## Modular arithmetic, evaluation, interpolation

In symbolic computation is essential representing the objects that we want to manipulate. For example, a polynomial can be represented by the list of its coefficients, but it can also be represented by the values it takes at a sufficient number of points, as we have seen in the FFT computation.

We are going to study a modular approach, of which "evaluation - interpolation" is a special case. In this approach we choose modules $m_i$, we perform calculations modulo each $m_i$ and we reconstruct the result using version of the Chinese remainder theorem. To ensure uniqueness and correctness of the result it is sufficient to choose a large number of $m_i$ (for example coprime and s.t. their product exceeds the final result).

This approach is especially useful when the coefficients in the intermediate calculations are much larger than those of the final result. For example when we compute the determinant of a polynomial matrix.

Example: Assume you want to compute the determinant of a $n \times n$ matrix where each entry is a polynomial of degree at most $d$. The first observation is that the pivot in the $i$-th row obtained with Gaussian elimination has degree roughly $2^i d$. This shows that the complexity of computing the determinant will not be polynomial in $n$.

Since the determinant is a polynomial of degree at most $nd$, it is sufficient to choose $nd+1$ distinct points, evaluate the matrix at those points, compute the $nd+1$ determinants of scalar matrices and then perform interpolation to obtain the determinant.

Another possibility one can choose modules in the form of polynomials $X - w^i$, where $w$ is a root of unity of sufficiently high order ($\approx 2dn$).

Problems of evaluation and interpolation are inverse to each other.

Statements: Let $R$ be a commutative ring, let $a_0, \dots, a_{n-1}$ be points in $R$

Multipoint evaluation: Given a polynomial $P$ in $R[X]$ with degree strictly less than $n$, compute the values $P(a_0), \dots, P(a_{n-1})$

Interpolation: Given $b_0, b_1, \dots, b_{n-1} \in R$, find a polynomial $P \in R[X]$ of degree strictly less than $n$ such that $P(a_0) = b_0, \dots, P(a_{n-1}) = b_{n-1}$

Remark: The interpolation problem has always a unique solution under the condition $a_i - a_j$ invertible in $R$ when $i \neq j$

Indeed, if $V = [a_i^{j-1}]_{i,j=1}^{n}$ is the Vandermonde matrix associated with $a_i$.

$V$ is a Vandermond matrix whose determinant is given by

$$\det(V) = \prod_{i<j} (a_j - a_i)$$

Then the interpolation problem translates in the system

$Vp = b$ where $b \in R^n$ is the vector of $b_i$

The system has a unique solution, since the condition

$a_i - a_j$ invertible

ensures that $\det(V)$ is invertible hence $V$ is also invertible

=> The naive algorithm for interpolation is therefore the unique solution of $V^{-1}b = p$ using Gaussian elimination in $O(n^3)$ operations in $R$ or in $O(n^\theta)$ operations where $2 < \theta < 3$ is the exponent of the fast matrix multiplication.

Similarly, the multipoint evaluation of $P$ at the points $a_i$ translates to the matrix-vector product $Vp$ where $p$ is the vector of coefficients of $P$. Thus it can be done in $O(n^2)$ arithmetic operations.

- We will see that Lagrange interpolation allows to solve the interpolation problem in $O(n^2)$.
- We will demonstrate that quasi-optimal complexity can be achieved for both multipoint evaluation and interpolation by using "divide and conquer" algorithms.
- Multipoint evaluation and interpolations are instances of the "multiple modulus problem" and the "Chinese remainder Theorem"
  - multiple modulus problem: Given a polynomial $P$ and polynomials $m_1, \dots, m_r \in R[x]$ s.t. $\deg P < n = \sum \deg(m_i)$, find the remainders $P \bmod m_1, \dots, P \bmod m_r$
  - Chinese remainder theorem: Given $b_1, \dots, b_r$, and $u_1, \dots, u_r$ in $R[x]$, with each $u_i$ monic, find a polynomial $P \in R[x]$ of degree strictly less than $n = \sum \deg(u_i)$, such that
    $$P \bmod u_1 = b_1, \dots, P \bmod u_r = b_r.$$

## Lagrange Interpolation

The Lagrange interpolation is a technique based on an explicit expression of the interpolating polynomial. Let $b_i$ be the values to interpolate and $a_i$ be the interpolation points satisfying the hypothesis

H: $a_i - a_j$ invertible in $R$ $\forall i \neq j$.

