Extended Essay in Computer Science

Comparing the efficiency of different algorithms for finding the n-th digit of π

Word count: 3474

Abstract

When solving programming problems online, once I came across a task to calculate the most digits of π possible. I was not even able to compare to the other people's solutions and it motivated me to look at this area of programming with deeper analysis.

There are already many algorithms for the computation of π discovered and explained among the internet, so I decided to compare the efficiency of such three algorithms, computing the n-th digit of π, which are most suitable for this Extended Essay.

These three algorithms were tested in order to compare their efficiency: Bellard's algorithm, Spigot algorithm of Rabinowitz and Wagon, Bailey-Borwein-Plouffe (BBP) algorithm. In order to compare them properly, there were two types of testing provided. In the first one the algorithms were adjusted to calculate a given number of digits of π – n, while the duration of the operations was measured. In the second testing the algorithms were supposed to calculate a digit of π on a given position – n, while the time it took them to compute it was measured again.

By providing various inputs of n, it was shown that the BBP algorithm is generally the most efficient one, but it also differs from the others by computing the digits only in hexadecimal base. The Rabinowitz's algorithm has proved to be suitable for calculating n digits of π, while using quite a big amount of memory, whereas the Bellard's algorithm seems to be more appropriate for computing the n-th digit of π, while using only very little memory.

Word count: 256

Table of Contents

Abstract ..... - 2 -

1 Introduction ..... - 5 -

2 Theoretical information ..... - 6 -

2.1 The irrational number π ..... - 6 -

2.2 Real-life application ..... - 7 -

2.3 Algorithms for finding the value of π ..... - 7 -

2.3.1 Bellard's algorithm (Plouffe's) ..... - 8 -

2.3.2 Spigot algorithm of Rabinowitz and Wagon ..... - 10 -

2.3.3 Bailey–Borwein–Plouffe algorithm (BBP) ..... - 12 -

3 Experiment ..... - 12 -

3.1 Building the programmes ..... - 13 -

4 Data collection and results ..... - 14 -

4.1 Method ..... - 14 -

4.2 Results ..... - 15 -

4.2.1 Computation of the whole value of π ..... - 15 -

4.2.2 Computation of a specific digit of π on a given position ..... - 16 -

5 Evaluation and Conclusion ..... - 19 -

Bibliography ..... - 21 -

Appendices ..... - 23 -

Appendix A – Source code ..... - 23 -

A.1 – Bellard’s algorithm edited for computing a given number of digits.....- 23 -

A.2 – Bellard’s algorithm edited for computing the exact n-th digit of pi.....- 28 -

A.3 – Spigot Algorithm of Rabinowitz and Wagon edited for both computing a given number of digits and computing the exact n-th digit of pi. .....- 32 -

A.4 – BBP algorithm edited for computing a given number of digits. .....- 34 -

A.5 – BBP algorithm edited for computing the exact n-th digit of pi. .....- 37 -

1 Introduction

Since I solved my first serious programming problem, which was to evaluate certain fractions by using the shortest code possible, I have found myself very interested in such challenging problems. There is a site with many problems of such kind, where I could develop my skills and also the interest in this type of programming throughout my high-school years. Due to its challenging nature, people from all over the world can compete by improving the efficiency of the solution to the problem. That is what I like about it the most. There is no right or certain solution, because it can be still upgraded by any programmer. And such ability of our civilization is leading us to evolutions, as it can be seen in the history, so this kind of programming can have a very beneficial impact on our future lives.

During the IBD programme, I do not have that much time for solving these problems online as before, but I still like to have a look at one at least once a month. Usually I tend to use programming languages such as Pascal, C++, or Java, which I learned in my Computer Science classes.

While solving these problems, two years ago I came across a question of how to evaluate the irrational number π with the most decimal digits possible. It was a great challenge that I immediately started to work on. But after certain amount of precise decimal places I just could not go further. Whether it was due to the lack of my knowledge, or a problem with the programming software, fourteen decimal places was as far as I got.

Later on I found myself struggling through the choice of the topic for my extended essay and I realised that the problem with π that I once tried to solve would be a great

base for my essay and also an opportunity to resolve this problem properly with in-depth analysis.

My research question thus is: **Comparing the efficiency of different algorithms for finding the n-th digit of π.**

In my Extended Essay I will analyse three different algorithms for finding the n-th digit of π, which I found the most complex and sophisticated, thus yielding the best results. Then I will implement them in C++ programming language and compare their time complexity and specific usefulness in various tasks. For example one might be more appropriate for finding the n-th digit of π, while n being a very large integer, but on the other hand, another one might be more appropriate for evaluating the exact π number as such. I am going to choose a specific number of digits that I want to calculate, so that the difference in results could be clearly seen. In addition, I will compare the result of the algorithms with the real value of π, so that I will be sure that it is working properly. Then I will discuss the specific differences of the algorithms by analysing the data.

2 Theoretical information

2.1 The irrational number π

The irrational number π had been a mystery for more than 4000 years, messing with scientists' heads sometimes throughout their whole life. But as the time goes by, the human civilization achieved many great technological advances, significantly in the last decades. Thanks to this evolution, people have been able to evaluate the number π with so many digits, it ceases to be imaginable. The algorithms for computing such

irrational number were, of course, invented a lot earlier, but the computation of the most digits possible came to life only after the massive development of computers.

Then the race began. The programmers tried to convert the mathematical algorithms into programmes, which was quite hard to do at higher number of digits, because of inefficient use of memory and loops. Since the algorithms were invented only for mathematical purposes, there was no need to think about such things. But as people tried to compute more and more decimal digits of π, they realised that there had to be a totally new kind of algorithm created.

2.2 Real-life application

These algorithms were invented to calculate the very last digit possible, which actually is only a part of theoretical mathematics. It means that the actual value of π up to 2700 billion digits, as it was calculated by Bellard with an ordinary inexpensive desktop computer (Bellard, 2010), is not needed practically in any scientific area. Also NASA Jet Propulsion Laboratory answered the question of how many decimals of π do we really need using this statement: "…by cutting π off at the 15th decimal point, we would calculate a circumference for that circle that is very slightly off. It turns out that our calculated circumference of the 25 billion mile diameter circle would be wrong by 1.5 inches." (NASA/JPL Edu, 2016)

2.3 Algorithms for finding the value of π

As an ordinary mathematical algorithm for finding the value of π did not work very well in programming, another way of calculating π was invented. Calculating it by computing one digit after another seemed to be the best approach. When there is no need to use the number of type double already evaluated before, it is possible to come

to millions of digits without stumbling upon the problem of memorizing so many digits in one simple variable. That is why I will compare the efficiency of algorithms for finding the n-th digit of π.

