CSCI B503
FALL 2013
DATE: 12/04/2013
PROJECT: FINAL TURNIN
Prachi Shah
pracshah@indiana.edu

PROJECT REPORT

PROJECT DESCRIPTION:

Karatsuba Algorithm:

- The project focuses on implementation of Karatsuba Algorithm [1] which is a fast algorithm for multiplying large n-bit integers.
- The existing traditional algorithm 1.8 [2] multiplies two n-bit integers in time $O(n^2)$. The time taken here for multiplication is more and hence there was a need to perform multiplication in a faster way. Karatsuba algorithm helps perform faster multiplication.
- The Karatsuba algorithm is a recursive algorithm.
- The algorithm uses divide and conquer technique.
- This algorithm performs multiplication of two numbers word by word.
- The algorithm is: [1]

Algorithm 5.2 Faster Multiplication: Input: The $2n$-bit number $U = U_1 2^n + U_2$ where $0 \le U_1 < 2^n$ and $0 \le U_2 < 2^n$, and the $2n$-bit number $V = V_1 2^n + V_2$ where $0 \le V_1 < 2^n$ and $0 \le V_2 < 2^n$. Output: The product UV represented by $UV = W_1 2^{3n} + W_2 2^{2n} + W_3 2^n + W_4$.

Step **1.** Set $T_1 \leftarrow U_1 + U_2$.

Step **2.** Set $T_2 \leftarrow V_1 + V_2$.

Step **3.** Set $W_3 \leftarrow T_1 T_2$.

Step **4.** Set $W_2 \leftarrow U_1 V_1$.

Step **5.** Set $W_4 \leftarrow U_2 V_2$.

Step **6.** Set $W_3 \leftarrow W_3 - W_2 - W_4$.

Step **7.** Set $C \leftarrow \lfloor W_4 / 2^n \rfloor$ and $W_4 \leftarrow W_4 \bmod 2^n$.

Step **8.** Set $W_3 \leftarrow W_3 + C$, $C \leftarrow \lfloor W_3 / 2^n \rfloor$ and $W_3 \leftarrow W_3 \bmod 2^n$.

Step **9.** Set $W_2 \leftarrow W_2 + C$, $W_1 \leftarrow \lfloor W_2 / 2^n \rfloor$ and $W_2 \leftarrow W_2 \bmod 2^n$.

Analyzing the Karatsuba Algorithm [1]:

- This turn in limits implementation of the algorithm [1] to multiply two 32-bit integers.
- Example: Consider two numbers: Num1 = 6789, Num2 = 2345

- If the two numbers are of unequal length, for example, Num1 = 6789 and Num2 = 234; the addition (algorithm 1.4) [3] and subtraction logic in the code stores the number with maximum size (Num1 = 6789) as the first number for multiplication. Thus the multiplication will be 6789 * 234 and not 234*6789.
- Now, we split the two numbers into halves.
  So, the upper half bits of Num1 is up1 = 67, the lower half bits of Num1 is lw1 = 89.
  The upper half bits of Num2 is up2 = 23, the lower half bits of Num2 is lw2 = 45.
  The two numbers are written as:
  Num1 = up1*$2^{n/2}$ + lw1 where 0< up1< $2^{n/2}$ and 0< lw1< $2^{n/2}$
  Num2 = up2*$2^{n/2}$ + lw2 where 0< up2< $2^{n/2}$ and 0< lw2< $2^{n/2}$
- Now, Perform addition:
  Do, up1+lw1; up2+lw2
- Perform multiplication:

up1*up2; lw1*lw2

- We now form the equation for faster multiplication:

Num1 * Num2 = [(up1*up2)* 2^n] + [[(up1+lw1) (up2+lw2) - (up1*up2) – (lw1*lw2)] * $2^{n/2}$] + lw1*lw2

Here, the least significant bits of the result are calculated with the help of lw1*lw2. The most significant bits of the result are calculated with the help of up1*up2.

- The different steps of Algorithm 5.2 [1] are implemented in the code submitted.

FILES INCLUDED:

scaffold32.c; FinalFirstRun.txt; FinalTestResults.txt

LANGUAGE:

Programming Language:

C programming language

Compilation:

bash> make clean

The compiler will remove any existing .o object files.

bash> make

The compiler will create .o object files for the .c files.

