Week 8

Geometry I

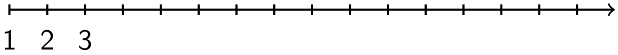
November 5, 2015

Plan for today

1. Exercises: Boats, Travel Costs, Permutation Pairs
2. Geometry

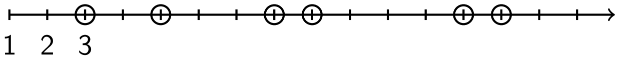
Exercise Discussion

Boats



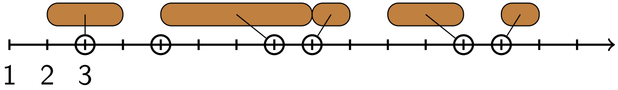
- Each wizard has a dedicated ring.
- The bounds of each boat must contain the corresponding ring.
- We need to place the maximum number of boats.

Boats



- Each wizard has a dedicated ring.
- The bounds of each boat must contain the corresponding ring.
- We need to place the maximum number of boats.

Boats



- Each wizard has a dedicated ring.
- The bounds of each boat must contain the corresponding ring.
- We need to place the maximum number of boats.

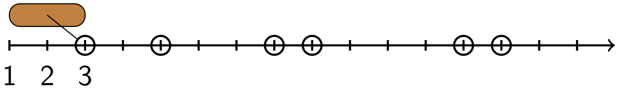
Boats

The problem has a greedy solution.

- Process the rings from left to right.
- For each ring:
  1. if possible, place the boat for this ring as far to the left as possible;
  2. if there is no place for the boat (because the previous boat overlaps the ring), check if you can reduce the right-most point of the solution by removing the previous boat and placing the current boat as far to the left as possible; if yes, do so.

Boats

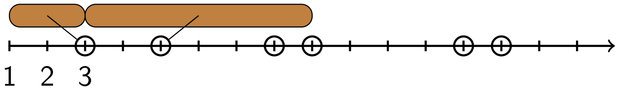
Worked example:



Place the first boat as far to the left as possible.

Boats

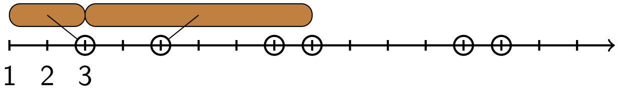
Worked example:



The second boat can be placed – again as far to the left as possible.

Boats

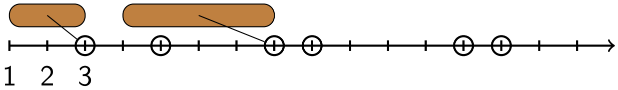
Worked example:



There is no space for the third boat!

Boats

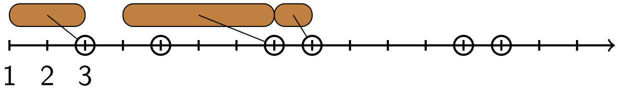
Worked example:



But by removing the previous boat, we move the right-most point ot the left.

Boats

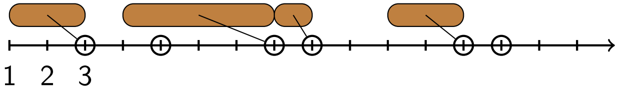
Worked example:



The remaining boats can be placed without problems.

Boats

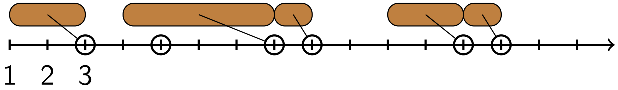
Worked example:



The remaining boats can be placed without problems.

Boats

Worked example:



Done!

Boats

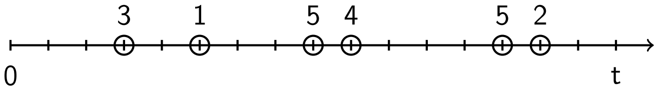
This algorithm is optimal. In fact:

Lemma

After the i-th step, the partial solution computed by the algorithm is an optimal solution on the first i boat/ring pairs, and, among all optimal solutions, it is one for which the right-most pair is as far left as possible.

Proof. By induction.

Travel Costs



- n gas stations with fuel prices are placed along the travel distance.
- Start at 0 with 100L of fuel. Travelling 100km requires 10L.
- Problem: what is the cheapest way to reach the destination t?

Travel Costs

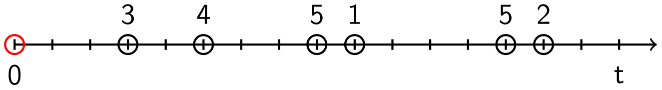
The solution is very intuitive:

- At every station, fill up just as much as you need to get either to the destination or to a cheaper gas station.
- If this is not possible, fill up the tank completely.

