Introduction to Programming (in C++)

Numerical methods I

Jordi Cortadella
Dept. of Computer Science, UPC

Living with floating-point numbers

- Standard normalized representation (sign + fraction + exponent):

$$0.15625_{10} = 0.00101_2 = 1.01 \times 2^{-3}$$

- Ranges of values:

single precision (float)	32 bits	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$
double precision (double)	64 bits	$\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$

Representations for: $-\infty$, $+\infty$, $+0$, -0, NaN (not a number)

- Be careful when operating with real numbers:

```
double x, y;
cin >> x >> y;          // 1.1  3.1
cout.precision(20);
cout << x + y << endl; // 4.2000000000000001776
```

Comparing floating-point numbers

- Comparisons:

```
a = b + c;
if (a - b == c) …          // may be false
```

- Allow certain tolerance for equality comparisons:

```
if (expr1 == expr2) …   // Wrong !

if (abs(expr1 - expr2) < 0.000001) …  // Ok !
```

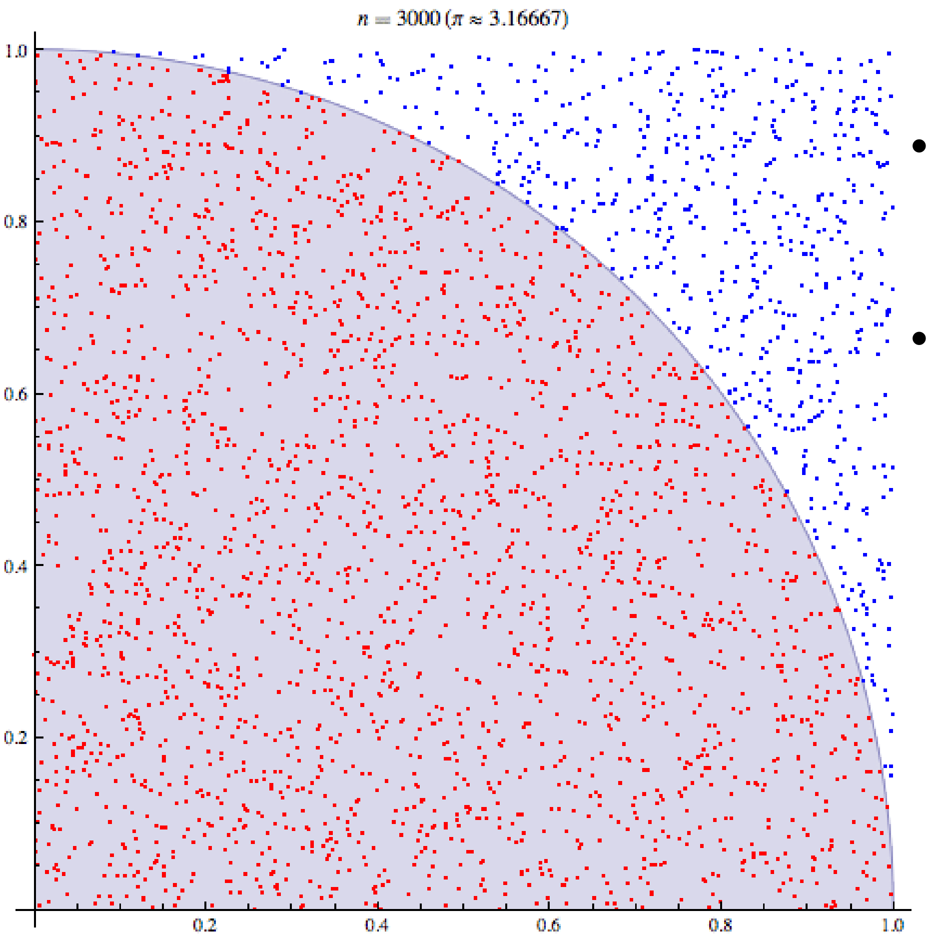
Monte Carlo methods

- Algorithms that use repeated generation of random numbers to perform numerical computations.

- The methods often rely on the existence of an algorithm that generates random numbers uniformly distributed over an interval.

- In C++ we can use **rand()**, that generates numbers in the interval **[0, RAND_MAX)**

Approximating π



- Let us pick a random point within the unit square.
- **Q:** What is the probability for the point to be inside the circle?
- **A:** The probability is $\pi/4$

<u>Algorithm:</u>

- Generate n random points in the unit square
- Count the number of points inside the circle (n_{in})
- Approximate $\pi/4 \approx n_{in}/n$

Approximating π

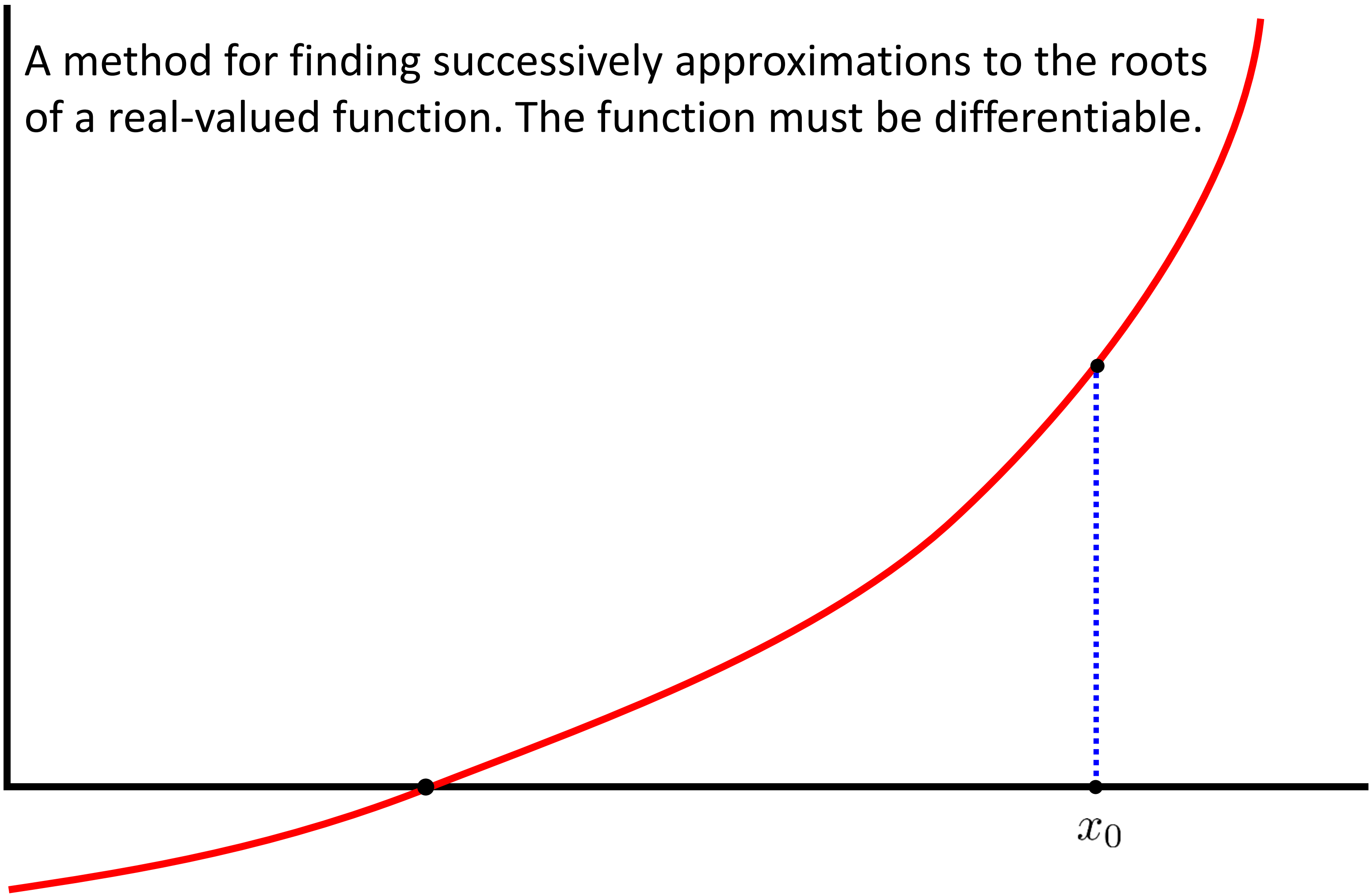
```
#include <cstdlib>

// Pre:  n is the number of generated points
// Returns an approximation of π using n random points

double approx_pi(int n) {
  int nin = 0;
  double randmax = double(RAND_MAX);
  for (int i = 0; i < n; ++i) {
    double x = rand()/randmax;
    double y = rand()/randmax;
    if (x*x + y*y < 1.0) nin = nin + 1;
  }
  return 4.0*nin/n;
}
```

