Introduction to Programming (in C++)

Numerical algorithms

Jordi Cortadella, Ricard Gavaldà, Fernando Orejas
Dept. of Computer Science, UPC

Product of polynomials

- Given two polynomials on one variable and real coefficients, compute their product

  (we will decide later how we represent polynomials)

- Example: given $x^2 + 3x - 1$ and $2x - 5$, obtain

  $2x^3 - 5x^2 + 6x^2 - 15x - 2x + 5 = 2x^3 + x^2 - 17x + 5$

Product of polynomials

- Key point:

  Given $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$
  and $q(x) = b_m x^m + b_{m-1} x^{m-1} + \ldots + b_1 x + b_0$,

  what is the coefficient c_i of x^i in $(p*q)(x)$?

- We obtain x^{i+j} whenever we multiply $a_i\, x^i \cdot b_j\, x^j$

- Idea: for every i and j, add $a_i \cdot b_j$ to the (i+j)-th coefficient of the product polynomial.

Product of polynomials

- Suppose we represent a polynomial of degree *n* by a vector of size *n+1*.

  That is, v[0..n] represents the polynomial

  $$v[n]\ x^n + v[n-1]\ x^{n-1} + \ldots + v[1]\ x + v[0]$$

- We want to make sure that v[v.size() - 1] $\neq$ 0 so that degree(v) = v.size() - 1

- The only exception is the constant-0 polynomial. We'll represent it by a vector of size 0.

Product of polynomials

```
typedef vector<double> Polynomial;

// Pre:  none
// Post: returns p*q
Polynomial product(const Polynomial& p,
                   const Polynomial& q);
```

Product of polynomials

```
Polynomial product(const Polynomial& p, const Polynomial& q) {

    // Special case for a polynomial of size 0
    if (p.size() == 0 or q.size() == 0) return Polynomial(0);
    else {
        int deg = p.size() – 1 + q.size() - 1; // degree of p*q
        Polynomial r(deg + 1, 0);
        for (int i = 0; i < p.size(); ++i) {
            for (int j = 0; j < q.size(); ++j) {
                r[i + j] = r[i + j] + p[i]*q[j];
            }
        }
        return r;
    }
}

// Invariant (of the outer loop): r = product p[0..i-1]*q
// (we have used the coefficients p[0] … p[i-1])
```

Sum of polynomials

- Note that over the real numbers,

  $$\text{degree}(p*q) = \text{degree}(p) + \text{degree}(q)$$

  (except if p = 0 or q = 0).

  So we know the size of the result vector from the start.

- This is not true for the polynomial sum, e.g.

  $$\text{degree}((x + 5) + (-x - 1)) = 0$$

Sum of polynomials

```
// Pre:  none
// Post: returns p+q
Polynomial sum(const Polynomial& p, const Polynomial& q);
    int maxdeg = max(p.size(), q.size()) - 1;
    int deg = -1;
    Polynomial r(maxdeg + 1, 0);

    // Inv r[0..i-1] = (p+q)[0..i-1] and
    // deg = largest j s.t. r[j] != 0 (or -1 if none exists)
    for (int i = 0; i <= maxdeg; ++i) {
        if (i >= p.size()) r[i] = q[i];
        else if (i >= q.size()) r[i] = p[i];
        else r[i] = p[i] + q[i];
        if (r[i] != 0) deg = i;
    }

    Polynomial rr(deg + 1);
    for (int i = 0; i <= deg; ++i) rr[i] = r[i];
    return rr;
}
```

Sum of sparse vectors

- In some cases, problems must deal with sparse vectors or matrices (most of the elements are zero).

- Sparse vectors and matrices can be represented more efficiently by only storing the non-zero elements. For example, a vector can be represented as a vector of pairs (index, value), sorted in ascending order of the indices.

- Example:

  `[0,0,1,0,-3,0,0,0,2,0,0,4,0,0,0]`

  can be represented as

  `[(2,1),(4,-3),(8,2),(11,4)]`

Sum of sparse vectors

- Design a function that calculates the sum of two sparse vectors, where each non-zero value is represented by a pair (index, value):

```
struct Pair {
    int index;
    int value;
}

typedef vector<Pair> SparseVector;
```

Sum of sparse vectors

```
// Pre:   --
// Post: returns v1+v2

SparseVector sparse_sum(const SparseVector& v1,
                        const SparseVector& v2);

// Inv:  p1 and p2 will point to the first
//       non-treated elements of v1 and v2.
//       vsum contains the elements of v1+v2 treated so far.
//       psum points to the first free location in vsum.
```

- Strategy:
  - Calculate the sum on a sufficiently large vector.
  - Copy the result on another vector of appropriate size.

