Algorithms

A two-lecture course

Alan Mycroft
Computer Laboratory, Cambridge University
`http://www.cl.cam.ac.uk/users/am/`

August 2013

Map of the course

This two-lecture course is intended to model a typical university course (real courses are often 16 or 24 lectures long).

- Lecture 1: simple algorithms which you might have seen before, and perhaps feel that you could have invented some of them yourself. Based around *sorting* (maybe even helpful for people doing some A-level modules in Decision Maths or CS).
- Lecture 2: geometric algorithms. Chosen so you might say “I can’t even think of where to start” *beforehand* (in fact most members of staff here would have said that too!) and “that’s neat” *afterwards*.

We unashamedly use mathematics as a tool (for modelling, for reasoning); arguably the nearest part of CS to mathematics.

What's an algorithm?

- A precise finite description of a process for solving a problem.
- And the word: 9th century Persian mathematician al-Khwarizmi (the 'khw' was Europeanised into 'g'). He is one of the fathers of 'algebra' (al-Jabr) – precise manipulation again.
- What's it to do with 'logarithms'? Nothing (or no more than 'manifesto' and 'man').

How precise?

- like all mathematics – precise enough for the task in hand. We might assume the reader knows (say) long division when explaining some algorithm which uses it.

Wikipedia.org says far more about 'algorithm'...

Our first algorithm

- Problem: given a list of integers as input, produce a list containing the same integers, but in increasing order ('sorting').
- Algorithm: ('selection sort') find the smallest, output it and remove it from the input list. Repeat this process until the list is empty.

Two issues (bugs!)

- What if the list is empty at the start and therefore has no smallest? (careless wording – need to write precisely).
- More significant: 'I don't know how to find the smallest'. Need to explain down to a level the reader can understand. When *programming* the reader is a dumb computer.

Finding the minimum of a list

Provided the list is non-empty then:

- Take the first element as provisional minimum.
- For each remaining element (maybe zero of them!)
  - if this element is less than the provisional minimum then use this value as new provisional minimum.
- The provisional minimum now is the result.

Programming language can be clearer

```
int smallest(int A[], int k) =
{ int sofar = A[k];
  for j=1 to k-1 do
    if (A[j] < sofar)
      swap(A[j],sofar)
  return sofar;
}

void sort(int A[], int n) =
  for k=n downto 1 do
    print(smallest(A[],k))
```

This is a bit sneaky – especially scanning backwards in `sort` and using `swap` to remove the smallest from `A[1..k]` to yield `A[1..k-1]`.

English or Programming language?

- English can be imprecise, even when you think you're being careful. Think about:
  "everyone loves someone", or "pick the x for which ..."
  (e.g. $x^2 = -1$)

- English can be very verbose.

- But, programming languages are usually far more formal than (even) mathematics.
  All the programming tricks/nonsense (like scanning backwards and swapping) hide what's really going on.

The usual solution:

- Describe algorithms in "pseudo-code" a mixture of English and programming-like notation.

Sorting in place

```
int smallest(int A[], int k) =
{ int sofar = A[k];
  for j=1 to k-1 do
    if (A[j] < sofar)
      swap(A[j],sofar)
  return sofar;
}

void sort(int A[], int n) =
  for k=n downto 1 do
     A[k] = smallest(A[],k)
```

Here we're sorting the list (or array) `A[]` *in place* (have to be very careful not to tread on our own toes!).

BTW, this makes `A[1]` be the *largest element*, and `A[n]` the *smallest*.

Bubble sort – digression

Some of you might have seen so-called bubble sort.

We're not going to talk about it much here because it's basically the same as the algorithm I've explained – repeatedly put the minimum element at position `k` (the end of `A[1..k]`):

```
bubblesort(int A[], int n)
{   for k = n downto 2 do     note 2 or 1 is OK!
      for j = 1 to k-1 do
         if (A[j]<A[j+1]) swap(A[j],A[j+1]);
}
```

It's shorter, so when you've understood it you can say "that's neat", but it's harder at first than our decomposition. BTW, neither are *efficient* algorithms – we'll talk about this soon.

Problems and algorithm

Algorithms are like solutions to problems!

Do all problems have algorithms?