Approximating π

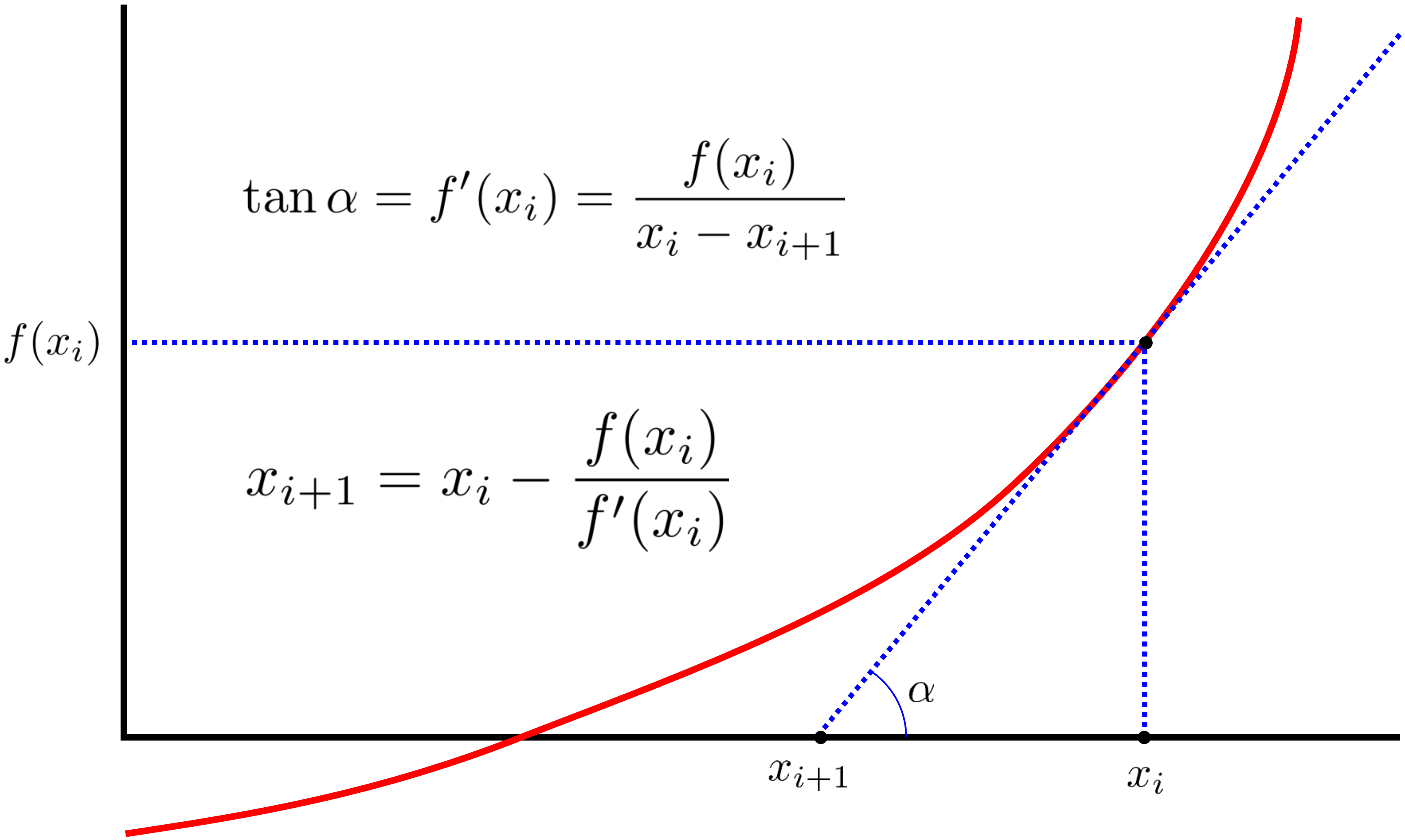
n	π
10	3.200000
100	3.120000
1,000	3.132000
10,000	3.171200
100,000	3.141520
1,000,000	3.141664
10,000,000	3.141130
100,000,000	3.141692
1,000,000,000	3.141604

The Newton-Raphson method

A method for finding successively approximations to the roots of a real-valued function. The function must be differentiable.



