Karatsuba Multiplication

Qishen Pang
Khoury College of Computer Science
Northeastern University
Boston, MA
pang.qis@northeastern.edu

Jack Craciun
Khoury College of Computer Science
Northeastern University
Boston, MA
craciun.j@northeastern.edu

Jillian Baltrus
Khoury College of Computer Science
Northeastern University
Boston, MA
baltrus.j@northeastern.edu

ABSTRACT

Our primary focus is to implement the Karatsuba multiplication algorithm and prove its equivalence with standard multiplication. The Karatsuba algorithm performs efficient multiplication on two natural numbers by recursively breaking its factors in half and operating on each of the halves instead of the whole factors. The algorithm requires many helper functions and a unique representation of numbers as lists, which we implement in ACL2. In addition to implementing and demonstrating the correctness of the Karatsuba algorithm, our final code also includes theorems and functions relating to our novel data type for representing numbers.

CCS CONCEPTS

• Computing methodologies → Symbolic and algebraic manipulation → Symbolic and algebraic algorithms → Optimization algorithms

KEYWORDS

Karatsuba, Multiplication Algorithm, Long Multiplication

1 INTRODUCTION

Using an efficient algorithm to multiply two sufficiently large numbers can significantly reduce computation time. In this paper, we compare the computational complexity advantage of one multiplication algorithm, Karatsuba, over the standard multiplication algorithm and prove the two algorithms' equivalence. To do this, we first define Karatsuba's necessary helper functions and a novel data type that represents natural numbers as a backward list of the number's digits. We then implement the Karatsuba algorithm and demonstrate that its computational results are the same as the standard multiplication.

1.1 Long Multiplication[1]

The standard multiplication algorithm, known as long multiplication, multiplies each digit of the two factors. This algorithm performs n^2 single-digit multiplication operations when calculating the product of two numbers of size n. Therefore, for any two natural numbers X, Y with size n, the computational complexity of long multiplication is $O(n^2)$.

```
          1 2 3 4
×         5 6 7 8
─────────────────
          9 8 7 2  (= 1234 × 8)
        8 6 3 8    (= 1234 × 70)
      7 4 0 4      (= 1234 × 600)
+ 6 1 7 0          (= 1234 × 5000)
─────────────────
  7 0 0 6 6 5 2
```

Figure 1: Example of Long Multiplication

1.2 Karatsuba Algorithm

The Karatsuba algorithm is a fast multiplication algorithm that computes the product of two large natural numbers by repeatedly splitting the two factors. For any two natural numbers X, Y with size n $(n > 1)$:

$$X = a \times 10^{\frac{n}{2}} + b$$
$$\mathrm{Y} = \mathrm{c} \times 10^{\frac{n}{2}} + \mathrm{d}$$

Then the $X \cdot Y$ can be rewritten as:

$$X \cdot Y = ac \times 10^{n} + (ad + bc) \times 10^{\frac{n}{2}} + bd \quad (1)$$

[1] Long Multiplication is also known as grade-school multiplication and standard algorithm.

```
          a    b
×         c    d
-------------------
         ad   bd
+   ac   bc
-------------------
    ac ad+bc  bd
```

Figure 2: Rewrite $X \cdot Y$

Since the size of $a,\ b,\ c,\ d$ are $\frac{n}{2}$ $(i, e, .\ half\ size\ of\ X,\ Y)$, the computational complexity of each multiplication of ac, ad, bc, bd is $O\left(\frac{n^2}{4}\right)$. The calculation of $ad + bc$ requires two $O\left(\frac{n^2}{4}\right)$ multiplications and one $O(n)$ addition. One can rewrite this with a reduced computational complexity in the below form:

$$ad + bc = ac + ad + bc + bd - ac - bd$$
$$= (a + b)(c + d) - ac - bd$$

After rewriting, the computational complexity of $ad + bc$ reduced to one $O\left(\frac{n^2}{4}\right)$ multiplication with four $O(n)$ additions Let e be a variable such that:

$$e = ad + bc = (a + b)(c + d) - ac - bd \quad (2)$$

By substituting equation (2) into equation (1) we get the Karatsuba algorithm:

$$X \cdot Y = ac \times 10^n + e \times 10^{\frac{n}{2}} + bd \quad (3)$$

1.3 Time Complexity

The Karatsuba algorithm splits the multiplication $X \cdot Y$ into three multiplications in each recursive step. Because each step splits X *and* Y into halves with size $\frac{n}{2}$, the algorithm takes $log_2\, n$ steps to reach the base case[2].

