Department of Computer Science

Comparison of Traditional, Karatsuba and Fourier Big Integer Multiplication

Mohamed Arabi

University of Bath

BSc (Hons) Computer Science

May 2005

Comparison of Traditional, Karatsuba and Fourier Big Integer Multiplication

Submitted by Mohamed Arabi

COPYRIGHT

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Batchelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed: …….

This thesis may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed: …….

Abstract

There are several different algorithms to multiply together integers. The most famous and widely used is the one called the Traditional method which everybody learns at school: Multiply the first number with every digit of the second number and then add up all the properly shifted results. This is easy to perform, but get slow very quickly as the multiplicands get larger.

For systems that need to multiply large numbers in the range of several hundreds of digits, such as computer algebra systems, the traditional algorithm is too slow. These systems employ Karatsuba multiplication.

Additionally, there exist even faster algorithms, based on the fast Fourier transform, a 'divide and conquer' algorithm, as the Karatsuba algorithm. For medium sized of small numbers it is relatively slow. However, it is extremely fast compared to Traditional and Karatsuba algorithms.

This project will implement and compare these three algorithms, and determine the ranges of numbers for which each is the best.

Acknowledgments

My supervisor, Dr. Russel J Bradford
For providing me with a good starting point and for his encouragement through out the duration of the project

My Family,
For their ongoing support

My Friends, Dave, Lamine, Brahim
For being very helpful during difficulties

All My Lecturers, including Dr. Alwyn Barry and members of the Department of Computer Science especially, Ms. Louise Oliver
For their encouragement and help during all my studies at the University of Bath

Contents

1 Introduction

The aim of this project is to implement three different existing algorithms that multiply two integers. These algorithms are the Traditional, Karatsuba and Fast Fourier Transform. Each of these algorithms will be dealt with separately and then compared according to their complexity over time spent and space used to fulfil the multiplication. It is important to outline that there is no attempt to introduce a new method for multiplication or to optimise our routines to deal with special cases such as squaring a number or multiply a number by zero for example. However, when implementing the three algorithms we will try to focus on their theoretical definitions.

The tow integers to be multiplied can be of any size. They can be small integers two or three digits long or they can be up to thousands of digits long. From a computer science perspective, a large integer is anything too big to be stored in a data type.

There exist systems that deal with large integers (up to the machine limit) and perform all sorts of calculations in an effective and efficient way. These systems include the computer algebra systems such as Maple, Matlab and many others. There are even some programming languages that support multiplication of very large integers such as Haskell, Ruby, and Common Lisp.

In order to fulfil the aim of the project some steps have to be followed. These steps include researching the background areas behind multiplying integers, identifying the problems and breaking them into small sub-problems that can be dealt with separately and easily. In addition, topics related to this project have to be identified and understood. So, let us start by defining and introducing the most needed concepts related to this dissertation.

1.1 Integers:

As mentioned above, this project is about introducing algorithms that multiply two integers and comparing there efficiency according to the size of the integer. So, the need to understand what an integer is, what a large integer is and how to represent them is clear.

1.2 Integers in Mathematics:

“God created the integers: every thing else is the work of man” said the great mathematician Kronecker.

By definition an **integer** consists of the positive natural numbers (1, 2, 3 …), the negative natural numbers (−1, −2, −3 ...) and the number zero. The set of all integers is usually denoted in mathematics by **Z** (or, $\mathbb{Z}$), which stands for *Zahlen* (German for "numbers"). They are also known as the **whole numbers**, although that term is also used to refer only to the positive integers (with or without zero). Like the natural numbers, the integers form a countable infinite set. The branch of mathematics which includes the study of the integers is called number theory.

1.2.1 Integers in Computing:

An **integer** is denoted *int* and is one of the primitive data types which uses a fixed number of bytes, made up of eight bits in computer languages. However, these "integers" can only represent a subset of all mathematical integers, since "real-world" computers are of finite capacity even a variable-length representations eventually run out of storage space when trying to represent especially large numbers.
An *int* is represented as a 32-bit binary number that requires 4 bytes of storage and covers whole numbers (without fractional parts) between -2^{31} *to* $2^{31}-1$.

1.3 Large integers:

Large integers are often found in science, and scientific notation was created to handle both these large numbers and also very small numbers. 1.0×10^{10}, for example, means ten billion, a 1 followed by ten zeros: 10,000,000,000, and 1.0×10^{-9} means one billionth, or 0.0000000001. Writing 10^9 instead of nine zeros is compact and saves the reader the effort and hazard of counting a long string of zeros to see how large the number is. So, roughly speaking in scientific view a large integer can be defined as anything with more than 10 digits long.
However, in a computer context a large integer can be defined to be anything too big to be stored in a fundamental data type as described in the table below.

Table 1.3.1 Fundamental data types and their dimensions.

TYPE	NUMBER OF BYTES	MINIMUM	MAXIMUM
Int	4	-2147483648	+2147483647
Short	2	-32768	+32767
Unsigned short	2	0	+65534
Long	4	-2147483648	+2147483647
Unsigned long	4	0	+4294967295
Float	4	1.17549435e-38F	3.40282347e+38F
Double	4	2.2250738585072014e-308	1.07976931348623157e+308

It can be seen from the table above that computers are capable of handling very large numbers, for example when using floating point data type. So, why shall we worry?
Large numbers can be manipulated and represented using the appropriate data type. However, there are some limitations. These large numbers are finite. What happened if larger numbers are needed as we will see in the next section? What if we want to multiply numbers near the maximum size of the data type, then the resulting number will be too big to be stored and an inaccurate result will be outputted or an overflow will occur. Another related problem is the loss of precision when using floating point representation. Some one might say why not use data type float to represent all numbers. Let us consider this example:
The number: 123,456,789,123,456,789,333 is far too big to be stored in the type Int or Long for example. So, why not represent it as a float: 1.2345678e20 this representation loses precision in the last 13 digits and if we re-write the number, we get: 123,456,780,000,000,000,000 and when we are multiplying two large numbers we are only interested in the precise and exact product.
In this project, the size of an integer is generally a reference to the number of digits it has rather than its magnitude.

1.4 The need of very large integers

1.4.1 Large numbers in the everyday world

Some examples of large numbers describing every day real-world objects are:

- Cigarettes smoked in the United States in one year, on the order of 10^{12} (one trillion)
- The total NHS budget for instance, at around £30 billion
- Bits on a computer hard disk (typically 10^{12} to 10^{13})
- Number of cells in the human body > 10^{14}
- International phone cards have a pin number of at least 11 digits long.
- Number of neuron connections in the human brain, 10^{14} (estimated)
- Number of atoms in the visible universe, perhaps 10^{79} to 10^{81}, see Mass, Size, and Density of the Universe (*http://www.sunspot.noao.edu/sunspot/pr/answerbook/universe.html#q70*)
- Avogadro's number, approximately 6.022×10^{23}

1.4.2 Large numbers and computers:

According to Moore's Law computers are estimated to double in speed every 18 months. This made people believe that computers will be able to solve any mathematical problem. However, this is not the case; as no matter how fast computers become, there will be more complicated problems which they can not deal with.

Since the 1980's, computers' hard disk sizes have increased from about 10 megabytes to over 250 gigabytes. A 100 gigabyte hard disk can store the names of all humans on earth which estimated to be six billion without using data compression. Assuming we want to store all possible passwords containing up to 40 characters and each character is one byte (eight bits) there are 2^{320} such passwords, which is about 2×10^{96}. If every particle in the universe could be used as part of a huge computer, it could only store about 10^{90} bits (*http://arxiv.org/abs/quant-ph/0110141*), less than one millionth the size of the passwords we want to store.

In cryptography, algorithms make use of multiplication of very large integers. For example the SSL (Secure Socket Layer) protocol in order to trade safely on the web. Since we can safely assume that any cipher-breaking technique which requires more than 10^{120} operations will never be feasible. Many ciphers have been broken by finding efficient techniques that are not known to the cipher's designer.

Calculating the GCD (Greatest Common Divisor) between two 10000 digit numbers can be done by computing all their factors by trial division. This will take up to 2×10^{5000} division operations. But the Euclidean algorithm uses a much more efficient technique that takes only a fraction of a second to compute the GCD for even huge numbers.

Some research projects such the GIMPS distributed Internet deals with numbers having several million digits and use the Fast Fourier transform based multiplication algorithm.

1.5 How large integers are represented

1.5.1 Polynomial representation of integers

The integer X is handled thanks to it representation in a given base B (B usually depends on the maximal size of the basic data types):

$$X = P(B) = x_0 + x_1 B + x_2 B^2 + \ldots + x_n B^n.$$

So, each integer is represented by a chosen base (B), its sign (+ or −) and a succession of digits or *coefficients* x_i that satisfy $0 \leq x_i < B$.

So, for example:

The number:
1234 is represented as $\mathbf{4 * 10^0 + 3 * 10^1 + 2 * 10^2 + 1 * 10^3}$

(**10** is the base in this case)

The choice of the base B is important for several reasons:

- The base B must fit in a basic data type.
- The base B must be as large as possible to decrease the size of the representation of large integers and to decrease the cost of the basic algorithms running on them (addition, multiplication, ...), but sufficiently small such that internal operations running on the coefficients always fit in the basic data type chosen.

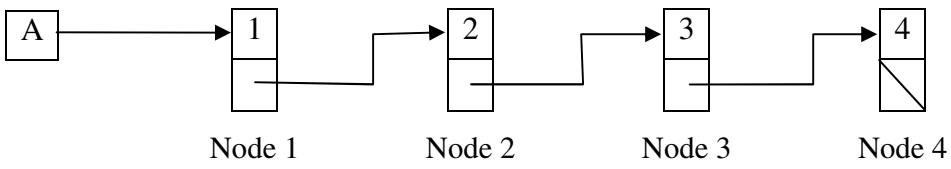
Once the integer is represented in a polynomial form all arithmetic operations such as subtraction, addition and multiplication are fairly easy. However, the product of two integers each contained in one word gives rise the problem of storing it. As, the latest requires double the size in this case two words for storing the product.

1.5.2 Linked lists representation of integers

Lists are made up of **nodes**, where each node contains a pointer to the next node in the list. In addition, each node contains a single digit of the integer.

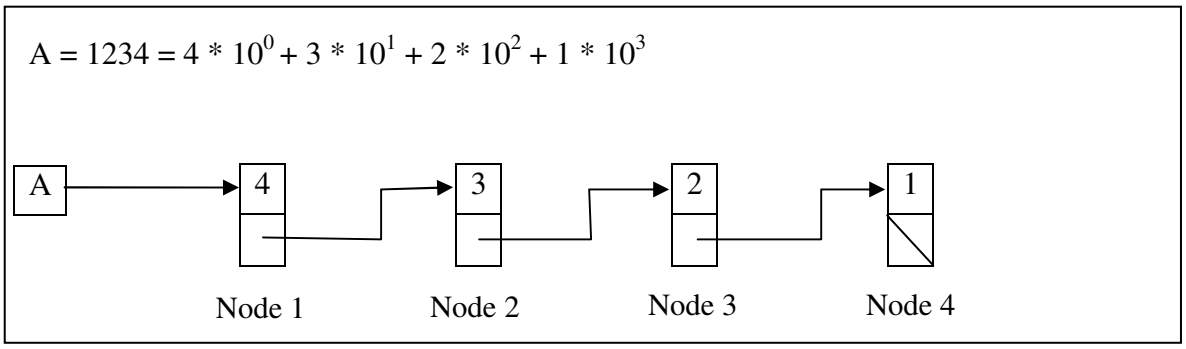
To illustrate this consider the integer 1234 represented by a linked list A in the following diagram.

$A = 1234 = 4 * 10^0 + 3 * 10^1 + 2 * 10^2 + 1 * 10^3$



The first node in list A points to represent the highest order digit in the integer given, which in turn points to the next node, which has the highest second element and so on. The final node (node 4 in the example) has a null pointer that does not point to anything.

An alternative representation is just the inverse of the above representation, and which would be to start with the lowest order digit and then the second and so on. And just to illustrate this, see the following diagram.



1.5.2.1 Advantages and disadvantages of linked list representation

Because a linked list stores a list of items, it has some similarities to an array. But the two are implemented quite differently. An array is a single piece of memory while a linked list contains as many pieces of memory as there are items in the list. Obviously, if your links get messed up, you not only lose part of the list, but you will lose any reference to those items no longer included in the list (unless you store another pointer to those items somewhere else).

Some advantages that a linked list has over an array are that you can quickly insert and delete items in a linked list. Inserting and deleting items in an array requires you to either make a room for new items or fill the "hole" left by deleting an item. With a linked list, you simply rearrange those pointers that are affected by the change. Linked lists also allow you to have different-sized nodes in the list. Some disadvantages to linked lists include that they are quite difficult to sort. Also, you cannot immediately locate, say, the hundredth element in a linked list the way you can in an array. Instead, you must traverse the list until you've found the hundredth element.

A linked list like I have described above, where each item has a pointer to the next item in the list, is called a singly linked list. To implement this list, you would also want to store a pointer to the first item in the list (the head), which you would use to access the other items. However, some operations are awkward with a singly linked list. For example, to remove an item, you may need to traverse the entire list to locate the item that came before the item you are removing in order to modify its NEXT pointer. For this reason, many linked lists are implemented as a doubly linked list. In a doubly linked list, each item contains a pointer to both the next and the previous item in the list. Because you may want to traverse the list in reverse order, you would probably want to store the last item in the list (the tail) in addition to the first item.

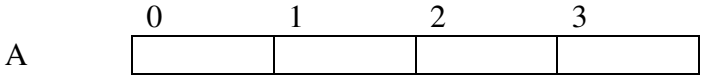
Insertion at *any* position in a linked list is a constant time operation, because it only involves changing a constant number of pointers. With an array, on the other hand, insertion requires moving all the elements past the point of insertion over to make

room for the new element, which takes linear time. A similar situation holds with deletion.
On the other hand, accessing (retrieving or storing) the value at a given position can be much slower for linked lists than for arrays. Given a position ***p*** in an array, if we need to obtain the value at some position ***p + k***, it can be done in constant time (i.e., independent of the size of ***k***), but in a linked list representation it takes linear time (linearly proportional to the size of ***k***). (With an array the address of the **(*p* + *k*)**-th position can be calculated as the address of position ***p*** plus ***k*** times the size of each value, so just with one multiplication and one addition operation; but with a linked list the only way to advance to position ***p + k*** from ***p*** is step by step, following links ***k*** times.)

1.5.3 Array representation of integers

Arrays are a series of elements (variables) of the same type placed consecutively in memory that can be individually referenced by adding an index to a unique name.
That means, for example, we can store 4 values of type `int` without having to declare 4 different variables each with a different identifier. Instead, using an *array* we can store 4 different values of the same type, `int` for example, with a unique identifier.
For example, an array to contain 4 integer values of type `int` called A could be represented this way:

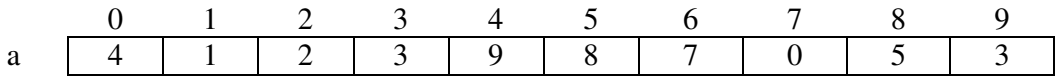
Array A of size 4



Where each blank panel represents an element of the array, that in this case are integer values of type `int`. These are numbered from **0** to **4** since in arrays the first index is always 0, independently of its length.

So, the **10** decimal digit integer **4,123,987,053** can be represented in a **10** cell array, a, of type int.

int a[10] = {**4,1,2,3,9,8,7,0,5,3**};



In each cell of the array could represent a digit of the integer we want to multiply.

There are obviously various configurations that we can employ apart from the one described above.

An alternative representation of number **4,123,987,053** would be to have the first digit "4" in the last cell of the array a. the second digit "1" in the N-2 cell assuming the array a has length N and the last digit "3" in the first cell which is at index 0.

int a[10] = {**3,5,0,7,8,9,3,2,1,4**};

	0	1	2	3	4	5	6	7	8	9
a	3	5	0	7	8	9	3	2	1	4

It is clear that Traditional multiplication uses the second representation as the multiplicands are multiplied left to right.

Advantages and disadvantages of array representation:

Arrays permit efficient (constant time, O(1)) random access but not efficient insertion and deletion of elements (which are O(*n*), where *n* is the size of the array). Linked lists have the opposite trade-off. Consequently, arrays are most appropriate for storing a fixed amount of data which will be accessed in an unpredictable fashion, and linked lists are best for a list of data which will be accessed sequentially and updated often with insertions or deletions.

Another advantage of arrays that has become very important on modern architectures is that iterating through an array has good locality of reference, and so is much faster than iterating through a linked list of the same size, which tends to jump around in memory. However, an array can also be accessed in a random way, as is done with large hash tables, and in this case this is not a benefit.

Arrays also are among the most compact data structures; storing 100 integers in an array takes only 100 times the space required to store an integer, plus perhaps a few bytes of overhead for the whole array. Any pointer-based data structure, on the other hand, must keep its pointers somewhere, and these occupy additional space. This extra space becomes more significant as the data elements become smaller. For example, an array of ASCII characters takes up one byte per character, while on a 32-bit platform, which has 4-byte pointers, a linked list requires at least five bytes per character. Conversely, for very large elements, the space difference becomes a negligible fraction of the total space.

Because arrays have a fixed size, there are some indexes which refer to invalid elements for example the index 17 in an array of size 5. What happens when a program attempts to refer to this varies from language to language and platform to platform.

In the previous examples the data type *int* has been used. This does not have to be the case. In fact, any of the fundamental data types such as float or double could have been used. We will see in the following sections that the Fourier transform starts by converting the integer digits from an array of data type char into an array of the floating type double as part of the process. When numbers are represented in arrays they are no longer operated on as a whole, instead they are operated on one digit at a time.

Another issue when representing large numbers is the choice of radix or base. A three-digit number in radix-256 is represented in array A of type char as:

	0	1	2
A	00011110	00001111	01010101

Its decimal equivalent is:

$$30 * 10^0 + 15 * 256^1 + 85 * 256^2$$
$$= 5574430$$

and would be represented, digit by digit, in an array a of char as:

	0	1	2	3	4	5	6
a	00000000	00000011	00000100	00000100	00000111	00000101	00000101

So, a decimal number of size **7** can be represented as a number of size **3** in base-256 which gives a significant saving in space.

Using large bases to represent numbers gives a significant saving of space. However, there are technical difficulties such as the numbers becoming more increasingly unmanageable. Assuming we are working in base-10 the biggest digits to be multiplied are 9 by 9 (000010001 * 000010001) and the product is 81 (01010001). The product 81 can still be represented in the same data type and requires only 1 bit word to store it. However, if we multiply 255 by 255 (11111111 * 11111111) in radix-256 the product 65536 (1111110111101111) requires at least 2 bit word to store it.

This will be discussed a little bit further in the design chapter.

1.6 What is an algorithm?

Informally, an algorithm is any finitely described computational procedure that takes a word in some formal language as *input* and produces some word as *output*. For each given *input* an algorithm works by performing a finite sequence of steps which are supposed to be elementary, i.e. very simple, and non-divisible.

Algorithms are tools for solving *computation problems*. Any computational problem consists of a description of possible inputs and an explanation how the output should be related to the input. In some cases it is not evident that a computational problem can be solved by any algorithm as there are some algorithmically unsolvable computational problems.

There exists a variety of formal definitions of the concept of algorithm: Turing machine, random-access machine (RAM), recursive function, λ-term, etc. All of them are equivalent, that is, any computational problem solvable with one definition is also solvable with any other. *Church Thesis* states that all reasonable formal definitions are equivalent.

1.6.1 What is Complexity of an algorithm?

Complexity is a measure of how efficient an algorithm is. Efficiency can be understood in many different ways. However, the most important characteristics of efficiency are:

The *time* that an algorithm takes or spends to generate an *output* from an *input* and is usually referred to as the function *T(n)* of the working time or running time. This depends on other factors like: the programming language, the compiler and the speed of the CPU of the computer being used.

The amount of memory used by an algorithm

Algorithms are compared by calculating the complexity of each one and then comparing the complexities. The next section will demonstrate this in more detail.

1.6.2 How to compare algorithms?

One of the main objectives of this project is to compare the three algorithms. So, how is this done?
First of all, an algorithm that produces an incorrect result is considered to be 100% inefficient and is the worst thing that might happen. When comparing two algorithms the main points to look at are how much time it spends and how much memory it uses. So, basically compare their complexities.

Algorithms are compared according to their complexities which are function defined on natural numbers. One of many different ways to compare algorithms is the big O notation which is defined as follow:
Suppose $f(x)$ and $g(x)$ are two functions defined on some subset of the real numbers. We say
$f(x)$ is $O(g(x))$ as $x \longrightarrow \infty$
if and only if
there exist numbers x_0 and M such that $|f(x)| \leq M\,|g(x)|$ for $x > x_0$.

The function *f(x)* in this project is the time taken for the algorithm to perform the multiplication one or a given number of times, and *g(x)* is the theoretical 'big oh' performance of the algorithm. For each of our multiplication algorithms we will look at the value *f(x) / g(x)*. As x approaches infinity this ratio will be checked if it approaches a value M. If it does approach this ratio then the implementation is said to be behaving close to its theoretical performance.
The notation can also be used to describe the behaviour of f near some real number a: we say
$f(x)$ is $O(g(x))$ as $x \longrightarrow a$
if and only if
there exists numbers $\delta>0$ and M such that $|f(x)| \leq M\,|g(x)|$ for $|x - a| < \delta$.
In mathematics, both asymptotic behaviours near ∞ and near a are considered. In computational complexity theory, only asymptotic near ∞ are used; furthermore, only positive functions are considered, so the absolute value bars may be left out.
Big O notation is useful when analyzing algorithms for efficiency. For example, the time (or the number of steps) it takes to complete a problem of size n might be found to be:

$T(n) = 4n^2 - 2n + 2$.
As n grows large, the n^2 term will come to dominate, so that all other terms can be neglected. Further, the coefficients will depend on the precise details of the implementation and the hardware it runs on, so they should also be neglected. Big O notation captures what remains: we write

$$T(n) \in O(n^2)$$

and say that the algorithm has *order of* n^2 time complexity. We will see in the next section that this complexity is actually equal to the traditional multiplication complexity.

1.6.3 Multiplication algorithms

A **multiplication algorithm** is an algorithm (or method) to multiply two numbers. Depending on the size of the numbers, different algorithms are in use.

1.6.3.1 Long multiplication / Traditional Multiplication

Multiply the first number with every digit of the second number and then add up all the properly shifted results. In order to perform this algorithm, one needs to know the products of all possible digits, which is why multiplication tables have to be memorized. Humans use this algorithm in base 10, while computers employ the same algorithm in base 2. The algorithm is a lot simpler in base 2, since the multiplication table has only 4 entries. Rather than first computing the products, and then adding them all together in a second phase, computers add the products to the result as soon as they are computed.

The pen and paper long multiplication method would probably look like:

			5	1	2
		*	2	5	6
			30	06	12
		25	05	10	
+	10	02	04		
	10	27	39	16	12
1	3	1	0	7	2

This algorithm will now be described algebraically.

Given an integer A, with base x and length m, we can represent it as:

$$A(x) = A_0 + A_1 x + A_2 x^2 + ... + A_{m-1} x^{m-1}$$

Given an integer B, with same base as A and length n, we can represent it as:

$$B(x) = B_0 + B_1 x + B_2 x^2 + ... + B_{n-1} x^{n-1}$$

A, B can be multiplied together giving the product C:

$$C(x) \times A(x) \times B(x) = C_0 + C_1 x + C_2 x^2 + ... + C_{m+n-1} x^{m+n-1}$$

The product C is in base *x* and of length *m+n-1*.

A pseudo-code version of the Traditional algorithm is presented in the design chapter.

To multiply two numbers with *n* digits using this method, one needs about n^2 multiplications, $n^2/2$ additions. By using the notation of Landau, the cost in calculating the multiplication is $O(n^2)$, i.e. quadratic. The traditional algorithm becomes extremely slow when multiplying very large numbers.

1.6.3.2 Karatsuba multiplication

For systems that need to multiply huge numbers in the range of several hundreds of digits, such as computer algebra systems, the traditional algorithm is too slow. These systems employ **Karatsuba multiplication** which was discovered in 1962 and proceeds as follows: suppose you work in base 10 (unlike most computer implementations) and want to multiply two *n*-digit numbers *U* and *V*, where *n* is even and equal to 2*n* (if *n* is odd instead, or the numbers are not of the same length, this can be corrected by adding zeros at the left end of *U* and/or *V*).

