DeepakKonidena (bkoniden), 04-24-2010

Algorithm 1.8

Overview

Algorithm1.8 presented in the textbook assumes the sizes of input operands are equal. This implementation modifies the original implementation by considering unequal sized inputs.

Analysis

The algorithm is pretty simple to analyze. Steps1-5 are executed wa times. Steps 6-10 are executed wb*wa times and step 11 is executed wa times. If we consider the case wa = wb, the runtime is dominated by the steps in the inner loop and hence the algorithm takes $O(n^2)$ time to perform multiplication.

Implementation

```
1:    for(i = 0; i < wa ; i++ )
2:    *c[i] = 0;
3:
4:    for(i = 0; i< wa; i++){
5:          carry = 0;
6:          for(j = 0; j < wb; j++){
7:                p = (int64)(*a[i])*(*b[j]) + *c[i + j] + carry;
8:                *c[i+j] = (int32)(p & (0xffffffff));
9:                carry = (int32)((p>>32) & (0xffffffff));
10:         }
11:         *c[i+wb] = carry;
12:   }
```

Optimization

In steps 8 and 9, instead of performing a modulus and division over 2^{32} (which is the base for 32-bit version), this implementation performs bit operations that are lot faster than performing division and modulus.
Also, if the outerloop is iterated over the smaller number and the inner loop by the larger number, there may be some optimization achieved.

Results and Statistics

The results here compare the 16-bit version with the 32-bit version. And as we can see the 32-bit version performs faster than the 16-bit version since larger bases need lesser words and hence lesser products

Number of bits	Product1.8 Basic 16bit	Product1.8 Basic 32bit
100	3.73	2.82
500	10.4	5.57
1000	10.55	5.16
2000	6.15	3.12
5000	8.08	4.38
10000	10.06	5.37
20000	12.67	15.14
100000	46.5	17.1

Table: Code/GMP values of Algorithm 1.8 basic 16 and 32bit versions averaged over a 1000 multiplications.

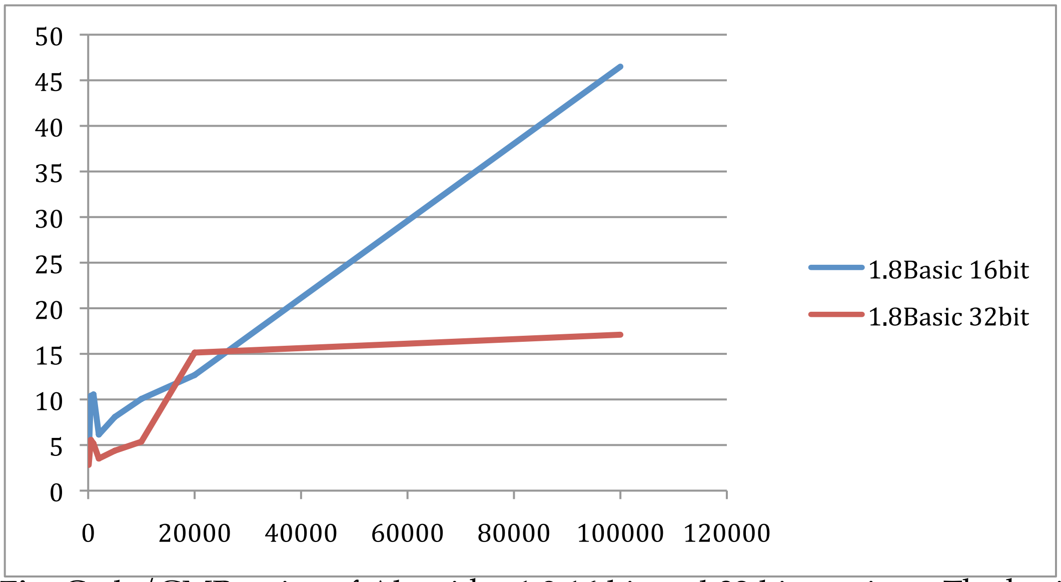

Fig: Code/GMP ratios of Algorithm1.8 16 bit and 32 bit versions. The horizontal axis determines the number of bits and the vertical axis determines the Code/GMP ratio values. All the values are averaged over 1000 multiplications.

Number of bits	Product1.8 Basic 32bit	GMP
100	0.0015	0.0005
500	0.0065	0.0011
1000	0.0014	0.0028
2000	0.0385	0.0109
5000	0.1771	0.0403
10000	0.6119	0.1139
20000	4.8627	0.3213
100000	98.4	5.73

Table: Actual runtime values(in sec) of Algorithm 1.8 32bit version and GMP averaged over a 1000 multiplications.