- Unclear: What's "The Ultimate Answer to The Great Question of Life, the Universe, and Everything" [Douglas Adams]? 42?

If a problem has an algorithmic solution, is it unique?

- No: here's a different (clearer, but slower) solution to sorting. "Make a list of all permutations of the input list, and choose a/the permutation which is increasing".

How do we choose among possible algorithms? Fastest? On which machine? For which input(s)?

Choosing between algorithms

A warning:

> "I've written a really neat program to do (whatever). It doesn't always get the right answer but it's really fast"

This isn't an algorithm to do "(whatever)". And, BTW, I can write an *even faster algorithm* which sometimes gets wrong answers – how about

> `MySuperFastAlgorithm`: "ignore input and print 42".

Correctness is vital, efficiency is secondary.

Choosing between algorithms (2)

Let's be speed-junkies for the moment and say:

- We compare algorithms by how fast they run.

But on which machine?

- We're going to use an abstract mathematical notion of time (counting steps in the computation), but this is generally realistic.

What if one algorithm wins on small input data, but is slower on large input data?

- We're going to treat *asymptotic* (large input size) behaviour as more important.

(Small problem inputs generally are solved quickly, it's the big inputs that we need to worry about.)

How fast is selection sort?

What's our abstract unit of time?

- time taken to do one comparison (and possible swap).
  (We can probably measure this to within a factor of two or so.)

So, let's look at

```
int smallest(int A[], int k) =
{ int sofar = A[k];
  for j=1 to k-1 do
    if (A[j] < sofar)
      swap(A[j],sofar)
  return sofar;
}
```

cost of `smallest(A[],`k`)` is $k - 1$.

How fast is selection sort? (2)

```
void sort(int A[], int n) =
  for k = n downto 1 do        note 2 or 1 is OK!
    print(smallest(A[],k))     remember: cost is k-1
```

Cost (let's call it S_n) of `sort(A[],`n`)` is:

$$S_n = (n-1) + (n-2) + \cdots + (1-1)$$

But this is an *arithmetic progression* (so reverse and add to solve):

$$\begin{aligned} S_n &= 0 + 1 + \cdots + (n-1) \\ 2S_n &= (n-1) + (n-1) + \cdots + (n-1) \end{aligned}$$

Hence:

$$S_n = n(n-1)/2 \quad \text{(phew)}.$$

How fast is selection sort? (3)

$$S_n = n(n-1)/2 = n^2/2 + n/2$$

But when n is big, the n^2 dominates, so we often write

$$S_n \approx n^2/2$$

or even

$$S_n \in O(n^2)$$

(some authors write $S_n = O(n^2)$ instead).

Digression: $O(n^2)$ means the set of functions which ultimately grow no faster than n^2 while ignoring constant factors.

$O(n^2)$ implies doubling n quadruples the result (cost).

Digression: $O(g)$ "big O" notation

$f \in O(g)$ if f ultimately grows no faster than g while ignoring constant factors.

- notation/idea is a good match for our desire for *asymptotic* (large input) bound on time taken

E.g.

$$
\begin{aligned}
n^2/2 &\in O(n^2) \\
n^2 + 1000 &\in O(n^2) \\
1000n^2 &\in O(n^2) \\
n^3/1000 &\notin O(n^2)
\end{aligned}
$$

We say "selection sort is an $O(n^2)$ algorithm".

What if the time taken depends on the data

UNIVERSITY OF CAMBRIDGE

```
lookup(int key, int A[], int n) =
{ for j = 1 to n do
      if (A[j] == key) return YES;
  return NO;
}
```

How much time (again counting comparisons) should we count?

- The best case (1)?
- The worst case (n)?
- The average case ($n/2$) assuming the item is going to be found?
- The average case ($0.95n$) assuming the item has 10% chance of being in the list?

What if the time taken depends on the data (2)

Almost always take the worst case. We don't want nasty surprises. "safety-critical systems".

Also *testing* to see how fast something runs is a *bad idea* unless you really are testing worst cases.

[Aside (rant!)] beware 'quicksort' – this has worst-case cost of $O(n^2)$ and a deceptively-good average-case cost of $O(n \log n)$ – especially if you don't want to get sued!

Sedition:

- If you're doing A-level Computer Science and it's suggested that quicksort is good, then ask "why is the worst-case speed so bad?" [Apologies to all teachers with such troublesome students]!

