Politecnico di Milano
Cryptography and Architectures for Computer Security - 095947

Efficient Software Arihmetics on GF2x for public key cryptography: C Implementation

Nicole Gervasoni
matr. 878439

Contents

1 Introduction

The aim of this project is to implement a library to perform arithmetics between polynomials of GF(2)[x], focusing on efficient multiplication. The `GF2x_ArithmeticsLib` handles any LIMB size and field dimension. A polynomial will be represented as a sequence of LIMBs, which are basically the chosen word size (in this document equal to `uint64_t`). The dimension of the field is represented by the `POWER_OF_TWO` constant which indicates the highest power of a polynomial and thus its maximum number of bits. The final project can be foud at `https://github.com/NGervasoni/GF2x_ArithmeticsLib`

2 Implementation

2.1 Definitions

```
#define LIMB uint64_t

//m : maximum degree
#define POWER_OF_TWO 12800000

#define LIMB_BITS (sizeof(LIMB) * 8) // W

// max number of limbs
#define T ((POWER_OF_TWO / LIMB_BITS) + ((POWER_OF_TWO % LIMB_BITS) != 0 ? 1 : 0))

// number of leftmost unused bit
#define S ((LIMB_BITS * T) - POWER_OF_TWO)

#define KARATSUBA_MIN_LIMBS 50

#define TOOM_MIN_LIMBS 50
```

Polynomials are of type MPN which represents them as a struct with a pointer to a sequence of LIMBs and a positive number indicating how many limbs compose the polynomial.

```
typedef struct gf2x_mp_t {  // in big-endian notation
    LIMB *num;
    unsigned limbNumber;
} MPN;
```

There are three different initialization methods for MPN struct:

```
//create an all zeros polynomial with chosen limbNumber
MPN init_empty(unsigned size);

//create a null polynomial
MPN init_null();

// create a polynomial with chosen value and limbNumber
MPN init(LIMB A[], unsigned sizeA);
```

2.2 Addition

a, b are polynomials of degree $<$ `POWER_OF_TWO`, *res points to an initialized MPN where the result will be stored.

```
void MP_Addition(MPN *result, MPN a, MPN b);
```

2.3 Multiplication

Factor 1 and factor 2 are polynomials of degree < `POWER_OF_TWO`, `irr_poly` is an irreducible polynomial of degree `POWER_OF_TWO`, *result points to an initialized MPN where the result will be stored, w divides `LIMB_BITS`.

```
void MP_ShiftAndAddMul(MPN *result, MPN factor1, MPN factor2, MPN irr_poly);

void MP_CombRtoLMul(MPN *result, MPN factor1, MPN factor2);
void MP_CombLtoRMul(MPN *result, MPN factor1, MPN factor2);
void MP_CombLtoRMul_w(MPN *res, MPN factor1, MPN factor2, unsigned w);

// if factors limbNumber < KARATSUBA_MIN_LIMBS, same as MP_CombRtoLMul
void MP_KaratsubaMul(MPN *result, MPN factor1, MPN factor2);

// if factors limbNumber < TOOM_MIN_LIMBS, calls MP_CombRtoLMul
void MP_Toom3(MPN *result, MPN factor1, MPN factor2);
void MP_Toom4(MPN *result, MPN factor1, MPN factor2);

MPN MP_Squaring(MPN poly);
```

2.4 Inversion

a has degree < `POWER_OF_TWO`, `irr_poly` is an irreducible polynomial of degree `POWER_OF_TWO`. This inversion is based on the extended euclidian algorithm.

```
MPN MP_Inversion_EE(MPN a, MPN irr_poly);
```

2.5 Division

a is not 0, b has degree < `POWER_OF_TWO` , `irr_poly` is an irreducible polynomial with degree `POWER_OF_TWO`. This method exploits the binary inversion algorithm and returns $c = b/a = b * a^{-1}$

```
MPN MP_Division_Bin_Inv(MPN a, MPN b, MPN irr_poly);

// exact divisions used by Toom3 and Toom4
static inline void MP_exactDivOnePlusX(MPN poly);
static inline void MP_exactDivXPlusXFour(MPN poly);
static inline void MP_exactDivXtwoPlusXFour(MPN poly);
```