Therefore, for X *and* Y with size $n = 2^k$ (k = $log_2\, n$), the algorithm will perform 3^k single-digit multiplications in total, which is equivalent to:

$$3^k = 3^{log_2 n} = 2^{log_2 3^{log_2 n}} = 2^{(\log_2 n \cdot \log_2 3)} = n^{\log_2 3}$$

and the computational complexity is:

$$T(n) = 3 \cdot T\left(\frac{n}{2}\right) + O(n) = O\left(n^{\log_2 3}\right) \approx O(n^{1.585})$$

Compare the computational complexity of long multiplication $O(n^2)$ and Karatsuba algorithm $O\left(n^{\log_2 3}\right)$: the computational complexity of the Karatsuba algorithm is less than the computational complexity of long multiplication as n grows.

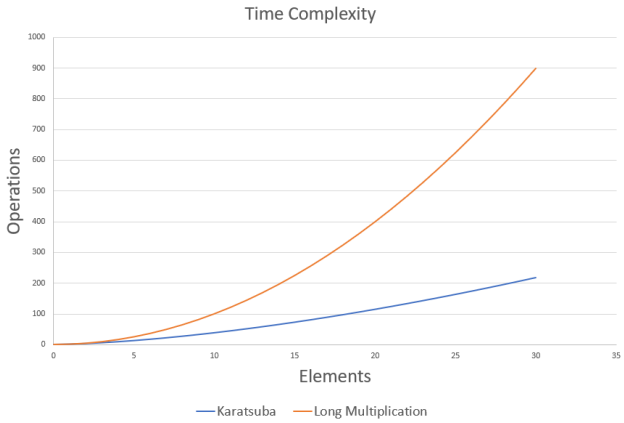

Figure 3: Computational Complexity Compare Between Karatsuba Algorithm and Long Multiplication

2 IMPLEMENT KARATSUBA IN ACL2

2.1 Define Data: Back-Nat

Before implementing the Karatsuba algorithm, we first must represent our data in the most efficient way. Rather than representing numbers in decimal form, we chose to represent numbers as a backward list of digits[3] that ends with a positive-digit[4] by defining a new data type called Back-Nat. The restriction that the final digit must be positive prevents Back-Nats from ending in 0, which would otherwise allow a single natural number to have infinite corresponding Back-Nat representations ending in an arbitrary amount of 0s.

```
(defconst *base* 10)
(defdata pos-digit (range integer (0 < _ < *base*)))
(defdata digit (oneof 0 pos-digit))
(defdata singleton-pos-nat (list pos-digit))
(defdata back-pos-nat (oneof (cons digit back-pos-nat) singleton-pos-nat))
(defdata back-nat (oneof nil back-pos-nat))
```

Figure 4: Back-Nat Definition

Operating on Back-Nats instead of decimal numbers is the most efficient choice because the Karatsuba algorithm calculates the length of a number as well as breaks it in half,

[2] Karatsuba works for any base. In this paper, the base case is when both X and Y size is equal to 1 (n = 1), i.e., single-digit multiplication is used as the base.

[3] A digit is a natural number less than 10.

[4] A positive-digit/pos-digit is an integer between 0 and 10 (exclusive).

and implementing these functionalities for a list rather than a regular number is simpler in ACL2. We choose to represent the numbers backwards because this made traversing through a list in recursive functions cleaner.

