Exercise 7.2.2.1: A line can also be described with this mathematical equation: $y = mx+c$ where m is the 'gradient'/'slope' of the line and c is the 'y-intercept' constant.
Which form is better ($ax + by + c = 0$ or the slope-intercept form $y = mx + c$)? Why?

Exercise 7.2.2.2: Compute line equation that pass through two points (2, 2) and (4, 3)!

Exercise 7.2.2.3: Compute line equation that pass through two points (2, 2) and (2, 4)!

Exercise 7.2.2.4: Suppose we insist to use the other line equation: $y = mx + c$. Show how to compute the required line equation given two points that pass through that line! Try on two points (2, 2) and (2, 4) as in **Exercise 7.2.2.3**. Do you encounter any problem?

Exercise 7.2.2.5: We can also compute the line equation if we are given *one* point and the gradient/slope of that line. Show how to compute line equation given a point and gradient!

Exercise 7.2.2.6: Translate a point c (3, 2) according to a vector ab defined by two points: a (2, 2) and b (4, 3). What is the new coordinate of the point?

Exercise 7.2.2.7: Same as **Exercise 7.2.2.6** above, but now the magnitude of vector ab is reduced by *half*. What is the new coordinate of the point?

Exercise 7.2.2.8: Same as **Exercise 7.2.2.6** above, then rotate the resulting point by 90 degrees counter clockwise around origin. What is the new coordinate of the point?

Exercise 7.2.2.9: Rotate a point c (3, 2) by 90 degrees counter clockwise around origin, then translate the resulting point according to a vector ab. Vector ab is the same as in **Exercise 7.2.2.6** above. What is the new coordinate of the point? Is the result similar with the previous **Exercise 7.2.2.8** above? What can we learn from this phenomenon?

Exercise 7.2.2.10: Rotate a point c (3, 2) by 90 degrees counter clockwise but around point p (2, 1) (note that point p is *not* the origin). Hint: You need to translate the point.

Exercise 7.2.2.11: We can compute the location of point c in line l that is closest to point p by finding the other line l' that is perpendicular with line l and pass through point p. The closest point c is the intersection point between line l and l'. Now, how to obtain a line perpendicular to l? Are there special cases that we have to be careful with?

Exercise 7.2.2.12: Given a point p and a line l (described by two points a and b), show how to compute the location of a reflection point r of point p when mirrored against line l.

Exercise 7.2.2.13: Given three points: a (2, 2), o (2, 4), and b (4, 3), compute the angle aob in degrees!

Exercise 7.2.2.14: Determine if point r (35, 30) is on the left side of, collinear with, or is on the right side of a line that passes through two points p (3, 7) and q (11, 13).

7.2.3 2D Objects: Circles

1. **Circle** centered at coordinate (a, b) in a 2D Euclidean space with **radius** r is the set of all points (x, y) such that $(x - a)^2 + (y - b)^2 = r^2$.

2. To check if a point is inside, outside, or exactly on the border of a circle, we can use the following function. Modify this function a bit for the floating point version.

```
int insideCircle(point_i p, point_i c, int r) { // all integer version
  int dx = p.x - c.x, dy = p.y - c.y;
  int Euc = dx * dx + dy * dy, rSq = r * r;              // all integer
  return Euc < rSq ? 0 : Euc == rSq ? 1 : 2; } //inside/border/outside
```

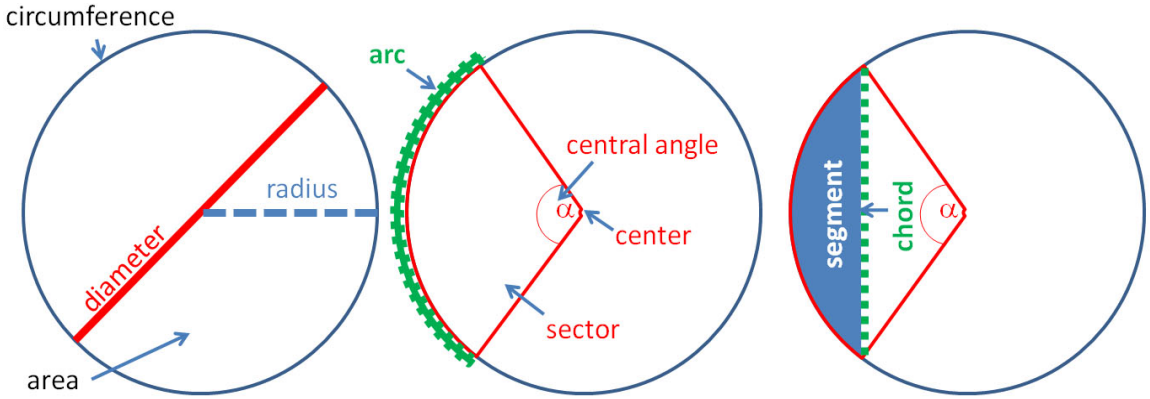


Figure 7.3: Circles

3. The constant **Pi** (π) is the ratio of *any* circle's circumference to its diameter. To avoid precision error, the safest value for programming contest if this constant π is not defined in the problem description is `pi = acos(-1.0)` or `pi = 2 * acos(0.0)`.

4. A circle with radius r has **diameter** $d = 2 \times r$ and **circumference** (or **perimeter**) $c = 2 \times \pi \times r$.

5. A circle with radius r has **area** $A = \pi \times r^2$

6. **Arc** of a circle is defined as a connected section of the circumference c of the circle. Given the central angle α (angle with vertex at the circle's center, see Figure 7.3—middle) in degrees, we can compute the length of the corresponding arc as $\frac{\alpha}{360.0} \times c$.

7. **Chord** of a circle is defined as a line segment whose endpoints lie on the circle[15]. A circle with radius r and a central angle α in degrees (see Figure 7.3—right) has the corresponding chord with length $sqrt(2 \times r^2 \times (1 - cos(\alpha)))$. This can be derived from the **Law of Cosines**—see the explanation of this law in the discussion about Triangles later. Another way to compute the length of chord given r and α is to use Trigonometry: $2 \times r \times sin(\alpha/2)$. Trigonometry is also discussed below.

8. **Sector** of a circle is defined as a region of the circle enclosed by two radius and an arc lying between the two radius. A circle with area A and a central angle α (in degrees)—see Figure 7.3, middle—has the corresponding sector area $\frac{\alpha}{360.0} \times A$.

9. **Segment** of a circle is defined as a region of the circle enclosed by a chord and an arc lying between the chord's endpoints (see Figure 7.3—right). The area of a segment can be found by subtracting the area of the corresponding sector from the area of an isosceles triangle with sides: r, r, and chord-length.

[15]Diameter is the longest chord in a circle.

10. Given 2 points on the circle ($p1$ and $p2$) and radius r of the corresponding circle, we can determine the location of the centers ($c1$ and $c2$) of the two possible circles (see Figure 7.4). The code is shown in **Exercise 7.2.3.1** below.

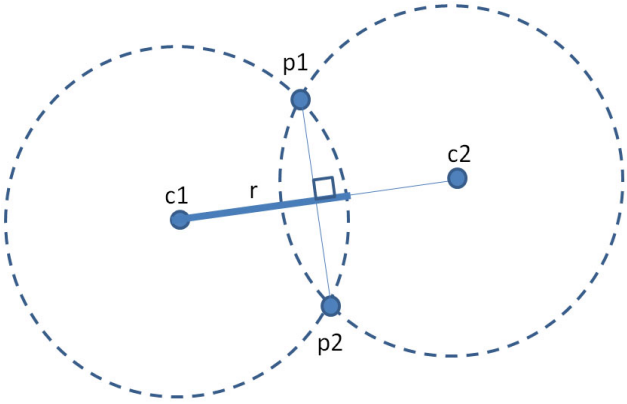


Figure 7.4: Circle Through 2 Points and Radius

Source code: `ch7_02_circles.cpp/java`

Exercise 7.2.3.1: Explain what is computed by the code below!

```
bool circle2PtsRad(point p1, point p2, double r, point &c) {
  double d2 = (p1.x - p2.x) * (p1.x - p2.x) +
              (p1.y - p2.y) * (p1.y - p2.y);
  double det = r * r / d2 - 0.25;
  if (det < 0.0) return false;
  double h = sqrt(det);
  c.x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
  c.y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
  return true; }          // to get the other center, reverse p1 and p2
```

7.2.4 2D Objects: Triangles

1. **Triangle** (three angles) is a polygon with three vertices and three edges.
   There are several types of triangles:
   a. **Equilateral**: Three equal-length edges and all inside (interior) angles are 60 degrees;
   b. **Isosceles**: Two edges have the same length and two interior angles are the same.
   c. **Scalene**: All edges have different length;
   d. **Right**: *One* of its interior angle is 90 degrees (or a **right angle**).

2. A triangle with base b and height h has **area** $A = 0.5 \times b \times h$.

3. A triangle with three sides: a, b, c has **perimeter** $p = a + b + c$ and **semi-perimeter** $s = 0.5 \times p$.

4. A triangle with 3 sides: a, b, c and semi-perimeter s has area $A = sqrt(s \times (s - a) \times (s - b) \times (s - c))$. This formula is called the **Heron's Formula**.

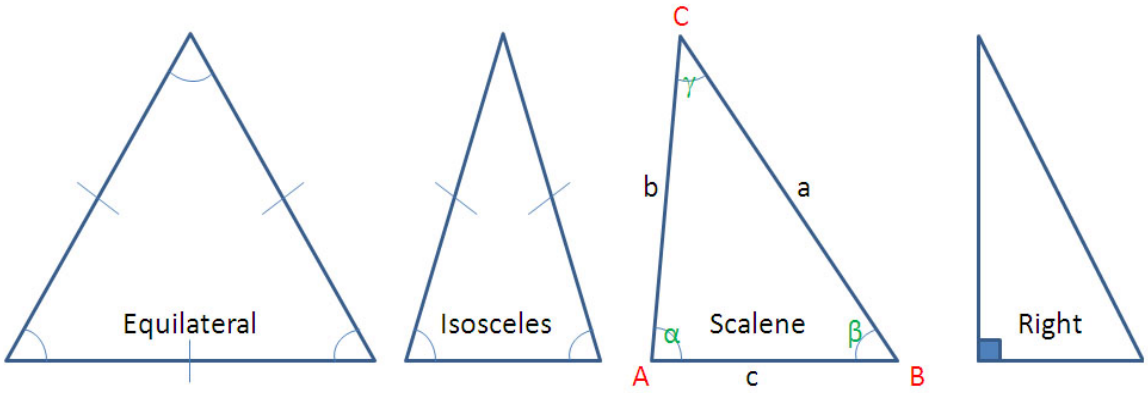


Figure 7.5: Triangles

5. A triangle with area A and semi-perimeter s has an **inscribed circle (incircle)** with radius $r = A/s$.

```
double rInCircle(double ab, double bc, double ca) {
  return area(ab, bc, ca) / (0.5 * perimeter(ab, bc, ca)); }

double rInCircle(point a, point b, point c) {
  return rInCircle(dist(a, b), dist(b, c), dist(c, a)); }
```

6. The center of incircle is the meeting point between the triangle's *angle bisectors* (see Figure 7.6—left). We can get the center if we have two angle bisectors and find their intersection point. The implementation is shown below:

```
// assumption: the required points/lines functions have been written
// returns 1 if there is an inCircle center, returns 0 otherwise
// if this function returns 1, ctr will be the inCircle center
// and r is the same as rInCircle
int inCircle(point p1, point p2, point p3, point &ctr, double &r) {
  r = rInCircle(p1, p2, p3);
  if (fabs(r) < EPS) return 0;                    // no inCircle center

  line l1, l2;                      // compute these two angle bisectors
  double ratio = dist(p1, p2) / dist(p1, p3);
  point p = translate(p2, scale(toVec(p2, p3), ratio / (1 + ratio)));
  pointsToLine(p1, p, l1);

  ratio = dist(p2, p1) / dist(p2, p3);
  p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));
  pointsToLine(p2, p, l2);

  areIntersect(l1, l2, ctr);           // get their intersection point
  return 1; }
```

7. A triangle with 3 sides: a, b, c and area A has an **circumscribed circle (circumcircle)** with radius $R = a \times b \times c/(4 \times A)$.

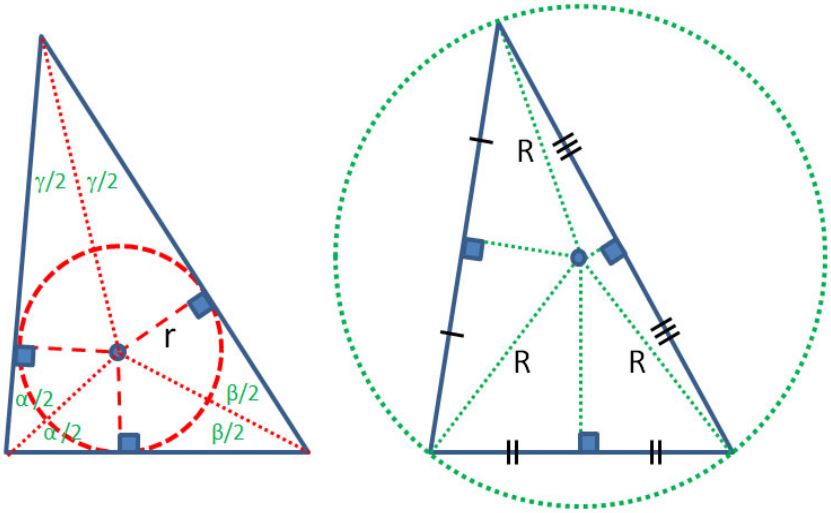


Figure 7.6: Incircle and Circumcircle of a Triangle

```
double rCircumCircle(double ab, double bc, double ca) {
  return ab * bc * ca / (4.0 * area(ab, bc, ca)); }

double rCircumCircle(point a, point b, point c) {
  return rCircumCircle(dist(a, b), dist(b, c), dist(c, a)); }
```

8. The center of circumcircle is the meeting point between the triangle's *perpendicular bisectors* (see Figure 7.6—right).