Mergesort – a ‘good’ sorting algorithm

Algorithm:

- if only one (or zero!) item in the list return it unchanged.
- if there are $n \geq 2$ items then
  - split into two lists (size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$)
  - ask someone else to sort these two lists using mergesort (recursion – a friend of ‘induction’)
  - *merge* the two resulting lists

But what’s ‘merging’?

Mergesort – a 'good' sorting algorithm (2)

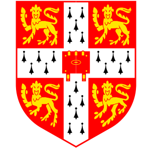
Merging is a playing-card "ripple shuffle" with intelligence ...

- while both lists (L and M) have items, pick the smaller of $L[1]$ and $M[1]$, output it and remove it from the appropriate list.
- when one list runs out merely output items from the other in turn

E.g. `merge`$([1, 8, 9, 15], [2, 3, 27, 28]) = [1, 2, 3, 8, 9, 15, 27, 28]$

Analysis (slightly over-pessimistic): worst-case cost of merging lists having l items and m items is $l + m$.

Mergesort – a 'good' sorting algorithm (3)



Let's write $M(n)$ to be to cost of using *mergesort* to sort a list of n items (and let's assume n is a power of two for simplicity).

We find:

- $M(1) = 0$
  (no comparisons to sort a one-element list)
- $M(n) = M(n/2) + M(n/2) + n$
  (two sublists to sort recursively and then a merge).

This is a *recurrence relation*.

Mergesort – a 'good' sorting algorithm (4)

Given

$$M(n) = n + 2M(n/2)$$

let's *guess* the solution by trial and error:

$$\begin{aligned}
M(64) &= 64 + 2M(32) \\
&= 64 + 2(32 + 2M(16)) \\
&= 64 + 64 + 4M(16) \qquad \ldots \\
&= 64 + 64 + 64 + 64 + 64 + 64 + 64M(1) \\
&= 64 \times 6 + 0 \\
&= 64 \log_2 64.
\end{aligned}$$

General solution (can *prove* by induction) is:

$$M(n) = n \log_2 n.$$

Speed: selection sort vs. merge sort

Selection sort takes time $n^2/2$.

Merge sort takes time $O(n \log_2 n)$, but with more administrative work (so we should perhaps allow for an additional cost factor of 4 or so).

Tabulate:

n	$n^2/2$	$n \log_2 n$	
2	1	2	
4	8	8	
16	128	64	
1024	512×1024	10×1024	50 times faster
10^6	$10^{12}/2$	20×10^6	25,000 times faster

The magnitude of the speedup justifies our earlier decision to "ignore constant factors" in $O(f(n))$ notation.

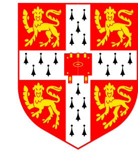
The really big picture of where we are

Spectrum:

- woolly thoughts: e.g. “I want an air traffic control system”; or “The government wants all NHS records to be computerised”.
- specification: near-formal English, e.g. “at all time planes must be 1000m apart (except when they are taxiing, landed, hmm...)”
- algorithm: precise set of actions which enables the 1000m separation to be achieved
- program: millions of lines of code in some programming language.

Other things I might say if I have time

- Can mergesort be done 'in place' – putting the sorted answer list in the same place as the input list without additional storage?
  [No.]
- Is there a better list sorting algorithm than $O(n \log n)$?
  [No, we can use mathematics to show that all sorting algorithms based on comparison need at least this number of comparisons.]
- Aren't mergesort and quicksort rather similar (both recursive)?
  [No, mergesort always halves the problem size (hence guaranteed speed); but in bad cases quicksort only reduces the problem size by one each step (in these cases it's very slow).]



Lecture 2

Lecture 2

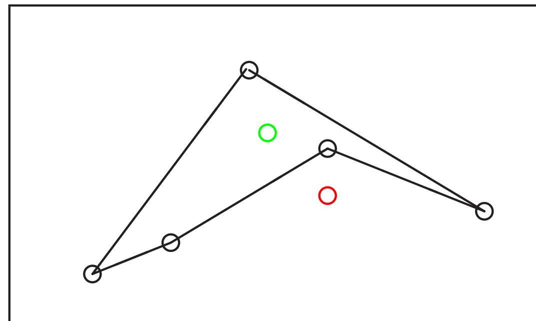
- Lecture 2: geometric algorithms. Chosen so you might say "I can't even think of where to start" *beforehand* (in fact most members of staff here would have said that too!) and "that's neat" *afterwards*.
- another reason: gentle link with (different parts of) mathematics too – perhaps useful for vectors in A-level maths?
- practical uses: computer graphics (output), computer vision (input), geometric modelling (everything from Sat-nav (GPS), robotics, computer-aided 3D-design, collision detection, games) etc.

