

COMP1730/COMP6730

Programming for Scientists

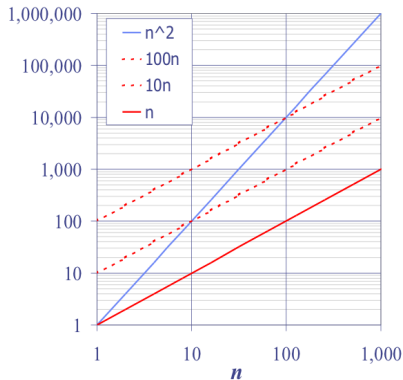
Algorithm and problem
complexity

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 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

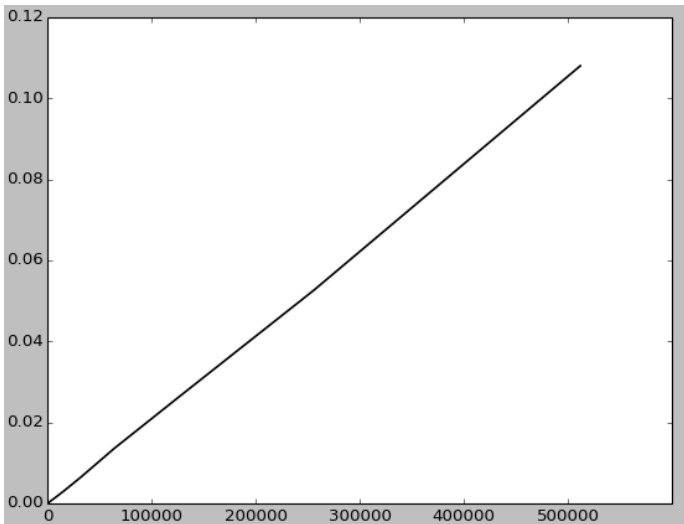
- ★ Find the greatest element $\leq x$ in an *unsorted* sequence of n elements. (For simplicity, assume some element $\leq x$ is in the sequence.)
- ★ Two approaches:
 - a) Search through the sequence; or
 - b) First sort the sequence, then find the greatest element $\leq x$ in a *sorted* sequence.

Searching an unsorted sequence

```
def unsorted_find(x, uelist):  
    best = min(ulist)  
    for elem in uelist:  
        if elem == x:  
            return elem  
        elif elem <= x:  
            if elem > best:  
                best = elem  
    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 $O(n)$



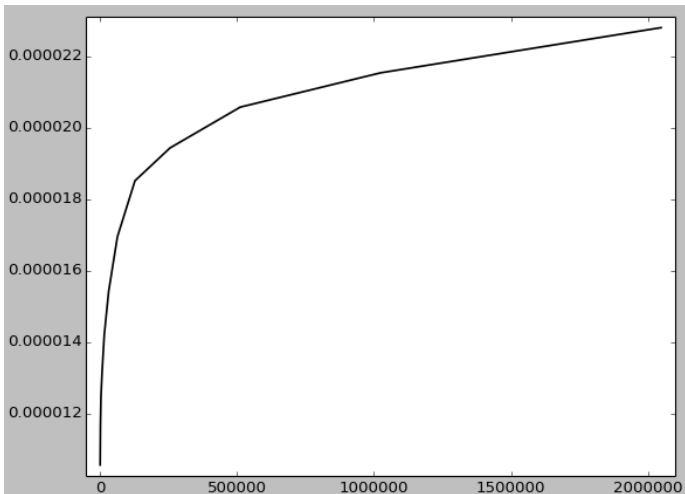
Measured runtime

Searching a sorted sequence

```
def sorted_find(x, slist):  
    if slist[-1] <= x:  
        return slist[-1]  
    lower = 0  
    upper = len(slist) - 1  
    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 $O(\log(n))$.