2.3.1 Bellard's algorithm (Plouffe's)

This algorithm was invented by Fabrice Bellard in 1997. It was basically an improvement of Simon Plouffe's new algorithm, which was using very little memory, but its time complexity was quite inefficient: $O(n^3 \log(n)^3)$. The improved Bellard's algorithm could operate within time complexity of $O(n^2)$ while its memory requirements stayed $O(1)$, which made it practical to compute for example the millionth digit of π on an ordinary desktop computer (Bellard, 1997).

It all started with Plouffe's inspiring new algorithm, which could calculate the specific digits of π in any base. It was based on the BBP algorithm, discovered a few months earlier, and a new algorithm that simply splits an ordinary fraction into its components. His new formula for π used is:

$$\pi + 3 = \sum_{n=1}^{\infty} \frac{n 2^n n!^2}{(2n)!}$$

The explicit key observation and formula can be accessed in Plouffe's work (Plouffe, 1996), where he gets through various mathematical operations to the total cost of $n^3 \log(n)^2$. This cost is for the computation of the k'th partial sum of

$$\sum_{n=1}^{\infty} \frac{2^n}{n\binom{2n}{n}}$$

where $\binom{2n}{n}$ is the central binomial coefficient. If we want at each step to compute (the final n-th digit) then we need $\log(n)$ steps to do it (Plouffe, 1996). Due to the fact that this is an actual explicit fraction independent of the base, this is the point, where the n-th digit of π in decimal base can be computed in $n^3\log(n)^3$ steps (Plouffe, 1996).

Although it was a great progress, such algorithm was still a more theoretical rather than practical approach to the calculation of the digits of π in decimal base. But later on, Fabrice Bellard took a look at Plouffe's algorithm and applied various other formulas to improve the algorithm. As he said in his article: "The running time is $O(n^2)$ because there are $O(\frac{n}{\log(n)})$ prime numbers between 2 and 2n. The memory requirements are, as expected, in $O(1)$." (Bellard, 1997) Here is the application of the algorithm:

- $N \leftarrow \lfloor (n+\varepsilon)\log_2(B) \rfloor$ where **ε** is a small integer to ensure we have the precision needed; $sum \leftarrow 0$.
- For each prime number **a** with **2<a<2N**, do:
  - $v^{\max} \leftarrow \left\lfloor \frac{\log(2N)}{\log(a)} \right\rfloor; m \leftarrow a^{v^{\max}}$.
  - $a \leftarrow 0; v \leftarrow 0; b \leftarrow 1; A \leftarrow 1$
  - For **k** in **2N** do:
    - $b \leftarrow \frac{k}{a^{v(a,k)}} b \bmod m; A \leftarrow \frac{(2k-1)}{a^{v(a,2k-1)}} A \bmod m; v \leftarrow v - v(a,k) + v(a,2k-1)$
    - If $v > 0$ do: $a \leftarrow a + k.b.A^{-1}.a^{v^{\max}-v} \bmod m$.
  - $a \leftarrow aB^{n-1} \bmod m; sum \leftarrow sum + \frac{a}{m} \bmod 1$.
- If we suppose that $\pi = (d_0 d_1 d_2 d_3 \ldots)_B$ then, if we neglect rounding errors, $sum = (0.d_n d_{n+1} d_{n+2} \ldots d_{n+q-1} \ldots)_B$. The number **q** of correct digits depends on **ε**.

The algorithm computes the n-th digit of π in any given base B without using any high precision computations, which is quite incredible, but it is still slower than the BBP algorithm. Later on, Fabrice Bellard improved his algorithm making it even more

efficient using the Gosper formula, which yields more than one decimal digit for each term.

2.3.2 Spigot algorithm of Rabinowitz and Wagon

The spigot algorithm for calculating the digits of π and other numbers have been invented by Stanley Rabinovitz in 1991 and investigated by Rabinovitz and Wagon in 1995. This algorithm is called “spigot”, because it pumps out digits one at a time and does not use the digits once they are computed (Rabinowitz & Wagon, 1995). It is based on the expansion:

$$\pi = \sum_{i=1}^{\infty} \frac{(i!)^2 2^{i+1}}{(2i+1)!}$$

Although their algorithm uses bounded integer arithmetic, it remains very efficient (Gibbons, 2005). No complicated multiple precision arithmetic library is used, which keeps it very simplistic. But on the other hand, the memory usage is very inefficient, because the program uses an array of length (10n/3) + 1, which raises with every increasing n by more than 3 times. The expansion above expands out to the expression:

$$\pi = 2 + \frac{1}{3}\left(2 + \frac{2}{5}\left(2 + \frac{3}{7}\left(\ldots\left(2 + \frac{i}{2i+1}(\ldots)\right)\right)\right)\right)$$

According to Jeremy Gibbons (Gibbons, 2005), one can view the expression above as representing a number (2;2,2,2,...) in a mixed-radix base $B = \left(\frac{1}{3}, \frac{2}{5}, \frac{3}{7}, \ldots\right)$, in the same way that the usual decimal expansion of π:

$$\pi = 3 + \frac{1}{10}\left(1 + \frac{1}{10}\left(4 + \frac{1}{10}\left(1 + \frac{1}{10}\left(1 + \frac{1}{10}(5 + \cdots)\right)\right)\right)\right)$$

represents (3;1,4,1,5,...) in the fixed-radix base F_{10}, where $F_m = \left(\frac{1}{m}, \frac{1}{m}, \frac{1}{m}, \ldots\right)$. So the final thing that the programme has to do in order to calculate digits in decimal base is simply converting each term from base B to base F_{10}.

The final algorithm is clearly shown in this table (Rabinowitz & Wagon, 1995):

	Digits of π	1/3	2/5	3/7	4/9	5/11	6/13	7/15	8/17	9/19	10/21	11/23	12/25
Initialize		2	2	2	2	2	2	2	2	2	2	2	2
× 10		20	20	20	20	20	20	20	20	20	20	20	20
Carry	3	+10	+12	+12	+12	+10	+12	+7	+8	+9	+0	+0	+0	=
		30	32	32	32	30	32	27	28	29	20	20	20	20
Remainders		0	2	2	4	3	10	1	13	12	1	20	20	20
× 10		0	20	20	40	30	100	10	130	120	10	200	200	200
Carry	1	+13	+20	+33	+40	+65	+48	+98	+88	+72	+150	+132	+96	=
		13	40	53	80	95	148	108	218	192	160	332	296	200
Remainders		3	1	3	3	5	5	4	8	5	8	17	20	0
× 10		30	10	30	30	50	50	40	80	50	80	170	200	0
Carry	4	+11	+24	+30	+40	+40	+42	+63	+64	+90	+120	+88	+0	=
		41	34	60	70	90	92	103	144	140	200	258	200	0
Remainders		1	1	0	0	0	4	12	9	4	10	6	16	0
× 10		10	10	0	0	0	40	120	90	40	100	60	160	0
Carry	1	+4	+2	+9	+24	+55	+84	+63	+48	+72	+60	+66	+0	=
		14	12	9	24	55	124	183	138	112	160	126	160	0

Firstly, there has to be an array of length (10n/3) + 1 initialized and in every element the number 2 must be stored. Then, the process begins - going from right to left carrying the remainders, which are important for calculating the digits on every decimal place. Every element in the array is multiplied by 10, then the entries are reduced modulo the denominator of the column head, and the quotients are carried left after multiplication by the numerator. Each time the process is executed, it pumps out one digit of π in decimal base. In order to get n decimal places of π, the process needs to be repeated n times.

