Flattening Karatsuba's recursion tree into a single summation

Thomas Baruchel[†]

[†]Éducation nationale, France
baruchel@riseup.net

February 2019*

Abstract

The recursion tree resulting from Karatsuba's formula is built here by using an interleaved splitting scheme rather than the traditional left/right one. This allows an easier access to the nodes of the tree and $2n-1$ of them are initially flattened all at once into a single recursive formula. The whole tree is then flattened further into a convolution formula involving less elementary multiplications than the usual Cauchy product — leading to iterative (rather than recursive) implementations of the algorithm. Unlike the traditional splitting scheme, the interleaved approach may also be applied to infinite power series, and corresponding formulas are also given.

1 Introduction

The fast multiplication algorithm discovered by Anatoly Karatsuba in 1960 (and published two years later) is known to be the oldest algorithm faster than the "grade school" method; while newer algorithms are still faster for sufficiently large numbers or polynomials, it is still widely used today for multiplicating medium-sized numbers or polynomials.

Due to its recursive divide-and-conquer approach, implementing this algorithm with no care about various issues (mostly related to storage of the temporary data) will lead to poor and often slow programs. Furthermore, the triple recursion involved by the algorithm, along with propagating changes in the computed data due to consecutive subtractions, makes implementing it in an iterative style more challenging.

The purpose of this paper is to deeply rewrite Karatsuba's formula in such a way that an efficient iterative implementation would at first glance naturally

*The first version of this paper was carefully reviewed by Aurélien Monteillet and Anthony Travers. Their comments have been taken into account in the current revision.

arise. The last example of high-level pseudocode given in the paper relies on a single simple loop — and in bounded stack space.

Instead of performing $3^{\log_2(n)}$ separate products, the algorithm descrived here computes n termwise multiplications on increasingly-sparse polynomials.

2 Karatsuba's recursion tree

While the ideas discussed here may be applied to any variant of Karatsuba's initial formula, we take the following one as a starting point and group all terms as factors around each of its three distinct branches:

$$AB = \underbrace{(X+1)\,A_0B_0}_{\text{branch 0}} + \underbrace{X\,(X+1)\,A_1B_1}_{\text{branch 1}} - \underbrace{X\,(A_1-A_0)\,(B_1-B_0)}_{\text{branch 2}} \qquad (1)$$

with $A = A_1X + A_0$ and $B = B_1X + B_0$. The formula is intended to be applied recursively until elementary products are encountered, and each node in the recursion tree is labelled according to a radix-3 labelling system.

When a node is reached (by starting from the root node) without walking on any branch-2 nodes, we call it a *direct* node; it will otherwise be called *indirect*.

Karatsuba's algorithm seems to be most of the time implemented or studied by using the same splitting scheme which involves taking apart terms of lower and higher degree (this may intuitively be seen as a left/right approach); it will be referred to here as the "traditional splitting scheme".

Several other splitting schemes are fully compliant with formula (1), namely any scheme taking apart groups of some power-of-2 sequential terms. The simplest one will be considered from now on: taking apart terms of even and odd rank. Such choice will have two main benefits: identifying the exact labelling number of a node is now easier and applying the algorithm to infinite power series will also be possible. This splitting scheme will be referred to here as the "interleaved splitting scheme". Applying it recursively is illustrated below:

$$\begin{aligned}
&a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0\\
&\quad = \left(a_7x^6 + a_5x^4 + a_3x^2 + a_1\right)x + \left(a_6x^6 + a_4x^4 + a_2x^2 + a_0\right)\\
&\quad = \left(\left(a_7x^4 + a_3\right)x^2 + \left(a_5x^4 + a_1\right)\right)x + \left(\left(a_6x^4 + a_2\right)x^2 + \left(a_4x^4 + a_0\right)\right)
\end{aligned}$$

3 Partially flattening the recursion tree

In this section, the following conventions are used:

- explicit symbols for multiplication and convolution ($\times$ and $*$ respectively) for formulae which are applied recursively;
- an implcit notation if one of the factors is a polynomial — or the generating function of an integer sequence — containing only 0 and 1 coefficients.