The two numbers *U* and *V* are written as:

$$\begin{cases} U(X) &= a_0 + a_1 X + a_2 X^2 + \ldots + a_{n-1} X^{n-1} \\ V(X) &= b_0 + b_1 X + b_2 X^2 + \ldots + b_{n-1} X^{n-1} \end{cases}$$

which is just:

$$\begin{cases} U(X) &= \sum_{i=0}^{n-1} a_i X^i \\ V(X) &= \sum_{i=0}^{n-1} b_i X^i \end{cases}$$

with the radix *X* in which the calculation is performed. Human use the radix 10. it seems reasonable as we have 10 fingers !

if we do the multiplication *P(X)=U(X)V(X)*, we obtain the polynomial:

$$P(X) = a_0 b_0 + (a_0 b_1 + a_1 b_0) X + (a_0 b_2 + a_1 b_1 + a_2 b_0) X^2 + \ldots + a_{n-1} b_{n-1} X^{2n-2}$$

This can be written in a more compact manner as:

$$P(X) = \sum_{k=0}^{2n-2} \sum_{i+j=k} a_i b_j X^k$$

With, $0 \leq i, j \leq n-1$.

By isolating the coefficient of the different exponents of X we get:

$$\begin{cases} P_0 & = & a_0b_0 \\ P_1 & = & a_0b_1 \\ P_2 & = & a_0b_2 + a_1b_1 + a_2b_0 \\ \dots & = & \dots \\ P_{2n-2} & = & a_{n-1}b_{n-1} \end{cases}$$

Thus, the multiplication of *U* by *V* becomes the calculations of *P0* to *P2n-2*. So, we have replaced the big problem (product *U*V*) by a series of small problems (the *Pi's*). if the numbers to multiply are very big, this turns to be a problem of managing the small problems. In 1962, A. Karatsuba discovered a method to reduce the number of these calculations.

In order to make what we have just said more clear. Assume that we want to multiply two numbers *U* and *V* of two digits.

$$\begin{cases} U(X) & = & a_0 + a_1X \\ V(X) & = & b_0 + b_1X \end{cases}$$

The product

$$P(X) = U(X)V(X)$$

is written:

$$P(X) = a_0b_0 + (a_0b_1 + a_1b_0)X + a_1b_1X^2$$

which is written as:

$$P(X) = p_0 + p_1X + p_2X^2$$

The idea of Karatsuba is to have:

$$\begin{cases} q_0 & = & a_0b_0 \\ q_1 & = & (a_0 + a_1)(b_0 + b_1) \\ q_2 & = & a_1b_1 \end{cases}$$

it is clear to see that:

$$q_1 = p_0 + p_1 + p_2$$

Knowing that:

$q_0 = p_0$ and $q_2 = p_2$, we get:

$$\begin{cases} p_0 & = & q_0 \\ p_1 & = & q_1 - q_0 - q_2 \\ p_2 & = & q_2 \end{cases}$$

So, we can get the product *P(X)* by using 3 multiplications instead of 4. There is a price to pay: some more additions and subtractions are needed.

As an example, suppose we want to multiply 12345678 by 87654321. We can write these numbers as:

$12345678 = 1234 * 10^4 + 5678$
$87654321 = 8765 * 10^4 + 4321$

So, $a_0 = 5678,\ a_1 = 1234,\ b_0 = 4321,\ b_1 = 8765$

$U * V = 5678 * 4321 + ((5678 * 8765) + (1234 * 4321)) * 10^4 + 1234 * 8765 * 10^8$
$= 24534638 + (49767670 + 5332114) * 10^4 + 10816010 * 10^8$
$= 24534638 + 550997840000 + 1081601000000000$
$= 1082152022374638$

A pseudo-code version of Karatsuba's algorithm is presented in the design chapter.

If $T(n)$ denotes the time it takes to multiply two n-digit numbers with Karatsuba's method, then we can write :

$T(n) = 3\ T(n/2) + cn + d$

for some constants c and d, and this recurrence relation can be solved, giving a time complexity of $O(n^{\log(3)})$. The number log 3 is approximately 1.585, so this method is significantly faster than long multiplication. Because of the overhead of recursion, Karatsuba's multiplication is not very fast for small values of n; typical implementations therefore switch to long multiplication if n is below some threshold.
It is possible to experimentally verify whether a given system uses Karatsuba's method or long multiplication: take two 100,000 digit numbers, multiply them and measure the time it takes. Then take two 200,000 digit numbers and measure the time it takes to multiply those. If Karatsuba's method is being used, the second time will be about three times as long as the first; if long multiplication is being used, it will be about four times as long. So, even karatsuba's algorithm is considerably faster than long multiplication algorithm; there exist a faster algorithm for multiplying integers.

1.6.3.3 Fast Fourier Transform:

There exist even faster algorithms, based on the **fast Fourier transform**, a 'devide and conquer' algorithm, such as the Karatsuba algorithm. The idea, due to Strassen (1968), is the following: multiplying two numbers represented as digit strings is virtually the same as computing the convolution of those two digit strings. Instead of computing a convolution, one can instead first compute the discrete Fourier transforms, multiply them entry by entry, and then compute the inverse Fourier transform of the result. The fastest known method based on this idea was described in 1971 by Schönhage and Strassen (Schönhage-Strassen algorithm) *(http://numbers.computation.free.fr/)* and has a time complexity of $O(n \log(n))$.

Before we describe how the FFT algorithm works, complex roots of unity have to be explained and understood.

Complex Roots of Unity

A complex number is represented by the equation

$$z = x + yi$$

where x and y are real numbers and i is the imaginary number $\sqrt{-1}$ and $i^2 = -1$.

It can also be represented in the polar form

$r(\cos \Theta + i \sin \Theta)$

where r is the modulus and theta Θ is the argument of $x + yi$.

If z is a complex number, $z^n = 1$ has n roots.

One root is z = 1 and, if n is even, z = -1.

The rest are found in the following way

$z = \cos \Theta + i \sin \Theta$

$= (\cos \Theta + i \sin \Theta)^n = 1$, by DeMoivre's theorem

$\cos n\Theta + i \sin n\Theta = 1$

$\cos n\Theta = 1$ and $\sin n\Theta = 0$

$n\Theta = 2\pi k$

$\Theta = 2\pi k / n$

Where k is an integer, k = 0, 1… n-1

So, n roots of z^n are

$z = 1$
$z = \cos 2\pi/n + i \sin 2\pi/n$

$z^2 = \cos 4\pi/n + i \sin 4\pi/n$
…
$z^{n-1} = \cos 2\pi(n-1)/n + i \sin 2\pi(n-1)/n$
DeMoivre's Theorem

So, a 5th root of unity would have 5 points where $z^5 = 1$. The roots of unity play an important role in the Fourier transform. The FFT is based upon choosing a special set of points (complex roots of unity) on which to do the evaluation and interpolation.

So, how does the FFT algorithm work?

Two integers A and B, of length n represented as a polynomial in base x. It is important to stress at this stage that the length n has to be a power of two (n is even). In the implementation there will be some processing to ensure that n is always even.

$$A(x) = A_0 + A_1 x + A_2 x^2 + ... + A_{n-1} x^{n-1}$$
$$B(x) = B_0 + B_1 x + B_2 x^2 + ... + B_{n-1} x^{n-1}$$

Split A, and B in the following manner:

$A_0 = a_0 + a_2 + ... + a_{n-2}$ and $A_1 = a_1 + a_3 + ... + a_{n-1}$

$B_0 = b_0 + b_2 + ... + b_{n-2}$ and $B_1 = b_1 + b_3 + ... + b_{n-1}$

Both halves are equal to *n/2* as the length *n* is a power of two. However, one half possesses the even positions and the other the odd positions.

As an example we would split 123456 int the two parts 135 and 246.
Now, given a sequence A = (a_0, a_2, … , a_{n-2}), compute its Fourier transform according to the formulae
Choosing for w_k the complex roots of unity

(Note: *i* is equal to $\sqrt{-1}$)

$$w_k = \exp\left(\frac{2ik\pi}{2n}\right) = \omega^k, \quad \omega = \exp\left(\frac{2i\pi}{2n}\right)$$

This formulae is used to compute the complex roots of unity needed to evaluate the number at *2n* distinct points.
This would give us:

$$F_{2n}(A) = (c_0, c_1, \ldots, c_{2n-1}), \qquad b_k = \sum_{j=0}^{2n-1} a_j \, \omega^{jk}.$$

From above we get the sequence $F_{2n}(B) = (d_0, d_1, \ldots, d_{2n-1})$,

The two sequences are multiplied together and produce the third sequence E

$E = (e_0, e_1, \ldots, e_{2n-1})$, where $e_k = a_k * b_k$

On sequence E, we use the inverse Fourier Transform. This produces the sequence G

Table 1.1

$$G = (g_0, g_1, \ldots, g_{2n-1}), \qquad g_k = \sum_{j=0}^{2n-1} e_j . \omega^{-jk}$$

Finally, dividing each of the resulting integers by 2n will give us the coefficients that construct the product of the multiplication.

A worked example:

Multiply 54 and 93

Step1. Perform Fourier transform on 54.

V0 = 4 + 5 = 9;
V1 = 4 + 5 * exp(2*Pi*I/4) = 4 + 5I;
V2 = 4 + 5 * exp(2*Pi*I*2/4) = -1;
V3 = 4 + 5 * exp(2*Pi*I*3/4) = 4 - 5I;

The same on 93.

U0 = 3 + 9 = 12;
U1 = 3 + 9 * exp(2*Pi*I/4) = 3 + 9I;
U2 = 3 + 9 * exp(2*Pi*I*2/4) = -6;
U3 = 3 + 9 * exp(2*Pi*I*3/4) = 3 - 9I;

Step2. Combine V and U By multiplying them together.

W0 = V0 * U0 = 108;
W1 = V1 * U1 = -33 + 51I;
W2 = V2 * U2 = 6;
W3 = V3 * U3 = - 33 – 51I;

Step3. Perform the inverse Fourier transform

Z0 = W0 + W1 + W2 + W3 = 48
Z1 = W0 + W1 * exp(-2 * Pi * I/4) + W2 * exp(-4 * Pi * I/4) + W3 * exp(-6 * Pi * I/4) = 204;
Z2 = W0 + W1 * exp(-2 * Pi * I*2/4) + W2 * exp(-4 * Pi * I*2/4) + W3 * exp(-6 * Pi * I*2/4) = 180;
Z3 = W0 + W1 * exp(-2 * Pi * I*3/4) + W2 * exp(-4 * Pi * I*3/4) + W3 * exp(-6 * Pi * I*3/4) = 0

Divide these integers by 4 we get

Z0 / 4 = 12;
Z1 / 4 = 51;
Z2 / 4 = 45;
Z3 / 4 = 0;

Step4. Perform the carrying on the coefficient to get the final product.
12 + (51 * 10) + (45 * 100) = 5022.

A pseudo-code of the Fourier transform algorithm is presented in the design chapter.

In the practice, one should use the basic data type double in C to handle floating point numbers appearing in the Fourier transform. Numerical errors appear during the computation. If the numerical errors are sufficiently small, they can be overpassed :

each final value z_i after the reverse FFT should be an integer, thus we must take its nearest integer value, the fractional part giving the error obtained.

Complexity of the FFT

In fact, the time complexity of multiplication with FFT is a little bigger than n log(n). Let us be more precise. To multiply two numbers of N digits, we write them in a base B which contains k digits (say $B = 10^k$), thus giving a number of coefficients equal to $n \cong N/k$. The discussion above tells us that to multiply those two numbers, FFT permits to perform O(n log(n)) operations on basic numbers (basic numbers express coefficients in the base B, they are usually basic numerical data types like double in C). Because of the numerical error bound, these basic numbers should be precise enough to represent integers up to $6n^2 B^2 \log(n)$.

For example, working in double precision, the base B should be chosen small enough so that the error bound is not too large... and this is not even possible if n is too large). Thus the number of digits of these basic numbers should be of the order of log(B)+log(n). As a consequence, the basic operations on these numbers has cost $O((\log(B)+\log(n))^2)$ and the final cost is $O(n \log(n) (\log(B) + \log(n))^2)$. The base B is chosen so that $k = \log_{10}(B)$ is of the same order than log(n), and finally, multiplying those two numbers of $N \cong kn$ digits have cost

$$O(n \log(n)^3) = O(N \log(N)^2).$$

Thus the cost of *Strassen multiplication* (this is the name of the floating-FFT we presented) to multiply numbers of N digits is $O(N\log^2(N))$.

If the Strassen's multiplication is also used to compute the operations on the basic numbers (one recursive level), the cost reduces to $O(N\log(N)\log\log^2(N))$. The best bound is obtained with complete recursive version and is O(Nlog(N) loglog(N) logloglog(N)).

The lowest known theoritical bound is obtained with the *Schönhage multiplication*, which generalizes the process by working in finite rings, and is O(N log(N) loglog(N)) *(see http://www.loria.fr/~zimmerma/bignum/option.ps.gz for a description of Schönhage multiplication).*

1.7 Summary:

In this chapter most of the materials and main areas related to this project have been identified, researched and understood. This formed the basis for the main project work such as requirement specification, design and implementation. The software and accompanying documentation must be produced by the author. There will be no one else working on the project.

2 Requirements Analysis

In this chapter, the major requirement analysis of this project will be discussed, outlined and determined. In the next chapter these requirements will be formally described. The requirement specification will give a foundation for the design of the software.

2.1 Project Goals

As mentioned previously, the goal of this project is to implement three different algorithms for multiplying two integers. These two integers can vary in size and length. These three implementations will be compared according to the size of the integers and the time taken for the multiplication to be performed and the result being outputted.

This project is neither about finding better algorithms for multiplying integers, nor about improving the different multiplication algorithms available. It is simply implementing the provided algorithms and comparing them.

2.2 Constraints

2.2.1 Software Constraints

A variety of software is available the developer of this project. This includes software available via the University of Bath computer labs. This includes windows XP operating system, UNIX operating system, various compilers such C, C++, Java, etc
Software available on the developer's personal computer, including windows XP Home Edition operating system, various compilers such java, C, C++, etc

2.2.2 Hardware Constraints

This includes the developer's personal portable computer running windows XP Home Edition as operating system with an Intel Pentium 4, 2.4 GHz processor and 512 Mb of RAM as well as the University of Bath PC's on campus. The developer will be most of the time using his own computer for compiling, running, and testing the software.

2.2.3 Time Constraint

Draft software and accompanying documentation is due to the 16th April 2005. The final and complete software and documentation is due to the 16th May 2005.

2.3 Data input and output

The input consists basically of two integers (the two integers to be multiplied). At this point there are some issues to be considered such what base would the two integers be represented in? Would the two integers be of same size or different sizes? Do they have to be of the same sign? And the main issue is how big or what length can the two integers be?

As the two integers could be very large in size, It would not seem practical to type them. Therefore, it would make sense to implement a function to generate two random integers depending on the needed size for our experiments and tests. The two

generated integers would be stored in a file and accessed from there as needed. The output file will include the result or the product of the multiplication, the length of the two integers and the time taken to multiply them.

2.4 Choice of User Interface

It would be possible to provide the user with a graphical user interface (GUI) to facilitate the interaction between the user and the system. For example the interface might have two fields where the user types in the two integers to be multiplied and a multiplication button for the calculation to be carried out. The interface might have radio buttons to select which multiplication algorithm (the traditional, Karatsuba's or the Fourier Transform) to use.

However, as outlined before this project is not concerned with developing software package aimed for general use or for business purposes. This project is not for developing new routines for integer multiplication or improving the existing algorithms. Therefore, the developer is the main user of this program and there will be no need for a graphical user interface, it will be basically executed and tested using command line on the given operating system.

2.5 Choice of Development and Implementation Environment

A decision about what development and implementation environment to use has to be made after considering all the software and hardware available to the developer and what operating system is best suited for this project and what programming language to use after comparing the most known and mastered by the developer.

2.6 Programming Language

As mentioned before the developer is familiar with java, C, C++. A comparison between these three languages is outlined in the next sections in order to determine the advantages and disadvantages of each one in order to make a decision on which one to choose for the implementation of the algorithms.

2.6.1 Advantages of Java:

- Garbage collection in java facilitates programming and the safety of program execution considerably as programmers do not need to dynamically allocate memory and then remember to free it later.
- Java is easier to learn because of its streamlined syntax. It inherits less baggage from the C language, most notably in bypassing the need to deal with pointers.
- The lack of pointers in java means that buffer overflow bugs (and, consequently, security exploits) are practically impossible in Java. Built in bounds checking prevents an errant program from overwriting the end of a memory buffer or assigning the value of an incompatible data type to a variable.
- These built in checking mechanisms in Java make it a more robust software environment. On the other hand, they also make Java execution slower (see below).

- The fact that compiling to machine code (which most Virtual Machines do now) is done at run time means that as better compilers are developed, the same byte codes will often perform better automatically. This is in contrast to C++, where taking advantage of a new compiler optimization requires recompiling.
- Java can be programmed for multiple platforms with little regard towards platform-specific characteristics like hardware data types, floating point implementations, or OS libraries. Java programs are compiled into binary byte code which will execute properly on any standards-compliant JVM, on any architecture, without modification.
- Although the language specification is controlled by Sun Microsystems, the specification of both the language and the platform are freely distributed. This is opposed to the C++ standard, which must be purchased from ISO, a disadvantage for students and others who do not use the language for commercial gain.

2.6.2 Disadvantages of Java

- No compile-time "template" generic containers, until recently. Version 1.5 (Java 2 version 5) has generics, although there are differences from C++.
- No support for multiple inheritance. Instead, Java supports interface classes that can not implement any methods but otherwise behave like abstract base classes. Multiple inheritance allows better program organization and avoids code duplication.
- Java does not support destructors but it provides call to *finalize* method when the garbage collector destroys the object. This makes harder to use automatic resource management. Instead any used operating system resources have to be released by hand or by implementing *finalize* method. (e.g. by calling Close() methods). This makes exception handling sometimes quite complicated and hard to get it right without leaking resources.

2.6.3 Advantages of C++

- Because C++ is designed to compile to native code, native code compilers for C++ are more mature than those for Java. Programs written in C++ typically consume less memory than those written in Java, and are often more efficient as well. This applies particularly to digital signal processing and other arithmetic-heavy code.
- In C++, the programmer does not incur overhead for features that they do not use. For example, automatic garbage collection and mandatory virtual members make Java performance unsuitable for some applications.
- It's possible to fit a C++ runtime library into a small memory and storage footprint, letting C++ code run on tiny embedded systems where even Java 2 Platform, Micro Edition won't run.
- C++ has a much more robust model for enforcing constants. It also provides stricter and more specific casting keywords.
- C++ provides pointers and inline assembler, which can at times be easier than the high-level workaround.

2.6.4 Disadvantages of C++

- The presence of hardware pointers makes it easy to write a program that will inadvertently overwrite code in memory with data. This leads to buffer overflow security holes. A workaround is to always check that the data fits in the specified buffer.
- The programmer must specifically free any dynamically allocated memory, or else create a memory leak which can eventually exhaust all available system memory. The risk can be reduced by prudent selection of third-party libraries and the use of features such as smart pointers.
- C++ takes longer to learn and the syntax is less forgiving. However, the basic syntax of C++ is very similar to that on Java. The differences arise only when using the more advanced features of the language.

2.6.5 Advantages of C:

- A simple core language, with important functionality such as math functions or file handling provided by sets of library routines instead
- Focus on the procedural programming paradigm, with facilities for programming in a structured style
- A simple type system which prevents many operations that are not meaningful
- Use of a pre-processor language, the C pre-processor, for tasks such as defining macros and including multiple source code files
- Low-level unchecked access to computer memory via the use of pointers
- Parameters that are always passed to functions by value, never by reference
- Function pointers, which allow for a rudimentary form of closures and polymorphism
- Lexical variable scoping
- Records, or user-defined aggregate data types (structs) which allow related data to be combined and manipulated as a whole

2.6.6 Disadvantages of C

- C permits many operations that are generally not desirable, and thus many simple errors made by a programmer are not detected by the compiler or even when they occur at runtime, leading to programs with unpredictable behaviour. Part of the reason for this is to avoid compile and runtime checks that were too expensive when C was originally designed. Rather than placing these checks in the compiler, additional tools, such as lint, were used. Today many tools are available to allow a C programmer to detect or correct various common problems.
- One problem is that automatically and dynamically allocated objects are not initialized; they initially have whatever value is present in the memory space they are assigned. This value is highly unpredictable, and can vary between two machines, two program runs, or even two calls to the same function. If the program attempts to use such an un-initialised value, the results are usually unpredictable. Most modern compilers detect and warn about this problem in some restricted cases.
- Pointers are one primary source of danger; because they are unchecked, a pointer can be made to point to any object of any type, including code, and

then written to, causing unpredictable effects. Although most pointers point to safe places, they can be moved to unsafe places using pointer arithmetic, the memory they point to may be de-allocated and reused, they may be un-initialized, or they may be directly assigned any value using a cast or through another corrupt pointer. Another problem with pointers is that C freely allows conversion between any two pointer types. Other languages attempt to address these problems by using more restrictive reference types.

- Although C has native support for static arrays, it does not verify that array indexes are valid. For example, one can write to the sixth element of an array with five elements, yielding unpredictable results. This is called a *buffer overflow*. This has been notorious as the source of a number of security problems in C-based programs.
- Another common problem is that heap memory cannot be reused until it is explicitly released by the programmer with free(). The result is that if the programmer accidentally forgets to free memory, but continues to allocate it, more and more memory will be consumed over time. This is called a *memory leak*. Conversely, it is possible to release memory too soon, and then continue to use it. Because the allocation system can reuse the memory at any time for unrelated reasons, this results in insidiously unpredictable behaviour. These issues in particular are ameliorated in languages with automatic garbage collection such as Java.

After a deep and a close look at the languages; the author has decided on the C programming language for the following reasons:

- C is quite powerful and can easily be used to work with problems at the bit level of abstraction and this is very helpful in this project.
- It is a relatively small language and procedural in nature.
- C is easy to learn and works extremely well for small projects.
- Programs written in C can be designed to be modular in the sense that small programs; representing different levels of abstractions kept in different files even though it is not an object oriented language.
- Because there are a small amount of objects involved, it would seem sensible to choose a language that suitable to small problems.
- Both C++ and Java are object oriented and have more extra attractive features. However, these are not necessary in this project. Besides, these extra features will they will yield to a longer running programs and this will affect the timing of the programs.

2.7 Choice of Compiler

As we chose the C programming language it would seem sensible to go for the C compiler too. The C compiler available to the developer is: *gcc* compiler on both UNIX and windows.

2.8 Choice of Operating System

The developer has two choices of operating systems. SunOS 5.9 UNIX operating system is available at University of Bath computer labs as well as windows operating system in both the University and at home. The use of the windows XP operating

system would mean that the developer could use his home PC to work on the project from home. The use of the operating system windows may be seen as being less stable than a UNIX operating system. Any code written will be re-compiled and tested on the SunOS system in order to demonstrate the software at the University Of Bath.

2.9 Comparison of algorithms

One of the main goals of this project is to compare the performance of the algorithms. This could be done by:

- Comparing the time it takes to produce the product of the multiplication of the same two numbers in each algorithm.
- The number of instructions executed by the algorithms.

In order to compare the three algorithms there are a number of points that have to consider the first place, which are:

- The same programming language and compiler must be used when implementing the algorithms.
- The same hardware and operating system must be used when running the programs for comparison reasons.

2.10 How the Experiment Should Be Performed

There will be a random generate number function that generates random numbers of varying length to a file each time the program is run. The same random numbers will be passed to each of the multiplication algorithm. Then, there will be an output file with the following information:
The result obtained in each algorithm. This is to check if the same product of the two integers is obtained in each algorithm.
The length of the two integers being multiplied.
The time spent producing the result in each algorithm.

The time and length of the two integers are needed in order to compare the efficiency of each algorithm. The timing of the code should be as true reflection as possible. This will not be easy to obtain as it might seem as there will a lots of factors that affect this; such as how much RAM is left in the computer, how many processes are running, etc…

2.11 Testing

It is good programming practice to test the software through out development and it must be tested before submission to ensure it meets the specification. The result of the multiplication has to be tested for correctness. It would not seem sensible to test that the algorithms output a correct result for each two number less than 10000 for example. Instead a set of sample numbers would be investigated. As we mentioned before; the output file will contain the result obtained by multiplying two numbers in each of the algorithms. So, the first thing to do would to compare that the result obtained in each algorithm is identical for the same two numbers that were multiplied. If the two numbers are small and the product is less than 15 digits, we can use a simple calculator to check the correctness of the result. Hopefully, the developer of this project has access to Maple and Matlab which will be used to check the correctness of the multiplication for very large numbers say up to 10000 digits.