Travel Costs

The solution is very intuitive:

- At every station, fill up just as much as you need to get either to the destination or to a cheaper gas station.
- If this is not possible, fill up the tank completely.
- Example (each tick is 200km):

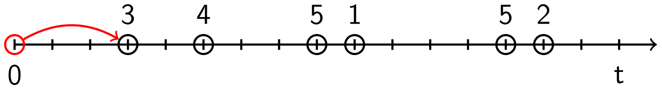


Start with 100 litres.

Travel Costs

The solution is very intuitive:

- At every station, fill up just as much as you need to get either to the destination or to a cheaper gas station.
- If this is not possible, fill up the tank completely.
- Example (each tick is 200km):

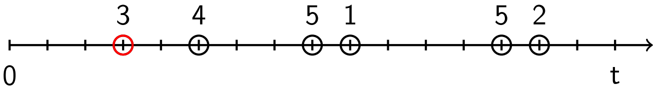


Move to first station.

Travel Costs

The solution is very intuitive:

- At every station, fill up just as much as you need to get either to the destination or to a cheaper gas station.
- If this is not possible, fill up the tank completely.
- Example (each tick is 200km):

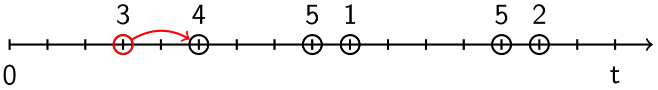


40L left. No cheaper station is in range: fill up completely.

Travel Costs

The solution is very intuitive:

- At every station, fill up just as much as you need to get either to the destination or to a cheaper gas station.
- If this is not possible, fill up the tank completely.
- Example (each tick is 200km):

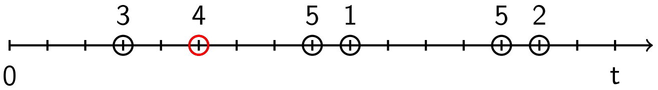


100L left. Go to next station.

Travel Costs

The solution is very intuitive:

- At every station, fill up just as much as you need to get either to the destination or to a cheaper gas station.
- If this is not possible, fill up the tank completely.
- Example (each tick is 200km):

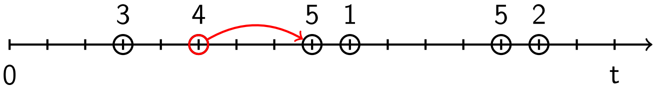


60L left. A cheaper station is in range, but we need to buy 20L to get there.

Travel Costs

The solution is very intuitive:

- At every station, fill up just as much as you need to get either to the destination or to a cheaper gas station.
- If this is not possible, fill up the tank completely.
- Example (each tick is 200km):

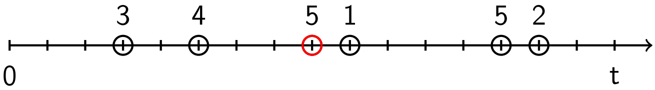


80L left. Go to next station.

Travel Costs

The solution is very intuitive:

- At every station, fill up just as much as you need to get either to the destination or to a cheaper gas station.
- If this is not possible, fill up the tank completely.
- Example (each tick is 200km):

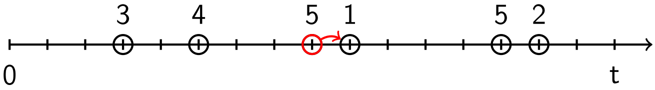


20L left. There is a cheaper station in range, and we already have enough fuel to get there.

Travel Costs

The solution is very intuitive:

- At every station, fill up just as much as you need to get either to the destination or to a cheaper gas station.
- If this is not possible, fill up the tank completely.
- Example (each tick is 200km):

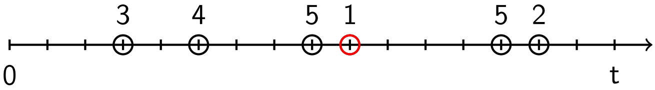


Go to next station.

Travel Costs

The solution is very intuitive:

- At every station, fill up just as much as you need to get either to the destination or to a cheaper gas station.
- If this is not possible, fill up the tank completely.
- Example (each tick is 200km):

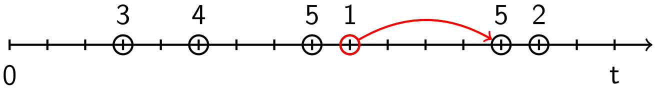


0L. No cheaper station in range: fill up completely.

Travel Costs

The solution is very intuitive:

- At every station, fill up just as much as you need to get either to the destination or to a cheaper gas station.
- If this is not possible, fill up the tank completely.
- Example (each tick is 200km):

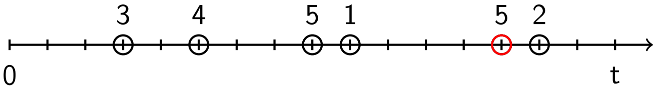


100L are enough to reach the next station.