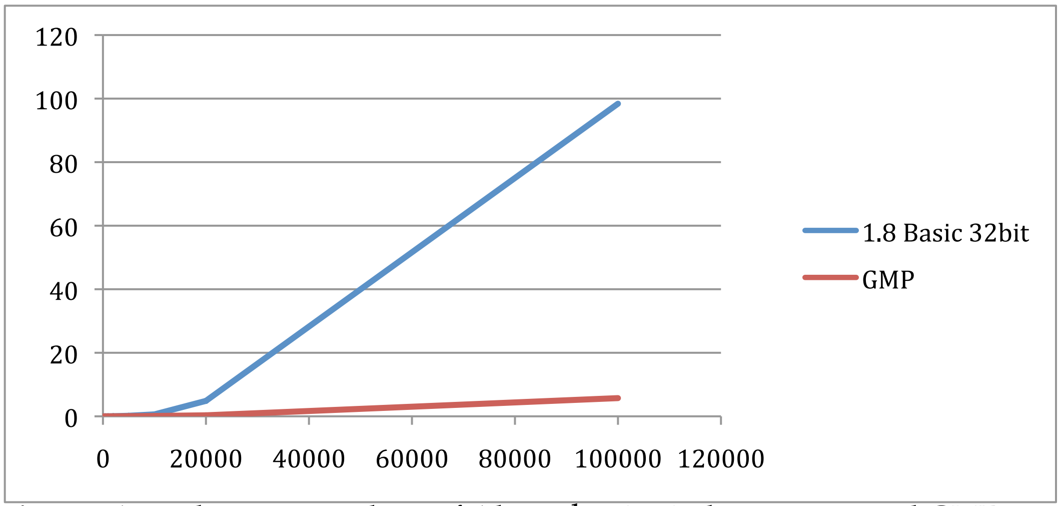

Fig: Actual runtime values of Algorithm1.8 32 bit version and GMP in secs.

The horizontal axis determines the number of bits and the vertical axis determines the runtime values in seconds. All the values are averaged over 1000 multiplications.

Karatsuba Multiplication

Overview

Karatsuba algorithm presented in the textbook makes certain assumptions on the wordsizes of the two operands. They have to be even (2n bit) and since the algorithm is recursive, the operands to the recursive calls need to be even too and that imposes a restriction that the original input operands should not only be even but also a power of 2 (2^k). This implementation is greatly modified from the original in that the wordsizes need not be equal. Also, this implementation calls Algorithm 1.8 instead of the recursive calls for the intermediate products.

Analysis

Lets consider the analysis for equal sized numbers here for the sake of simplicity. The algorithm makes 3 calls to Product1_8_Basic(), 4 calls to Addition1_4() and 2 calls to Difference(). In step1, since U1 and U2 are equal sized half words, and since Addition1_4() grows linearly with n, step1 takes O(n/2) and same goes with step 2. Step3 is the product of (U1+U2) and (V1+V2) and since Product1_8_Basic grows quadratically with n, step3 typically takes $O((n/2)^2)$. Similarly steps 4 and 5 typically take $O((n/2)^2)$. Steps 6 and 7 calculate the differences (W3 – U1*V1) and (W3 – U1*V1- U2*V2) respectively and since Difference() grows linearly with n, they take O(n) (since W3 is of size n). Steps 8 and 9 calculate the offset and add them to the respective quartet (W3 and W2 respectively). The top 2^n digits of W4 are added to W3 and the top 2^n digits of the sum of W3 and the offset are added to W2. Step8 is dominated by the size of W3, which is n and hence it grows linearly with n, takes O(n) time. Step9 is dominated by the size of W2, which is n/2 and hence takes O(n/2).

The overall runtime is given by

$$2*O(n/2) + 3*O((n/2)^2) + 2*O(n) + O(n) + O(n/2)$$
$$=> 3*[O((n/2)^2) + O(n) + O(n/2)]$$

Implementation

```
1:    Addition1_4(u1, u2, U1SIZE, U2SIZE, &res[2*RESSIZE]);
2:    Addition1_4(v1, v2, V1SIZE, V2SIZE, &res[3*RESSIZE]);
3:    Product1_8_Basic(&res[2*RESSIZE],&res[3*RESSIZE],&res[4*RESSIZE],T1SIZE,T
      2SIZE);

4:    Product1_8_Basic(&u1[0], &v1[0], &res[0],U1SIZE, V1SIZE );
5:    Product1_8_Basic(&u2[0], &v2[0], &res[RESSIZE],U2SIZE, V2SIZE );

6:    Difference(&res[4*RESSIZE],&res[0],W3SIZE,(U1SIZE+V1SIZE),
      &res[6*RESSIZE]);

7:    Difference(&res[6*RESSIZE],&res[RESSIZE],W3SIZE,(U2SIZE+V2SIZE),
      &res[7*RESSIZE]);

8:    Addition1_4(&res[7*RESSIZE],&res[RESSIZE+W4SIZE],W3SIZE,C1SIZE,
      &res[8*RESSIZE]);

9:    Addition1_4(&res[0],&res[8*RESSIZE+dig],(U1SIZE + V1SIZE),  C2SIZE,
      &res[9*RESSIZE]);
```

Optimization
When the input operands are unequal and lets say U is the larger number and V is the smaller number, this implementation divides U and V in the following fashion.

U = U1 (size = (u – v2))+ U2 (size = v2)
V = 0 + V2.

The larger number is divided according to the wordsize of the smaller number and this achieves some optimization. As V1 = 0, the product U1*V1(step4) , the sum (V1+V2) (step2) and the difference(step6) are omitted and runtime becomes
$3*[O((n/2)^2) + O(n) + O(n/2)] – [O((n/2)^2) + O(n) + O(n/2)]$
$=> 2*[O((n/2)^2) + O(n) + O(n/2)]$

Also, while performing the modulus and division operations, bit operations are considered instead of '%' or '/' operations and this improved the performance.

Results and Statistics

Number of bits	Karatsuba 16bit	Karatsuba 32bit
100	30.972	14.2
500	73	30.6
1000	54.6	26.3
2000	5.98	4.86
5000	7.01	4.95
10000	5.52	18.6
20000	25.64	18.
100000	46.5	17.1

Table: Code/GMP values of Karatsuba 16 and 32bit versions averaged over a 1000 multiplications.

