C02 Computational Complexity

Time and Space Complexity
Algorithm vs Problem Complexity
Big O Notation
Examples

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

Computational Complexity

Key computational resources:

- **Time**
- **Space**
- Energy, communications, I/O, samples...

Computational complexity is the study of how problem size affects resource consumption (how it *scales*). Distinguish:

- **Algorithm Complexity**: for a given algorithm / implementation
- **Problem Complexity**: for *any* algorithm that solves the problem
  - Inherit difficulty of the problem (Computational Complexity Theory)

Algorithm Complexity

- Identify ***n***, the number that characterizes the problem size.
  - Number of pixels on screen
  - Number of elements to be sorted
  - etc.
- Study the algorithm to determine how resource consumption changes as a function of ***n***.
- The *content* of the input, not just its size, can be important. Can study:
  - **Worst** case (the worst input of size ***n***)
  - **Best** case (the best input of size ***n***)
  - **Average** case (average of distribution of inputs of size ***n***)
  - **Amortized** analysis (amortized cost over a sequence of ***n*** typical operations)
    - Useful for an operation with state that occasionally has an expensive step

Big O Notation

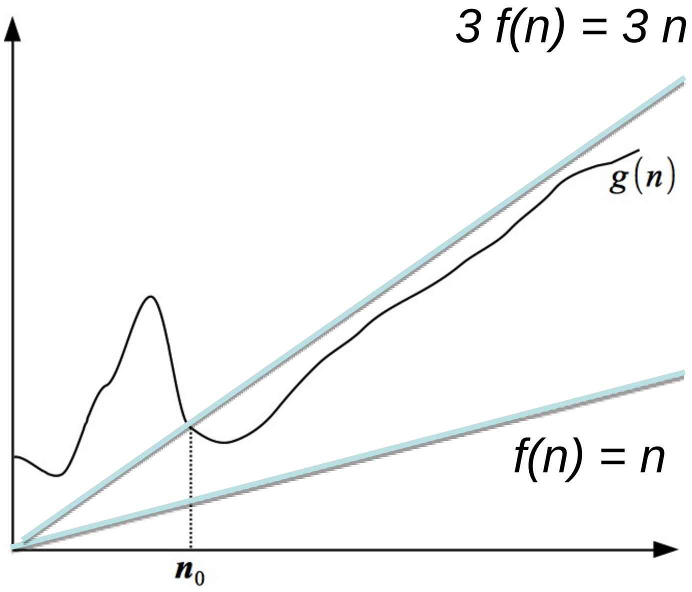
Suppose we have a problem of size n that takes $g(n)$ time to execute in the average case.

We say:

$$g(n) \in O(f(n))$$

iff there exists constants $c > 0$ and $n_0 > 0$ such that for all $n > n_0$:

$$g(n) \leq c \times f(n)$$



Time complexity

In analysis of algorithm time complexity, we are interested in the number of “**elementary operations/statements**” (not μs).

- Simple statements are constant time.
- Remember the factor c in $O(f(n))$.
- Beware: Library/subroutine calls can have arbitrary complexity.

Summing a List

Consider summing a list of size n…

```
public int sum(ArrayList<Integer> list) {
    int rtn = 0;
    for (var i: list) {
        rtn += i;
    }
    return rtn;
}
```

Linear time, $O(n)$

Minimum Difference

Note: $n - 1 + n - 2 + \ldots 2 + 1 = n(n-1)/2$

```
public int minDiff(ArrayList<Integer> values) {
    int min = Integer.MAX_VALUE;  1
    for (int i = 0; i < values.size(); i++) {   n
        for (int j = i + 1; j < values.size(); j++) {   n(n – 1)/2
            int diff = values.get(i) - values.get(j);   n(n – 1)/2
            if (Math.abs(diff) < min)   n(n – 1)/2
                min = Math.abs(diff);   n(n – 1)/2
        }
    }
}
```

$$
\begin{aligned}
S(n) &= 1 + n + 4\,(n(n-1)/2) \\
&= 1 + n + 2\,n^2 - 2n \\
&= 2n^2 - n + 1 \in O(n^2)
\end{aligned}
$$

More Examples

- Constant O(1)
  - Time to perform an addition
- Logarithmic O(log(n))
  - Time to find an element in a B-Tree (self-balancing tree)
- Linear O(n)
  - Time to find an element within a list
- O(n log(n))
  - Average time to sort using mergesort
- Quadratic $O(n^2)$
  - Time to compare n elements with each other pair-wise

Example: Greatest Up To

Find the greatest element ≤ x in an unsorted sequence of *n* elements (or else return `null`).