9. To check if three line segments of length a, b and c can form a triangle, we can simply check these *triangle inequalities*: $(a + b > c)$ && $(a + c > b)$ && $(b + c > a)$.
   If the result is false, then the three line segments cannot form a triangle.
   If the three lengths are sorted, with a being the smallest and c the largest, then we can simplify the check to just $(a + b > c)$.

10. When we study triangle, we should not forget **Trigonometry**—a study about the relationships between triangle sides and the angles between sides.

    In Trigonometry, the **Law of Cosines** (a.k.a. the **Cosine Formula** or the **Cosine Rule**) is a statement about a general triangle that relates the lengths of its sides to the cosine of one of its angles. See the scalene (middle) triangle in Figure 7.5. With the notations described there, we have: $c^2 = a^2 + b^2 - 2 \times a \times b \times cos(\gamma)$, or $\gamma = acos(\frac{a^2+b^2-c^2}{2\times a\times b})$. The formula for the other two angles α and β are similarly defined.

11. In Trigonometry, the **Law of Sines** (a.k.a. the **Sine Formula** or the **Sine Rule**) is an equation relating the lengths of the sides of an arbitrary triangle to the sines of its angle. See the scalene (middle) triangle in Figure 7.5. With the notations described there and R is the radius of its circumcircle, we have: $\frac{a}{\sin(\alpha)} = \frac{b}{\sin(\beta)} = \frac{c}{\sin(\gamma)} = 2R$.

12. The **Pythagorean Theorem** specializes the Law of Cosines. This theorem only applies to right triangles. If the angle γ is a right angle (of measure 90° or $\pi/2$ radians), then $cos(\gamma) = 0$, and thus the Law of Cosines reduces to: $c^2 = a^2 + b^2$. Pythagorean theorem is used in finding the Euclidean distance between two points shown earlier.

13. The **Pythagorean Triple** is a triple with three positive integers a, b, and c—commonly written as (a, b, c)—such that $a^2 + b^2 = c^2$. A well-known example is $(3, 4, 5)$. If (a, b, c) is a Pythagorean triple, then so is (ka, kb, kc) for any positive integer k. A Pythagorean Triple describes the integer lengths of the three sides of a Right Triangle.

Source code: `ch7_03_triangles.cpp/java`

Exercise 7.2.4.1: Let a, b, and c of a triangle be 2^{18}, 2^{18}, and 2^{18}. Can we compute the area of this triangle with Heron's formula as shown in point 4 above without experiencing overflow (assuming that we use 64-bit integers)? What should we do to avoid this issue?

Exercise 7.2.4.2*: Implement the code to find the center of the circumCircle of three points a, b, and c. The function structure is similar as function `inCircle` shown in this section.

Exercise 7.2.4.3*: Implement another code to check if a point d is inside the circumCircle of three points a, b, and c.

7.2.5 2D Objects: Quadrilaterals

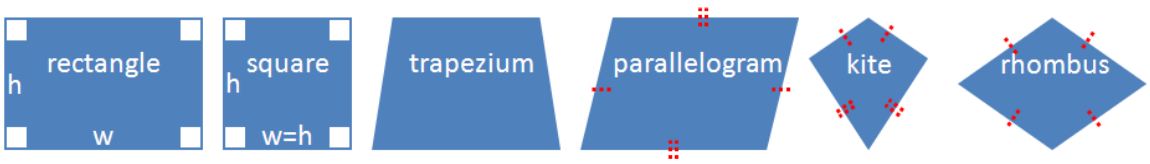


Figure 7.7: Quadrilaterals

1. **Quadrilateral** or **Quadrangle** is a polygon with four edges (and four vertices). The term 'polygon' itself is described in more details below (Section 7.3). Figure 7.7 shows a few examples of Quadrilateral objects.
2. **Rectangle** is a polygon with four edges, four vertices, and four right angles.
3. A rectangle with width w and height h has **area** $A = w \times h$ and **perimeter** $p = 2 \times (w + h)$.
4. **Square** is a special case of a rectangle where $w = h$.
5. **Trapezium** is a polygon with four edges, four vertices, and one pair of parallel edges. If the two non-parallel sides have the same length, we have an **Isosceles Trapezium**.
6. A trapezium with a pair of parallel edges of lengths $w1$ and $w2$; and a height h between both parallel edges has area $A = 0.5 \times (w1 + w2) \times h$.
7. **Parallelogram** is a polygon with four edges and four vertices. Moreover, the opposite sides must be parallel.
8. **Kite** is a quadrilateral which has two pairs of sides of the same length which are adjacent to each other. The area of a kite is $diagonal_1 \times diagonal_2/2$.
9. **Rhombus** is a special parallelogram where every side has equal length. It is also a special case of kite where every side has equal length.

Remarks about 3D Objects

Programming contest problems involving 3D objects are rare. But when such a problem does appear in a problem set, it can be one of the hardest. In the list of programming exercises below, we include an initial list of problems involving 3D objects.

Programming Exercises related to Basic Geometry:

- Points and Lines:
  1. UVa 00152 - Tree's a Crowd (sort the 3D points first)
  2. UVa 00191 - Intersection (line segment intersection)
  3. UVa 00378 - Intersecting Lines (use `areParallel`, `areSame`, `areIntersect`)
  4. UVa 00587 - There's treasure everywhere (Euclidean distance `dist`)
  5. UVa 00833 - Water Falls (recursive check, use the `ccw` tests)
  6. UVa 00837 - Light and Transparencies (line segments, sort x-coords first)
  7. **UVa 00920 - Sunny Mountains *** (Euclidean distance `dist`)
  8. UVa 01249 - Euclid (LA 4601, Southeast USA Regional 2009, vector)
  9. UVa 10242 - Fourth Point (`toVector`; `translate` points w.r.t that vector)
  10. *UVa 10250 - The Other Two Trees* (vector, rotation)
  11. **UVa 10263 - Railway *** (use `distToLineSegment`)
  12. UVa 10357 - Playball (Euclidean distance `dist`, simple Physics simulation)
  13. UVa 10466 - How Far? (Euclidean distance `dist`)
  14. UVa 10585 - Center of symmetry (sort the points)
  15. *UVa 10832 - Yoyodyne Propulsion ...* (3D Euclidean distance; simulation)
  16. UVa 10865 - Brownie Points (points and quadrants, simple)
  17. UVa 10902 - Pick-up sticks (line segment intersection)
  18. **UVa 10927 - Bright Lights *** (sort points by gradient, Euclidean distance)
  19. UVa 11068 - An Easy Task (simple 2 linear equations with 2 unknowns)
  20. UVa 11343 - Isolated Segments (line segment intersection)
  21. UVa 11505 - Logo (Euclidean distance `dist`)
  22. *UVa 11519 - Logo 2* (vectors and angles)
  23. *UVa 11894 - Genius MJ* (about rotating and translating points)
- Circles (only)
  1. *UVa 01388 - Graveyard* (divide the circle into n sectors first and then into $(n + m)$ sectors)
  2. **UVa 10005 - Packing polygons *** (complete search; use `circle2PtsRad` discussed in Chapter 7)
  3. UVa 10136 - Chocolate Chip Cookies (similar to UVa 10005)
  4. UVa 10180 - Rope Crisis in Ropeland (closest point from AB to origin; arc)
  5. UVa 10209 - Is This Integration? (square, arcs, similar to UVa 10589)
  6. UVa 10221 - Satellites (finding arc and chord length of a circle)
  7. *UVa 10283 - The Kissing Circles* (derive the formula)
  8. UVa 10432 - Polygon Inside A Circle (area of n-sided reg-polygon in circle)
  9. UVa 10451 - Ancient ... (inner/outer circle of n-sided reg polygon)
  10. UVa 10573 - Geometry Paradox (there is no 'impossible' case)
  11. **UVa 10589 - Area *** (check if point is inside intersection of 4 circles)

12. **UVa 10678 - The Grazing Cows *** (area of an *ellipse*, generalization of the formula for area of a circle)
13. *UVa 12578 - 10:6:2* (area of rectangle and circle)

- Triangles (plus Circles)
  1. UVa 00121 - Pipe Fitters (use Pythagorean theorem; grid)
  2. UVa 00143 - Orchard Trees (count integer points in triangle; precision issue)
  3. UVa 00190 - Circle Through Three ... (triangle's circumcircle)
  4. UVa 00375 - Inscribed Circles and ... (triangle's incircle**s**!)
  5. UVa 00438 - The Circumference of ... (triangle's circumcircle)
  6. UVa 10195 - The Knights Of The ... (triangle's incircle, Heron's formula)
  7. UVa 10210 - Romeo & Juliet (basic trigonometry)
  8. UVa 10286 - The Trouble with a ... (Law of Sines)
  9. UVa 10347 - Medians (given 3 medians of a triangle, find its area)
  10. UVa 10387 - Billiard (expanding surface, *trigonometry*)
  11. UVa 10522 - Height to Area (derive the formula, uses Heron's formula)
  12. **UVa 10577 - Bounding box *** (get center+radius of outer circle from 3 points, get all vertices, get the min-x/max-x/min-y/max-y of the polygon)
  13. *UVa 10792 - The Laurel-Hardy Story* (derive the trigonometry formulas)
  14. UVa 10991 - Region (Heron's formula, Law of Cosines, area of sector)
  15. **UVa 11152 - Colourful ... *** (triangle's (in/circum)circle; Heron's formula)
  16. *UVa 11164 - Kingdom Division* (use Triangle properties)
  17. *UVa 11281 - Triangular Pegs in ...* (the min bounding circle of a non obtuse triangle is its circumcircle; if the triangle is obtuse, the the radii of the min bounding circle is the largest side of the triangle)
  18. *UVa 11326 - Laser Pointer* (trigonometry, tangent, reflection trick)
  19. *UVa 11437 - Triangle Fun* (hint: $\frac{1}{7}$)
  20. UVa 11479 - Is this the easiest problem? (property check)
  21. UVa 11579 - Triangle Trouble (sort; greedily check if three successive sides satisfy triangle inequality and if it is the largest triangle found so far)
  22. UVa 11854 - Egypt (Pythagorean theorem/triple)
  23. **UVa 11909 - Soya Milk *** (Law of Sines (or tangent); two possible cases!)
  24. UVa 11936 - The Lazy Lumberjacks (see if 3 sides form a valid triangle)
- Quadrilaterals
  1. UVa 00155 - All Squares (recursive counting)
  2. **UVa 00460 - Overlapping Rectangles *** (rectangle-rectangle intersection)
  3. UVa 00476 - Points in Figures: ... (similar to UVa 477 and 478)
  4. UVa 00477 - Points in Figures: ... (similar to UVa 476 and 478)
  5. **UVa 11207 - The Easiest Way *** (cutting rectangle into 4-equal-sized squares)
  6. UVa 11345 - Rectangles (rectangle-rectangle intersection)
  7. UVa 11455 - Behold My Quadrangle (property check)
  8. UVa 11639 - Guard the Land (rectangle-rectangle intersection, use flag array)
  9. *UVa 11800 - Determine the Shape* (use `next_permutation` to help you try all possible $4! = 24$ permutations of 4 points; check if they can satisfy square, rectangle, rhombus, parallelogram, trapezium, in that order)
  10. **UVa 11834 - Elevator *** (packing two circles in a rectangle)
  11. *UVa 12256 - Making Quadrilaterals* (LA 5001, KualaLumpur 10, start with three sides of 1, 1, 1, then the fourth side onwards must be the sum of the previous three to make a line; repeat until we reach the n-th side)

- 3D Objects
  1. **UVa 00737 - Gleaming the Cubes *** (cube and cube intersection)
  2. **UVa 00815 - Flooded *** (volume, greedy, sort by height, simulation)
  3. **UVa 10297 - Beavergnaw *** (cones, cylinders, volumes)

Profile of Algorithm Inventor

Pythagoras of Samos ($\approx$ 500 BC) was a Greek mathematician and philosopher born on the island of Samos. He is best known for the Pythagorean theorem involving right triangle.

Euclid of Alexandria ($\approx$ 300 BC) was a Greek mathematician, the 'Father of Geometry'. He was from the city of Alexandria. His most influential work in mathematics (especially geometry) is the 'Elements'. In the 'Elements', Euclid deduced the principles of what is now called Euclidean geometry from a small set of axioms.