Then we can write $P$ in the form

$$P(X) = \sum_{i=0}^{n-1} b_i \prod_{\substack{0 \leq j \leq n-1 \\ j \neq i}} \frac{X - a_j}{a_i - a_j} \qquad (*)$$

Equality (*) is called Lagrange interpolation formula. To prove it we observe that $\forall i$ the product vanishes at $a_j$ $(j \neq i)$ and equals 1 for $i$. Set

$$A = \prod_{j} (X - a_j)$$

$$A_i = \prod_{j \neq i} (X - a_j) = \frac{A}{(X - a_i)} \quad \text{for } i = 0, \dots, n-1$$

$\Rightarrow A(a_i) = 0$ and $A_i(a_j) = 0$ $\forall j \neq i$

We can rewrite $P(X) = \sum_{i=0}^{n-1} b_i \frac{A_i(X)}{A_i(a_i)}$.

We have the following algorithm.

Input: $a_0, \dots, a_{n-1} \in A$ verifying H, and $b_0, \dots, b_{n-1}$ in $R$

Output: The unique polynomial $P(X) \in R[X]$, with $\deg P < n$ s.t. $P(a_i) = b_i$

$A = 1$
$P = 0$
for $i = 0, \dots, n-1$ do
  $A = A \cdot (X - a_i)$
end for
for $i = 0, \dots, n-1$ do
  $A_i = \frac{A}{(X - a_i)}$
  $q_i = A_i(a_i)$
  $P = P + b_i \frac{A_i}{q_i}$
end for
return $P$.

Proposition: Previous algorithm uses $O(n^2)$ arithmetic operations in $\mathbb{K}$

Proof: The multiplication of a polynomial of degree $d$ with one of degree 1, requires $C \cdot d$ operations for some constant $C$.
Hence computing $A$ requires $C \cdot (1+2+\dots+n-1) = O(n^2)$.
Next, each iteration in the second for-cycle takes linear time in $n$, which yields $O(n^2)$ since there are $n$ iterations.
Hence $O(n^2) + O(n^2) = O(n^2)$.

Fast algorithms (fast multipoint evaluation and fast interpolation)

The ideas: The fundamental idea behind the fast multipoint evaluation algorithm is to use "divide and conquer" technique.
To simplify, suppose that $n$ is even and split the set of points $a_0, a_1, \dots, a_{n-1}$ in two groups $\{a_0, \dots, a_{\frac{n}{2}-1}\}$, and $\{a_{\frac{n}{2}}, \dots, a_{n-1}\}$. Hence we can define two polynomials of degree strictly less than $\frac{n}{2}$ $P_0$, and $P_1$ such that

$$P_0(a_0) = P(a_0) \qquad P_1(a_{\frac{n}{2}}) = P(a_{\frac{n}{2}})$$
$$\vdots \qquad\qquad \vdots$$
$$P_0(a_{\frac{n}{2}-1}) = P(a_{\frac{n}{2}-1}) \qquad P_1(a_{n-1}) = P(a_{n-1})$$

To achieve this construction, we can make the following choice:
$P_0 = P \bmod (x-a_0) \cdot \dots \cdot (x-a_{\frac{n}{2}-1})$
$P_1 = P \bmod (x-a_{\frac{n}{2}}) \cdots (x-a_{n-1})$

Hence we reduce the computation from degree $n$ to 2 evaluations of degree $\frac{n}{2}$, plus two polynomial divisions with degrees at most $n$. This process is then iterated. In the end we obtain the values $P(a_i)$ since they can be expressed as $P \bmod (x-a_i)$

