Complexity and programming

Computational complexity of algorithms

Example: multiplication

- Big numbers harder than small numbers. How much harder?

		1	2	3
	$\times$	3	2	1
_	_	1	2	3
_	2	4	6	
3	6	9		
3	9	4	8	3

- For n digits have to perform n^2 single digit multiplications
- Add together n resulting n-digit numbers
- Overall number of operations is proportional to n^2: $\times 2$ number of digits will make problem four times harder
- Exactly how long this takes will depend on many things, but you can't get away from the basic quadratic scaling law of this algorithm

Defining complexity

- The **complexity** of a problem refers to this *scaling of the number of steps involved*
- Difficulty of particular task (or calculation) may vary considerably — 100×100 is easy, for example

- Instead ask about how a particular *general* algorithm performs on a *class* of tasks
- In CS multiplication of n digit numbers is a **problem**. *Particular pair* of n digit numbers is an **instance**
- Above algorithm for multiplication that has **quadratic complexity**, or "$O(n^2)$ complexity" (say "order n squared").

- Description only keeps track of how the difficulty scales with the size of the problem
  1. Allows us to gloss over what exactly we mean by a *step*. Are we working in base ten or binary? Looking the digit multiplications up in a table or doing them from scratch?
  2. Don't have to worry about how the algorithm is implemented exactly in software or hardware, what language used, and so on
  3. It is important to know whether our code is going to run for twice as long, four times as long, or 2^{10} times as long

Best / worst / average case

- Consider *search*: finding an item in an (unordered) list of length n. How hard is this?
- Have to check every item until you find the one you are looking for, so this suggests the complexity is $O(n)$
- Could be lucky and get it first try (or in first ten tries). The *best case complexity* of search is $O(1)$.
- Worst thing that could happen is that the sought item is last: the *worst case complexity* is $O(n)$
- On average, find your item near the middle of the list on attempt $\sim n/2$, so the *average case complexity* is $O(n/2)$. This is the same as $O(n)$ (constants don't matter)

- Thus for *linear search* we have:

	Complexity

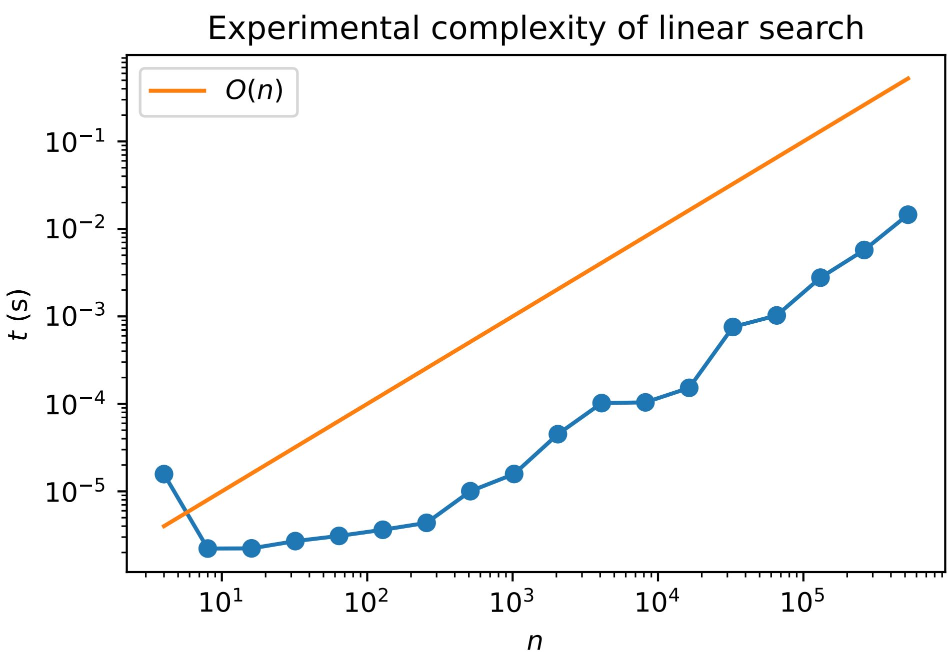
	Complexity
Best case	$O(1)$
Worst case	$O(n)$
Average case	$O(n)$