Running the code:
bash> ./main32
The code runs for 32-bit multiplication using Algorithm 5.2 [1]. Two randomly generated numbers are multiplied and the result is stored in the 'finalAnswer'. The carry value is stored in the 'toCarry' variable.
bash> ./test_32.sh
Test cases are executed for 32-bit multiplication.

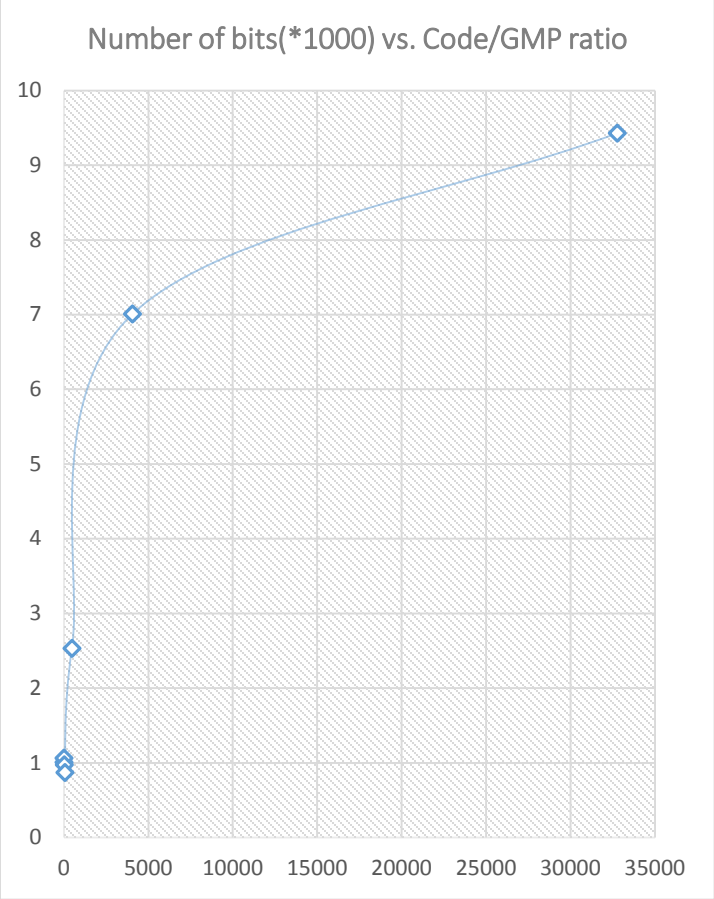
BASE:
Base used for the calculations is 2 ^ (32) which is 'wordLen= 4294967296llu" as mentioned in the code file.

TIME COMPLEXITY:
The time required to perform this algorithm[1] is $\mathbf{T_n = 3T_{(n/2+1)} + O(n/2)}$
The time complexity reduces because the algorithm runs recursively.
The time complexity therefore is[1]:
$\mathbf{T_n = O(n^{\log_2 3}) \approx O(n^{1.59})}$

OUTPUT:
The table shows values for Algorithm 5.2 [1] implemented for test cases in test_32.sh:

Number of bits	Ratio (Product time of code/GMP)
1,1	1.061219
0,0	1.102989
0,1	1.074058
1,0	1.087798
32, 32	0.971255
64, 64	0.870355
480, 480	2.533659
4064, 4064	7.008038
32736, 32736	9.427792
32, 64	0.878412
32, 480	1.068490
32, 4064	1.609399
32, 32736	3.929992
480, 4064	5.050129
480, 32736	6.581963
4064, 32736	11.192275

GRAPH
The graphs plots Number of bits vs. Ratio (Product time of code/GMP) for test cases in test_32.sh:



REFERENCES:

- [1] The Analysis of Algorithms Paul Walton Purdom, Jr; Cynthia A. Brown: Algorithm 5.2
- [2] The Analysis of Algorithms Paul Walton Purdom, Jr; Cynthia A. Brown: Algorithm 1.8
- [3] The Analysis of Algorithms Paul Walton Purdom, Jr; Cynthia A. Brown: Algorithm 1.4
- General discussion on algorithm understanding with Chinmay Deshpande.
- 3 Karatsuba Multiplication 13 min: https://www.youtube.com/watch?v=OtzDFLnIREc