2.3.3 Bailey–Borwein–Plouffe algorithm (BBP)

The BBP formula for finding the value of π was discovered in 1995 by Simon Plouffe, Peter Borwein and David Bailey:

$$\pi = \sum_{k=0}^{\infty} \left[\frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \right]$$

By using this formula, one can compute the n-th digit of π, without computing all the first n digits. This method permits to obtain the n-th bit of π in time $O(nlog^3 n)$ and space $O(\log n)$ (Gourdon & Sebah, 2003). This algorithm is considered to be one of the quickest algorithms to compute the n-th digit of π. It has been also generally used to calculate the most digits of π possible, using it pretty efficiently: Due to the fact that the computation does not depend on the first n digits, there can run millions of these computations simultaneously, separated on many CPUs. Then the CPUs get the result together after a really short period of time. The only disadvantage is that it calculates the digits in hexadecimal base. Then it depends also on the time complexity of the converting algorithm, which converts the result from hexadecimal to decimal base. In this Extended Essay I will observe only the BBP algorithm as it is, without any converting algorithms, which could greatly influence the results.

3 Experiment

In order to evaluate the efficiency of the three algorithms, multiple tasks have to be done. Firstly, I will implement the algorithms in C++, also including the method for calculating the time elapsed during the execution of the algorithm. Secondly, while observing the appropriate use cases of each algorithm, in addition to focusing on evaluating the whole number π with precision to the most decimal places possible, I

will also compare the efficiency of algorithms from the point of evaluating the exact n-th digit. Then I will gather the data and results, which I will be gained from the programmes. This includes the time of the process and the number of digits of π evaluated. The second part of my research will include the time of the process, which will find the exact digit on the given position.

3.1 Building the programmes

I will make three programmes, while each of them will represent one of the algorithms presented afore. I could find most of the explanations on how to implement the algorithms from various sources already, which were quite accurate, but I had to edit them due to my specific research purposes.

The Bellard's algorithm was designed to calculate the exact digit at a given position, followed by the next eight digits. In order to use this algorithm also as an evaluation of the whole number π, some changes had to be made. I made a for loop, where the given position of the digit that I want to calculate increases by nine after every loop. The next thing I added to the programme is timer. It will show the time it took to do the given task in seconds with precision up to three decimal digits. Such precision will be enough, when it is taken into account that I will choose the most last digit positions possible to make the results clear. In addition, more than three decimal places would not even be needed, because the computer is making an error of approximately 0.010 seconds by itself. Because of this fact, I will approximate the time data up to two decimal places later on. After all, I also made the programme count the number of digits every time it writes them, to make sure that it prints out the exact number of digits I wanted.

The Spigot Algorithm of Rabinowitz and Wagon was already designed to calculate the whole number π, with computation of one digit after another. In order to capture the time, for how long the algorithm does operate, there had to be added the time computation again. By using the *ctime* and *iomanip* libraries of C++, the programme returns the exact time of the operation with precision up to three decimal places, like in the other programmes. Also a digit counter is included, by which it can be proved that it evaluated the exact number of digits it was supposed to evaluate.

The implementation of the Bailey–Borwein–Plouffe (BBP) algorithm in C was published by Bailey in 2006 fortunately, so it could be used for my research (Bailey, 2006). As always, firstly I had to understand the programme properly, so that I could edit it for my special needs. It is used for computation of the digit on a given position and other nine following digits in hexadecimal base. That means that in order to calculate the most possible number of digits of π, there had to be a for loop added, very similarly as in the case of Bellard's algorithm. Although it is computing the digits in hexadecimal base, it is better not to convert it to the decimal base, because it could significantly influence the operation time, and therefore it would cause very misleading results in the end. The research question is only about comparing the algorithms as they are and not rewriting them in such a significant way.

4 Data collection and results

4.1 Method

To make the data collection the most efficient, there will be several different numbers of digits given to each of the algorithms. At every computation, the time it took the algorithm to do it will be written in the table. This process will be repeated three times

for each given number of digits and then the average time will be calculated from the three results and written into the table. The bigger the number of digits, the more correct the result will be. After this observation, a second type of testing of these algorithms will take place. Several different positions of digits of π will be given, which should be displayed by the specific programme. Again, the average time for each position will be calculated and processed into the data. After obtaining all the data described above, various graphs for every algorithm will be created to show the specific differences, which then will be evaluated in the conclusion.

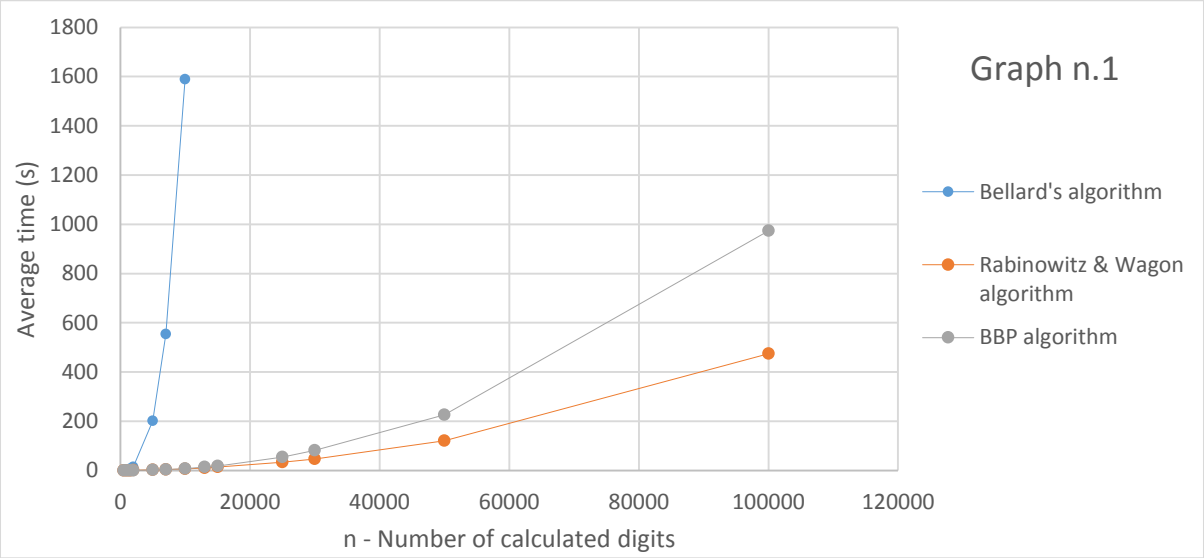
4.2 Results

4.2.1 Computation of the whole value of π

In the first type of testing, the time of computation of a given number of digits was measured:

N - NUMBER OF DIGITS	**AVERAGE TIME (S)**		
	Bellard's algorithm	Rabinowitz & Wagon algorithm	BBP algorithm
500	0.31	0.08	0.03
1000	2.01	0.19	0.08
1500	6.16	0.28	0.17
2000	14.32	0.40	0.31
5000	201.68	1.68	1.83
7000	553.50	3.02	3.74
10000	1589.39	5.97	7.91
13000	-	10.03	13.42
15000	-	13.52	18.12
25000	-	33.22	54.30
30000	-	45.90	81.65
50000	-	120.76	226.43
100000	-	474.04	974.17

When it comes to the Bellard's algorithm, it is very inefficient to use it for computing all the digits of π. As the number of digits approached 15000, it would take nearly an hour to calculate it. I simply left the rows with bigger values of n empty, because it is not needed to prove its inefficiency to such extent. The graph is already adjusted to the two other algorithms, so the next points of Bellard's algorithm curve would not even be seen on the graph.



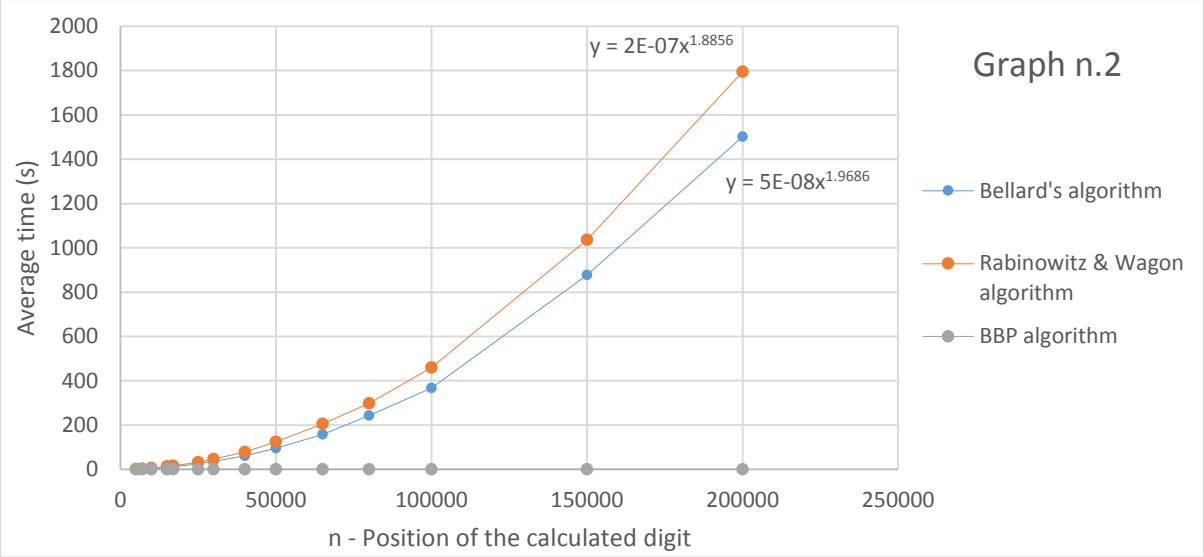
The Spigot algorithm of Rabinowitz & Wagon is clearly ahead of the other two algorithms, which shows that it has the best time complexity when it comes to displaying a given number of digits of π. Among the small values of n, until 5000 digits, the BBP algorithm has better time complexity, but then with increasing n, Rabinowitz's algorithm takes the lead. Bellard's algorithm has a horrible time complexity compared to the other two algorithms.

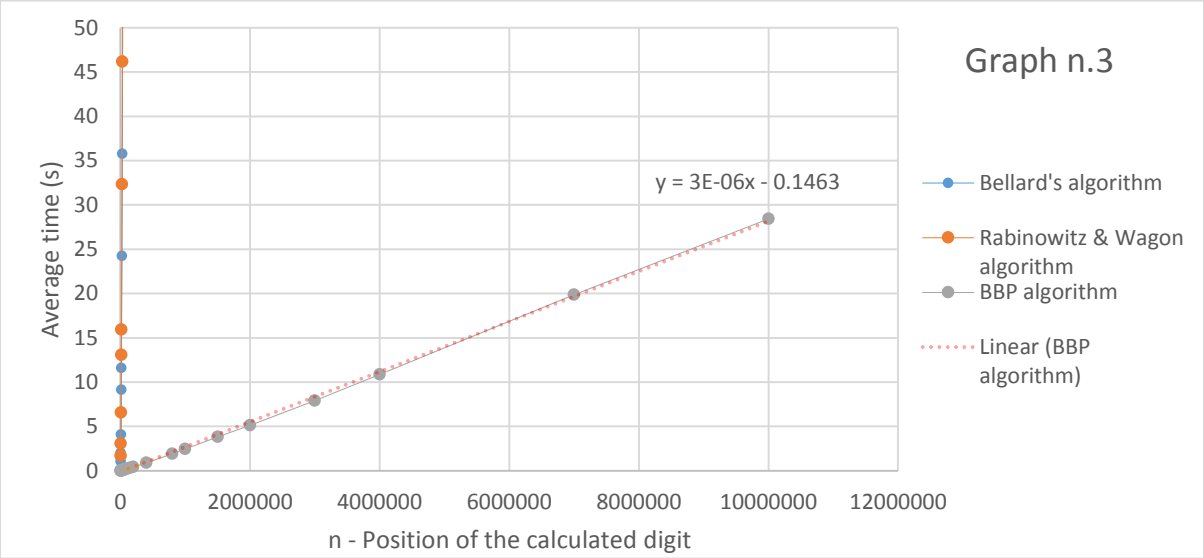
4.2.2 Computation of a specific digit of π on a given position

In the second type of testing, the time of computation of a given position of a specific digit was measured:

N - POSITION OF THE DIGIT	AVERAGE TIME (S)		
	Bellard's algorithm	Rabinowitz & Wagon algorithm	BBP algorithm
5000	1.05	1.68	0.01
7000	2.01	3.05	0.01
10000	4.08	6.55	0.02
15000	9.14	13.06	0.03
17000	11.59	15.92	0.03
25000	24.24	32.32	0.05
30000	35.76	46.18	0.06
40000	61.01	77.76	0.08
50000	96.22	124.83	0.10
65000	157.14	206.03	0.13
80000	243.00	298.37	0.16
100000	366.99	460.36	0.21
150000	877.96	1036.68	0.32
200000	1501.69	1794.23	0.44
400000	-	-	0.91
800000	-	-	1.92
1000000	-	-	2.44
1500000	-	-	3.81
2000000	-	-	5.13
3000000	-	-	7.91
4000000	-	-	10.88
7000000	-	-	19.86
10000000	-	-	28.42

The BBP algorithm calculates the n-th digit incredibly quicker compared to the other two algorithms. It could compute the ten millionth digit in less than half a minute, whereas the Bellard's and Rabinowitz's algorithms would stuck on computing only the 400000^{th} digit for more than an hour. Even though the two algorithms were so inefficient compared to the BBP algorithm, I decided to continue in the experiment, at least observing only the BBP algorithm. I came to these two graphs:





In the first graph is shown the similar time complexity of Bellard's and Rabinowitz's & Wagon's algorithms. They both seem to approximately represent a quadratic function, due to the trendlines' equations, which are nearly of x^2 base.

BBP algorithm has so incomparably lower time complexity than the other ones that I had to show its data on a graph with different axis values. There it is clear, that it is nearly a perfect linear function with very small gradient = 0.000003.

5 Evaluation and Conclusion

The aim of this essay was to compare the efficiency of three different algorithms for finding the n-th digit of π. Thus, I divided my research into two experiments. The first one was comparing the time complexity of the algorithms, while calculating all the digits of π up to n-th digit. The second experiment included comparing the time complexity of the algorithms, while calculating the exact n-th digit without previous digits, if possible.

It is clear, that Bellard's algorithm has for both experiments pretty high time complexity compared to the other algorithms. Such difference may be caused by the different purposes of the algorithms. Francis Bellard only made the Plouffe's formula more practical by decreasing its time complexity, but the original purpose of this formula was the very little memory needed in order to run this algorithm on any ordinary desktop computer, even for higher n (O(1)). On the other hand, it is still quite outstanding in the matter of computing the exact n-th digit of π. Although the BBP algorithm is incomparably quicker, the Bellard's algorithm can be in for any output base, not only hexadecimal.

When it comes to Spigot algorithm of Rabinowitz and Wagon, it was created only with the purpose of displaying all digits up to given n-th digit. In the first experiment it showed the best results, so I can say that it has a very great efficiency, which met the expectations. On the other hand, it operates with quite much memory, compared to the Bellard's algorithm, because with every increase of given n, the length of the array inside the programme increases approximately 3,3 times. Because of its only purpose, its time complexity did not change during the second experiment, while the other algorithms operated more efficiently. It is due to the fact that there was no other way

to calculate the exact n-th digit than simply calculating the first n digits, as in the first experiment.

The BBP algorithm proved that it is averagely the most efficient one in general. When calculating all the digits up to the n-th, its time complexity was a bit worse than the Rabinowitz's algorithm had, but it is still usable even for such experiment. Furthermore, the results from the second experiment were impressive. While the other two algorithms it took almost half an hour to calculate the 200000^{th} digit, the BBP got it in less than half a second. That is an incredible difference. The third graph supports its unbelievable efficiency by showing its time in relation to n in nearly perfect linear function. That is also why it is so efficient for calculating the digits of π on different computers simultaneously. The only problem is that without a conversion to decimal base, it would have no meaning, so then the time complexity depends mostly on the converting algorithm. Since this Extended Essay is supposed to research only the algorithms for computing the n-th digit of π, no converting algorithm is included in it.

Bibliography

1. Bailey, D. H. (2006, September 8) Retrieved November 23, 2016, from David H. Bailey: http://www.experimentalmath.info/bbp-codes/piqpr8.c
2. Bellard, F. (1997, January 12). *Computation of the n'th digit of pi in any base in O(n^2).* Retrieved November 14, 2016, from Fabrice Bellard: http://bellard.org/pi/pi_n2/pi_n2.html#bbp95
3. Bellard, F. (2010, February 11). *Computation of 2700 billion decimal digits of Pi using a Desktop Computer.* Retrieved November 14, 2016, from Fabrice Bellard: http://www.bellard.org/pi/pi2700e9/pipcrecord.pdf
4. Bogomolny, A. *Calculation of the Digits of pi by the Spigot Algorithm of Rabinowitz and Wagon*. Retrieved November 12, 2016, from Interactive Mathematics Miscellany and Puzzles: http://www.cut-the-knot.org/Curriculum/Algorithms/SpigotForPi.shtml
5. Gibbons, J. (2005) *Unbounded Spigot Algorithms for the Digits of Pi*. Retrieved November 23, 2016, from Jeremy Gibbons: http://www.cs.ox.ac.uk/jeremy.gibbons/publications/spigot.pdf
6. Gourdon, X., & Sebah, P. (2003, February 12) *N-th digit computation.* Retrieved November 10, 2016, from Xavier Gourdon & Pascal Sebah: http://www.plouffe.fr/simon/articles/nthdigit.pdf
7. NASA/JPL Edu (2016, March 16). *How Many Decimals of Pi Do We Really Need?* Retrieved November 15, 2016, from NASA Jet Propulsion Laboratory: http://www.jpl.nasa.gov/edu/news/2016/3/16/how-many-decimals-of-pi-do-we-really-need/

8. Plouffe, S. (1996, November 30). *On the computation of the n'th decimal digit of various transcendental numbers.* Retrieved November 12, 2016, from Simon Plouffe: http://vixra.org/pdf/1409.0080v1.pdf
9. Rabinowitz, S., & Wagon, S. (1995, March) *A Spigot Algorithm for the Digits of Pi.* Retrieved November 10, 2016, from Mathematical Association of America: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.693.8211&rep=rep1&type=pdf

Appendices

Appendix A – Source code

A.1 – Bellard's algorithm edited for computing a given number of digits.