The Inside-or-Outside a Polygon Problem

Given a closed polygon, expressed as a list of points

$$[(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)]$$

then is another point (x_0, y_0) inside it or outside it?



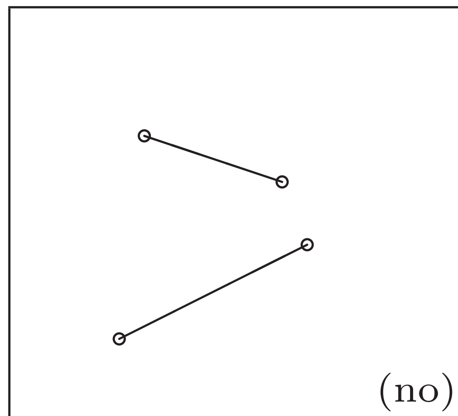
Two-year old humans can do it, but it's really quite non-obvious how to program a computer to do it! It is simple to start by reading $2n + 2$ numbers but then ...???

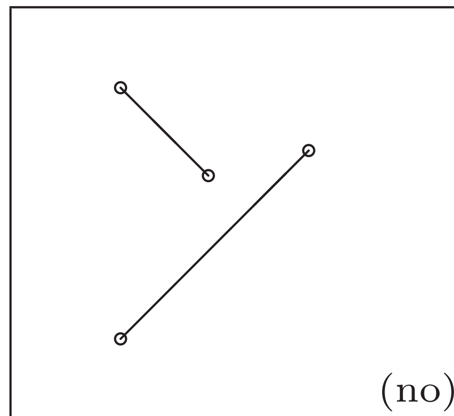
Excursion: Geometry on a computer

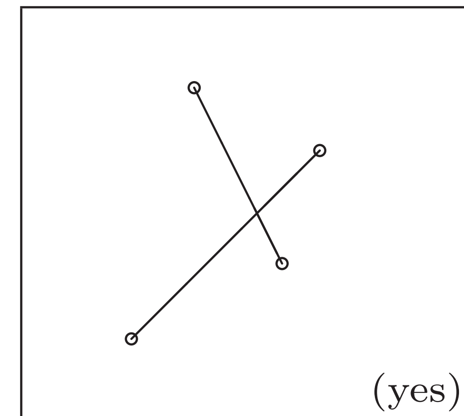
Collision detection – when do two lines intersect?

- Always – unless they are parallel.

But what if they are *line segments*?







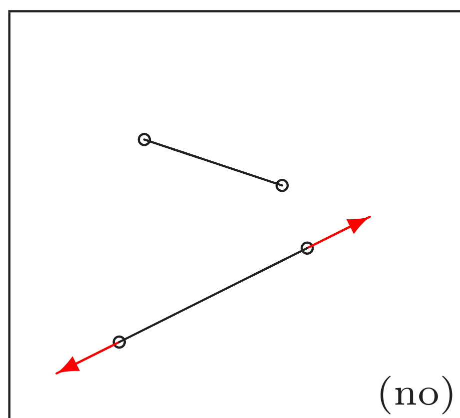
So: given 2 line segments (i.e. 4 points) do they intersect?