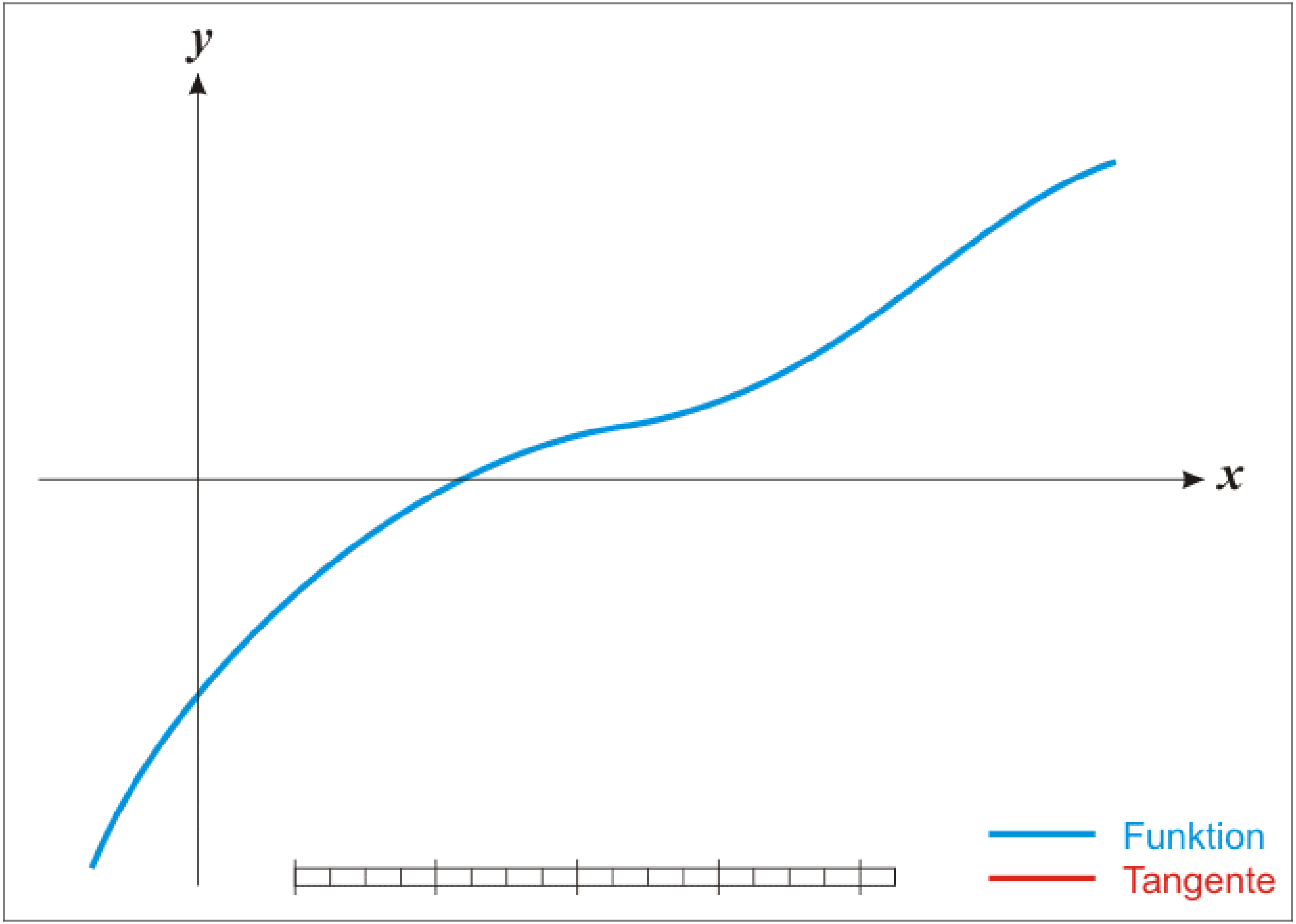
The Newton-Raphson method



$$\tan\alpha = f'(x_i) = \frac{f(x_i)}{x_i - x_{i+1}}$$

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

The Newton-Raphson method



source: http://en.wikipedia.org/wiki/Newton's_method

Square root (using Newton-Raphson)

- Calculate $x = \sqrt{a}$

- Find the zero of the following function:

$$f(x) = x^2 - a$$

where $f'(x) = 2x$

- Recurrence:

$$x_{i+1} = x_i - \frac{x_i^2 - a}{2x_i} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right)$$

Square root (using Newton-Raphson)

```
// Pre: a ≥ 0
// Returns x such that |x²-a| < ε

double square_root(double a) {

    double x = 1.0; // Makes an initial guess

    // Iterates using the Newton-Raphson recurrence
    while (abs(x*x – a) >= epsilon) x = 0.5*(x + a/x);

    return x;
}
```

Square root (using Newton-Raphson)

- Example: square_root(1024.0)

x
1.00000000000000000000
512.50000000000000000000
257.24902439024390332634
130.61480157022683101786
69.227324054488946103447
42.009585631008270922848
33.192487416854376647279
32.021420905000240964000
32.000007164815897908738
32.000000000000802913291

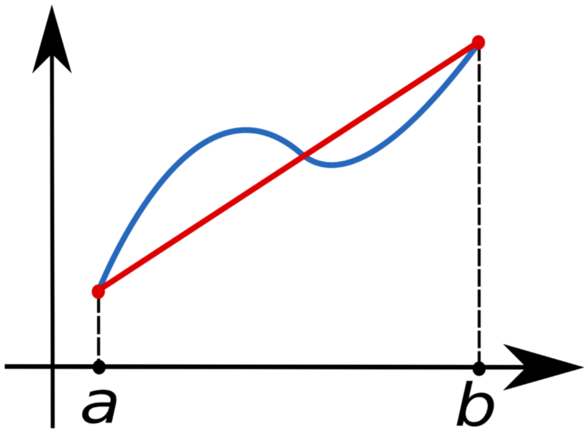
Approximating definite integrals

- There are various methods to approximate a definite integral:

$$\int_a^b f(x)dx.$$

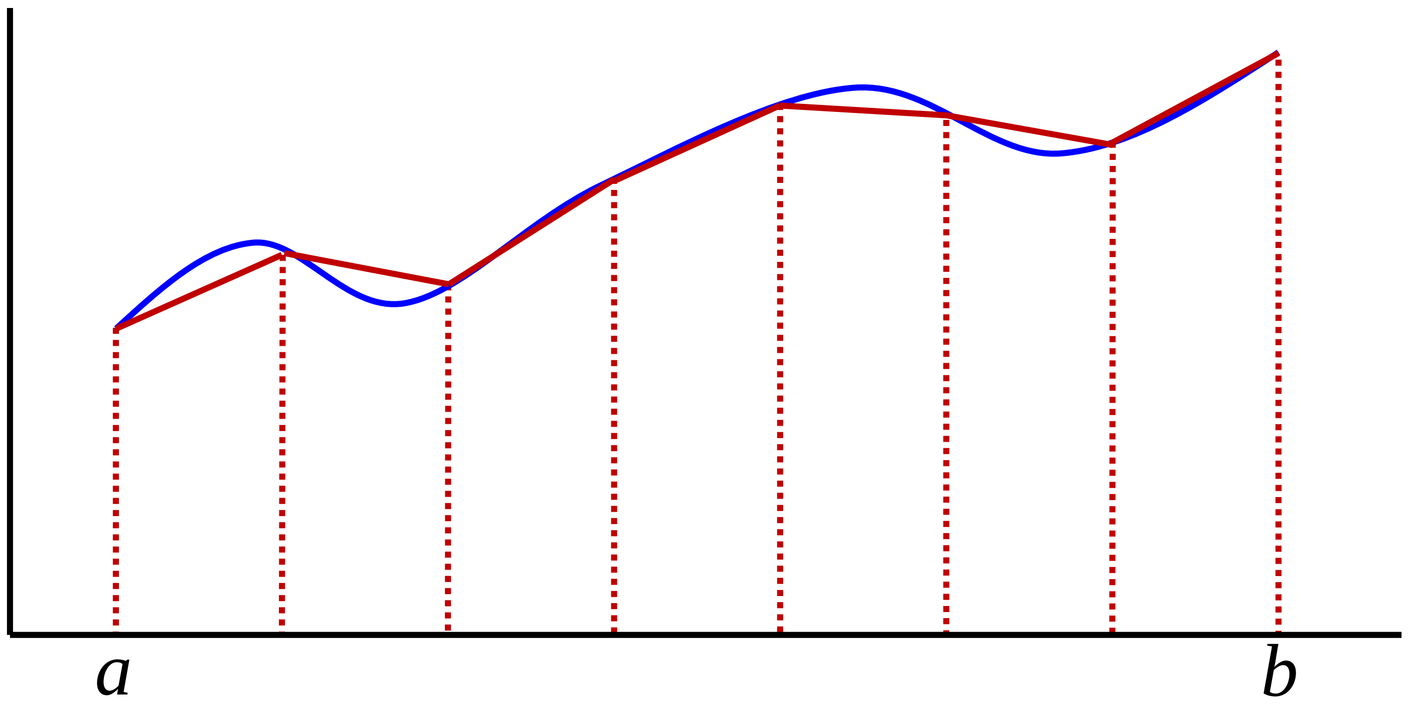
- The trapezoidal method approximates the area with a trapezoid:

$$\int_a^b f(x)dx \approx (b-a)\left(\frac{f(a)+f(b)}{2}\right)$$

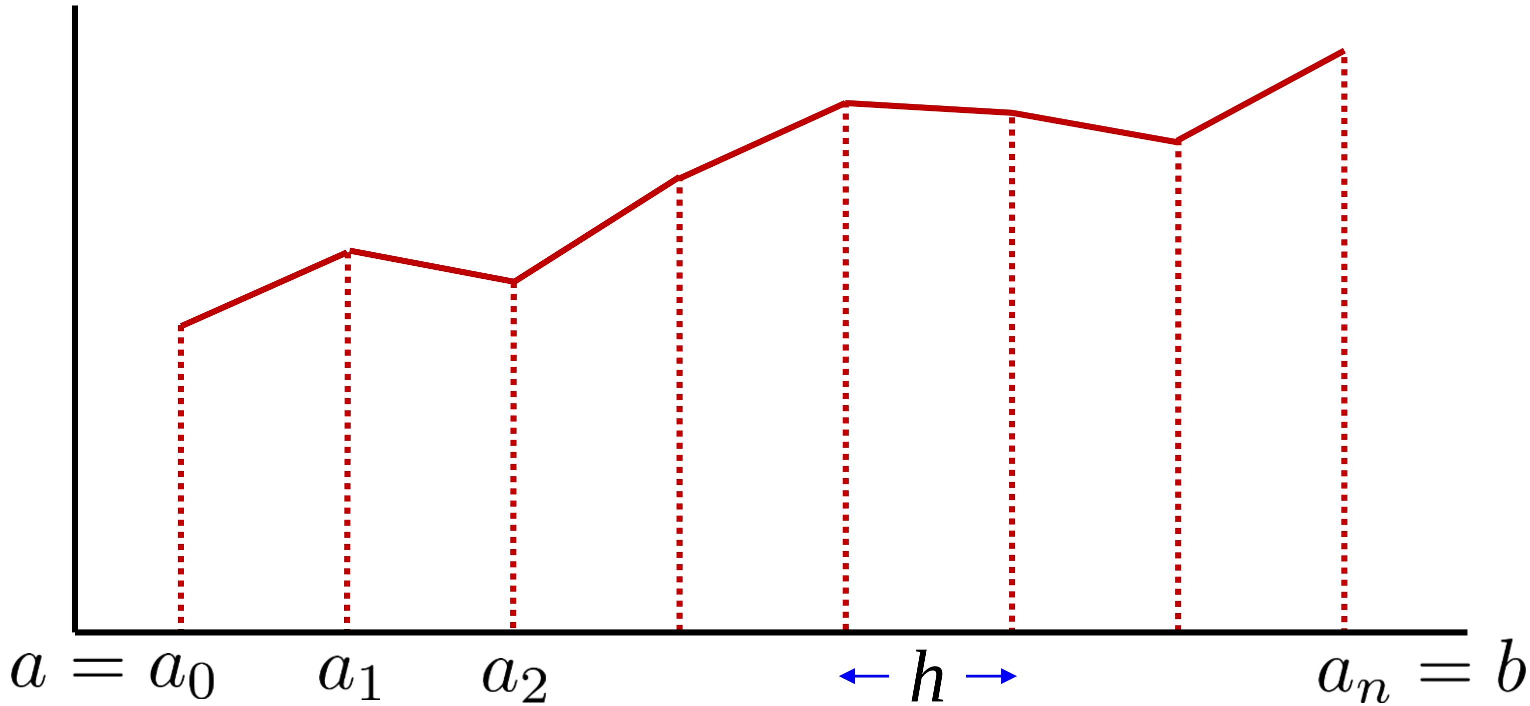


Approximating definite integrals

- The approximation is better if several intervals are used:



Approximating definite integrals



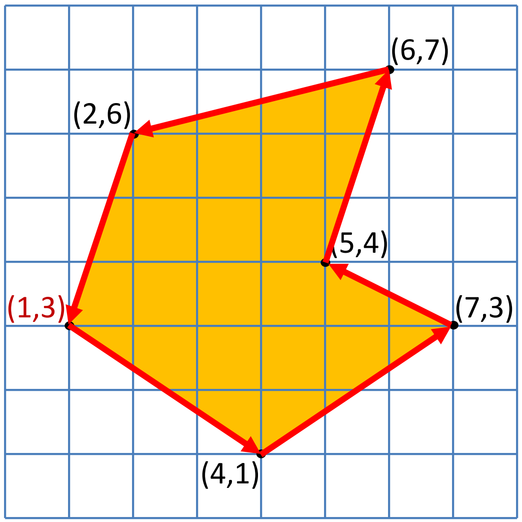
$$S = \sum_{i=0}^{n-1} h \cdot \frac{f(a_i) + f(a_{i+1})}{2}$$

$$= \frac{h}{2}\left(f(a) + f(b) + 2\sum_{i=1}^{n-1} f(a_i)\right)$$

Approximating definite integrals

```
// Pre:  b >= a, n > 0
// Returns an approximation of the definite integral
// of f between a and b using n intervals.

double integral(double a, double b, int n) {

  double h = (b - a)/n;

  double s = 0;
  for (int i = 1; i < n; ++i) s = s + f(a + i*h);

  return (f(a) + f(b) + 2*s)*h/2;
}
```