Sum of sparse vectors

```
SparseVector sparse_sum(const SparseVector& v1, const SparseVector& v2) {
    int n1 = v1.size();
    int n2 = v2.size();
    SparseVector vsum(n1 + n2);

    int p1, p2, psum;
    p1 = p2 = psum = 0;
    while (p1 < n1 and p2 < n2) {
        if (v1[p1].index < v2[p2].index) { // Element only in v1
            vsum[psum] = v1[p1]; ++p1; ++psum;
        }
        else if (v1[p1].index > v2[p2].index) { // Element only in v2
            vsum[psum] = v2[p2]; ++p2; ++psum;
        }
        else if (v1[p1].value + v2[p2].value != 0) { // Element in both
            vsum[psum].index = v1[p1].index;
            vsum[psum].value = v1[p1].value + v2[p2].value;
            ++p1; ++p2; ++psum;
        }
        else { // but do not store zeros
            ++p1; ++p2;
        }
    }
```

Sum of sparse vectors

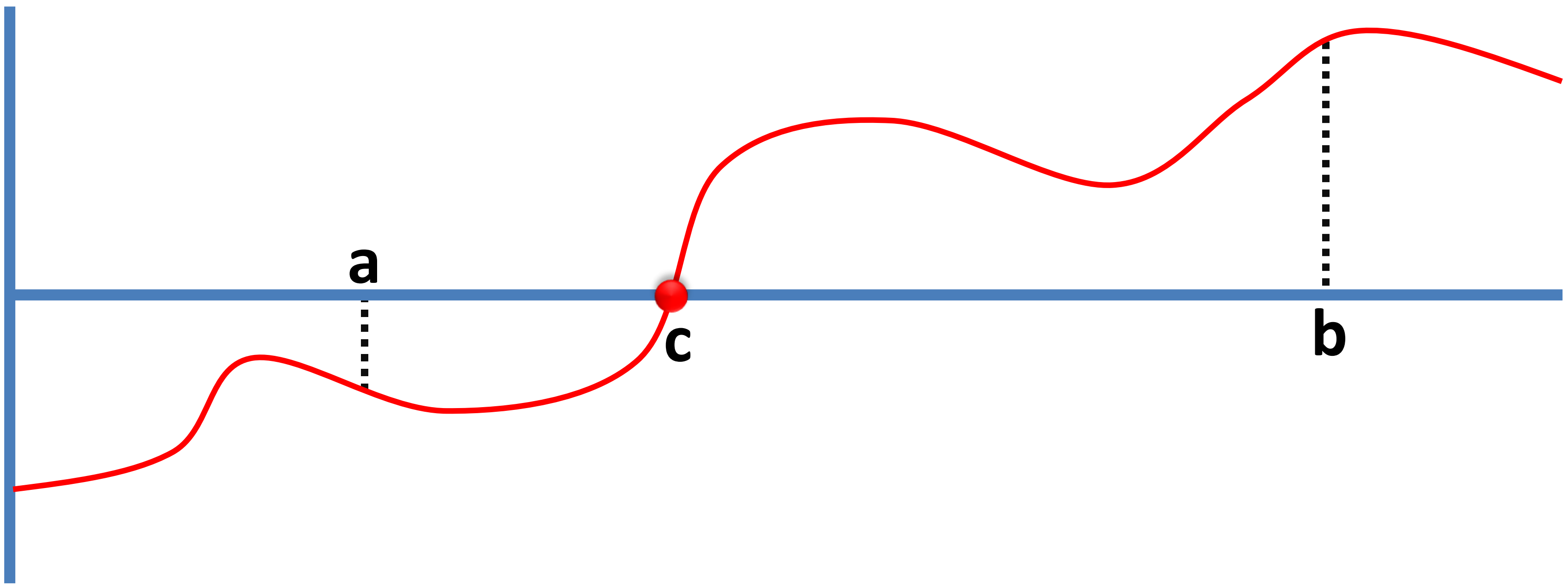
```
    // Copy the remaining elements of v1
    while (p1 < n1) {
        vsum[psum] = v1[p1];
        ++p1; ++psum;
    }

    // Copy the remaining elements of v2
    while (p2 < n2) {
        vsum[psum] = v2[p2];
        ++p2; ++psum;
    }

    // Create and return the result
    SparseVector result(psum);
    for (int i = 0; i < psum; ++i) result[i] = vsum[i];
    return result;
}
```

