

**SCHOOL OF COMPUTING**

**DEPARTMENT OF SOFTWARE ENGINEERING, INFORMATION TECHNOLOGY, COMPUTER SCIENCE AND ENGINEERING**

**FACULTY OF ENGINEERING & TECHNOLOGY**

**MINI PROJECT REPORT**

**SUBJECT TITLE: ALGORITHM DESIGN AND ANALYSIS**

**SUBJECT CODE: 15CS204J**

**KARATSUBA ALGORITHM FOR FAST MULTIPLICATION: A DETAILED INSIGHT INTO THE ALGORITHM AND ITS EFFICIENCY**

**TEAM MEMBERS**

**SANKALPA SARKAR RA1711003010017**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**



**SRM Institute of Science and Technology**

**SRM Nagar, Kattankulathur-603203**

**Kancheepuram District, Tamil Nadu**

**INDEX**

|  |  |  |
| --- | --- | --- |
| **S.No** | **TOPIC** | **Page No** |
| **1** | Problem Definition | 4 |
| **2** | Problem Explanation | 4 |
| **3** | History | 5 |
| **4** | Design Technique | 6 |
| **5** | Pseudo-code | 7 |
| **6** | Explanation of pseudo-code | 7 |
| **7** | Program in C++ | 10 |
| **8** | Output screenshot | 12 |
| **9** | Complexity Analysis | 12 |
| **10** | Conclusion | 13 |

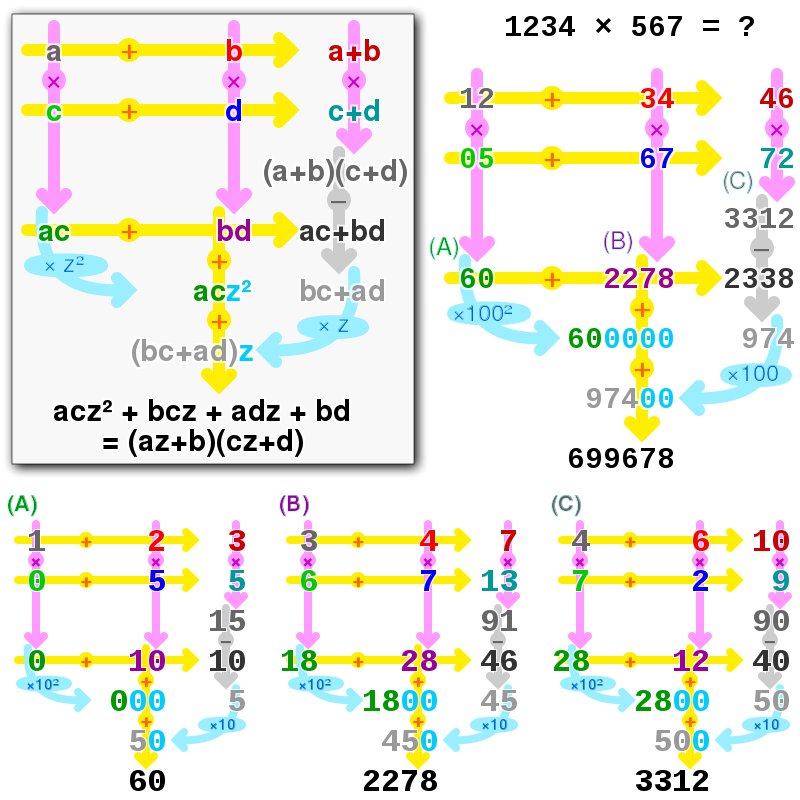
**PROBLEM DEFINITION**

Given two binary strings that represent value of two integers, find the product of two strings. For example, if the first bit string is “1100” and second bit string is “1010”, output should be 120.