Two approaches:

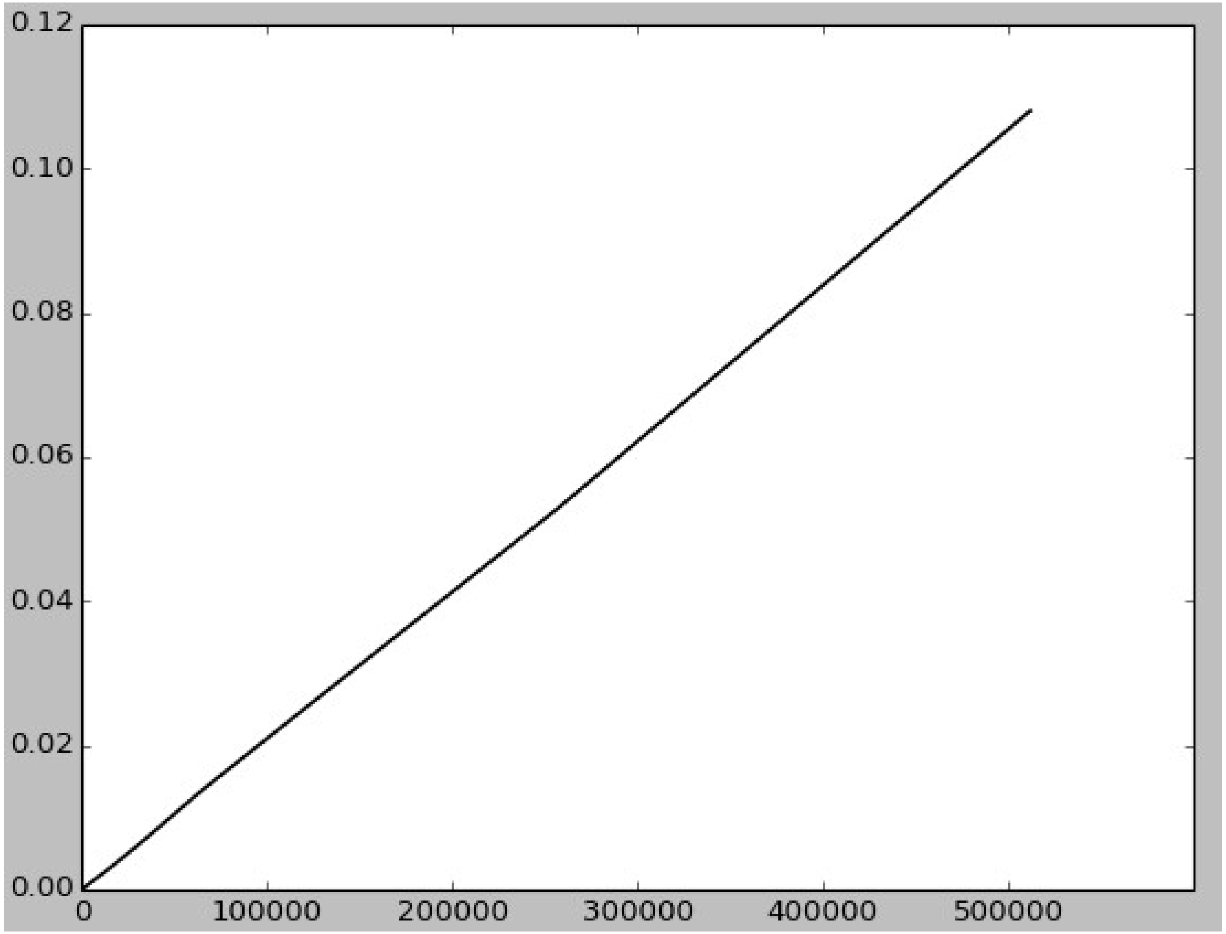
- a) search the unsorted sequence; or
- b) first sort the sequence, then search the sorted sequence.

Unsorted Greatest Up To

```
static Integer unsortedFind(int x, List<Integer> uList) {
    Integer best = null;
    for (var e : uList) {
        if (e == x)
            return e;
        if (e <= x && (best == null || e > best))
            best = e;
    }
    return best;
}
```

Analysis

- If we're lucky, `uList[0] == x`.
- Worst case?
    - `uList = {x – n, ..., x – 2, x – 1}`
    - $f(n) = 6n$, so $O(n)$