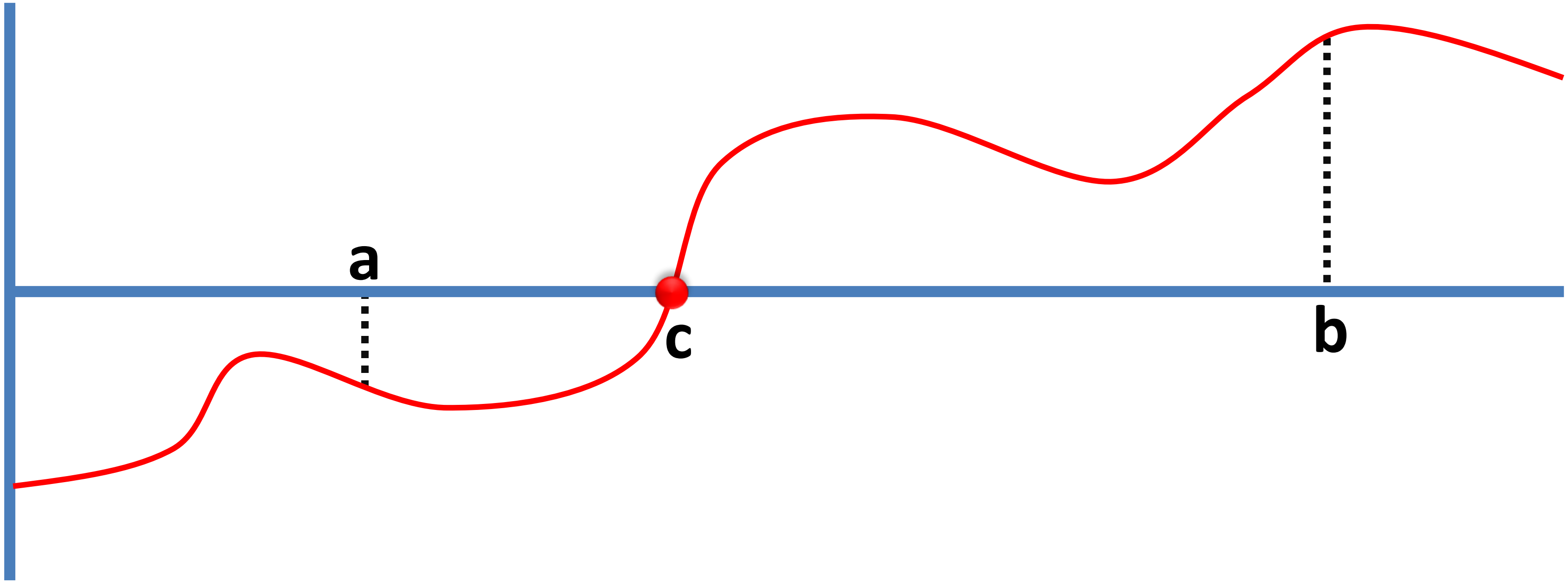
Root of a continuous function

- **Bolzano's theorem**:
  Let *f* be a real-valued continuous function.
  Let *a* and *b* be two values such that $a < b$ and $f(a) \cdot f(b) < 0$.
  Then, there is a value $c \in [a,b]$ such that $f(c)=0$.