2.12 Documentation

The submission of the project consists of an implementation of the three algorithms and documentation. The documentation will cover the main areas relating the production of the software. The documentation will consist of the following:

- An introduction
- Requirement analysis and specification
- Design of algorithms
- Design of experiments
- Results of experiment
- Analysis of results
- Conclusion/Critique
- A bibliography
- List of Source Code
- List of results of testing

A print out of the documentation will be submitted along with a copy of the software on a CD. A demonstration of the software may be given to the project supervisor.

2.13 Training

There is no training needed for the software that will be produced. This project is an experimental project and any software produced is not aimed at any specific user.

2.14 Security

There are no specific security issues regarding this project. The only security to be taken into account is to back up any software and documentation produced.

2.15 Maintenance

There are no particular maintenance features to this project.

3 Requirements Specification

3.1 Project Goals

The goal of this project is to implement three algorithms. The implementation will be as close as possible to the theoretical descriptions. The efficiency of each algorithm will be investigated and experiment will be performed in order to compare them according to their domains of competence.
In this project there is no attempt to make the algorithms more efficient. No special cases will be dealt with such as multiplying by zero or squaring a number.

3.2 Hardware and Software Constraints

The developer own machine will be used throughout the development of the project this is a laptop running a windows XP on a Pentium 4 2.4 GHz processor and 512MB of RAM.

3.3 Time Constraint

The project must submitted by the 16th May 2005.

3.4 User Input

As there is no point typing the numbers; even if we do it would impractical to type numbers of hundreds of digits. Therefore, there will be a function that generates random numbers of a specific length. These numbers will be stored in a file and read appropriately from there for the program to multiply them.

3.5 Choice of User Interface

There will be no need for a graphical user interface; as it is only an experimental project. The software will be executed via command line.

3.6 Programming Language

The C programming language will be used to develop the software.

3.7 Choice of Compiler

The *gcc* compiler will be used to write the C source code of this project.

3.8 Choice of Operating System

The windows XP home edition operating system will be used to develop the software. This will be running on laptop with a Pentium 4 2.4 GHz processor and 512MB of Ram. The same operating system will be used to perform the experiments. The software will be made sure to run on a UNIX SunOS 5.9 operating system for demonstration purposes.

3.9 Performing the Experiments

For each of the given two integers, the time taken when executing each algorithm will be calculated. The integers multiplied will be randomly generated. However, their lengths will be dealt with when designing the experiments. The algorithms will be compared according the time taken multiplying the same two integers. Then, their

performance will be discussed and some graphs will be drawn to show what have just been discussed.

3.10 Testing

The software will be tested throughout development of the software. A final test will ensure that the software meets the requirements. A detailed test plan will be produced.

3.11 Documentation

The submission of the project consists of an implementation of the three algorithms and documentation. The documentation will cover the main areas relating the production of the software. The documentation will consist of about 70 pages and include of the following:

- A brief introduction to the various concepts needed in this project.
- Analyses of the requirement the system should satisfy.
- Requirement specification: specify the requirements the project must satisfy.
- A detailed design of the different algorithms to be implemented as well as the other function they use.
- Experiments: The experiments to be performed need to be designed. Then, the results obtained from these experiments will be analysed and the performance of each algorithm must be determined.
- Conclusions/Critique
- List of Source Code
- List of results of testing

3.12 Training

There no special training necessary for this project.

3.13 Security

Buck up copies of all work carried out will be kept for security reasons.

3.14 Maintenance

There are no specific maintenance issues related to this project.

3.15 Summary

In this chapter, the requirements that the final software must meet have been listed. The next chapter mainly focuses on the design of the system.

4 System Design

The software will consist of two different programs. The first program is aimed at producing random numbers. These random numbers are the multiplicands. This program will output the randomly generated numbers into a file. The second program holds the multiplication routines. The Traditional, Karatsuba, and Fourier are implemented in here. This program will output the product of the multiplication into a file. This output file will hold all information needed to carry the experiments and comparisons.

In chapter 1, different ways to represent integers were outlined. The advantages and disadvantages were discussed in order to work out which one is the best to use throughout the development and implementation of the different algorithms.
The array representation of integers is a lot simpler to code in C and more time efficient than the linked list representation. However, the linked list representation is more space efficient than the array representation.

As the algorithms are easier to compare according to the amount of time they spend multiplying the same two multiplicands. It is wiser to choose time efficiency over space efficiency. This leads to the conclusion that the integer representation to be used is the array representation.

It remains to decide on what data type the arrays should be and what base or radix the integers have to be represented in.

In all cases and what ever base is chosen we always need to convert the final integers in the resulting array to base 10 as it is easy to check for correctness.The developer is aiming to implement the software so that it copes with different bases in order to see the behaviour of the algorithms in different choices of bases. It is important to stress that this is not part of the requirement specification but just to see if it really affects the performance of the algorithms.

Some data types in the programming language C were introduced in chapter 1. For each one of those there are the unsigned versions. The unsigned versions just deal with positive numbers and as we are dealing with large positive integers it makes sense to use them. Double or float data types are used when dealing with approximation numbers and as multiplying numbers such 2.34, and √43232 it does not seem helpful to use them. Besides, it seems appropriate to use the data type int as we are dealing with integers but just the positive side. So, we shall use the unsigned int data type which 4 bytes of storage (full-word).

Figure 4.1 Overall System design

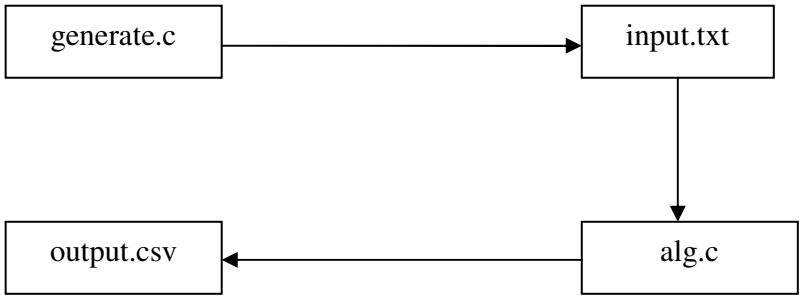
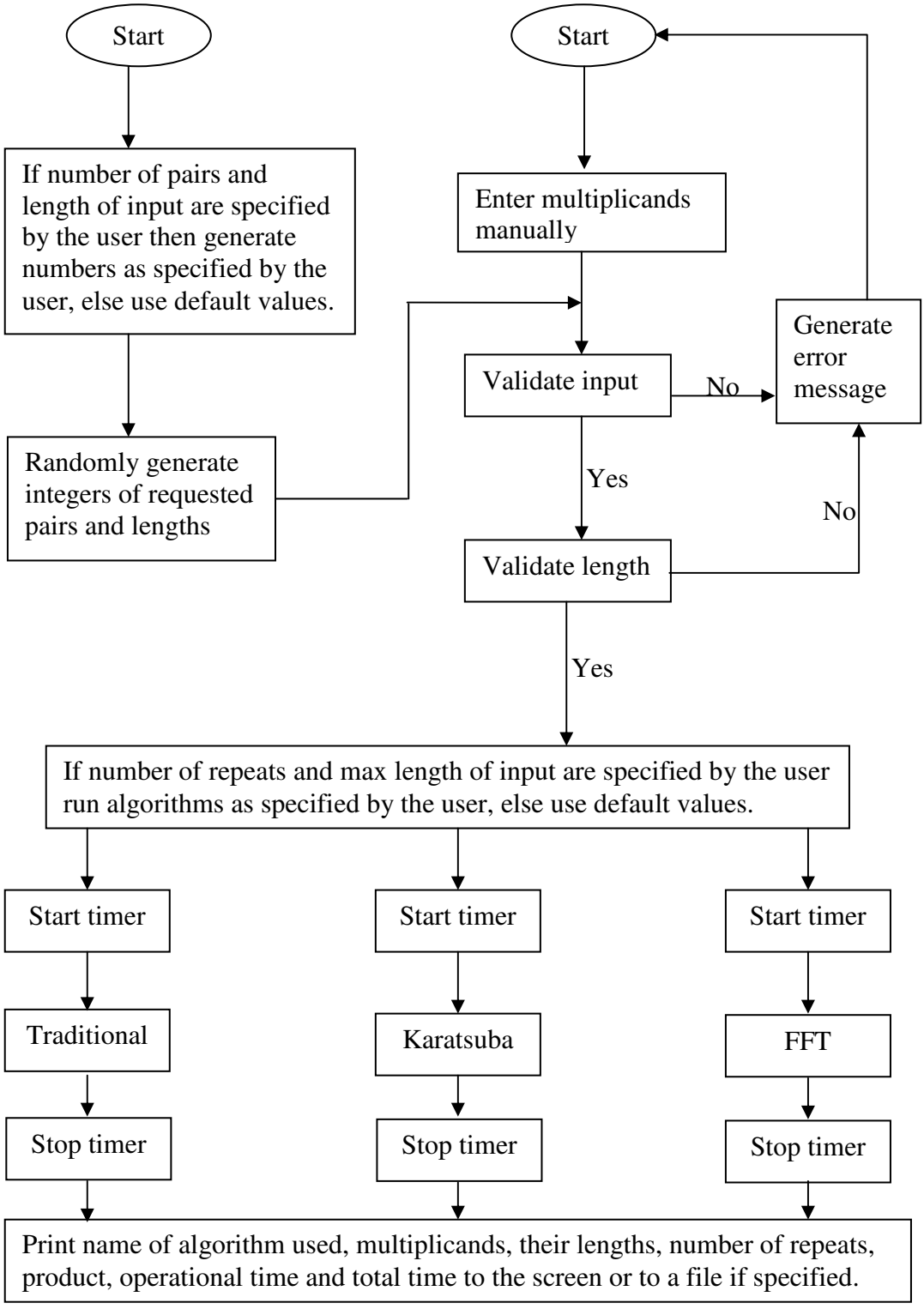


Figure 4.2 Flow Chart to show how the program will run



4.1 Software Design

In this section the author gives a detailed description of how the main modules are designed, a pseudo code of all algorithms used.

4.1.1 File alg.c

This file contains the integer multiplication algorithms as well as the functions they use. All of these functions are described below and a pseudo-code of each one is given.

4.1.1.1 Function read_input()

This function reads the inputs either from the screen or from a file. It reads the input character by character and stores it in array already initialised to zero.
This function does the following checks:
If the end of the file is reached then the message 'end of input reached' is displayed on the screen.
If a none positive integer is found then the message 'input was not an integer' is displayed.
If the number is larger than the program can handle then the message 'not set to handle input size larger than 100000 digits long' is displayed.
If the escape character is pressed followed by a return, then quit the program.
If no input was found, then notify the user.
If a new line is found, then we finished reading the input.
These characters in the array are then stored backward as Traditional and Karatsuba multiplication is from right to left.

```
void read_input(){

        declare variables;

        call fuction set_zero() to initialise the array to zeros;

        while( true ){
                call getchar() function to get characters one by one from the input;

                if(character == End Of File){
                        print end of input reached;
                        exit;
                }
                if(escape key is pressed){
                        print user terminated program;
                        exit;
                }
                if(new line found)     break;
                if(next character is less than 48 or next character is greater than
57){
```

```
                    In ASCII notation, 48 = 0 and 57 = 9
                    disregard any other characters
                    print Input was not an integer;
                    exit;
            }
            If(length of input is greater than the length allowed)
                    print each multiplicand must be less or equal to maximum
                    length of input handled;
                    exit;
            }
    }

    if(no input found){
            print Zero length input found;
            exit;
    }

    for(i = 0; i < array length; i++){
            store array backward
    }

}
```

4.1.1.2 Function set_zero()

This fuction initialise any array passed to it to zero.

```
void set_zeros(array to be initialised){

    declare variables;
    fill array with zeros;

}
```

4.1.1.3 Function trad_multiply()

This is the pseudo code of the Traditional multiplication technique. It multiplies each digit of the first array with all digits of the second array and so on.

```
void trad_multiply(two arrays to be multiplied, product array){
    declare variables;
    for(i = 0; i < array length; i++){
            for(j = 0; j < array length; j++){
                multiply each digit of array of small length with each digit of
                the second array, and store results in a third array of size twice
                the largest array;
            }
    }
```

```
}
```

4.1.1.4 Function Karatsuba()

This is the pseudo code of the Karatsuba multiplication technique.

```
void karatsuba(two arrays to be multiplied, product array){

        declare variables and arrays;

        if (input is of size less than 4 digits long){

                call trad_multiply() to perform multiplication of small inputs;
                return;
        }
        else{
                allocate memory for the two halves of the two arrays;

                if(no memory allocated){
                        print 'Failed to allocate memory';
                }
                for(i = 0; i < (array length / 2); i++){
                        add the appropriate halves of the arrays appropriately;
                }

                Recursive calls to karatsuba() to keep dividing the arrays into
halves;

                for (i = 0; i < array length; i++){
                        perform the appropriate subtractions;
                }

                for(i = 0; i < twice array length; i++){
                        construct the final array that holds the product;
                }
                Free allocated space using the function free();
        }

}
```

4.1.1.5 Function carry()

This function takes as input the returned arrays from trad_mul() and karatsuba() functions which holds the product arrays and perform the carrying on them in order to insure that each slot of the array has a decimal number (numbers from 0 to 9).
lets consider the following example:

Before the function carry the product array is:

	0	1	2	3	4	5	6
a	6	16	8	12	5	13	0

Function carry does the following:

```
                        6
                  1     6
               0  8
            1  2
         0  5
1        3
-------------------------
1     3  6  2  9  6  6
```

After the function carry the product array is:

	0	1	2	3	4	5	6
a	6	6	9	2	6	3	1

The pseudo-code for carry() function is the following:

```
void carry(product array){
        declare variables;
        set variable tmp to be 0;

        for(I = 0; I < array length; i++){
                array[i] += tmp;
                tmp = array[i] / 10;
                array[i] -= tmp * 10;
        }
}
```

4.1.1.6 Function print_array()

This function simply receives the array holding the product from the function carry() and store it backward and prints it out to the screen or to a file. So,

before the function print_array() the product array is:

	0	1	2	3	4	5	6
a	6	6	9	2	6	3	1

after the function print_array() the product array is:

	0	1	2	3	4	5	6
a	1	3	6	2	9	6	6

The pseudo-code for print_array() function is the following:

```
void print_array(product array unsorted){
        declare variables;
```

```
        while(array[ i ] == 0){
                i--;
        }
        while(i >= 0){
                printf(print array);
                i--;
        }
}
```

4.1.1.7 Function read_fourier_input()

This function reads the inputs either from the screen or from a file. It reads the input character by character and stores it in array already initialised to zero.
It works just as the read_input() function described above. The only difference is that it does not store the array backward (hence Fourier multiplication is from left to right).

The pseudo-code for read_fourier_input() function is the following:

```
void read_fourier_input(){

        declare variables;
        call fuction set_zero() to initialise the array to zeros;
        while( true ){
                call getchar() function to get characters one by one from the input;

                if(character == End Of File){
                        print end of input reached;
                        exit;
                }
                if(escape key is pressed){
                        print user terminated program;
                        exit;
                }
                if(new line found)      break;
                if(next character is less than 48 or next character is greater than 57){
                        In ASCII notation, 48 = 0 and 57 = 9
                        disregard any other characters
                        print Input was not an integer;
                        exit;
                }
                If(length of input is greater than the length allowed)
                        print each multiplicand must be less or equal to maximum
                        length of input handled;
                        exit;
                }
        }
        if(no input found){
                print Zero length input found;
                exit;
        }
}
```

4.1.1.8 Function fourier()

The Fast Fourier Transform algorithm is by far the most difficult to implement. The function fourier() uses the drealft() function in order to calculate the product of the two multiplicands.
The pseudo-code for fourier() function is the following:

```
void fourier(two input arrays, product array, lengths of the two arrays to be
multiplied){
        declare necessary variables;
        while(array length < largest array length){
                get next power of two;
        }

        Call dvector() function to allocate memory for the multiplicands;

        for(i = 1; i <= array length; i++)
                convert arrays from data type unsigned int to data type doubles;

        for(i = 3; i <= array length; i += 2){
                multiply the real and imaginary parts in each term;
        }

        Call drealfr(array, array length, flag = -1);

        Initialise to variable cy to 0.0;

        for(i = array length; i >= 1; i--){
                perform carrying on the product array;
        }

        if (cy >= base){
                print 'value greater than the base';
        }

        for(i = 2; i <= twice array length; i++){
                output result;
        }
        free allocated space using the function free();
}
```

4.1.1.9 Function drealft()

This function divides the input array into even and odd halves. It then calls the discrete_fourier_transform() function to perform the discrete Fourier Transform if flag is 1, or perform the inverse if flag is -1.

The pseudo-code for drealft() function is the following:

```
void drealft(array, array length, flag){
        declare variables;
```

```
        if(flag == 1){
                set up for forward transforms;
                discrete_fourier_transform(input array, array length, flag = 1);
        }else{
                Set up for reverse transforms;

        }
        Split input into even and odd halves;
        Recombine the data;
        }

        if(flag == 1){
                set up for forward transforms;
        }else{
                set up for forward transforms;
                discrete_fourier_transform(input array, array length /2,flag = -1);
        }
}
```

4.1.1.10 Function discrete_fourier_transform()

This function replaces the input data by either its discrete Fourier transform if the flag returned by the drealft() function is equal to 1, or by its inverse discrete Fourier transform if the flag is equal to -1.

The pseudo-code for discrete_fourier_transform() function is the following:

```
void discrete_fourier_transform(input array, array length, flag){

        declare variables;
        initialise n to the next power of 2;
        initialise j to 1;

        for(i = 1; i < n; i += 2){
                if( j > i ){
                        carry out the bit reversal of the data in the array;
                        swap each digit in the same array;
                }
                Shift the binary bits of n by 1 by one and assign it to variable m;
                while(m >= 2 && j > m){
                        decrement the value j by value m;
                        shift the binary bits of m by one;
                }
                Increment the value of j by m;
        }
        Initialise mmax to 2;
        while(n > mmax){

                for ( m = 1 ; m < mmax; m += 2 ){
```

```
                    for (i = m; i <= n; i += istep){
                        convert the values in the array to their Fourier
                        trandform values;
                    }
                }
                mmax = istep;
        }
}
```

The previous three functions all together construct the Fast Fourier Transform multiplication technique.

NOTE: The implementation of the Fast Fourier Transform is an adaptation of an existing code written in the C programming language.

4.1.1.11 Function dvector()

This function allocates space for arrays of type double used in the Fourier multiplication routine

The pseudo-code for dvector() function is the following:

```
double dvector(array of type double){

        allocate space to an array of type double;

}
```

4.1.1.12 Function print_fourier_array()

This function prints the product array to the screen or to a file as specified by the user.

The pseudo-code for print_fourier_array() function is the following:

```
void print_fourier_array(product array, array length){
        initialise necessary variables
        delete the leading zero's in the product array;
            print the input as stored in the array;
}
```

4.1.1.13 Function main()

This is the driver function of the all the different algorithms.

The pseudo-code for main() function is the following:

```
int main(int argc, char** argv) {

        declare variables

        if(argc > 1){
```

```
        for (i = 1; i < argc; i++) {
                if (argv[i][0] == '-' || argv[i][0] == '/') {
                        switch (argv[i][1]) {
                            depending on the user input, select which
                            algorithm to use for multiplication. Case T | t for
                            Traditional, K | k for Karatsuba and F | f for
                            Fourier. Otherwise, print to the screen the
                            message 'invalid choice entered'.
                        }
                }
        }
}
else{
        if no arguments passed, print help info to the user
}


switch(algorithm){
        repeat the switch statement to avoid repeating ourselves if the user
        runs for example 'alg -t -k', in this case run 'karatsuba'
}

if(input length is greater than maximum input length){
        print 'input length is larger than the maximum length allowed';
}

if(repeats not equal to ){
        print how many times the program is repeated
}
print out a header, only print border on stdout not to the .csv file

print which algorithm is carrying the multiplication, the multiplicands,
product, number of repeats, length of the number to be multiplied, time
for one run of algorithm, total time.


while( true ){

        if(algorithm chosen is FOURIER){

                call read_fourier_input(first multiplicand, address of the
                length of a, maximum length);

                call read_fourier_input(second multiplicand, address of the
                length of b, maximum length);
        }
        else{
                call read_input(afirst multiplicand, address of the length of
                a, maximum length);
```

```
                call read_input(second multiplicand, address of the length
                of b, maximum length);
        }

        Check which multiplicand is larger in length

        if(algorithm chosen is TRADITIONAL){

                print input and output results

                call print_array(array a, largest length);
                call print_array(array b, largest length);

                call to start calculating time

                for (i = 0; i < num_repeats; i++){
                        call set_zeros(array c which will hold the result,
                        (twice maximum length));
                        call trad_multiply(array a, array b, array c, largest
length);
                        call carry(array c, (twice largest length));
                }

                Call to stop time;

                Calculate total time:  end time – start time;
                Calculate Operation time: total time / number of repeats;

                Call print_array(c, (2 * largest_length));
        }
        else if(algorithm chosen is KARATSUBA){

                shift bits up a power of two using C's built in shift operator
                power = 1;
                while(less than largest length){
                        get the next power of two greater largest length;
                }
                Set largest length to be equal to power;

                print input and output
                 call print_array(array a, largest length);
                 call print_array(array b, largest length);

                call to start calculating time;

                for (i = 0; i < num_repeats; i++){
                        call set_zeros(c, (twice maximum length));
                        call karatsuba(a, b, c, largest length);
                        call carry(c, (twice largest length));
                }
```

```
                    Stop timer;

                    Calculate total time:  end time – start time;
                    Calculate Operation time: total time / number of repeats;

                    Call print_array(c, (twice largest_length));

            }
            else if(algorithm chose is FOURIER){

                    print input and output results

                    call print_fourier_array(a, a_length);

                    call print_fourier_array(b, b_length);

                    call to start calculating time

                    for (i = 0; i < num_repeats; i++){
                            call set_zeros(c, (2 * max_length));
                            call fourier(a, b, c, a_length, b_length);
                    }

                    Stop timer;

                    Calculate total time:  end time – start time;
                    Calculate Operation time: total time / number of repeats;

                    Call print_fourier_array(c, (a_length + b_length));

            }
          print results number of repeats, largest length, operation time, total
          time
      }
}
```

4.1.2 File generate.c

This file contains three functions they are as follow:

4.1.2.1 Function create_seed()

In order to generate true random numbers it is suggested to use as *seed* a value that changes often, like the one returned by *time* function included in <time.h> (the number of seconds elapsed since newyear 1970).

The pseudo-code for create_seed() function is the following:

```
int create_seed( void ){
      initialize random generator like the one returned by time function
      included in <time.h>
```

```
}
```

4.1.2.2 Function random_digit()

This function randomly generates a digit.

The pseudo-code for random_digit() function is the following:

```
int random_digit( void ){
        declare necessary variables;
        generate a random decimal digit;
}
```

4.1.2.3 Function main()

This is the driver program. The other two functions are called from here. The random_digit() function is called recursively and the digits returned from it are stored in an array of size equal to either the length the user specified or the default one.

The pseudo-code for main() function is the following:

```
main(int argc, char** argv){

        declare necessary variables;

        if(argc > 1){
                for (i = 1; i < argc; i++) {
                        if (argv[ i ][ 0 ] == '-' || argv[ i ][ 0 ] == '/') {
                                switch (argv[ i ][ 1 ]) {

                                case 'p' or ‘P’:
                                        set number of pairs to be the number
                                        specified by the user;
                                        break;
                                case 'd' or  'D':
                                        set Difference to the difference specified by
                                        the user;
                                        break;
                                default:
                                        Print ‘Invalid switch encountered’;
                                        break;
                                }
                        }
                }
        }
    if(number of pairs not equal to the default one){
                Print ‘Generating “number” pairs of integers’;
    }else{
                Print Generating the default “number” pairs of integers to change
```

```
                use flag '–p';
    }

        if(difference between pairs not equal to the default one){
                Print 'Generating pairs with %d digit(s) difference\n", difference);

        }else{
                Print Using the default difference to change use flag –d';

        }

        Call the function create_seed();

        for(i = 1; i <= num_arrays; i++){

                Generate pairs of random numbers with first pair of length
                difference then increment by i * difference;

                Generate pairs of random numbers with first pair of length
                difference then increment exponentially

                if(maximum length > maximum length allowed){
                        break;
                }

                for(j = 0; j < maximum length; j++){
                Fill multiplicands or array pairs with these random generated
                digits;

                print output;
                }

                Print new line in order to distinguish between the integers
generated;

                for(j = 0; j < max_length; j++){
                        print output;
                }

                Print new line in order to distinguish between the integers
generated;

        }

}
```

4.2 Summary:

In this chapter, each of the functions used to build the program were specified and then a pseudo-code for each one was given.