2.6 Utilities

```

void MP_Reduce(MPN *result, MPN polyToreduce, MPN irr_poly);

void print(char *str, MPN poly);

void removeLeadingZeroLimbs(MPN *poly);

unsigned degree(MPN poly);

bool MP_compare(MPN a, MPN b);


static inline void sum_in_first_arg(MPN a, MPN b);
static inline unsigned lead_zero_limbs_count(MPN poly);
```

```
static inline void MP_free(MPN poly);
static inline void MP_bitShiftLeft(MPN *a, int bitsToShift, bool checkSize);
static inline void MP_bitShiftRight(MPN *a);
static inline void limbShiftLeft(MPN *a, int n, bool checkSize);
static inline bool isZero(MPN poly);
static inline bool isOne(MPN poly);
```

3 Multiplication Benchmarks

The time to perform 100 multiplications, between two random polynomials with the same size, is been registered to evaluate the performance of different multiplications. LIMB has been set to `uint64_t` and `POWER_OF_TWO` accordingly to at least $2 * 64 * limbNumber$ to contain all the results.

It has to be noted that since all calculations are performed in the stack space, it is advisable to increment its size when dealing with large number and recursive functions (such as *Toom3*, *Toom4*, *Karatsuba*); this can be done with the *setrlimit()* function directly in the code (see usage example in the Example of testing code. The *ShiftAndAdd multiplication* is not considered in these benchmarks. All multiplications here are supposed to belong to the chosen ring thus the results do not need to be reduce. In this scenario the *ShiftAndAdd multiplication*, which is written to perform reduction while multiplicating, is always slower then other methods. This due to the fact that it starts working with polynomials of the size of the irriducible one (that depends on the chosen `POWER_OF_TWO`) even if the factors have only one LIMB. In the following pages all registered times are reported, with MPN limbs number varying from 10 to 10000.

3.1 Data and Graphs

Table 1: Recursion limit set to 10 limbs for both Tooms and Karatsuba.

Limbs	CombRtoL	CombLtoR	Comb_W=4	Comb_W=8	Karatsuba	Toom3	Toom4
10	0.000339	0.000362	0.002764	0.038565	0.000686	0.000335	0.000335
20	0.000797	0.0009	0.00583	0.074498	0.001606	0.00079	0.000791
30	0.001445	0.001515	0.009223	0.108754	0.00292	0.001438	0.001435
40	0.002293	0.002586	0.01235	0.146432	0.004573	0.002265	0.002273
50	0.003275	0.003447	0.01685	0.212949	0.006546	0.005006	0.005277
60	0.004432	0.004903	0.020681	0.254039	0.004436	0.00623	0.006212
70	0.005803	0.006639	0.024652	0.310362	0.005974	0.007659	0.007732
80	0.008281	0.008283	0.028348	0.376649	0.007079	0.010681	0.008717
90	0.009704	0.009648	0.032687	0.445079	0.008271	0.012261	0.010216
100	0.012251	0.012694	0.037008	0.511365	0.009976	0.014524	0.011418
110	0.014157	0.014615	0.041564	0.576928	0.011879	0.017694	0.013118
120	0.016397	0.016759	0.04613	0.643803	0.013773	0.019109	0.01488
130	0.019082	0.019051	0.050836	0.712518	0.015902	0.023925	0.023689
140	0.028343	0.026691	0.057382	0.767805	0.018474	0.022436	0.018324