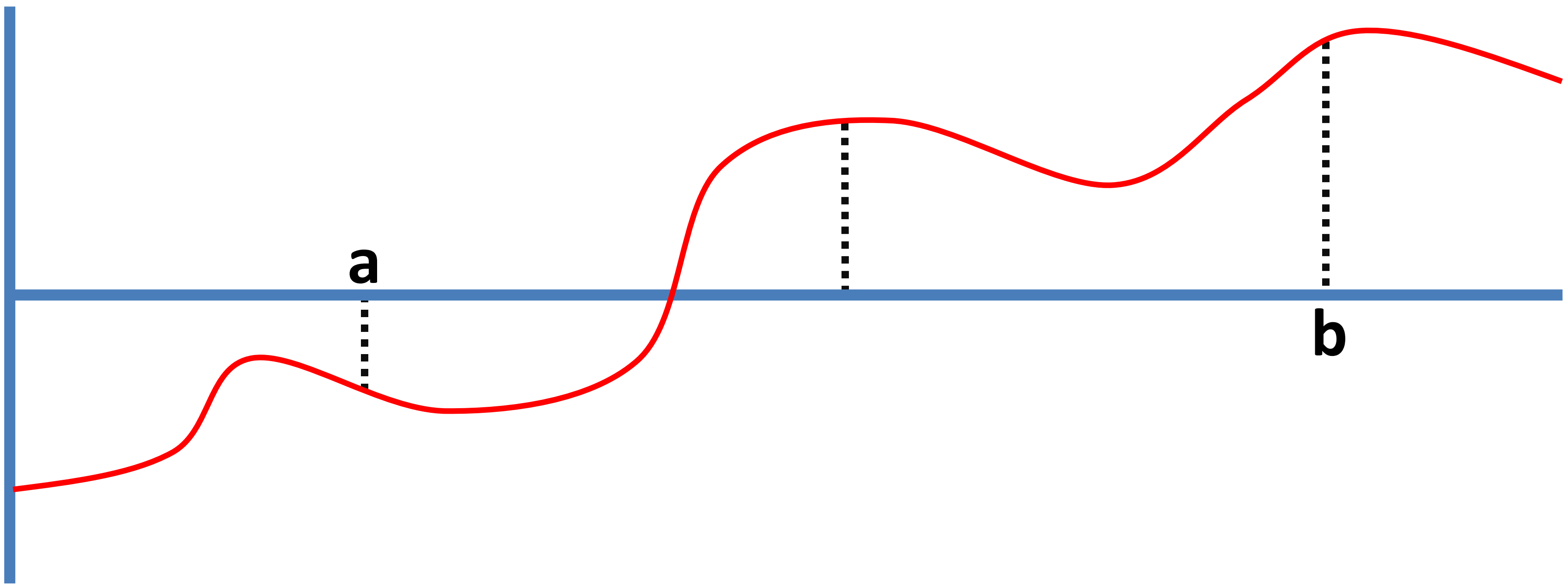
Root of a continuous function

- Design a function that finds a root of a continuous function *f* in the interval *[a, b]* assuming the conditions of Bolzano's theorem are fulfilled. Given a precision (ε), the function must return a value *c* such that the root of *f* is in the interval *[c, c+ε].*



Root of a continuous function

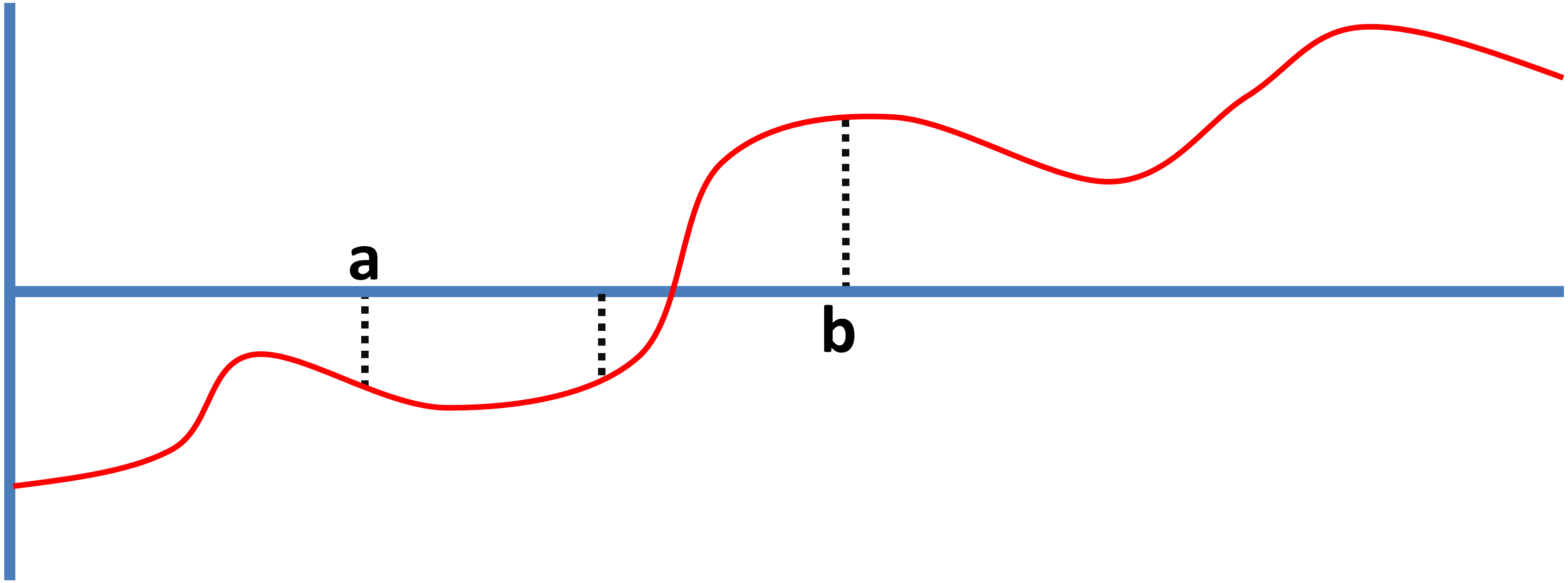
- Strategy: narrow the interval *[a, b]* by half, checking whether the value of *f* in the middle of the interval is positive or negative. Iterate until the width of the interval is smaller ε.