**PROBLEM EXPLANATION**

The Karatsuba algorithm is a fast multiplication algorithm. It was discovered by Anatoly Karatsuba in 1960 and published in 1962. It reduces the multiplication of two n-digit numbers to at most single-digit multiplications in general (and exactly when n is a power of 2). It is therefore faster than the classical algorithm, which requires single-digit products. For example, the Karatsuba algorithm requires 310 = 59,049 single-digit multiplications to multiply two 1024-digit numbers (n = 1024 = 210), whereas the classical algorithm requires (210)2 = 1,048,576 (a speedup of 17.75 times).

The Karatsuba algorithm was the first multiplication algorithm asymptotically faster than the quadratic "grade school" algorithm. The Toom–Cook algorithm (1963) is a faster generalization of Karatsuba's method, and the Schönhage–Strassen algorithm (1971) is even faster, for sufficiently large n.



Karatsuba multiplication of az+b and cz+d (boxed), and 1234 and 567. Magenta arrows denote multiplication, amber denotes addition, silver denotes subtraction and cyan denotes shift. (A), (B) and (C) show recursion used to obtain intermediate values.

**HISTORY**

The standard procedure for multiplication of two n-digit numbers requires a number of elementary operations proportional to , or in big-O notation. Andrey Kolmogorov conjectured that the classical algorithm was asymptotically optimal, meaning that any algorithm for that task would require elementary operations.

In 1960, Kolmogorov organized a seminar on mathematical problems in cybernetics at the Moscow State University, where he stated the conjecture and other problems in the complexity of computation. Within a week, Karatsuba, then a 23-year-old student, found an algorithm (later it was called "divide and conquer") that multiplies two n-digit numbers in elementary steps, thus disproving the conjecture. Kolmogorov was very excited about the discovery; he communicated it at the next meeting of the seminar, which was then terminated. Kolmogorov gave some lectures on the Karatsuba result at conferences all over the world and published the method in 1962, in the Proceedings of the USSR Academy of Sciences. The article had been written by Kolmogorov and contained two results on multiplication, Karatsuba's algorithm and a separate result by Yuri Ofman; it listed "A. Karatsuba and Yu. Ofman" as the authors. Karatsuba only became aware of the paper when he received the reprints from the publisher.

**DESIGN TECHNIQUE USED**

Divide and conquer design technique has been used to solve this problem. Divide and conquer is an [algorithm design paradigm](https://en.wikipedia.org/wiki/Algorithm_design_paradigm)-based on multi-branched [recursion](https://en.wikipedia.org/wiki/Recursion). A divide-and-conquer [algorithm](https://en.wikipedia.org/wiki/Algorithm) works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

This divide-and-conquer technique is the basis of efficient algorithms for all kinds of problems, such as [sorting](https://en.wikipedia.org/wiki/Sorting_algorithm) (e.g., [quicksort](https://en.wikipedia.org/wiki/Quicksort), [merge sort](https://en.wikipedia.org/wiki/Merge_sort)), [multiplying large numbers](https://en.wikipedia.org/wiki/Multiplication_algorithm) (e.g. the [Karatsuba algorithm](https://en.wikipedia.org/wiki/Karatsuba_algorithm)), finding the [closest pair of points](https://en.wikipedia.org/wiki/Closest_pair_of_points_problem), [syntactic analysis](https://en.wikipedia.org/wiki/Syntactic_analysis) (e.g., [top-down parsers](https://en.wikipedia.org/wiki/Top-down_parser)), and computing the [discrete Fourier transform](https://en.wikipedia.org/wiki/Discrete_Fourier_transform) ([FFT](https://en.wikipedia.org/wiki/Fast_Fourier_transform)). The correctness of a divide-and-conquer algorithm is usually proved by [mathematical induction](https://en.wikipedia.org/wiki/Mathematical_induction), and its computational cost is often determined by solving [recurrence relations](https://en.wikipedia.org/wiki/Recurrence_relation).