Let A and B be two polynomials of degree $n-1$ (for clarity, n being a power of 2). Once Karatsuba's formula has been applied repeatedly $\log(n)$ times, matching coefficients by degree in A and B are multiplied together. All these n elementary multiplications obviously correspond to *direct* nodes of the tree (involving no subtraction). With the interleaved splitting scheme, the radix-3 label of such nodes also gives the binary notation (in the reversed order) of each degree. By using the $\odot$ symbol for the *termwise* multiplication of polynomials (or power series), we can group these n elementary multiplications as:

$$\frac{1-x^n}{1-x}\left(A(x)\odot B(x)\right) \tag{2}$$

where, from a computational point of view, the left factor does not involve a true multiplication but rather n *shift/add* operations.

We also collect all other *direct* nodes, which are subtracting nodes reached whithout having previously walked on any branch-2 nodes. There are 2^k of them on the level k of the tree ($k=0$ corresponding to the first level under the root node). It is easy again to use the binary counterpart of their radix-3 label for correctly identifying them. On the level k of the tree, all terms of degree j are subtracted from the corresponding terms of degree $j+2^{k-1}$. Generating functions are used as masks for extracting the relevant terms (as well as for shifting and adding them):

$$\begin{aligned}
-\sum_{k=1}^{\log_2(n)}\sum_{j=0}^{2^{k-1}-1}&\frac{1-x^{2^{k-1}}}{(1-x)\,x^{2^{k-1}+j}}\\
&\left(\frac{1-x^n}{1-x^{2^k}}x^{2^{k-1}+j}\odot\left(1-x^{2^{k-1}}\right)A(x)\right)\\
&\times\left(\frac{1-x^n}{1-x^{2^k}}x^{2^{k-1}+j}\odot\left(1-x^{2^{k-1}}\right)B(x)\right)
\end{aligned}$$

where k is the level of each considered row of nodes, and j the rank of each *direct* subtracting node on the level k.

Since $2^{k-1}+j$ merely iterates over $1,2,3,\ldots,n-1$, it is easy to use a single summation for directly iterating over all the considered subtrees:

$$\begin{aligned}
-\sum_{m=1}^{n-1}&\frac{1-x^{2^{\lfloor\log_2(m)\rfloor}}}{(1-x)\,x^m}\\
&\left(\frac{(1-x^n)\,x^m}{1-x^{2^{\lfloor\log_2(m)\rfloor+1}}}\odot\left(1-x^{2^{\lfloor\log_2(m)\rfloor}}\right)A(x)\right)\\
&\times\left(\frac{(1-x^n)\,x^m}{1-x^{2^{\lfloor\log_2(m)\rfloor+1}}}\odot\left(1-x^{2^{\lfloor\log_2(m)\rfloor}}\right)B(x)\right)
\end{aligned} \tag{3}$$

Summing both parts (2) and (3) results in $A(x)\times B(x)$ and is more or less equivalent to Karatsuba's algorithm from a computational point of view — as long as some variable substitution is done before each recursive call in order to map sparse polynomials to new polynomials of smaller degree.

Using the previously described interleaved splitting scheme now allows to apply Karatsuba's recursive formula to infinite power series (which is not the case with the traditional splitting scheme).

In the formulas (2) and (3), all $1-x^n$ numerators in the generating functions are intended to truncate periodical sequences of unitary and null coefficients to the required length. Extending these formulas to infinite power series is then easily achieved by removing such numerators:

$$\begin{aligned} f(x) * g(x) = \frac{f(x) \odot g(x)}{1-x} - \sum_{m=1}^{\infty} \frac{1-x^{2^{\lfloor \log_2 m \rfloor}}}{(1-x)\, x^m} & \\ \left(\frac{x^m}{1-x^{2^{\lfloor \log_2 m \rfloor + 1}}} \odot \left(1 - x^{2^{\lfloor \log_2 m \rfloor}}\right) f(x) \right) & \\ * \quad \left(\frac{x^m}{1-x^{2^{\lfloor \log_2 m \rfloor + 1}}} \odot \left(1 - x^{2^{\lfloor \log_2 m \rfloor}}\right) g(x) \right) & \end{aligned} \tag{4}$$