Heron of Alexandria ($\approx$ 10-70 AD) was an ancient Greek mathematician from the city of Alexandria, Roman Egypt—the same city as Euclid. His name is closely associated with his formula for finding the area of a triangle from its side lengths.

Ronald Lewis Graham (born 1935) is an American mathematician. In 1972, he invented the Graham's scan algorithm for finding convex hull of a finite set of points in the plane. There are now many other algorithm variants and improvements for finding convex hull.

7.3 Algorithm on Polygon with Libraries

Polygon is a plane figure that is bounded by a closed path (path that starts and ends at the same vertex) composed of a finite sequence of straight line segments. These segments are called edges or sides. The point where two edges meet is the polygon's vertex or corner. Polygon is a source of many (computational) geometry problems as it allows the problem author to present more realistic objects than the ones discussed in Section 7.2.

7.3.1 Polygon Representation

The standard way to represent a polygon is to simply enumerate the vertices of the polygon in either clockwise or counter clockwise order, with the first vertex being equal to the last vertex (some of the functions mentioned later in this section require this arrangement, see **Exercise 7.3.4.1***). In this book, our default vertex ordering is counter clockwise. The resulting polygon after executing the code below is shown in Figure 7.8—right.

```
// 6 points, entered in counter clockwise order, 0-based indexing
vector<point> P;
P.push_back(point(1, 1));  // P0
P.push_back(point(3, 3));  // P1
P.push_back(point(9, 1));  // P2
P.push_back(point(12, 4)); // P3
P.push_back(point(9, 7));  // P4
P.push_back(point(1, 7));  // P5
P.push_back(P[0]); // important: loop back
```

7.3.2 Perimeter of a Polygon

The perimeter of a polygon (either convex or concave) with n vertices given in some order (either clockwise or counter-clockwise) can be computed via this simple function below.

```
// returns the perimeter, which is the sum of Euclidian distances
// of consecutive line segments (polygon edges)
double perimeter(const vector<point> &P) {
  double result = 0.0;
  for (int i = 0; i < (int)P.size()-1; i++)  // remember that P[0] = P[n-1]
    result += dist(P[i], P[i+1]);
  return result; }
```

7.3.3 Area of a Polygon

The signed area A of (either convex or concave) polygon with n vertices given in some order (either clockwise or counter-clockwise) can be found by computing the determinant of the matrix as shown below. This formula can be easily written into the library code.

$$A = \frac{1}{2} \times \begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ \dots & \dots \\ x_{n-1} & y_{n-1} \end{bmatrix} = \frac{1}{2} \times (x_0 \times y_1 + x_1 \times y_2 + \dots + x_{n-1} \times y_0 - x_1 \times y_0 - x_2 \times y_1 - \dots - x_0 \times y_{n-1})$$

```
// returns the area, which is half the determinant
double area(const vector<point> &P) {
  double result = 0.0, x1, y1, x2, y2;
  for (int i = 0; i < (int)P.size()-1; i++) {
    x1 = P[i].x; x2 = P[i+1].x;
    y1 = P[i].y; y2 = P[i+1].y;
    result += (x1 * y2 - x2 * y1);
  }
  return fabs(result) / 2.0; }
```

7.3.4 Checking if a Polygon is Convex

A polygon is said to be **Convex** if any line segment drawn inside the polygon does not intersect any edge of the polygon. Otherwise, the polygon is called **Concave**.

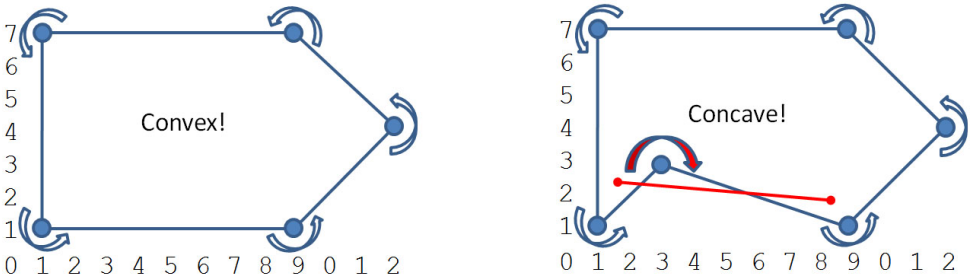


Figure 7.8: Left: Convex Polygon, Right: Concave Polygon

However, to test if a polygon is convex, there is an easier computational approach than "trying to check if all line segments can be drawn inside the polygon". We can simply check whether all three consecutive vertices of the polygon form the same turns (all left turns/ccw if the vertices are listed in counter clockwise order or all right turn/cw if the vertices are listed in clockwise order). If we can find at least one triple where this is false, then the polygon is concave (see Figure 7.8).

```
bool isConvex(const vector<point> &P) {        // returns true if all three
  int sz = (int)P.size();  // consecutive vertices of P form the same turns
  if (sz <= 3) return false;   // a point/sz=2 or a line/sz=3 is not convex
  bool isLeft = ccw(P[0], P[1], P[2]);               // remember one result
  for (int i = 1; i < sz-1; i++)            // then compare with the others
    if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)
      return false;            // different sign -> this polygon is concave
  return true; }                                    // this polygon is convex
```

Exercise 7.3.4.1*: Which part of the code above that you should modify to accept collinear points? Example: Polygon {(0,0), (2,0), (4,0), (2,2), (0,0)} should be treated as convex.

Exercise 7.3.4.2*: If the first vertex is not repeated as the last vertex, will the function `perimeter`, `area`, and `isConvex` presented as above work correctly?

7.3.5 Checking if a Point is Inside a Polygon

Another common test performed on a polygon P is to check if a point pt is inside or outside polygon P. The following function that implements 'winding number algorithm' allows such check for *either* convex or concave polygon. It works by computing the sum of angles between three points: $\{P[i], pt, P[i+1]\}$ where ($P[i]$-$P[i+1]$) are consecutive sides of polygon P, taking care of left turns (add the angle) and right turns (subtract the angle) respectively. If the final sum is 2π (360 degrees), then pt is inside polygon P (see Figure 7.9).

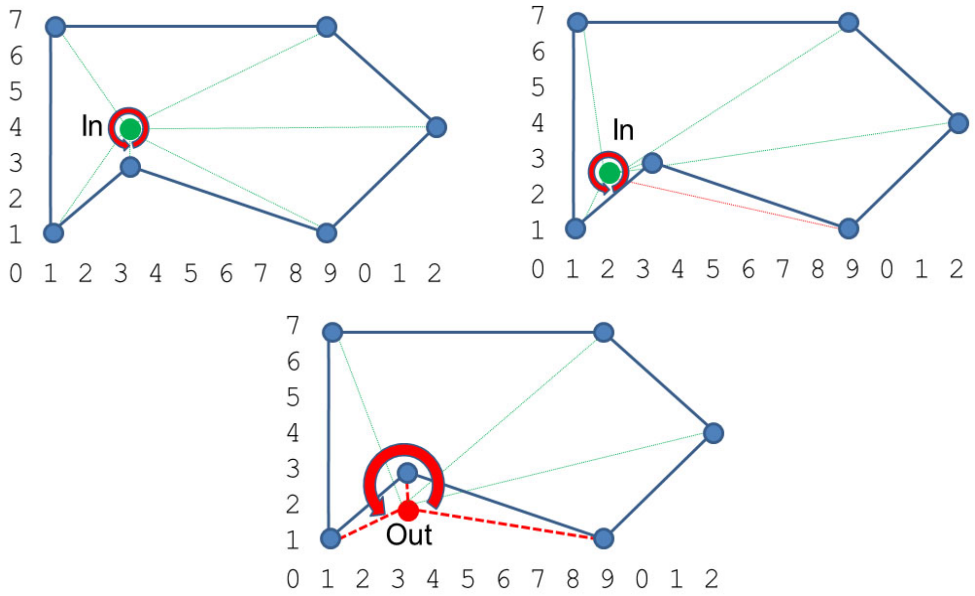


Figure 7.9: Top Left: inside, Top Right: also inside, Bottom: outside

```
// returns true if point p is in either convex/concave polygon P
bool inPolygon(point pt, const vector<point> &P) {
  if ((int)P.size() == 0) return false;
  double sum = 0;    // assume the first vertex is equal to the last vertex
  for (int i = 0; i < (int)P.size()-1; i++) {
    if (ccw(pt, P[i], P[i+1]))
         sum += angle(P[i], pt, P[i+1]);                    // left turn/ccw
    else sum -= angle(P[i], pt, P[i+1]); }                  // right turn/cw
  return fabs(fabs(sum) - 2*PI) < EPS; }
```

Exercise 7.9.1*: What happen to the `inPolygon` routine if point `pt` is on one of the edge of polygon `P`, e.g. `pt = P[0]` or `pt` is the mid-point between `P[0]` and `P[1]`, etc? What should be done to address that situation?

Exercise 7.9.2*: Discuss the pros and the cons of the following alternative methods for testing if a point is inside a polygon:

1. Triangulate a convex polygon into triangles and check if the sum of triangle areas equal to the area of the convex polygon.
2. Ray casting algorithm: We draw a ray from the point to any fixed direction so that the ray intersects the edge(s) of the polygon. If there are odd/even number of intersections, the point is inside/outside, respectively.

7.3.6 Cutting Polygon with a Straight Line

Another interesting thing that we can do with a *convex* polygon (see **Exercise 7.3.6.2*** for concave polygon) is to cut it into two convex sub-polygons with a straight line defined with two points a and b. See some programming exercises listed below that use this function.

The basic idea of the following `cutPolygon` routine is to iterate through the vertices of the original polygon Q one by one. If line ab and polygon vertex v form a left turn (which implies that v is on the left side of the line ab), we put v inside the new polygon P. Once we find a polygon edge that intersects with the line ab, we use that intersection point as part of the new polygon P (see Figure 7.10—left, point 'C'). We then skip the next few vertices of Q that are located on the right side of line ab. Sooner or later, we will revisit another polygon edge that intersect with line ab again (see Figure 7.10—left, point 'D' which happens to be one of the original vertex of polygon Q). We continue appending vertices of Q into P again because we are now on the left side of line ab again. We stop when we have returned to the starting vertex and returns the resulting polygon P (see Figure 7.10—right).

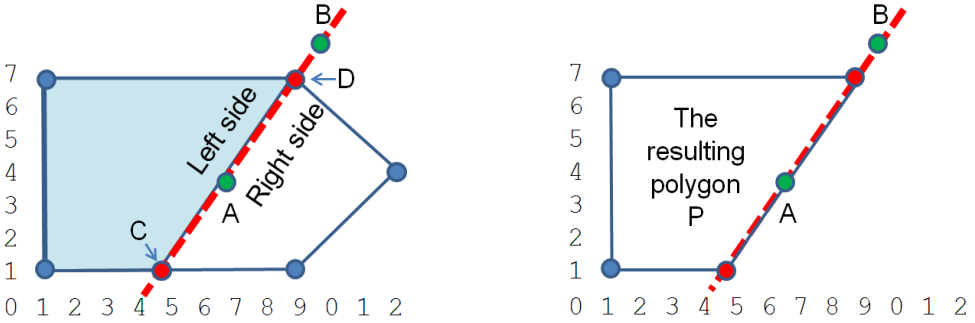


Figure 7.10: Left: Before Cut, Right: After Cut

```
// line segment p-q intersect with line A-B.
point lineIntersectSeg(point p, point q, point A, point B) {
  double a = B.y - A.y;
  double b = A.x - B.x;
  double c = B.x * A.y - A.x * B.y;
  double u = fabs(a * p.x + b * p.y + c);
  double v = fabs(a * q.x + b * q.y + c);
  return point((p.x * v + q.x * u) / (u+v), (p.y * v + q.y * u) / (u+v)); }

// cuts polygon Q along the line formed by point a -> point b
// (note: the last point must be the same as the first point)
vector<point> cutPolygon(point a, point b, const vector<point> &Q) {
  vector<point> P;
  for (int i = 0; i < (int)Q.size(); i++) {
    double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2 = 0;
    if (i != (int)Q.size()-1) left2 = cross(toVec(a, b), toVec(a, Q[i+1]));
    if (left1 > -EPS) P.push_back(Q[i]);       // Q[i] is on the left of ab
    if (left1 * left2 < -EPS)        // edge (Q[i], Q[i+1]) crosses line ab
      P.push_back(lineIntersectSeg(Q[i], Q[i+1], a, b));
  }
  if (!P.empty() && !(P.back() == P.front()))
    P.push_back(P.front());        // make P's first point = P's last point
  return P; }
```

To further help readers to understand these algorithms on polygon, we have build a visualization tool for the third edition of this book. The reader can draw their own polygon and asks the tool to visually explain the algorithm on polygon discussed in this section.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/polygon.html`

Exercise 7.3.6.1: This `cutPolygon` function only returns the left side of the polygon Q after cutting it with line ab. What should we do if we want the right side instead?

Exercise 7.3.6.2*: What happen if we run `cutPolygon` function on a *concave* polygon?