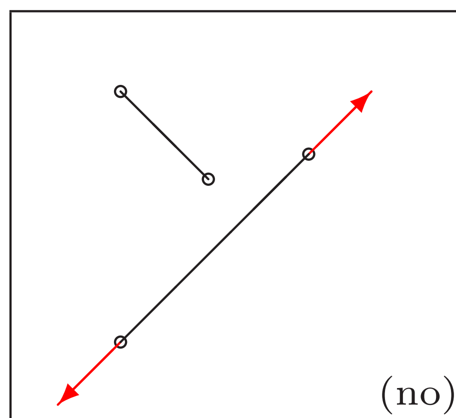
The line segment intersection problem

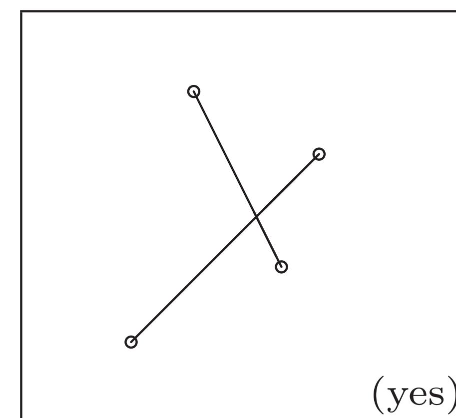
Given four points P_1, P_2, P_3, P_4 when does line *segment* P_1P_2 intersect with line segment P_3P_4?

First step – exactly when [need inspiration to invent this!]:

- P_1 and P_2 are on opposite sides of P_3P_4 (infinitely extended); and
- P_3 and P_4 are on opposite sides of P_1P_2 (infinitely extended).



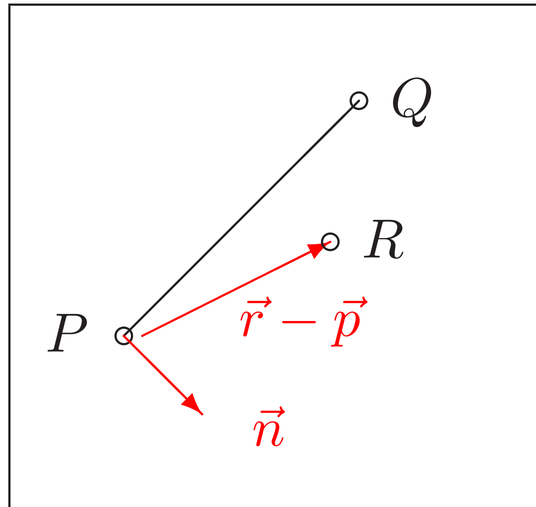




Why: take two intersecting lines and explore ways of trimming them.

The line segment intersection problem (2)

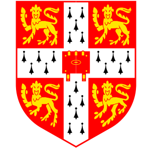
So, want to know on which side of a line PQ a point R falls.



Draw a normal $\vec{n}$ to PQ, and find the angle between this and $\vec{r} - \vec{p}$. Less than 90° means R is on the same side as $\vec{n}$ and greater than 90° means the opposite side of the line.

Finding the normal is easy (remember GCSE maths $mm' = -1$ for perpendicular lines?) If $\vec{q} - \vec{p} = (u, v)$ then $\vec{n} = (v, -u)$.

The line segment intersection problem (3)



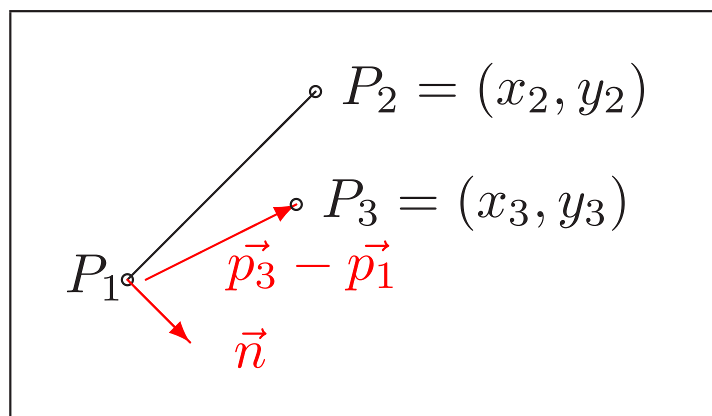
Now, sorry, just a little bit of A2 Maths you've probably not done yet (the 'dot product of two vectors').

Given two vectors $\vec{u} = (a, b)$ and $\vec{v} = (c, d)$, we define $\vec{u} \cdot \vec{v} = ac + bd$. Now it turns out that $\vec{u} \cdot \vec{v} = ||u|| \; ||v|| \; \cos\theta$ where θ is the angle between $\vec{u}$ and $\vec{v}$ (ask your teacher or look at Wikipedia).