**PSEUDO-CODE**

procedure karatsuba(num1, num2)

**if** (num1 < 10) or (num2 < 10)

**return** num1\*num2

*/\* calculates the size of the numbers \*/*

m = min(size\_base10(num1), size\_base10(num2))

m2 = floor(m/2)

*/\*m2 = ceil(m/2) will also work \*/*

*/\* split the digit sequences in the middle \*/*

high1, low1 = split\_at(num1, m2)

high2, low2 = split\_at(num2, m2)

*/\* 3 calls made to numbers approximately half the size \*/*

z0 = karatsuba(low1, low2)

z1 = karatsuba((low1 + high1), (low2 + high2))

z2 = karatsuba(high1, high2)

**return** (z2 \* 10 ^ (m2 \* 2)) + ((z1 - z2 - z0) \* 10 ^ m2) + z0

**EXPLANATION OF PSEUDO-CODE**

The basic step of Karatsuba's algorithm is a formula that allows one to compute the product of two large numbers x and y using three multiplications of smaller numbers, each with about half as many digits as x or y, plus some additions and digit shifts. This basic step is, in fact, a generalization of Gauss's complex multiplication algorithm, where the imaginary unit i is replaced by a power of the base.

Let x and y be represented as n-digit strings in some base B. For any positive integer m less than n, one can write the two given numbers as

where and are less than . The product is then

where

These formulae require four multiplications and were known to Charles Babbage. Karatsuba observed that xy can be computed in only three multiplications, at the cost of a few extra additions. With and as before one can observe that

An issue that occurs, however, when computing is that the above computation of and may result in overflow (will produce a result in the range 0<=result<=2Bm), which require a multiplier having one extra bit. This can be avoided by noting that

This computation of and will produce a result in the range of -Bm < result< Bm. This method may produce negative numbers, which require one extra bit to encode signedness, and would still require one extra bit for the multiplier. However, one way to avoid this is to record the sign and then use the absolute value of and to perform an unsigned multiplication, after which the result may be negated when both signs originally differed. Another advantage is that even though  {\displaystyle (x\_{0}-x\_{1})(y\_{1}-y\_{0})} may be negative, the final computation of only involves additions.

**Example:**

To compute the product of 12345 and 6789, where B = 10, choose m = 3. Then we decompose the input operands using the resulting base (Bm = 1000), as:

12345 = 12 · 1000 + 345

6789 = 6 · 1000 + 789

Only three multiplications, which operate on smaller integers, are used to compute three partial results:

z2 = 12 × 6 = 72

z0 = 345 × 789 = 272205

z1 = (12 + 345) × (6 + 789) − z2 − z0 = 357 × 795 − 72 − 272205 = 283815 − 72 − 272205 = 11538

We get the result by just adding these three partial results, shifted accordingly (and then taking carries into account by decomposing these three inputs in base 1000 like for the input operands):

result = z2 · (Bm)2 + z1 · (Bm)1 + z0 · (Bm)0, i.e.

result = 72 · 10002 + 11538 · 1000 + 272205 = 83810205.

Note that the intermediate third multiplication operates on an input domain which is less than two times larger than for the two first multiplications, its output domain is less than four times larger, and base-1000 carries computed from the first two multiplications must be taken into account when computing these two subtractions.

**Recursive application:**

If n is four or more, the three multiplications in Karatsuba's basic step involve operands with fewer than n digits. Therefore, those products can be computed by [recursive](https://en.wikipedia.org/wiki/Recursion) calls of the Karatsuba algorithm. The recursion can be applied until the numbers are so small that they can (or must) be computed directly.

In a computer with a full 32-bit by 32-bit [multiplier](https://en.wikipedia.org/wiki/Binary_multiplier), for example, one could choose B = 231 = 2147483648, and store each digit as a separate 32-bit binary word. Then the sums x1 + x0 and y1 + y0 will not need an extra binary word for storing the carry-over digit (as in [carry-save adder](https://en.wikipedia.org/wiki/Carry-save_adder)), and the Karatsuba recursion can be applied until the numbers to multiply are only one-digit long.

