Today

- Efficiency of Algorithms
- Complexity classes

See Russell and Norvig, appendix A; a fuller account is in Aho, Hopcraft and Ullman, "Data structures and Algorithms", chapter 1.

Acknowledgements to Chris Mellish for slides.

Measuring Efficiency

Want a measure that is independent of
– which inputs are provided
– the hardware that is used.

The standard approach measures *complexity* in terms of

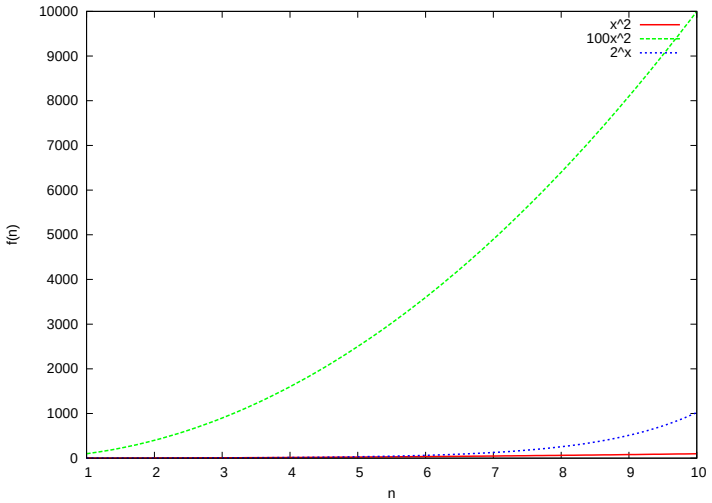
- a function which for a given input of size n gives the number of operations in the worst case; and
- that function being approximated by something giving the right order of magnitude.

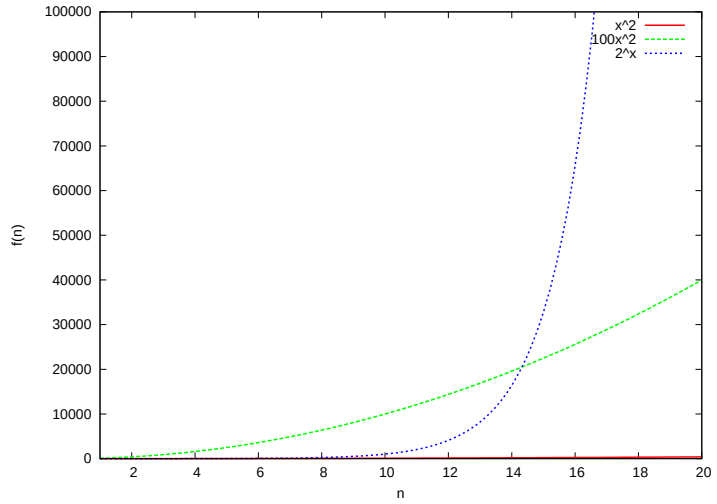
How does the work scale up as the input gets more complex?

Here look just at *time* (not space) and a single input only.

Some functions of n

$log_e(n)$	0	0.7	2.3	3	4.6	6.9
n	1	2	10	2×10	1×10^2	1×10^3
n^2	1	4	10^2	2×10^2	1×10^4	1×10^6
n^3	1	8	10^3	8×10^3	1×10^6	1×10^9
$100n^3$	10^2	8×10^2	10^5	8×10^5	1×10^8	1×10^{11}
n^4	1	16	10^4	2×10^5	1×10^8	1×10^{12}
n^{25}	1	3×10^{27}	10^{25}	3×10^{32}	1×10^{50}	1×10^{75}
$1000n^{25}$	10^3	3×10^{30}	10^{28}	3×10^{35}	1×10^{53}	1×10^{78}
2^n	2	4	10^3	1×10^6	1×10^{30}	1×10^{301}





Ranking of Functions of n

k
$log(n)$
n
n^2
n^3
$\ldots$
2^n
$\ldots$

Given a large enough n, a high function will always dominate a lower one.
– Multiplying by a constant makes no difference
– The sum of two functions behaves like the higher one.

Complexity Classes

- *constant* – the amount of work does not depend on n
- *logarithmic* – the amount of work behaves like $log_k(n)$ for some constant k
- *polynomial* – the amount of work behaves like n^k, for some constant k. More precisely, distinguish cases *linear* ($k = 1$), *quadratic* ($k = 2$), *cubic* ($k = 3$), . . .
- *exponential* – the amount of work behaves like k^n, for some constant k.

Orders of Magnitude

- Imagine a function that in the worst case takes $200 + 78n^2 + 12n^3$ steps.
- Describe this simply as cubic complexity, $O(n^3)$ in "big oh" notation.
- In practice, we do not need to work out the function in such detail in order to come to the conclusion.
- We can analyse the algorithm, or plot the time taken for various inputs.

Different Computing Devices

We attempt to get an idea of how the algorithm will run, regardless for example of the speed of the processor.

However, the distinctions within the polynomial class are not necessarily robust across different types of computing devices — may vary according to memory characteristics.

Determining Complexity

- Decide which input(s) complexity is to be relative to;
- Decide how "size" is to be measured;
- "Count" how many constant-time operations will happen in the worst case for an input of size n, bearing in mind that only order of magnitude will be required.

Constant time operations

- getting/setting the value in a given memory location
- arithmetic operations (add, multiply, . . .) (if not too large)
- checking if two numbers are the same (if not too large)
- testing the type of a data representation (list? real?)
- accessing/setting a component of a pair (d_1, d_2)
- accessing/setting a component of an array/record.

Example: Is something in a list?

The list data structure supports in constant time: finding first, finding rest, putting element on front

```
To see if X is in list Y
   Until Y is NIL do
      If X is the first of Y
        Return True as result, halt
      Else
        Set Y to rest of Y
   Return False
```

This is a version of pseudo-code, with standard imperative programming constructs.

Simulation

Is 3 in the list $[1, 2, 3, 4]$?

Work through the algorithm – how many steps are needed?

Complexity

- Let n be the length of the list (meaning what?)
- Worst case is when the item is *not* in the list
- At each step:
  – check if `Y` is `NIL`
  – get `first of Y`
  – test if `first of Y = X`
  – get rest of `Y`
  – set `Y`

Each is a constant time operation. So, complexity is *linear* in n

In general for loops, complexity is work in each iteration (if it's constant) $\times$ number of iterations.

Example: reversing a list

```
To reverse list A:
   If A is NIL, return NIL, halt
   Else
      Set B = NIL
      Until A is NIL do
         Set C = first of A
         Set A = rest of A
         Set B = C in front of B
      Return B, halt
```


Complexity

Same ideas

- Measure on input?
- How much work in the iteration?
- How many times through the loop?

Example: Sorting a vector

The vector data structure supports constant time access and update of the components.

```
To sort elements of vector V, size n:
   For X going down from n to 1 do
       For Y going from 1 up to X-1 do
           If V[Y+1] < V(Y) swap V[Y+1],V[Y]
```

This works – we'll ignore why.

Simulation

Sort `[|5,1,2,4|]`

Keep track of `X,Y`

Complexity

- The inner loop involves a bounded number of constant time operations (doesn't depend on n).
- The inner loop executes this `X-1` times
- The outer loop does this with `X` from n to 1
- Inner stuff executed $(n-1)+(n-2)+\cdots+1$ times
- This is quadratic
  $n-1$ terms each on average $n/2$, gives $\frac{1}{2}n(n-1)$;
  complexity is $O(n^2)$.

Summary

- Time complexity in worst case
- Basic complexity classes
- Analysing loops for basic data structures