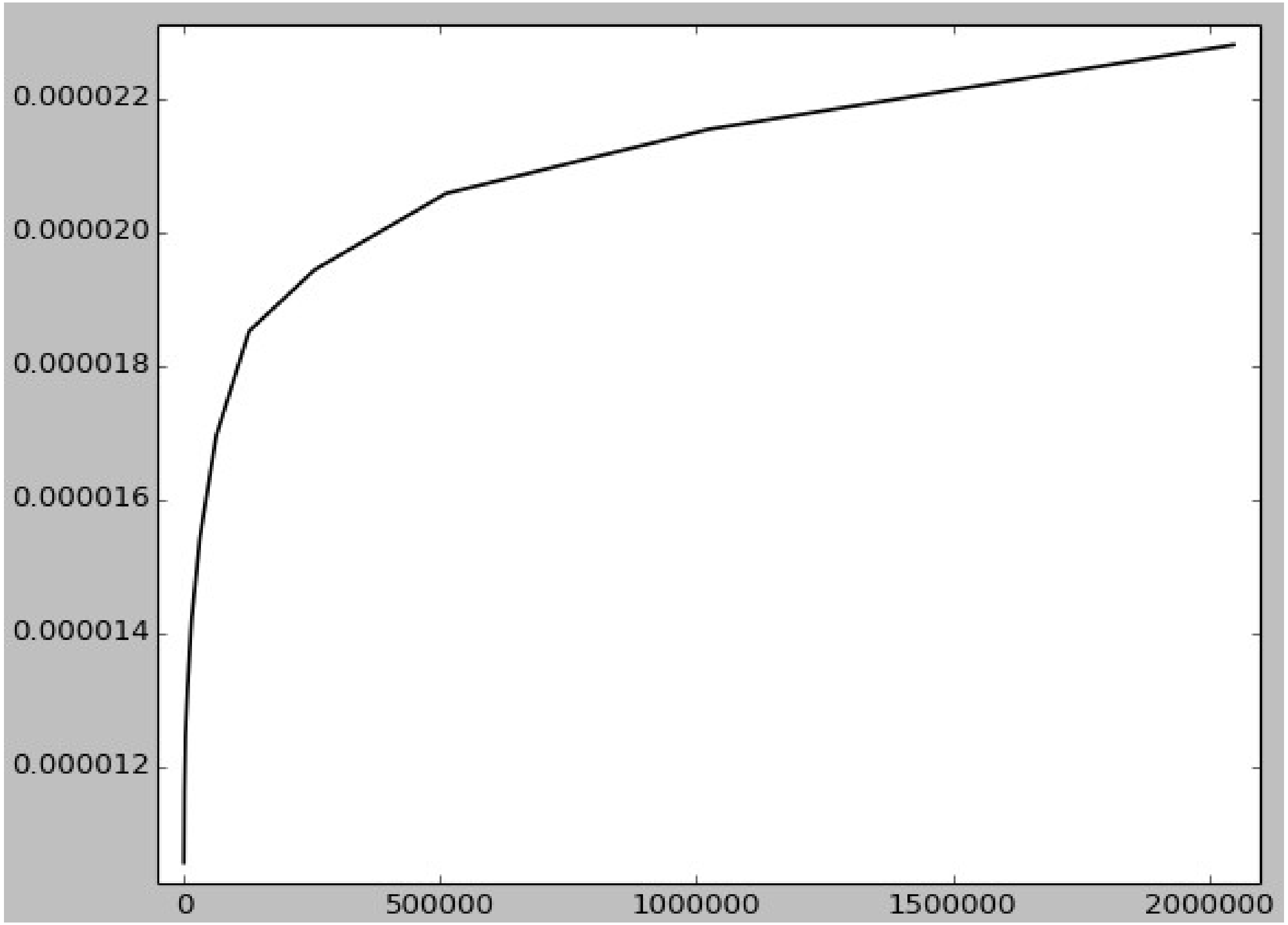

0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
100000
200000
300000
400000
500000

Sorted Greatest Up To

```java
static Integer sortedFind(int x, ArrayList<Integer> sList) {
    if (sList.isEmpty() || sList.get(0) > x)
        return null;
    int lower = 0;
    int upper = sList.size();  // one past the end
    while (upper - lower > 1) {
        int mid = (lower + upper) / 2;
        if (sList.get(mid) <= x)
            lower = mid;
        else
            upper = mid;
    }
    return sList.get(lower);
}
```

Analysis

- How many iterations of the loop?
- Initially, `upper – lower = n`.
- The difference is halved in every iteration.
- Can halve it at most $log_2(n)$ times before it becomes 1.
- $f(n) = a\ log_2(n) + b$, so $O(log(n))$.