It is also easy to build the set S_d of all indices m occuring in the previous formulas (3) or (4) which are involved in the resulting coefficient of degree d:

$$S_d = \left\{ m \,\middle|\, 1 \leqslant m \leqslant d, \left((d-m) \bmod 2^{\lfloor \log_2 m \rfloor + 1} \right) < 2^{\lfloor \log_2 m \rfloor} \right\} \tag{5}$$

The cardinality $|S_d|$ of such sets of indices is empirically found to be the sequence `A268289` in the *On-Line Encyclopedia of Integer Sequences*, namely the cumulated differences between the number of digits 1 and the number of digits 0 in the binary expansions of integers up to d[1]. An explicit expression can be given for $|S_d|$ by resorting to the τ Takagi function:

$$|S_d| = (d+1)(m-k+1) - (2 + \tau(\xi - 1))\, 2^m + 2^{k+1} - 1$$

with $k = \lfloor \log_2(d) \rfloor$, $m = \lfloor \log_2(d+1) \rfloor$ and $\xi = (d+1)2^{-m}$.

4 Fully flattening the recursion tree

Having described in the previous section how to handle two branches of the tree at once by using the termwise multiplication formula, we now go one step further. In order to finally flatten the whole tree, the initial formula (1) is rewritten with only two branches (by merging the previous branches 0 and 1),

[1] Other interesting integer sequences may be related to these consecutive sets, some being already known with other definitions and published in the *On-Line Encyclopedia of Integer Sequences*. Most of the conjectured identities gathered through experimental computations involve the binary expansion of integers; some give the required number of moves to solve the chinese rings puzzle, etc. They may be published separately later.

leading now to a more familiar binary tree:

$$AB = \underbrace{(X+1)\,(A_0B_0 + XA_1B_1)}_{\text{branch 0 (termwise)}} - \underbrace{X\,(A_1 - A_0)\,(B_1 - B_0)}_{\text{branch 1 (subtracting)}} \tag{6}$$

As shown above, the new first branch is called "termwise" because it will be flattened as the termwise product of $A = A_1X + A_0$ and $B = B_1X + B_0$; the new second branch is the "subtracting" one. Iterating over all nodes of the tree is achieved by using a suitable bit-testing function.

Discussing the whole process being now easier from a coding point of view, we give two pieces of pseudocode intended to be used with any computer algebra system handling the polynomial type; they do not focus on low-level implementation issues (how more or less sparse polynomials are internally represented in order to give the most efficient access to their coefficients). Termwise multiplication of two polynomials should of course be already implemented.

The following code shows how subtracting and masking factors are accumulated while iterating on the branches of the tree; the key idea is to use the $X+1$ factors for selecting subtracted terms as well as for propagating them and the $1-X$ corresponding factors for performing the subtractions:

```
Multiply(A, B)
d ← ⌈log₂(1 + max(deg A, deg B))⌉
n ← 2^d
s ← 0
for k ← 0 to n − 1
      do f ← 1
         f′ ← 1
         for j ← 0 to d − 1
              do if k & 2^j ≠ 0                     ▷ test if bit j of k is set
                    then f′ ← (1 − X^(2^j)) f′      ▷ subtracting factor
                    else  f ← (1 + X^(2^j)) f       ▷ termwise mask
         degree ← deg f′                            ▷ degree of f′
         lt ← coeff(f′, X, degree) X^degree         ▷ leading term in f′
         s ← s + f (f lt ⊙ f′A ⊙ f′B)
return s
```

where it can be seen that the variable f has two distinct purposes: an algebraic purpose coming from (6) but also a tracking purpose for identifying how many terms have to be kept in the termwise multiplication.

Trying to write down the previous algorithm as a mathematical formula is achievable but rather heavy if we stick to mimicking all bitwise operations. A more elegant approach is allowed by noticing that iterating with f over all divisors of some suitable polynomial leads to a more concise typesetting, though we still have to introduce some conventional notations:

$$A \times B \;=\; \sum_{\substack{f \,\in\, \mathbb{Z}[X],\\ f \,\mid\, \sum_{k=0}^{n-1} X^k}} f\left(f\,\dot{f}^{\,\mathrm{W}} \odot \dot{f}A \odot \dot{f}B\right) \tag{7}$$

with $\dot{f}$ selecting all unselected $1 + X^{2^k}$ factors in f and negating — for each one — the coefficient of their non constant term, and $\dot{f}^{\mathrm{W}}$ the leading term (including the coefficient) of $\dot{f}$. The superscript character W stands for "weight"[2].