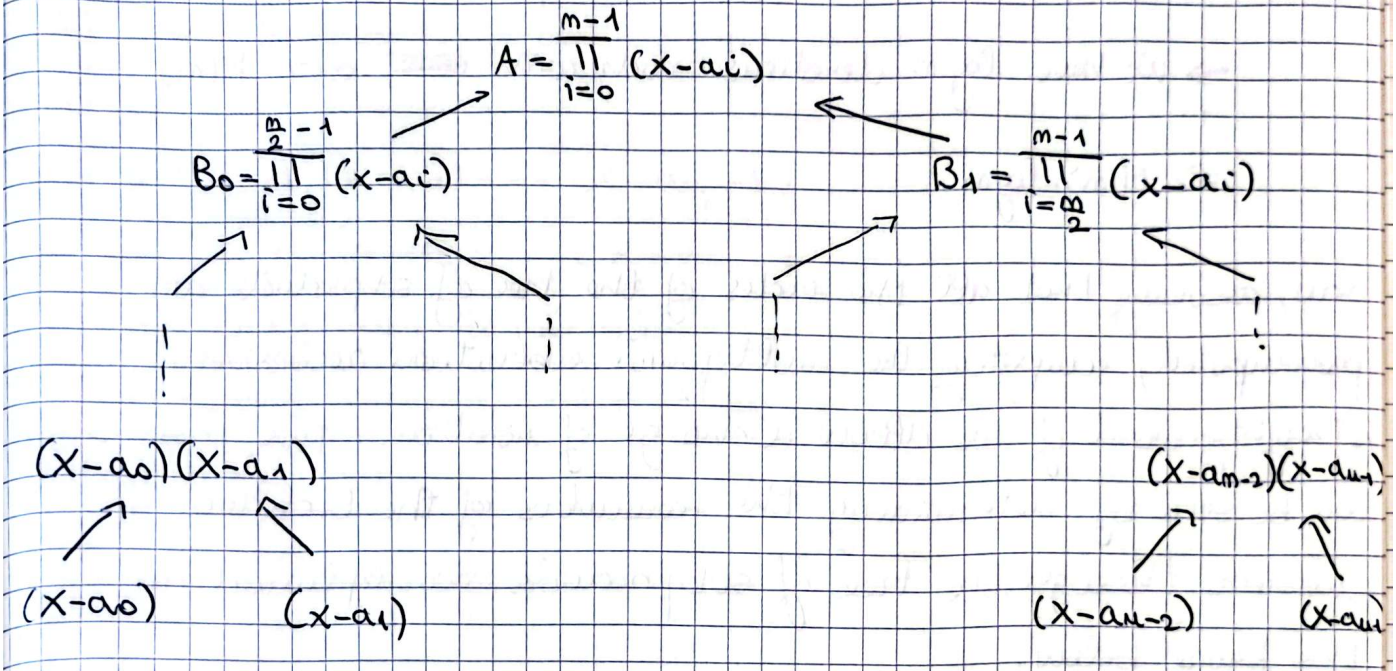
Note that the polynomials used in the euclidean divisions have to be precomputed. To do this, the idea is to stack them in a

binary tree structure, where the internal nodes are labeled by polynomials.

To simplify, we assume that the number of points is a power of 2, $n = 2^k$ (When this is not the case, the tree will be missing some internal nodes).

Definition of the tree $B$

- If $k=0$, $B$ reduces to a single node $x-a_0$
- Otherwise, let $\overline{B_0}$ and $\overline{B_1}$ be the trees associated respectively with the points $a_0, \dots, a_{2^{k-1}-1}$ and $a_{2^{k-1}}, \dots, a_{2^k-1}$.
Let $P_0$ and $P_1$ be the polynomials which are at the roots of $\overline{B_0}$ and $\overline{B_1}$. Then $\overline{B}$ is the tree whose root contains the product $P_0P_1$ and the nodes are $\overline{B_0}$ and $\overline{B_2}$.




The fast algorithm for the computation of the tree of the sub-product is the following:

Input: $a_0, \dots, a_{n-1}$ in $R$ with $n = 2^k$ ($k \in \mathbb{N}$)

Output: The Tree $\overline{B}$ of the subproducts associated to $a_i$.

Call the algorithm RecursiveTree($a_0, \dots, a_{n-1}$)

If $n=1$ then

return $x-a_0$

else

$\overline{B_0}$ = RecursiveTree $(a_0, \dots, a_{\frac{n}{2}-1})$

$\overline{B_1}$ = RecursiveTree $(a_{\frac{n}{2}}, \dots, a_{n-1})$

$A = P_0 P_1$ (where $P_0$ and $P_1$ are the roots of $\overline{B_0}$ and $\overline{B_1}$)

return the tree with root $A$ and nodes $\overline{B_0}$, $\overline{B_1}$.