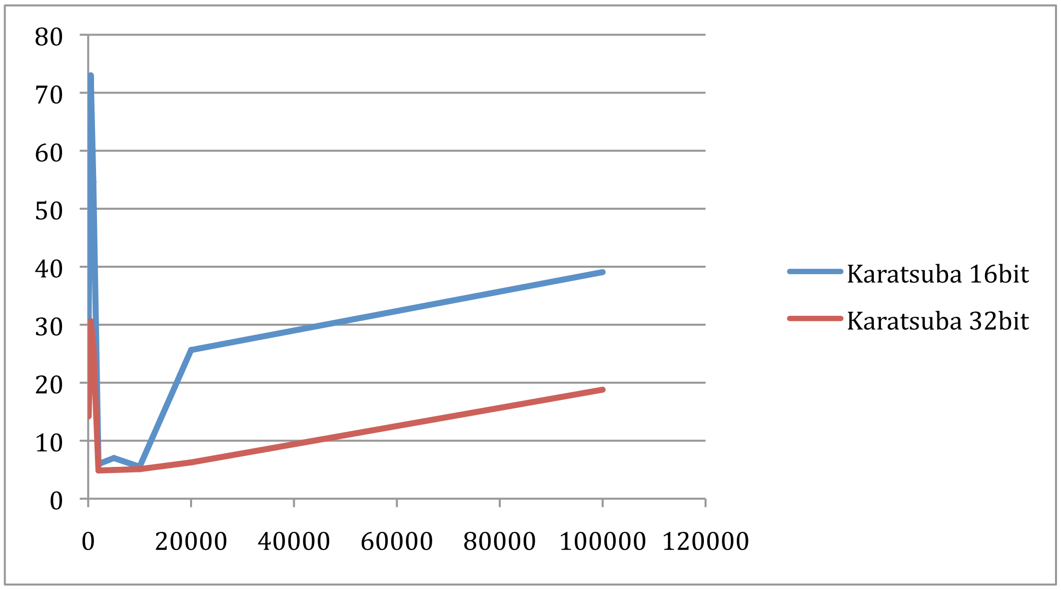

Fig: Code/GMP ratios of Karatsuba 16 bit and 32 bit versions. The horizontal axis determines the number of bits and the vertical axis determines the Code/GMP ratio values. All the values are averaged over 1000 multiplications.

Number of bits	Karatsuba 32bit	GMP
100	0.0079	0.0005
500	0.0365	0.0011
1000	0.0738	0.0028
2000	0.1415	0.0291
5000	0.4164	0.0848
10000	1.0283	0.2036
20000	7.0867	1.1334
100000	162.473	8.697

Table: Actual runtime values(in sec) of Karatsuba 32bit version and GMP averaged over a 1000 multiplications.

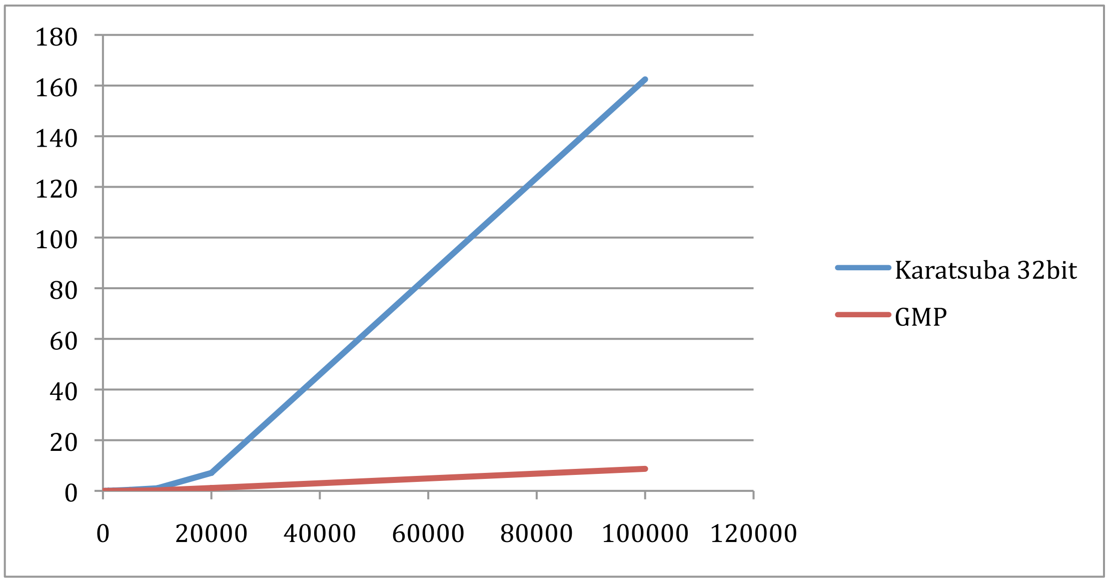

Fig: Actual runtime values of Karatsuba 32 bit version and GMP in secs. The horizontal axis determines the number of bits and the vertical axis determines the runtime values in seconds. All the values are averaged over 1000 multiplications.

The graph below shows a comparative study of all the algorithms implemented as part of this project. Clearly, the 32-bit versions are faster than their 16-bit counterparts . Whereas, with respect to the comparison between Product1.8 Basic and Karatsuba, Product 1.8 Basic seems to be faster than Karatsuba for very small numbers and slower for numbers upto 100000 bits.

