Australian National University

COMP1730/COMP6730

Programming for Scientists

(Algorithm and problem) Computational complexity

Announcements

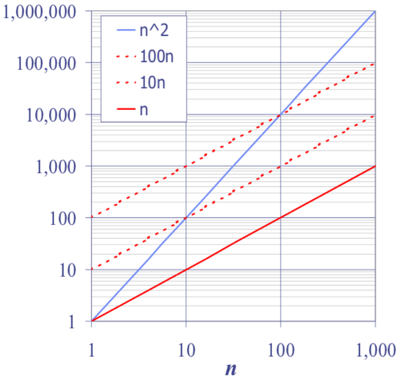
* The last date for students to drop courses without failure is **this Friday** – 6/10/2023
* Final exam has been scheduled – 14/11/2023
* Two separate exams for COMP1730 (9am-12pm) and COMP6730 (2pm-5pm)
* Centrally invigilated exam, CSIT and HN computer labs

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 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

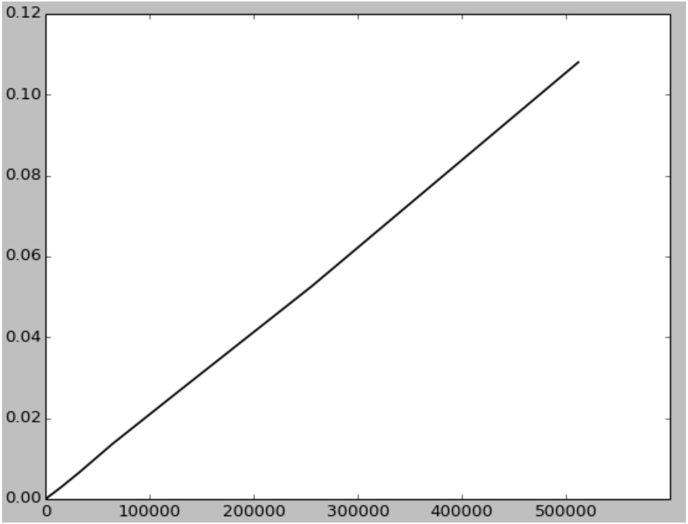
b) First sort the sequence, then find the greatest element $\leq x$ in a *sorted* sequence.