5 Software Testing

The objective of this chapter is to introduce techniques that will be used to test the software and to discover program faults. The first technique to be used is white box testing and the second is the black box testing.

White box testing is applied to relatively small program units. Analysis of code will be used to find out how many tests need to be run in order to guarantee that all components are executed at least once during the testing process.

In Black box testing, the functionality of the software will be tested. Inputs are presented to the system and the corresponding outputs are examined. If the outputs are not equivalent to those predicted then the test has successfully detected a fault with the program.

5.1 White box testing

Each section and sub-section of the code is to be tested. So, that each function carries out what it is meant to do properly and the system in general executes as expected, produces correct results and meets its specification.
Let us look at the different paths the software can take by looking at the "if, else" clauses and "while" loops in the alg.c file.

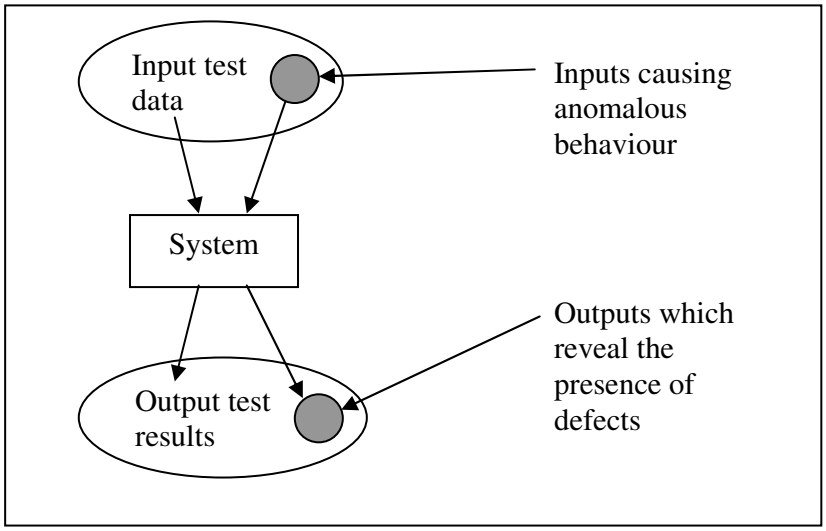
Function:	**Code to test:**	**To execute code:**	**To avoid code:**
main()	1st IF clause	Assuming you called the compiled program alg. Type alg –t or alg /t when you first run program	Type alg when you first run the program
	1st ELSE clause	Type alg when you first run the program	Type alg –t or alg /t when you first run program
	2nd IF clause	Using a maximum length of digits for the multiplicands not equal to default maximum length	Using a maximum length of digits for the multiplicands equal to default maximum length
	2nd ELSE clause	Using a maximum length of digits for the multiplicands equal to default maximum length	Using a maximum length of digits for the multiplicands not equal to default maximum length
	3rd IF clause	Number of repeats of the algorithm not equal to the default number of repeats	Number of repeats of the algorithm equal to the default number of repeats
	3rd ELSE clause	Number of repeats of the algorithm equal to the default number of repeats	Number of repeats of the algorithm not equal to the default number of repeats
	4th IF clause	The user chose to perform	Choose to run the

		the multiplication using Fast Fourier Transform	code with Traditional or Karatsuba
	4th ELSE clause	The user chose to run the code with Traditional or Karatsuba	Run the code with Fast Fourier Trasform
	5th IF clause	Choose to run the code with Traditional technique	Choose to run the code using Karatsuba's or FFT techniques
	1st ELSE IF clause	Choose to run the code with Karatsuba's technique	Choose to run the code using Traditional or FFT techniques
	2nd ELSE IF clause	Choose to run the code with FFT technique	Choose to run the code using Traditional or Karatsuba's techniques
read_input()	1st IF clause	End of file reached	End of file has not been reached yet
	2nd IF clause	Press Escape	Do not press escape
	3rd IF clause	New line found	No new line found
	4th IF clause	Enter positive decimal numbers	Enter characters including the '-' character
	5th IF clause	Enter a number bigger in length than the maximum number length allowed	Enter a number less or equal in length to the maximum length allowed
	6th IF clause	Enter no input	Enter some input
Karatsuba()	1st IF clause	Enter a number less in length than 4	Enter a number that has at least 4 digits
	2nd IF clause	Enter a number that can not be stored in the computer memory	Enter a number that can be stored in the computer memory
read_fourier_input()	1st IF clause	End of file reached	End of file has not been reached yet
	2nd IF clause	Press Escape	Do not press escape
	3rd IF clause	New line found	No new line found
	4th IF clause	Enter positive decimal numbers	Enter characters including the '-' character
	5th IF clause	Enter a number bigger in length than the maximum number length allowed	Enter a number less or equal in length to the maximum length allowed
	6th IF clause	Enter no input	Enter some input

File name and purpose	Test criterion	Results
File: generate.c Purpose: to generate a file with random numbers.	That the random numbers are truly random. The random numbers generated are not just repeated. That each pair of random numbers is of the same size. That the sizes of random numbers do not exceed the maximum length specified in the header file. That the difference in size between each pair of random numbers is different as specified.	The numbers in the generated file are effectively random, with no repeats in sequences. The file has exactly the number of random numbers as specified. The different in size between the pairs of random numbers is exactly as specified. The sizes of the random numbers do not exceed the maximum length specified in the code. The file tested positively and is fit for use.
File: alg.c Purpose: Read input from the generated file or from command line if no file specified. Perform the multiplication using one of the techniques at a time. Calculate the time taken to perform the multiplication. Print the result and the time either to the screen or to a file.	That the input is read properly. The chosen algorithm is used for the multiplication. That a precise and accurate result of the result is printed. That time taken to produce the result is accurate.	Invalid inputs were detected. The valid inputs were read properly and the right multiplication algorithm was used as specified by the user. A correct result of the multiplication was produced. An accurate calculation of the time taken by the algorithm to produce the result was printed.

5.2 Black box testing

Our objective from running the black box is to make sure that all algorithms implemented produce a correct and accurate result. The diagram below gives an overview about how these testing will be performed.



(Based on Ian sommerville (2001) Software Engineering) pp.444

For small numbers, testing the accuracy and correctness of the results can be achieved by running the individual routines and comparing it to the results obtained from a pocket calculator. (See the following screen shot)

For Traditional technique

```
amos $ alg -t -r 10000

        +----------------------------+
        | Big Integer Multiplication |
        +----------------------------+

debug:  Using Traditional algorithm...
debug:  Accepting a maximum of 135168 digits
debug:  Repeating 10000 times

--------------------------------------------------------------------------------
algorithm, a, b, result, num_repeats, largest_length, operation_time (ms), total_time (ms)
--------------------------------------------------------------------------------

a:      11
b:      12
traditional, 11, 12, 132, 10000, 2, 0.001000, 10.000000

a:      13
b:      0
traditional, 13, , , 10000, 2, 0.001000, 10.000000

a:      12345
b:      56789
traditional, 12345, 56789, 701060205, 10000, 5, 0.004000, 40.000000

a:      324
b:      543232
traditional, 324, 543232, 176007168, 10000, 6, 0.006000, 60.000000

a:      ^[
debug:  User terminated program
amos $
```

For Karatsuba technique

```
amos $ alg -k -r 10000

        +----------------------------+
        | Big Integer Multiplication |
        +----------------------------+

debug:  Using Karatsuba algorithm...
debug:  Accepting a maximum of 135168 digits
debug:  Repeating 10000 times

--------------------------------------------------------------------------------
algorithm, a, b, result, num_repeats, largest_length, operation_time (ms), total_time (ms)
--------------------------------------------------------------------------------

a:      12345
b:      56789
karatsuba, 12345, 56789, 701060205, 10000, 5, 0.022000, 220.000000

a:      123
b:      1234521
karatsuba, 123, 1234521, 151846083, 10000, 7, 0.022000, 220.000000

a:      1234
b:      10
karatsuba, 1234, 10, 12340, 10000, 4, 0.006000, 60.000000

a:      324
b:      543232
karatsuba, 324, 543232, 176007168, 10000, 6, 0.023000, 230.000000

a:      ^[
debug:  User terminated program
amos $
```

For FFT technique

```
amos $ alg -f -r 10000

        +----------------------------+
        | Big Integer Multiplication |
        +----------------------------+

debug:  Using Fourier algorithm...
debug:  Accepting a maximum of 135168 digits
debug:  Repeating 10000 times

-------------------------------------------------------------------------------
algorithm, a, b, result, num_repeats, largest_length, operation_time (ms), total_time (ms)
-------------------------------------------------------------------------------

a:      11
b:      11
fourier, 11, 11, 121, 10000, 2, 0.007000, 70.000000

a:      123
b:      321
fourier, 123, 321, 39483, 10000, 3, 0.013000, 130.000000

a:      12345
b:      56789
fourier, 12345, 56789, 701060205, 10000, 5, 0.024000, 240.000000

a:      1252
b:      12354432128
fourier, 1252, 12354432128, 15467749024256, 10000, 11, 0.046000, 460.000000

a:      ^[
debug:  User terminated program
amos $
```

For very large integers, testing the accuracy and correctness of the results can be achieved by running the individual routines on these same results and outputting the results to a file. Then, using the UNIX command *diff* or the windows command prompt *fc* to compare the three files generated. If no difference is found; it is assumed that the three different programs outputted the same correct results for the multiplication.

Generate a file of 5 pairs of integers with difference 500; call it file input.txt

```
amos $ generate -d 500 -p 5 > input.txt
debug:  Generating 5 pairs of integers
debug:  Generating pairs with 500 digit(s) difference
debug:  The last pair has 2500 digit(s) in length
amos $
```

The last pair has indeed 2500

Run Traditional technique to multiply the pairs in input.txt file and storing the output in test_traditional.csv

```
amos $ gcc alg.c -lm -o alg
amos $ alg -t -r 500 -m 2500 < input.txt > test_Traditional.csv

        +----------------------------+
        | Big Integer Multiplication |
        +----------------------------+

debug:  Using Traditional algorithm...
debug:  Accepting a maximum of 2500 digits
debug:  Repeating 500 times

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------

a:      b:
a:      b:
a:      b:
a:      b:
a:      b:
a:      debug:  End of input reached
amos $
```

Run Karatsuba technique to multiply the pairs in input.txt file and storing the output in test_Karatsuba

```
amos $ alg -k -r 500 -m 2500 < input.txt > test_Karatsuba.csv

        +----------------------------+
        | Big Integer Multiplication |
        +----------------------------+

debug:  Using Karatsuba algorithm...
debug:  Accepting a maximum of 2500 digits
debug:  Repeating 500 times

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------

a:      b:
a:      b:
a:      b:
a:      b:
a:      b:
a:      debug:  End of input reached
amos $
```

Run FFT technique to multiply the pairs in input.txt file and storing the output in test_FFT.csv

```
amos $ alg -f -r 500 -m 2500 < input.txt > test_FFT.csv

        +----------------------------+
        | Big Integer Multiplication |
        +----------------------------+

debug:  Using Fourier algorithm...
debug:  Accepting a maximum of 2500 digits
debug:  Repeating 500 times

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------

a:      b:
a:      b:
a:      b:
a:      b:
a:      b:
a:      debug:  End of input reached
amos $
```

These files have been designed to contain all necessary information such as time spent to multiply each pair which is needed in the experiments, the integers being multiplied, their lengths and the results. However, we are interested in comparing only the results. So, delete the unnecessary information and keep the pairs to be multiplied, results, numbers of repeats and length.

Running comparisons in UNIX:

```
amos $ diff test_Traditional.csv test_Karatsuba.csv
amos $
```

Only differences between the two files are returned. Files tested to be the same.

```
amos $ diff test_Karatsuba.csv test_FFT.csv
amos $
```

All files are exactly the same.

Conclusion: the results are accurate and correct.

Running comparisons in windows command prompt:

```
H:\big_integer_multiplication>fc test_Traditional.csv test_Karatsuba.csv
Comparing files test_Traditional.csv and TEST_KARATSUBA.CSV
FC: no differences encountered

H:\big_integer_multiplication>
```

```
H:\big_integer_multiplication>fc test_FFT.csv test_Karatsuba.csv
Comparing files test_FFT.csv and TEST_KARATSUBA.CSV
FC: no differences encountered

H:\big_integer_multiplication>
```

The developer of this project did also choose random pairs from these test files, copied and multiplied them in maple and the results were exactly as those obtained from the three programs.

Summary

The tests show that the programs are fit for use and ready to perform experiments on them.

6 Algorithm efficiency and experiments

In this chapter, experiments are performed in order to determine the efficiency of each of the algorithms implemented in order to be able to compare them. The experiments will be carried out using the BUCS *amos* machine unless stated otherwise.
All the implemented algorithms were run on the same sets of inputs (integers varying in length from 2 to 100000 digits as shown in the following tables) to compare there speed.

The sets of inputs are obtained by executing the generate function, which generates pairs of integers with variant length. Using the flags –d and –p we can specify the length and number of pairs needed.

For each different size of input integers, the implemented algorithms were run a number of times and the time to accomplish the task was recorded. Then, by dividing the total time by the number of repeats we get a good approximation to the exact time that the different algorithms take for one run.

For inputs of up to 150 digits, programs were run 10000 times using the flag *-r*. Then, the numbers of repeats is decreased to 10 when multiplying integers of length ranging from 10000 to 100000.

Decreasing the number of repeats will result in calculating the time spent by the algorithms to perform the multiplication less effectively. There is no other option as the system will run out of memory.

Suppose the executable file of *alg.c* is called *alg* and the execucutable file of *generate.c* is called *generate*

$ generate –d 2 –p 5 > input.txt

This generates the file input.txt with 5 pairs of integer of lengths 2, 4, 6, 8 and 10. By changing the code slightly, it can be generates pairs' lengths of powers of any number such:

$ generate –d 3 –p 3 > input.txt

will generate 3 random integer pairs of lengths: 3, 9 and 27. Then,

$ alg –t –r 10000 < input.txt >> output.csv

-t flag says carry the multiplication of the numbers in the input.txt file using Traditional technique, repeat the multiplication 10000 times (notice the *-r* flag) and generate a new file output.csv. It is possible to set a maximum length of the multiplicands using the flag –*m*. so, for example –m 10 says the maximum length the multiplicands can have is 10.

To run Karatsuba just use the flag –*k*, and –*f* for Fast Fourier algorithm. When the length of the multiplicands is of about 80000 digits running the Traditional algorithm twice using the flag –*r* resulted in CPU LIMIT EXCEEDED message which just means that the system has run out of memory. To overcome this and run the traditional algorithm at least

10 times for inputs of sizes in the range 90000 digits long. We had to change our tactics slightly. The new method consisted of setting the flag –r to do one repeat. Then create a file of ten or so of these commands as follows:

```
alg –t  –r 1 < input.txt > output.csv
alg –t  –r 1 < input.txt >> output.csv
                .
                .
                .
alg –k  –r 1 < input.txt >> output.csv
                .
                .
alg –f  –r 1 < input.txt >> output.csv
```

Call it *experiments* for example and save it with no extension. See the following clipboard:

```
amos $ flip -u experiments
amos $ chmod 777 ../big_integer_multiplication/experiments
amos $ experiments

        +---------------------------+
        | Big Integer Multiplication |
        +---------------------------+

debug:  Using Traditional algorithm...
debug:  Accepting a maximum of 135168 digits
debug:  Repeating 1 times

------------------------------------------------------------
------------------------------------------------------------

a:      b:
```

Then, get an average of the time spent to carry the multiplication. This method has been proved to be efficient and meets our requirement specification.

The new generated file includes the following data:

	In file output.csv	**Description**
1	Algorithm	Algorithm chosen to carry the multiplication
2	A	First multiplicand
3	B	Second multiplicand
4	Result	Result or product
5	num_repeats	Number of times the algorithm has been repeated for
6	largest_length	Largest length of the two multiplicands
7	operation_time (ms)	Time taken to carry the multiplication one time
8	total_time (ms)	Time taken to carry the multiplication N times

To perform the experiments; we only need to know the length of the integers being multiplied and the time performance of each algorithm. These data are used for comparisons between algorithms and they are presented below in tables and the appropriate Graphs are produced using Microsoft Office Excel.

It is predicted that there will be a point where Karatsuba's algorithm becomes faster than the Traditional algorithm and a point where Fast Fourier Transform becomes faster than both of them. The purpose of these experiment is to find out at what integer length this will happen.

All of the algorithms were implemented to run until the system runs out of memory. For the Traditional algorithm the BUCS *amos* machine runs out of memory when the multiplicands are of 100,001 digits long. However, Karatsuba and Fourier algorithms still keep running correctly. The results obtained from running the Traditional, Karatsuba and Fourier on the same 100,000 digits integer were exactly the same. This was checked using the method described in the previous chapter.

6.1 Algorithms efficiency

The purpose of the experiments is it to be able to measure the success of each of the algorithm and their efficiency. The results obtained from running the experiments allow us to determine the interval or limit at which one algorithm becomes faster than another.

So, how are the results of the experiments used to compare the algorithms efficiency?

Recall that a function $f(n)$ is said to be $O(g(n))$ if and only if there exists constants n_0 and c_0 such that for all $n > n_0 f(n) < n_0 g(n)$. The function $f(n)$ in our case is the time taken for the algorithm to perform the multiplication one or a given number of times, and $g(n)$ is the theoretical 'big oh' performance of the algorithm. For each of our multiplication algorithms we will look at the value $f(n) / g(n)$. As n approaches infinity this ratio will be checked if it approaches a value c_0. If it does approach this ratio then the implementation is said to be behaving close to its theoretical performance.

Recall as well that the theoretical performance of the algorithms is the following:

- Traditional: $O(n^2)$
- Karatsuba: $O(n^{log(3)})$
- Fast Fourier Transform: $O(n \log n)$

6.1.1 Traditional Algorithm

In this section, the results for the Traditional Algorithm are presented. The table below is a list of the Traditional algorithm time performance in milliseconds, the lengths of the integers, and the number of times the multiplication has been repeated for. Graphs are then produced with multiplicands length against the time took to multiply them.

Table 6.1.1.1 Time performance of the Traditional algorithm

Repeats	Integer Length	Operation Time (ms)	Total Time (ms)
10000	10	0.013	130
10000	20	0.041	410
10000	30	0.089	890
10000	40	0.164	1640

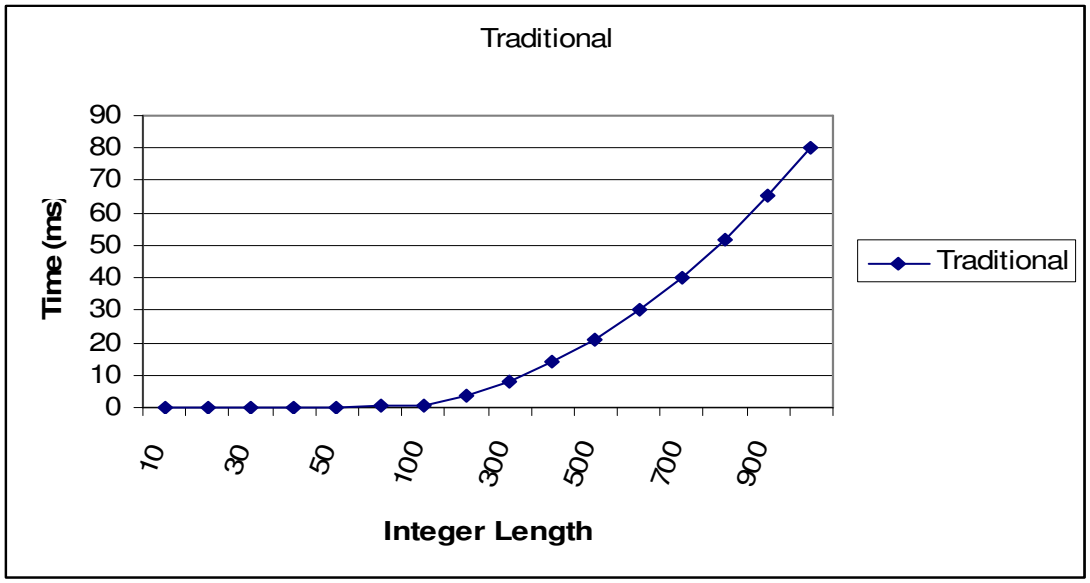
10000	50	0.241	2410
10000	60	0.345	3450
1000	100	0.89	890
1000	200	3.54	3540
1000	300	7.9	7900
1000	400	13.9	13900
1000	500	21.11	21110
1000	600	29.91	29910
1000	700	40.24	40240
1000	800	52.02	52020
1000	900	65.31	65310
1000	1000	80.25	80250

From these results it is easy to see that if the length of the multiplicands is increased by a factor of 2, then the algorithm takes about 4 times to complete the multiplication.

For integer length 20 the multiplication took 0.041 ms to complete and for 40 digits it took 0.164 ms to complete.

This was expected as the theoretical efficiency of the Traditional algorithm is quadratic (n^2) and $2^2 = 4$. See the corresponding graph shown below.

Graph 6.1.1.1 TimePerformance of the Traditional Algorithm



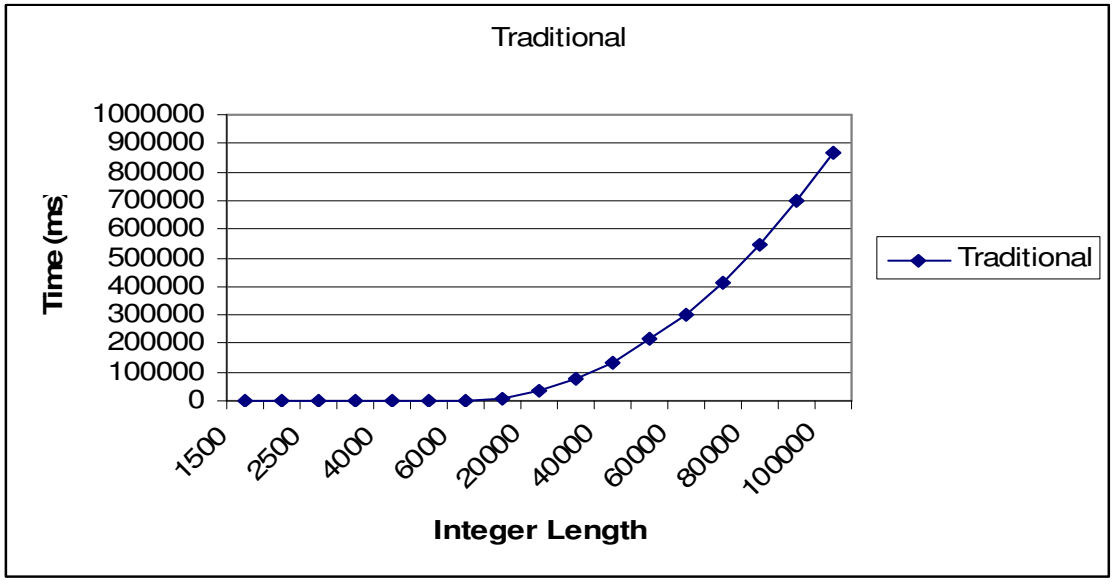
The message 'CPU LIMIT EXCEEDED' appeared on the screen when carrying out the multiplication on multiplicands of length bigger than 100,000 digits. This proves that the Traditional algorithm runs with no errors until the system runs out of memory.

The observation made on the previous table still remains true in the following table.