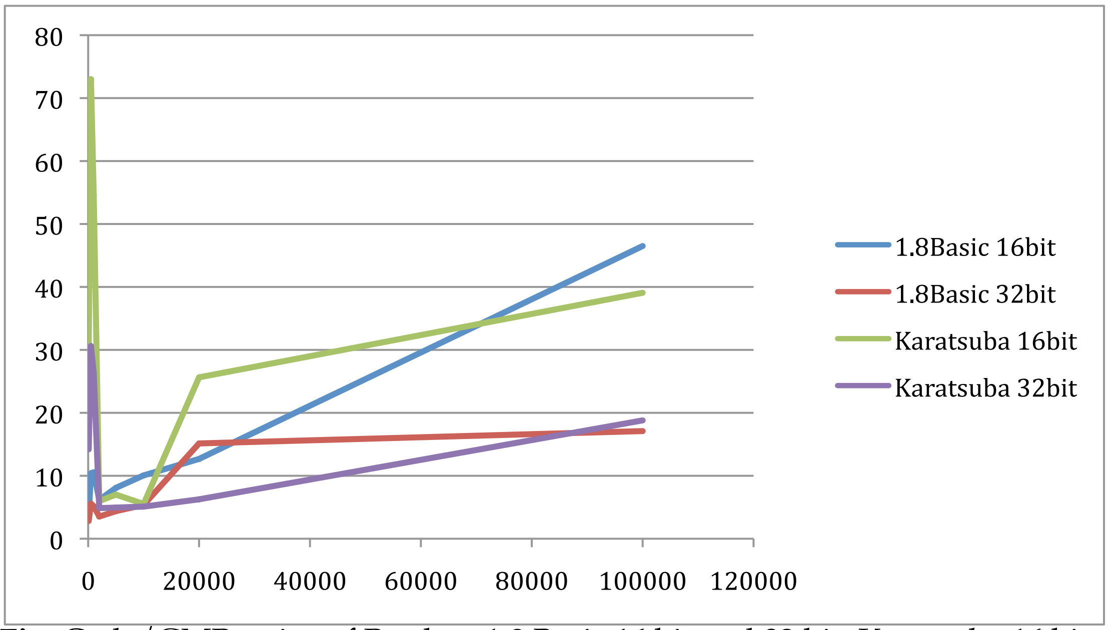

Fig: Code/GMP ratios of Product 1.8 Basic 16 bit and 32 bit, Karatsuba 16 bit and 32 bit versions. The horizontal axis determines the number of bits and the vertical axis determines the Code/GMP ratio values. All the values are averaged over 1000 multiplications.

Other routines used as part of the project

Addition1_4()

Addition1_4() differs from the one presented in the text in that it adds two numbers with unequal word sizes.

Implementation

```
1:     for(i = 0; i < wy; i++){
2:           s = (int64)(*x[i]) + *y[i] + carry;
3:           *sum[i] = (int32)(s & (0xffffffff));
4:           carry = (int32)((s>>32) & (0xffffffff));
5:     }
6:
7:     for(i = wy; i < wx; i++){
8:           s = (int64)(*x[i]) + carry;
9:           *sum[i] = (int32)(s & (0xffffffff));
10:          carry = (int32)((s>>32) & (0xffffffff));
11:    }
```

Analysis

Addition1_4() is easy to analyse. If the input operands are equal and are of size n, steps 1-4 are each executed n times. And the algorithm grows linearly with n. Hence Addition1_4() takes O(n) time.

Difference()

Difference() differs from Addition1_4() slightly in that it subtracts the borrow from the next word.

Implementation

```
1:      for(i =0; i < wy; i++){
2:              s = (int64)(*x[i]) - *y[i] - borrow;
3:              *diff[i] = (int32)(s & (0xffffffff));
4:              //printf("\n%x - %x - %x = %x ", *x[i], *y[i], borrow, *diff[i]);
5:              borrow = (int32)((s>>32) & (0xffffffff));
6:              if(borrow != 0)
7:                      borrow = 1;
8:      }
9:
10:     for(i = wy; i < wx; i++){
11:             s = (int64)(*x[i]) - borrow;
12:             *diff[i] = (int32)(s & (0xffffffff));
13:             borrow = (int32)((s>>32) & (0xffffffff));
14:             if(borrow != 0)
15:                     borrow = 1;
        }
```

Analysis

Difference() is very similar to addition. If the input operands are equal and are of size n, steps 1-4 are each executed n times. And the algorithm grows linearly with n. Hence Difference() takes O(n) time too.

Max() & Min()

Max() and Min() find the maximum and minimum among two given numbers. They are constant time operations.

toBinary() & toBinary64()

toBinary() & toBinary64() are methods used for debugging purposes. They print the passed unsigned integers into their binary representations reversed. Even though %x specifier in printf() prints the unsigned integers in hex notation, I found it quite useful to print them in binary especially when dealing with carries and borrows. These methods are commented during the actual run.

memcpy(), calloc() and free()

The project used memcpy(), calloc() and free() while implementing the karatsuba algorithm. The initial operands are copied to two dimensional pointers using memcpy(). While calloc() and free() are used to allocate and deallocate space for the pointers. I ignored the time overhead caused by these calls. Their effect on the runtime performance is beyond the scope of this project.

References

1) Analysis of Algorithms, Paul W Purdom
2) http://en.wikipedia.org/wiki/Karatsuba_algorithm
3) http://www.cs.cmu.edu/~cburch/251/karat/
4) http://www.programmersheaven.com/mb/CandCPP/354368/354368/karatsuba-algorithm/
5) http://www.cplusplus.com/
6) Discussed at various times with Georgi Chunev, Bing Jing and Joshua Bonner