Formula (7) can easily be adapted to infinite power series by changing the definition of the variable f to $f \in \mathbb{Z}[X], \exists n \in \mathbb{N} : f \mid \sum_{k=0}^{2^n-1} X^k$.

The previous pseudocode however loses all benefits of traditional implementations of Karatsuba's algorithm because the same subtractions are computed for distinct values of k. Fortunately iterating over the binary expansions of k by using the reflected binary code (Gray code) instead of the standard radix-2 labelling system preserves the required number of subtractions.

The following version, though not optimized by itself from an implementation point of view (because it still relies on high-level polynomial types), gives the prototype of a rather optimized iterative version of Karatsuba's algorithm:

MULTIPLY2(A, B)

$d \leftarrow \lceil \log_2(1 + \max(\deg A, \deg B)) \rceil$
$n \leftarrow 2^d$
$f \leftarrow (1 - X^n)/(1 - X)$ ▷ initial mask (all bits set)
$f' \leftarrow 1$
$g \leftarrow 0$ ▷ Gray-code counterpart of k
$s \leftarrow f\,(f \odot A \odot B)$ ▷ case $g = k = 0$
for $k \leftarrow 1$ **to** $n - 1$
  do $j \leftarrow \lfloor \log_2(k \text{ xor } k - 1) \rfloor$ ▷ least significant set bit in k
    if $g \mathbin{\&} 2^j = 0$ ▷ test if bit j has to be set in g
      then $f' \leftarrow (1 - X^{2^j})\, f'$ ▷ subtracting factor
        $f \leftarrow f/(1 + X^{2^j})$ ▷ termwise mask
      else $f' \leftarrow f'/(1 - X^{2^j})$ ▷ subtracting factor
        $f \leftarrow (1 + X^{2^j})\, f$ ▷ termwise mask
    $g \leftarrow g \text{ xor } 2^j$ ▷ update g (Gray-code of k)
    $\mathit{degree} \leftarrow \deg f'$ ▷ degree of f'
    $\mathit{lt} \leftarrow \textsc{coeff}(f', X, \mathit{degree})\, X^{\mathit{degree}}$ ▷ leading term in f'
    $s \leftarrow s + f\,(f\, \mathit{lt} \odot f'A \odot f'B)$
return s

A lower-level implementation of this pseudocode should avoid actually storing the f' polynomial in a separate buffer and computing both $f'A$ and $f'B$ products — the idea being rather to directly store $f'A$ and $f'B$, and merely update them at each step of the loop.

Furthermore, efficiently implementing the previous pseudocode should take care of the subtracting and adding steps: since polynomials become very sparse for some values of k, very few terms should be manipulated at these points. Two main directions should be explored for that purpose: using linked lists for representing polynomials or tracking the remaining non-null coefficients by using

[2]This symbol is compact but not very common; it can be found however in an article by Shigeru Kuroda, *Shestakov-Umirbaev reductions and Nagata's conjecture on a polynomial automorphism* (2007).

an elaborated system of strides[3]. Elementary multiplications should of course be aware of the mask to be applied in order to avoid useless computation.

References

[1] *The On-Line Encyclopedia of Integer Sequences*, published electronically at `https://oeis.org`.

[2] Richard Brent, Paul Zimmermann, *Modern Computer Arithmetic*, Cambridge, 2011.

[3] Ronald L. Graham, Donald E. Knuth, Oren Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, second edition, 1994.

[4] Anatoly Karatsuba, Yuri Ofman, *Multiplication of many-digital numbers by automatic computers*, Dokl. Akad. Nauk SSSR, 1962.

[5] Donald E. Knuth, *The Art of Computer Programming*, vol. 2 "Seminumerical Algorithms", third edition, 1998.

[6] Sergei K. Lando, *Lectures on Generating Functions*, American Mathematical Society, 2003.

[7] Richard P. Stanley, Sergey P. Fomin, *Enumerative Combinatorics*, vol. 2, Cambridge, 1999.

[3] This is one of the most important concepts behind the famous *Numpy* module for Python; strides allow to build views on parts of an existing array without actually copying it.