Complexity Analysis

Dr Anton Gerdelan <gerdela@scss.tcd.ie>

Why do analysis?

- Which algorithm is going to be faster?

- How many items can I fit in memory using this data structure?

Types of Complexity

- Execution time complexity
  - **wall clock seconds** - “how long does the function take in seconds?”
  - **count of operations** - “how many code instructions?”
  - **computer clock tics** - “and if we add up the actual CPU cost of all the instructions?”
  - **count of investigations** - “how many things do we need to look at to e.g. find our item?”

Types of Complexity

- Space complexity
  - **bytes used per item**
  - **amount of storage actually filled vs wasted space**

Look at Several Cases

- Best case
- Worst case
- Average case
- How does it grow with different sizes *n*
  - number of “items” in data
  - items can be bytes / nodes / records / array indices / clients connected etc.

Counting Instructions

- Approx CPU time cost of each operation
  + - * / % =
- Function calls, pointer dereference, sqrt() etc etc.
- Problems with this?
- Function T(n)

Timers

- Problem - you have to actually implement the programs
- Depends on hardware tested on
- Hard to make a general comparison between algorithms
- Very commonly used in graphics/games/networks
  - Latency (usually in ms)
  - Frequency (Hz or frames per second)
  - Worst/best/average times

Bachmann-Landau Notation

aka “asymptotic notation”

- “*What type of **growth** function is this?*”
    - how does cost grow as size n increases?
- Don’t usually care about exact details
    - asymptotes are curves representing **limits** of a function
    - function never breaks upper or lower asymptote
        - .: upper and lower bounds
        - .: worst and best case approximations
    - we can also plot an average fit
    - all useful generalisations to describe our complexity