**IMPLEMENTATION OF ALGORITHM IN C++ LANGUAGE**

// C++ implementation of Karatsuba algorithm for bit string multiplication.

#include<iostream>   
#include<stdio.h>   
using namespace std;   
int makeEqualLength(string &str1, string &str2)   
{   
 int len1 = str1.size();   
 int len2 = str2.size();   
 if (len1 < len2)   
 {   
 for (int i = 0 ; i < len2 - len1 ; i++)   
 str1 = '0' + str1;   
 return len2;   
 }   
 else if (len1 > len2)   
 {   
 for (int i = 0 ; i < len1 - len2 ; i++)   
 str2 = '0' + str2;   
 }   
 return len1; // If len1 >= len2   
}   
// The main function that adds two bit sequences and returns the addition   
string addBitStrings( string first, string second )   
{   
 string result; // To store the sum bits   
 // make the lengths same before adding   
 int length = makeEqualLength(first, second);   
 int carry = 0; // Initialize carry   
 // Add all bits one by one   
 for (int i = length-1 ; i >= 0 ; i--)  
 {   
 int firstBit = first.at(i) - '0';   
 int secondBit = second.at(i) - '0';   
 // boolean expression for sum of 3 bits   
 int sum = (firstBit ^ secondBit ^ carry)+'0';  
 result = (char)sum + result;   
 // boolean expression for 3-bit addition   
 carry = (firstBit&secondBit) | (secondBit&carry) | (firstBit&carry);  
 }

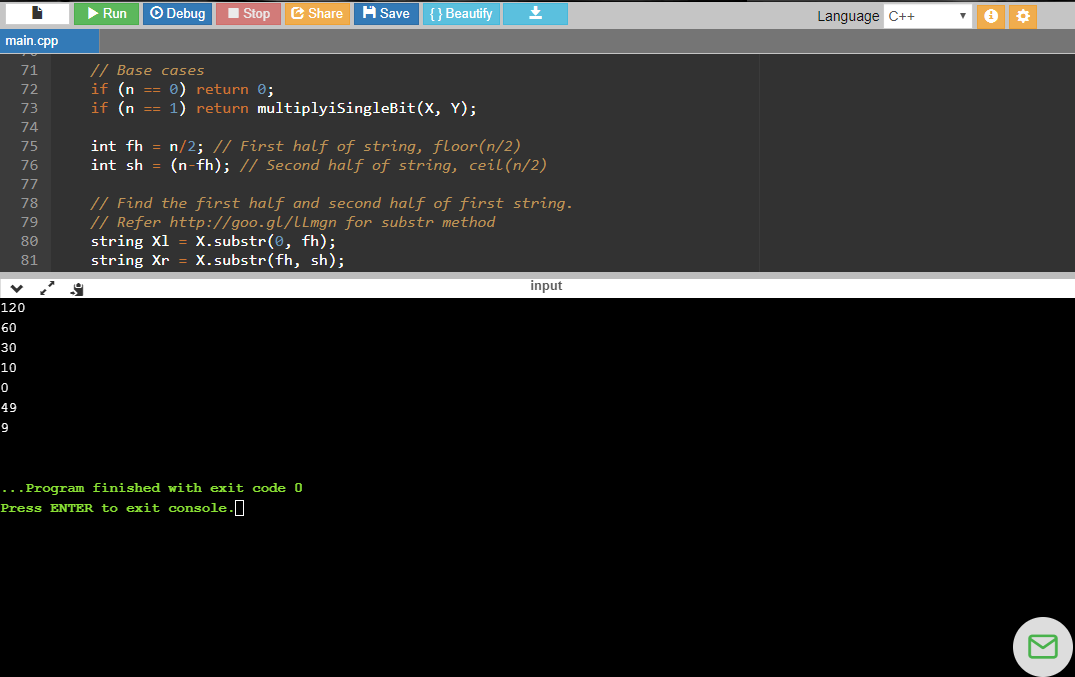
// if overflow, then add a leading 1   
 if (carry) result = '1' + result;   
 return result;

