BULETINUL Universităţii Petrol – Gaze din Ploieşti	Vol. LX No. 2/2008	35-42	Seria Matematică - Informatică - Fizică

A Comparative Study About the Complexity of Some Recursive Algorithms

Alexe Călin Mureşan

Universitatea Petrol-Gaze din Ploieşti, Bd. Bucureşti 39, Ploieşti, Catedra de Matematică
e-mail: acmuresan@upg-ploiesti.ro

Abstract

The aim of this paper is to offer a comparative study about the costs involved in fundamental recursive algorithms. We present here the „Fast Fourier Transform", the „Karatsuba" method and also a method for evaluating the polynomial using the cost of x^n x^n *and Stirling formula.*

Key words: *Fast Fourier Transform, Karatsuba method, Stirling formula*

Introduction

Remark 1. The algorithm for „ *The Fast Fourier Transform*" presented in Theorem 1 is one of the 10 algorithms with the greatest influence on the development and practice of software and engineering in the 20th century, see [1], [2], [3]. In [4] the authors generalized the above result. In this paper we present „ The Fast Fourier Transform" for evaluating and interpolating the polynomials.

The complexities of this problem is $O(n\log(n))$ or $O(n\log(n)\log\log(n))$ and not $O(n^2)$ as we can see in the naive method. Also we can compute the polynomials into a fixed point using the computation for x^n, where we use the writing for n in 2-base sistem, see [5], [6]. Evaluating the complexities problem, we can find using the „Stirling formula" also the cost $O(n\log(n))$. Then we use „ The Fast Fourier Transform" and the „Karatsuba" method for seeing and comparing the complexities costs for multypling the polynomials.

Definition 1. For a given function $g(x)$, $g:R\to R$, we denote by $O(g(x))$ the set of functions: $O(g(x))=\{f(x)\ /\ f:R\to R$ and there exists positive constants c and $x_0\in R$ such that $0\le f(x)\le c\cdot g(x)$ for all $x\ge x_0\}$. In this case for every $f(x)$ we denote: $O(g(x))=f(x)$.

Definition 2. For a given function $g(x)$, $g:R\to R$, we denote by $\Theta(g(x))$ the set of functions: $\Theta(g(x))=\{f(x)\ /\ f:R\to R$ and there are the constants $c_1>0$, $c_2>0$, and $x_0\in R$ so that for all $x>x_0$ then it is true that $c_1 g(x)<f(x)<c_2 g(x)\}$. In this case for every $f(x)$ we denote: $\Theta(g(x))=f(x)$.

Definition 3. We consider that $f(x) \sim g(x)$ if $f(x)= \Theta(g(x))$ and $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 1$.

Proposition 1. a) $\left(\frac{n}{e}\right)^n \le n! \le \frac{(n+1)^{n+1}}{e^n}$, b) Stirling's formula: $x! \sim \left(\frac{x}{e}\right)^x \sqrt{2x\pi}$.

See [7], [8] or [9] to References.

Proposition 2. The Complexity of Divide-and-Conquer algorithms

The *recursively algorithms* typically follow a *divide-and-conquer* paradigm for solving a computational problem which involves three levels of the recursion:

A recurrence for the running time of a divide-and-conquer algorithm, $T(n)$ on a problem of size n is:

$$\mathrm{T}(n) = \begin{cases} \theta(1) & \text{if } n \le k,\ k \in R \text{ is given} \\ aT(n/b) + D(n) + C(n) & \text{if } n > k. \end{cases}$$

Where we divide the problem into 'a' subproblems, each of them being '$1/b$' the rise size of the original and we denote by: $D(n)$ the time to *divide* the problem into subproblems, $\Theta(1)$ if $n \le k$, k is a given real constant, the time for *conquer* a subproblem and $C(n)$ the time to *combine* the solutions to the subproblems.

Proof. It is based on three steps of the paradigm. If the problem size is small enough, say $n \le k$ for some constant k, the simplest solution takes constant time, which we will write as $\Theta(1)$. Otherwise, we follow the paradigm and obtain the result.