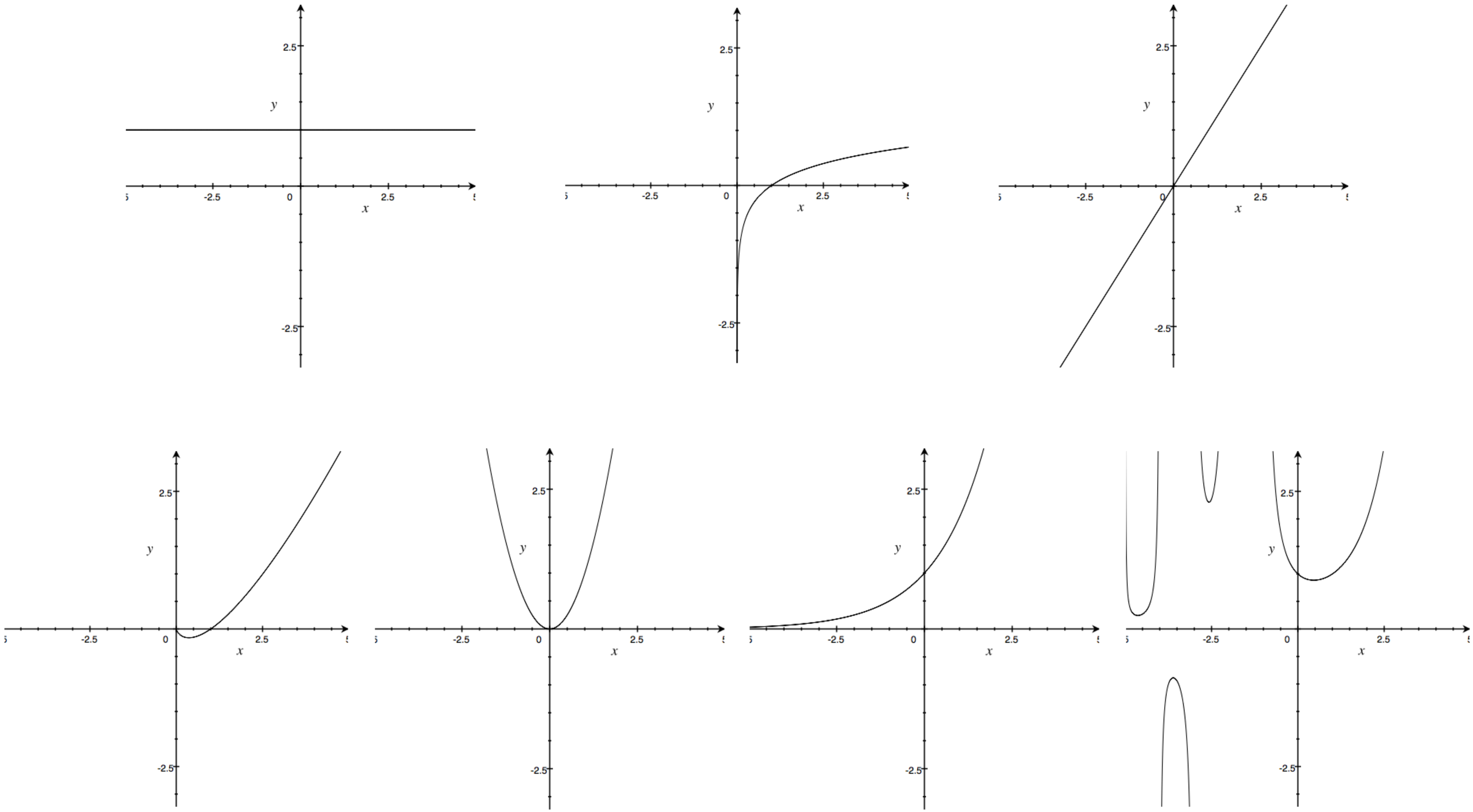
$\mathcal{O}\sim$

- Approximate “order” function ***O(n)***
- Called “Big O” (oh) notation - <u>very</u> commonly used.
- Big O is an **upper bound** -> cost in the <u>worst case</u>
  - i.e. if we don’t find out item until we’ve looked through all the others