7.3.7 Finding the Convex Hull of a Set of Points

The **Convex Hull** of a set of points P is the smallest convex polygon $CH(P)$ for which each point in P is either on the boundary of $CH(P)$ or in its interior. Imagine that the points are nails on a flat 2D plane and we have a long enough rubber band that can enclose all the nails. If this rubber band is released, it will try to enclose as small an area as possible. That area is the area of the convex hull of these set of points/nails (see Figure 7.11). Finding convex hull of a set of points has natural applications in *packing* problems.

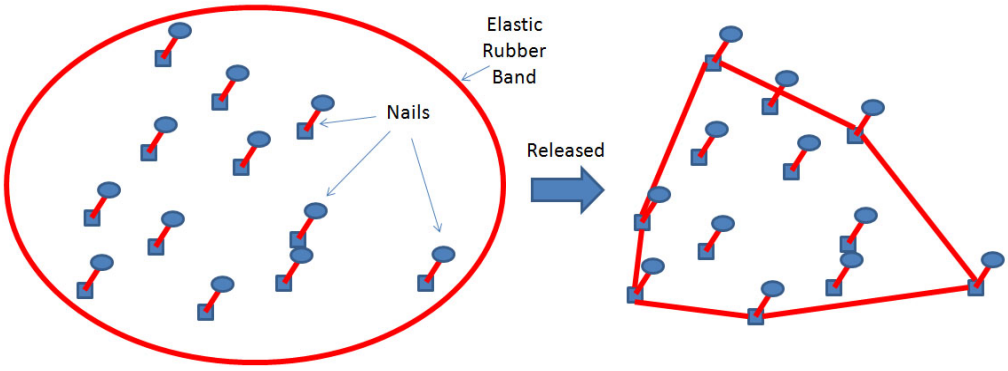


Figure 7.11: Rubber Band Analogy for Finding Convex Hull

As every vertex in $CH(P)$ is a vertex in the set of points P, the algorithm for finding convex hull is essentially an algorithm to decide which points in P should be chosen as part of the convex hull. There are several convex hull finding algorithms available. In this section, we choose the $O(n \log n)$ Ronald *Graham's Scan* algorithm.

Graham's scan algorithm first sorts all the n points of P where the first point does not have to be replicated as the last point (see Figure 7.12.A) based on their angles w.r.t a point called pivot. In our example, we pick the bottommost and rightmost point in P as pivot. After sorting based on angles w.r.t this pivot, we can see that edge 0-1, 0-2, 0-3, ..., 0-10, and 0-11 are in counter clockwise order (see point 1 to 11 w.r.t point 0 in Figure 7.12.B)!

```
point pivot(0, 0);
bool angleCmp(point a, point b) {                  // angle-sorting function
  if (collinear(pivot, a, b))                              // special case
    return dist(pivot, a) < dist(pivot, b);    // check which one is closer
  double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
  double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
  return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0; }   // compare two angles
```

```
vector<point> CH(vector<point> P) {   // the content of P may be reshuffled
  int i, j, n = (int)P.size();
  if (n <= 3) {
    if (!(P[0] == P[n-1])) P.push_back(P[0]); // safeguard from corner case
    return P; }                          // special case, the CH is P itself

  // first, find P0 = point with lowest Y and if tie: rightmost X
  int P0 = 0;
  for (i = 1; i < n; i++)
    if (P[i].y < P[P0].y || (P[i].y == P[P0].y && P[i].x > P[P0].x))
      P0 = i;

  point temp = P[0]; P[0] = P[P0]; P[P0] = temp;    // swap P[P0] with P[0]

  // second, sort points by angle w.r.t. pivot P0
  pivot = P[0];                    // use this global variable as reference
  sort(++P.begin(), P.end(), angleCmp);              // we do not sort P[0]
// to be continued
```

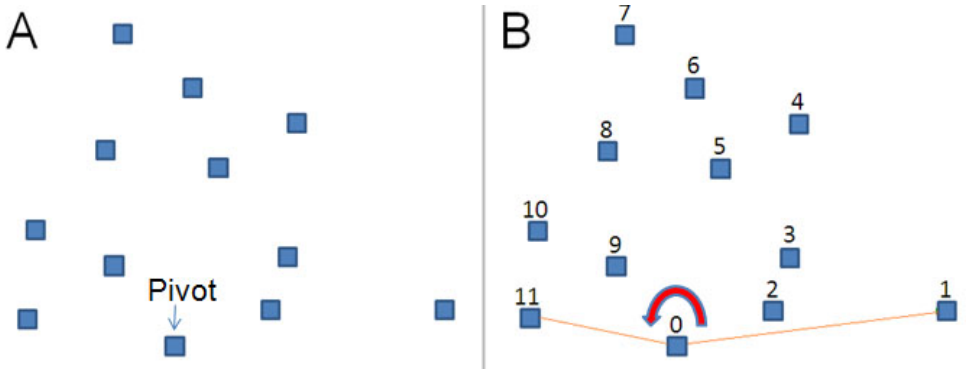


Figure 7.12: Sorting Set of 12 Points by Their Angles w.r.t a Pivot (Point 0)

Then, this algorithm maintains a stack S of candidate points. Each point of P is pushed *once* on to S and points that are not going to be part of $CH(P)$ will be eventually popped from S. Graham's Scan maintains this invariant: The top three items in stack S must always make a left turn (which is a basic property of a convex polygon).

Initially we insert these three points, point N-1, 0, and 1. In our example, the stack initially contains (bottom) 11-0-1 (top). This always form a left turn.

Now, examine Figure 7.13.C. Here, we try to insert point 2 and 0-1-2 is a left turn, so we accept point 2. Stack S is now (bottom) 11-0-1-2 (top).

Next, examine Figure 7.13.D. Here, we try to insert point 3 and 1-2-3 is a *right* turn. This means, if we accept the point before point 3, which is point 2, we will not have a convex polygon. So we have to pop point 2. Stack S is now (bottom) 11-0-1 (top) again. Then we re-try inserting point 3. Now 0-1-3, the *current* top three items in stack S form a left turn, so we accept point 3. Stack S is now (bottom) 11-0-1-3 (top).

We repeat this process until all vertices have been processed (see Figure 7.13.E-F-G-...-H). When Graham's Scan terminates, whatever that is left in S are the points of $CH(P)$ (see Figure 7.13.H, the stack contains (bottom) 11-0-1-4-7-10-11 (top)). Graham Scan's eliminates all the right turns! As three consecutive vertices in S always make left turns, we have a convex polygon.

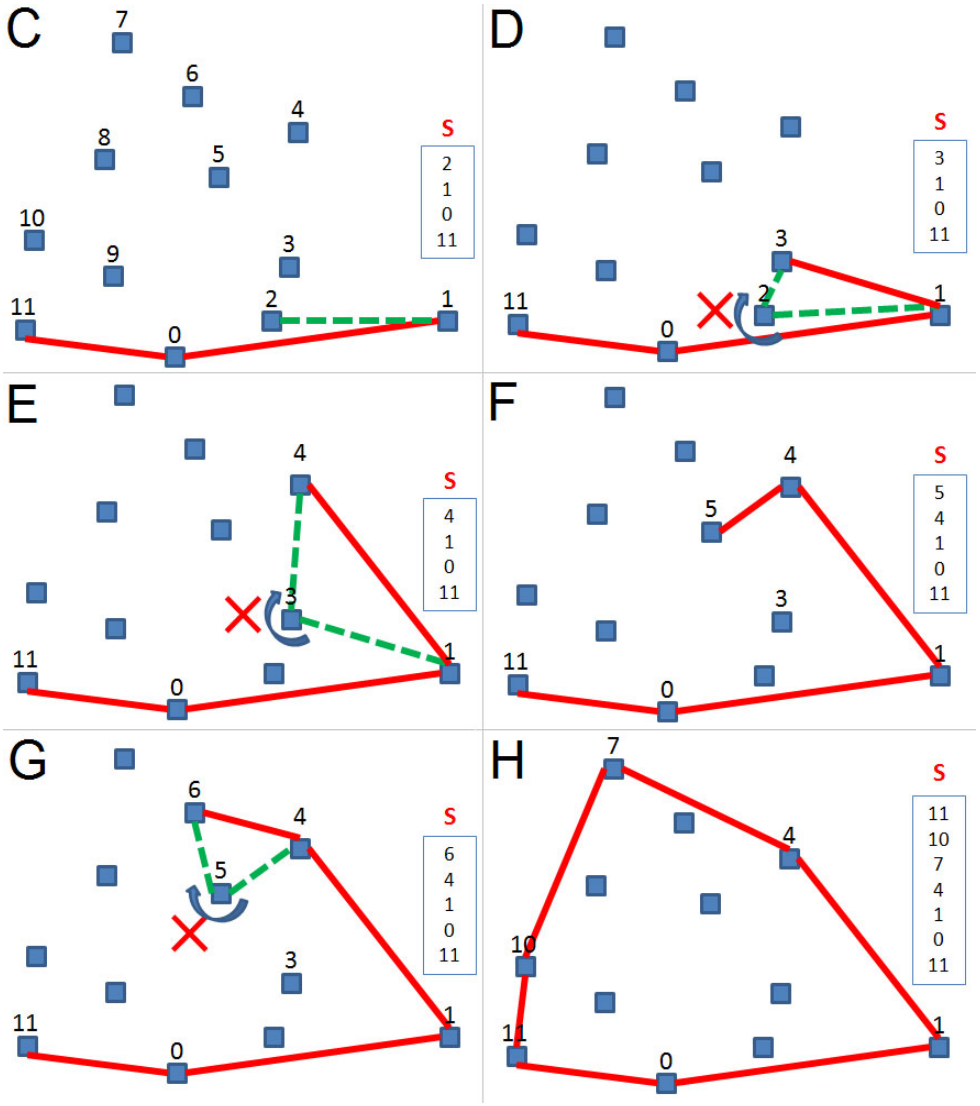


Figure 7.13: The Main Part of Graham's Scan algorithm

The implementation of Graham's Scan is shown below. We simply use a `vector<point> S` that behaves like a stack instead of using `stack<point> S`. The first part of Graham's Scan (finding the pivot) is just $O(n)$. The third part (the ccw tests) is also $O(n)$. This can be analyzed from the fact that each of the n vertices can only be pushed onto the stack once and popped from the stack once. The second part (sorts points by angle w.r.t pivot `P[0]`) is the *bulkiest* part that requires $O(n \log n)$. Overall, Graham's scan runs in $O(n \log n)$.

```
// continuation from the earlier part
  // third, the ccw tests
  vector<point> S;
  S.push_back(P[n-1]); S.push_back(P[0]); S.push_back(P[1]);   // initial S
  i = 2;                                                 // then, we check the rest
  while (i < n) {           // note: N must be >= 3 for this method to work
    j = (int)S.size()-1;
    if (ccw(S[j-1], S[j], P[i])) S.push_back(P[i++]);  // left turn, accept
    else S.pop_back(); }   // or pop the top of S until we have a left turn
  return S; }                                            // return the result
```

We end this section and this chapter by pointing readers to another visualization tool, this time the visualization of several convex hull algorithms, including Graham's Scan, Andrew's Monotone Chain algorithm (see **Exercise 7.3.7.4***), and Jarvis's March algorithm. We also encourage readers to explore our source code to solve various programming exercises listed in this section.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/convexhull.html`

Source code: `ch7_04_polygon.cpp/java`

Exercise 7.3.7.1: Suppose we have 5 points, $P = \{(0,0),(1,0),(2,0),(2,2),(0,2))\}$. The convex hull of these 5 points are actually these 5 points themselves (plus one, as we loop back to vertex (0, 0)). However, our Graham's scan implementation removes point (1, 0) as (0, 0)-(1, 0)-(2, 0) are collinear. Which part of the Graham's scan implementation that we have to modify to accept collinear points?

Exercise 7.3.7.2: In function `angleCmp`, there is a call to function: `atan2`. This function is used to compare the two angles but what is actually returned by `atan2`? Investigate!

Exercise 7.3.7.3*: Test the Graham's Scan code above: `CH(P)` on these corner cases. What is the convex hull of:

1. A single point, e.g. $P_1 = \{(0,0)\}$?
2. Two points (a line), e.g. $P_2 = \{(0,0),(1,0)\}$?
3. Three points (a triangle), e.g. $P_3 = \{(0,0),(1,0),(1,1)\}$?
4. Three points (a collinear line), e.g. $P_4 = \{(0,0),(1,0),(2,0)\}$?
5. Four points (a collinear line), e.g. $P_5 = \{(0,0),(1,0),(2,0),(3,0)\}$?

Exercise 7.3.7.4*: The Graham's Scan implementation above can be inefficient for large n as `atan2` is recalculated every time an angle comparison is made (and it is quite problematic when the angle is close to 90 degrees). Actually, the same basic idea of Graham's Scan also works if the input is sorted based on x-coordinate (and in case of a tie, by y-coordinate) instead of angle. The hull is now computed in 2 steps producing the *upper* and *lower* parts of the hull. This modification was devised by A. M. Andrew and known as Andrew's Monotone Chain Algorithm. It has the same basic properties as Graham's Scan but avoids costly comparisons between angles [9]. Investigate this algorithm and implement it!

Below, we provide a list of programming exercises related to polygon. Without pre-written library code discussed in this section, many of these problems look 'hard'. With the library code, they become manageable as the problem can now be decomposed into a few library routines. Spend some time to attempt them, especially the **must try *** ones.