(Original retrieved from Bellard's implementation: http://bellard.org/pi/pi1.c)

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include <conio.h>
#include <ctime>
#include <iomanip>
using namespace std;

#define HAS_LONG_LONG

#define mul_mod(a,b,m) (( (long long) (a) * (long long) (b) ) % (m))

/* return the inverse of x mod y */
int inv_mod(int x, int y)
{
    int q, u, v, a, c, t;

    u = x;
    v = y;
    c = 1;
    a = 0;
    do {
    q = v / u;

    t = c;
    c = a - q * c;
    a = t;

    t = u;
    u = v - q * u;
    v = t;
    } while (u != 0);
    a = a % y;
    if (a < 0)
    a = y + a;
    return a;
}

/* return the inverse of u mod v, if v is odd */
int inv_mod2(int u, int v)
{
    int u1, u3, v1, v3, t1, t3;

    u1 = 1;
```

```
    u3 = u;

    v1 = v;
    v3 = v;

    if ((u & 1) != 0) {
    t1 = 0;
    t3 = -v;
    goto Y4;
    } else {
    t1 = 1;
    t3 = u;
    }

    do {

    do {
        if ((t1 & 1) == 0) {
        t1 = t1 >> 1;
        t3 = t3 >> 1;
        } else {
        t1 = (t1 + v) >> 1;
        t3 = t3 >> 1;
        }
      Y4:;
    } while ((t3 & 1) == 0);

    if (t3 >= 0) {
        u1 = t1;
        u3 = t3;
    } else {
        v1 = v - t1;
        v3 = -t3;
    }
    t1 = u1 - v1;
    t3 = u3 - v3;
    if (t1 < 0) {
        t1 = t1 + v;
    }
    } while (t3 != 0);
    return u1;
}

/* return (a^b) mod m */
int pow_mod(int a, int b, int m)
{
    int r, aa;

    r = 1;
    aa = a;
    while (1) {
    if (b & 1)
        r = mul_mod(r, aa, m);
    b = b >> 1;
    if (b == 0)
        break;
    aa = mul_mod(aa, aa, m);
    }
    return r;
}
```

```
/* return true if n is prime */
int is_prime(int n)
{
    int r, i;
    if ((n % 2) == 0)
    return 0;

    r = (int) (sqrt(n));
    for (i = 3; i <= r; i += 2)
    if ((n % i) == 0)
        return 0;
    return 1;
}

/* return the prime number immediatly after n */
int next_prime(int n)
{
    do {
    n++;
    } while (!is_prime(n));
    return n;
}

#define DIVN(t,a,v,vinc,kq,kqinc)             \
{                                             \
  kq+=kqinc;                                  \
  if (kq >= a) {                              \
    do { kq-=a; } while (kq>=a);              \
    if (kq == 0) {                            \
      do {                                    \
        t=t/a;                                \
        v+=vinc;                              \
      } while ((t % a) == 0);                 \
    }                                         \
  }                                           \
}

int main(int argc, char *argv[])
{
    int av, a, vmax, N, n, num, den, k, kq1, kq2, kq3, kq4, t, v, s, i, t1;
    double sum;

    clock_t start = clock();
    int dig;
    for (int digits = 1; digits < 10001; digits += 9){                /*
computing the whole pi by printing the computed 9 digits all over and over
*/

        n = digits;
        N = (int) ((n + 20) * log(10) / log(13.5));
        sum = 0;

        for (a = 2; a <= (3 * N); a = next_prime(a)) {
        vmax = (int) (log(3 * N) / log(a));
        if (a == 2) {
            vmax = vmax + (N - n);
            if (vmax <= 0)
            continue;
        }
        av = 1;
        for (i = 0; i < vmax; i++)
```

```
        av = av * a;

    s = 0;
    den = 1;
    kq1 = 0;
    kq2 = -1;
    kq3 = -3;
    kq4 = -2;
    if (a == 2) {
        num = 1;
        v = -n;
    } else {
        num = pow_mod(2, n, av);
        v = 0;
    }

    for (k = 1; k <= N; k++) {

        t = 2 * k;
        DIVN(t, a, v, -1, kq1, 2);
        num = mul_mod(num, t, av);

        t = 2 * k - 1;
        DIVN(t, a, v, -1, kq2, 2);
        num = mul_mod(num, t, av);

        t = 3 * (3 * k - 1);
        DIVN(t, a, v, 1, kq3, 9);
        den = mul_mod(den, t, av);

        t = (3 * k - 2);
        DIVN(t, a, v, 1, kq4, 3);
        if (a != 2)
        t = t * 2;
        else
        v++;
        den = mul_mod(den, t, av);

        if (v > 0) {
        if (a != 2)
            t = inv_mod2(den, av);
        else
            t = inv_mod(den, av);
        t = mul_mod(t, num, av);
        for (i = v; i < vmax; i++)
            t = mul_mod(t, a, av);
        t1 = (25 * k - 3);
        t = mul_mod(t, t1, av);
        s += t;
        if (s >= av)
            s -= av;
        }
    }
    t = pow_mod(5, n - 1, av);
    s = mul_mod(s, t, av);
    sum = fmod(sum + (double) s / (double) av, 1.0);
    }
    printf("%09d", (int) (sum * 1e9));
    dig = digits + 9 - 1;
}
long double duration = (clock() - start)/(double)CLOCKS_PER_SEC;
```