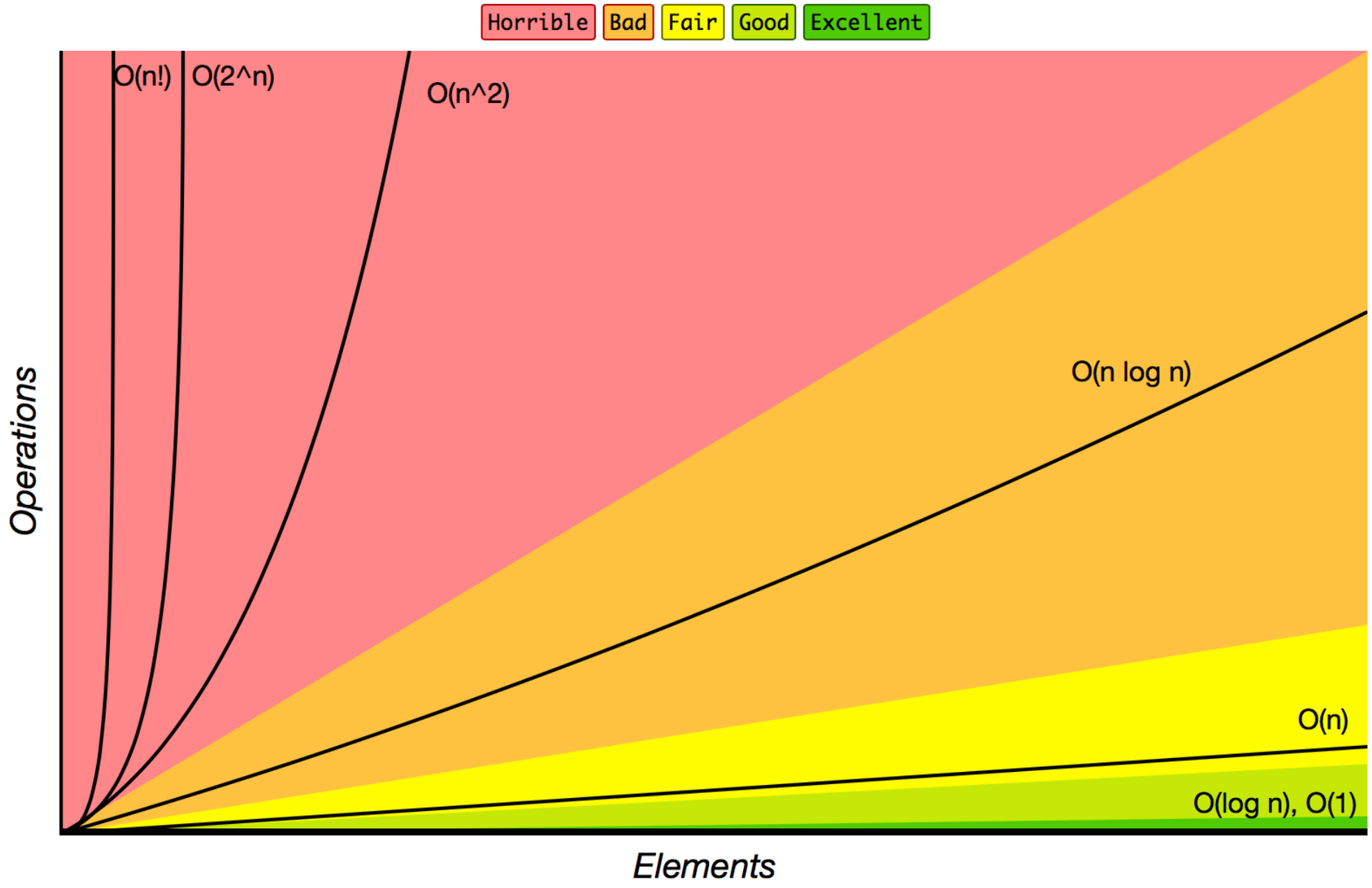
Fit Approximation

- Big O is a fit approximation
    - Collect some data on costs
    - “What type of function plot does this resemble?”
    - Choose closest match
- e.g. f*(n) = n² + n + 4*
    - usually drop the constants and lower order components - not a significant % of cost at large sizes of *n*.
    - can use *k* for constant if important
    - our algorithm is therefore order O(n²) - all quadratics

Which growth rate fits?



abandon all hope! *O(n!)*



src: http://bigocheatsheet.com
neat website - has a table / poster of complexities for all the fund. DS & A

Other Bachmann-Landau Notations

- O(n) - worst case - Big O - comparing functions
- Θ(n) - average case - Big Theta
- Ω(n) - best case - Big Omega
- Cormen et. al. are very specific about these
- Ties closely to mathematical field
- Often see O(n) notation used for all cases

Caveat I

- It only makes sense to compare complexity of same type
  - search algorithm vs search algorithm
  - data structure space vs data structure space
  - sorting algorithm vs sorting algorithm
  - n and O(n) have to measure ~the same thing

An Easy Way to Estimate O~ ?

```
// update all the boids
for (int i = 0; i < n; i++) {
  // move each boid away from all the other boids
  for (int j = 0; j < n; j++) {
    if (i == j) {
      continue;
    }
    Position their_pos = boids[j].position;
    move_boid_away_from( i, their_pos );
  }
}
```

implementing “Boids” algorithm from *Craig Reynolds’* “*Flocks, herds and schools: A distributed behavioral model.*". *SIGGRAPH '87*

Binary Search Algorithm

- E.g. looking for a name in a phonebook
- Assumption - search space is pre-sorted array/list.

1. Bisect search space into 2 lists (e.g. store index to start and middle)
2. Look up item at middle point
3. If desired value found -> halt
4. If mid item > desired value -> use left list, go to (2)
5. Else use right list, go to (2)

Binary Search

- A phonebook has 450,000 entries
- For searching time complexity T(n) we count if() statements - investigations
- What is T(n)?
  - Best case?
  - Worst case?
  - Average case?

Binary Search - Worst Case

#investigations	0	1	2	…	n
search space remaining	sz	sz / 2	sz / 4	…	sz / (2^n)

- what would a plot look like -> what Big O is this?

- best case?

- average case?
  - could estimate general trend we observe in table.
  - or plot large number of real T(0)…T(1)… values and use closest plot fit through data points.

Binary Search

- Worst case time - O(log n)
- Best case time - O(1)
- Average case time - O(log n)
- How does this compare to linear search (search every item from first to last until found)?

Linear Search

- Worst case time - O(n)
- Best case time - O(1)
- Average case time - O(n)
- Binary search is extremely fast - has fastest best, worst, and average times or common searches in order notation.
- Binary search is .: faster but requires a sorted array.
- Or is it…

Linear Search

- Does not require items to be sorted
- Can be done on a linked list
- Cache speed advantages for linear operations on arrays ~10-20x speed up (if set up right)
- Do not under-estimate the power of **brute force algorithms**
- If array is short linear search beats binary search
- Is phonebook with 450,000 entries a small size of n? - try it!
    - make a trivial program with a boring array of integers this big and put a timer for binary and linear search

Caveat II

- Don’t get hung up on growth difference for small sizes of *n*
- Most problems are very small sizes of *n*
- Even terrible-looking curves may be irrelevant
- This is why just looping over arrays is usually the best answer (and least bother)
- Looping over arrays also uses secret weapons at hardware level

log N	sqrt(N)	N	N log N	N^(3/2)	N^2
3	3	10	30	30	100
36	10	100	600	1,000	10,000
81	31	1,000	9,000	31,000	1,000,000
169	100	10,000	130,000	1,000,000	100,000,000
256	316	100,000	1,600,000	31,600,000	ten billion
361	1,000	1,000,000	19,000,000	one billion	one trillion

Adapted from *R. Sedgewick, Algorithms in C++, 1990*

Summary

- Mathematical analysis of algorithm or code
  - e.g. binary search space = sz + sz/2 + sz/4…
- Empirical analysis (actually measure times etc.)
  - sensitive to hardware, code quality, noise, temperature
- Math vs empirical can produce wildly differing results
- Beware comparing implementations that may be more or less optimised (common problem in CS)