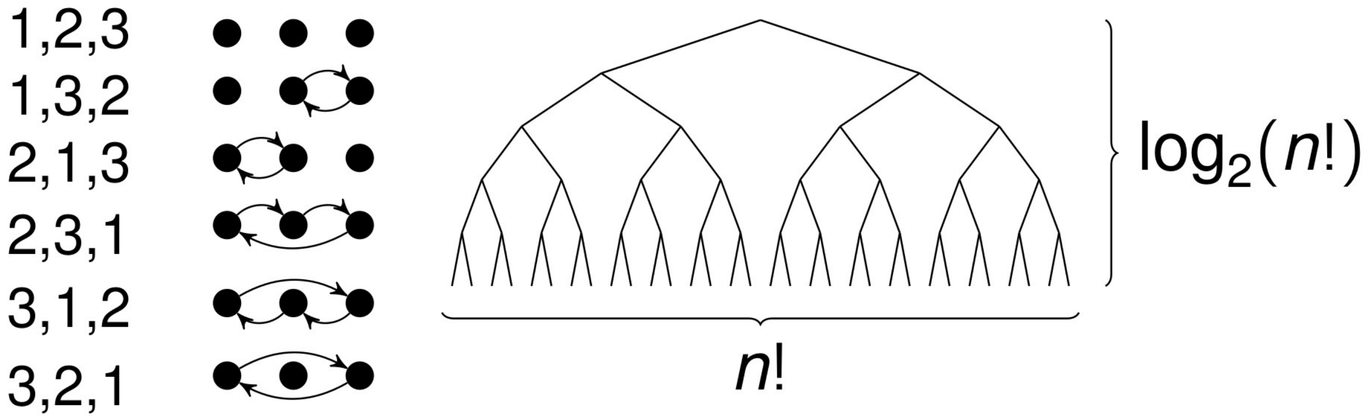

0.000022
0.000020
0.000018
0.000016
0.000014
0.000012
0
500000
1000000
1500000
2000000

Problem complexity

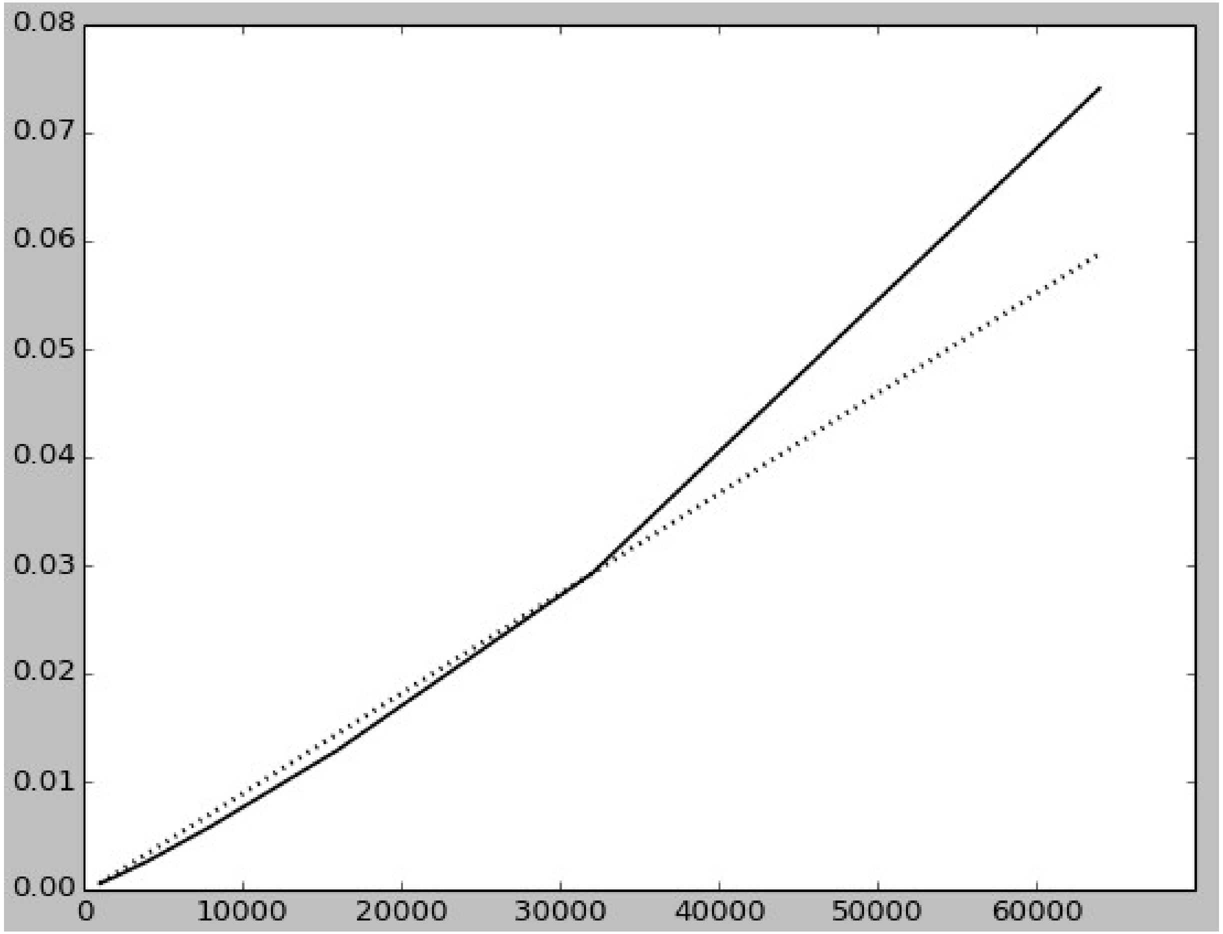
The complexity of a **problem** is the resources (time, memory, etc) that any algorithm *must* use, in the worst case, to solve the problem, as a function of instance size.