Programming Exercises related to Polygon:

1. UVa 00109 - Scud Busters (find `CH`, test if point `inPolygon`, `area` of polygon)
2. *UVa 00137 - Polygons* (convex polygon intersection, line segment intersection, `inPolygon`, `CH`, `area`, inclusion-exclusion principle)
3. UVa 00218 - Moth Eradication (find `CH`, `perimeter` of polygon)
4. UVa 00361 - Cops and Robbers (check if a point is inside `CH` of Cop/Robber; if a point *pt* is inside a convex hull, then there is definitely a triangle formed using three vertices of the convex hull that contains *pt*)
5. UVa 00478 - Points in Figures: ... (`inPolygon`/inTriangle; if the given polygon P is *convex*, there is another way to check if a point *pt* is inside or outside P other than the way mentioned in this section; we can triangulate P into triangles with *pt* as one of the vertex, then sum the areas of the triangles; if it is the same as the area of polygon P, then *pt* is inside P; if it is larger, then *pt* is outside P)
6. *UVa 00596 - The Incredible Hull* (`CH`, output formatting is a bit tedious)
7. UVa 00634 - Polygon (`inPolygon`; the polygon can be convex or concave)
8. UVa 00681 - Convex Hull Finding (pure `CH` problem)
9. UVa 00858 - Berry Picking (ver line-polygon intersect; sort; alternating segments)
10. ***UVa 01111 - Trash Removal ** (LA 5138, World Finals Orlando11, `CH`, distance of each CH side—which is parallel to the side—to each vertex of the CH)
11. UVa 01206 - Boundary Points (LA 3169, Manila06, convex hull `CH`)
12. *UVa 10002 - Center of Mass?* (centroid, center of `CH`, `area` of polygon)
13. UVa 10060 - A Hole to Catch a Man (`area` of polygon)
14. UVa 10065 - Useless Tile Packers (find `CH`, `area` of polygon)
15. UVa 10112 - Myacm Triangles (test if point `inPolygon`/inTriangle, see UVa 478)
16. UVa 10406 - Cutting tabletops (vector, `rotate`, `translate`, then `cutPolygon`)
17. **UVa 10652 - Board Wrapping *** (`rotate`, `translate`, `CH`, `area`)
18. UVa 11096 - Nails (very classic `CH` problem, start from here)
19. **UVa 11265 - The Sultan's Problem *** (`cutPolygon`, `inPolygon`, `area`)
20. UVa 11447 - Reservoir Logs (`area` of polygon)
21. UVa 11473 - Campus Roads (`perimeter` of polygon)
22. UVa 11626 - Convex Hull (find `CH`, be careful with collinear points)

7.4 Solution to Non-Starred Exercises

Exercise 7.2.1.1: 5.0.

Exercise 7.2.1.2: (-3.0, 10.0).

Exercise 7.2.1.3: (-0.674, 10.419).

Exercise 7.2.2.1: The line equation $y = mx + c$ cannot handle all cases: Vertical lines has 'infinite' gradient/slope in this equation and 'near vertical' lines are also problematic. If we use this line equation, we have to treat vertical lines separately in our code which decreases the probability of acceptance. Fortunately, this can be avoided by using the better line equation $ax + by + c = 0$.

Exercise 7.2.2.2: -0.5 * x + 1.0 * y - 1.0 = 0.0

Exercise 7.2.2.3: 1.0 * x + 0.0 * y - 2.0 = 0.0. If you use the $y = mx + c$ line equation, you will have $x = 2.0$ instead, but you cannot represent a vertical line using this form $y =?$.

Exercise 7.2.2.4: Given 2 points $(x1, y1)$ and $(x2, y2)$, the slope can be calculated with $m = (y2-y1)/(x2-x1)$. Subsequently the y-intercept c can be computed from the equation by substitution of the values of a point (either one) and the line gradient m. The code will looks like this. See that we have to deal with vertical line separately and awkwardly.

```
struct line2 { double m, c; };      // another way to represent a line

int pointsToLine2(point p1, point p2, line2 &l) {
 if (p1.x == p2.x) {                    // special case: vertical line
   l.m = INF;                    // l contains m = INF and c = x_value
   l.c = p1.x;                   // to denote vertical line x = x_value
   return 0;   // we need this return variable to differentiate result
 }
 else {
   l.m = (double)(p1.y - p2.y) / (p1.x - p2.x);
   l.c = p1.y - l.m * p1.x;
   return 1;     // l contains m and c of the line equation y = mx + c
} }
```

Exercise 7.2.2.5:

```
// convert point and gradient/slope to line
void pointSlopeToLine(point p, double m, line &l) {
  l.a = -m;                                                 // always -m
  l.b = 1;                                                   // always 1
  l.c = -((l.a * p.x) + (l.b * p.y)); }                 // compute this
```

Exercise 7.2.2.6: (5.0, 3.0).

Exercise 7.2.2.7: (4.0, 2.5).

Exercise 7.2.2.8: (-3.0, 5.0).

Exercise 7.2.2.9: (0.0, 4.0). The result is different from **Exercise 7.2.2.8**. 'Translate then Rotate' is different from 'Rotate then Translate'. Be careful in sequencing them.

Exercise 7.2.2.10: (1.0, 2.0). If the rotation center is not origin, we need to translate the input point c (3, 2) by a vector described by $-p$, i.e. (-2, -1) to point c' (1, 1). Then, we perform the 90 degrees counter clockwise rotation around origin to get c'' (-1, 1). Finally, we translate c'' to the final answer by a vector described by p to point (1, 2).

Exercise 7.2.2.11: The solution is shown below:

```
void closestPoint(line l, point p, point &ans) {
  line perpendicular;          // perpendicular to l and pass through p
  if (fabs(l.b) < EPS) {               // special case 1: vertical line
    ans.x = -(l.c);   ans.y = p.y;       return; }

  if (fabs(l.a) < EPS) {             // special case 2: horizontal line
    ans.x = p.x;      ans.y = -(l.c);    return; }

  pointSlopeToLine(p, 1 / l.a, perpendicular);          // normal line
  // intersect line l with this perpendicular line
  // the intersection point is the closest point
  areIntersect(l, perpendicular, ans); }
```

Exercise 7.2.2.12: The solution is shown below. Other solution exists:

```
// returns the reflection of point on a line
void reflectionPoint(line l, point p, point &ans) {
  point b;
  closestPoint(l, p, b);                      // similar to distToLine
  vec v = toVector(p, b);                           // create a vector
  ans = translate(translate(p, v), v); }         // translate p twice
```

Exercise 7.2.2.13: 63.43 degrees.

Exercise 7.2.2.14: Point p (3,7) $\rightarrow$ point q (11,13) $\rightarrow$ point r (35,30) form a right turn. Therefore, point p is on the right side of a line that passes through point p and point r. Note: If point r is at (35, 31), then p, q, r are collinear.

Exercise 7.2.3.1: See Figure 7.14 below.

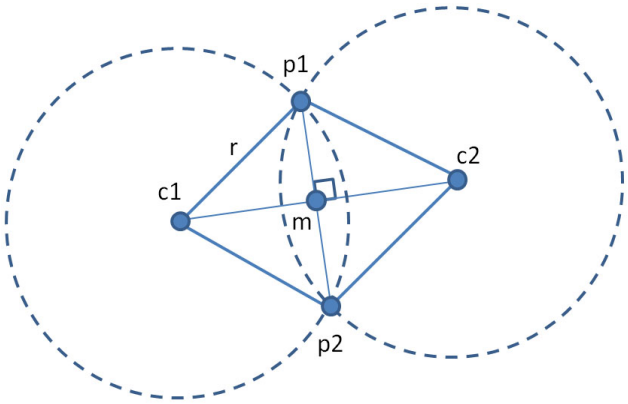


Figure 7.14: Explanation for Circle Through 2 Points and Radius

Let $c1$ and $c2$ be the centers of the 2 possible circles that go through 2 given points $p1$ and $p2$ and have radius r. The quadrilateral $p1-c2-p2-c1$ is a rhombus, since its four sides are equal. Let m be the intersection of the 2 diagonals of the rhombus $p1-c2-p2-c1$. According to the property of a rhombus, m bisects the 2 diagonals, and the 2 diagonals are perpendicular to each other. We realize that $c1$ and $c2$ can be calculated by scaling the vectors $mp1$ and $mp2$ by an appropriate ratio ($mc1/mp1$) to get the same magnitude as $mc1$, then rotating the points $p1$ and $p2$ around m by 90 degrees. In the implementation given in **Exercise 7.2.3.1**, the variable h is *half* the ratio $mc1/mp1$ (one can work out on paper why h can be calculated as such). In the 2 lines calculating the coordinates of one of the centers, the first operands of the additions are the coordinates of m, while the second operands of the additions are the result of scaling and rotating the vector $mp2$ around m.

Exercise 7.2.4.1: We can use double data type that has larger range. However, to further reduce the chance of overflow, we can rewrite the Heron's formula into $A = sqrt(s) \times sqrt(s-a) \times sqrt(s-b) \times sqrt(s-c))$. However, the result will be slightly less precise as we call $sqrt$ 4 times instead of once.

Exercise 7.3.6.1: Swap point a and b when calling `cutPolygon(a, b, Q)`.

Exercise 7.3.7.1: Edit the `ccw` function to accept collinear points.

Exercise 7.3.7.2: The function `atan2` computes the inverse tangent of $\frac{y}{x}$ using the signs of arguments to correctly determine quadrant.

7.5 Chapter Notes

Some material in this chapter are derived from the material courtesy of **Dr Cheng Holun, Alan** from School of Computing, National University of Singapore. Some library functions are customized from **Igor Naverniouk**'s library: `http://shygypsy.com/tools/`.

Compared to the first edition of this book, this chapter has, just like Chapter 5 and 6, grown to about twice its original size. However, the material mentioned here are still far from complete, especially for ICPC contestants. If you are preparing for ICPC, it is a good idea to dedicate one person in your team to study this topic in depth. This person should master basic geometry formulas and advanced computational geometry techniques, perhaps by reading relevant chapters in the following books: [50, 9, 7]. But not just the theory, he must also train himself to code *robust* geometry solutions that are able to handle degenerate (special) cases and precision errors.

The other computational geometry techniques that have not been discussed yet in this chapter are the **plane sweep** technique, intersection of **other geometric objects** including line segment-line segment intersection, various Divide and Conquer solutions for several classical geometry problems: **The Closest Pair Problem**, **The Furthest Pair Problem**, **Rotating Calipers** algorithm, etc. Some of these problems are discussed in Chapter 9.

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	13	22 (+69%)	29 (+32%)
Written Exercises	-	20	22+9*=31 (+55%)
Programming Exercises	96	103 (+7%)	96 (-7%)

The breakdown of the number[16] of programming exercises from each section is shown below:

Section	Title	Appearance	% in Chapter	% in Book
7.2	**Basic Geometry Objects ...**	74	77%	4%
7.3	**Algorithm on Polygon ...**	22	23%	1%

[16]The total decreases a bit although we have added several new problems because some of the problems are moved to Chapter 8


Bina Nusantara
06
L-R: Mr Raymond, Felix, Andrian, Andoko @ ACM ICPC World Finals, Tokyo 2007

Chapter 8

More Advanced Topics

Genius is one percent inspiration, ninety-nine percent perspiration.
— **Thomas Alva Edison**

8.1 Overview and Motivation

The main purpose of having this chapter is organizational. The first two sections of this chapter contain the harder material from Chapter 3 and 4. In Section 8.2 and 8.3, we discuss the more challenging variants and techniques involving the two most popular problem solving paradigms: Complete Search and Dynamic Programming. Putting these material in the earlier chapters will probably scare off some new readers of this book.

Section 8.4 contains discussions of complex problems that require *more than one* algorithms and/or data structures. These discussions can be confusing for new programmers if they are listed in the earlier chapters. It is more appropriate to discuss them in this chapter, after various (easier) data structures and algorithms have been discussed. Therefore, it is a good idea to read Chapter 1-7 first before reading this section.

We also encourage readers to avoid rote memorization of the solutions but more importantly, please try to understand the key ideas that may be applicable to other problems.

8.2 More Advanced Search Techniques

In Section 3.2, we have discussed various (simpler) iterative and recursive (backtracking) Complete Search techniques. However, some harder problems require *more clever* Complete Search solutions to avoid the Time Limit Exceeded (TLE) verdict. In this section, we discuss some of these techniques with several examples.

8.2.1 Backtracking with Bitmask

In Section 2.2, we have seen that bitmask can be used to model a small set of Boolean. Bitmask operations are very lightweight and therefore every time we need to use a small set of Boolean, we can consider using bitmask technique to speed up our (Complete Search) solution. In this subsection, we give two examples.

The N-Queens Problem, Revisited

In Section 3.2.2, we have discussed UVa 11195 - Another n-Queen Problem. But even after we have improved the left and right diagonal checks by storing the availability of each of the

n rows and the $2 \times n - 1$ left/right diagonals in three `bitset`s, we still get TLE. Converting these three `bitset`s into three bitmasks help a bit, but this is still TLE.