But this is just what we want:

> $ac + bd$ will be negative if θ is greater than 90° degrees, and positive if less.

The line segment intersection problem (4)



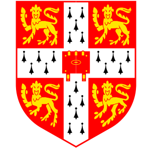
We calculate:

$$
\begin{aligned}
P_1P_2 &= (x_2 - x_1, y_2 - y_1) \\
\vec{n} &= (y_2 - y_1, -(x_2 - x_1)) \\
\vec{n} \cdot (\vec{p_3} - \vec{p_1}) &= (y_2 - y_1)(x_3 - x_1) - (x_2 - x_1)(y_3 - y_1)
\end{aligned}
$$

So P_3 and P_4 are on opposite sides of P_1P_2 iff

$$
\begin{aligned}
&((y_2 - y_1)(x_3 - x_1) - (x_2 - x_1)(y_3 - y_1)) \times \\
&((y_2 - y_1)(x_4 - x_1) - (x_2 - x_1)(y_4 - y_1)) < 0
\end{aligned}
$$

The line segment intersection problem (5)



So, line segments P_1P_2 and P_3P_4 intersect if

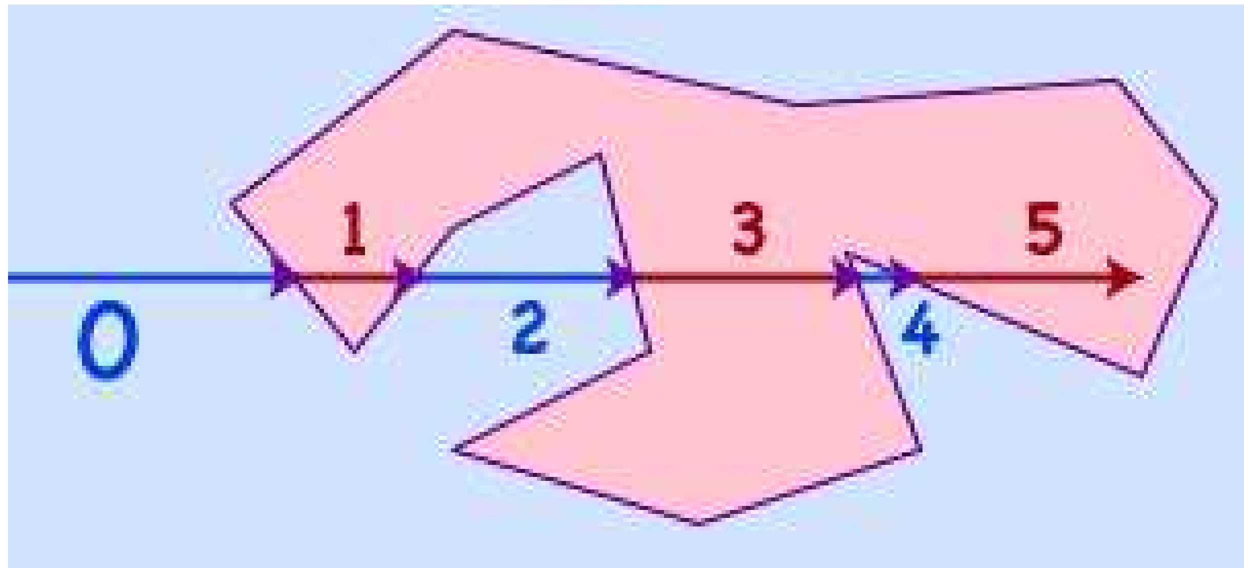
$$\begin{array}{ll} ((y_2 - y_1)(x_3 - x_1) - (x_2 - x_1)(y_3 - y_1)) \times & \\ ((y_2 - y_1)(x_4 - x_1) - (x_2 - x_1)(y_4 - y_1)) & < \quad 0 \end{array}$$

and

$$\begin{array}{ll} ((y_4 - y_3)(x_1 - x_3) - (x_4 - x_3)(y_1 - y_3)) \times & \\ ((y_4 - y_3)(x_2 - x_3) - (x_4 - x_3)(y_2 - y_3)) & < \quad 0 \end{array}$$

No-one would ever have guessed this – but it's very quick to calculate.