Table 6.1.1.2 Time performance of the Traditional algorithm

Repeats	Integer Length	Operation Time (ms)	Total Time (ms)
500	1500	182.7	91350
500	2000	323	161500
500	2500	501.1	250550
100	3000	722.8	72280
100	4000	1288.5	128850
100	5000	2029.4	202940
100	6000	2886.6	288660
100	10000	8160.2	816020
10	20000	33360	333600
10	30000	75322	753220
10	40000	134012	1340120
10	50000	219926	2199260
10	60000	301749	3017490
10	70000	410413	4104130
10	80000	548795	5487950
10	90000	699913	6999130
10	100000	866745	8667450
10	110000	CPU Limit Exceeded	

Graph 6.1.1.2 TimePerformance of the Traditional Algorithm

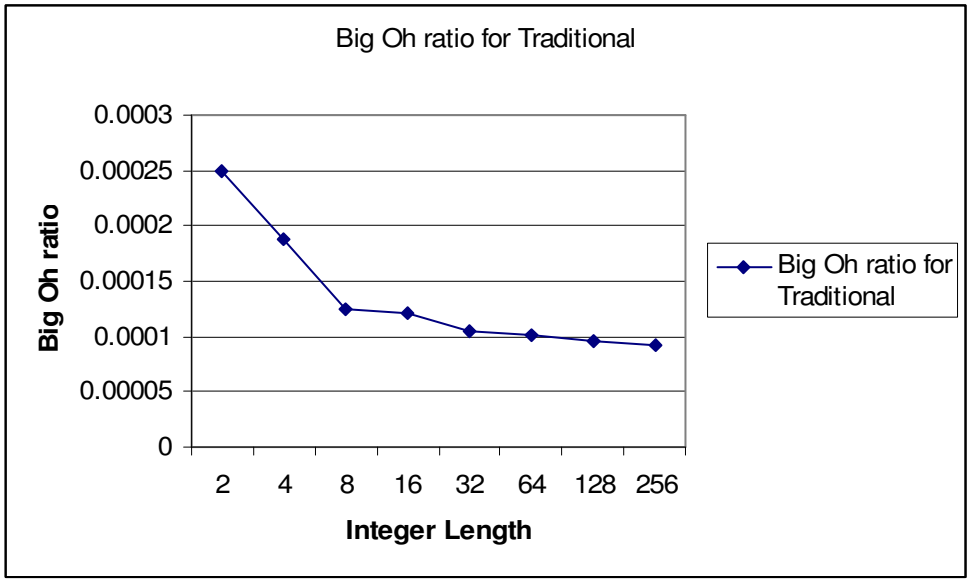


In order to determine whether the traditional algorithm performs close to its theoretical expectation or not, the traditional algorithm was run 10,000 times on multiplicands of length of up to the eighth power of two (256) and the results are shown in the table below. Then, the average time for each power of two is divided by n^2 (where n is the length of the multiplicands). If this ratio (time / n^2) gets smaller and smaller when increasing the length of the multiplicands then it is said that the Traditional behaves close to its theoretical efficiency.

Table 6.1.1.3 Performance of Traditional

Multiplicands length	Operation time (ms) Traditional	n^2	Big *O* ratio or (time / n^2)
2	0.001	4	0.00025000
4	0.003	16	0.00018750
8	0.008	64	0.00012550
16	0.031	256	0.00012109
32	0.108	1024	0.00010546
64	0.412	4096	0.00010058
128	1.566	16384	0.00009558
256	6.045	65536	0.00009223

Graph 6.1.1.3 'Big *O*' ratio for Traditional



These results show that the 'Big *O* ratio' starts of at 0.00025 and gradually decreases to 0.00009223 at integer length 256. It is predicted to keep decreasing for greater powers of two than 256. It will eventually reach a constant c_0. Therefore, the implementation of this algorithm is considered successful as it is so close to its theoretical expectation.

6.1.2 Karatsuba Algorithm

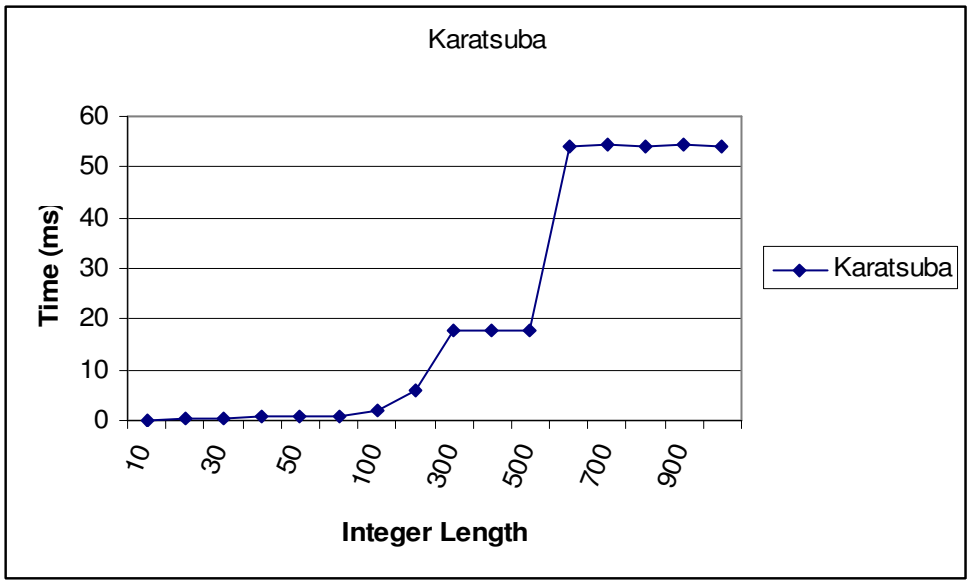
The tables below and the corresponding graphs show how the Karatsuba algorithm behaved through out experiments. They are then analysed and the implementation efficiency is determined.

Table 6.2.1.1 Time performance of the Karatsuba Algorithm

Repeats	Integer Length	Operation Time (ms) Karatsuba	Total Time (ms)
10000	10	0.071	710
10000	20	0.224	2240

10000	30	0.23	2300
10000	40	0.68	6800
10000	50	0.686	6860
10000	60	0.668	6680
1000	100	1.99	1990
1000	200	5.97	5970
1000	300	17.94	17940
1000	400	17.94	17940
1000	500	17.93	17930
1000	600	54.26	54260
1000	700	54.36	54360
1000	800	54.13	54130
1000	900	54.4	54400
1000	1000	54.24	54240

Graph 6.2.1.1 TimePerformance of the Karatsuba Algorithm



The first obvious observation to make is that the time to carry out Karatsuba multiplication increases in steps. This was expected, as the implementation of the Karatsuba algorithm pads out the length of the multiplicands to the next power of two. So, multiplying two integers of length 400 digits long will take almost the same amount of time taken to multiply two 512 digits integers.

The table below shows kratasuba's implantation tested on much large integers.

Table 6.2.1.2 TimePerformance of the Karatsuba Algorithm

Repeats	Integer Length	Operation Time (ms)	Total Time (ms)
500	1500	163.8	81900
500	2000	163.98	81990
500	2500	492.2	246100
100	3000	492.2	49220
100	4000	491.4	49140

100	5000	1481	148100
100	6000	1479.4	147940
100	10000	4487	448700
10	20000	13210	132100
10	30000	13227	132270
10	40000	39358	393580
10	50000	40808	408080
10	60000	39326	393260
10	70000	118302	1183020
10	80000	119848	1198480
10	90000	120747	1207470
10	100000	120768	1207680
10	110000	120326	1203260
10	120000	120537	1205370
10	130000	120814	1208140

From the results shown in the previous tables, it is easy to see that if the length of the multiplicands is increased by a factor of 2, then the algorithm takes about 3 times to complete the multiplication.

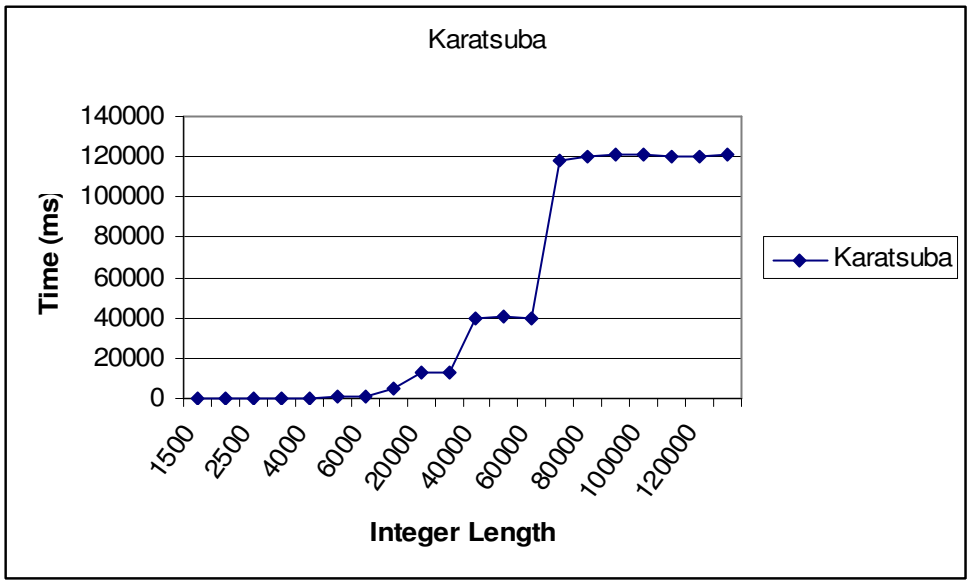
For integer length 10,000 the multiplication took 4.487 seconds to complete and for 20,000 digits it took 13.210 seconds to complete.

This was expected as the theoretical efficiency of the Karatsuba algorithm is $O(n^{1.585})$ and $2^{1.585} \approx 2.99 \approx 3$.

It is also noticed that the Karatsuba algorithm shows no sign of having an upper limit on the length of the multiplicands. It is assumed to keep running until the system runs out of memory. Due to the large amount of time it takes to finish, the multiplication increases as the length increases. The developer decided to stop at the stage where the multiplicands are of length 130,000 digits each. There is no point in continuing as this project mainly focuses on comparing the algorithms and not how big the multiplicands it can multiply.

The corresponding graph to the previous table is shown below.

Graph 6.2.1.2 TimePerformance of the Karatsuba Algorithm

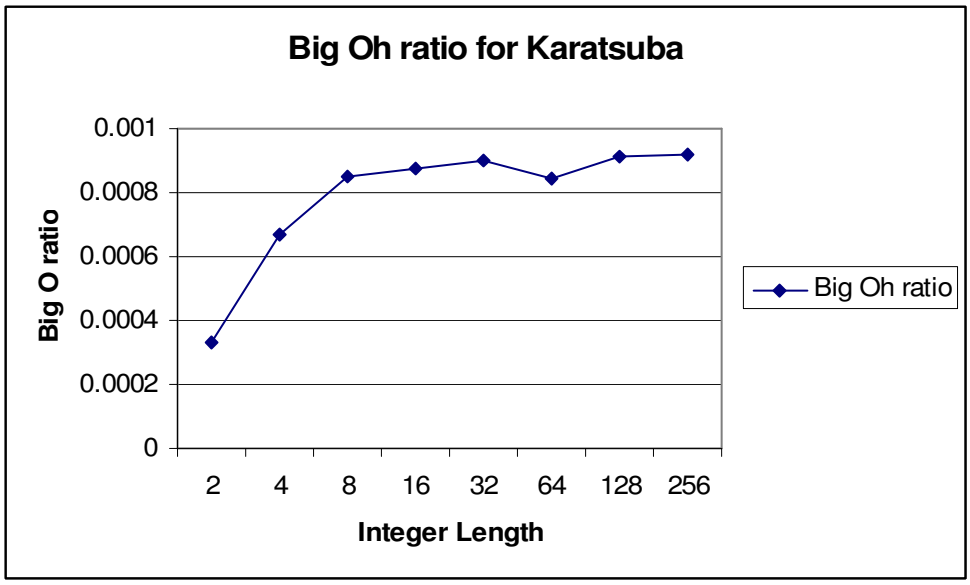


In order to determine whether the Karatsuba algorithm performs close to its theoretical expectation or not, the Karatsuba algorithm was run 10,000 times on multiplicands of length of up to the eighth power of two (256) and the results are shown in the table below. Then, the average time for each power of two is divided by $n^{1.585}$ (where n is the length of the multiplicands). If this ratio (time / $n^{1.585}$) gets smaller and smaller when increasing the length of the multiplicands then it is said that Karatsuba behaves close to its theoretical efficiency.

Table 6.2.1.3 Performance of Karatsuba

Multiplicands length	Operation time (ms) Karatsuba	$n^{1.585}$	Big *O* ratio or (time / n^{log3})
2	0.001	3	0.0003333
4	0.006	9	0.0006666
8	0.023	27.002	0.0008517
16	0.071	81.008	0.0008764
32	0.218	243.03	0.0008970
64	0.668	792.11	0.0008433
128	1.993	2187.39	0.0009111
256	6.015	6562.36	0.0009166

Graph 6.2.1.3 Performance of Karatsuba



These results show that the 'Big *O* ratio' starts of at about 0.0006666 and gradually increases to 0.0008970 at integer length 32, then it drops to 0.0008433 at integer length of 64 digits. It then climbs to 0.0009166 ms at length 256 digits. It is predicted that there will be a point where it starts decreasing probably at integer length of 1024 digits long onwards. It will eventually reach a constant c_0. Therefore, the implementation of this algorithm is considered successful for big multiplicands of large lengths.

The values of the 'big Oh ratio' indicate that it is slower than predicted. There are a number of reasons for why this is happening. The main two reasons are:

- Allocating 5 spaces for the different arrays by calling calloc() at each recursive stage.
- Freeing the allocated 5 spaces by calling free() at each recursive stage.

These two factors really slow the execution down. However, when the multiplicands are of size 1024 and greater the time to allocate the array spaces compared to the time to perform the multiplication is negligible. Therefore, Karatsuba implementation is said to be slow for small inputs and performs close to its theoretical efficiency for multiplicands of large lengths.

6.1.3 Fast Fourier Trasform

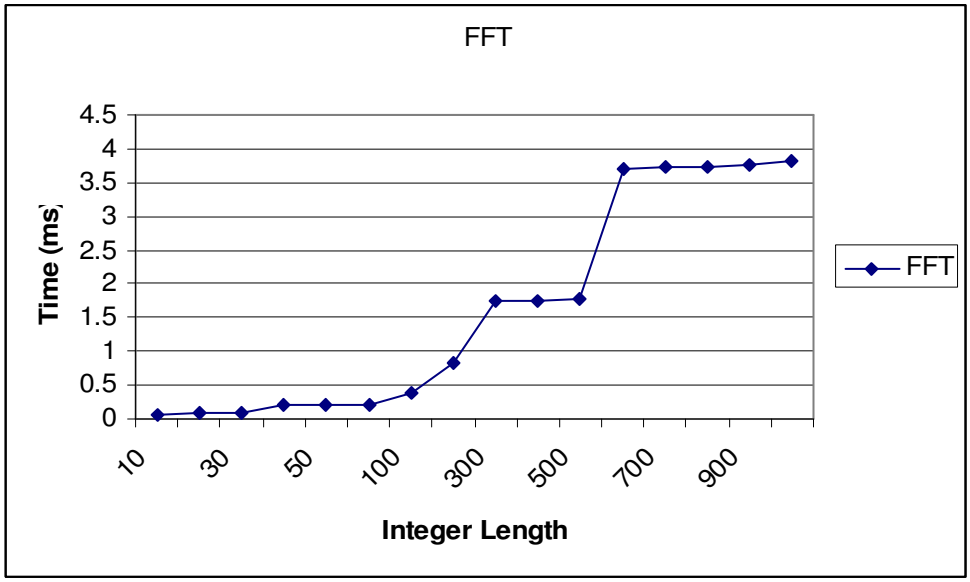
The tables below and the corresponding graphs show how the Fast Fourier Transform algorithm behaved through out experiments. They are then analysed and the implementation efficiency is determined.

Table 6.1.3.1 TimePerformance of the Fast Fourier Transform Algorithm

Repeats	Integer Length	Operation Time (ms)	Total Time (ms)
10000	10	0.046	460
10000	20	0.094	940

10000	30	0.094	940
10000	40	0.202	2020
10000	50	0.199	1990
10000	60	0.193	1930
1000	100	0.38	380
1000	200	0.84	840
1000	300	1.75	1750
1000	400	1.76	1760
1000	500	1.77	1770
1000	600	3.7	3700
1000	700	3.72	3720
1000	800	3.73	3730
1000	900	3.75	3750
500	1000	3.82	1910

Graph 6.1.3.1 TimePerformance of the Fast Fourier Transform Algorithm



From the results shown in the previous tables, it is easy to see that if the length of the multiplicands is increased by a factor of 2, then the algorithm takes about 2 times to complete the multiplication.

For integers of length 10 the multiplication took 0.046 milliseconds to complete and for 20 digits it took 0.094 milliseconds to complete.

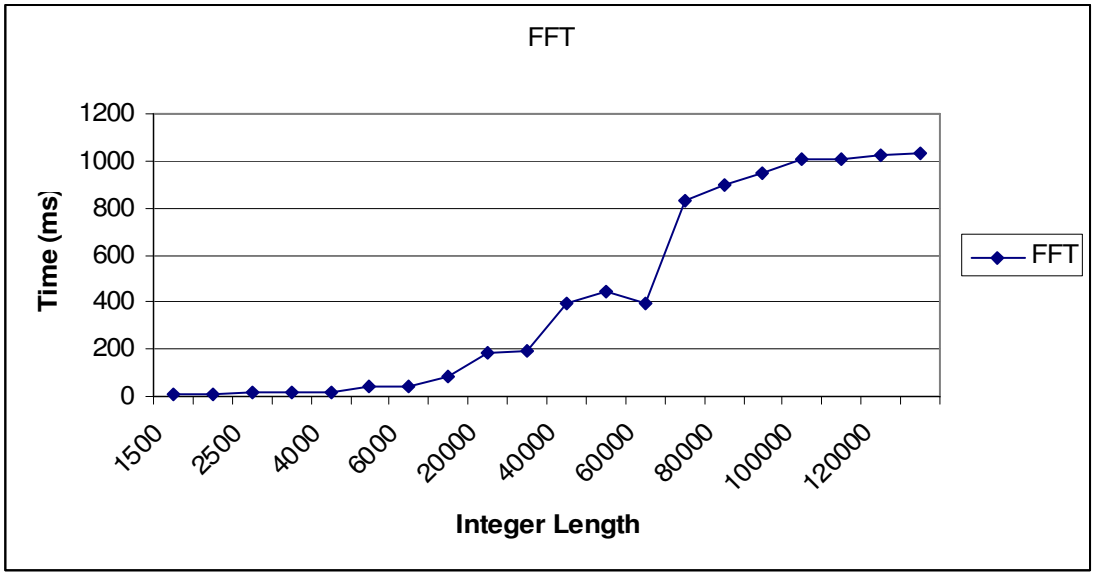
This is to be expected as the theoretical efficiency of the Fourier algorithm is $O(n \log n)$ and $2 \log 2 = 2$.

It is also noticed that as Karatsuba the Fourier algorithm shows no sign of having an upper limit on the length of the multiplicands. It is predicted to keep running until the system runs out of memory. The developer decided to stop at the stage where the multiplicands are of 130,000 digits each. As outlined previously, it is not feasible to keep testing larger and larger multiplicands as this project mainly focuses on comparing the algorithms and determining their efficiencies.

Table 6.1.3.2 TimePerformance of the Fast Fourier Transform Algorithm

Repeats	Integer Length	Operation Time (ms)	Total Time (ms)
500	1500	8.04	4020
500	2000	8.08	4040
500	2500	17.38	8690
100	3000	17.3	1730
100	4000	17.9	1790
100	5000	41.2	4120
100	6000	41.8	4180
100	10000	87.9	8790
10	20000	187	1870
10	30000	189	1890
10	40000	397	3970
10	50000	448	4480
10	60000	395	3950
10	70000	828	8280
10	80000	902	9020
10	90000	945	9450
10	100000	1004	10040
10	110000	1011	10110
10	120000	1023	10230
10	130000	1035	10350

Graph 6.1.3.2 TimePerformance of the Fast Fourier Transform Algorithm



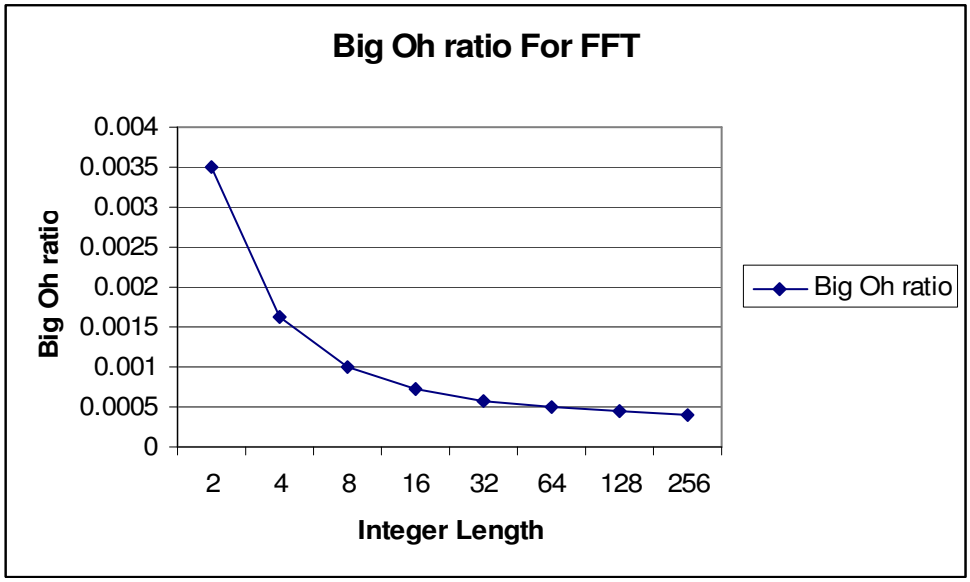
In order to determine whether the Fourier algorithm performs close to its theoretical expectation or not, the Fourier algorithm is run 10,000 times on multiplicands of length of up to the eighth power of two (256) and the results are shown in the table below. Then, the average time for each power of two is divided by *n log n* (where *n* is

the length of the multiplicands). If this ratio (time / n log n) gets smaller and smaller when increasing the length of the multiplicands then the Fourier implantation is said to be behaving close to its theoretical efficiency.

Table 6.1.3.3 Performance of FFT

Multiplicands length	Operation time (ms) FFT	n log n	Big *O* ratio or (time / n log n)
2	0.007	2	0.00350
4	0.013	8	0.001625
8	0.024	24	0.001000
16	0.047	64	0.000734
32	0.092	160	0.000575
64	0.191	384	0.000497
128	0.398	896	0.000444
256	0.841	2048	0.000410

Graph 6.1.3.3 Performance of FFT



These results show that the 'Big *O* ratio' starts of at 0.00350 and gradually decreases to 0.000410 at integers of length 256. It is predicted to keep decreasing for greater powers of two than 256. It will eventually reach a constant c_0. Therefore, the implementation of this algorithm is considered successful as it is so close to its theoretical expectation.

6.2 Comparison of Algorithms

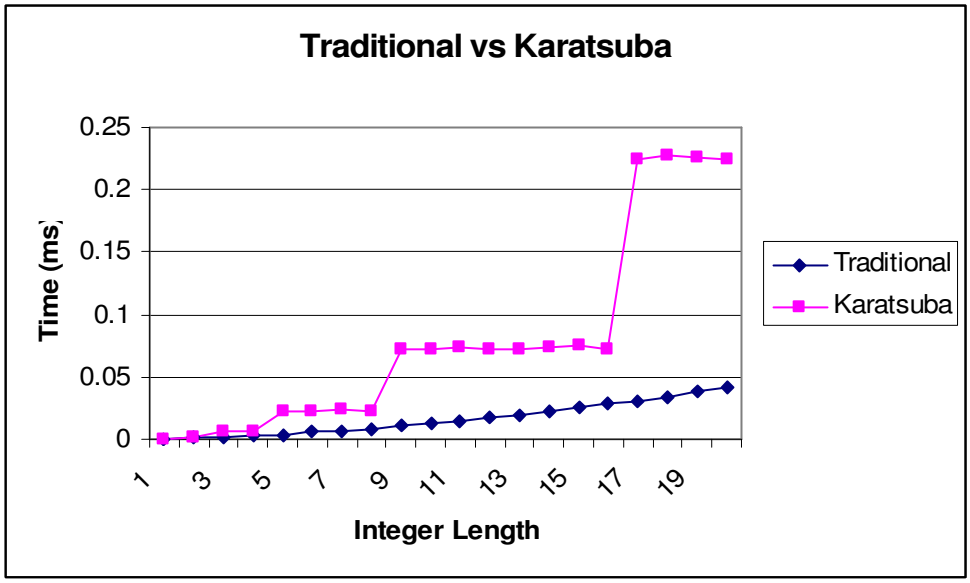
In this section, Traditional, Karatsuba and Fast Fourier Transform algorithms are run together on the same sets of inputs of varying lengths. The purpose of the following experiments is to determine the range of competence of the different algorithms and to show at what range they beat each other. The point where they intersect will be determined.

