École Polytechnique — CSE202 Design and Analysis of Algorithms

# Fast multiplication of polynomials and matrices

We will implement algorithms (naive or using the diving-and-conquer paradigm) for polynomial multiplication and matrix multiplication, and we will compare their costs.

## 1 Multiplication of polynomials

Download the files `mult.py` and `test_mult.py` on moodle. A polynomial such as

$$P(x) = 1 - 3x + 8x^2 + 5x^3$$

will be represented in Python as the list `P = [1, -3, 8, 5]`. We will compare the costs of algorithms for computing the product of two polynomials. The costs will here be measured by the number of arithmetic operations (each such operation is either an addition or a multiplication of two coefficients).

**Question 1.** Implement a simple (non-recursive) function `poly_mult(P,Q)` for computing the product $P * Q$. The algorithm should consist of a double loop, using the formula :

$$P = \sum_{i=0}^{n-1} a_i x^i, \quad Q = \sum_{j=0}^{m-1} b_i x^i, \quad P * Q = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} a_i b_j x^{i+j}.$$

Test your method, e.g. for $P = 1 + 2x + 3x^2$ and $Q = 4 + 5x + 6x^2 + 7x^3$, the product is $P * Q = 4 + 13x^1 + 28x^2 + 34x^3 + 32x^4 + 21x^5$. Test your method by executing the file `test_mult.py`.

By this approach, the cost is $nm$ multiplications of coefficients and $(n-1)(m-1)$ additions of coefficients, hence an overall cost of $2nm - n - m + 1$. Complete the function `cost_poly_mult(n)` that returns the cost of multiplying two polynomials with $n$ coefficients accordingly.

**Question 2.** In view of implementing the Karatsuba algorithm, implement the following functions : `poly_add(P,Q)` computes $P + Q$, `neg(P)` computes $-P$, and `shift(P,k)` computes $x^k P$. Test your methods by executing the file `test_mult.py`.

**Question 3.** Implement `poly_kara_mult(P,Q)`, which computes $P * Q$ using the Karatsuba algorithm seen in class (as in the lecture, we assume that $P$ and $Q$ have the same length). Test your method, e.g., for `P=[1,4,7,2,3,5,9]` and `Q=[3,7,-8,9,-1,-2,5]`, the product should be `[3, 19, 41, 32, 2, 77, 46, 54, 1, 80, -4, 7, 45]`. Test your methods by executing the file `test_mult.py`.

Implement the function `cost_poly_kara_mult(n)` which computes the cost of a call, for $P$ and $Q$ of length $n$. This cost obeys the recurrence

$$C(1) = 1, \quad C(n) = 3C(\lceil n/2 \rceil) + 4n \quad \text{for } n \geq 2.$$

The sequence of costs should start as `[1, 11, 45, 49, 155, 159, 175, 179, 501, 505, 521, 525, 577, 581]`.

**Question 4.** The Karatsuba algorithm cuts the input polynomials into 2 parts. Question 1 of the exercise sheet this week gives a divide-and-conquer algorithm (which we do NOT ask you to implement !), where the input polynomials are cut into 3 parts. The cost of that algorithm obeys the recurrence

$$C(1) = 1, \; C(2) = 3, \; C(n) = 5C(\lceil n/3 \rceil) + 30n \quad \text{for } n \geq 3.$$

Implement accordingly the function `cost_poly_tc3_mult(n)`. The sequence of costs should start as `[1, 3, 95, 135, 165, 195, 685, 715, 745, 975, 1005, 1035, 1215, 1245]`. Test your methods by executing the file `test_mult.py`.

To have a compared plot of the costs of the two methods, call the function `compare_kara_tc3()` in the file `test_mult.py`.

**Question 5.** The approach of cutting into 3 parts instead of 2 gives a more efficient (for large values of $n$) algorithm, but also more involved, so we will not implement it here. Instead, we will see here a simple switch-variant of the Karatsuba algorithm that yields some improvement when conveniently adjusted. Implement the function `poly_switch_mult(d,P,Q)` (again we assume that $P$ and $Q$ have the same length $n$) that either calls the naive product algorithm if $n \leq d$, or applies the Karatsuba recursive calls otherwise. **Your function `poly_switch_mult(d,P,Q)` must call itself recursively and NOT call `poly_kara_mult(d,P,Q)`.** Test your methods by executing the file `test_mult.py`.

The cost function obeys the recurrence

$$C(n) = 2n^2 - 2n + 1 \text{ for } n \leq d, \quad C(n) = 3C(\lceil n/2 \rceil) + 4n \quad \text{for } n \geq 2.$$

Complete the method `cost_switch_mult(d,n)` accordingly. For $d = 3$ the sequence of costs should start as `[1, 5, 13, 31, 59, 63, 121, 125, 213, 217, 233, 237, 415, 419]`.