**Proposition**: The algorithm above computes the polynomials in $\mathcal{B}$ in $O(M(n)\log n)$ arithmetic operations in $R$.

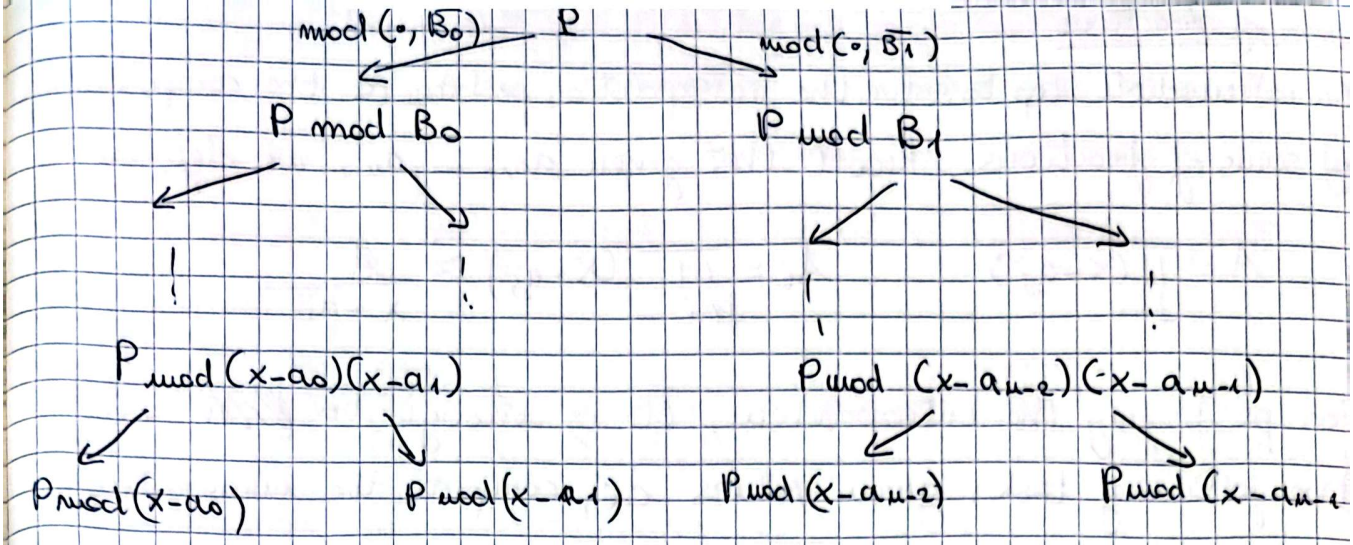
**Proof**: Let $T(n)$ be the cost of the computation of the tree with $n$ points. Then we have

$$T(n) \leq 2T\left(\frac{n}{2}\right) + M\left(\frac{n}{2}\right)$$

$\rightarrow$ We have $\log n$ iterations each with cost $M(n)$

$\Rightarrow M(n) \cdot \log n$.

Now, assuming that all the nodes of the tree of subproducts are precomputed, computing the multipoint evaluation recursively is straightforward if we allow a change of notation. This process can be seen by determining the remainders of the Euclidean divisions through the tree of subproducts as explained in the figure below.




We get the following algorithm for multipoint evaluation

Input: $a_0, \dots, a_{n-1}$ in $R$, with $n = 2^k$ ($k \in \mathbb{N}$), the tree $\overline{B}$ of subproduct obtained by $a_0, \dots, a_{n-1}$ and $P \in R[x]$, $\deg P < n$

Output: $P(a_0), \dots, P(a_{n-1})$

If $n = 1$, return $P$.

else

$P_0 = P \bmod (x-a_0) \cdots (x-a_{\frac{n}{2}-1})$

$P_1 = P \bmod (x-a_{\frac{n}{2}}) \cdots (x-a_{n-1})$

Recursively compute $L_0 = P_0(a_0), \dots, P_0(a_{\frac{n}{2}-1})$

Recursively compute $L_1 = P_1(a_{\frac{n}{2}}), \dots, P_1(a_{n-1})$

Return $L_0, L_1$.