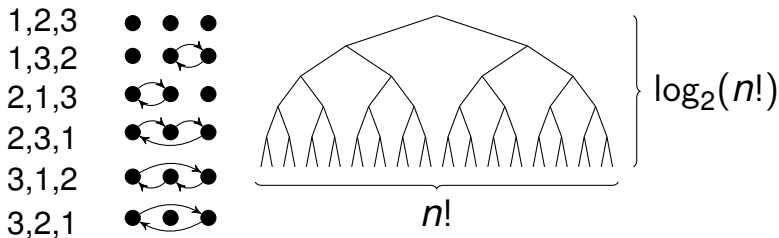
Measured runtime

Problem complexity

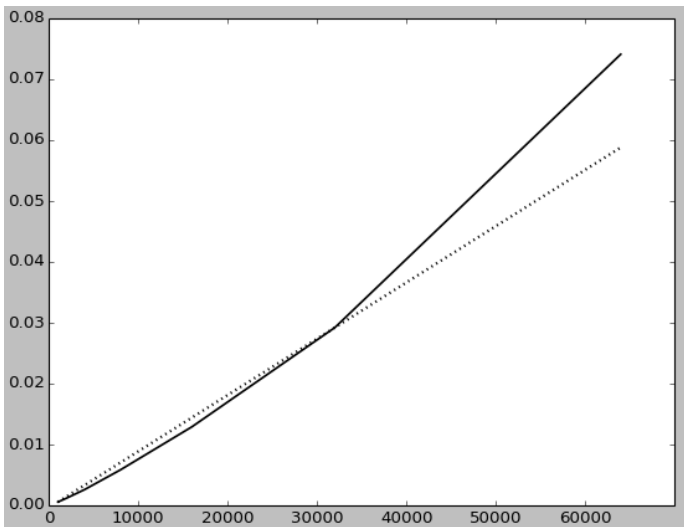
- * The complexity of a problem is the time (memory) that *any* algorithm *must* use, in the worst case, to solve the problem, as a function of the size of the arguments.
- * The hierarchy theorem: For any computable function $f(n)$ there is a problem that requires time greater than $f(n)$. (Analogous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

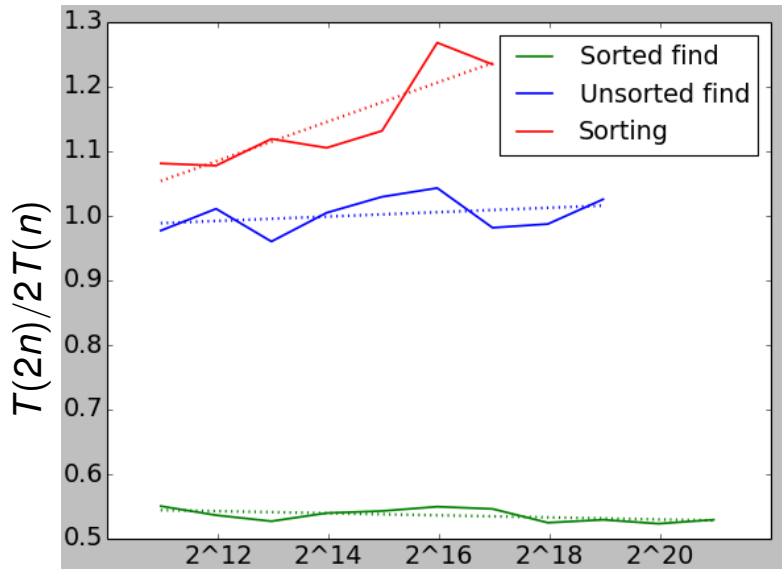
Points of comparison

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

	$n = 64k$	$n = 128k$	$n = 512k$
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?



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*.