Back-Nat example:
‘() == 0
‘(4 3 2 1) == 1234

Invalid Back-Nat example:
‘(0) is not a Back-Nat
‘(43 2 1) is not a Back-Nat
‘(4 3 2 1 0) is not a Back-Nat

Figure 5: Back-Nat Example

2.2 Conversion Between Decimal Numbers and Back-Nat

Since we aim to prove that the Karatsuba algorithm implementation in ACL2 calculates equivalent results to standard multiplication, we first define the conversion function between decimal natural numbers and Back-Nats. The conversion from Back-Nat to natural numbers is relatively simple: we take the first element from Back-Nat and add it to the remaining natural number represented by Back-Nat times ten. To convert a natural number to a Back-Nat, we take the last digit of the natural number and cons it onto the Back-Nat represented by the remaining part of the natural number

```
;; converts back-pos-nat to regular old decimal representation
(definec back-pos-nat-to-pos (n1 :back-pos-nat) :pos
  (declare (xargs :measure-debug t :guard-debug t))
  (cond
   ((singleton-pos-natp n1) (car n1))
   (t (+ (car n1) (* (back-pos-nat-to-pos (cdr n1)) *base*)))))

;; converts back-nat to regular old decimal representation
(definec back-nat-to-nat (n1 :back-nat) :nat
  (declare (xargs :measure-debug t :guard-debug t))
  (cond
   ((lendp n1) 0)
   (t (back-pos-nat-to-pos n1))))

;; converts decimal representation to back-nat
(definec nat-to-back-nat (n :nat) :back-nat
  (cond
   ((zp n) '())
   (t (cons (mod n *base*) (nat-to-back-nat (floor n *base*))))))
```

Figure 6: Nat Back-Nat Conversion Functions in ACL2

After implementing the conversion functions, we prove the correctness of the conversion between the natural numbers and Back-Nat.

```
;; Lemma2. Round-trip conversion between Back-Nat and Nat
(defthm round-trip
  (implies (natp n)
           (equal (back-nat-to-nat (nat-to-back-nat n)) n)))
```

Figure 7: Theorem Stating Round-trip Conversion Between Back-Nat and Nat

In addition, we prove the one-to-one conversion between Back-Nat and natural numbers through ACL2: if there are two equal natural numbers $n1$ and $n2$, then the Back-Nat represented by these two natural numbers should also be equal. We also prove the converse: if two Back-Nats are equal, then the natural numbers represented by these Back-Nats are equal as well.

```
;; Lemma3. One-to-one conversion between Back-Nat and Nat
(defthm one-to-one-pt2
  (implies (and (natp n1)
                (natp n2)
                (equal n1 n2))
           (equal (nat-to-back-nat n1) (nat-to-back-nat n2))))
```

Figure 8: Theorem Stating Round-trip Conversion Between Back-Nat and Nat

2.3 Helper-Functions

With the new data representation, we then implement the Karatsuba algorithm. This function requires many helper functions, including adding, subtracting, and multiplying Back-Nats, multiplying a Back-Nat to a power of 10, calculating the size of a Back-Nat, and taking a sub-list of a Back-Nat. We initially thought that implementing these functions would be trivial. For example, our original idea for multiplying a Back-Nat by a power of ten was to simply append 0s onto the beginning of the Back-Nat. However, we encountered obstacles while implementing the *backnat-expt* function. ACL2 cannot prove the termination and function-contract of *backnat-expt* function on its own. To help ACL2 prove the termination of the *backnat-expt* function, we implement the *backnat-expt-measure* function, which accepts a Back-Nat and a natural number, and returns the accepted natural number. Proving the function-contract requires more steps. We first define the *not-zero-nat-sub1-is-still-nat* lemma, which states that a positive natural number minus one is still a natural number. Then, since ACL2 does not induct at a step that can be easily proven with induction, we also instruct ACL2 to induct at *Goal '5'*.

```
; multiplies backnat by 10 to the given power, used in karatsuba algo
(definec backnat-expt (backnat :back-nat nat :nat) :back-nat
  :function-contract-hints (("Goal" :use ((:instance not-zero-nat-sub1-is-still-nat)))
                            ("Goal'5'" :induct t))
  (declare (xargs :measure (if (and (back-natp backnat)
                                    (natp nat))
                               (backnat-expt-measure backnat nat)
                               0)
                  :hints (("goal" :do-not-induct t))))
  (cond
   ((endp backnat) nil)
   ((zerop nat) backnat)
   (t (cons 0 (backnat-expt backnat (- nat 1))))))
```