150	0.02464	0.02479	0.061064	0.830656	0.019581	0.030426	0.020058
160	0.027532	0.027803	0.066406	0.89563	0.020961	0.03391	0.021937
170	0.031203	0.031233	0.070785	0.968134	0.023772	0.035665	0.024912
180	0.034191	0.03437	0.081397	1.035975	0.025307	0.037859	0.02571
190	0.03787	0.03891	0.08161	1.126765	0.028884	0.04106	0.028258
200	0.042077	0.041686	0.089771	1.150542	0.030201	0.043186	0.046836
210	0.045423	0.045563	0.097408	1.218378	0.034233	0.045829	0.049127
220	0.049332	0.05229	0.098573	1.270928	0.036512	0.048928	0.056401
230	0.067284	0.055232	0.105141	1.393562	0.038638	0.056559	0.053446
240	0.058251	0.058585	0.112203	1.418825	0.041298	0.069883	0.054279
250	0.063001	0.063253	0.116508	1.462851	0.045915	0.072542	0.059534
260	0.068161	0.068981	0.122103	1.516653	0.048247	0.075623	0.061585
270	0.072688	0.075581	0.12659	1.581944	0.051204	0.079275	0.062563
280	0.078122	0.080846	0.132763	1.639579	0.055346	0.084456	0.066687
290	0.083527	0.086484	0.139641	1.701668	0.056463	0.089189	0.069646
300	0.088878	0.092286	0.144783	1.768234	0.059271	0.092963	0.070676
310	0.09429	0.097495	0.151603	1.82917	0.065586	0.096985	0.073823
320	0.100186	0.104279	0.157407	1.889662	0.063786	0.100389	0.074751
330	0.109185	0.106825	0.163863	1.953579	0.069796	0.105241	0.078823
340	0.114937	0.113618	0.170035	2.012303	0.07329	0.108364	0.08222
350	0.121122	0.119922	0.177503	2.072716	0.073879	0.112628	0.083527
360	0.126619	0.163925	0.212879	2.186963	0.07644	0.117694	0.087239
370	0.133801	0.13373	0.191546	2.196192	0.080936	0.121765	0.090984
380	0.140323	0.139939	0.196947	2.259455	0.085147	0.124019	0.092292
390	0.148599	0.147002	0.204096	2.33522	0.092197	0.128805	0.095828
400	0.156376	0.154287	0.211093	2.382582	0.091814	0.130787	0.097285
410	0.163616	0.162995	0.217846	2.447668	0.103046	0.134483	0.101412
420	0.171485	0.169645	0.225253	2.513945	0.10857	0.138277	0.105242
430	0.177516	0.177312	0.23204	2.56531	0.108724	0.143853	0.106497
440	0.186146	0.186215	0.239596	2.626451	0.113222	0.14723	0.110557
450	0.194264	0.193694	0.247147	2.699117	0.116818	0.182467	0.114422
460	0.202345	0.202309	0.254188	2.756283	0.120202	0.189617	0.116258
470	0.211111	0.211506	0.261828	2.815737	0.126897	0.194289	0.121103

480	0.220883	0.220806	0.269752	2.881589	0.130426	0.197996	0.12213
490	0.22829	0.228832	0.277041	2.93612	0.13575	0.204544	0.127793
500	0.238174	0.237699	0.285026	2.996196	0.141548	0.209252	0.132179
1000	0.919552	0.920365	0.786387	6.102311	0.459243	0.6518	0.501495
1500	2.045314	2.047616	1.505743	9.365185	0.816359	1.21387	0.765021
2000	3.620003	3.623007	2.446668	12.736455	1.596805	1.924676	1.089624
2500	5.657789	5.663581	3.605209	16.279647	2.250583	3.092623	1.460985
3000	8.125971	8.135163	4.991237	20.075804	2.795232	3.840036	1.808733
3500	11.037615	11.106285	6.597138	23.939765	5.060465	4.551234	3.580192
4000	14.432153	14.427716	8.500919	27.98962	6.153981	5.738763	4.19563
4500	18.215817	18.258201	10.484516	32.20561	7.284759	6.921369	4.68134
5000	22.468759	22.520709	12.741701	36.986281	8.475483	7.739398	5.166575
5500	27.384488	27.304001	15.229421	41.05916	9.454283	8.527324	5.732255
6000	32.332847	32.430934	17.985499	45.745144	10.455074	11.644759	6.307831
6500	37.9152	38.237092	20.986608	50.640377	19.974545	15.060792	6.896482
7000	44.965029	44.295042	24.256703	55.79436	22.753918	16.479591	7.504347
7500	50.944226	50.763854	27.565973	60.778111	24.48449	17.995728	8.183943
8000	57.833328	57.979172	31.12963	66.099571	26.820293	19.231873	8.881917
8500	65.468466	66.104066	35.001945	71.543517	29.064761	20.699232	9.71862
9000	73.970455	73.67486	39.040217	77.13883	31.328375	22.188941	10.2335
9500	83.954667	82.12312	43.97131	83.533527	33.334921	23.828611	10.954675
10000	91.466389	92.24539	48.106729	88.989439	36.040281	24.999889	11.770787