Fortunately, there is a better way to use these row, left diagonal, and right diagonal checks, as described below. This formulation[1] allows for efficient backtracking with bitmask. We will straightforwardly use three bitmasks for `rw`, `ld`, and `rd` to represent the state of the search. The on bits in bitmasks `rw`, `ld`, and `rd` describe which *rows* are attacked in the *next column*, due to *row*, *left diagonal*, or *right diagonal* attacks from previously placed queens, respectively. Since we consider one column at a time, there will only be n possible left/right diagonals, hence we can have three bitmasks of the same length of n bits (compared with $2 \times n - 1$ bits for the left/right diagonals in the earlier formulation in Section 3.2.2).

Notice that although both solutions (the one in Section 3.2.2 and the one above) use the same data structure: Three bitmasks, the one described above is much more efficient. This highlights the need for problem solver to think from various angles.

We first show the short code of this recursive backtracking with bitmask for the (general) n-queens problem with $n = 5$ and then explain how it works.

```
int ans = 0, OK = (1 << 5) - 1;                    // testing for n = 5 queens

void backtrack(int rw, int ld, int rd) {
  if (rw == OK) { ans++; return; }              // if all bits in 'rw' are on
  int pos = OK & (~(rw | ld | rd));       // the '1's in 'pos' are available
  while (pos) {                                 // this loop is faster than O(n)
    int p = pos & -pos;               // Least Significant One---this is fast
    pos -= p;                                          // turn off that on bit
    backtrack(rw | p, (ld | p) << 1, (rd | p) >> 1);               // clever
} }

int main() {
  backtrack(0, 0, 0);                                    // the starting point
  printf("%d\n", ans);                  // the answer should be 10 for n = 5
} // return 0;
```

For $n = 5$, we start with state `(rw, ld, rd) = (0, 0, 0) = (00000, 00000, 00000)`$_2$. This state is shown in Figure 8.1. The variable `OK = (1 << 5) - 1 = (11111)`$_2$ is used both as terminating condition check and to help decide which rows are available for a certain column. The operation `pos = OK & (~(rw | ld | rd))` *combines* the information of which rows in the next column are attacked by the previously placed queens (via row, left diagonal, or right diagonal attacks), *negates* the result, and *combines* it with `OK` to yield the rows that are *available* for the next column. Initially, all rows in column 0 are available.

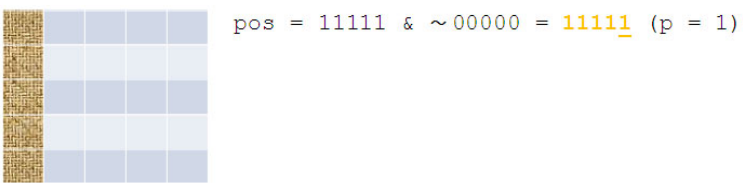


Figure 8.1: 5 Queens problem: The initial state

[1]While this solution is customized for this N-Queens problem, we can probably use parts of the solution for another problem.

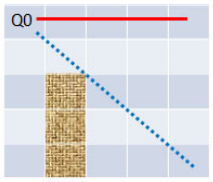
Complete Search (the recursive backtracking) will try all possible rows (that is, all the *on bits* in variable `pos`) of a certain column one by one. Previously in Section 3.2.1, we have seen a way to explore all the on bits of a bitmask in $O(n)$:

```
for (p = 0; p < n; p++)                                          // O(n)
  if (pos && (1 << p))                    // if this bit 'p' is on in 'pos'
    // process p
```

However, this is not the most efficient way. As the recursive backtracking goes deeper, less and less rows are available for selection. Instead of trying all n rows, we can speed up the loop above by just trying all the on bits in variable `pos`. The loop below runs in $O(k)$:

```
while (pos) { // O(k), where k is the number of bits that are on in 'pos'
  int p = pos & -pos;     // determine the Least Significant One in 'pos'
  pos -= p;                                          // turn off that on bit
  // process p
}
```

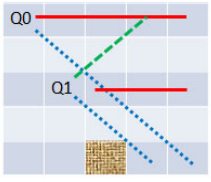
Back to our discussion, for `pos = (11111)`$_2$, we will first start with `p = pos & -pos = 1`, or row 0. After placing the first queen (queen 0) at row 0 of column 0, row 0 is no longer available for the next column 1 and this is quickly captured by bit operation `rw | p` (and also `ld | p` and `rd | p`). Now here is the beauty of this solution. A left/right diagonal increases/decreases the row number that it attacks by one as it changes to the next column, respectively. A shift left/right operation: `(ld | p) << 1` and `(rd | p) >> 1` can nicely capture these behaviour effectively. In Figure 8.2, we see that for the next column 1, row 1 is not available due to left diagonal attack by queen 0. Now only row 2, 3, and 4 are available for column 1. We will start with row 2.




```
rw              = 00001 (1)
ld = 00001 << 1 = 00010 (2)
rd = 00001 >> 1 = 00000|1 (0, the LSB is removed)
---------------------- OR
                  00011 -> NEGATE -> 11100
pos = 11111 & 11100 = 11100 (p = 4)
```

Figure 8.2: 5 Queens problem: After placing the first queen

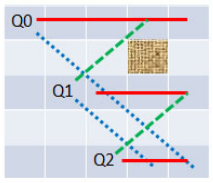
After placing the second queen (queen 1) at row 2 of column 1, row 0 (due to queen 0) and now row 2 are no longer available for the next column 2. The shift left operation for the left diagonal constraint causes row 2 (due to queen 0) and now row 3 to be unavailable for the next column 2. The shift right operation for the right diagonal constraint causes row 1 to be unavailable for the next column 2. Therefore, only row 4 is available for the next column 2 and we have to choose it next (see Figure 8.3).




```
rw              = 00101 (5)
ld = 00110 << 1 = 01100 (12)
rd = 00100 >> 1 = 00010 (2)
---------------------- OR
                  01111 -> NEGATE -> 10000
pos = 11111 & 10000 = 10000 (p = 16)
```

Figure 8.3: 5 Queens problem: After placing the second queen

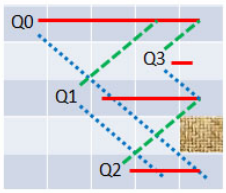
After placing the third queen (queen 2) at row 4 of column 2, row 0 (due to queen 0), row 2 (due to queen 1), and now row 4 are no longer available for the next column 3. The shift left operation for the left diagonal constraint causes row 3 (due to queen 0) and row 4 (due to queen 1) to be unavailable for the next column 3 (there is no row 5—the MSB in bitmask `ld` is unused). The shift right operation for the right diagonal constraint causes row 0 (due to queen 1) and now row 3 to be unavailable for the next column 3. Combining all these information, only row 1 is available for the next column 3 and we have to choose it next (see Figure 8.4).



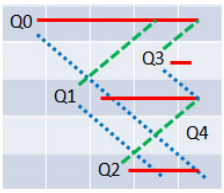

```
rw              =  10101 (21)
ld = 11100 << 1 = 111000 (56, but the MSB is unused)
rd = 10010 >> 1 =  01001 (9)
------------------------ OR
                   111101 -> NEGATE -> 000010
pos = 11111 & 000010 = 00010 (p = 2)
```

Figure 8.4: 5 Queens problem: After placing the third queen

The same explanation is applicable for the fourth and the fifth queen (queen 3 and 4) as shown in Figure 8.5. We can continue this process to get the other 9 solutions for $n = 5$.




```
rw              =   10111 (23)
ld = 111010 << 1 = 1110100 (116, two MSBs are unused)
rd =  01011 >> 1 =   00101 (5)
--------------------------- OR
                   1110111 -> NEGATE -> 0001000
pos = 11111 & 0001000 = 01000 (p = 8)
```




```
rw              =   11111 (we get one solution)
```

Figure 8.5: N-Queens, after placing the fourth and the fifth queens

With this technique, we can solve UVa 11195. We just need to modify the given code above to take the bad cells—which can also be modeled as bitmasks—into consideration. Let's roughly analyze the worst case for $n \times n$ board with no bad cell. Assuming that this recursive backtracking with bitmask has approximately two less rows available at each step, we have a time complexity of $O(n!!)$ where $n!!$ is a notation of multifactorial. For $n = 14$ with no bad cell, the recursive backtracking solution in Section 3.2.2 requires up to $O(14!) \approx 87178M$ operations whereas the recursive backtracking with bitmask above only require around $O(14!!) = 14 \times 12 \times 10 \times \ldots \times 2 = 645120$ operations.

Compact Adjacency Matrix Graph Data Structure

The UVa 11065 - Gentlemen Agreement problem boils down to computation of two integers: The number of Independent Set and the size of the Maximum Independent Set (MIS—see Section 4.7.4 for the problem definition) of a given *general* graph with $V \leq 60$. Finding the MIS of a general graph is an NP-hard problem. Therefore, there is no hope for a polynomial algorithm for this problem.

One solution is the following clever recursive backtracking. The state of the search is a triple: $(i, used, depth)$. The first parameter i implies that we can consider vertices in $[i..V\text{-}1]$ to be included in the Independent Set. The second parameter *used* is a bitmask of length V bits that denotes which vertices are no longer available to be used anymore for the current Independent Set because at least one of their neighbors have been included in the Independent Set. The third parameter *depth* stores the depth of the recursion—which is also the size of the current Independent Set.

There is a clever bitmask trick for this problem that can be used to speed up the solution significantly. Notice that the input graph is small, $V \leq 60$. Therefore, we can store the input graph in an Adjacency Matrix of size up to $V \times V$ (for this problem, we set all cells along the main diagonal of the Adjacency Matrix to true). However, we can compress *one row* of V Booleans ($V \leq 60$) into one bitmask using a 64-bit signed integer.

With this compact Adjacency Matrix $AdjMat$—which is just V rows of 64-bit signed integers—we can use a fast bitmask operation to flag neighbors of vertices efficiently. If we decide to take a free vertex i—i.e. $(used \ \& \ (1 << i)) == 0$, we increase *depth* by one and then use an $O(1)$ bitmask operation: $used \mid AdjMat[i]$ to flag *all* neighbors of i including itself (remember that $AdjMat[i]$ is also a bitmask of length V bits with the i-th bit on).

When all bits in bitmask *used* is turned on, we have just found one more Independent Set. We also record the largest *depth* value throughout the process as this is the size of the Maximum Independent Set of the input graph. The key parts of the code is shown below:

```
void rec(int i, long long used, int depth) {
  if (used == (1 << V) - 1) {               // all intersection are visited
    nS++;                                          // one more possible set
    mxS = max(mxS, depth);                               // size of the set
  }
  else {
    for (int j = i; j < V; j++)
      if (!(used & (1 << j)))          // if intersection j is not yet used
        rec(j + 1, used | AdjMat[j], depth + 1);      // fast bit operation
  }
}

// inside int main()
  // a more powerful, bit-wise adjacency list (for faster set operations)
  for (int i = 0; i < V; i++)
    AdjMat[i] = (1 << i);                                    // i to itself
  for (int i = 0; i < E; i++) {
    scanf("%d %d", &a, &b);
    AdjMat[a] |= (1 << b);
    AdjMat[b] |= (1 << a);
  }
```

Exercise 8.2.1.1*: Sudoku puzzle is another NP-complete problem. The recursive backtracking to find one solution for a standard 9×9 ($n = 3$) Sudoku board can be speed up using bitmask. For each empty cell `(r, c)`, we try putting a digit `[1..`n^2`]` one by one if it is a valid move. The n^2 row, n^2 column, and $n \times n$ square checks can be done with three bitmasks of length n^2 bits. Solve two similar problems: UVa 989 and UVa 10957 with this technique!

8.2.2 Backtracking with Heavy Pruning

Problem I - 'Robots on Ice' in ACM ICPC World Finals 2010 can be viewed as a 'tough test on pruning strategy'. The problem description is simple: Given an $M \times N$ board with 3 check-in points {A, B, C}, find a Hamiltonian[2] path of length $(M \times N)$ from coordinate (0, 0) to coordinate (0, 1). This Hamiltonian path must hit the three check points: A, B, and C at one-quarter, one-half, and three-quarters of the way through its path, respectively. Constraints: $2 \leq M, N \leq 8$.

Example: If given the following 3×6 board with A = (row, col) = (2, 1), B = (2, 4), and C = (0, 4) as in Figure 8.6, then we have two possible paths.