The Inside-or-Outside a Polygon Problem (2)



[diagram source: Wikipedia]

Observation: a point P is *inside* a polygon if a line drawn from infinity to P crosses an odd number of edges of the polygon!

For ‘infinity’ we can (probably) just use $((\max x_i) + 1, (\max y_i) + 1)$.

The Inside-or-Outside a Polygon Problem (3)

But this is just the line segment intersection problem . . .

```
IsInside(point p, polygon G)
{   let q be a point outside G (‘infinity’)
    int count = 0;
    for E an edge of G do
         if (pq intersects E) count := count + 1;
    if (count is odd) return YES;
    return NO;
}
```

The Inside-or-Outside a Polygon Problem (4)

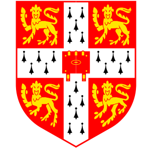
Choosing the point at infinity. I said:

> "we can (probably) just use $((\max x_i) + 1, (\max y_i) + 1)$."

but I need to make sure that the line does not intersect a vertex (graze the polygon) – because then my "edge crossing" algorithm will give the wrong result.

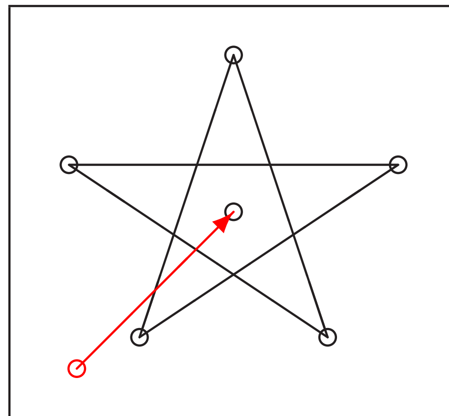
So need a bit more care about choosing it. There are many ways, for example there's a simple algorithm to test whether 3 points (P, Q, R) are on (or nearly on) the same line – get vector PQ and PR and find the angle between them.

The Inside-or-Outside a Polygon Problem (5)



A final subtlety: do *we* understand the idea of inside and outside? Because if we don't understand it fully then our algorithms may not work as expected in areas we didn't think of . . .

Consider:



Is the central point inside or outside the polygon? *Perhaps* our algorithm gives wrong answers for self-intersecting polygons?
[See `http://en.wikipedia.org/wiki/Point_in_polygon` for more.]

How likely are *'unusual'* cases?

If we take four random points in a plane, then the mathematical probability of:

- two of them being identical; or
- any two lines connecting them being parallel

is vanishingly small (even 'Probability zero').

So, I don't have to worry about this when programming?

Wrong!

People draw squares or abut two objects together. Real geometric examples result in lots of horrible 'unlikely' cases.

Another nasty: computer floating-point arithmetic (which models real numbers) is not exact. Problems when points are very close.

The Sat-nav display problem

Let's assume someone else has done all the complicated decoding of GPS satellite signals, and got us an (x, y) co-ordinate in the UK.

Our sat-nav has a map of the UK expressed as a list of (straight) line-segments. (Curvy roads are just lots of segments.)

Problem: which road are we on?

- almost certainly none – because GPS is not 100% precise

So, what do we do?

- search all(!) line segments for the one which has a point closest to the GPS co-ordinate. Probably display *this point* on the map rather than actual the GPS co-ordinate to avoid showing me driving down a field at the side of the road!
  It might be entertaining trying to do this efficiently!

Discrete Geometric Algorithms

Often in computational geometry, or in computer games, we don't have a mathematical plane (x, y) with x and y being *real numbers*, but we have a discretised version – i.e. x and y are integers. E.g.

- graphic pixels (Microsoft paint, or Adobe photoshop);
- a game grid like Minesweeper.

Discreteness often needs care for the above algorithms (e.g. could diagonal lines at 45° pass through each other without intersecting?).

But it offers scope for other new algorithms:

- Image processing (e.g. blurring/pixelating faces)
- Game route-planning searches (PacMan and later).

Boundary fill

Used in painting programs (and, after tweaking, in MineSweeper).