Figure 1: Y-axis: seconds to perform 100 multiplications, X-axis: number of limbs; recursion limit = 10 limbs.

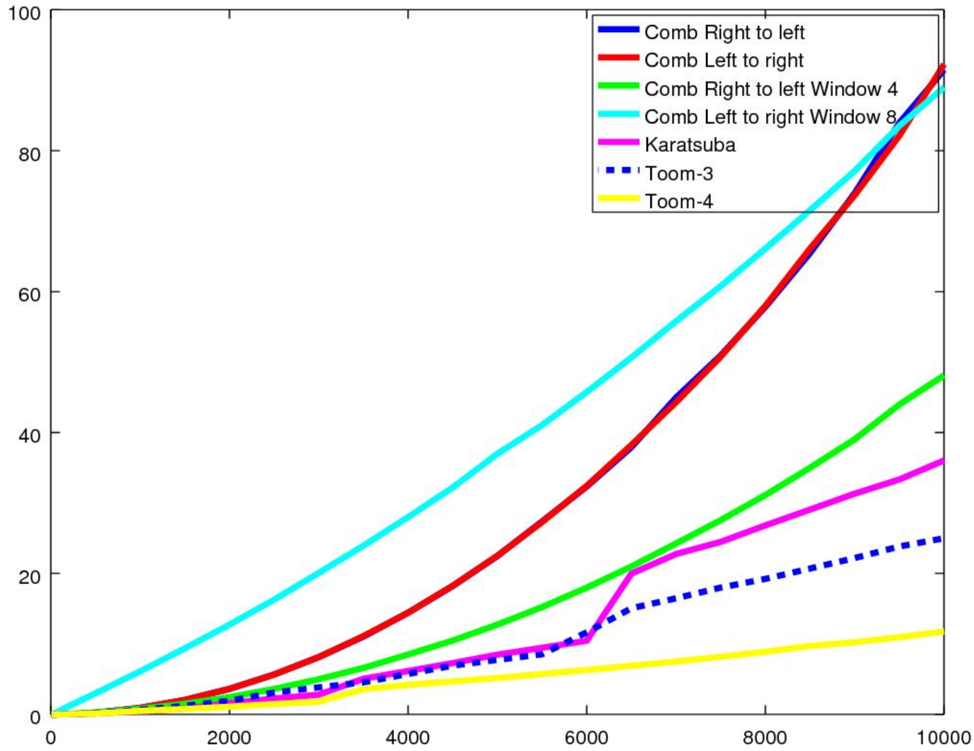


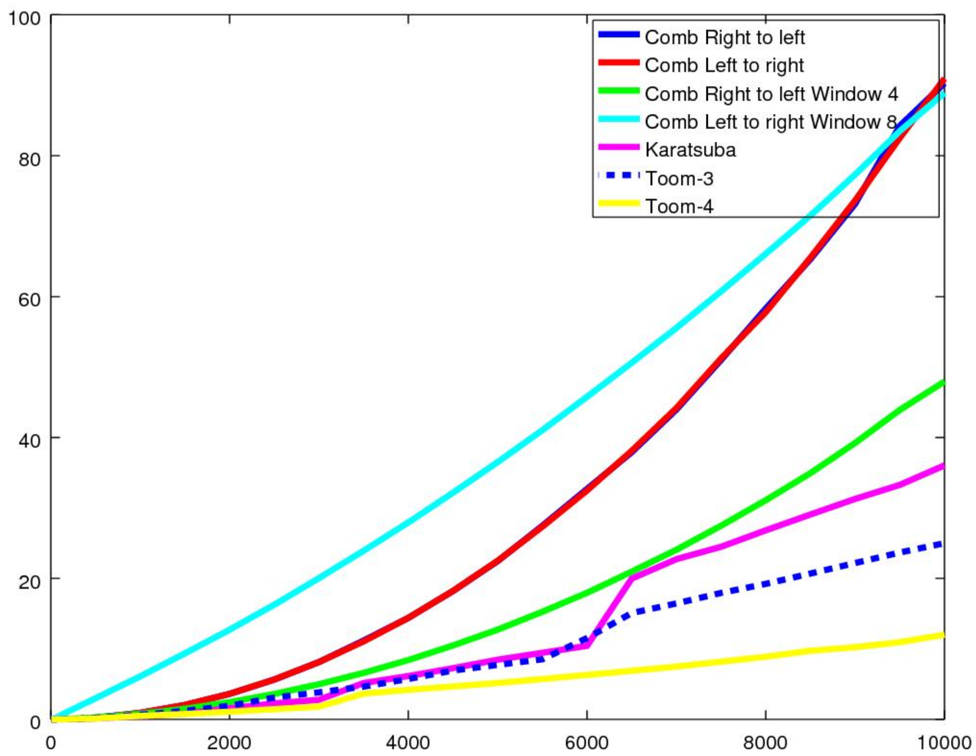
Table 2: Recursion limit set to 10 limbs for both Tooms and Karatsuba.

Limbs	CombRtoL	CombLtoR	Comb_W=4	Comb_W=8	Karatsuba	Toom3	Toom4
10	0.000339	0.000373	0.002802	0.038131	0.000686	0.000335	0.000334
20	0.000798	0.000889	0.005831	0.080208	0.001615	0.000787	0.0008
30	0.001447	0.001517	0.00901	0.111661	0.002931	0.001438	0.001437
40	0.002286	0.00258	0.012719	0.148463	0.004592	0.002274	0.002273
50	0.003268	0.003444	0.020371	0.215655	0.006545	0.005045	0.005242
60	0.004433	0.005247	0.021368	0.256815	0.004467	0.006224	0.006218
70	0.005925	0.0066	0.024985	0.313654	0.00599	0.007673	0.007717
80	0.00839	0.008353	0.028712	0.379494	0.006907	0.010683	0.008699
90	0.00992	0.009728	0.033198	0.447633	0.008279	0.012192	0.0102
100	0.012043	0.012463	0.036962	0.511184	0.009947	0.014534	0.011469
110	0.014302	0.014515	0.041396	0.57716	0.012003	0.016616	0.013102
120	0.016634	0.017293	0.046362	0.640588	0.013601	0.018778	0.014605
130	0.019552	0.019267	0.051	0.706559	0.015862	0.020513	0.016734