In the table below, the different multiplication technique have been tested on inputs of length from 1 to 20.

Table 6.2.1 Comparison between Traditional, Karatsuba and FFT

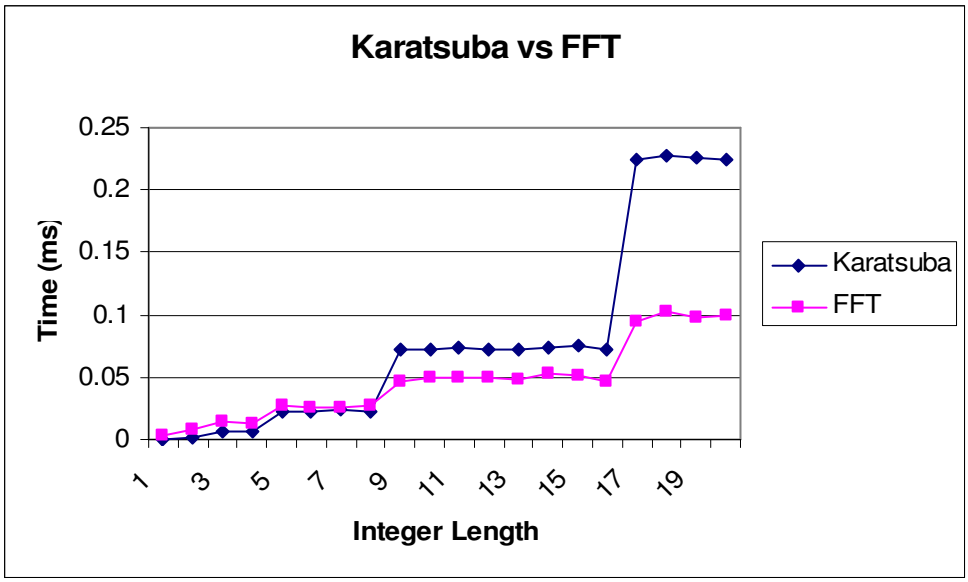
Repeats	Largest length	operation_time (ms) Traditional	operation_time (ms) Karatsuba	operation_time (ms) FFT
10000	1	0	0	0.004
10000	2	0.001	0.001	0.008
10000	3	0.002	0.007	0.014
10000	4	0.003	0.006	0.013
10000	5	0.004	0.023	0.027
10000	6	0.006	0.022	0.025
10000	7	0.007	0.024	0.025
10000	8	0.008	0.023	0.027
10000	9	0.011	0.072	0.047
10000	10	0.013	0.072	0.049
10000	11	0.014	0.073	0.049
10000	12	0.017	0.072	0.05
10000	13	0.02	0.072	0.048
10000	14	0.022	0.074	0.053
10000	15	0.026	0.075	0.051
10000	16	0.029	0.072	0.047
10000	17	0.031	0.224	0.095
10000	18	0.033	0.227	0.103
10000	19	0.039	0.226	0.097
10000	20	0.042	0.225	0.1

Graph 6. 2.1 Comparison between Traditional and karatsuba



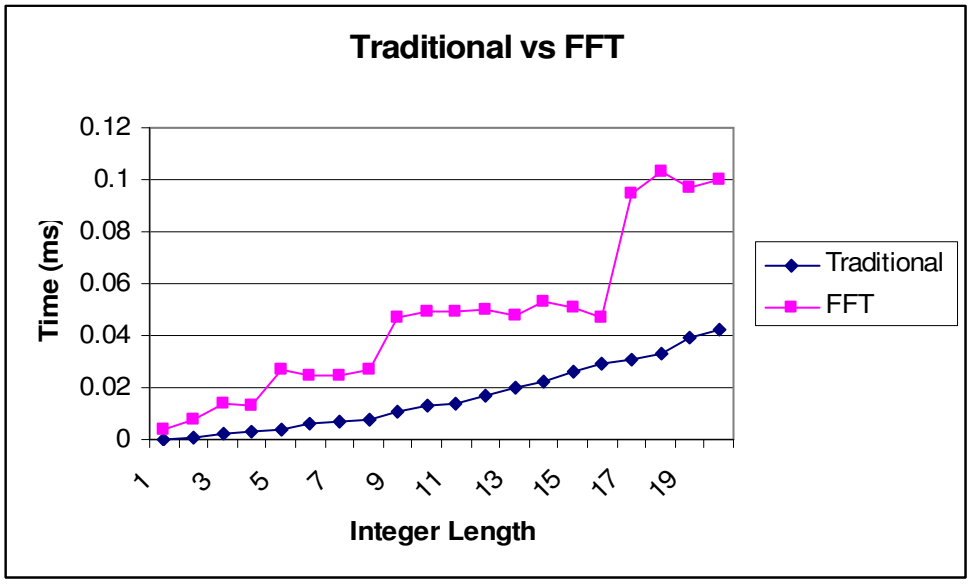
In the graph above it clear that Traditional is always faster than Karatsuba.

Graph 6.2.2 Comparison between Traditional and FFT



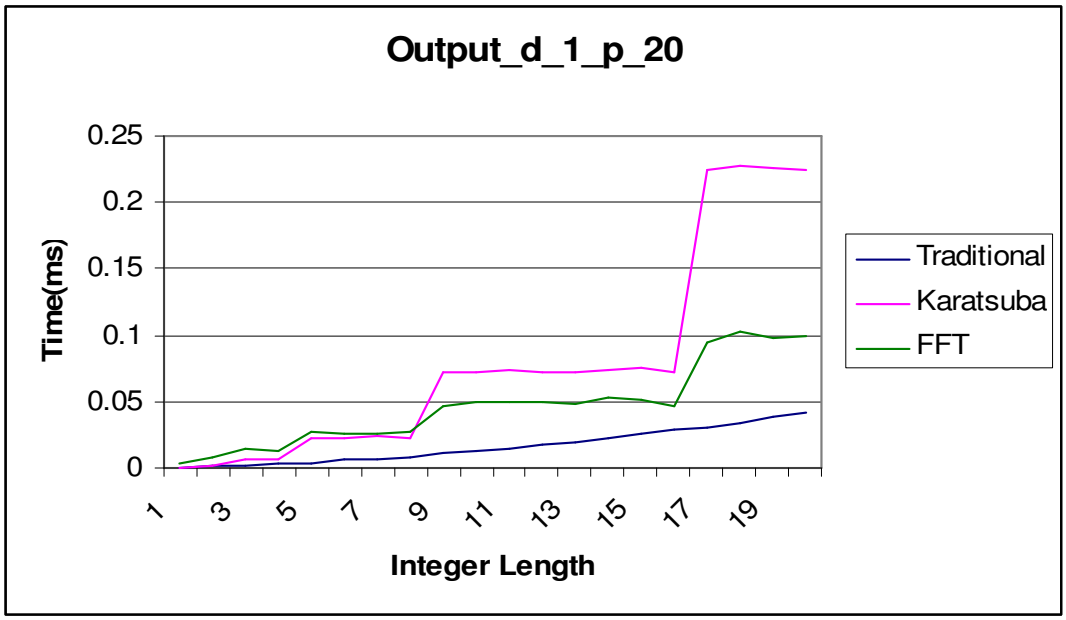
In the previous graph, it is clear to see that FFT becomes faster than Karatsuba when the multiplicands are of length equal to 9. This is to be expected as explained previously the implementation of Karatsuba's algorithm has shown to be slow for small inputs.

Graph 6.2.3 Comparison between Traditional and FFT



The graph above show that Traditional is much faster than Fourier.

Graph 6.2.4 Test results of the different algorithms plotted on the same graph

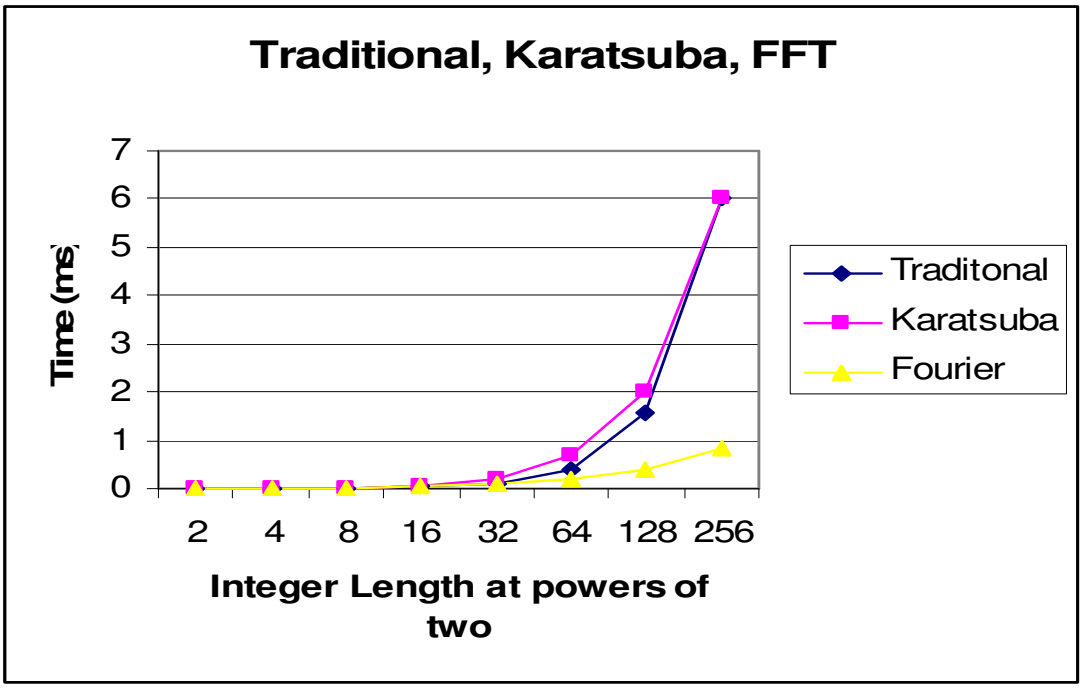


As we discussed previously the performance of each algorithm separately on the first eight powers of 2, we will now compare their performance. The table below shows the test results obtained by running the algorithms 10,000 times on the first eight powers of two. Then, it is followed by the corresponding graph.

Table 6.2.5 Test results of the different algorithms plotted on the same graph

Repeats	Multiplicands Length	Operation time (ms) Traditional	Operation time (ms) Karatsuba	Operation time (ms) FFT
10000	2	0.001	0.001	0.007
10000	4	0.003	0.006	0.013
10000	8	0.008	0.023	0.024
10000	16	0.031	0.071	0.047
10000	32	0.108	0.218	0.092
10000	64	0.412	0.668	0.191
10000	128	1.566	1.993	0.398
10000	256	6.045	6.015	0.841

Graph 6.2.5 Test results of the different algorithms plotted on the same graph



From the previous table and the graph presented above, it is clear to notice that the Fourier becomes faster than Traditional at a much early stage. At input length of 32 digits to perform Traditional multiplication it took 0.108 ms and Fourier multiplication took 0.092 ms.

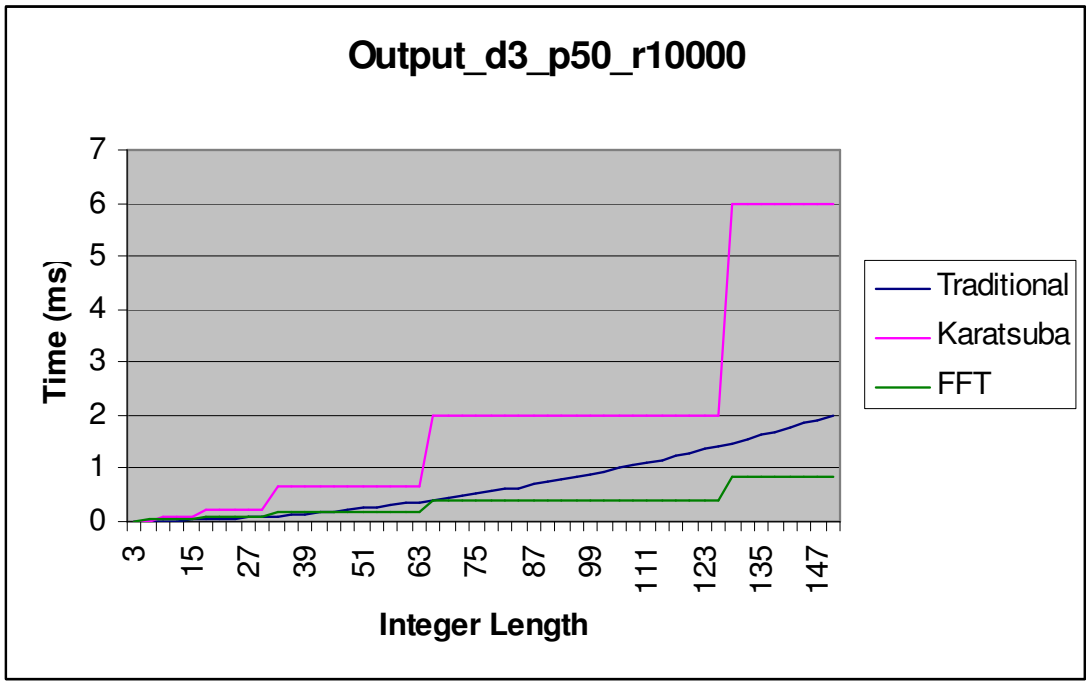
The following experiments consisted on running experiment on inputs of even and odd lengths up to 150 digits long. The results are shown in the following table and graph.

Table 6.2.6 Experiments with odd and even inputs

Repeats	Input Length	Operation time (ms) Traditional	Operation Time (ms) Karatsuba	Operation Time (ms) FFT
10000	3	0.002	0.007	0.013
10000	6	0.006	0.022	0.023
10000	9	0.011	0.071	0.045
10000	12	0.017	0.071	0.046
10000	15	0.025	0.071	0.046
10000	18	0.034	0.216	0.09
10000	21	0.045	0.217	0.09
10000	24	0.059	0.217	0.091
10000	27	0.073	0.217	0.091
10000	30	0.089	0.216	0.091
10000	33	0.106	0.662	0.184
10000	36	0.125	0.66	0.185
10000	39	0.146	0.661	0.187
10000	42	0.168	0.663	0.186
10000	45	0.193	0.662	0.187
10000	48	0.216	0.66	0.188

10000	51	0.244	0.662	0.187
10000	54	0.272	0.661	0.189
10000	57	0.309	0.662	0.188
10000	60	0.345	0.661	0.189
10000	63	0.367	0.663	0.189
10000	66	0.4	1.981	0.388
10000	69	0.439	1.984	0.387
10000	72	0.475	1.981	0.389
10000	75	0.514	1.98	0.388
10000	78	0.555	1.983	0.389
10000	81	0.599	1.982	0.389
10000	84	0.64	1.978	0.389
10000	87	0.688	1.982	0.392
10000	90	0.736	1.981	0.393
10000	93	0.782	1.983	0.392
10000	96	0.835	1.983	0.393
10000	99	0.884	1.982	0.393
10000	102	0.945	1.982	0.394
10000	105	1.005	1.982	0.394
10000	108	1.051	1.984	0.394
10000	111	1.112	1.987	0.395
10000	114	1.169	1.976	0.396
10000	117	1.227	1.979	0.395
10000	120	1.293	1.982	0.396
10000	123	1.357	1.982	0.397
10000	126	1.418	1.986	0.396
10000	129	1.483	5.991	0.828
10000	132	1.559	5.986	0.827
10000	135	1.629	5.977	0.824
10000	138	1.698	5.993	0.826
10000	141	1.778	5.966	0.825
10000	144	1.851	5.967	0.827
10000	147	1.926	5.967	0.826
10000	150	2.003	5.967	0.828

Graph 6.2.6 Experiments with odd and even inputs



By analysing the table and graph shown above, Fourier becomes faster than Karatsuba when the input is of 9 digits long. However, Fourier does not become faster than the traditional until the multiplicands are of length of about 45 digits and remains faster than the Traditional.

The time it takes to carry out Karatsuba and Fourier multiplications increases in chunks or steps. This was expected, as their implementations pads out the length of the multiplicands to the next power of two. So, multiplying two integers of length 87 digits long will take almost the same amount of time taken to multiply two 128 digits integers.

Previously, we mentioned that Fourier is faster than Traditional at input of size 32 digits long. However, at input of size 33 digits long, Traditional is faster than Fourier (hence the input has been padded to the next power of two which is 64).

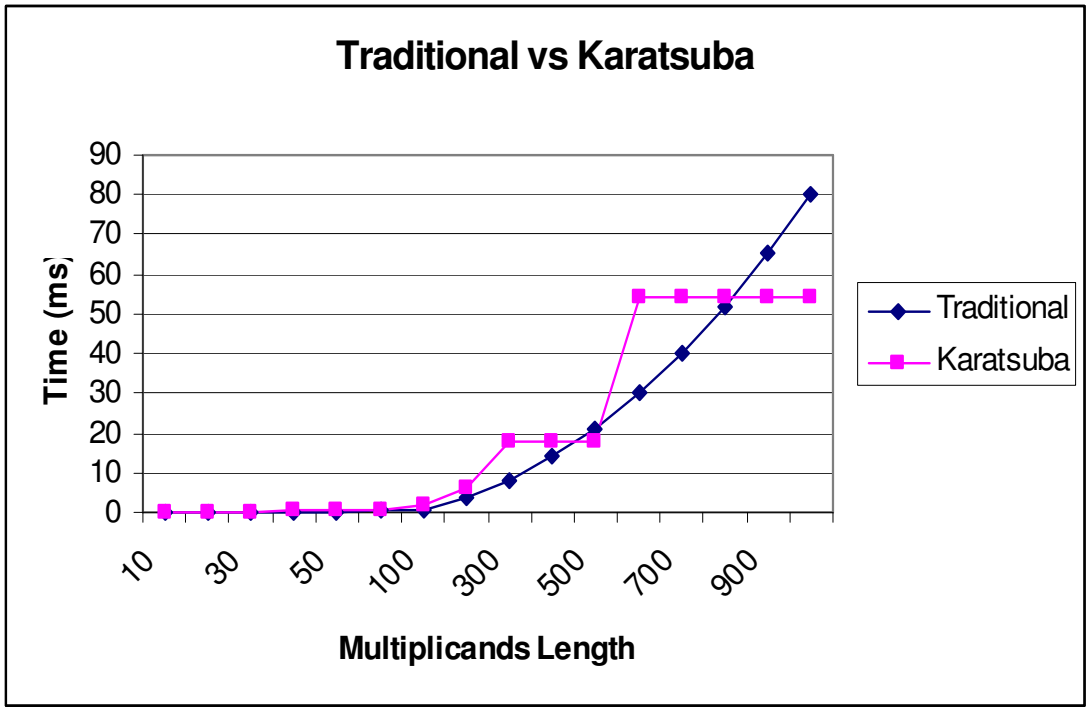
The following experiments consist on testing the algorithms at large size inputs of up 1000 digits.

Table 6.2.7 Experiments with input size up to 1000

Repeats	**Input Length**	**Operation time (ms) Traditional**	**Operation Time (ms) Karatsuba**	**Operation Time (ms) FFT**
10000	10	0.013	0.071	0.046
10000	20	0.041	0.224	0.094
10000	30	0.089	0.23	0.094
10000	40	0.164	0.68	0.202
10000	50	0.241	0.686	0.199
10000	60	0.345	0.668	0.193
1000	100	0.89	1.99	0.38

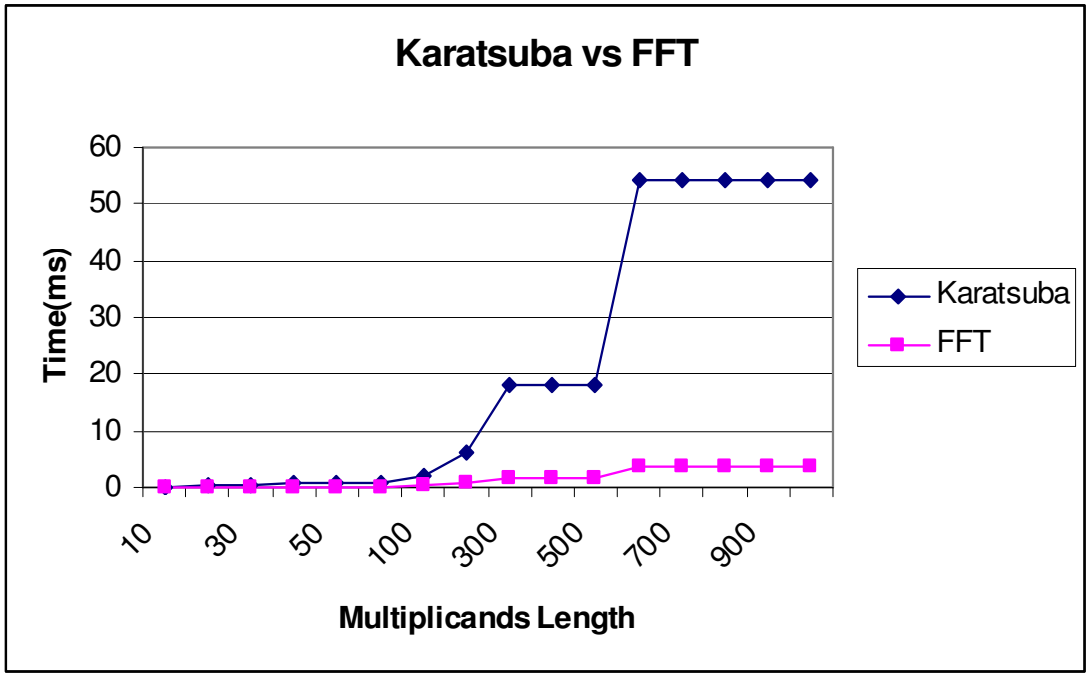
1000	200	3.54	5.97	0.84
1000	300	7.9	17.94	1.75
1000	400	13.9	17.94	1.76
1000	500	21.11	17.93	1.77
1000	600	29.91	54.26	3.7
1000	700	40.24	54.36	3.72
1000	800	52.02	54.13	3.73
1000	900	65.31	54.4	3.75
500	1000	80.25	54.24	3.82

Graph 6.2.7 Performance of Traditional and Karatsuba on large inputs



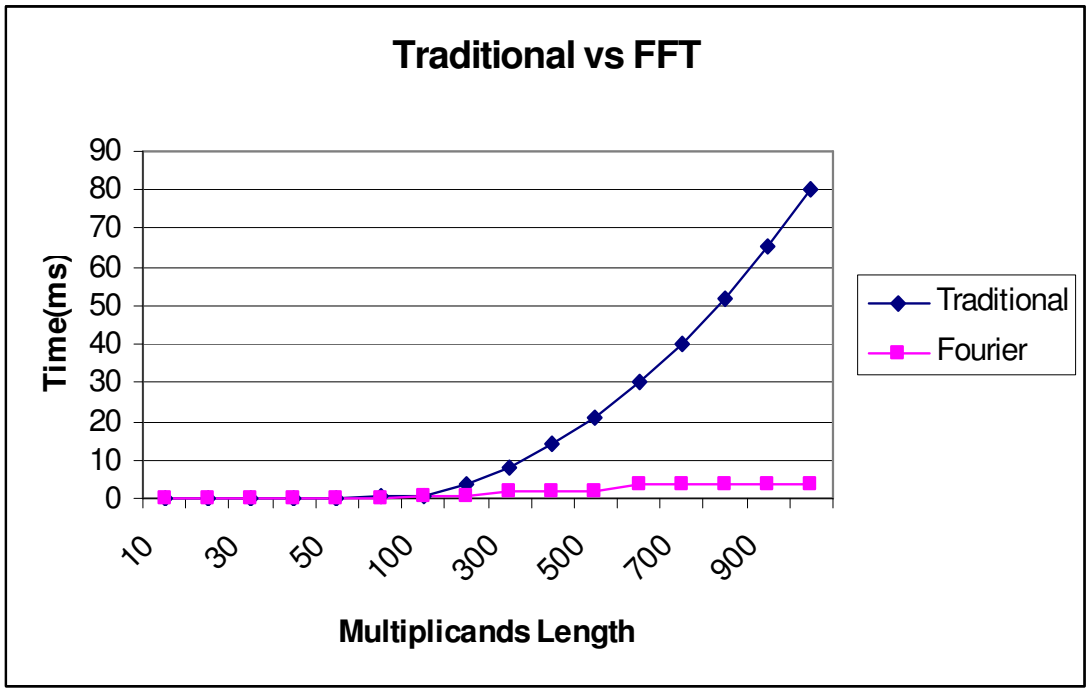
This above graph show Karatsuba does indeed performs a faster than Traditional for large inputs. This shows that our prediction discussed previously is correct. The time spent allocating memory is negligible compared to the time spent performing Karatsuba multiplication for large inputs. The table and graph show that Karatsuba is faster than Traditional at inputs of about 900 digits long.