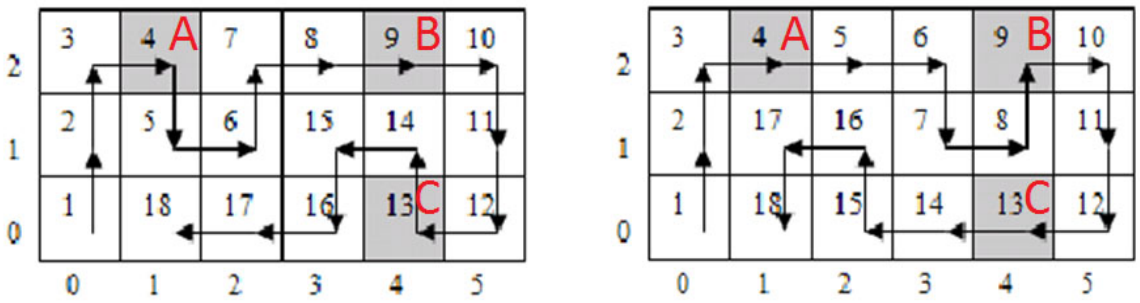


Figure 8.6: Visualization of UVa 1098 - Robots on Ice

A naïve recursive backtracking algorithm will get TLE as there are 4 choices at every step and the maximum path length is $8 \times 8 = 64$ in the largest test case. Trying all 4^{64} possible paths is infeasible. To speed up the algorithm, we must prune the search space if the search:

1. Wanders outside the $M \times N$ grid (obvious),
2. Does not hit the appropriate target check point at 1/4, 1/2, or 3/4 distance—the presence of these three check points actually *reduce* the search space,
3. Hits target check point earlier than the target time,
4. Will not be able to reach the next check point on time from the current position,
5. Will not be able to reach certain coordinates as the current partial path self-block the access to those coordinates. This can be checked with a simple DFS/BFS (see Section 4.2). First, we run DFS/BFS from the goal coordinate (0, 1). If there are coordinates in the $M \times N$ grid that are *not* reachable from (0, 1) and *not yet visited* by the current partial path, we can prune the current partial path.

Exercise 8.2.2.1*: The five pruning strategies mentioned in this subsection are good but actually insufficient to pass the time limit set for LA 4793 and UVa 1098. There is a faster solution for this problem that utilizes the meet in the middle technique (see Section 8.2.4). This example illustrates that the choice of time limit setting may determine which Complete Search solutions are considered as fast enough. Study the idea of meet in the middle technique in Section 8.2.4 and apply it to solve this Robots on Ice problem.

[2]A Hamiltonian path is a path in an undirected graph that visits each vertex exactly once.

8.2.3 State-Space Search with BFS or Dijkstra's

In Section 4.2.2 and 4.4.3, we have discussed two standard graph algorithms for solving the Single-Source Shortest Paths (SSSP) problem. BFS can be used if the graph is unweighted while Dijkstra's should be used if the graph is weighted. The SSSP problems listed in Chapter 4 are still easier in the sense that most of the time we can easily see 'the graph' in the problem description. This is no longer true for some harder graph searching problems listed in this section where the (usually implicit) graphs are no longer trivial to see and the state/vertex can be a complex object. In such case, we usually name the search as 'State-Space Search' instead of SSSP.

When the state is a complex object—e.g. a pair (position, bitmask) in UVa 321 - The New Villa, a quad (row, col, direction, color) in UVa 10047 - The Monocycle, etc—, we normally do not use `vector<int> dist` to store the distance information as in the standard BFS/Dijkstra's implementation. This is because such state may not be easily converted into integer indices. One solution is to use `map<VERTEX-TYPE, int> dist` instead. This trick adds a (small) $\log V$ factor to the time complexity of BFS/Dijkstra's. But for complex State-Space Search, this extra runtime overhead may be acceptable in order to bring down the overall coding complexity. In this subsection, we show one example of such complex State-Space Search.

Exercise 8.2.3.1: How to store `VERTEX-TYPE` if it is a pair, a triple, or a quad of information in both C++ and Java?

Exercise 8.2.3.2: Similar question as in **Exercise 8.2.3.1**, but `VERTEX-TYPE` is a much more complex object, e.g. an array.

Exercise 8.2.3.3: Is it possible that State-Space Search is cast as a maximization problem?

UVa 11212 - Editing a Book

Abridged Problem Description: Given n paragraphs numbered from 1 to n, arrange them in the order of 1, 2, ..., n. With the help of a clipboard, you can press Ctrl-X (cut) and Ctrl-V (paste) several times. You cannot cut twice before pasting, but you can cut several contiguous paragraphs at the same time and these paragraphs will later be pasted in order. What is the minimum number of steps required?

Example 1: In order to make {2, 4, (1), 5, 3, 6} sorted, we cut paragraph (1) and paste it before paragraph 2 to have {1, 2, 4, 5, (3), 6}. Then, we cut paragraph (3) and paste it before paragraph 4 to have {1, 2, 3, 4, 5, 6}. The answer is two steps.

Example 2: In order to make {(3, 4, 5), 1, 2} sorted, we cut three paragraphs at the same time: (3, 4, 5) and paste them after paragraph 2 to have {1, 2, 3, 4, 5}. This is just one single step. This solution is not unique as we can have the following alternative answer: We cut two paragraphs at the same time: (1, 2) and paste them before paragraph 3 to get {1, 2, 3, 4, 5}—this is also one single step.

The loose upper bound of the number of steps required to rearrange these n paragraphs is $O(k)$, where k is the number of paragraphs that are initially in the wrong positions. This is because we can use the following 'trivial' algorithm (which is incorrect): Cut a single paragraph that is in the wrong position and paste that paragraph in the correct position. After k such cut-paste operation, we will definitely have a sorted paragraph. But this may not be the shortest way.

For example, the 'trivial' algorithm above will process {5, 4, 3, 2, 1} as follows: {(5), 4, 3, 2, 1} → {(4), 3, 2, 1, 5} → {(3), 2, 1, 4, 5} → {(2), 1, 3, 4, 5} → {1, 2, 3, 4, 5} of total 4 cut-paste steps. This is not optimal, as we can solve this instance in only 3 steps: {5, 4, (3, 2), 1} → {3, (2, 5), 4, 1} → {3, 4, (1, 2), 5} → {1, 2, 3, 4, 5}.

This problem has a *huge* search space that even for an instance with small $n = 9$, it is near impossible for us to get the answer manually, e.g. We likely will not start drawing the recursion tree just to verify that we need at least 4 steps to sort {5, 4, 9, 8, 7, 3, 2, 1, 6} and at least 5 steps to sort {9, 8, 7, 6, 5, 4, 3, 2, 1}.

The state of this problem is a *permutation* of paragraphs. There are at most $O(n!)$ permutations of paragraphs. With maximum $n = 9$ in the problem statement, this is 9! or 362880. So, the number of vertices of the State-Space graph is not that big actually.

The difficulty of this problem lies in the number of *edges* of the State-Space graph. Given a permutation of length n (a vertex), there are ${}_nC_2$ possible cutting points (index $i, j \in [1..n]$) and there are n possible pasting points (index $k \in [1..(n-(j-i+1))]$). Therefore, for each of the $O(n!)$ vertex, there are about $O(n^3)$ edges connected to it.

The problem actually asked for the shortest path from the source vertex/state (the input permutation) to the destination vertex (a sorted permutation) on this unweighted but huge State-Space graph. The worst case behavior if we run a single $O(V + E)$ BFS on this State-Space graph is $O(n! + (n! * n^3)) = O(n! * n^3)$. For $n = 9$, this is $9! * 93 = 264539520 \approx 265M$ operations. This solution most likely will receive a TLE (or maybe MLE) verdict.

We need a better solution, which we will see in the next Section 8.2.4.

8.2.4 Meet in the Middle (Bidirectional Search)

For some SSSP (but usually State-Space Search) problems on huge graph and we know two vertices: The source vertex/state s and the destination vertex/state t, we may be able to *significantly* reduce the time complexity of the search by searching from *both directions* and hoping that the search will *meet in the middle*. We illustrate this technique by continuing our discussion of the hard UVa 11212 problem.

Before we continue, we need to make a remark that the meet in the middle technique does not always refer to bidirectional BFS. It is a problem solving strategy of 'searching from two directions/parts' that may appear in another form in other difficult searching problem, e.g. see **Exercise 3.2.1.4***.

UVa 11212 - Editing a Book (Revisited)

Although the worst case time complexity of the State-Space Search of this problem is bad, the largest possible answer for this problem is small. When we run BFS on the largest test case with $n = 9$ from the destination state t (the sorted permutation {1, 2, ..., 9}) to reach all other states, we find out that for this problem, the maximum depth of the BFS for $n = 9$ is just 5 (after running it for *a few minutes*—which is TLE in contest environment).

This important information allows us to perform bidirectional BFS by choosing only to go to depth 2 from each direction. While this information is not a necessary condition for us to run a bidirectional BFS, it can help reducing the search space.

There are three possible cases which we discuss below.

Case 1: Vertex s is within two steps away from vertex t (see Figure 8.7).
We first run BFS (max depth of BFS = 2) from the target vertex t to populate distance information from t: `dist_t`. If the source vertex s is already found, i.e. `dist_t[s]` is not `INF`, then we return this value. The possible answers are: 0 (if $s = t$), 1, or 2 steps.

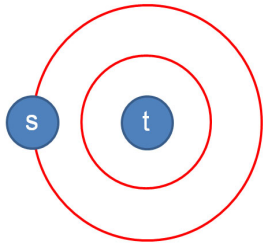


Figure 8.7: Case 1: Example when s is two steps away from t

Case 2: Vertex s is within three to four steps away from vertex t (see Figure 8.8).
If we do not manage to find the source vertex s after Case 1 above, i.e. `dist_t[s] = INF`, we know that s is located further away from vertex t. We now run BFS from the source vertex s (also with max depth of BFS = 2) to populate distance information from s: `dist_s`. If we encounter a common vertex v 'in the middle' during the execution of this second BFS, we know that vertex v is within two layers away from vertex t and s. The answer is therefore `dist_s[v] + dist_t[v]` steps. The possible answers are: 3 or 4 steps.

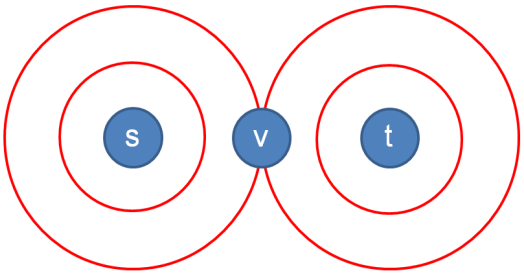


Figure 8.8: Case 2: Example when s is four steps away from t

Case 3: Vertex s is exactly five steps away from vertex t (see Figure 8.9).
If we do not manage to find any common vertex v after running the second BFS in Case 2 above, then the answer is clearly 5 steps that we know earlier as s and t must always be reachable. Stopping at depth 2 allows us to skip computing depth 3, which is *much more time consuming* than computing depth 2.

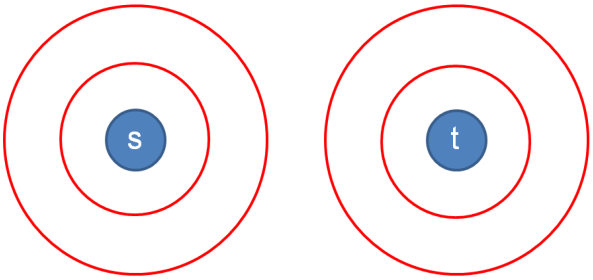


Figure 8.9: Case 3: Example when s is five steps away from t

We have seen that given a permutation of length n (a vertex), there are about $O(n^3)$ branches in this huge State-Space graph. However, if we just run each BFS with at most depth 2, we only execute at most $O((n^3)^2) = O(n^6)$ operations per BFS. With $n = 9$, this is $9^6 = 531441$ operations (this is greater than 9! as there are some overlaps). As the destination vertex t is unchanged throughout the State-Space search, we can compute the first BFS from destination vertex t just once. Then we compute the second BFS from source vertex s per query. Our BFS implementation will have an additional log factor due to the usage of table data structure (e.g. `map`) to store `dist_t` and `dist_s`. This solution is now Accepted.

8.2.5 Informed Search: A* and IDA*

The Basics of A*

Figure 8.10: 15 Puzzle

Complete Search algorithms that we have seen earlier in Chapter 3, 4, and the earlier subsections of this Section are 'uninformed', i.e. all possible states reachable from the current state are *equally good*. For some problems, we do have access to more information (hence the name 'informed search') and we can use the clever A* search that employs heuristic to 'guide' the search direction.

We illustrate this A* search using a well-known 15-puzzle problem. There are 15 slide-able tiles in the puzzle, each with a number from 1 to 15 on it. These 15 tiles are packed into a 4×4 frame with one tile missing. The possible actions are to slide the tile adjacent to the missing tile to the position of that missing tile. Another way of viewing these actions is: "To slide the *blank tile* rightwards, upwards, leftwards, or downwards". The objective of this puzzle is to arrange the tiles so that they looks like Figure 8.10, the 'goal' state.

This seemingly small puzzle is a headache for various search algorithms due to its enormous search space. We can represent a state of this puzzle by listing the numbers of the tiles row by row, left to right into an array of 16 integers. For simplicity, we assign value 0 to the blank tile so the goal state is {1, 2, 3, ..., 14, 15, 0}. Given a state, there can be up to 4 reachable states depending on the position of the missing tile. There are 2/3/4 possible actions if the missing tile is at the 4 corners/8 non-corner sides/4 middle cells, respectively. This is a huge search space.