Appendix

```
/*
 * scaffold32.c
 *
 *  Created on: Mar 28, 2010
 *      Author: deepakkoni
 */

#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>
#include <time.h>
#include "scaffold32.h"
#include <math.h>
#include <string.h>
#include <limits.h>


int32 Max(int32 a, int32 b){

	if(a > b)
		return a;
	else if(a < b)
		return b;
	else
		return a;
}

void clear(int32** x, int32 size){
	int32 i = 0;
	for(i = 0; i < size; i++)
		*x[i] = 0;
}

int32 Min(int32 a, int32 b){

	if(a < b)
		return a;
	else if(b < a)
		return b;
	else
		return a;
}

char* toBinary(int32 i, char *str){

	unsigned int num = i;
	strcpy(str, "");

	while(num){
		if( 0x0001 & num){
			strcat(str,"1");
			num = num >> 1;
		}
		else{
			strcat(str,"0");
			num = num >> 1;
		}
	}
	return str;
}

char* toBinary64(int64 i, char *str){

	unsigned long long int num = i;
	strcpy(str, "");

	while(num){
		if( 0x0001 & num){
			strcat(str,"1");
			num = num >> 1;
		}
```

```
             else{
                   strcat(str,"0");
                   num = num >> 1;
             }
       }
       return str;
}

int32** Product1_8_Basic(int32** a, int32** b, int32** c, int32 wa, int32 wb){

             int32 i, j, carry, flag= 0;
             int64 p;


             if(wa > wb)
                   flag = 1;
             else if(wa < wb)
                   flag = 2;
             else if (wa == wb)
                   flag = 0;


          if(flag == 0){

             //zeroing out the remaining digits
             for(i = 0; i <      wa ; i++ )
                   *c[i] = 0;

             //Actual 1.8 implementation
             for(i = 0; i< wa; i++){
                   carry = 0;
                   for(j = 0; j < wa; j++){

                         p = (int64)(*a[i])*(*b[j]) + *c[i + j] + carry;
                         *c[i+j] = (int32)(p & (0xffffffff));
                         carry = (int32)((p>>32) & (0xffffffff));
                   }
                   *c[i+wa] = carry;
             }
          }

          if(flag == 1){
              //zeroing out the remaining digits
              for(i = 0; i < wa ; i++ )
                 *c[i] = 0;

              //Actual 1.8 implementation
              for(i = 0; i< wa; i++){
                   carry = 0;
                 for(j = 0; j < wb; j++){
                   p = (int64)(*a[i])*(*b[j]) + *c[i + j] + carry;
                   *c[i+j] = (int32)(p & (0xffffffff));
                   carry = (int32)((p>>32) & (0xffffffff));
                 }
                 *c[i+wb] = carry;
              }
         }

          if(flag == 2){
              //zeroing out the remaining digits
              for(i = 0; i < wb ; i++ )
                 *c[i] = 0;
              //Actual 1.8 implementation
              for(i = 0; i< wb; i++){
                   carry = 0;
                 for(j = 0; j < wa; j++){
                   p = (int64)(*b[i])*(*a[j]) + *c[i + j] + carry;
                   *c[i+j] = (int32)(p & (0xffffffff));
                   carry = (int32)((p>>32) & (0xffffffff));
                 }
                 *c[i+wa] = carry;
              }
         }
```

```
		return &c[0];
}

void Karatsuba(int32** a, int32** b, int32** res, int32 dig, int32 wa, int32 
wb){

	//a is always either larger or equal to b
	// u1, u2 point to larger number and v1,v2 point to smaller number
	int32 i;
	int32 **u1, **v2, **u2;
	int32 U1SIZE = (wa - dig), U2SIZE = dig, T1SIZE = Max((wa - dig), dig), 
T2SIZE = Max((wb - dig), dig);
	int32 V1SIZE = (wb - dig), V2SIZE = dig, RESSIZE = (wa + wb);
	int32 W3SIZE = (T1SIZE + T2SIZE), W4SIZE = dig;
	int32 C1SIZE = (U2SIZE + V2SIZE - dig), C2SIZE = (W3SIZE - dig);


	if(dig == 1 || dig == 0)
		Product1_8_Basic(a,b,&res[10*RESSIZE],wa,wb);


	else{
		if(wa == wb){

			int32  **v1;

			if((u1 = (int32 **)calloc(U1SIZE,sizeof(int32*)))== NULL)
				fprintf(stderr,  "\n  Memory  cannot  be  allocated  to 
u1\n");
			if((u2 = (int32 **)calloc(U2SIZE,sizeof(int32*)))== NULL)
				fprintf(stderr,  "\n  Memory  cannot  be  allocated  to 
u2\n");
			if((v2 = (int32 **)calloc(V2SIZE,sizeof(int32*)))== NULL)
				fprintf(stderr,  "\n  Memory  cannot  be  allocated  to 
v1\n");
			if((v1 = (int32 **)calloc(V1SIZE,sizeof(int32*)))== NULL)
				fprintf(stderr,  "\n  Memory  cannot  be  allocated  to 
v2\n");

			  //Allocating space for individual pointers
			for(i = 0; i < U1SIZE; i++)
			   if((u1[i] = (int32 *)calloc(1, sizeof(int32))) == NULL)
				  fprintf(stderr,"\nMemory cannot be allocated\n");
			for(i = 0; i < U2SIZE; i++)
			   if((u2[i] = (int32 *)calloc(1, sizeof(int32))) == NULL)
				  fprintf(stderr,"\nMemory cannot be allocated\n");
			for(i = 0; i < V2SIZE; i++)
			   if((v2[i] = (int32 *)calloc(1, sizeof(int32))) == NULL)
				  fprintf(stderr,"\nMemory cannot be allocated\n");
			for(i = 0; i < V1SIZE; i++)
			   if((v1[i] = (int32 *)calloc(1, sizeof(int32))) == NULL)
				  fprintf(stderr,"\nMemory cannot be allocated\n");