The Fast Fourier Transform

Definition 4

a) The Fourier *transform* is a method of converting from one representation of a polynomial, by the sequence of *coefficients* of the polynomial, to another where the representations are the sequence of *values* of polynomial at a certain set of points.

b) For a polynomial with the degree n-1 if we take the sequence of *values* of that polynomial in the n^{th} roots of unity we denote the previous method Fast Fourier Transform, *FFT*.

Theorem 1. For evaluating $P(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} \in C[x]$ using *FFT* we need $O(n \log_2 n)$ multiplications of complex numbers.

Proof. The polynomial $P(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} \in C[x]$ is known as a sequence of n complex numbers of his coefficients: $a_0, a_1, ..., a_{n-1}$.

We can extend with additional 0's the length of the array until it becomes a power of 2, $a_0, a_1, ..., a_{n-1}, 0, 0...0$ and then the *FFT* procedure can be considered for a polynomial with degree n a power of 2. We suppose now, without losing the generality, that $n = 2^s,\ s \in N^*$.

We compute the polynomial values at the n^{th} roots of unity:

$$\xi_i = e^{2\pi ij/n},\ i \in \{0,\ 1,\ ...\ n\text{-}1\} \text{ where } j = (0,1) \in C \tag{1}$$

We find the Fourier transform of the given sequence to be the sequence:

$$P\left(\xi_i\right)=\sum_{k=0}^{n-1} a_k \xi_i^k=\sum_{k=0}^{n-1} a_k e^{2\pi ikj/n}, \qquad i\in\{0,\ 1,\ \ldots\ n-1\}. \tag{2}$$

Then the values of P, a polynomial of degree 2^s -1, at the $(2^s)^{th}$ roots of unity are:

$$P(\xi_i)=\sum_{k=0}^{n-1} a_k e^{2\pi ikj/2^s}, \quad i\in\left\{0,\ 1,\ldots,2^s-1\right\}. \tag{3}$$

We divide the previous sum into two sums, containing respectively the terms, where $k=2m$ and those where $k = 2m + 1$ for natural $m\in\{0,\ 1,\ldots,2^{s-1}-1\}$. Then, for each $i\in\left\{0,\ 1,\ldots,2^s-1\right\}$ we can write:

$$P(\xi_i)=\sum_{m=0}^{2^{s-1}-1} a_{2m} e^{2\pi ij(2m)/2^s}+\sum_{m=0}^{2^{s-1}-1} a_{2m+1} e^{2\pi ij(2m+1)/2^s}, \tag{4}$$

$$P(\xi_i)=\sum_{m=0}^{2^{s-1}-1} a_{2m} e^{2\pi imj/2^{s-1}}+e^{\pi ij/2^{s-1}}\sum_{m=0}^{2^{s-1}-1} a_{2m+1} e^{2\pi imj/2^{s-1}} \tag{5}$$

We want to compute $P\left(\xi_i\right)$ where $\xi_i = e^{2\pi ij/n}$, $i\in\left\{0,\ 1,\ldots,2^s-1\right\}$, that means for 2^s values.

The first sum of the sums that appear in the previous equality is a Fourier transform of the array $a_0, a_2, a_4\ldots, a_{2^s\text{-}2}$, and the second sum is a Fourier transform of $a_1, a_3, a_5\ldots, a_{2^s\text{-}1}$; these sums are defined for only 2^{s-1} values, that means for $m\in\left\{0,\ 1,\ldots,2^{s-1}-1\right\}$

For solving the *FFT* problem we use the previous recursive relation into a recursive program.

Let

$$g(i)=\sum_{m=0}^{2^{s-1}-1} a_{2m} e^{2\pi imj/2^{s-1}} \tag{6}$$

denote the first sum. Then $g(i)$ is a periodic function of i, of period 2^{s-1}, for all integers i, because

$$g\left(i+2^{s-1}\right)=\sum_{m=0}^{2^{s-1}-1} a_{2m} e^{\left[2\pi mj\left(i+2^{s-1}\right)\right]/2^{s-1}}=\sum_{m=0}^{2^{s-1}-1} a_{2m} e^{2\pi imj/2^{s-1}} e^{2\pi mj}=g(i) \tag{7}$$

Now for computing $g(i)$, $i\in\left\{0,\ 1,\ldots,2^s-1\right\}$ first we compute $g(i)$, $0\le i\le 2^{s-1}-1$ and for some i so that $2^{s-1}\le i\le 2^s$-1 we can get that value being equally to $g(i \bmod 2^{s-1})$.