Travel Costs

The solution is very intuitive:

- At every station, fill up just as much as you need to get either to the destination or to a cheaper gas station.
- If this is not possible, fill up the tank completely.
- Example (each tick is 200km):

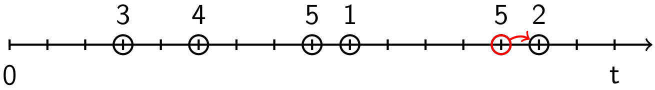


20L left. A cheaper station is reachable.

Travel Costs

The solution is very intuitive:

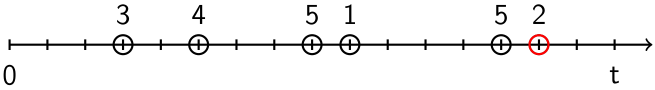
- At every station, fill up just as much as you need to get either to the destination or to a cheaper gas station.
- If this is not possible, fill up the tank completely.
- Example (each tick is 200km):



Travel Costs

The solution is very intuitive:

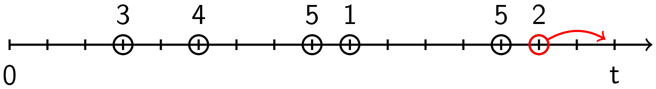
- At every station, fill up just as much as you need to get either to the destination or to a cheaper gas station.
- If this is not possible, fill up the tank completely.
- Example (each tick is 200km):



0L left. The destination is in range – fill up just enough to get there (40L).

The solution is very intuitive:

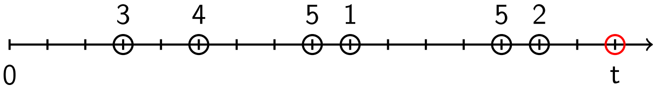
- At every station, fill up just as much as you need to get either to the destination or to a cheaper gas station.
- If this is not possible, fill up the tank completely.
- Example (each tick is 200km):



Travel Costs

The solution is very intuitive:

- At every station, fill up just as much as you need to get either to the destination or to a cheaper gas station.
- If this is not possible, fill up the tank completely.
- Example (each tick is 200km):



Done.

Permutation Pairs

Problem

Given a number n, find the maximum number of trailing zeroes that can be obtained by summing two permutations of n.

Caveat. The two permutations have to be valid representations of a number:

$$10020 \quad \text{vs.} \quad 00210$$

Permutation Pairs

Example:

$$n = 19270.$$

We can sum two permutations

$$\begin{array}{r} 12790 \\ +\ \ 97210 \\ \hline 110000 \end{array}$$

To get four trailing zeroes.

Permutation Pairs

Example:

$$n = 19270.$$

We can sum two permutations

$$\begin{array}{r} 12790 \\ +\ \ 97210 \\ \hline 110000 \end{array}$$

To get four trailing zeroes.

Is this optimal?

Permutation Pairs

How can we create trailing zeroes?

$$\begin{array}{r} 12790 \\ +\ 97210 \\ \hline 110000 \end{array}$$

- Each pair (0,0) can be used to create a zero by placing both zeroes at the right.
- If a pair of digits sums to 10, they can be used to create a zero.
- Afterwards, all pairs that sum to 9 give an additional zero.

Permutation Pairs

How can we create trailing zeroes?

$$\begin{array}{r} 12790 \\ +\ 97210 \\ \hline 110000 \end{array}$$

- Each pair (0,0) can be used to create a zero by placing both zeroes at the right.
- If a pair of digits sums to 10, they can be used to create a zero.
- Afterwards, all pairs that sum to 9 give an additional zero.
- And that is really all that we can do (think about it).

Permutation Pairs

Observation

If possible, it is always better to use two pairs (9,0) and (0,9) instead of the single pair (0,0).

Permutation Pairs

Observation

If possible, it is always better to use two pairs (9,0) and (0,9) instead of the single pair (0,0).

We still have to place pairs of the form (0,0) if

- there are more zeroes than nines, or
- using (9,0) and (0,9) would create an invalid permutation.

Permutation Pairs: Algorithm

- For $i = 0, 1, \ldots, 9$, let $m[i] = m'[i]$ count how often the digit i appears in n.