			//populating u1, u2, v1, v2
		    //copying values
		    for(i = 0; i < U2SIZE; i++)
		        memcpy(u2[i], a[i], sizeof(int32));
		    for(i = 0; i < U1SIZE; i++)
		       memcpy(u1[i], a[i + U2SIZE], sizeof(int32));
		    for(i = 0; i < V2SIZE; i++)
		        memcpy(v2[i], b[i], sizeof(int32));
		    for(i = 0; i < V1SIZE; i++)
		        memcpy(v1[i], b[i + V2SIZE], sizeof(int32));

		/*    //printing u1, u2, v1, v2
		    for(i = 0; i < U1SIZE; i++)
		       printf("\n u1[%u] = %x" ,i, *u1[i]);
		    for(i = 0; i < U2SIZE; i++)
		       printf("\n u2[%u] = %x" ,i, *u2[i]);
		    for(i = 0; i < V1SIZE; i++)
		       printf("\n v1[%u] = %x" ,i, *v1[i]);
		    for(i = 0; i < V2SIZE; i++)
		       printf("\n v2[%u] = %x" ,i, *v2[i]);
```

```
              */
                 //Calculating the sum (U1 + U2)
                    Addition1_4(u1, u2, U1SIZE, U2SIZE, &res[2*RESSIZE]);
                    Addition1_4(v1, v2, V1SIZE, V2SIZE, &res[3*RESSIZE]);

             /*     //Printing (U1+U2) and (V1+V2)
                    for(i = 0; i < T1SIZE; i++)
                          printf("\n (u1+u2) = %x", *res[2*RESSIZE + i]);
                    for(i = 0; i < T2SIZE; i++)
                          printf("\n (v1+v2) = %x", *res[3*RESSIZE + i]);
              */
                    //product of T1,T2 & U1,V1 and U2,V2
                    Product1_8_Basic(&res[2*RESSIZE],          &res[3*RESSIZE],
&res[4*RESSIZE],T1SIZE, T2SIZE );
                    Product1_8_Basic(&u1[0], &v1[0], &res[0],U1SIZE, V1SIZE );
                    Product1_8_Basic(&u2[0],    &v2[0],    &res[RESSIZE],U2SIZE,
V2SIZE );

              /*    for(i = 0; i < W3SIZE; i++)
                          printf("\n (u1 + u2)*(v1 + v2) = %x" , *res[4*RESSIZE
+ i]);
                    for(i = 0; i < (U1SIZE + V1SIZE); i++)
                       printf("\n (u1)*(v1) = %x" , *res[i]);
                    for(i = 0; i < (U2SIZE + V2SIZE); i++)
                       printf("\n (u2)*(v2) = %x" , *res[RESSIZE + i]);
               */
                    //Calculating  the  difference  of  W3=(T1*T2),  W2  =  (U1*V1)
and W4 = (U2*V2)
                    Difference(&res[4*RESSIZE],  &res[0],  W3SIZE,  (U1SIZE  +
V1SIZE),&res[6*RESSIZE]);
                    Difference(&res[6*RESSIZE],  &res[RESSIZE],  W3SIZE,  (U2SIZE
+ V2SIZE),&res[7*RESSIZE]);
              /*
                    for(i = 0; i < W3SIZE; i++)
                          printf("\n   (u1+   u2)*(v1   +   v2)   -   u1*v1   =   %x"
,*res[6*RESSIZE + i]);
                    for(i = 0; i < W3SIZE; i++)
                          printf("\n (u1+ u2)*(v1 + v2) - u1*v1 - u2*v2= %x" ,
*res[7*RESSIZE + i]);
             */
                    // ADD W3 + C1, W2 + C2
                    Addition1_4(&res[7*RESSIZE],   &res[RESSIZE   +   W4SIZE],
W3SIZE, C1SIZE, &res[8*RESSIZE]);
                    if((U1SIZE + V1SIZE) > C2SIZE)
                          Addition1_4(&res[0], &res[8*RESSIZE + dig], (U1SIZE +
V1SIZE), C2SIZE, &res[9*RESSIZE]);
                    else
                          Addition1_4(&res[8*RESSIZE  +  dig],  &res[0],  C2SIZE,
(U1SIZE + V1SIZE), &res[9*RESSIZE]);

                    for(i = 0; i < Max(C2SIZE,(U1SIZE + V1SIZE)); i++){
                    //    printf("\n W1 and W2 = %x", *res[9*RESSIZE + i]);
                          memcpy(res[10*RESSIZE  +  2*dig  +  i],  res[9*RESSIZE  +
i], sizeof(int32));
                    }
                    for(i = 0; i < dig; i++){
                    //    printf("\n W3 = %x", *res[8*RESSIZE + i]);
                          memcpy(res[10*RESSIZE + dig + i], res[8*RESSIZE + i],
sizeof(int32));
                    }
                    for(i = 0; i < W4SIZE; i++){
                    //    printf("\n W4 = %x", *res[RESSIZE + i]);
                          memcpy(res[10*RESSIZE   +i],   res[RESSIZE   +   i],
sizeof(int32));
                    }

                    // freeing the pointers
                 for(i = 0; i < V1SIZE; i++)
                     free(v1[i]);

                 free(v1);
             }
             else{


                    if((u1 = (int32 **)calloc(U1SIZE,sizeof(int32*)))== NULL)
                          fprintf(stderr,  "\n  Memory  cannot  be  allocated  to
u1\n");
                    if((u2 = (int32 **)calloc(U2SIZE,sizeof(int32*)))== NULL)
                          fprintf(stderr,  "\n  Memory  cannot  be  allocated  to
u2\n");
                    if((v2 = (int32 **)calloc(V2SIZE,sizeof(int32*)))== NULL)
                          fprintf(stderr,  "\n  Memory  cannot  be  allocated  to
v1\n");


                      //Allocating space for individual pointers
                    for(i = 0; i < U1SIZE; i++)
                       if((u1[i] = (int32 *)calloc(1, sizeof(int32))) == NULL)
                             fprintf(stderr,"\nMemory cannot be allocated\n");
                    for(i = 0; i < U2SIZE; i++)