140	0.022722	0.021723	0.056147	0.766748	0.018104	0.022205	0.018023
150	0.024812	0.024579	0.060936	0.835312	0.01978	0.030385	0.020076
160	0.027773	0.027636	0.06567	0.896329	0.021061	0.033329	0.021841
170	0.03083	0.030882	0.07097	0.953787	0.024063	0.035465	0.024955
180	0.034479	0.034271	0.077893	1.021089	0.02518	0.038104	0.02561
190	0.038056	0.037867	0.082725	1.079431	0.027845	0.040936	0.028174
200	0.041877	0.041635	0.088944	1.144855	0.030189	0.043229	0.046054
210	0.045419	0.045754	0.0934	1.207989	0.034604	0.046981	0.049174
220	0.05068	0.05255	0.097908	1.26872	0.036397	0.048778	0.050254
230	0.054479	0.05668	0.103308	1.334326	0.038458	0.056615	0.053427
240	0.058384	0.058457	0.111273	1.398233	0.041323	0.069831	0.054527
250	0.064081	0.064908	0.115658	1.457467	0.045975	0.072587	0.059233
260	0.068196	0.069566	0.120843	1.518522	0.048811	0.075367	0.06164
270	0.074069	0.076173	0.126627	1.58211	0.051296	0.079136	0.062531
280	0.079197	0.08046	0.132489	1.640118	0.055807	0.084576	0.066707
290	0.083915	0.087165	0.139055	1.703375	0.056607	0.089429	0.069756
300	0.089269	0.092364	0.144826	1.768124	0.059288	0.093084	0.07057
310	0.094559	0.097148	0.151091	1.827846	0.065546	0.097608	0.07426