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 $O(n)$



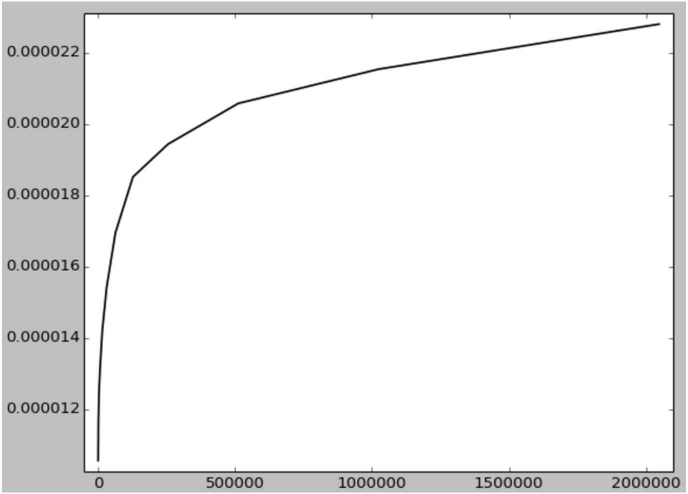
Measured runtime

Searching a sorted sequence

```
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

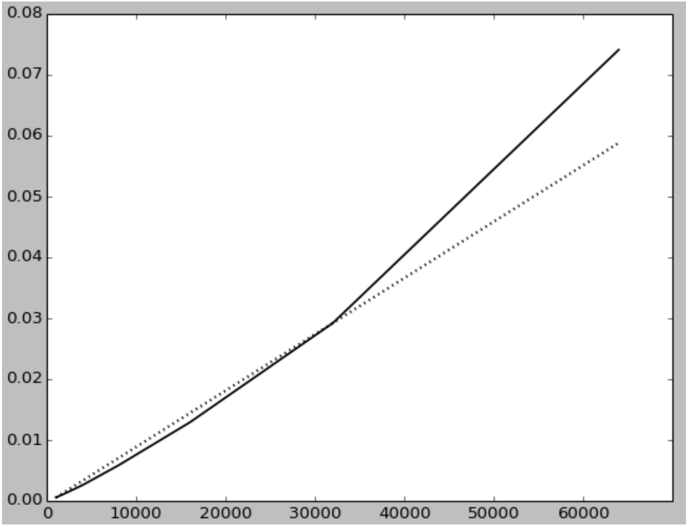
* Loop invariant: `slist[lower] <= x` and `x < slist[upper]`.
* How many iterations of the loop?
  - Initially, `upper - lower` $= n - 1$.
  - The difference is halved in every iteration.
  - Can halve it at most $\log_2(n)$ times before it becomes 1.
* $f(n) = \log_2(n) + 1$, so $O(\log(n))$.



Measured runtime

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



Measured runtime (`list.sort`)

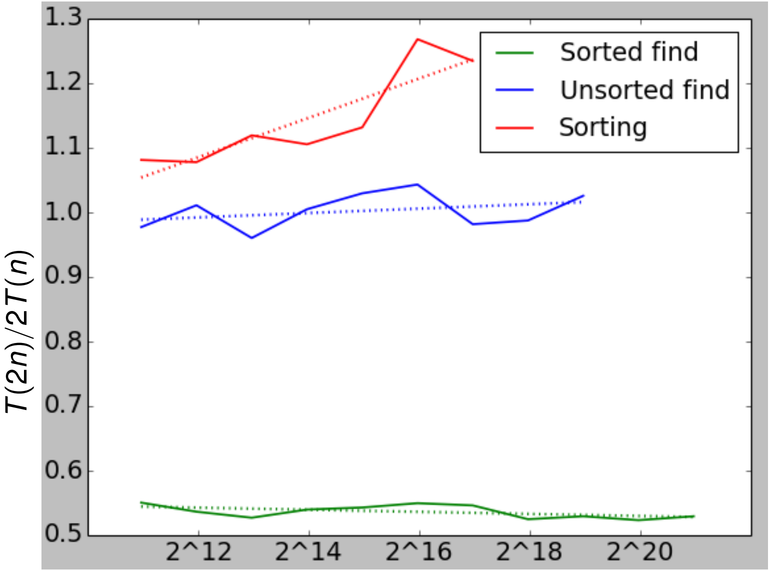
Points of comparison

* 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?


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

Caution

* "Premature optimisation is the root of all evil in programming."

  – C.A.R. Hoare

* Remember: Scaling behaviour becomes important when (and *only* when) problems become *large*, or when they need to be solved *many times*.

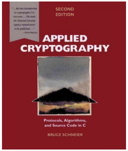
NP-Completeness

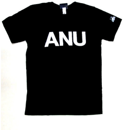
Example

* 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":







$\Rightarrow$


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

```python
def subset_sum(w, C):
    """
    Returns a tuple with two elements.

    The first element is True if there is a subset of
    a list w summing to C. Otherwise, it is False.

    The second element is the list of elements of
    w that sum to C
    """
    if len(w) == 0:
        return C == 0, []
    # including w[0]
    if w[0] <= C:
        can_do, subset = subset_sum(w[1:], C - w[0])
        if can_do:
            return True, [w[0]] + subset
    # excluding w[0]
    can_do, subset = subset_sum(w[1:], C)
    if can_do:
        return True, subset
    return False, None
```

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 $f(n) = 2^n$.

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).
* NP stands for **N**on-deterministic **P**olynomial time.
* A problem is **NP-complete** if it's in NP and *at least as hard as every other problem in NP*.
* We think there is no polynomial time algorithm for solving NP-complete problems, but *we don't know*.

There are many NP-complete problems...

* Most populous intractable problem class
  - Solving a system of *integer* linear equations
  - The Knapsack problem
* You can click here for a list of NP-complete problem examples

Takehome messages

- Time and memory complexity is expressed in big-O notation as a function of the input size
- See, for example, time complexity of operations on Python built-in types available at the Python wiki
- Computational complexity is a major determinant in choosing a given algorithm/data structure for an application
- Many real-world problems are computationally hard (NP)