Figure 9: Function Multiply Back-Nat by Power of Ten

Calculating the size of a Back-Nat is the same as simply finding the length of the Back-Nat list and can be easily done with the *len2* function. Then we implement *my-n* function which divides the result of *size* function and round up.

```
;; classic len2 function, returns length of list
(definec len2 (x :tl) :nat
  (if (endp x)
      0
    (+ 1 (len2 (rest x)))))

; given 2 lists, returns the size of the longer list
(definec size (n1 :tl n2 :tl) :nat
  (max (len2 n1) (len2 n2)))

; given 2 backnats, computes the midpoint index of the longer backnat
; (used in karatsuba algo for breaking numbers in half)
(definec my-n (n1 :back-nat n2 :back-nat) :nat
  (ceiling (/ (size n1 n2) 2) 1))
```

Figure 10: Function Returns the Size of The Bigger Back-Nat

Taking a sub-list of the Back-Nat is also somewhat trivial. We do this with a function that accepts a start and end index[5] and recursively *conses* elements of the Back-Nat onto the resultant list if the element falls into the range of indices. We also encountered some difficulties when implementing the *sub-list* function. Under normal circumstances, the start and end index is not larger than the size of the given Back-Nat. However, since there is no limit to the size of the two Back-Nats accepted by our Karatsuba, the *my-n* function could output a value with a start or end index larger than the size of the given Back-Nat. To solve this problem, we define the *sub-list-helper* function, which rejects any start or end index larger than the size of the given Back-Nat. We also define the *sub-list* function to return nil when the start index is greater than the end index or the size of the input Back-Nat, and set the end index to the size of the given Back-Nat if the end index is greater than the size of the input Back-Nat.

```
; used in algo for breaking numbers in half
(definec sub-list-helper (backnat :back-nat start :nat end :nat) :back-nat
  :ic (and (<= start end)
           (not (== 0 start))
           (not (== 0 end))
           (<= end (len2 backnat)))
  (cond
   ((and (== start 1) (== end 1)) (if (== 0 (car backnat))
                                      nil
                                      (list (car backnat))))
   ((== start 1)
    (let ((res (sub-list-helper (cdr backnat) 1 (- end 1))))
      (cond
       ((and (== (car backnat) 0) (endp res)) res)
       (t (cons (car backnat) res)))))
   (t (sub-list-helper (cdr backnat) (- start 1) (- end 1)))))

; used in algo for breaking numbers in half
(definec sub-list (backnat :back-nat start :nat end :nat) :back-nat
  :ic (not (== 0 start))
  (cond
   ((== 0 end) nil)
   ((or (> start end) (> start (len2 backnat))) nil)
   ((> end (len2 backnat)) (sub-list-helper backnat start (len2 backnat)))
   (t (sub-list-helper backnat start end))))
```

[5] The index starts from 1. For example, call sub-list function on a Back-Nat (list 4 3 2 1) with start index 1 and end index 2 would return a list (list 4 3).

Figure 11: Function Returns A Sub-List from Back-Nat

Simple math operations such as addition, subtraction and multiplication are more complicated to implement. Our addition and subtraction functions operate in similar ways. They both traverse through the Back-Nat digit by digit and subtract or add each corresponding digit until one of the lists is empty, and recursively build the result by cons-ing the resulting digit onto the resultant Back-Nat. Initially, we used accumulators in both the *add-back-nat* and *back-nat-sub* function to indicate whether the previous addition or subtraction overflowed. However, ACL2 did not accept the implementations that used accumulators. Fortunately, with the help of Professor Hemann, we replace the accumulators with two helper functions called *back-nat-add1* and *back-nat-sub1*. These new helper functions address overflow by adding or subtracting one from the rest of the first input Back-Nat.

```
;; add two back-nat together
(definec add-back-nat (n1 :back-nat n2 :back-nat) :back-nat
  (cond
   ((endp n1) n2)
   ((endp n2) n1)
   (t (if (< (+ (car n1) (car n2)) *base*)
          (cons (+ (car n1) (car n2)) (add-back-nat (cdr n1) (cdr n2)))
          (cons (- (+ (car n1) (car n2)) *base*)
                (add-back-nat (back-nat-add1 (cdr n1)) (cdr n2)))))))
```