The Fast Fourier Transform algorithm in recursive form, where n is supposing to be $n=2^S$ and using the type *complex array* to denote an array of complex numbers from relation (1) and (2) have the next form.

{Be it $a_0, a_1, ..., a_{n\text{-}1}$ and $n = 2^s$, we denote $P(\xi_i) = f(i)$ }

function $f(n = 2^s$: integer; $a_0, a_1, ..., a_{2^s\text{-}1}$: complex array): complex array;

if s = 0 then f[0] := a_0

else *array1* := $\{a_0, a_2, a_4 ..., a_{2^s\text{-}2}\}$;

array2 := $\{ a_1, a_3, a_5 ..., a_{2^s\text{-}1}\}$;

$\{c_0, c_1, c_2 ..., c_{2^{s-1}\text{-}1}\}$:= *f*(2^{s-1}, *array1*);

$\{d_0, d_1, d_2 ..., d_{2^{s-1}\text{-}1}\}$:= *f*(2^{s-1}, *array2*);

for i := 0 to $2^s - 1$ do

t := $e^{\pi i/2^{s-1}}$,

f[*i*] := $c_{i \bmod 2^{s-1}} + t \cdot d_{i \bmod 2^{s-1}}$,

End {f}.

We study now the complexity of *FFT*. Let $T(k)$ denote the number of multiplications of complex numbers that will be done, for the worst-case running time, if we call *FFT* on an array whose length is 2^k. If k=0, the array is formed by a_0 and $T(0)$=0. The procedure will be continued until we follow $s = \log_2 n$ steps.

Divide: The divide step just splits the middle of the array with length 2^k, the step takes constant time. Thus, $D(k) = T(1)$.

Conquer: We recursively solve two subproblems, each of size $2^{k\text{-}1}$, which contributes

$T(k) = 2 \cdot T(k\text{-}1)$ to the running time because the call to *f*(2^{k-1}, *array1*); costs $T(k - 1)$

multiplications as does the call to *f*(2^{k-1},*array2*).

Combine: In the merge procedure we have:

The cycle 'for i := 0 to n' loop requires $n = 2^k$ more multiplications. Hence $C(k) = 2^k$.

When we add the functions $D(k)$ and $C(k)$ for the *FFT* "conquer" step gives the recurrence for the worst-case running time $T(k)$ of *FFT*:

$$T(k) = \begin{cases} T(0) = 0 \text{ if } k = 0, \\ T(k) = 2T(k\text{-}1) + 2^k, \text{ if } k \geq 0. \end{cases} \quad (8)$$

If we change variables by writing $T(k) = 2^k t_k$, then we find that $t_k = t_{k\text{-}1} + 1$, which, together with $t_0 = 0$, implies that $t_k = k$ for all $k \geq 0$, and therefore that $T(k) = k\, 2^k$.

Now k become $s = \log_2 n$ then $T(s) = n \log_2 n$ is the cost of *FFT* for the worst-case running time. Then $O(n \log_2 n)$, running time for large enough inputs, is the cost of *FFT*.

Theorem 2. *FFT* for Interpolating the Polynomial, or the *Inverse Fourier Transform.*

For $P(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} \in C[x]$ if we have the given values $P(\xi_i)$ at the n^{th} roots of unity $\xi_i = e^{2\pi i/n}, i \in \{0, 1, \ldots n-1\}$, then we can recover the coefficient sequence $\{a_0, a_1, \ldots, a_{n-1}\}$ to in $O(n \cdot \log_2 n)$ multiplications of complex numbers.

Proof. We know from the last Theorem that if we have a given sequence $\{a_0, a_1, \ldots, a_{n-1}\}$ with the coefficients of P then the Fourier transform of the sequence is:

$$P(\xi_i) = \sum_{k=0}^{n-1} a_k e^{-2\pi ikj/n},\ i \in \{0,1,\ldots,n-1\}. \tag{9}$$

Conversely, if we have the given values $P(\xi_j)$, $i \in \{0,1,\ldots,n-1\}$ then we can recover the coefficient sequence $\{a_0, a_1, \ldots, a_{n-1}\}$ by the inverse formulas:

$$a_k = \frac{1}{n}\sum_{k=0}^{n-1} P(\xi_i)e^{-2\pi ikj/n},\ i \in \{0,1,\ldots,n-1\}. \tag{10}$$

The cost is obviously equal to the cost of the *FFT* plus a linear number of conjugations and divisions by n so the cost is $O(n \cdot \log_2 n)$.