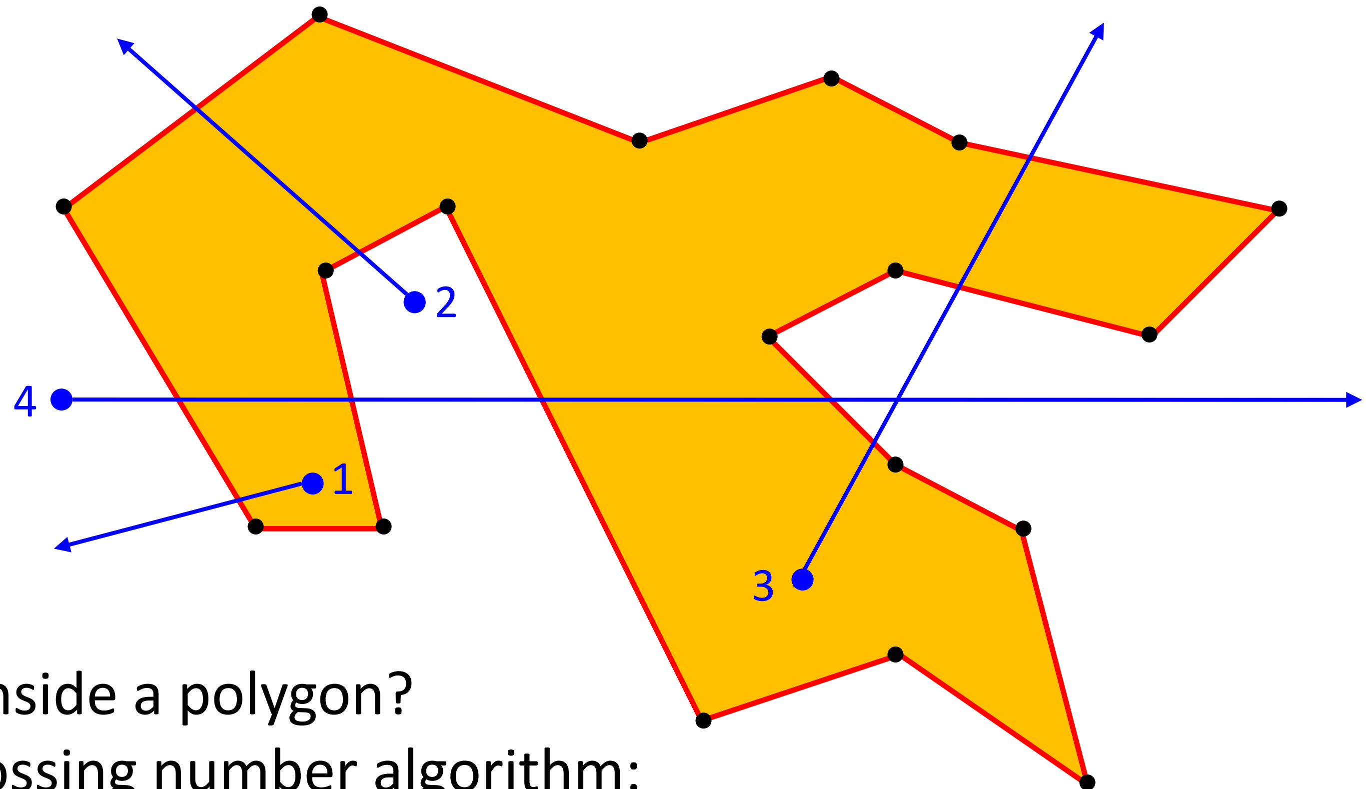
Representation of polygons



- A polygon can be represented by a sequence of vertices.
- Two consecutive vertices represent an edge of the polygon.
- The last edge is represented by the first and last vertices of the sequence.

```
Vertices:  (1,3) (4,1) (7,3) (5,4) (6,7) (2,6)

Edges: (1,3)-(4,1)-(7,3)-(5,4)-(6,7)-(2,6)-(1,3)
```

Point in polygon



- Is a point inside a polygon?
- Use the crossing number algorithm:
  - Draw a ray from the point
  - Count the number of crossing edges:
    - even → outside, odd → inside.

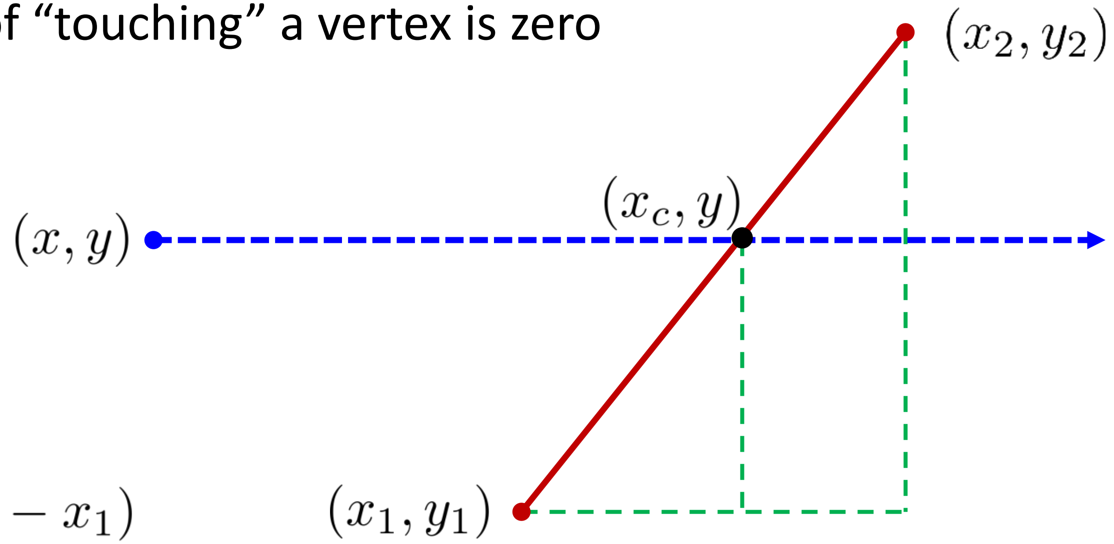
Point in polygon

```
// A data structure to represent a point
struct Point {
    double x;
    double y;
};

// A data structure to represent a polygon
// (an ordered set of vertices)
typedef vector<Point> Polygon;
```

Point in polygon

- Use always the horizontal ray increasing x (y is constant)

- Assume that the probability of “touching” a vertex is zero



$$\frac{y_2 - y_1}{x_2 - x_1} = \frac{y - y_1}{x_c - x_1}$$

$$\Downarrow$$

$$x_c = x_1 + \frac{y - y_1}{y_2 - y_1}(x_2 - x_1)$$

- The ray crosses the segment if:
  - y is between y_1 and y_2 and
  - $x_c > x$

Point in polygon

```
// Returns true if point q is inside polygon P,
// and false otherwise.

bool  in_polygon(const Polygon& P, const Point& q) {
  int nvert = P.size();
  int src = nvert - 1;
  int ncross = 0;

  // Visit all edges of the polygon
  for (int dst = 0; dst < nvert; ++dst) {
    if (cross(P[src], P[dst], q) ++ncross;
    src = dst;
  }

  return ncross%2 == 1;
}
```

Point in polygon

```
// Returns true if the horizontal ray generated from q by
// increasing x crosses the segment defined by p1 and p2,
// and false otherwise.

bool cross(const Point& p1, const Point& p2, const Point& q) {

  // Check whether q.y is between p1.y and p2.y
  if ((p1.y > q.y) == (p2.y > q.y)) return false;

  // Calculate the x coordinate of the crossing point
  double xc = p1.x + (q.y - p1.y)*(p2.x - p1.x)/(p2.y - p1.y);
  return xc > q.x;
}
```

Cycles in permutations

- Let P be a vector of n elements containing a permutation of the numbers 0…n-1.
- The permutation contains cycles and all elements are in some cycle.

i	0	1	2	3	4	5	6	7	8	9
$P[i]$	**6**	**4**	**2**	**8**	**0**	**7**	**9**	**3**	**5**	**1**

```
Cycles:
    (0 6 9 1 4)
    (2)
    (3 8 5 7)
```

- Design a program that writes all cycles of a permutation.