Figure 12: Function Add Two Back-Nat

In addition, we prove the add equivalent of the *add-back-nat* function and the normal add function predefined in ACL2.

```
;; Lemma4. add two back-pos-nat will return a cons
(defthm add-equiv-help
  (IMPLIES (AND (BACK-POS-NATP N1)
                (BACK-POS-NATP N2)
                (CONSP N1)
                (CONSP N2))
           (CONSP (ADD-BACK-NAT N1 N2))))

;; Lemma5. add two back-nats is equiv to add two decimal natural numbers
(defthm add-equiv
  (implies (and (back-natp n1)
                (back-natp n2))
           (equal (+ (back-nat-to-nat n1) (back-nat-to-nat n2))
                  (back-nat-to-nat (add-back-nat n1 n2))))
  :hints (("Goal" :use ((:instance add-equiv-help)))))
```

Figure 13: Theorem Stating Add Two Back-Nat Is Equivalent to Add Two Natural Numbers

The implementation of the *back-nat-sub* function is more complex than the *back-nat-add*. Under the special case when the two input Back-Nats are the same, the *back-nat-sub* function should return an empty list instead of a list of 0s. Therefore, when the *car* of the two input Back-Nats are equal, we check if subtracting the *cdrs* of the two input Back-Nats would return an empty list. If so, we simply return an empty list. Otherwise, we *cons* 0 on to the result. Unfortunately, we are still missing body and function-contract proofs for the *back-nat-sub* function. In order for the *back-nat-sub* function to work, we need to set all the *contract-strict*s to nil.

Therefore, we cannot prove any theorem or lemma that uses the *back-nat-sub* function.

```
;; subtracts a - b (numbers represented as back-nats)
(definec back-nat-sub (a :back-nat b :back-nat) :back-nat
  :ic (back-nat-<= b a)
  :body-contracts-hints (("Goal"
                          :use
                          ((:instance
                            backnat-a->=-b-and-unempty-b-implies-backposnat-a)))
                         ("Goal" :use
                                 ((:instance cdr-of-backnat-is-backnat))))
  (cond
   ((endp b) a)
   ((< (car b) (car a)) (cons (- (car a) (car b))
                              (back-nat-sub (cdr a) (cdr b))))
   ((= (car b) (car a))
    (let ((res (back-nat-sub (cdr a) (cdr b))))
      (cond
       ((endp res) nil)
       (t (cons 0 res)))))
   (t (cons (- (+ *base* (car a)) (car b))
            (back-nat-sub (back-nat-sub1 (cdr a)) (cdr b))))))
```

Figure 14: Function Subtracts Two Back-Nat

Multiplication has a somewhat simpler implementation since the Karatsuba algorithm only multiples numbers with a single digit (this happens at the base case). Therefore, in our multiplication function, we simply multiply the car of the two Back-Nats, and if the product has overflow—meaning it cannot be represented by a single digit—we form the resultant Back-Nat by creating a list from the remainder of the product divided by 10 onto the floor of the product divided by 10. Since the *mult-back-nat* function is non-recursive, we initially thought that this function would go through ACL2 trivially. However, ACL2 cannot directly prove the termination of the *mult-back-nat* function. Therefore, to help ACL2 prove the function-contract, we define the *floor-of-mult-two-posdigit-bigger-than-10-is-posdigit* lemma, which states that when the result of multiplying two pos-digits is greater than ten, taking the floor of this product divided by ten will result in a pos-digit.

```
; multiplies 2 backnats. We will only ever need to call this function
; on backnats that are empty or have length 1.
(definec mult-back-nat (n1 :back-nat n2 :back-nat) :back-nat
  :ic (and (<= (len2 n1) 1) (<= (len2 n2) 1))
  :function-contract-hints
  (("Goal"
    :use
    ((:instance
      floor-of-mult-two-posdigit-bigger-than-10-is-posdigit))))
  (cond
   ((or (endp n1) (endp n2)) '())
   ((< (* (car n1) (car n2)) *base*)
    (cons (* (car n1) (car n2)) nil))
   (t (cons (mod (* (car n1) (car n2)) *base*)
            (cons (floor (* (car n1) (car n2)) *base*) nil)))))
```