Root of a continuous function

```
// Pre:  f is continuous, a < b and f(a)*f(b) < 0.
// Post: returns c∈[a,b] such that a root exists in
//       the interval [c,c+ε].

// Inv:  a root of f exists in the interval [a,b]
```



Root of a continuous function

```
double root(double a, double b, double epsilon) {
    while (b – a > epsilon) {
        double c = (a + b)/2;
        if (f(a)*f(c) <= 0) b = c;
        else a = c;
    }
    return a;
}
```

Root of a continuous function

```
// A recursive version

double root(double a, double b, double epsilon) {
    if (b – a <= epsilon) return a;
    double c = (a + b)/2;
    if (f(a)*f(c) <= 0) return root(a,c,epsilon);
    else return root(c,b,epsilon);
}
```

Barcode

- A barcode is an optical machine-readable representation of data. One of the most popular encoding systems is the UPC (Universal Product Code).

- A UPC code has 12 digits. Optionally, a check digit can be added.

3 800065 711135

Barcode

- The check digit is calculated as follows:
  1. Add the digits in odd-numbered positions (first, third, fifth, etc.) and multiply by 3.
  2. Add the digits in the even-numbered positions (second, fourth, sixth, etc.) to the result.
  3. Calculate the result modulo 10.
  4. If the result is not zero, subtract the result from 10.

- Example: 380006571113

  - $(3+0+0+5+1+1)*3 = 30$
  - $8+0+6+7+1+3 = 25$
  - $(30+25) \bmod 10 = 5$
  - $10 - 5 =$ **5**

Barcode

- Design a program that reads a sequence of 12-digit numbers that represent UPCs without check digits and writes the same UPCs with the check digit.

- Question: do we need a data structure to store the UPCs?

- Answer: no, we only need a few auxiliary variables.

Barcode

- The program might have a loop treating a UPC at each iteration. The invariant could be as follows:

```
// Inv: all the UPCs of the treated codes
//      have been written.
```

- At each iteration, the program could read the UPC digits and, at the same time, write the UPC and calculate the check digit. The invariant could be:

```
// Inv: all the treated digits have been
//      written. The partial calculation of
//      the check digit has been performed
//      based on the treated digits.
```

Barcode

```
int main() {

// Pre:  the input contains a sequence of UPCs without check digits.
// Post: the UPCs at the input have been written with their check digits.

    char c;
    while (cin >> c) {
        cout << c;
        int d = 3*(int(c) - int('0')); // first digit in an odd location
        for (int i = 2; i <= 12; ++i) {
            cin >> c;
            cout << c;
            if (i%2 == 0) d = d + int(c) - int('0');
            else d = d + 3*(int(c) - int('0'));
        }
        d = d%10;
        if (d > 0) d = 10 - d;
        cout << d << endl;
    }
}
```