Start at (x, y) and mark the *connected* region of (say) RED cells starting there by colouring it all BLACK (red-eye removal)

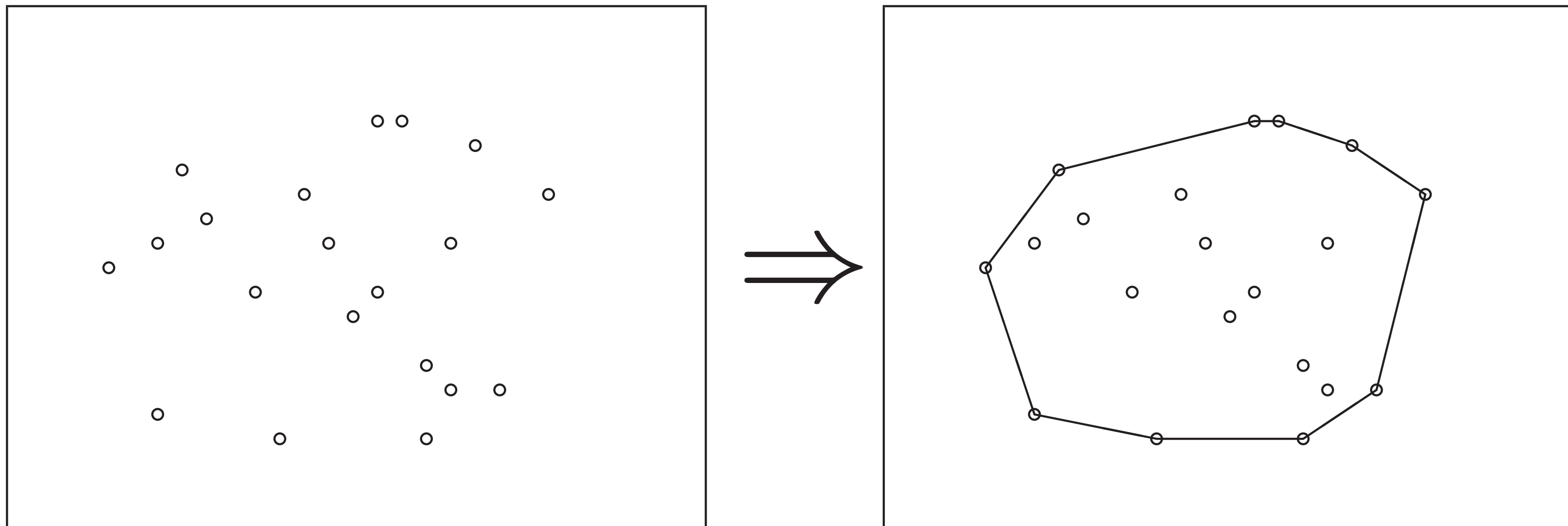
Assumes a non-RED boundary delimits RED cells (so we don't fall off the end of `A[]`) . . .

```
void boundaryFill(int x, int y)
{   if (A[x,y] == RED)
    {   A[x,y] = BLACK;
        boundaryFill(x+1,y);
        boundaryFill(x-1,y);
        boundaryFill(x,y+1);
        boundaryFill(x,y-1);
    }
}
```

Convex Hull

Here's another neat problem which is easy when you understand it (i.e. a nice solution to sell) but which is hard to understand even how to start (i.e. your boss needs to pay you lots of money to solve it).

Given some points $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ on the plane (think nails in pin-board) which ones would an elastic band touch?



Lecture $2\frac{3}{4}$

[J.K. Rowling gratefully acknowledged]

More research oriented – uses more sophisticated maths [magic?] than A-level (number theory for cryptography or logic for 'program proving').

Safety-critical or cash-critical

Software can be safety critical:

- Airbus A320-series and up uses "fly-by-wire".

Need high quality engineering (including mechanised verification).

Software can be also security-critical:

- Much of the world's banking security (encryption) relies on the fact that no one knows how to do things like efficiently factor a product of two primes. E.g. $1591 = 37 \times 43$ or $25957 = 101 \times 257$ are hard for humans. All known algorithms involve dumb searching.
  So we pick primes large enough that computer searching would take longer than the universe has existed ...
  But what if someone finds a better algorithm?

Conclusions

- Algorithms are like the essence of programs.
- We want to *analyse* them to compare their efficiencies
- They can be non-obvious – and until we've designed the *algorithm* we can't even think of writing a *program*.
- Mathematics can help you design and analyse algorithms.
- There are still algorithms to find (or to prove cannot exist!).