2(n!) \geq n\log(n)$ for large enough n.



Measured runtime (`list.sort`)

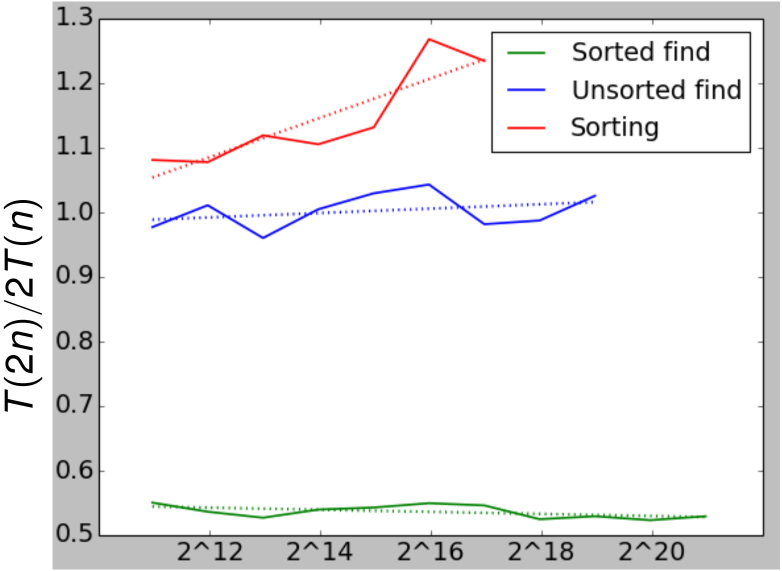
Points of comparison

* Algorithm (a): $O(n)$
* Algorithm (b): $n\log(n) + \log(n) = O(n\log(n))$
* If we know that the input is already sorted in our application then is $O(\log(n))$

	$n = 64$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 a *many times*.
* e.g. an algorithm may work for a small test sample in a scientific pipeline, but by infeasible for a full genomic analysis.

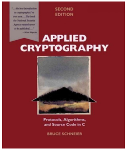
NP-Completeness

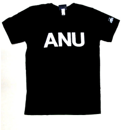
Example

- The subset sum problem: Given n integers $w_1, \ldots, w_n$, is there a subset of them that sums to exactly C?

  (Also known as the "(exact) knapsack problem":







$\Rightarrow$


$w_0 = 5 \quad w_1 = 2 \quad w_2 = 9 \quad w_3 = 1 \qquad C = 16.$)

```python
def subset_sum(w, C):
    """
    Returns True if there is a subset of a list w summing to C.
    Otherwise, returns False.
    """
    if len(w) == 0:
        return C == 0
    # including w[0]
    if w[0] <= C:
        if subset_sum(w[1:], C - w[0]):
            return True
    # excluding w[0]
    if subset_sum(w[1:], C):
        return True
    return False
```

Analysis

- Count recursive function calls (no loops, so every call does a constant max amount of work).
- Assume argument size (n) is number of weights.
- Worst case?
  - If the answer is `False` and C is less than but close to $\sum_i w_i$, almost every call makes two recursive calls.
- $f(n+1) = 2f(n)$, $f(0) = 1$ means that $f(n) = 2^n$.

Decision problems: Finding vs. checking an answer

* Sorting a list vs.	$O(n\log(n))$
checking if it's already sorted	$O(n)$
* Finding a subset of $w_1, \ldots, w_n$	$O(2^n)$
that sums to C vs.	
checking if a sum is equal to C	$O(n)$

NP-complete problems

* A problem is **in NP** iff there is an answer-checking algorithm that runs in polynomial time ($O(n^c)$, c constant).
* Polynomial time is considered “feasible”.
* NP stands for **N**on-deterministic **P**olynomial time. “Non-deterministic” describes a brute force algorithm that would require infinite parallelism to find a solution.
* A problem is **NP-complete** if it’s in NP and *at least as hard as every other problem in NP*.

NP-complete problems

* The Boolean satisfiability problem (SAT) is to determine if a propositional logic formula can be made true by an appropriate assignment of truth values to its variables.
* It is fast to verify whether a given logical assignment makes the formula true, no essentially faster method to find a satisfying assignment is known than to try all assignments in succession.
* Cook and Levin proved that every such decision problem with a polynomial time solution can be converted to SAT and so solved as fast as SAT.

NP-complete problems

* This class of problem is called NP-complete .
* It is a major unsolved problem in CS: we think there is no polynomial time algorithm for solving NP-complete problems, but *we don't know*.
* Knowing that your problem is NP-complete suggests that you should look for useful heuristic solutions to your problem.
* (Note that this refers to worst-case time. In fact, fast heuristic algorithms for SAT have been developed that are practically useful for most inputs.)

There are many NP-complete problems...

* Most populous intractable problem class.
  - Solving a system of *integer* linear equations.
  - The Knapsack problem.
* `https://en.wikipedia.org/wiki/List_of_NP-complete_problems` lists many NP-complete problems.

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 (amortised) constant time lookup compared to linear time for an unsorted list and so may be preferred for some applications.