Cycles in permutations

i	0	1	2	3	4	5	6	7	8	9
$P[i]$	**6**	**4**	**2**	**8**	**0**	**7**	**9**	**3**	**5**	**1**
$visited[i]$	✓	✓			✓		✓			✓

- Use an auxiliary vector (visited) to indicate the elements already written.
- After writing one permutation, the index returns to the first element.
- After writing one permutation, find the next non-visited element.

Cycles in permutations

```
// Pre:  P is a vector with a permutation of 0..n-1
// Post: The cycles of the permutation have been printed in cout

void print_cycles(const vector<int>& P) {
  int n = P.size();
  vector<bool> visited(n, false);

  int i = 0;
  while (i < n) {

    // All the cycles containing 0..i-1 have been written
    bool cycle = false;
    while (not visited[i]) {
      if (not cycle) cout << '(';
      else cout << ' '; // Not the first element
      cout << i;
      cycle = true;
      visited[i] = true;
      i = P[i];
    }
    if (cycle) cout << ')' << endl;

    // We have returned to the beginning of the cycle
    i = i + 1;
  }
}
```

Taylor and McLaurin series

- Many functions can be approximated by using Taylor or McLaurin series, e.g.:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(0)}{n!} x^n = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f^{(3)}(0)}{3!}x^3 + \cdots + \frac{f^{(n)}(0)}{n!}x^n + \cdots$$

- Example: sin x

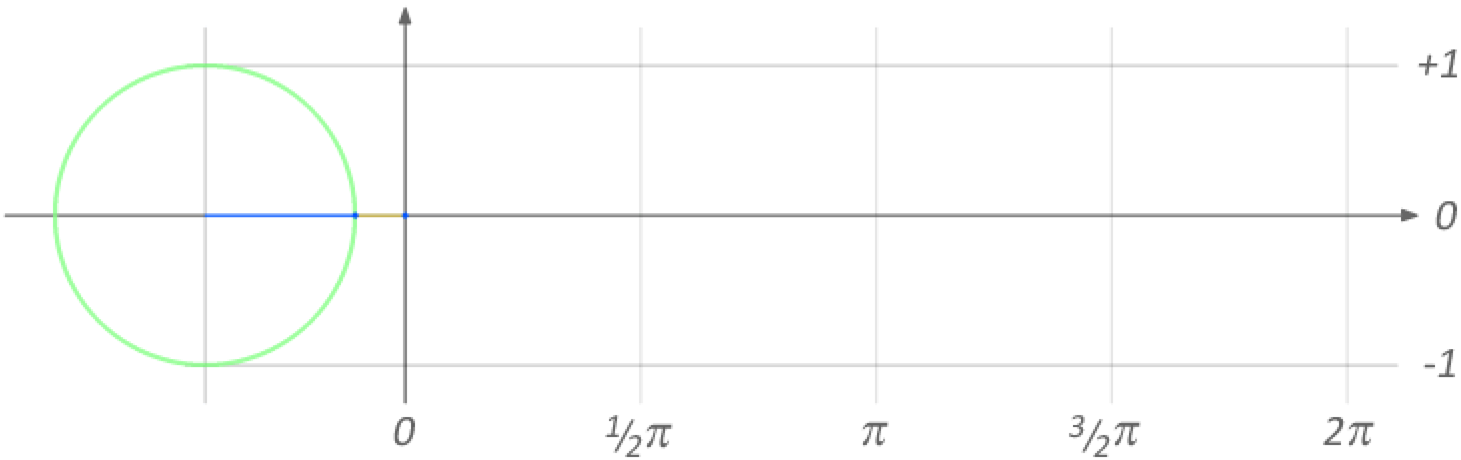
$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

Calculating *sin x*

- McLaurin series:

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

- It is a periodic function (period is 2π)



- Convergence improves as *x* gets closer to zero

Calculating *sin x*

- Reducing the computation to the (-2π,2π) interval:

$$k = \left\lfloor \frac{x}{2\pi} \right\rfloor, \qquad \sin x = \sin(x - 2k\pi).$$

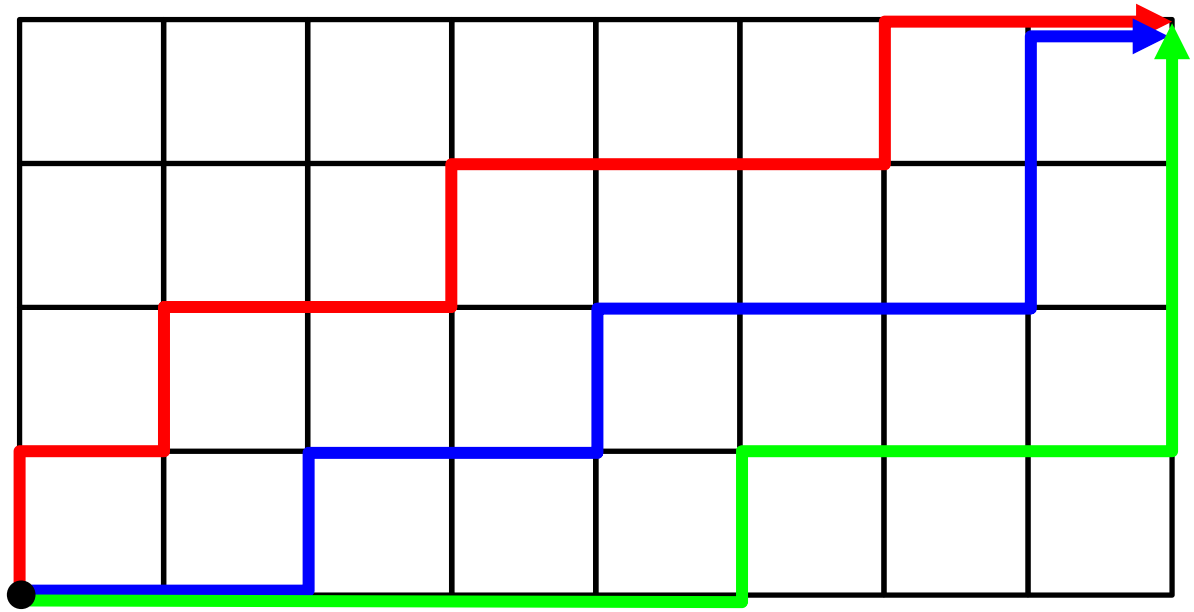
- Incremental computation of terms:

$$t_i = \frac{(-1)^i x^{2i+1}}{(2i+1)!}, \quad t_{i+1} = \frac{(-1)^{i+1} x^{2i+3}}{(2i+3)!} = -t_i \cdot \frac{x^2}{(2i+2)(2i+3)}$$