However, these states are not equally good. There is a nice heuristic for this problem that can help guiding the search algorithm, which is the sum of Manhattan[3] distances between each (non blank) tile in the current state and its location in the goal state. This heuristic gives the lower bound of steps to reach the goal state. By combining the cost so far (denoted by $g(s)$) and the heuristic value (denoted by $h(s)$) of a state s, we have a better idea on where to move next. We illustrate this with a puzzle with starting state A below:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & \underline{\mathbf{0}} \\ 13 & 14 & 15 & 12 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & \underline{\mathbf{0}} \\ 9 & 10 & 11 & \underline{\mathbf{8}} \\ 13 & 14 & 15 & 12 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & \underline{\mathbf{0}} & \underline{\mathbf{11}} \\ 13 & 14 & 15 & 12 \end{bmatrix} \quad D = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & \underline{\mathbf{12}} \\ 13 & 14 & 15 & \underline{\mathbf{0}} \end{bmatrix}$$

The cost of the starting state A is $g(s) = 0$, no move yet. There are three reachable states $\{B, C, D\}$ from this state A with $g(B) = g(C) = g(D) = 1$, i.e. one move. But these three states are *not* equally good:

1. The heuristic value if we slide tile 0 upwards is $h(B) = 2$ as tile 8 and tile 12 are both off by 1. This causes $g(B) + h(B) = 1 + 2 = 3$.
2. The heuristic value if we slide tile 0 leftwards is $h(C) = 2$ as tile 11 and tile 12 are both off by 1. This causes $g(C) + h(C) = 1 + 2 = 3$.
3. But if we slide tile 0 downwards, we have $h(D) = 0$ as all tiles are in their correct position. This causes $g(D) + h(D) = 1 + 0 = 1$, the lowest combination.

[3]The Manhattan distance between two points is the sum of the absolute differences of their coordinates.

If we visit the states in ascending order of $g(s) + h(s)$ values, we will explore the states with the smaller expected cost first, i.e. state D in this example—which is the goal state. This is the essence of the A* search algorithm.

We usually implement this states ordering with the help of a priority queue—which makes the implementation of A* search very similar with the implementation of Dijkstra's algorithm presented in Section 4.4. Note that if $h(s)$ is set to 0 for all states, A* *degenerates* to Dijkstra's algorithm again.

As long as the heuristic function $h(s)$ never overestimates the true distance to the goal state (also known as **admissible heuristic**), this A* search algorithm is optimal. The hardest part in solving search problems using A* search is in finding such heuristic.

Limitations of A*

The problem with A* (and also BFS and Dijkstra's algorithms when used on large State-Space graph) that uses (priority) queue is that the memory requirement can be very huge when the goal state is far from the initial state. For some difficult searching problem, we may have to resort to the following related techniques.

Depth Limited Search

In Section 3.2.2, we have seen recursive backtracking algorithm. The main problem with pure backtracking is this: It may be trapped in an exploration of a very deep path that will not lead to the solution before eventually backtracks after wasting precious runtime.

Depth Limited Search (DLS) places a limit on how deep a backtracking can go. DLS stops going deeper when the depth of the search is longer than what we have defined. If the limit happens to be equal to the depth of the shallowest goal state, then DLS is faster than the general backtracking routine. However, if the limit is too small, then the goal state will be unreachable. If the problem says that the goal state is 'at most d steps away' from the initial state, then use DLS instead of general backtracking routine.

Iterative Deepening Search

If DLS is used wrongly, then the goal state will be unreachable although we have a solution. DLS is usually not used alone, but as part of Iterative Deepening Search (IDS).

IDS calls DLS with *increasing limit* until the goal state is found. IDS is therefore complete and optimal. IDS is a nice strategy that sidesteps the problematic issue of determining the best depth limit by trying all possible depth limits incrementally: First depth 0 (the initial state itself), then depth 1 (those reachable with just one step from the initial state), then depth 2, and so on. By doing this, IDS essentially combines the benefits of lightweight/memory friendly DFS and the ability of BFS that can visit neighboring states layer by layer (see Table 4.2 in Section 4.2).

Although IDS calls DLS many times, the time complexity is still $O(b^d)$ where b is the branching factor and d is the depth of the shallowest goal state. Reason: $O(b^0 + (b^0 + b^1) + (b^0 + b^1 + b^2) + ... + (b^0 + b^1 + b^2 + ... + b^d)) \leq O(c \times b^d) = O(b^d)$.

Iterative Deepening A* (IDA*)

To solve the 15-puzzle problem faster, we can use IDA* (Iterative Deepening A*) algorithm which is essentially IDS with modified DLS. IDA* calls modified DLS to try the all neighboring states in a fixed order (i.e. slide tile 0 rightwards, then upwards, then leftwards, then finally downwards—in that order; we do not use a priority queue). This modified DLS is

stopped not when it has exceeded the depth limit but when its $g(s) + h(s)$ exceeds the best known solution so far. IDA* expands the limit gradually until it hits the goal state.

The implementation of IDA* is not straightforward and we invite readers to scrutinize the given source code in the supporting website.

Source code: `ch8_01_UVa10181.cpp/java`

Exercise 8.2.5.1*: One of the hardest part in solving search problems using A* search is to find the correct admissible heuristic and to compute them efficiently as it has to be repeated many times. List down admissible heuristics that are commonly used in difficult searching problems involving A* algorithm and show how to compute them efficiently! One of them is the Manhattan distance as shown in this section.

Exercise 8.2.5.2*: Solve UVa 11212 - Editing a Book that we have discussed in depth in Section 8.2.3-8.2.4 with A* instead of bidirectional BFS! Hint: First, determine what is a suitable heuristic for this problem.

Programming Exercises solvable with More Advanced Search Techniques:

- More Challenging Backtracking Problems
  1. *UVa 00131 - The Psychic Poker Player* (backtracking with 2^5 bitmask to help deciding which card is retained in hand/exchanged with the top of deck; use 5! to shuffle the 5 cards in hand and get the best value)
  2. *UVa 00710 - The Game* (backtracking with memoization/pruning)
  3. *UVa 00711 - Dividing up* (reduce search space first before backtracking)
  4. *UVa 00989 - Su Doku* (classic Su Doku puzzle; this problem is NP complete but this instance is solvable with backtracking with pruning; use bitmask to speed up the check of available digits)
  5. *UVa 01052 - Bit Compression* (LA 3565 - WorldFinals SanAntonio06, backtracking with some form of bitmask)
  6. **UVa 10309 - Turn the Lights Off *** (brute force the first row in 2^{10}, the rest follows)
  7. *UVa 10318 - Security Panel* (the order is not important, so we can try pressing the buttons in increasing order, row by row, column by column; when pressing one button, only the 3×3 square around it is affected; therefore after we press button (i, j), light $(i-1, j-1)$ must be on (as no button afterward will affect this light); this check can be used to prune the backtracking)
  8. *UVa 10890 - Maze* (looks like a DP problem but the state—involving bitmask—cannot be memoized, fortunately the grid size is 'small')
  9. *UVa 10957 - So Doku Checker* (very similar with UVa 989; if you can solve that one, you can modify your code a bit to solve this one)
  10. **UVa 11195 - Another n-Queen Problem *** (see **Exercise 3.2.1.3*** and the discussion in Section 8.2.1)
  11. ***UVa 11065 - A Gentlemen's Agreement ** (independent set, bitmask helps in speeding up the solution; see the discussion in Section 8.2.1)
  12. *UVa 11127 - Triple-Free Binary Strings* (backtracking with bitmask)
  13. *UVa 11464 - Even Parity* (brute force the first row in 2^{15}, the rest follows)
  14. *UVa 11471 - Arrange the Tiles* (reduce search space by grouping tiles of the same type; recursive backtracking)

- More Challenging State-Space Search with BFS or Dijkstra's
  1. UVa 00321 - The New Villa (s: (position, bitmask 2^{10}), print the path)
  2. *UVa 00658 - It's not a Bug ...* (s: bitmask—whether a bug is present or not, use Dijkstra's as the State-Space graph is weighted)
  3. UVa 00928 - Eternal Truths (s: (row, col, direction, step))
  4. ***UVa 00985 - Round and Round ... **** (4 rotations is the same as 0 rotations, s: (row, col, rotation = [0..3]); find the shortest path from state [1][1][0] to state [R][C][x] where $0 \leq x \leq 3$)
  5. *UVa 01057 - Routing* (LA 3570, World Finals SanAntonio06, use Floyd Warshall's to get APSP information; then model the original problem as another weighted SSSP problem solvable with Dijkstra's)
  6. UVa 01251 - Repeated Substitution ... (LA 4637, Tokyo09, SSSP solvable with BFS)
  7. UVa 01253 - Infected Land (LA 4645, Tokyo09, SSSP solvable with BFS, tedious state modeling)
  8. UVa 10047 - The Monocyle (s: (row, col, direction, color); BFS)
  9. *UVa 10097 - The Color game* (s: (N1, N2); implicit unweighted graph; BFS)
  10. *UVa 10923 - Seven Seas* (s: (ship_position, location of enemies, location of obstacles, steps_so_far); implicit weighted graph; Dijkstra's)
  11. ***UVa 11198 - Dancing Digits **** (s: permutation; BFS; tricky to code)
  12. ***UVa 11329 - Curious Fleas **** (s: bitmask of 26 bits, 4 to describe the position of the die in the 4×4 grid, 16 to describe if a cell has a flea, 6 to describe the sides of the die that has a flea; use `map`; tedious to code)
  13. UVa 11513 - 9 Puzzle (reverse the role of source and destination)
  14. UVa 11974 - Switch The Lights (BFS on implicit unweighted graph)
  15. UVa 12135 - Switch Bulbs (LA 4201, Dhaka08, similar to UVa 11974)
- Meet in the Middle/A*/IDA*
  1. UVa 00652 - Eight (classical sliding block 8-puzzle problem, IDA*)
  2. **UVa 01098 - Robots on Ice *** (LA 4793, World Finals Harbin10, see the discussion in Section 8.2.2; however, there is a faster 'meet in the middle' solution for this problem)
  3. UVa 01217 - Route Planning (LA 3681, Kaohsiung06, solvable with A*/IDA*; test data likely only contains up to 15 stops which already include the starting and the last stop on the route)
  4. **UVa 10181 - 15-Puzzle Problem *** (similar as UVa 652, but this one is larger, we can use IDA*)
  5. UVa 11163 - Jaguar King (another puzzle game solvable with IDA*)
  6. **UVa 11212 - Editing a Book *** (meet in the middle, see Section 8.2.4)
- Also see some more Complete Search problems in Section 8.4

8.3 More Advanced DP Techniques

In Section 3.5, 4.7.1, 5.4, 5.6, and 6.5, we have seen the introduction of Dynamic Programming (DP) technique, several classical DP problems and their solutions, plus a gentle introduction to the easier non classical DP problems. There are several more advanced DP techniques that we have not covered in those sections. Here, we present some of them.

8.3.1 DP with Bitmask

Some of the modern DP problems require a (small) set of Boolean as one of the parameters of the DP state. This is another situation where bitmask technique can be useful (also see Section 8.2.1). This technique is suitable for DP as the integer (that represents the bitmask) can be used as the index of the DP table. We have seen this technique once when we discuss DP TSP (see Section 3.5.2). Here, we give one more example.

UVa 10911 - Forming Quiz Teams

For the abridged problem statement and the solution code of this problem, please refer to the very first problem mentioned in Chapter 1. The grandiose name of this problem is "minimum weight perfect matching on a small general weighted graph". In the general case, this problem is hard. However, if the input size is small, up to $M \leq 20$, then DP with bitmask solution can be used.

The DP with bitmask solution for this problem is simple. The matching state is represented by a `bitmask`. We illustrate this with a small example when $M = 6$. We start with a state where nothing is matched yet, i.e. `bitmask=000000`. If item 0 and item 2 are matched, we can turn on two bits (bit 0 and bit 2) at the same time via this simple bit operation, i.e. `bitmask | (1 << 0) | (1 << 2)`, thus the state becomes `bitmask=000101`. Notice that index starts from 0 and counted from the right. If from this state, item 1 and item 5 are matched next, the state will become `bitmask=100111`. The perfect matching is obtained when the state is all '1's, in this case: `bitmask=111111`.

Although there are many ways to arrive at a certain state, there are only $O(2^M)$ distinct states. For each state, we record the minimum weight of previous matchings that must be done in order to reach this state. We want a perfect matching. First, we find one 'off' bit i using one $O(M)$ loop. Then, we find the best other 'off' bit j from `[`i`+1..`M`-1]` using another $O(M)$ loop and recursively match i and j. This check is again done using bit operation, i.e. we check `if (!(bitmask & (1 << i)))`—and similarly for j. This algorithm runs in $O(M \times 2^M)$. In problem UVa 10911, $M = 2N$ and $2 \leq N \leq 8$, so this DP with bitmask solution is feasible. For more details, please study the code.

Source code: `ch8_02_UVa10911.cpp/java`

In this subsection, we have shown that DP with bitmask technique can be used to solve small instances ($M \leq 20$) of matching on general graph. In general, bitmask technique allows us to represent a small set of up to ≈ 20 items. The programming exercises in this section contain more examples when bitmask is used as *one of the parameters* of the DP state.

Exercise 8.3.1.1: Show the required DP with bitmask solution if we have to deal with "Maximum Cardinality Matching on a small general graph ($V \leq 18$)".

Exercise 8.3.1.2*: Rewrite the code `ch8_02_UVa10911.cpp/java` with the `LSOne` trick shown in Section 8.2.1 to speed it up!