To have a compared plot of the costs of the two methods, call the function `compare_kara_switch(L)` in the file `test_mult.py`. It receives a list `L` of values of $d$ and plots together the cost functions for each $d$ in the list.

Try it for several values of $d$, small ones as well as large ones. What do you observe ?

## 2 Matrix multiplication

Download the files `strassen.py` and `strassen_test.py` on moodle. A square matrix such as

$$A = \begin{pmatrix} 3 & -1 & 2 \\ 0 & 2 & 0 \\ -2 & 1 & 3 \end{pmatrix}$$

will be represented in Python as the list of lists `A = [[3, -1, 2], [0, 2, 0], [-2, 1, 3]]`.

**Question 6.** Implement `mult_matrix(A,B)` that computes the matrix $A*B$ using the standard formula for matrix multiplication.

**Question 7.** Implement `cost_mult_matrix(n)` that computes the number of operations (additions and multiplications) performed by `mult_matrix(A,B)` where $A$ and $B$ are $n \times n$ matrices.

As seen in lecture 2, Strassen's algorithm uses a *divide-and-conquer* approach to perform matrix multiplication.

### Matrix Multiplication: Strassen's Algorithm

Input: two $n \times n$ matrices $A, X$ with $n = 2^k$

Output: $AX$

1. If $n = 1$, return $AX$
2. Split $A = \left(\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right), X = \left(\begin{array}{c|c} x & y \\ \hline z & t \end{array}\right)$, with $(n/2) \times (n/2)$ blocks
3. Compute recursively the 7 products

$q_1 = a(x+z)$, $q_2 = d(y+t)$, $q_3 = (d-a)(z-y)$,
$q_4 = (b-d)(z+t)$, $q_5 = (b-a)z$,
$q_6 = (c-a)(x+y)$, $q_7 = (c-d)y$

Exercise: prove the complexity in $O(n^{\log_2 7})$ operations.

4. Return $\begin{pmatrix} q_1 + q_5 & q_2 + q_3 + q_4 - q_5 \\ q_1 + q_3 + q_6 - q_7 & q_2 + q_7 \end{pmatrix}$

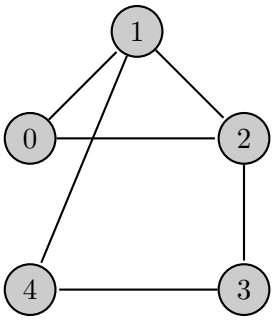
**Question 8.** Implement `strassen(A,B)` that computes the matrix $A * B$ using Strassen's algorithm. We assume that matrices $A$ and $B$ are of size $2^n \times 2^n$. You can use `split(A)` that computes the four matrices $a, b, c, d$ as pictured above, and `merge(a,b,c,d)` which computes the reverse operation.

**Question 9.** Implement `cost_strassen(n)` that computes the number of operations (additions and multiplications) performed by `strassen(A,B)` where $A$ and $B$ are $n \times n$ matrices.

To test your code, you can write a loop to display the comparated costs of `matrix_mult(A,B)` and `strassen(A,B)` for matrices of size $2^n \times 2^n$, with $n$ from 0 to 8.

## 3 Transitive closure of a graph

We consider finite graphs with vertices labeled with integers $0, \ldots, n-1$. A graph $G$ can be represented by its adjacency matrix $M$, which is the $0-1$ matrix of size $n \times n$ with $M[i][j] = 1$ iff there is an edge between vertex $i$ and vertex $j$.




$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

The objective of this section is to compute the transitive closure of $G$ – or reachability matrix of $G$ – which is the $0-1$ matrix $R$ of size $n \times n$ with $R[i][j] = 1$ iff there exists a path from $i$ to $j$ in $G$. For instance if the graph $G$ represents a railway network, computing the transitive closure tells which station is accessible from another. One can prove that

$$R = (I_n + M)^{n-1}$$

where $x + y$ is `x or y` and $x \cdot y$ is `x and y` for $0-1$ inputs $x, y \in \{0, 1\}$. $I_n$ is the identity matrix of size n.

**Question 10.** Implement `convert_01(M)` that computes the matrix $M$ into a $0-1$ matrix $B$ such that $B[i][j] = 1$ iff $M[i][j] > 0$.

**Question 11.** Implement `transitive_closure_strassen(M)` that computes the transitive closure of the matrix $M$ using Strassen's algorithm and `convert_bool(M)` (and binary powering).