Calculating *sin x*

```
#include <cmath>

// Returns an approximation of sin x.
double sin_approx(double x) {
  int k = int(x/(2*M_PI));
  x = x - 2*k*M_PI; // reduce to the (-2π,2π) interval
  double term = x;
  double x2 = x*x;
  int d = 1;
  double sum = term;

  while (abs(term) >= 1e-8) {
    term = -term*x2/((d+1)*(d+2));
    sum = sum + term;
    d = d + 2;
  }

  return sum;
}
```

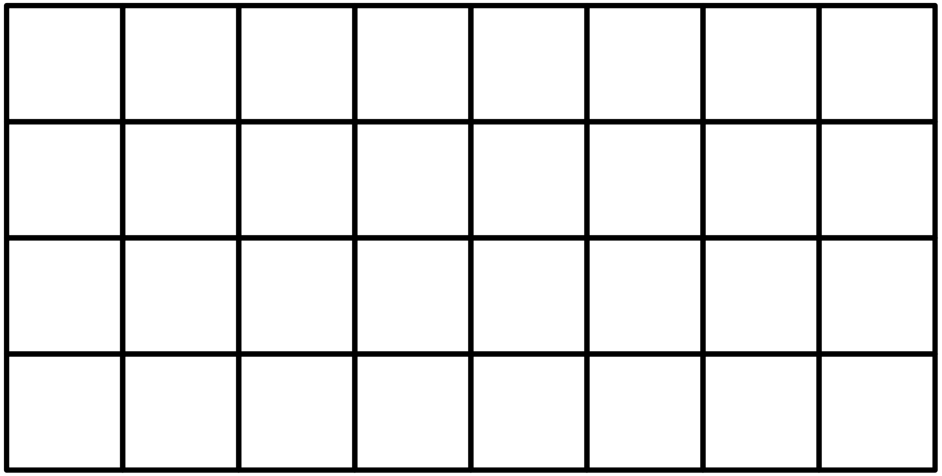
Lattice paths



We have an $n \times m$ grid.

How many different routes are there from the bottom left corner to the upper right corner only using right and up moves?

Lattice paths



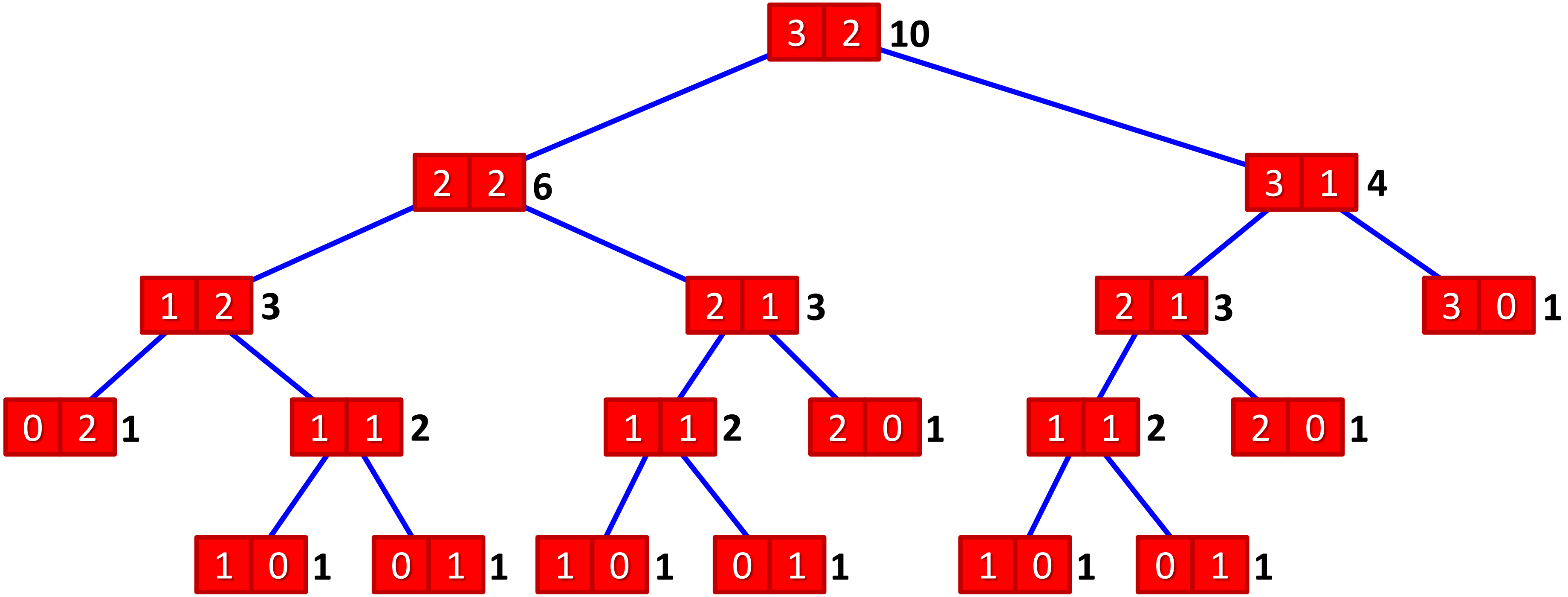
Some properties:

- paths(n, 0) = paths(0, m) = 1
- paths(n, m) = paths(m, n)
- If n > 0 and m > 0:
  paths(n, m) = paths(n-1, m) + paths(n, m-1)

Lattice paths

```
// Pre: n and m are the dimensions of a grid
//      (n ≥ 0 and m ≥ 0).
// Returns the number of lattice paths in the grid.

int paths(int n, int m) {
  if (n == 0 or m == 0) return 1;
  return paths(n – 1, m) + paths(n, m – 1);
}
```

Lattice paths



- How large is the tree (cost of the computation)?
- Observation: many computations are repeated