Permutation Pairs: Algorithm

- For $i = 0, 1, \ldots, 9$, let $m[i] = m'[i]$ count how often the digit i appears in n.
- For each pair (a, b) such that $a + b = 10$, do the following:
  - temporarily decrement $m[a]$ and $m'[b]$;
  - count the number of pairs that sum to 9:

$$x := \sum_{i=0}^{9} \min\{m[i], m'[9-i]\};$$

Permutation Pairs: Algorithm

- For $i = 0, 1, \ldots, 9$, let $m[i] = m'[i]$ count how often the digit i appears in n.
- For each pair (a, b) such that $a + b = 10$, do the following:
  - temporarily decrement $m[a]$ and $m'[b]$;
  - count the number of pairs that sum to 9:

$$x := \sum_{i=0}^{9} \min\{m[i], m'[9-i]\};$$

  - count how many zeroes are left:

$$y := m[0] - \min\{m[0], \max\{m[9], m'[9]\}\};$$

Permutation Pairs: Algorithm

- For $i = 0, 1, \ldots, 9$, let $m[i] = m'[i]$ count how often the digit i appears in n.
- For each pair (a, b) such that $a + b = 10$, do the following:
  - temporarily decrement $m[a]$ and $m'[b]$;
  - count the number of pairs that sum to 9:

$$x := \sum_{i=0}^{9} \min\{m[i], m'[9-i]\};$$

  - count how many zeroes are left:

$$y := m[0] - \min\{m[0], \max\{m[9], m'[9]\}\};$$

- output the maximal value of $1 + x + y$ obtained.

Permutation Pairs: Algorithm

Special cases:

- If there is no pair that sums to 10, simply output $m[0]$.

Permutation Pairs: Algorithm

Special cases:

- If there is no pair that sums to 10, simply output $m[0]$.
- Bordercasess:
  - For $n = 905$, the algorithm outputs 3:

$$\begin{array}{r} 905 \\ +\ 095 \\ \hline 1000 \end{array}$$

Permutation Pairs: Algorithm

Special cases:

- If there is no pair that sums to 10, simply output $m[0]$.
- Bordercasess:
  - For $n = 905$, the algorithm outputs 3:

$$\begin{array}{r} 905 \\ +\ 095 \\ \hline 1000 \end{array}$$

  - This can only happen if there is one five, any number of zeroes and at most as many nines as zeroes (and no other digits).

Permutation Pairs: Algorithm

Special cases:

- If there is no pair that sums to 10, simply output $m[0]$.
- Bordercasess:
  - For $n = 905$, the algorithm outputs 3:

$$\begin{array}{r} 905 \\ +\ 095 \\ \hline 1000 \end{array}$$

  - This can only happen if there is one five, any number of zeroes and at most as many nines as zeroes (and no other digits).
  - Check for this manually.

Geometry

- Two-dimensional Euclidean geometry
- To avoid (some) rounding errors: points with integer coordinates
- Lines, Vectors
- Polygons

Points

Storing a point:

```
struct Point {
  int x, y;
  Point (int x, int y) : x(x), y(y) { }
};
```

Points

Sometimes, we want to be able to sort points lexicographically:

```
struct Point {
  int x, y;
  Point (int x, int y) : x(x), y(y) { }

  bool operator<(Point other) const {
    if (x == other.x) {
      return (y < other.y);
    } else {
      return (x < other.x);
    }
  }
};
```

Points

We have the usual Euclidean metric:

```
double dist(Point p, Point q) {
  double dx = p.x-q.x;
  doubel dy = p.y-q.y;
  return sqrt(dx*dx + dy*dy);
}
```

Lines and Vectors

The triple (a, b, c) represents the line given by the equation

$$ax + by + c = 0\,:$$

```
struct Line {
  double a, b, c;
};
```

Lines and Vectors

The line determined by points p and q can be computed as follows:

```
struct Line {
  double a, b, c;

  Line(Point p, Point q) {
    if (p.x == q.x) {
      a = 1.0; b = 0; c = -p.x;
    } else {
      a = -(double)(p.y-q.y)/(p.x-q.x);
      b = 1.0;
      c = -(double)(a*p.x) - p.y;
    }
  }
};
```

Lines and Vectors

We also need the notion of a vector from p to q:

```
struct Vec {
  int x, y;

  Vec(Point p, Point q) {
    x = q.x-p.x;
    y = q.y-p.y;
  }
};
```

Lines and Vectors

The dot product of two vectors is defined as

$$\left\langle \begin{pmatrix} x \\ y \end{pmatrix}, \begin{pmatrix} x' \\ y' \end{pmatrix} \right\rangle := xx' + yy'.$$

The norm of $\vec{v}$ is

$$\|\vec{v}\| := \sqrt{\langle \vec{v}, \vec{v} \rangle}.$$

```
double dot(Vec v, Vec u) {
  return v.x*u.x + v.y*u.y;
}

double norm(Vec v) {
  return sqrt(dot(v,v));
}
```

Lines and Vectors

The dot product satisfies

$$\langle \vec{v}, \vec{u} \rangle = \|\vec{v}\| \|\vec{u}\| \cos(\alpha),$$

if α is the angle formed by $\vec{v}$ and $\vec{u}$.

```
double angle(Vec v, Vec u) {
  return acos(dot(v,u)/(norm(v)*norm(u)));
}
```