How fast can you sort?

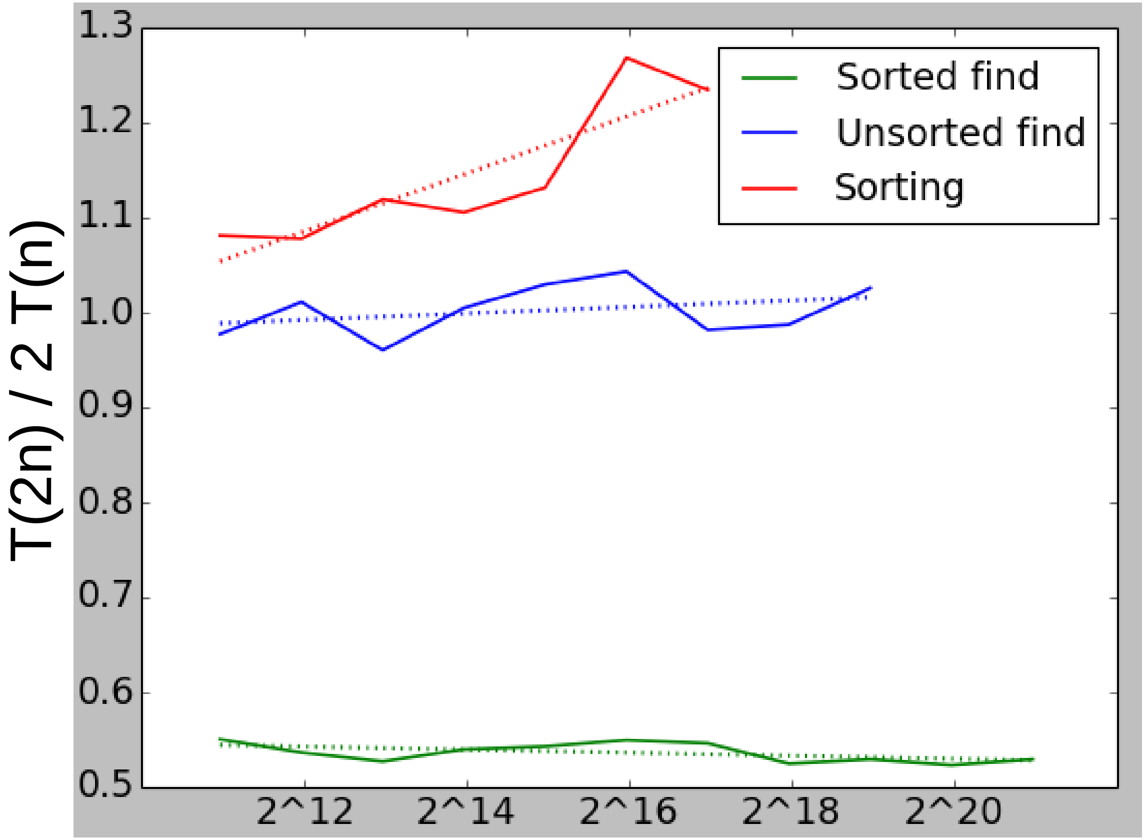
Any sorting algorithm that uses only pair-wise comparisons needs *O(n log(n))* comparisons in the worst case.



$log(n!) = log(1) + log(2) + \ldots + log(n) \leq n\ log(n)$ for large enough n.


0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
0
10000
20000
30000
40000
50000
60000

Rate of Growth



Caution

“Premature optimization is the root of all evil in programming.”

(C.A.R. Hoare)

Scaling behaviour becomes important when problems become large, or when they need to be solved very frequently.