```
    cout << endl << fixed << setprecision(3) << "Time: " << duration <<
endl;
    cout << "Digits: " << dig << endl;
    return 0;
}
```

A.2 – Bellard's algorithm edited for computing the exact n-th digit of pi.

(Original retrieved from Bellard's implementation: http://bellard.org/pi/pi1.c)

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include <conio.h>
#include <ctime>
#include <iomanip>
using namespace std;

#define HAS_LONG_LONG

#define mul_mod(a,b,m) (( (long long) (a) * (long long) (b) ) % (m))

/* return the inverse of x mod y */
int inv_mod(int x, int y)
{
    int q, u, v, a, c, t;

    u = x;
    v = y;
    c = 1;
    a = 0;
    do {
    q = v / u;

    t = c;
    c = a - q * c;
    a = t;

    t = u;
    u = v - q * u;
    v = t;
    } while (u != 0);
    a = a % y;
    if (a < 0)
    a = y + a;
    return a;
}

/* return the inverse of u mod v, if v is odd */
int inv_mod2(int u, int v)
{
    int u1, u3, v1, v3, t1, t3;

    u1 = 1;
    u3 = u;

    v1 = v;
    v3 = v;

    if ((u & 1) != 0) {
    t1 = 0;
    t3 = -v;
    goto Y4;
```

```
    } else {
    t1 = 1;
    t3 = u;
    }

    do {

    do {
        if ((t1 & 1) == 0) {
        t1 = t1 >> 1;
        t3 = t3 >> 1;
        } else {
        t1 = (t1 + v) >> 1;
        t3 = t3 >> 1;
        }
      Y4:;
    } while ((t3 & 1) == 0);

    if (t3 >= 0) {
        u1 = t1;
        u3 = t3;
    } else {
        v1 = v - t1;
        v3 = -t3;
    }
    t1 = u1 - v1;
    t3 = u3 - v3;
    if (t1 < 0) {
        t1 = t1 + v;
    }
    } while (t3 != 0);
    return u1;
}

/* return (a^b) mod m */
int pow_mod(int a, int b, int m)
{
    int r, aa;

    r = 1;
    aa = a;
    while (1) {
    if (b & 1)
        r = mul_mod(r, aa, m);
    b = b >> 1;
    if (b == 0)
        break;
    aa = mul_mod(aa, aa, m);
    }
    return r;
}

/* return true if n is prime */
int is_prime(int n)
{
    int r, i;
    if ((n % 2) == 0)
    return 0;

    r = (int) (sqrt(n));
    for (i = 3; i <= r; i += 2)
```

```
    if ((n % i) == 0)
        return 0;
    return 1;
}

/* return the prime number immediatly after n */
int next_prime(int n)
{
    do {
    n++;
    } while (!is_prime(n));
    return n;
}

#define DIVN(t,a,v,vinc,kq,kqinc)              \
{                                              \
  kq+=kqinc;                                   \
  if (kq >= a) {                               \
    do { kq-=a; } while (kq>=a);               \
    if (kq == 0) {                             \
      do {                                     \
        t=t/a;                                 \
        v+=vinc;                               \
      } while ((t % a) == 0);                  \
    }                                          \
  }                                            \
}

int main(int argc, char *argv[])
{
    int av, a, vmax, N, n, num, den, k, kq1, kq2, kq3, kq4, t, v, s, i, t1;
    double sum;

    clock_t start = clock();
    int dig;
        n = 150000;
        N = (int) ((n + 20) * log(10) / log(13.5));
        sum = 0;

        for (a = 2; a <= (3 * N); a = next_prime(a)) {
        vmax = (int) (log(3 * N) / log(a));
        if (a == 2) {
            vmax = vmax + (N - n);
            if (vmax <= 0)
            continue;
        }
        av = 1;
        for (i = 0; i < vmax; i++)
            av = av * a;

        s = 0;
        den = 1;
        kq1 = 0;
        kq2 = -1;
        kq3 = -3;
        kq4 = -2;
        if (a == 2) {
            num = 1;
            v = -n;
        } else {
            num = pow_mod(2, n, av);
```

```
            v = 0;
        }

        for (k = 1; k <= N; k++) {

            t = 2 * k;
            DIVN(t, a, v, -1, kq1, 2);
            num = mul_mod(num, t, av);

            t = 2 * k - 1;
            DIVN(t, a, v, -1, kq2, 2);
            num = mul_mod(num, t, av);

            t = 3 * (3 * k - 1);
            DIVN(t, a, v, 1, kq3, 9);
            den = mul_mod(den, t, av);

            t = (3 * k - 2);
            DIVN(t, a, v, 1, kq4, 3);
            if (a != 2)
            t = t * 2;
            else
            v++;
            den = mul_mod(den, t, av);

            if (v > 0) {
            if (a != 2)
                t = inv_mod2(den, av);
            else
                t = inv_mod(den, av);
            t = mul_mod(t, num, av);
            for (i = v; i < vmax; i++)
                t = mul_mod(t, a, av);
            t1 = (25 * k - 3);
            t = mul_mod(t, t1, av);
            s += t;
            if (s >= av)
                s -= av;
            }
        }
        t = pow_mod(5, n - 1, av);
        s = mul_mod(s, t, av);
        sum = fmod(sum + (double) s / (double) av, 1.0);
        }
        printf("%09d", (int) (sum * 1e9));
    long double duration = (clock() - start)/(double)CLOCKS_PER_SEC;
    cout << endl << fixed << setprecision(3) << "Time: " << duration <<
endl;
    cout << "Digit: " << n << endl;
    return 0;
}
```

A.3 – Spigot Algorithm of Rabinowitz and Wagon edited for both computing a given number of digits and computing the exact n-th digit of pi.

(Original retrieved from Haenel's implementation including bug fixes:

http://www.jjj.de/hfloat/spigot.haenel.txt)

```
/*Spigot Algorithm for π

1.  Initialize: Let A = (2, 2, 2, 2,... ,2) be an array of length [10n/3] +
1.
2.  Repeat n times:
    Multiply by 10: Multiply each entry of A by 10.
    Put A into regular form: Starting from the right, reduce the ith
element of A (corresponding to b-entry (i - 1)/(2i - 1)) modulo 2i - 1, to
get a quotient q and a remainder r. Leave r in place and carry q(i - 1) one
place left. The last integer carried (from the position where i - 1 = 2)
may be as large as 19.
3.  Get the next predigit: Reduce the leftmost entry of A (which is at most
109 (= 9 - 10 + 191)) modulo 10. The quotient, q, is the new predigit of π,
the remainder staying in place.
4.  Adjust the predigits: If q is neither 9 nor 10, release the held
predigits as true digits of π and hold q. If q is 9, add q to the queue of
held predigits. If q is 10 then:
    set the current predigit to 0 and hold it;
    increase all other held predigits by 1(9 becomes 0);
    release as true digits of π all but the current held predigit.*/

#include <math.h>
#include <stdio.h>
#include <iostream>
//#include <conio.h>
#include <ctime>
#include <iomanip>

using namespace std;

#define N    150000                        // Decimals to compute.
#define LEN (10L * N) / 3 + 1           // Chain length

unsigned j, predigit, nines, a[LEN];
long x, q, k, len, i;

int main()
{
    int dig = 0;
    clock_t start = clock();
    for(j=N; j; )
    {
        q = 0;
        k = LEN+LEN-1;
```

```
        for(i=LEN; i; --i)
        {
            x = (j == N ? 20 : 10L*a[i-1]) + q*i;
            q = x / k;
            a[i-1] = (unsigned)(x-q*k);
            k -= 2;
        }

        k = x % 10;
        if (k==9)  ++nines;

        else
        {
            if (j)
            {
            --j;
            printf("%ld", predigit+x/10);
            ++dig;
            }

            for(; nines; --nines)
            {
                if (j){
                    --j;
                    printf(x >= 10 ? "0" : "9");
                    ++dig;
                }
            }

            predigit = (unsigned)k;
        }
    }
    long double duration = (clock() - start)/(double)CLOCKS_PER_SEC;
    cout << endl << fixed << setprecision(3) << "Time: " << duration <<
endl;
    cout << "Digits: " << dig << endl;
    return 0;
}
```

A.4 – BBP algorithm edited for computing a given number of digits.