The Cross Product

An important operation is the cross product of vectors:

$$\begin{pmatrix} x \\ y \end{pmatrix} \times \begin{pmatrix} x' \\ y' \end{pmatrix} := \det \begin{bmatrix} x & x' \\ y & y' \end{bmatrix} = xy' - x'y.$$

The Cross Product

An important operation is the cross product of vectors:

$$\begin{pmatrix} x \\ y \end{pmatrix} \times \begin{pmatrix} x' \\ y' \end{pmatrix} := \det \begin{bmatrix} x & x' \\ y & y' \end{bmatrix} = xy' - x'y.$$

This operation is bilinear (linear in each argument), and satisfies

$$\vec{v} \times \vec{u} = -\vec{u} \times \vec{v}.$$

Since $\det AB = \det A \cdot \det B$, the cross product is invariant under rotation:

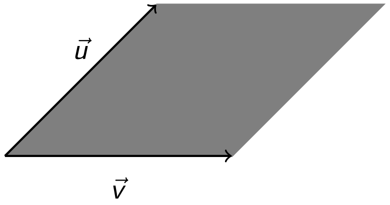
$$\begin{aligned}\det \begin{bmatrix} R\vec{v} & R\vec{u} \end{bmatrix} &= \det(R \cdot \begin{bmatrix} \vec{v} & \vec{u} \end{bmatrix}) \\ &= \det R \cdot \det \begin{bmatrix} \vec{v} & \vec{u} \end{bmatrix} \\ &= \det \begin{bmatrix} \vec{v} & \vec{u} \end{bmatrix}.\end{aligned}$$

So

$$R\vec{v} \times R\vec{u} = \vec{v} \times \vec{u}.$$

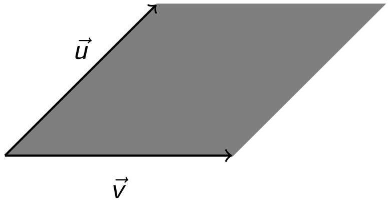
The Cross Product

Geometrically, $\vec{v} \times \vec{u}$ is the signed area of the parrallelogram spanned by $\vec{v}$ and $\vec{u}$:



The Cross Product

Geometrically, $\vec{v} \times \vec{u}$ is the signed area of the parrallelogram spanned by $\vec{v}$ and $\vec{u}$:



Proof. By rotational invariance, assume that $\vec{v} = (x, 0)$ and $\vec{u} = (x', y')$. Then

$$\vec{v} \times \vec{u} = xy'.$$

The Cross Product

Thus, $\vec{v} \times \vec{u}$ is

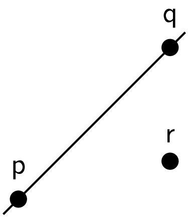
- negative if $\vec{v}$ is to the left of $\vec{u}$,
- zero if $\vec{v}$ and $\vec{u}$ are colinear, and
- positive if $\vec{v}$ is to the right of $\vec{u}$.

The CCW Test

Problem:

- let L be a line determined by points p and q,
- let r be a third point,
- determine if r lies on L, to the left of L, or to the right of L.

To the right: pqr in clockwise orientation

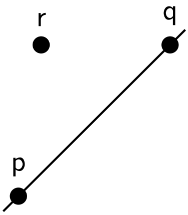


The CCW Test

Problem:

- let L be a line determined by points p and q,
- let r be a third point,
- determine if r lies on L, to the left of L, or to the right of L.

To the left: pqr in counter-clockwise orientation

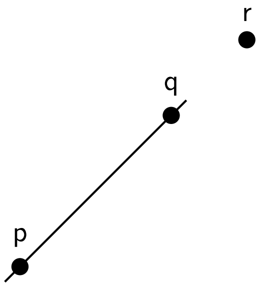


The CCW Test

Problem:

- let L be a line determined by points p and q,
- let r be a third point,
- determine if r lies on L, to the left of L, or to the right of L.

On L: pqr colinear



The CCW Test

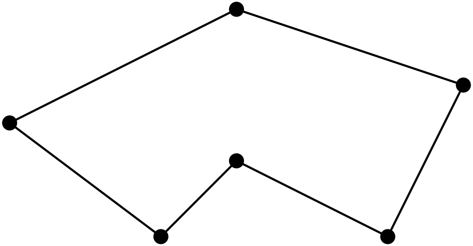
The solution is called the CCW (counter-clockwise) test:

```
int cross(Vec v, Vec u) {
  return v.x*u.y - u.x*v.y;
}

int ccw(Point p, Point q, Point r) {
  // negative: clockwise
  // 0: colinear
  // positive: counter-clockwise
  return cross(Vec(p,q), Vec(p,r));
}
```

Algorithms on Polygons

This is a polygon:



Algorithms on Polygons

We represent a polygon as a `vector<Point>` in which the vertices appear in counter-clockwise order:

```
vector<Point> p;
p.push_back(Point(0,0));
p.push_back(Point(1,0));
p.push_back(Point(1,1));
p.push_back(p[0]); // important: loop back
```

Algorithms on Polygons: Area

- Assume that the vertices are, in counter-clockwise order,

$$(x_1, y_1), \dots, (x_n, y_n).$$

- Then the area of the polygon is

$$A = \frac{1}{2}(x_1y_2 + x_2y_3 + \dots x_ny_1) - \frac{1}{2}(x_2y_1 + x_3y_2 + \dots x_1y_n).$$

Algorithms on Polygons: Area

- Assume that the vertices are, in counter-clockwise order,

$$(x_1, y_1), \ldots, (x_n, y_n).$$

- Then the area of the polygon is

$$A = \frac{1}{2}(x_1 y_2 + x_2 y_3 + \ldots x_n y_1) - \frac{1}{2}(x_2 y_1 + x_3 y_2 + \ldots x_1 y_n).$$

- Using the cross-product, this can be proved by induction.

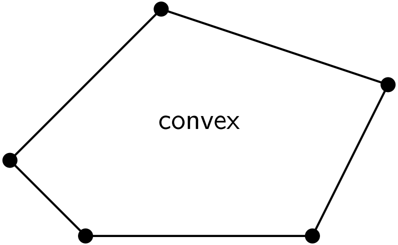
Algorithms on Polygons: Area

Thus:

```
double area(vector<Point> p) {
  int result = 0;
  for (int i = 0; i < p.size()-1; i++) {
    result += p[i].x*p[i+1].y - p[i+1].x*p[i].y;
  }
  return (double)result/2.0;
}
```

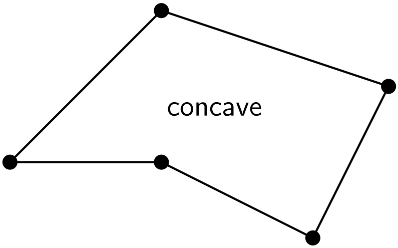
Algorithms on Polygons: Convexity

Problem: checking if a polygon is convex.



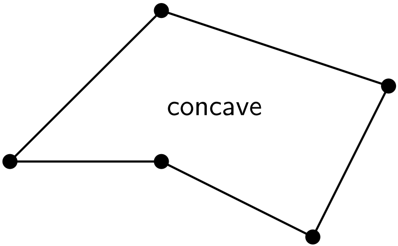
Algorithms on Polygons: Convexity

Problem: checking if a polygon is convex.



Algorithms on Polygons: Convexity

Problem: checking if a polygon is convex.



We can do this easily using the CCW test!

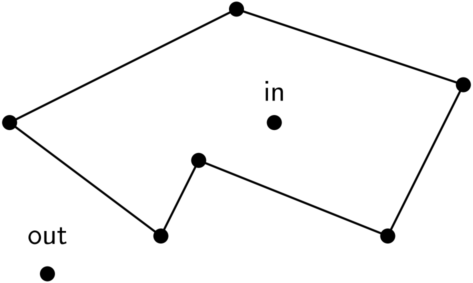
Algorithms on Polygons: Convexity

Checking if a polygon is convex:

```
// assumes counter-clockwise ordering
bool isConvex(vector<Point> p) {
  for (int i = 0; i < p.size()-1; i++) {
    int j = (i+2 == p.size()) ? 1 : (i+2);
    if (ccw(p[i], p[i+1], p[j]) < 0) return false;
  }
  return true;
}
```

Algorithms on Polygons: Point Containment

How to check if a given point is inside a polygon?



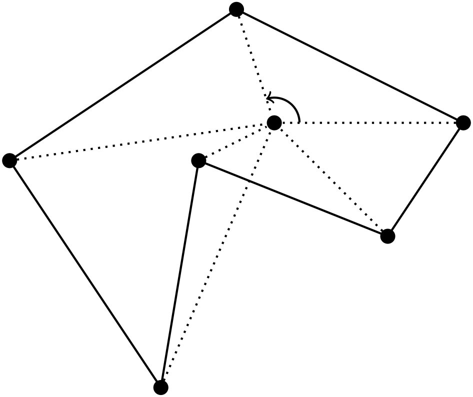
Algorithms on Polygons: Point Containment

If p is convex, then just check if p is to the left of eache edge of p:

```
// assumes counter-clockwise ordering
// only for convex polygons
bool contains(vector<Point> p, Point q) {
  for (int i = 0; i < p.size()-1; i++) {
    if (ccw(p[i], p[i+1], q) < 0) return false;
  }
  return true;
}
```