Lattice paths

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	1	1	1	1	1	1	1	1	1
1	1	2	3	4	5	6	7	8	9
2	1	3	6	10	15	21	28	36	45
3	1	4	10	20	35	56	84	120	165
4	1	5	15	35	70	126	210	330	495
5	1	6	21	56	126	252	462	792	1287
6	1	7	28	84	210	462	924	1716	**3003**

$$M[i][0] = M[0][i] = 1$$

$$M[i][j] = M[i-1][j] + M[i][j-1], \qquad for\; i > 0, j > 0$$

Lattice paths

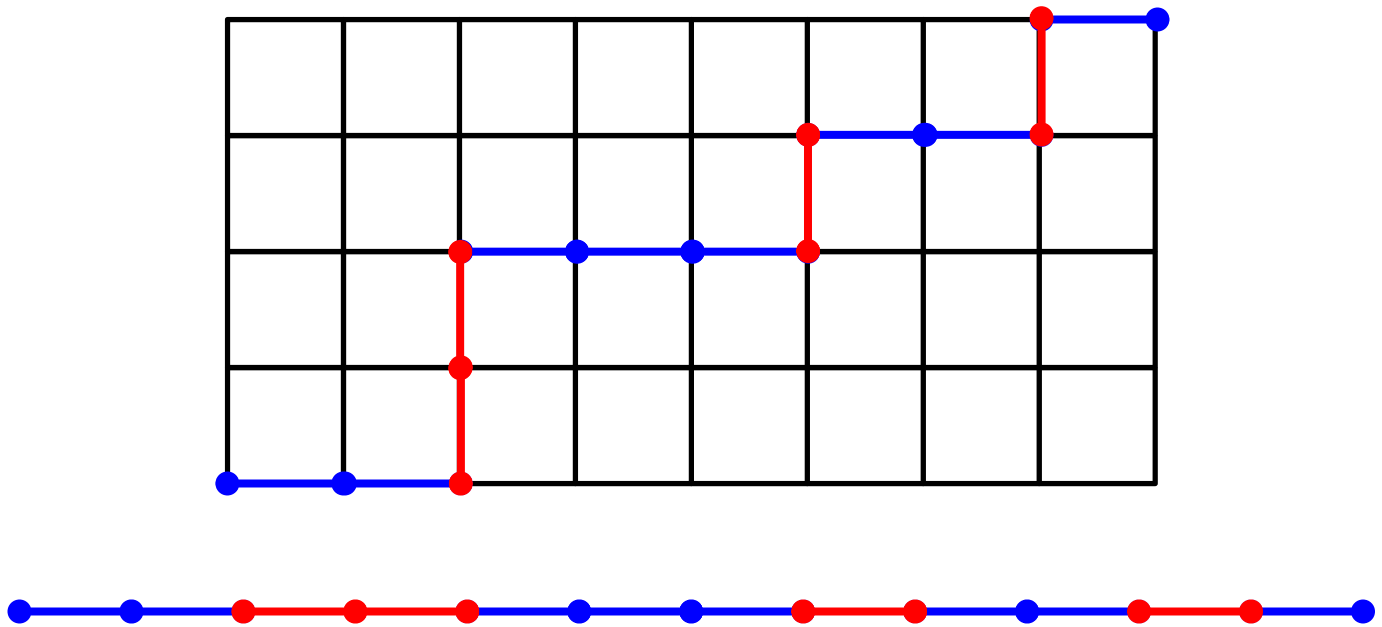
```
// Pre: n and m are the dimensions of a grid
//      (n ≥ 0 and m ≥ 0).
// Returns the number of lattice paths in the grid.

int paths(int n, int m) {
  vector< vector<int> > M(n + 1, vector<int>(m + 1));
  // Initialize row 0
  for (int j = 0; j <= m; ++j) M[0][j] = 1;

  // Fill the matrix from row 1
  for (int i = 1; i <= n; ++i) {
    M[i][0] = 1;
    for (int j = 1; j <= m; ++j) {
      M[i][j] = M[i - 1][j] + M[i][j - 1];
    }
  }
  return M[n][m];
}
```

Lattice paths

	0	1	2	3	4	5	6	7	8
0	$\binom{0}{0}$	$\binom{1}{0}$	$\binom{2}{0}$	$\binom{3}{0}$	$\binom{4}{0}$	$\binom{5}{0}$	$\binom{6}{0}$	$\binom{7}{0}$	$\binom{8}{0}$
1	$\binom{1}{1}$	$\binom{2}{1}$	$\binom{3}{1}$	$\binom{4}{1}$	$\binom{5}{1}$	$\binom{6}{1}$	$\binom{7}{1}$	$\binom{8}{1}$	$\binom{9}{1}$
2	$\binom{2}{2}$	$\binom{3}{2}$	$\binom{4}{2}$	$\binom{5}{2}$	$\binom{6}{2}$	$\binom{7}{2}$	$\binom{8}{2}$	$\binom{9}{2}$	$\binom{10}{2}$
3	$\binom{3}{3}$	$\binom{4}{3}$	$\binom{5}{3}$	$\binom{6}{3}$	$\binom{7}{3}$	$\binom{8}{3}$	$\binom{9}{3}$	$\binom{10}{3}$	$\binom{11}{3}$
4	$\binom{4}{4}$	$\binom{5}{4}$	$\binom{6}{4}$	$\binom{7}{4}$	$\binom{8}{4}$	$\binom{9}{4}$	$\binom{10}{4}$	$\binom{11}{4}$	$\binom{12}{4}$
5	$\binom{5}{5}$	$\binom{6}{5}$	$\binom{7}{5}$	$\binom{8}{5}$	$\binom{9}{5}$	$\binom{10}{5}$	$\binom{11}{5}$	$\binom{12}{5}$	$\binom{13}{5}$
6	$\binom{6}{6}$	$\binom{7}{6}$	$\binom{8}{6}$	$\binom{9}{6}$	$\binom{10}{6}$	$\binom{11}{6}$	$\binom{12}{6}$	$\binom{13}{6}$	$\boldsymbol{\binom{14}{6}}$

$$M[i][j] = \binom{i+j}{i} = \binom{i+j}{j}$$

Lattice paths



- In a path with *n+m* segments, select *n* segments to move right (or *m* segments to move up)
- Subsets of *n* elements out of *n+m*

Lattice paths

Calculating $\binom{n}{k} = \dfrac{n!}{k!(n-k)!}$

– Naïve method: $2\mathrm{n}$ multiplications and 1 division (potential overflow problems with $\mathrm{n}!$)

– Recursion:

$$\binom{n}{0} = \binom{n}{n} = 1$$

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} = \frac{n-k+1}{k}\binom{n}{k-1}$$

$$= \frac{n}{n-k}\binom{n-1}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Lattice paths

```
// Pre: n and m are the dimensions of a grid
//      (n ≥ 0 and m ≥ 0).
// Returns the number of lattice paths in the grid.

int paths(int n, int m) {
  return combinations(n + m, n);
}


// Pre: n ≥ k ≥ 0
// Returns the number of k-combinations of a set of
// n elements.

int combinations(int n, int k) {
  if (k == 0) return 1;
  return n*combinations(n – 1, k – 1)/k;
}
```

Lattice paths

Computational cost:

– Recursive version: $O\left(\binom{n+m}{m}\right)$

– Matrix version: $O(n \cdot m)$

– Combinations: $O(m)$

Lattice paths

- How about counting paths in a 3D grid?
- And in a k-D grid?