The Cost for Computing x^n and Evaluating the Polynomials with Stirling Formula

Definition 5. The cost of an algorithm that calculates x^n is given by the number of the multiplications effectuated until we obtain the result; then its cost will be $C(\boldsymbol{x^n})$.

Proposition 3. We can find out x^n where 'n' has a representation in the 2-base system with the cost $C(\boldsymbol{x^n})$ where $\log_2 n \le C(x^n) \le 2\log_2 n$ and $C(\boldsymbol{x^n}) = \theta(\log_2 n)$.

We can see [5] and [6].

Proposition 4. The cost $C(P(x))$ for Evaluating the Polynomial

$P(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} \in R[x]$ is computed using only $O(n\log_2 n)$ multiplications of complex numbers.

Proof: We consider the writing cost for Evaluating the Polynomial. represented by the number of multiplications and then: $C(P(x)) = O(\lg_2 n) + O(\lg_2(n-1)) + \ldots + O(\lg_2 2) + O(\lg_2 1)$,

$C(P(x)) = a_n \lg_2 n + a_{n-1}\lg_2(n-1) + \ldots + a_1 \lg_2 1$ where $a_n \in \boldsymbol{R}$.

Be it

$$\min_{i\in\{1,2,\ldots,n\}} a_i = c;\ \max_{i\in\{1,2,\ldots,n\}} a_i = C \tag{11}$$

Then:

$$c \cdot \sum_{i=1}^{n} \lg_2 i \le C(x) \le C \sum_{i=1}^{n} \lg_2 i \Leftrightarrow c \cdot \lg_2 (n!) \le C(x) \le C \cdot \lg_2 (n!) \Leftrightarrow$$

$$C(x) = O\left(\log_2(n!)\right). \tag{12}$$

By Proposition 1 we have:

$$n! \sim \left(\frac{n}{e}\right)^n \sqrt{2n\pi} \quad \text{or} \quad \left(\frac{n}{e}\right)^n \le n! \le \frac{(n+1)^{n+1}}{e^n}. \tag{13}$$

Then

$$\log_2(n!) \le \log_2\left(\left(\frac{n+1}{e}\right)^n \cdot (n+1)\right) = n\log_2\left(\frac{n+1}{e}\right) + \log_2(n+1) \tag{14}$$

and

$$(\exists)\ n_0 \in N\ (\exists) c > 0 \text{ so that } \log_2(n!) \le c \cdot \log_2 n,\ (\forall)\ n \ge n_0. \tag{15}$$

Now, because $C(x) = O\left(\log_2(n!)\right)$, then

$$C(x) = O\left(n\log_2(n)\right). \tag{16}$$

Multiplication of Polynomials

Theorem 3. For two complex polynomials P and Q with degree m-1 and n-1 the coefficients of product PQ can be given using the *FFT* in $O((m + n) \cdot \log_2(m + n))$ arithmetic operations.

Proof: The degree of PQ is $m + n$-2. Let be $s = \left[\log_2(m+n-2)\right] + 1$, then $p - 1 = 2^s$ is the smallest integer that is a power of 2 and $p - 1 \ge m + n - 2$.

The given polynomials to the degrees m and n positive natural numbers, $a_{n-1}, b_{m-1} \ne 0$ can be written:

$$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + 0x^n + \ldots + 0x^{p-1} \in C[x],$$
$$Q(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{m-1} x^{m-1} + 0x^m + \ldots + 0x^{p-1} \in C[x],$$

The array of coefficients of P and the array of coefficients of Q are considered at the same length p. Now we compute the *FFT* at the same $\xi_i = e^{2\pi ij/p}$, $i \in \{0,1,\ldots p-1\}$, p^{th} roots of unity for the polynomials P and Q. From Theorem 1 the cost of this computation is

$$O\left(p \cdot \log_2 p\right) = O((m + n) \cdot \log_2(m + n)). \tag{17}$$

Because the degree of PQ is $m + n$-2 and $p - 1 \ge m + n - 2$ for each $\xi_i = e^{2\pi ij/p}$, $i \in \{0,1,\ldots p-1\}$ of the p^{th} roots of unity we calculate

$$(PQ)(\xi_i) = P(\xi_i) \cdot Q(\xi_i)$$

and we give the p values, wanted for knowing(identifying) the polynomial *PQ* with *FFT.*