Algorithms on Polygons: Point Containment

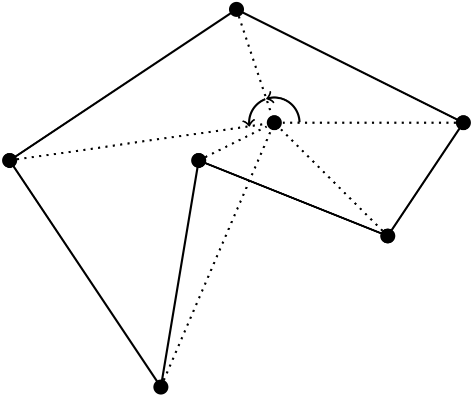
In general, q is inside if the angles formed with edges of p add up to 360°:



110°

Algorithms on Polygons: Point Containment

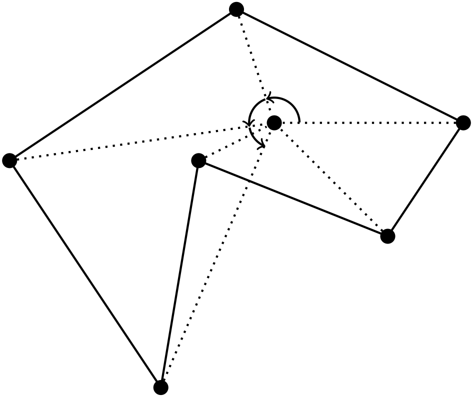
In general, q is inside if the angles formed with edges of p add up to 360°:



$$110^\circ + 80^\circ$$

Algorithms on Polygons: Point Containment

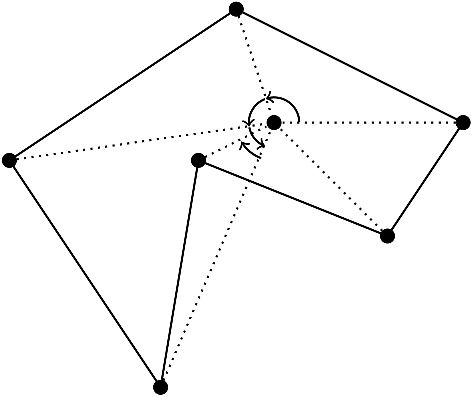
In general, q is inside if the angles formed with edges of p add up to 360°:



$$110^\circ + 80^\circ + 60^\circ$$

Algorithms on Polygons: Point Containment

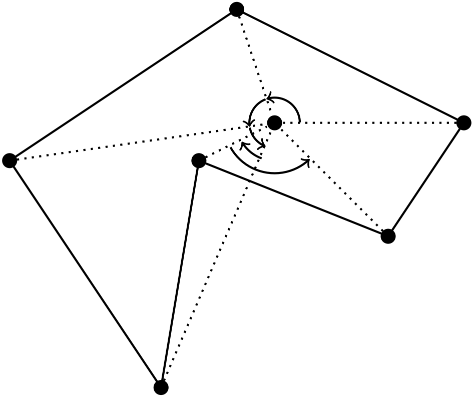
In general, q is inside if the angles formed with edges of p add up to 360°:



$$110^\circ + 80^\circ + 60^\circ - 40^\circ$$

Algorithms on Polygons: Point Containment

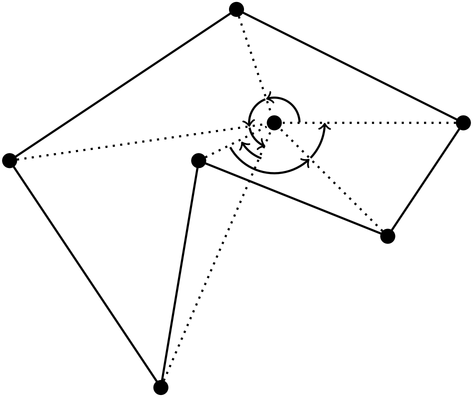
In general, q is inside if the angles formed with edges of p add up to 360°:



$$110^\circ + 80^\circ + 60^\circ - 40^\circ + 105^\circ$$

Algorithms on Polygons: Point Containment

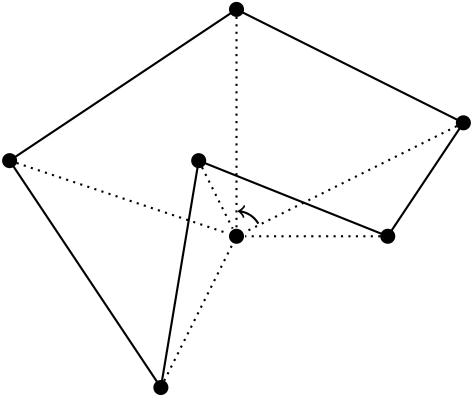
In general, q is inside if the angles formed with edges of p add up to 360°:



$$110^\circ + 80^\circ + 60^\circ - 40^\circ + 105^\circ + 45^\circ = 360^\circ$$