(Original retrieved from Bailey's implementation:
http://www.experimentalmath.info/bbp-codes/piqpr8.c)

```
#include <stdio.h>
#include <math.h>
#include <ctime>
#include <iomanip>
#include <iostream>

using namespace std;

int main()
{
  clock_t start = clock();
  double pid, s1, s2, s3, s4;
  double series (int m, int n);
  void ihex (double x, int m, char c[]);
  int id = 0;
#define NHX 16
  char chx[NHX];
  int dig = 0;

/*  id is the digit position.  Digits generated follow immediately after
id. */
  for (int digits = 0; digits < 25000; digits += 10){
      id = digits;
      s1 = series (1, id);
      s2 = series (4, id);
      s3 = series (5, id);
      s4 = series (6, id);
      pid = 4. * s1 - 2. * s2 - s3 - s4;
      pid = pid - (int) pid + 1.;
      ihex (pid, NHX, chx);
      printf ("%10.10s",chx);
      dig += 10;
  }
  long double duration = (clock() - start)/(double)CLOCKS_PER_SEC;
  cout << endl << fixed << setprecision(3) << "Time: " << duration << endl;
  cout << "Digits: " << dig << endl;
  return 0;
}

void ihex (double x, int nhx, char chx[])

/*  This returns, in chx, the first nhx hex digits of the fraction of x. */

{
  int i;
  double y;
  char hx[] = "0123456789ABCDEF";

  y = fabs (x);

  for (i = 0; i < nhx; i++){
    y = 16. * (y - floor (y));
    chx[i] = hx[(int) y];
  }
```

```
}

double series (int m, int id)

/*  This routine evaluates the series  sum_k 16^(id-k)/(8*k+m)
    using the modular exponentiation technique. */

{
  int k;
  double ak, eps, p, s, t;
  double expm (double x, double y);
#define eps 1e-17

  s = 0.;

/*  Sum the series up to id. */

  for (k = 0; k < id; k++){
    ak = 8 * k + m;
    p = id - k;
    t = expm (p, ak);
    s = s + t / ak;
    s = s - (int) s;
  }

/*  Compute a few terms where k >= id. */

  for (k = id; k <= id + 100; k++){
    ak = 8 * k + m;
    t = pow (16., (double) (id - k)) / ak;
    if (t < eps) break;
    s = s + t;
    s = s - (int) s;
  }
  return s;
}

double expm (double p, double ak)

/*  expm = 16^p mod ak.  This routine uses the left-to-right binary
    exponentiation scheme. */

{
  int i, j;
  double p1, pt, r;
#define ntp 25
  static double tp[ntp];
  static int tp1 = 0;

/*  If this is the first call to expm, fill the power of two table tp. */

  if (tp1 == 0) {
    tp1 = 1;
    tp[0] = 1.;

    for (i = 1; i < ntp; i++) tp[i] = 2. * tp[i-1];
  }

  if (ak == 1.) return 0.;

/*  Find the greatest power of two less than or equal to p. */
```

```
  for (i = 0; i < ntp; i++) if (tp[i] > p) break;

  pt = tp[i-1];
  p1 = p;
  r = 1.;

/*  Perform binary exponentiation algorithm modulo ak. */

  for (j = 1; j <= i; j++){
    if (p1 >= pt){
      r = 16. * r;
      r = r - (int) (r / ak) * ak;
      p1 = p1 - pt;
    }
    pt = 0.5 * pt;
    if (pt >= 1.){
      r = r * r;
      r = r - (int) (r / ak) * ak;
    }
  }

  return r;
}
```

A.5 – BBP algorithm edited for computing the exact n-th digit of pi.

(Original retrieved from Bailey's implementation:

http://www.experimentalmath.info/bbp-codes/piqpr8.c)

```
#include <stdio.h>
#include <math.h>
#include <ctime>
#include <iomanip>
#include <iostream>

using namespace std;

int main()
{
  clock_t start = clock();
  double pid, s1, s2, s3, s4;
  double series (int m, int n);
  void ihex (double x, int m, char c[]);
  int id = 150000;
#define NHX 16
  char chx[NHX];

/*  id is the digit position.  Digits generated follow immediately after
id. */

  s1 = series (1, id);
  s2 = series (4, id);
  s3 = series (5, id);
  s4 = series (6, id);
  pid = 4. * s1 - 2. * s2 - s3 - s4;
  pid = pid - (int) pid + 1.;
  ihex (pid, NHX, chx);
  printf ("%10.10s\n", chx);
  long double duration = (clock() - start)/(double)CLOCKS_PER_SEC;
  cout << endl << fixed << setprecision(3) << "Time: " << duration << endl;
  cout << "Digit: " << id << endl;
  return 0;
}

void ihex (double x, int nhx, char chx[])

/*  This returns, in chx, the first nhx hex digits of the fraction of x. */

{
  int i;
  double y;
  char hx[] = "0123456789ABCDEF";

  y = fabs (x);

  for (i = 0; i < nhx; i++){
    y = 16. * (y - floor (y));
    chx[i] = hx[(int) y];
  }
}
```

```
double series (int m, int id)

/*  This routine evaluates the series  sum_k 16^(id-k)/(8*k+m)
    using the modular exponentiation technique. */

{
  int k;
  double ak, eps, p, s, t;
  double expm (double x, double y);
#define eps 1e-17

  s = 0.;

/*  Sum the series up to id. */

  for (k = 0; k < id; k++){
    ak = 8 * k + m;
    p = id - k;
    t = expm (p, ak);
    s = s + t / ak;
    s = s - (int) s;
  }

/*  Compute a few terms where k >= id. */

  for (k = id; k <= id + 100; k++){
    ak = 8 * k + m;
    t = pow (16., (double) (id - k)) / ak;
    if (t < eps) break;
    s = s + t;
    s = s - (int) s;
  }
  return s;
}

double expm (double p, double ak)

/*  expm = 16^p mod ak.  This routine uses the left-to-right binary
    exponentiation scheme. */

{
  int i, j;
  double p1, pt, r;
#define ntp 25
  static double tp[ntp];
  static int tp1 = 0;

/*  If this is the first call to expm, fill the power of two table tp. */

  if (tp1 == 0) {
    tp1 = 1;
    tp[0] = 1.;

    for (i = 1; i < ntp; i++) tp[i] = 2. * tp[i-1];
  }

  if (ak == 1.) return 0.;

/*  Find the greatest power of two less than or equal to p. */

  for (i = 0; i < ntp; i++) if (tp[i] > p) break;
```

```
  pt = tp[i-1];
  p1 = p;
  r = 1.;

/*  Perform binary exponentiation algorithm modulo ak. */

  for (j = 1; j <= i; j++){
    if (p1 >= pt){
      r = 16. * r;
      r = r - (int) (r / ak) * ak;
      p1 = p1 - pt;
    }
    pt = 0.5 * pt;
    if (pt >= 1.){
      r = r * r;
      r = r - (int) (r / ak) * ak;
    }
  }

  return r;
}
```