}   
// A utility function to multiply single bits of strings a and b   
int multiplyiSingleBit(string a, string b)   
{  
 return (a[0] - '0')\*(b[0] - '0');   
}   
// The main function that multiplies two bit strings X and Y and returns   
// result as long integer   
long int multiply(string X, string Y)   
{   
 // Find the maximum of lengths of x and Y and make length   
 // of smaller string same as that of larger string   
 int n = makeEqualLength(X, Y);   
 // Base cases   
 if (n == 0)   
 return 0;   
 if (n == 1)   
 return multiplyiSingleBit(X, Y);   
 int fh = n/2; // First half of string, floor(n/2)   
 int sh = (n-fh); // Second half of string, ceil(n/2)   
 // Find the first half and second half of first string.   
 string Xl = X.substr(0, fh);   
 string Xr = X.substr(fh, sh);   
 // Find the first half and second half of second string   
 string Yl = Y.substr(0, fh);  
 string Yr = Y.substr(fh, sh);   
 // Recursively calculate the three products of inputs of size n/2   
 long int P1 = multiply(Xl, Yl);   
 long int P2 = multiply(Xr, Yr);   
 long int P3 = multiply(addBitStrings(Xl, Xr), addBitStrings(Yl, Yr));   
 // Combine the three products to get the final result.   
 return P1\*(1<<(2\*sh)) + (P3 - P1 - P2)\*(1<<sh) + P2;

}   
// Driver program to test above functions   
int main()   
{   
 printf ("%ld\n", multiply("1100", "1010"));   
 printf ("%ld\n", multiply("110", "1010"));   
 printf ("%ld\n", multiply("11", "1010"));   
 printf ("%ld\n", multiply("1", "1010"));   
 printf ("%ld\n", multiply("0", "1010"));   
 printf ("%ld\n", multiply("111", "111"));   
 printf ("%ld\n", multiply("11", "11"));

}

**OUTPUT SCREENSHOT**



**COMPLEXITY ANALYSIS**

Time complexity of the above solution is O(n1.59).

Karatsuba's basic step works for any base B and any m, but the recursive algorithm is most efficient when m is equal to n/2, rounded up. In particular, if n is 2k, for some integer k, and the recursion stops only when n is 1, then the number of single-digit multiplications is 3k, which is nc where c = log23.

Since one can extend any inputs with zero digits until their length is a power of two, it follows that the number of elementary multiplications, for any n, is at most . Since the additions, subtractions, and digit shifts (multiplications by powers of B) in Karatsuba's basic step take time proportional to n, their cost becomes negligible as n increases. More precisely, if t(n) denotes the total number of elementary operations that the algorithm performs when multiplying two n-digit numbers, then

for some constants c and d. For this [recurrence relation](https://en.wikipedia.org/wiki/Recurrence_relation), the [master theorem for divide-and-conquer recurrences](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)) gives the asymptotic bound

It follows that, for sufficiently large n, Karatsuba's algorithm will perform fewer shifts and single-digit additions than longhand multiplication, even though its basic step uses more additions and shifts than the straightforward formula. For small values of n, however, the extra shift and add operations may make it run slower than the longhand method. The point of positive return depends on the [computer platform](https://en.wikipedia.org/wiki/Computer_platform) and context. As a rule of thumb, Karatsuba's method is usually faster when the multiplicands are longer than 320–640 bits.

**CONCLUSION**

Unlike the brute force algorithm which becomes increasingly inefficient with higher and larger numbers having a time complexity of n2, the Karatsuba algorithm, which divides the problem into three subdivisions, is more effective as the complexity decrease significantly to n1.59. The Karatsuba algorithm has several applications in fluid mechanics, solid geometry, differential applications of a finite system and complex analysis among others. The algorithm is used as the cornerstone of discrete Fourier Transforms (FFT).