320	0.101632	0.102622	0.157151	1.889051	0.063876	0.100659	0.075164
330	0.108795	0.107149	0.163773	1.957058	0.070355	0.10528	0.078981
340	0.114038	0.11307	0.170464	2.012425	0.073125	0.108496	0.082567
350	0.121346	0.119546	0.176967	2.074277	0.074031	0.112416	0.083227
360	0.127763	0.126079	0.183611	2.140237	0.076902	0.117985	0.087054
370	0.134985	0.13287	0.190143	2.198574	0.081386	0.121645	0.09073
380	0.141739	0.14093	0.197072	2.260108	0.085689	0.124205	0.092252
390	0.148114	0.147588	0.204242	2.326856	0.091042	0.128537	0.096083
400	0.15586	0.154263	0.211054	2.384494	0.091836	0.13052	0.097257
410	0.162249	0.162496	0.218161	2.448281	0.102782	0.134588	0.10167
420	0.171155	0.169633	0.226217	2.512245	0.106541	0.138441	0.105445
430	0.179181	0.177387	0.232216	2.567327	0.108575	0.142801	0.106376
440	0.186732	0.185404	0.239724	2.629191	0.113747	0.147348	0.110788
450	0.195098	0.193663	0.247863	2.702011	0.116513	0.181889	0.116251
460	0.202523	0.202578	0.254261	2.753894	0.12005	0.189959	0.116261
470	0.211742	0.21054	0.262278	2.815565	0.126344	0.194835	0.120868
480	0.221303	0.219296	0.269738	2.883586	0.128441	0.197496	0.123061
490	0.229515	0.228604	0.276864	2.937876	0.135523	0.204728	0.127913
500	0.238629	0.23754	0.285572	2.998899	0.14153	0.209445	0.132653
1000	0.918771	0.919954	0.784505	6.103248	0.457675	0.649376	0.501597
1500	2.044129	2.049399	1.504924	9.365008	0.817179	1.214509	0.764467
2000	3.618965	3.625101	2.448415	12.723638	1.596003	1.929095	1.088303
2500	5.648993	5.656293	3.604784	16.271578	2.258801	3.096299	1.454545
3000	8.133907	8.1341	4.99131	20.045246	2.783625	3.839699	1.80742
3500	11.189631	11.069441	6.598308	23.933277	5.15902	4.651867	3.658287
4000	14.402996	14.412317	8.419013	27.94805	6.151386	5.73537	4.190454
4500	18.217591	18.228074	10.491663	32.165283	7.266419	6.938985	4.680015
5000	22.461564	22.512327	12.73654	36.517503	8.473339	7.733455	5.162412
5500	27.47771	27.350786	15.268641	41.043912	9.447823	8.530258	5.730176
6000	32.699553	32.491842	17.964437	45.777892	10.442894	11.544905	6.308641
6500	37.917313	38.115396	20.964686	50.612423	19.981392	15.092513	6.897962
7000	43.970751	44.185629	24.072685	55.551755	22.739937	16.456361	7.491922
7500	51.010399	51.230357	27.485718	60.74851	24.495114	17.947197	8.184292

8000	58.307057	57.790662	31.101578	66.069166	26.83025	19.233947	8.891496
8500	65.31335	65.534564	34.946666	71.507503	29.079143	20.698084	9.739129
9000	73.156507	73.644646	39.231657	77.308775	31.289495	22.191175	10.242994
9500	84.237875	82.524079	43.914638	83.495162	33.27895	23.684644	10.955086
10000	90.278719	90.939658	47.949669	88.897291	36.048978	25.006839	12.020239

Figure 2: Y-axis: seconds to perform 100 multiplications, X-axis: number of limbs; recursion limit = 50 limbs.