We can check the average case performance experimentally by using randomly chosen lists:

```
def linear_search(x, val):
    "Return True if val is in x, otherwise return False"
    for item in x:
        if item == val:
            return True
    return False
```

```
import numpy as np
# Create array of problem sizes n we want to test (powers of 2)
N = 2**np.arange(2, 20)

# Generate the array of integers for the largest problem to use in plotting times
x = np.arange(N[-1])

# Initialise an empty array to stores times for plotting
times = []

# Time the search for each problem size
for n in N:

    # Time search function (repeating 3 times) to find a random integer in x[:n]
    t = %timeit -q -n4 -r1 -o linear_search(x[:n], np.random.randint(0, n))

    # Store best case time (best on a randomly chosen problem)
    times.append(t.best)
```

```python
import matplotlib.pyplot as plt
# Plot and label the time taken for linear search
plt.loglog(N, times, marker='o')
plt.xlabel('$n$')
plt.ylabel('$t$ (s)')

# Show a reference line of O(n)
plt.loglog(N, 1e-6*N, label='$O(n)$')

# Add legend
plt.legend(loc=0)
plt.title("Experimental complexity of linear search")

plt.show()
```



- "Experimental noise" arises because don't have full control over exactly what computer is doing at any moment: lots of other processes running.
- Takes a while to reach the linear regime: overhead associated with starting the program

---

Polynomial complexity

- You've already learnt a lot of algorithms in mathematics (even if you don't think of them this way)
- Let's revisit some them through lens of computational complexity

---

Matrix-vector multiplication

- Multiplying a n-dimensional vector by a $n \times n$ matrix?

$$\sum_{j=1}^{n} M_{ij} v_j$$

- Sum contains n terms, and have to perform n such sums
- Thus the complexity of this operation is $O(n^2)$.

Matrix-matrix multiplication

$$\sum_j A_{ij} B_{jk}$$

- Involves n terms for each of the n^2 assignments of i and k. Complexity: $O(n^3)$

- To calculate $M_1 M_2 \cdots M_n \mathbf{v}$, do *not* calculate the matrix product first, but instead

$$M_1 \left(M_2 \cdots \left(M_n \mathbf{v}\right)\right)$$

[Wikipedia has a nice summary](#) of computational complexity of common mathematical operations

---

- If algorithm has complexity $O(n^p)$ for some p it has *polynomial complexity*
- Useful heuristic is that if you have p nested loops that range over $\sim n$, the complexity is $O(n^p)$

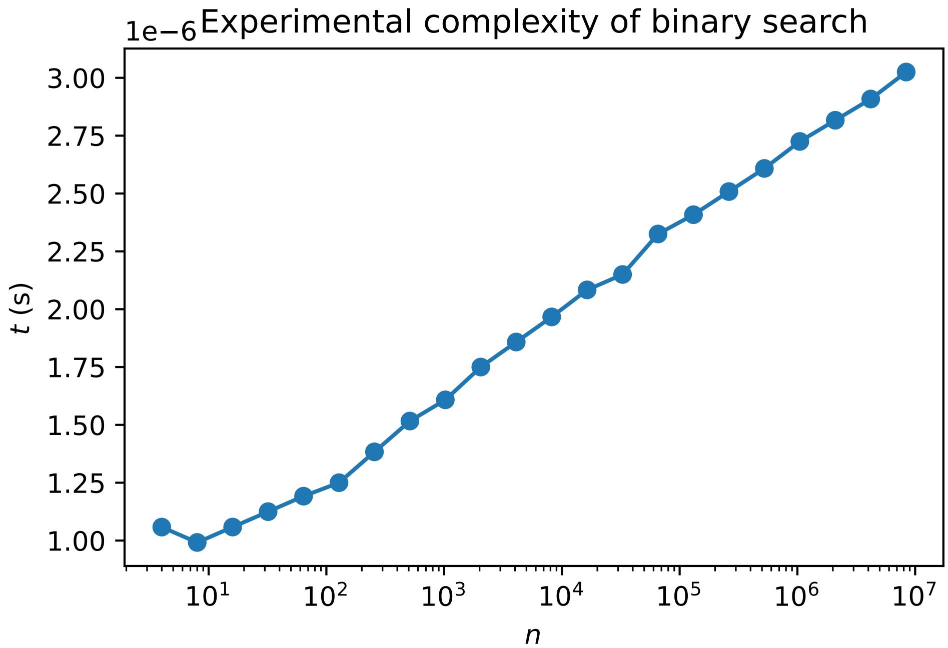
Better than linear?