Figure 15: Function Multiplies Two Back-Nat With Length Smaller or Equal to Two

2.4 Karatsuba

After implementing all the helper functions, we were left with the daunting task of implementing Karatsuba itself. We adapted the algorithm to use our Back-Nat data type and our Back-Nat helper functions. Unfortunately, ACL2 would timeout during the rewriting and preprocessing step. Therefore, we need to set the *termination-strictp*, *function-contract-strictp* and *body-contracts-strictp* to nil in order to let the *karatsuba* function pass.

```
; the algorithm itself
(definec karatsuba (n1 :back-nat n2 :back-nat) :back-nat
  (cond
   ((and (<= (len2 n1) 1) (<= (len2 n2) 1)) (mult-back-nat n1 n2))
   (t (add-back-nat
       (add-back-nat
        (backnat-expt (karatsuba (sub-list n1 (+ (my-n n1 n2) 1) (len2 n1))
                                 (sub-list n2 (+ (my-n n1 n2) 1) (len2 n2)))
                      (* 2 (my-n n1 n2)))
        (backnat-expt (back-nat-sub
                       (back-nat-sub
                        (karatsuba (add-back-nat (sub-list n1 (+ (my-n n1 n2) 1) (len2 n1))
                                                 (sub-list n1 1 (my-n n1 n2)))
                                   (add-back-nat (sub-list n2 (+ (my-n n1 n2) 1) (len2 n2))
                                                 (sub-list n2 1 (my-n n1 n2))))
                        (karatsuba (sub-list n1 (+ (my-n n1 n2) 1) (len2 n1))
                                   (sub-list n2 (+ (my-n n1 n2) 1) (len2 n2))))
                       (karatsuba (sub-list n1  1 (my-n n1 n2))
                                  (sub-list n2  1 (my-n n1 n2))))
                      (my-n n1 n2)))
       (karatsuba (sub-list n1 1 (my-n n1 n2))
                  (sub-list n2 1 (my-n n1 n2)))))))

(test? (implies (and (back-natp x)
                     (back-natp y))
                (equal (nat-to-back-nat (* (back-nat-to-nat x) (back-nat-to-nat y)))
                       (karatsuba x y))))
```

Figure 16: Karatsuba Function

3 Personal Progress

The original goal was to implement the Karatsuba algorithm and prove its equality to regular built-in multiplication on natural numbers using a theorem in ACL2. Our final code almost achieves this goal. ACL2 only accepts the subtraction helper function and Karatsuba algorithm after setting *termination-strictp*, *function-contract-strictp* and *body-contracts-strictp* to *nil*. Because ACL2 will not accept a theorem that uses a function without its function contract proven, we cannot define a theorem that shows the equivalence of Karatsuba multiplication and regular multiplication. Instead, our final code includes a *test?* statement that shows the equivalence.

```
(test? (implies (and (back-natp x)
                     (back-natp y))
                (equal (nat-to-back-nat (* (back-nat-to-nat x) (back-nat-to-nat y)))
                       (karatsuba x y))))
```

Figure 17: Test Karatsuba Function Equality

Although this is not the ideal outcome, the passable *test?* statement is very similar to the theorem we hoped to prove and still demonstrates the equality between Karatsuba multiplication and regular multiplication.