                       if((u2[i] = (int32 *)calloc(1, sizeof(int32))) == NULL)
                             fprintf(stderr,"\nMemory cannot be allocated\n");
                    for(i = 0; i < V2SIZE; i++)
                       if((v2[i] = (int32 *)calloc(1, sizeof(int32))) == NULL)
                             fprintf(stderr,"\nMemory cannot be allocated\n");


                    //populating u1, u2, v1, v2
                  //copying values
                  for(i = 0; i < U2SIZE; i++)
                      memcpy(u2[i], a[i], sizeof(int32));
                  for(i = 0; i < U1SIZE; i++)
                    memcpy(u1[i], a[i + U2SIZE], sizeof(int32));
                  for(i = 0; i < V2SIZE; i++)
                      memcpy(v2[i], b[i], sizeof(int32));


             /*    //printing u1, u2, v1, v2
                  for(i = 0; i < U1SIZE; i++)
                    printf("\n u1[%u] = %x" ,i, *u1[i]);
                  for(i = 0; i < U2SIZE; i++)
                    printf("\n u2[%u] = %x" ,i, *u2[i]);
                  for(i = 0; i < V2SIZE; i++)
                    printf("\n v2[%u] = %x" ,i, *v2[i]);
              */
                  //Calculating the sum (U1 + U2)
                  if(U1SIZE > U2SIZE)
                    Addition1_4(u1, u2, U1SIZE, U2SIZE, &res[2*RESSIZE]);
                  else
                    Addition1_4(u2, u1, U2SIZE, U1SIZE, &res[2*RESSIZE]);

                    for(i = 0; i < T2SIZE; i++)
                          memcpy(res[3*RESSIZE + i], v2[i], sizeof(int32));

               /*   //Printing (U1+U2) and (V1+V2)
                    for(i = 0; i < T1SIZE; i++)
                          printf("\n (u1+u2) = %x", *res[2*RESSIZE + i]);
                    for(i = 0; i < T2SIZE; i++)
                          printf("\n (v1+v2) = %x", *res[3*RESSIZE + i]);
                */
                    //product of T1,T2 & U1,V1 and U2,V2
                    Product1_8_Basic(&res[2*RESSIZE],           &res[3*RESSIZE],
&res[4*RESSIZE], T1SIZE, T2SIZE );
                    Product1_8_Basic(&u2[0],   &v2[0],   &res[RESSIZE],   U2SIZE,
V2SIZE );

               /*   for(i = 0; i < W3SIZE; i++)
                          printf("\n (u1 + u2)*(v1 + v2) = %x" , *res[4*RESSIZE
+ i]);
                    for(i = 0; i < (U2SIZE + V2SIZE); i++)
                          printf("\n (u2)*(v2) = %x" , *res[RESSIZE + i]);
                */
```

```
                        //Calculating the difference of W3=(T1*T2), W2 = (U1*V1)
and W4 = (U2*V2)
                                Difference(&res[4*RESSIZE], &res[RESSIZE], W3SIZE,
(U2SIZE + V2SIZE),&res[5*RESSIZE]);

                //              for(i = 0; i < W3SIZE; i++)
                        //              printf("\n (u1+ u2)*(v1 + v2) - u1*v1 - u2*v2=
%x" , *res[5*RESSIZE + i]);

                                // ADD W3 + C1, W2 + C2
                                Addition1_4(&res[5*RESSIZE], &res[RESSIZE + W4SIZE],
W3SIZE, C1SIZE, &res[6*RESSIZE]);

                                for(i = 0; i < W3SIZE; i++){
                                //      printf("\n W1,W2 and W3 = %x", *res[6*RESSIZE
+ i]);
                                        memcpy(res[10*RESSIZE + i + dig],
res[6*RESSIZE + i], sizeof(int32));
                                }
                                for(i = 0; i < W4SIZE; i++){
                                //      printf("\n W4 = %x", *res[RESSIZE + i]);
                                        memcpy(res[10*RESSIZE + i], res[RESSIZE + i],
sizeof(int32));
                                }

                }



                // freeing the pointers
            for(i = 0; i < U1SIZE; i++)
                free(u1[i]);
            for(i = 0; i < U2SIZE; i++)
                free(u2[i]);
            for(i = 0; i < V2SIZE; i++)
                free(v2[i]);
            free(u1);
            free(u2);
            free(v2);
            //free(str);
        }
}


void Addition1_4(int32** x, int32** y, int32 wx, int32 wy, int32** sum){

        int32 i, carry = 0;
        int64  s = 0;

        for(i = 0; i < wy; i++){
                s = (int64)(*x[i]) + *y[i] + carry;
                *sum[i] = (int32)(s & (0xffffffff));
                carry = (int32)((s>>32) & (0xffffffff));
        }

        for(i = wy; i < wx; i++){
                s = (int64)(*x[i]) + carry;
                *sum[i] = (int32)(s & (0xffffffff));
                carry = (int32)((s>>32) & (0xffffffff));
        }
}

void Difference(int32** x, int32** y, int32 wx, int32 wy, int32** diff){

        int32 i, borrow = 0;
        int64 s = 0;


        for(i =0; i < wy; i++){
                s = (int64)(*x[i]) - *y[i] - borrow;
                *diff[i] = (int32)(s & (0xffffffff));
                //printf("\n%x - %x - %x = %x ", *x[i], *y[i], borrow, *diff[i]);
                borrow = (int32)((s>>32) & (0xffffffff));
                if(borrow != 0)
                        borrow = 1;