→ cost: $O(M(n) \log n)$.

## Sum of fractions

An intermediate step to solve the interpolation problem is the computation of sums of fractions. Recall that given $a_0, \dots, a_{n-1}$ we defined

$$A = \prod_j (x - a_j) \qquad A_i = \prod_{j \neq i} (x - a_j) \equiv \frac{A}{x - a_i}$$

For performing the interpolation, it is enough to solve the following task: given values $c_i$, compute the numerator and denominator of

$$\sum_{i=0}^{n-1} \frac{c_i}{x - a_i}$$

We use again "divide and conquer": by making two recursive calls, we compute

$$S_1 = \sum_{i=0}^{\frac{n}{2}-1} \frac{c_i}{x - a_i} \quad \text{and} \quad \sum_{i=\frac{n}{2}}^{n-1} \frac{c_i}{x - a_i}$$

and then return $S_1 + S_2$. The algorithm is below:

Input: $a_0, \dots, a_{n-1} \in R$, $c_0, \dots, c_{n-1} \in R$, $n = 2^k$ ($k \in \mathbb{N}$)

Output: $N, P \in R[x]$ s.t. $P = (x - a_1) \cdots (x - a_{n-1})$ and

$$N/P = \sum_{i=0}^{n-1} \frac{c_i}{x - a_i}$$

If $n = 1$ then

return $c_0$ and $x - a_0$

else

- Recursively call the algorithm on $a_0, \dots, a_{\frac{n}{2}-1}$ and $c_0, \dots, c_{\frac{n}{2}-1}$ and let $N_1$ and $P_1$ the obtained polynomials.
- Recursively call the algorithm on $a_{\frac{n}{2}}, \dots, a_{n-1}$, $c_{\frac{n}{2}}, \dots, c_{n-1}$ and call $N_2, P_2$ the two polynomials

Return $N_1 P_2 + N_2 P_1$ and $P_1 P_2$.

This algorithm cost $O(M(n)\log n)$ arithmetic operations in $R$.

## Interpolation

Now interpolating polynomial for the values $b_i$ at the points $a_i$ is given by

$$P(X) = \sum_{i=0}^{n-1} \frac{b_i}{A_i(x)} \frac{A(x)}{A_i(a_i)} = A(x) \cdot \sum_{i=0}^{n-1} \frac{b_i}{A_i(a_i)(X-a_i)}$$

We know how to compute fast $\frac{b_i}{A_i(a_i)} = c_i$

The iterative computation of the constants $c_i$ would involve evaluating $n$ different polynomials, each at one point, which in the worst case will give $O(n^2)$.

With the next result we can replace $n$ evaluations by a multipoint evaluation of a polynomial at $n$ points.

**Proposition:** $\forall\ i=0,\dots,n-1$, we have $A_i(a_i) = A'(a_i)$

**Proof:** Differentiate $A$:

$$A(x) = \sum_{i=0}^{n-1} \prod_{j\neq i} (x-a_j) = \sum_{i=0}^{n-1} A_i$$

We have seen that $A_i(a_j) = 0\ \forall\ j \neq i$ hence we get the result.

$\longrightarrow$ We deduce that the interpolation will cost $O(M(n)\log n)$.

The algorithm is the following.

**Input:** $a_0,\dots,a_{n-1} \in R$ satisfying $a_i - a_j$ invertible in $R$ and $b_0,\dots,b_{n-1} \in R$.

**Output:** Unique polynomial $P \in R[X]$ with $\deg(P) < n$ s.t. $P(a_i) = b_i\ \forall\ i$.

- Compute the tree of subproduct $\overline{B}$ with root $A$
- Compute $d_i = A'(a_i)$ $\forall$ $0 \leq i \leq u-1$ with the fast algorithm for multipoint evaluation
- $(c_0, \ldots, c_{u-1}) = (b_0/d_0, \ldots, b_{u-1}/d_{u-1})$
- Compute the sum $R = \frac{c_0}{x-a_0} + \ldots + \frac{c_{u-1}}{x-a_{u-1}}$
- Return the numerator of $R$.