- Seems obvious that for search you can't do better than linear
- What if the list is *ordered*? (numerical for numbers, or lexicographic for strings)
- Extra structure allows gives binary search that you may have seen before
- Look in middle of list and see if item you seek should be in the top half or bottom half
- Take the relevant half and divide it in half again to determine which quarter of the list your item is in, and so on

---

```python
def binary_search(x, val):
    """Peform binary search on x to find val. If found returns position, otherwise returns

    # Intialise end point indices
    lower, upper = 0, len(x) - 1

    # If values is outside of interval, return None
    if val < x[lower] or val > x[upper]:
        return None

    # Perform binary search
    while True:

        # Compute midpoint index (integer division)
        midpoint = (upper + lower)//2

        # Check which side of x[midpoint] val lies, and update midpoint accordingly
        if val < x[midpoint]:
            upper = midpoint - 1
        elif val > x[midpoint]:
            lower = midpoint + 1
        elif val == x[midpoint]:  # found, so return
            return midpoint

        # In this case val is not in list (return None)
        if upper < lower:
            return None
```

```python
# Create array of problem sizes we want to test (powers of 2)
N = 2**np.arange(2, 24)

# Creat array and sort
x = np.arange(N[-1])
x = np.sort(x)

# Initlise an empty array to capture time taken
times = []

# Time search for different problem sizes
for n in N:
    # Time search function for finding '2'
    t = %timeit -q -n5 -r2 -o binary_search(x[:n], 2)

    # Store average
    times.append(t.best)

# Plot and label the time taken for binary search
plt.semilogx(N, times, marker='o')
plt.xlabel('$n$')
plt.ylabel('$t$ (s)')

# Change format on y-axis to scientific notation
plt.ticklabel_format(style='sci', axis='y', scilimits=(0,0))
plt.title("Experimental complexity of binary search")
plt.show()
```



- If length is a power of 2 i.e. $n = 2^p$, we are going to need p bisections to locate our value
- Complexity is $O(\log n)$ (we don't need to specify the base as overall constants don't matter)

Divide and conquer

- Binary search and FFTs are examples of divide and conquer algorithms
- Achieve performance by breaking task into two (or more) sub-problems of same type

Practical Programming

We want to write programs which have the following attributes

- They produce results which are reliable.
- They produce results of acceptable accuracy.

- They produce results in an acceptable amount of time.

Adopting software engineering best practices will help achieve these aims

- Break the problem into small, well-defined parts.
- Write readable code.
- Test, test, test.

Most people start out writing code using the unstructured programming style:

- Put everything in the main program.
- All the data are defined there and used when needed.
- This is tolerable for "Hello, World" programs, say up to 20-30 lines.

Build large programs from small functions

- Write functions which solve a small, well-defined part of the problem
- Test these functions
- Combine them to solve the problem as a whole.

Breaking code into functions helps to manage *cognitive* complexity

- The number of possible interactions between different parts of a program increases as a **combinatorial** function of the number of variables which are shared between different parts of the program.
- Functions help to reduce "coupling" between different parts of the program, so the program state is easier to understand (c.f. separable vs non-separable Hamiltonians); functions help to isolate sections of code from the state of the rest of the program.
- Global variables and "side effects" increase coupling between functions and should be avoided where possible.

A “functional” style of programming aims to minimise coupling from “side effects”

```python
def test(x, y, z):
    x += 2
    y += (2,)
    z += [2]
    return x, y, z

a, b, c = 1, (1,), [1]
print("a =", a, "b =", b, "c =", c)
```

```
a = 1 b = (1,) c = [1]
```

```python
d, e, f = test(a, b, c)
f"{d=} {e=} {f=}"
```

```
'd=3 e=(1, 2) f=[1, 2]'
```

```python
f"{a=} {b=} {c=}"
```

```
'a=1 b=(1,) c=[1, 2]'
```

Write code which explains to humans what the computer should do

- Humans include yourself one week later
- **Use names to convey meaning.** Use descriptive names for important variables and functions, e.g. `derivative(x)` instead of `d(x)`; trivial loop variables may not always need a long name though.
- **Use comments to help the reader.** At minimum, explain what each function does and what its arguments are. However, *too much* commenting (e.g. one comment per line) can distract the reader from what the code itself is saying.
- **Use visual markers of logical structure**. Use “white space” generously (I have squeezed some of my examples for display purposes).

Test, test, test

- Test by running the code in situations where the answer is known.
- Test early: write small functions and test that these work (see "test driven design").
- Test often: put the functions together and test the result.

Debugging is a key programming (and experimental) skill

- Review what you have written (preferably with someone else - see "rubber duck debugging")
- Print out intermediate results and/or use a debugger
- We can use a "divide and conquer" algorithm to help locate bugs:
  - divide the program/function into "half";
  - check the correctness of the behaviour of each half;
  - subdivide the erroneous half (assuming there is only one);
  - repeat until you find the bug.
- Beware the **Heisenbug** — the bug that disappears when you try to observe it.

Version control helps to manage code complexity

- Version/revision control systems such as `git` or Mercurial (`hg`) save the history of your code (see also "Revision history" in Colab).
- Allows you to experiment
- Allows you to see what changed between versions