        }

        for(i = wy; i < wx; i++){
                s = (int64)(*x[i]) - borrow;
                *diff[i] = (int32)(s & (0xffffffff));
                borrow = (int32)((s>>32) & (0xffffffff));
                if(borrow != 0)
                        borrow = 1;
        }
}

//unsigned int gradeSchool(unsigned int  *x, unsigned int *y, unsigned int *z,
)
/* You are suppose to change the routine Product32 here to your own routine
 * The mpz calls in the scaffolded Product32 below are the normal GMP function
 *  calls  and  should  be  neglected.  By  casting  the  void  pointers  as  normal
unsigned
 * integers, you should be able to access the data values as normal 4 bytes
words.
 */


/* wa is word length of a, ba is bit length of a */
void Product32(void *a, void *b, void *c, unsigned int wa, unsigned int ba,
unsigned int wb, unsigned int bb, unsigned int *wc, unsigned int *bc){

        int32 SMALLER_WORDSIZE = 0;
        int32 i = 0;

     /* Cast a and b into short integers of size 32 bits */
     int32 *int_a = (int32 *) a;
     int32 *int_b = (int32 *) b;
     int32 *int_c = (int32 *) c;

     // Finding the smaller of wa and wb
     if(wa == wb)
        SMALLER_WORDSIZE = wa/2;
     else
        SMALLER_WORDSIZE = (wa < wb) ? wa : wb;

     *wc = wa + wb;

   //  printf("\n wa = %u, wb = %u \n", wa, wb);


     //creating space for and copying a and b, creating space for c
     int32** int32_a = (int32 **)calloc(wa, sizeof(int32 *));
     if(int32_a == NULL)
        fprintf(stderr,"\nMemory cannot be allocated\n");
     int32** int32_b = (int32 **)calloc(wb, sizeof(int32 *));
     if(int32_b == NULL)
         fprintf(stderr,"\nMemory cannot be allocated\n");
     int32** int32_c = (int32 **)calloc(*wc, sizeof(int32 *));
     if(int32_c == NULL)
         fprintf(stderr,"\nMemory cannot be allocated\n");


     //Allocating space for individual pointers
     for(i = 0; i < wa; i++)
        if((int32_a[i] = (int32 *)calloc(1, sizeof(int32))) == NULL)
              fprintf(stderr,"\nMemory cannot be allocated\n");
     for(i = 0; i < wb; i++)
        if((int32_b[i] = (int32 *)calloc(1, sizeof(int32))) == NULL)
              fprintf(stderr,"\nMemory cannot be allocated\n");
     for(i = 0; i < *wc; i++)
        if((int32_c[i] = (int32 *)calloc(1, sizeof(int32))) == NULL)
              fprintf(stderr,"\nMemory cannot be allocated\n");

    //Allocating space for recursive workspace
     int32** result = (int32 **)calloc(12*(wa + wb), sizeof(int32 *));
         if(result == NULL)
```

```
              fprintf(stderr,"\nMemory cannot be allocated\n");

       for(i = 0; i < 12*(wa + wb); i++)
          if((result[i] = (int32 *)calloc(1, sizeof(int32))) == NULL)
             fprintf(stderr,"\nMemory cannot be allocated\n");

       //copying values
       for(i = 0; i < wa; i++)
           memcpy(int32_a[i], &int_a[i], sizeof(int32));
       for(i = 0; i < wb; i++)
          memcpy(int32_b[i], &int_b[i], sizeof(int32));

     /*  printf("a = \n");
       //printing the values
       for(i = 0; i < wa; i++)
             printf(" %x" , *int32_a[i]);
       printf("b = \n");
          for(i = 0; i < wb; i++)
             printf(" %x" , *int32_b[i]);
   */
          if(wa > wb)
          //karatsuba
               Karatsuba(int32_a, int32_b, result , SMALLER_WORDSIZE, wa, wb);
          else if(wa < wb)
               Karatsuba(int32_b, int32_a, result , SMALLER_WORDSIZE, wb, wa);
          else
               Karatsuba(int32_a, int32_b, result , SMALLER_WORDSIZE, wa, wb);

          for(i = 0; i < *wc; i++)
               memcpy(&int_c[i],result[10*(wa+wb)+i], sizeof(int32));

   /*
       printf("\nThe value of 1<<32 is %s", toBinary64((int64)1<<32 , str));
       printf("\nThe size of unsigned short int is %u bytes", sizeof(int16));
       printf("\nThe size of unsigned int is %u bytes", sizeof(int32));
       printf("\nThe size of unsigned long int is %u bytes", sizeof(int32));
       printf("\nThe size of unsigned long long int is %u bytes", sizeof(int64));

          Product1_8_Basic(int32_a, int32_b, int32_c, wa, wb);

          for(i = 0; i < *wc; i++)
               memcpy(&int_c[i],int32_c[i], sizeof(int32));
   */


          /*  mpz_import(x, wa, ORDER, WORDBYTES, ENDIAN, NAILS, a);
               mpz_import(y, wb, ORDER, WORDBYTES, ENDIAN, NAILS, b);
             mpz_mul(z,x,y);
               c = mpz_export(c, wc, ORDER, WORDBYTES, ENDIAN, NAILS, z);
          */

          // freeing the pointers
             for(i = 0; i < wa; i++)
               free(int32_a[i]);
             for(i = 0; i < wb; i++)
               free(int32_b[i]);
             for(i = 0; i < *wc; i++)
               free(int32_c[i]);

             for(i = 0; i < 12*(wa + wb); i++)
               free(result[i]);

             free(int32_a);
             free(int32_b);
             free(int32_c);
             free(result);

   }
```