4 Conclusion

HYPOTHESYS
I was not able to find previous performance tests or intervals, so I was expecting *Comb multiplications* to be faster for smaller sequence of limbs and to become slower as the limbNumber increases. At this point the *karatsuba*, *toom3* or *toom4* should have been faster.

RESULTS
As expected, for relatively small numbers of limbs methods *CombRtoL* and *CombLtoR* are faster than others (see previous graphs). As the number of limbs increases any other multiplication becomes more convenient to choose with respect to *CombRtoL* and *CombLtoR*.
It has to been highlighted that the performance of *karatsuba*, *toom3* and *toom4* highly depends on the choice of the minimum limbs number for recursion. By being too low this value causes the number of recursive call to considerably grow slowing down the multiplications.

4.1 Example of testing code

```
#include <stdio.h>
#include <time.h>
#include <sys/resource.h>
#include "GF2x_Arithmetics.h"

#define BILLION 1000000000L

LIMB myRand(LIMB low, LIMB high) {
    return rand() % (high - low + 1) + low;
}

struct timespec diff(struct timespec start, struct timespec end) {
    struct timespec temp;
    if ((end.tv_nsec - start.tv_nsec) < 0) {
        temp.tv_sec = end.tv_sec - start.tv_sec - 1;
        temp.tv_nsec = 1000000000 + end.tv_nsec - start.tv_nsec;
    } else {
        temp.tv_sec = end.tv_sec - start.tv_sec;
        temp.tv_nsec = end.tv_nsec - start.tv_nsec;
    }
    return temp;
}

void setResultArray(MPN *result, int RANDOM_NUMBERS) {
    for (int m = 0; m < RANDOM_NUMBERS; ++m) {
        MP_free(result[m]);
        result[m] = init_null();
    }
}

bool everything_is_fine(MPN a, MPN b) {
    MPN r = init_null();
    MPN result = init_null();

    MP_CombRtoLMul(&result, a, b);

    MP_CombLtoRMul_w(&r, a, b, 4);

    if (!MP_compare(result, r))
```

```
        return false;

    MP_CombLtoRMul_w(&r, a, b, 8);

    if (!MP_compare(result, r))
        return false;

    MP_KaratsubaMul(&r, a, b)
    if (!MP_compare(result, r))
        return false;

    MP_Toom3(&r, a, b);
    if (!MP_compare(result, r))
        return false;

    MP_Toom4(&r, a, b);
    if (!MP_compare(result, r))
        return false;

    MP_CombLtoRMul(&r, a, b);
    if (!MP_compare(result, r))
        return false;

    return true;
}


static inline void MP_free(MPN poly) {
    free(poly.num);
} //end MP_free

void main(int argc, char *argv[]) {


    // ---------------------------------------- optional ----------------------------------------

// changing stack size to avoid stack overflow during large number multiplications

    const rlim_t stackSize = 128L * 1024L * 1024L;   // min stack size = 128 Mb
    struct rlimit rl;
    int response;

    // current stack limit
    int dim = getrlimit(RLIMIT_STACK, &rl);

    rl.rlim_cur = stackSize;
    response = setrlimit(RLIMIT_STACK, &rl);
    if (response != 0)
        printf("error when changing stack limit!\n");

    // ---------------------------------------- optional ----------------------------------------

    setvbuf(stdout, 0, 2, 0);

    int factors_size = atoi(argv[1]);
    int random_numbers = atoi(argv[2]);
    MPN result[random_numbers];

    for (int m = 0; m < random_numbers; ++m) {
        result[m] = init_null();
```

```
    }


// Calculate time taken by a request
    struct timespec requestStart, requestEnd;

    int l = factors_size;

    printf("\n%d", l);

// random factor initialization
    LIMB limbs[l];
    MPN factors1[random_numbers], factors2[random_numbers];

    for (int j = 0; j < random_numbers; ++j) {
        for (int i = 0; i < l; ++i) {
            limbs[i] = myRand(1, 0xffffffffffff);
        }
        factors1[j] = init(limbs, l);
    }

    for (int j = 0; j < random_numbers; ++j) {
        for (int i = 0; i < l; ++i) {
            limbs[i] = myRand(1, 0xffffffffffff);
        }