The cost is p multiplications of numbers

$$P(\xi_i) = P\left(e^{2\pi ij/p}\right),\ i \in \{0,1,...p-1\}. \tag{18}$$

To go backwards, from values $P(\xi_i) = P\left(e^{2\pi ij/p}\right),\ i \in \{0,1,...p-1\}$ to coefficients of the polynomial *PQ*, we use the *Inverse Fourier Transform* see Theorem 2 and then we can recover the coefficient sequence $\{c_0, c_1, ..., c_{p-1}\}$ by the inverse formulas

$$c_k = \frac{1}{p}\sum_{k=0}^{p\text{-}1} P(\xi_i)e^{-2\pi ikj/p},$$

$$k \in \{0,1,...,p-1\}. \tag{19}$$

The cost is obviously equal to the cost of the *FFT* plus a linear number of conjugations and divisions by p so the cost is

$$O(p \cdot \log_2 p) = O((m+n) \cdot \log_2(m+n)). \tag{20}$$

From (17), (18) and (20) the coefficients of the polynomial *PQ* have been created at a total cost of $O((m+n) \cdot \log_2(m+n))$ arithmetic operations.

Theorem 4. Karatsuba [10] Let it be P and Q two polynomials of degree n, if we split each of them into another two polynomials of degree

$j = n/2$ if n is even or $j = n+1/2$ if n is odd adding the coefficient equally with zeros if necessary, for computing the product PQ is necessary $K(n) = O(n^{\log_2(3)})$ multiplications.

Table 1. Polynomial multiplication algorithms with the same degree *n*

Algorithm	
classical	$2n^2$
Karatsuba	$O(n^{\log(3)}) \subset O(n^{1.585})$
FFT	$O(n\log(n)\log\log(n))$

References

1. D o n g a r r a, J., S u l l i v a n, F. - Top Ten Algorithms, *Computing in Science & Engineering* 2, 1, 2000
2. C o o l e y, J.W. - The re-discovery of the Fast Fourier Transform algorithm, *Mikrochimica Acta* 3, pp. 33–45, 1987
3. S c h o n h a g e, A., S t r a s s e n, V., S c h n e l l e - Multiplikation großer Zahlen, *Computing* 7, pp. 281–292, 1971
4. C a n t o r, D.G., K a l t o f e n, E. - On fast multiplication of polynomials over arbitrary algebras, *Acta Informatica* 28, 7 (1991), pp. 693–701, 1991
5. M i g n o t t e, M.- *Introduction to Computational Algebra and Linear Programming*, Ed. Univ. Bucuresti, 2000
6. M u r e s a n, A.C. - Computerised Algebra used to calculate cost and some costs from conversions of p-base system with references of p-adics numbers, *U.P.B. Sci. Bull.*, Series A, Vol. 68, No. 3, 2006
7. K n u t h, D.E. - *The Art of Computer programming*, vol I-III, second edition, Ed. Addison Wesley, 1981
8. W i l f, H.S. - *Algorithms and Complexity* 2000, A.K. Peters Ltd Publishers of Science and Tehnology, 2002
9. C o r m e n, T.H., L e i s e r s o n, C.E., R i v e s t, R.R. - *Introduction To Algorithms*, The MIT Press, McGraw-Hill Book, 1989
10. K a r a t s u b a, A., O f f m a n, Y. - Multiplication of multidigit numbers on automata, *Soviet Physics Doklady* 7 (1963), pp. 595–596, 1963

Un studiu comparativ asupra complexității unor algoritmi recursivi

Rezumat

Scopul acestei lucrari, este acela de a oferi un studiu comparativ asupra costurilor unor algoritmi recursivi fundamentali. Prezentam aici „Fast Fourier Transform”, metoda „Karatsuba” si de asemenea o metoda pentru evaluarea polinoamelor folosind costul lui x^n *si „formula lui Stirling”*