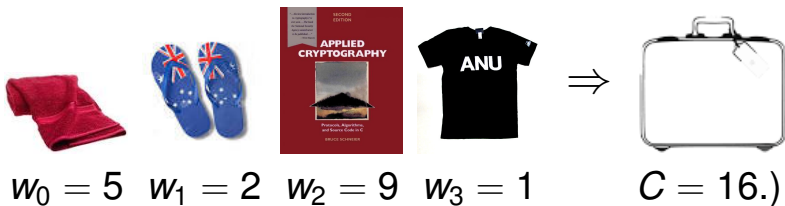


NP-Completeness

Example

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

(Also known as the “(exact) knapsack problem”:



```
def subset_sum(w, C):  
    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 $f(n) = 2^n$.

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, \dots, 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 **Non-deterministic Polynomial** time.
- * A problem is **NP-complete** if it's in NP and *at least as hard as every other problem in NP*.

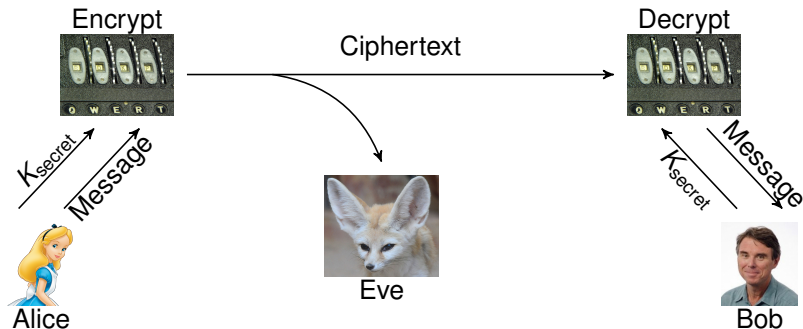
NP-complete problems...

- * Most populous intractable problem class.
 - Solving a system of *integer* linear equations.
 - The Knapsack problem.
- * <http://www.nada.kth.se/~viggo/wwwcompendium/wwwcompendium.html> lists over 700 NP-complete optimisation problems.
- * We think there is no polynomial time solution to NP-complete problems, but *we don't know*.
- * This is one of the *Millennium Problems* posed by Clay Mathematics Institute and there is a one million USD prize attached to a solution.



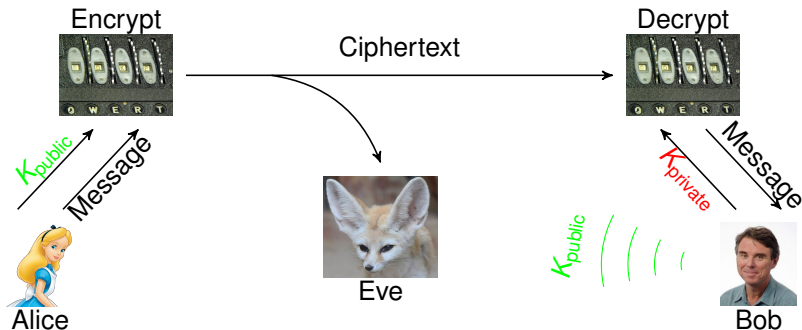
Why Complexity is (Sometimes) a Good Thing

Cryptographic Characters



- * Eve can intercept the ciphertext, but without knowing K_{secret} can't read the message.
- * Alice and Bob must agree on K_{secret} .

Public Key Cryptography



- * K_{public} can only be used to encrypt.
- * Decrypting with K_{private} is easy, but decrypting without knowing K_{private} is *(NP-)hard*.

Example: Proof of Identity

- * Alice is chatting with “Bob” on-line, but wants to be sure it’s really Bob.
 1. Alice picks a random number N and sends $C = \text{Encrypt}(K_{\text{public}}, N)$ to “Bob”.
 2. Bob *quickly* computes $N = \text{Decrypt}(K_{\text{private}}, C)$ and sends N back to Alice.

Repeat **1–2** many times to make sure “Bob” didn’t make a lucky guess.

Succeeding every time proves he knows K_{private} , which we assume only Bob does.