        factors2[j] = init(limbs,(unsigned)l);
    }

    double accum;
    struct timespec time;

  // check if all multiplications are working
    if (!everything_is_fine(factors1[0], factors2[0])) {
        printf("Something is not working! Aborting...\n");
        exit(EXIT_FAILURE);
    }

  // get cpu time before computation
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &requestStart);
    for (int k = 0; k < random_numbers; ++k) {
        MP_CombRtoLMul(&result[k], factors1[k], factors2[k]);
    }
    // get cpu time after computation
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &requestEnd);
    time = diff(requestStart, requestEnd);
    accum = time.tv_nsec + time.tv_sec * BILLION;
    accum /= BILLION;
    printf("\t%lf", accum);

    setResultArray(result, random_numbers);

    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &requestStart);
    for (int k = 0; k < random_numbers; ++k) {
        MP_CombLtoRMul(&result[k], factors1[k], factors2[k]);
    }
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &requestEnd);
    time = diff(requestStart, requestEnd);
    accum = time.tv_nsec + time.tv_sec * BILLION;
    accum /= BILLION;
    printf("\t%lf", accum);
```

```c

    setResultArray(result, random_numbers);

    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &requestStart);
    for (int k = 0; k < random_numbers; ++k) {
        MP_CombLtoRMul_w(&result[k], factors1[k], factors2[k], 4);
    }
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &requestEnd);
    time = diff(requestStart, requestEnd);
    accum = time.tv_nsec + time.tv_sec * BILLION;
    accum /= BILLION;
    printf("\t%lf", accum);

    setResultArray(result, random_numbers);

    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &requestStart);
    for (int k = 0; k < random_numbers; ++k) {
        MP_CombLtoRMul_w(&result[k], factors1[k], factors2[k], 8);
    }
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &requestEnd);
    time = diff(requestStart, requestEnd);
    accum = time.tv_nsec + time.tv_sec * BILLION;
    accum /= BILLION;
    printf("\t%lf", accum);

    setResultArray(result, random_numbers);

    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &requestStart);
    for (int k = 0; k < random_numbers; ++k) {
        MP_KaratsubaMul(&result[k], factors1[k], factors2[k]);
    }
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &requestEnd);
    time = diff(requestStart, requestEnd);
    accum = time.tv_nsec + time.tv_sec * BILLION;
    accum /= BILLION;
    printf("\t%lf", accum);

    setResultArray(result, random_numbers);

    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &requestStart);
    for (int k = 0; k < random_numbers; ++k) {

        MP_Toom3(&result[k], factors1[k], factors2[k]);
    }
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &requestEnd);
    time = diff(requestStart, requestEnd);
    accum = time.tv_nsec + time.tv_sec * BILLION;
    accum /= BILLION;
    printf("\t%lf", accum);

    setResultArray(result, random_numbers);

    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &requestStart);
    for (int k = 0; k < random_numbers; ++k) {
        MP_Toom4(&result[k], factors1[k], factors2[k]);
    }
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &requestEnd);
    time = diff(requestStart, requestEnd);
    accum = time.tv_nsec + time.tv_sec * BILLION;
    accum /= BILLION;
```

```c
    printf("\t%lf", accum);

    for (int k = 0; k < random_numbers; ++k) {
        MP_free(factors1[k]);
        MP_free(factors2[k]);
        MP_free(result[k]);
    }
}
```

References

[1] Marco Bodrato *Towards Optimal Toom-Cook Multiplication for Univariate and Multivariate Polynomials in Characteristic 2 and 0* (2007) `http://bodrato.it/papers/#WAIFI2007`.

[2] Hankerson, Menezes, Vanstone *Guide to Elliptic Curve Cryptography* (2003) `http://www.springer.com/it/book/9780387952734`