Graph 6.2.8 Performance of Karatsuba and FFT on large inputs



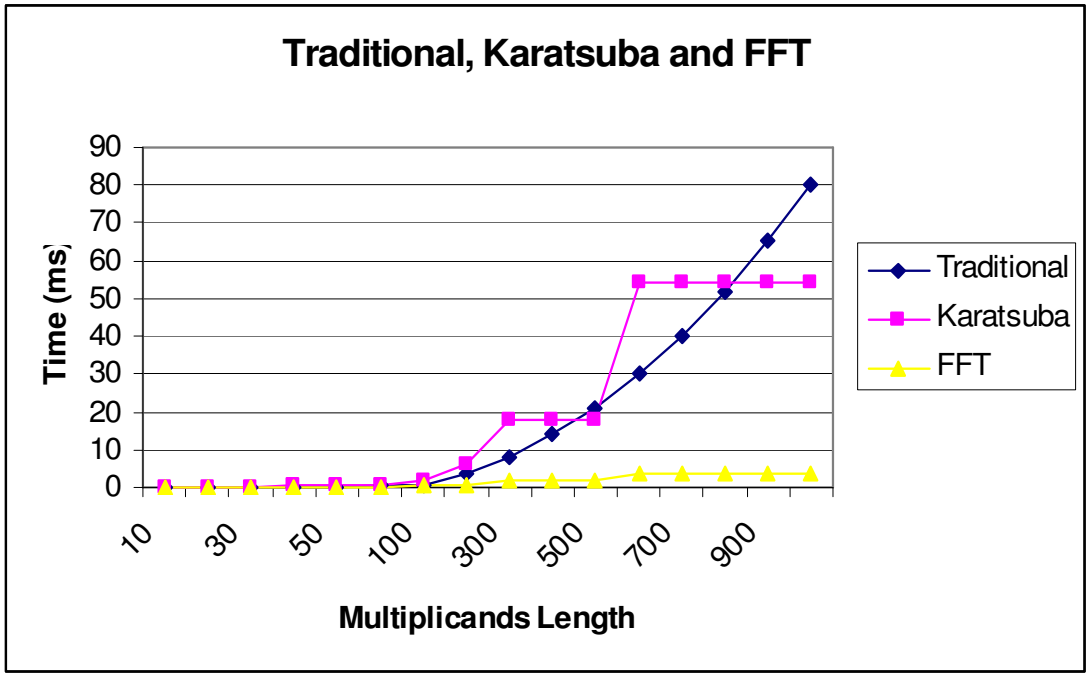
The graph above shows that Fourier is much faster than Karatsuba on large inputs. This shows that the performance of the Fourier algorithm is much better than the performance of the Karatsuba algorithm on large inputs.

Graph 6.2.9 Performance of Traditional and FFT on large inputs



The graph above shows that Fourier is much faster than Traditional on large inputs. This shows that the performance of the Fourier algorithm is much better than the performance of the Traditional algorithm on large inputs.

Graph 6.2.10 Performance of Traditional, Karatsuba and FFT on large inputs

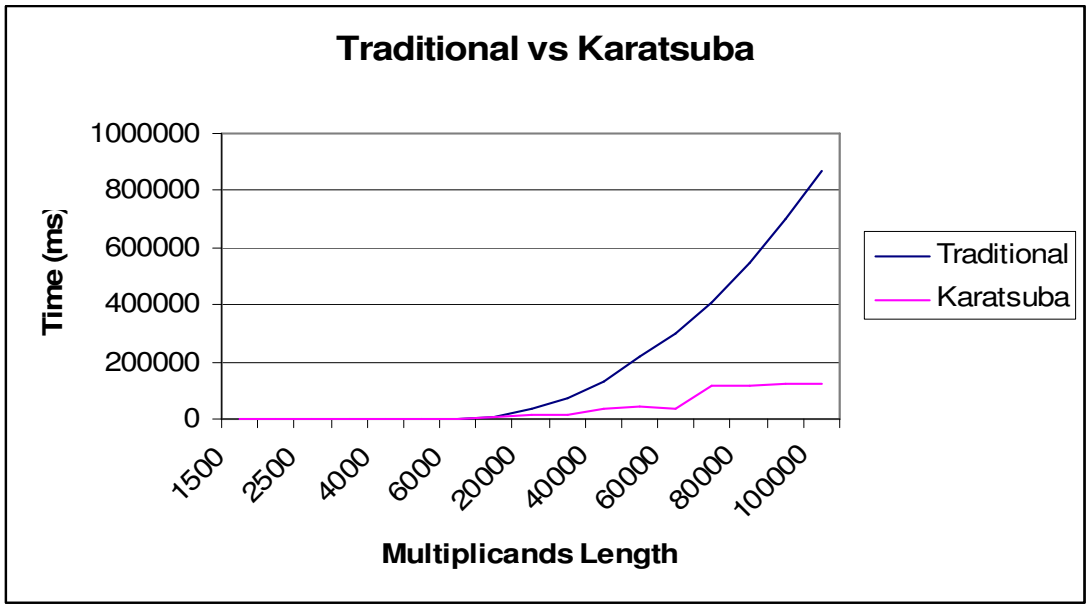


The following experiments consist on testing the algorithms at very large size inputs of up 130,000 digits.

Table 6.2.2 Performance of Traditional and Karatsuba on very large inputs

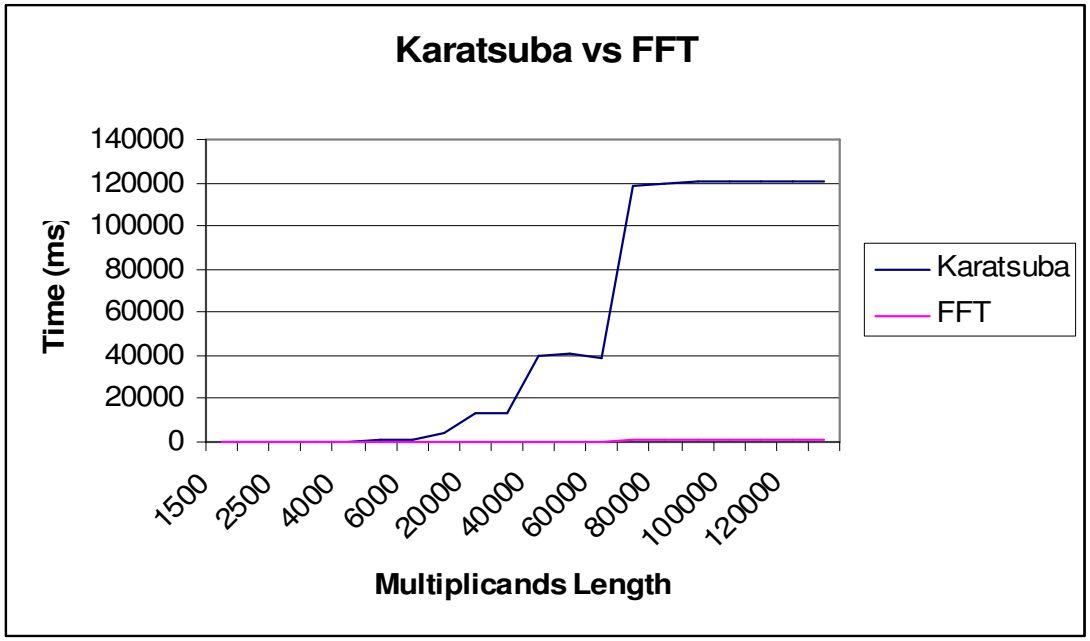
Repeats	Input Length	Operation time (ms) Traditional	Operation Time (ms) Karatsuba	Operation Time (ms) FFT
500	1500	182.7	163.8	8.04
500	2000	323	163.98	8.08
500	2500	501.1	492.2	17.38
100	3000	722.8	492.2	17.3
100	4000	1288.5	491.4	17.9
100	5000	2029.4	1481	41.2
100	6000	2886.6	1479.4	41.8
100	10000	8160.2	4487	87.9
10	20000	33360	13210	187
10	30000	75322	13227	189
10	40000	134012	39358	397
10	50000	219926	40808	448
10	60000	301749	39326	395
10	70000	410413	118302	828
10	80000	548795	119848	902
10	90000	699913	120747	945
10	100000	866745	120768	1004
10	110000	CPU Limit Exceeded	120326	1011
10	120000	CPU Limit Exceeded	120537	1023
10	1300000	CPU Limit Exceeded	120814	1035

Graph 6.2.11 Performance of Traditional and Karatsuba on very large inputs



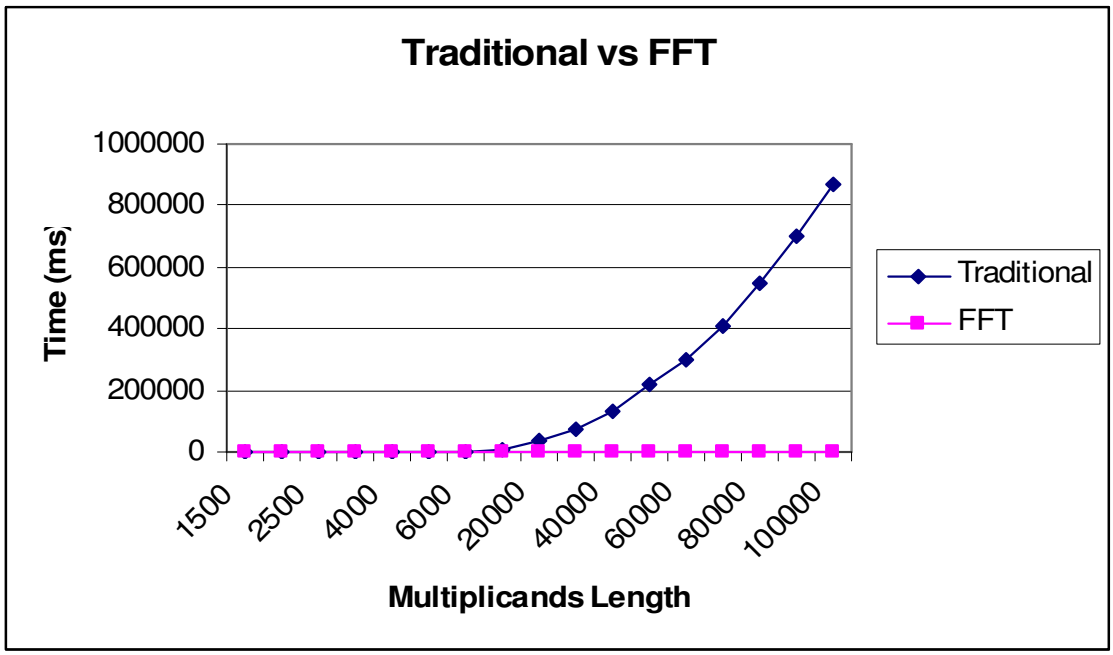
Karatsuba seems to perform a lot better than Traditional on very large inputs. At this stage it is clear that the Karatsuba technique when multiplying large integers is much efficient than Traditional technique.

Graph 6.2.12 Performance of Karatsuba and FFT on very large inputs



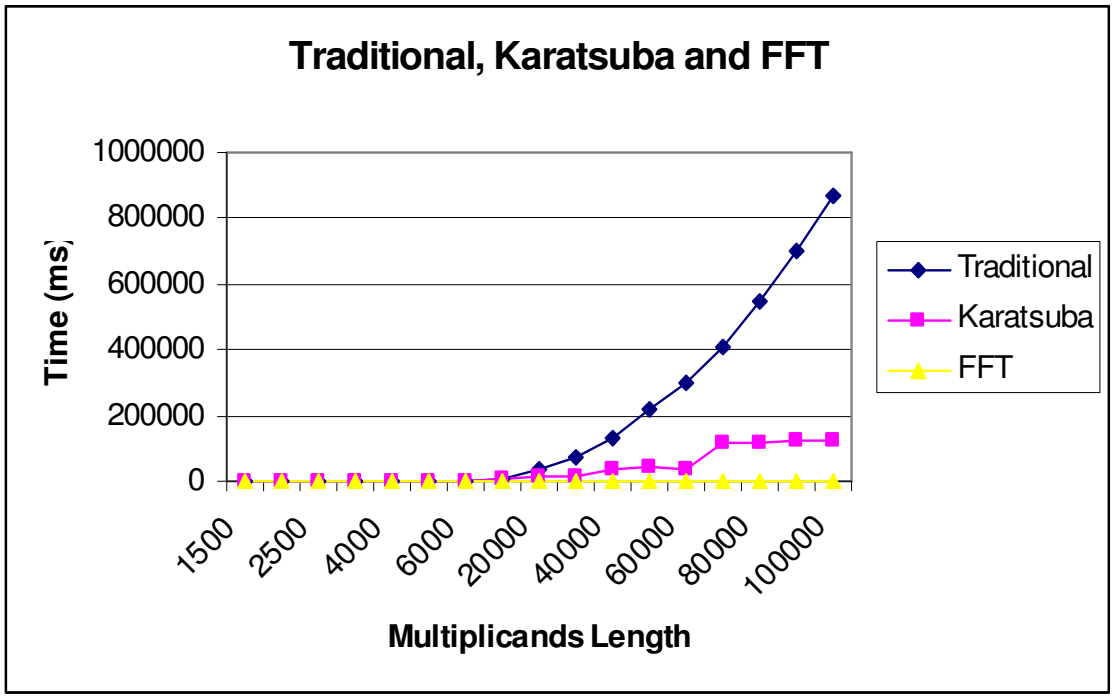
In the graph shown above, Fourier is so fast to the extent that it hardly rises off the X-axis.

Graph 6.2.13 Performance of Traditional and FFT on very large inputs



Again, Fourier is the fastest.

Graph 6.2.14 Performance of Traditional, Karatsuba and FFT on very large inputs



The graph shown above allows us to conclude that:
The Karatsuba technique used to multiply large integers is much more efficient than the Traditional technique. However, the Fast Fourier Technique is the most efficient.

6.3 Experiments Summary:

By the analysis of the results obtained from running the experiment and the behaviour of the various algorithms on different sets of inputs. The following conclusions are drawn:

- The implementations of the Traditional, Karatsuba and Fourier algorithms are close to their theoretical performance ($O(n^2)$, $O(n^{1.585})$ and $O(n \log n)$ respectively).
- Karatsuba is a lot slower on inputs of less than 800 digits long than predicted. The reasons for this happening where presented and explained.
- Fourier's implementation was a lot faster than predicted but sill follow its theoretical performance.
- Traditional performance was almost the same as the theoretical one.
- The timing was not as precise as expected. Running the algorithms many times of up to 10,000 times helped in getting close to the precise timing. However, this was not possible when dealing with inputs of very large sizes (i.e. 50,000 digits) and the algorithms were run only 10 times due to the large amount of time spent when multiplying integers of these sizes.
- The timing varied slightly on repeat runs as the BUCS *amos* machine might have been interrupted to run other processes at the same time as the experiments.
- All the algorithms can multiply very large inputs and give accurate results until the system runs out of memory. This confirms that the program is robust.
- The 'cross over' points were determined. However, these points were not as expected. Traditional was faster than Karatsuba and Fourier on inputs of up to 9 digits long. Then, Fourier became faster than Traditional and Karatsuba and Karatsuba did not become faster than Traditional until we reached inputs of about 850 digits long.

7 Conclusions / Critical Evaluation

7.1 Were the goals of this project fulfilled?

The main goals of this project were to implement three different algorithms close to their theoretical performance, find out the 'cross over' points, determine their competencies and run various sets of experiments in order to compare them.

All these goals were achieved to some level. The three algorithms were indeed implemented (the Fast Fourier Transform implementation was an adaptation of an existing one).

The 'cross over' points were determined and they were outlined in the previous chapter.

The analysis of the experiments' results shown that both Traditional and Fourier behave close to their theoretical performance. However, Karatsuba does not behave close to its theoretical performance until the input is of about 800 digits. The reason for Karatsuba being slower than expected goes back to the time spent in allocating memory for the arrays and freeing it after that.

The results also show some unexpected behaviour of the Fast Fourier multiplication as it is a lot faster than predicted.

All the three multiplication techniques do run until system runs out of memory. They all output correct and precise products even for multiplicands of size 100,000 digits. See the accompanying test results along with the program source code in the CD.

7.2 Where the requirements specification met?

Yes, the program was written in the C programming language. It runs on the BUCS machine (see appendix for how to load and run the program). The result of the multiplication is correct. The software as a whole meets its requirements specification.

7.3 Can the software be improved?

Yes, there are many things that can be done differently. For example, implement the algorithms using different radices.

The only bug found when carrying the testing was that if we multiply 0 by 1234 using Fourier technique the output would be 34. This is false of course. However, if we multiply 00 by 1234 then the output is 0. It seems that Fourier implementation multiplies 0 by the first two digits only. This could have been fixed of course, and to be honest, this was only noticed the night before the deadline of the project.

Karatsuba seems to need lots of improvements. Karatsuba is meant to be quicker than Fourier for at least up to quite big numbers. This was not the case.

The timing mechanism is not precise as wanted it to be. There are lots of factors causing this. One of them is that the software is not the only process running in the BUCS *amos* machine. The other is that the C library functions use the system clock that returns approximations of the system time used by a program. This is the reason for having to repeat the multiplication 10,000 times in order to get a good timing average.

8 Bibliography

Books used

A. Karatsuba, Ofman Yu, *Multiplication of multiple numbers by mean of automata*, Dokadly Akad. Nauk SSSR 145, no 2, 1962, pp293-294.

D. E. Knuth, *The Art of Computer Programming*, Volume 2 Seminumerical Algorithms, Readings, Edition Addison-Wesley, 1981.

A. Schönhage, V. Strassen, *Schnelle Multiplikation Grosser Zhalen*, Computing, 7(1971), pp281-292.

J. Lipson, *Elements of algebra and algebraic computing*, 1981

Crandall R. and Pomerance C, *Prime Numbers: a Computational Perspective*, Springer (2000)

Daryl Pike. Comparing algorithms for the multiplication of large integers. University Of Bath library, september 2000

P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic Complexity Theory*. Springer, 1997.

Jean M.Firth. *Discrete Transforms,* First edition 1992, Chapman & Hall.

URLs used

Mathematical Constants and Computation. http://numbers.computation.free.fr/

Karatsuba. http://www.swox.com/gmp/manual/

http://www.answers.com

9 Appendices

Appendix A - Compiling and running the code

Installation guide

Copy the code to your work space

Make sure you copy all three files:

1. generate.c
2. alg.c
3. header.h

1. To compile use the command

```
mary $ gcc generate.c -lm -o generate
mary $
```

```
mary $ gcc alg.c -lm -o alg
mary $
```

2. To generate random numbers do:

```
mary $ generate > input.txt
debug:  Generating the default 10 pairs of integers to change use flag -p
debug:  Using the default difference 10 to change use flag -d
mary $
```

This will output 10 pairs of integers, starting with pair length of 10 and increment by ten. So, the last pair would have 100 digits of length.

3. To specify length of pairs as well as there number use:

```
mary $ generate -d 4 -p 5 > input.txt
debug:  Generating 5 pairs of integers
debug:  Generating pairs with 4 digit(s) difference
mary $
```

4. If you want to see what numbers have been generated

```
mary $ flip -m input.txt
mary $
```

Now, double click on inpt.txt in your directory. You will be able to see what numbers have been generated.

5. To see what the program do type in UNIX

```
mary $ alg

        +----------------------------+
        | Big Integer Multiplication |
        +----------------------------+

Useage:
        ALG [ -T | -K | -F ] {-M xxx | -R xxx}

        [ T ] = Traditional Algorithm
        [ K ] = Karatsuba Algorithm
        [ F ] = Fast Fourier Transform Algorithm

        [ M ] = Allow a maximum of xxx digit inputs (default 100000)
        [ R ] = Repeat the test xxx times (default 10000)

This program is designed to output results in .csv format.
It is recommended that you run the program in the following fashion:

        ALG x < input_file >> output.csv

Where x is your choice of algorithm and input_file
contains an even number of lines of integers
mary $
```

6. To run Karatsuba do:

```
mary $ alg -k -r 10000 -m 10

        +----------------------------+
        | Big Integer Multiplication |
        +----------------------------+

debug:  Using Karatsuba algorithm...
debug:  Accepting a maximum of 10 digits
debug:  Repeating 10000 times

------------------------------------------------------------------------------
algorithm, a, b, result, num_repeats, largest_length, operation_time (ms), total_time (ms)
------------------------------------------------------------------------------

a:      2
b:      3
karatsuba, 2, 3, 6, 10000, 1, 0.002000, 20.000000

a:      ^[
debug:  User terminated program
mary $
```

- You can specify the maximum length as you like, as well as the number of repeats.
- If you do not then the program will use the default values 100,000 and 10,000 respectively.
- For Traditional and FFT use the flags –t and –f respectively.

7**. To read from the file generated previously and output to a file do the following:**

```
mary $ flip -u input.txt
mary $ alg -k -r 10000 -m 10 < input.txt >> output.csv

        +---------------------------+
        | Big Integer Multiplication |
        +---------------------------+

debug:  Using Karatsuba algorithm...
debug:  Accepting a maximum of 10 digits
debug:  Repeating 10000 times

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------

a:      b:
a:      b:
a:      debug:  Each multiplicand must be <= 10 digit(s) long
mary $
```

- Got error message as the second pair in the input.txt file was larger than 10 digits long. Remember we generated a file of 10 pairs starting from 10 and incrementing by ten so the second multiplicands do have 20 digits which are greater then length that we specified and which is 10.

- Double click the output.csv file in your directory to see the results.

Maintenance guide

1. To generate numbers of length power of a number. Open generate.c file, comment out line 81 and comment line 78.

2. To generate numbers incrementing by a factor of a number. Comment line 81, and comment out line 78.
3. the rest can be done automatically using the flage –d (to set a factor or a power), -p (to specify the numbers of pairs to generate), -r (to set number of repeats), -m (to set a maximum length on multiplicands), -t, -k, and –f (to perform the multiplication using one of the techniques).