Algorithms on Polygons: Point Containment

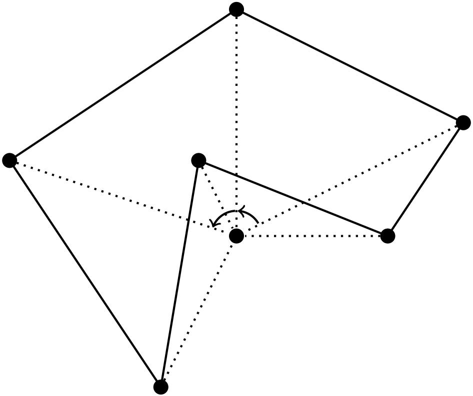
If q is outside, then the angles add up to zero:



60°

Algorithms on Polygons: Point Containment

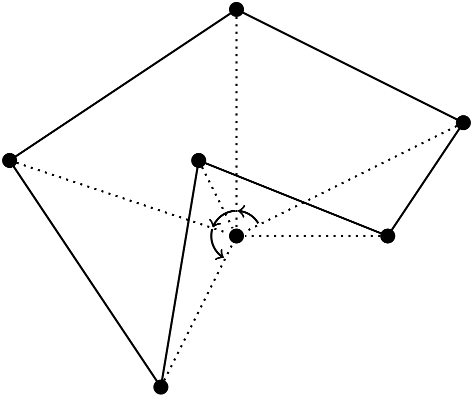
If q is outside, then the angles add up to zero:



$$60^\circ + 70^\circ$$

Algorithms on Polygons: Point Containment

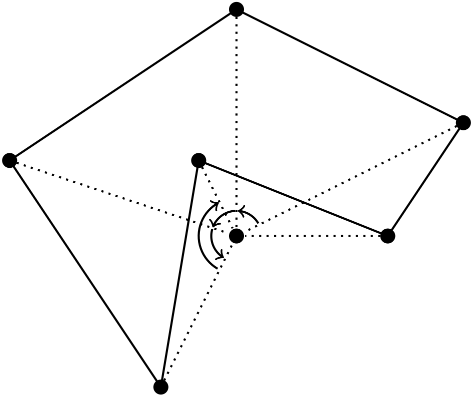
If q is outside, then the angles add up to zero:



$60^\circ + 70^\circ + 75^\circ$

Algorithms on Polygons: Point Containment

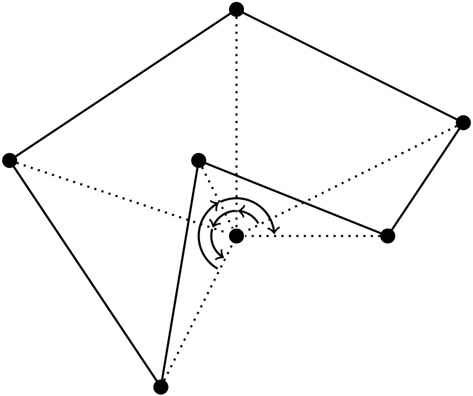
If q is outside, then the angles add up to zero:



$$60^\circ + 70^\circ + 75^\circ - 115^\circ$$

Algorithms on Polygons: Point Containment

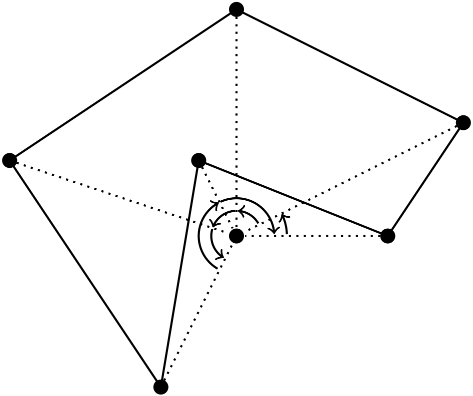
If q is outside, then the angles add up to zero:



$$60^\circ + 70^\circ + 75^\circ - 115^\circ - 115^\circ$$

Algorithms on Polygons: Point Containment

If q is outside, then the angles add up to zero:



$$60^\circ + 70^\circ + 75^\circ - 115^\circ - 115^\circ + 25^\circ = 0^\circ$$

Algorithms on Polygons: Point Containment

```
// assumes counter-clockwise ordering
bool contains(vector<Point> p, Point q) {
  double sum = 0.0;
  for (int i = 0; i < p.size()-1; i++) {
    if (ccw(q, p[i], p[i+1]) >= 0) {
      sum += angle(Vec(q,p[i]), Vec(q,p[i+1]));
    } else {
      sum -= angle(Vec(q,p[i]), Vec(q,p[i+1]));
    }
  }
  return (sum > 3);
}
```