Moreover, the final code also includes many interesting data structures, lemmas, and theorems beyond Karatsuba. Therefore, the final code achieves some originally unintended goals, such as the functions and properties proven about Back-Nats. After implementing many functions that operate on Back-Nats, curiosity led us to define theorems about their properties. Using functions that convert between Back-Nats and decimal natural numbers, we define two *round-trip* theorems. These prove the equality of a natural number and

its Back-Nat representation converted back into a natural number, as well as the equality of a Back-Nat and its regular decimal representation converted back into a Back-Nat. We also prove that converting two equal decimal numbers into Back-Nats results in two equal Back-Nats, and converting two equal Back-Nats into decimal numbers results in two equal decimal numbers. Finally, we also prove that the sum of two Back-Nats is equal to the sum of two equivalent decimal numbers. Although not part of the original plan and not necessary to prove Karatsuba, these theorems about Back-Nats and their relationships to decimal numbers supplement and enhance the final code.

Despite our general success, our results have a few limitations. As previously mentioned, ACL2 does not prove every function—in particular, the Karatsuba and subtraction functions. Thus, proving other theorems that use these operations and generalizing about them is difficult because of ACL2's uncertainty about their correctness. Furthermore, our Back-Nat representation is limited to natural numbers. Thus, the properties proven about them cannot be generalized to negative integers. This is unproblematic when using Back-Nats in the context of Karatsuba multiplication because the algorithm only works for natural numbers, but the Back-Nat theorems unrelated to Karatsuba are therefore limited.

4 CONCLUSIONS

Our final code shows that the product of two natural numbers using standard multiplication equals the result of passing their corresponding Back-Nat representations into the Karatsuba function. Because the Karatsuba algorithm and its equivalence to standard multiplication is not new, the novelty of our code comes from the ability to implement Karatsuba multiplication, its necessary functions and lemmas, the Back-Nat representation, and the Back-Nat properties in ACL2. Our implementation is educational because it clearly decomposes the Karatsuba algorithm into its smaller subroutines. Each line of the algorithm uses one or more helper functions to operate on the given Back-Nats. This allows us to see the algorithm broken down step-by-step and examine what each part does at an extremely magnified level, thus providing a deep insight into how the Karatsuba algorithm functions. Furthermore, our Back-Nat implementation is a useful and versatile representation of natural numbers. They can be useful in functions that operate on natural numbers and are difficult to implement using regular decimal numbers. This situation motivates using Back-Nats in the Karatsuba algorithm: many of the helper functions cleanly traverse through the Back-Nat list structure, rather than awkwardly break down a regular decimal number. Thus, the Back-Nat data structure may be a valuable contribution for implementing functions in the future, especially given the theorems proven about Back-Nats and their relationship to decimal natural numbers.

With our working but unproven Karatsuba function, we hope to leave an entirely provable implementation in close reach. Hopefully, this complete implementation would enable ACL2 to accept a theorem, rather than a *test?* statement, proving the equality of the product of two decimal natural numbers and two Back-Nats multiplied with the Karatsuba function. This would likely require a provable implementation of Back-Nat subtraction, another open issue. With a subtraction function accepted by ACL2, proving the equivalence of Back-Nat subtraction and decimal subtraction is another potential future goal. Also, future research could expand the definition of Back-Nats to include negative integers as well. This would expand the versatility of the data structure. Our resultant code contributes to the conquering of these areas of further research and investigation, which are likely within close reach.

5 OUR CODE

Here is a link to our code:
https://github.com/jillbaltrus/karatsuba

REFERENCES

[1] Bernadette Shade, Craig Rodkin, 2021. *ACM Primary Article Template: New Workflow for ACM Publications*. Association for Computing Machinery, DOI: https://www.acm.org/publications/proceedings-template

[2] GeeksForGeekes. 2019. Karatsuba algorithm for fast multiplication using Divide and Conquer algorithm. DOI: https://www.geeksforgeeks.org/karatsuba-algorithm-for-fast-multiplication-using-divide-and-conquer-algorithm/