Appendix B - Program source code

1. header.h

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <math.h>
#include <time.h>

#define TRADITIONAL 0
#define KARATSUBA 1
#define FOURIER 2

#define SWAP(a, b) tmp = (a); (a) = (b); (b) = tmp

#define RADIX 10.0

#define NE 1

// how many times we want to run each algorithm. this is for comparaison purposes

#define NUM_REPEATS 10000


/* using 132MB of Ram we can have up to 132 * 1024 = 135168.
 However, at least 528MB of RAM is needed to hold the multiplicands and the product
*/

#define MAX_LENGTH 100000
```

```
#define PI 3.141592653589793

void set_zeros(unsigned int *array, unsigned int array_length);

void carry(unsigned int *array, unsigned int array_length);

void read_input(unsigned int *array, unsigned int *array_length, unsigned int max_length);

void read_fourier_input(unsigned int *array, unsigned int *array_length, unsigned int max_length);

void print_array(unsigned int *array, unsigned int array_length);

void print_fourier_array(unsigned int *array, unsigned int array_length);

void trad_multiply(unsigned int *array_a, unsigned int *array_b, unsigned int *array_c, unsigned int array_length);

void karatsuba(unsigned int *array_a, unsigned int *array_b, unsigned int *array_c, unsigned int array_length);

void fourier(unsigned int u[], unsigned int v[], unsigned int w[], int n, int m);

void dfour1(double data[], unsigned long nn, int isign);

void drealft(double data[], unsigned long n, int isign);

double *dvector(long n1, long nh);

void free_dvector(double *v, long n1, long nh);
```

2. alg.c

```
#include "header.h"

/**
 * Supervisor: Dr Russel Bradford, Student: Mohamed Arabi
 * University of Bath, Department of Computer Science
 * Final Version produced: 08/05/2005
 *
 * Introduction:
 *
 * 1. In this file the three different algorithms are implemented.
 * 2. The Maximum length of the multiplicands is specified in the header file.
 *    you can specify the maximum length when running the code as alg -t -m 20000
 *    here -t refers to traditional technique, -m refers to the maximum length which is 20000
 * 3. the same applies to the number of repeats and the flag for it is -r
 * 4. if u do not specify a read from file, it means you want to type in the multiplicands your self.
 * 5. press Esc and hit return to quit program
 * 6. you can not run all techniques at the same time so, "alg -t -k" , will only run karatsuba routine
 * 7. if u do not specify which algorithm to use, you will be redirected to the how to run and use this system
 *    introduction
 *
 * look in the accompanying documentation for all options that you can do with this program
 */


int main(int argc, char** argv) {
```

```
// these values may get overidden by command line switches
int algorithm = TRADITIONAL;
int num_repeats = NUM_REPEATS;
int max_length = MAX_LENGTH;

unsigned int a[ max_length ];
unsigned int b[ max_length ];
unsigned int c[ 2 * max_length ];
unsigned int a_length, b_length;
unsigned int largest_length, shifted_length;
unsigned int i;

double start_time, end_time;
double total_time, operation_time;

fprintf(stderr, "\n\t+----------------------------+\n\t| Big Integer Multiplication |\n\t+----------------------------+\n\n");

if(argc > 1){
        for (i = 1; i < argc; i++) {
                if (argv[ i ][ 0 ] == '-' || argv[ i ][ 0 ] == '/') {
                        switch (argv[ i ][ 1 ]) {
                        case 't':
                        case 'T':
                                algorithm = TRADITIONAL;
                                break;
                        case 'k':
                        case 'K':
                                algorithm = KARATSUBA;
                                break;
                        case 'f':
```

```
                                case 'F':
                                        algorithm = FOURIER;
                                        break;
                                case 'm':
                                case 'M':
                                        max_length = atoi(argv[++i]);
                                        break;
                                case 'r':
                                case 'R':
                                        num_repeats = atoi(argv[++i]);
                                        break;
                                default:
                                        fprintf(stderr, "debug:\tInvalid switch '%s' encountered\n", argv[i]);
                                        break;
                                }
                        }
                }
        }
        else{
                // if no arguments passed, show help info
                fprintf(stderr, "Useage:\n\tALG [ -T | -K | -F ] {-M xxx | -R xxx}\n\n\t[ T ] = Traditional Algorithm\n\t[ K ] = Karatsuba Algorithm\n\t[ F ] = Fast Fourier Transform Algorithm\n\n\t[ M ] = Allow a maximum of xxx digit inputs (default %d)\n\t[ R ] = Repeat the test xxx times (default %d)\n\nThis program is designed to output results in .csv format.\nIt is recommended that you run the program in the following fashion:\n\n\tALG x < input_file >> output.csv\n\nWhere x is your choice of algorithm and input_file\ncontains an even number of lines of integers\n", MAX_LENGTH, NUM_REPEATS);
                exit(0);
        }

        // repeating the switch statement to avoid repeating ourselves if the user runs for example 'alg -t -k'
        switch(algorithm){
```

```
		case TRADITIONAL:
			fprintf(stderr, "debug:\tUsing Traditional algorithm...\n");
			break;
		case KARATSUBA:
			fprintf(stderr, "debug:\tUsing Karatsuba algorithm...\n");
			break;
		case FOURIER:
			fprintf(stderr, "debug:\tUsing Fourier algorithm...\n");
			break;
}

if(max_length != MAX_LENGTH){
		fprintf(stderr, "debug:\tAccepting a maximum of %d digits\n", max_length);
}else{
fprintf(stderr, "debug:\tAccepting a maximum of %d digits\n", MAX_LENGTH);
}

if(num_repeats != NUM_REPEATS){
		fprintf(stderr, "debug:\tRepeating %d times\n", num_repeats);
}else{
fprintf(stderr, "debug:\tRepeating %d times\n", NUM_REPEATS);
}


// print out a header, only print border on stdout not to the .csv file
fprintf(stderr, "\n--------------------------------------------------------------------------------\n");
printf("algorithm, a, b, result, num_repeats, largest_length, operation_time (ms), total_time (ms)\n");
fprintf(stderr, "--------------------------------------------------------------------------------\n");

while( 1 ){
```

```
if(algorithm == FOURIER){
	fprintf(stderr, "\na:\t");
	read_fourier_input(a, &a_length, max_length);
	fprintf(stderr, "b:\t");
	read_fourier_input(b, &b_length, max_length);
}
else{
	fprintf(stderr, "\na:\t");
	read_input(a, &a_length, max_length);
	fprintf(stderr, "b:\t");
	read_input(b, &b_length, max_length);
}

largest_length = (a_length > b_length) ? a_length : b_length;

if(algorithm == TRADITIONAL){

	// print input and output results
	printf("traditional, ");
	print_array(a, largest_length);
	printf(", ");
	print_array(b, largest_length);
	printf(", ");



	start_time = (double) clock();

	for (i = 0; i < num_repeats; i++){
```

```
                trad_multiply(a, b, c, largest_length);
                carry(c, (2 * largest_length));
        }

        end_time = (double) clock();

        print_array(c, (2 * largest_length));
        printf(", ");

}
else if(algorithm == KARATSUBA){

        // shift bits up a power of two using C's built in shift operator
        shifted_length = 1;
        while(shifted_length < largest_length){
                shifted_length <<= 1;
        }

        // print input and output results
        printf("karatsuba, ");
        print_array(a, shifted_length);
        printf(", ");
        print_array(b, shifted_length);
        printf(", ");

        set_zeros(c, (2 * largest_length));

        start_time = (double) clock();
```

```
        for (i = 0; i < num_repeats; i++){

                karatsuba(a, b, c, shifted_length);
                carry(c, (2 * shifted_length));
        }

        end_time = (double) clock();

        print_array(c, (2 * largest_length));
        printf(", ");

}
else if(algorithm == FOURIER){

        // print input and output results
        printf("fourier, ");
        print_fourier_array(a, a_length);
        printf(", ");
        print_fourier_array(b, b_length);
        printf(", ");

        set_zeros(c, (2 * largest_length));

        start_time = (double) clock();

        for (i = 0; i < num_repeats; i++){

                fourier(a, b, c, a_length, b_length);
        }
```

```
                end_time = (double) clock();

                print_fourier_array(c, (a_length + b_length));
                printf(", ");

            }

            total_time = ( end_time - start_time ) / 1000;
            operation_time = total_time / num_repeats;

            // print statistics
            printf("%d, %d, %f, %f\n", num_repeats, largest_length, operation_time, total_time);
        }
        return 0;
}



/**
 * set_zeros() fill array with zero's


 *


 * @param     array          A pointer to the array to be initialised to zero
 * @param     array_length   Length of possible output result
 */
```

```
void set_zeros(unsigned int *array, unsigned int array_length){
        int i;
        for(i = 0; i < array_length; i++){
                array[ i ] = 0;
        }
}
```

```
/**
 * read_input() read input array either from a file or from the command line
 * the way traditional and karatsuba, read the array is different from fourier
 * as traditional and karatsuba read array from right to left and fourier from


 * left to right
 *
 * this function read array from right to left
 *
 *
 @param         array               A pointer to the array to be read in our case it will be
 *                          the multiplicands
 *
 * @param       array_length    A pointer to the length of array
 *
 * @param       max_length      Maximum length of possible input
 */
```

```
void read_input(unsigned int *array, unsigned int *array_length, unsigned int max_length){
	int next_char;
	int i;
	int tmp;
	*array_length = 0;

	set_zeros(array, max_length);

	while( 1 ){
		next_char = getchar();
		if(next_char == EOF){
			fprintf(stderr, "debug:\tEnd of input reached\n");
			exit(0);
		}
		if(next_char == 27){
			fprintf(stderr, "debug:\tUser terminated program\n");
			exit(0);
		}
		if(next_char == '\n'){
			break;
		}
		if(next_char < 48 || next_char > 57){
			// In ascii notation, 48 = 0 and 57 = 9
			// disregard any other characters
			fprintf(stderr, "debug:\tInput was not a positive integer\n");
			exit(1);
		}
		if(*array_length >= max_length){
			// not setup to handle large integers due to memory constraints
```

```
                fprintf(stderr, "debug:\tEach multiplicand must be <= %d digit(s) long\n", max_length);
                exit(1);
            }
            array[ *array_length ] = next_char - '0';
            ++( *array_length );
        }

        if(*array_length == 0){
            fprintf(stderr, "debug:\tZero length input found\n");
            exit(0);
        }



        // storing the array backwards

        for(i = 0; (i * 2) < (*array_length - 1); i++){
            tmp = array[ i ];
            array[ i ] = array[ *array_length - i - 1 ];
            array[ *array_length - i - 1 ] = tmp;
        }

}



/**
 * trad_multiply() multiplies two integer numbers using the traditional technique
```

```
 *


 * @param      array_a          A pointer to the first array to be multiplied
 * @param      array_b                  Pointer to the second array to be multiplied

 * @param      array_c          Array to hold the result in

 * @param      array_length     Length of possible output result
 */

void trad_multiply(unsigned int *array_a, unsigned int *array_b, unsigned int *array_c, unsigned int array_length){
        int i, j;

        set_zeros(array_c, (2 * array_length));

        for(i = 0; i < array_length; i++){
                for(j = 0; j < array_length; j++){
                        array_c[ i + j ] += array_a[ i ] * array_b[ j ];
                }
        }
}



/**
 * karatsuba() multiplies two integer numbers using the karatsuba technique
```

```
*


* @param      array_a         A pointer to the first array to be multiplied
* @param      array_b                 Pointer to the second array to be multiplied

* @param      array_c         Array to hold the result in

* @param      array_length    Length of possible output result
*/

void karatsuba(unsigned int *array_a, unsigned int *array_b, unsigned int *array_c, unsigned  int array_length){
        int i;
        // note that a_left refers to the lefthand side of the array, not the number that is represented in the array
        unsigned int *a_left = &array_a[ 0 ];
        unsigned int *a_right = &array_a[ array_length / 2 ];
        unsigned int *b_left = &array_b[ 0 ];
        unsigned int *b_right = &array_b[ array_length / 2 ];
        unsigned int *a_sum, *b_sum, *c_mid, *c_left, *c_right;

        // break out of the recursive loop if the length is small
        if (array_length < 4){
                // run traditional multiplication as length of input is small
                trad_multiply(array_a, array_b, array_c, array_length);
                return;
        }
        else{
                a_sum = calloc((array_length / 2), sizeof(unsigned int));
                b_sum = calloc((array_length / 2), sizeof(unsigned int));
```

```
if(!a_sum || !b_sum){
        fprintf(stderr, "debug:\tFailed to allocate memory in karatsuba()\n");
}
for(i = 0; i < (array_length / 2); i++){
        a_sum[ i ] = a_left[ i ] + a_right[ i ];
        b_sum[ i ] = b_left[ i ] + b_right[ i ];
}
c_mid = calloc(array_length, sizeof(unsigned int));
karatsuba(a_sum, b_sum, c_mid, (array_length / 2));

c_right = calloc(array_length, sizeof(unsigned int));
karatsuba(a_right, b_right, c_right, (array_length / 2));

c_left = calloc(array_length, sizeof(unsigned int));
karatsuba(a_left, b_left, c_left, (array_length / 2));

for (i = 0; i < array_length; i++){
        c_mid[ i ] = c_mid[ i ] - c_right[ i ] - c_left[ i ];
}
for(i = 0; i < array_length; i++){
        array_c[ i ] = c_left[ i ];
        array_c[ array_length + i ] = c_right[ i ];
}
for(i = 0; i < array_length; i++){
        array_c[ i + (array_length / 2) ] += c_mid[ i ];
}
free(a_sum);
free(b_sum);
free(c_left);
```

```
                free(c_right);
                free(c_mid);
        }
}



/**
 * carry() does the addition and shifting, making sure the array cells contains only one digit


 *


 * @param      array           A pointer to the array to be carried
 * @param      array_length    Length of possible output result
 */


void carry(unsigned int *array, unsigned int array_length){
        int i;
        int tmp;
        tmp = 0;

        for(i = 0; i < array_length; i++){
                array[ i ] += tmp;
                tmp = array[ i ] / 10;
                array[ i ] -= tmp * 10;
        }
}
```

```
/**
 * print_array() print array resulting from traditional and karatsuba
 * to the screen or to the output file as specified by the user
 * print the product array to the screen


 *


 * @param      array          A pointer to the array to be printed
 * @param      array_length   Length of possible output result
 */

void print_array(unsigned int *array, unsigned int array_length){
        int i = array_length - 1;

        while(array[ i ] == 0){
                i--;
        }
        while(i >= 0){
                printf("%d", array[ i ]);
                i--;
        }
}
```

```
/**
 * read_fourier_input() read input array either from a file or from the command line
 *
 * read array from right to left
 *
 *
 @param         array               A pointer to the array to be read in our case it will be
 *                          the multiplicands
 *
 * @param       array_length    A pointer to the length of array
 *
 * @param       max_length     Maximum length of possible input
 */


void read_fourier_input(unsigned int *array, unsigned int *array_length, unsigned int max_length){
        int next_char;
        array[ 0 ] = 0;
        *array_length = 0;

        set_zeros(array, (*array_length + 1));

        while(1){
                next_char = getchar();
                if(next_char == EOF){
                        fprintf(stderr, "debug:\tEnd of input reached\n");
                        exit(0);
                }
                if(next_char == 27){
```

```
                fprintf(stderr, "debug:\tUser terminated program\n");
                //break;
                exit(0);
            }
            if(next_char == '\n'){
                break;
            }
            if(next_char < 48 || next_char > 57){
                fprintf(stderr, "error:\tInput was not a positive integer\n");
                exit(1);
            }
            if(*array_length >= max_length){
                fprintf(stderr, "error:\tEach multiplicand must be <= %d digit(s) long\n", max_length);
                exit(1);
            }
            array[ *array_length + 1 ] = next_char - '0';
            ++( *array_length );
        }

        if(*array_length == 0){
            fprintf(stderr, "debug:\tZero length input found\n");
            exit(0);
        }
}



/**
 * fourier() multiplies two integer numbers using the fourier technique
```

```
*


* @param     array_a         A pointer to the first array to be multiplied
* @param     array_b                 Pointer to the second array to be multiplied

* @param     array_c         Array to hold the result in

* @param     a_length        Length of the first multiplicand
* @param     b_length        Length of the second multiplicand
*/


void fourier(unsigned int *array_a, unsigned int *array_b, unsigned int *array_c, int a_length, int b_length){
        int i, nn = 1, mn;
        double cy, t;
        double *a, *b;
        mn = b_length;

        while(nn < mn){
                nn <<= 1;
        }
        nn <<= 1;

        a = dvector(1, nn);
        b = dvector(1, nn);

        for(i = 1; i <= a_length; i++)
                a[ i ] = (double) array_a[ i ];
```

```
for(i = (a_length + 1); i <= nn; i++)
        a[ i ] = 0.0;

for(i = 1; i <= b_length; i++)
        b[ i ] = (double) array_b[ i ];

for(i = (b_length + 1); i <= nn; i++)
        b[ i ] = 0.0;

drealft(a, nn, 1);
drealft(b, nn, 1);
b[ 1 ] *= a[ 1 ];
b[ 2 ] *= a[ 2 ];

for(i = 3; i <= nn; i += 2){
        b[ i ] = (t = b[ i ]) *a[ i ] - b[ i + 1 ]*a[ i + 1 ];
        b[ i + 1 ] = t*a[ i + 1 ] + b[ i + 1 ]*a[ i ];
}

drealft(b, nn, -1);

cy = 0.0;

for(i = nn; i >= 1; i--){
        t = b[ i ] / ( nn >> 1 ) + cy + 0.5;
        cy = (unsigned long) ( t / RADIX );
        b[ i ] = t - cy * RADIX;
}
```

```
	if (cy >= RADIX){
		fprintf(stderr, "debug:\tValue greater than Radix\n");
	}
	array_c[ 1 ] = (unsigned char) cy;

	for(i = 2; i <= a_length + b_length; i++) array_c[ i ] = (unsigned char) b[ i - 1 ];

	free_dvector(a, 1, nn);
	free_dvector(b, 1, nn);
}


/**
 * discrete_fourier_transform() part of the fourier technique


 *


 * @param	data		A pointer to the array holding the DFT values
 * @param	nn		a power of two
 * @param	isign		1 for DFT or -1 for inverse DFT
 */


void discrete_fourier_transform(double *data, unsigned long nn, int isign){
	unsigned long i, j;
	unsigned long n, m;
	unsigned long mmax, istep;
	double wtemp, wr, wpr, wpi, wi, theta;
```

```
double tmp, tmpi;

n = nn << 1;
j = 1;

for(i = 1; i < n; i += 2){
        if( j > i ){
                SWAP(data[ j ], data[ i ]);
                SWAP(data[ j + 1 ], data[ i + 1 ]);
        }
        m = n >> 1;
        while(m >= 2 && j > m){
                j -= m;
                m >>= 1;
        }
        j += m;
}
mmax = 2;
while(n > mmax){
        istep = mmax << 1;
        theta = isign * ((PI * 2) / mmax);
        wtemp = sin(0.5 * theta);
        wpr = -2.0 * wtemp * wtemp;
        wpi = sin(theta);
        wr = 1.0;
        wi = 0.0;
        for ( m = 1 ; m < mmax; m += 2 ){
                for (i = m; i <= n; i += istep){
                        j = i + mmax;
                        tmp = wr * data[ j ] - wi * data[ j + 1 ];
```

```
                                tmpi = wr * data[ j + 1 ] + wi * data[ j ];
                                data[ j ] = data[ i ] - tmp;
                                data[ j + 1 ] = data[ i + 1 ] - tmpi;
                                data[ i ] += tmp;
                                data[ i + 1 ] += tmpi;
                        }
                        wr = (wtemp = wr) * wpr - wi * wpi + wr;
                        wi = wi * wpr + wtemp * wpi + wi;
                }
                mmax = istep;
        }
}


/**
 * drealft() part of the fourier technique


 *


 * @param      data            A pointer to the array holding the DFT values
 * @param      nn              a power of two
 * @param      isign           1 for DFT or -1 for inverse DFT
 */



void drealft(double *data, unsigned long n, int isign){
        unsigned long i, i1, i2, i3, i4, np3;
```

```
double c1 = 0.5, c2, h1r, h1i, h2r, h2i;
double wr, wi, wpr, wpi, wtemp, theta;
theta = PI / (double) (n >> 1);

if(isign == 1){
        c2 = -0.5;
        discrete_fourier_transform(data, n >> 1, 1);
}else{
        c2 = 0.5;
        theta = -theta;
}
wtemp = sin(0.5 * theta);
wpr = -2.0 * wtemp * wtemp;
wpi = sin(theta);
wr = 1.0 + wpr;
wi = wpi;
np3 = n + 3;

for(i = 2; i <= (n >> 2); i++){
        i4 = 1 + (i3 = np3 - (i2 = 1 + (i1 = i + i - 1)));
        h1r= c1 * (data[ i1 ] + data[ i3 ]);
        h1i= c1 * (data[ i2 ] - data[ i4 ]);
        h2r= -c2 * (data[ i2 ] + data[ i4 ]);
        h2i= c2 * (data[ i1 ] - data[ i3 ]);

        data[ i1 ] = h1r + wr * h2r - wi * h2i;
        data[ i2 ] = h1i + wr * h2i + wi * h2r;
        data[ i3 ] = h1r - wr * h2r + wi * h2i;
        data[ i4 ] = -h1i + wr * h2i + wi * h2r;
```

```
                wr = (wtemp = wr) * wpr - wi * wpi + wr;
                wi = wi * wpr + wtemp * wpi + wi;
        }

        if(isign == 1){
                data[ 1 ] = (h1r = data[ 1 ]) + data[ 2 ];
                data[ 2 ] = h1r - data[ 2 ];
        }else{
                data[ 1 ] = c1 * ((h1r = data[ 1 ]) + data[ 2 ]);
                data[ 2 ] = c1 * (h1r - data[ 2 ]);
                discrete_fourier_transform(data, n >> 1, -1);
        }
}



// allocate memory space for the two arrays used in fourier technique

double *dvector(long n1, long nh){
        double *v;
        v = (double *) malloc((size_t) ((nh - n1 + 1 + NE) * sizeof(double)));
        if(!v){
                fprintf(stderr, "debug:\tFailed to allocate memory in dvector()\n");
        }
        return v - n1 + NE;
}

//  free memory allocated

void free_dvector(double *v, long n1, long nh){
```

```
	free((char*) (v + n1 - NE));
}


/**
 * print_fourier_array() print array resulting from the fast Fourier transform
 * to the screen or to the output file as specified by the user. in our case the
 * array is the product


 *


 * @param	array		A pointer to the array to be printed
 * @param	array_length	Length of possible output result
 */

void print_fourier_array(unsigned int *array, unsigned int array_length){
	int i = 0;

	while(array[ i ] == 0){
		i++;
	}
	while(i <= array_length){
		printf("%d", array[ i ]);
		i++;
	}
}
```

3. generate.c

```
#include <stdio.h>
#include <stdlib.h>
#include "header.h"



#defineNUM_ARRAYS  10


#define DIFFERENCE  10

/**
 *        generate random numbers, the number of these random numbers can
```

```
 *      be changed, by changing the number of num_arrays, as well as the difference
 *      between each pair by changing difference in the code below
 */



main(int argc, char** argv){


        // number of integer pairs to create
        unsigned int num_arrays = NUM_ARRAYS;

        // difference between lengths
        unsigned int difference = DIFFERENCE;

        unsigned int array_a[ MAX_LENGTH ];
        unsigned int array_b[ MAX_LENGTH ];
        unsigned int i,j, max_length;


        if(argc > 1){
                for (i = 1; i < argc; i++) {
                        if (argv[ i ][ 0 ] == '-' || argv[ i ][ 0 ] == '/') {
                                switch (argv[ i ][ 1 ]) {

                                case 'p':
                                case 'P':
                                        num_arrays = atoi(argv[++i]);
                                        break;
                                case 'd':
```

```
                        case 'D':
                                difference = atoi(argv[++i]);
                                break;
                        default:
                                fprintf(stderr, "debug:\tInvalid switch '%s' encountered\n", argv[i]);
                                break;
                        }
                }
        }
    }


if(num_arrays != NUM_ARRAYS){
            fprintf(stderr, "debug:\tGenerating %d pairs of integers\n", num_arrays);
}else{
        fprintf(stderr, "debug:\tGenerating the default %d pairs of integers to change use flag -p\n", NUM_ARRAYS);
}

    if(difference != DIFFERENCE){
            fprintf(stderr, "debug:\tGenerating pairs with %d digit(s) difference\n", difference);

    }else{
        fprintf(stderr, "debug:\tUsing the default difference %d to change use flag -d\n", DIFFERENCE);

    }




    create_seed();
```

```
for(i = 1; i <= num_arrays; i++){

	// to generate pairs of random numbers with first pair of length difference then increment by i * difference
	max_length = i * difference;

	// to generate pairs of random numbers with first pair of length difference then increment exponentially
	//max_length = pow(difference,i);

	if(max_length > MAX_LENGTH){
		break;
	}

	for(j = 0; j < max_length; j++){
		array_a[ j ] = random_digit();
		array_b[ j ] = random_digit();

		while(array_a[ 0 ] == 0){array_a[ 0 ] = random_digit();}
		while(array_b[ 0 ] == 0){array_b[ 0 ] = random_digit();}

		printf("%d", array_a[ j ]);
	}

	printf("\n");

	for(j = 0; j < max_length; j++){
		printf("%d", array_b[ j ]);
	}

	printf("\n");
```

```
        }

}

/**
 *      Srand will seed the random number generator to prevent random numbers
 *      from being the same every time the program is executed and to allow
 *      more pseudorandomness.
 */

int create_seed( void ){
        srand((unsigned) time( NULL )); // Casts time's return to unsigned

}

/**
 *      create a random digit
 */

int random_digit( void ){
        int r;
        r = rand() % 10;
        return r;
}
```