Distribution of Primitive Roots *modulo m*

Sílvia Casacuberta

1 Introduction

This year we have studied congruences and modular arithmetic [1] in the IB Option of Discrete Mathematics. We learnt that, if a and b are integers and n is a positive integer, a is *congruent* to b *modulo* n if a and b yield the same remainder when divided by n. This is written as

$$a \equiv b \pmod{n}. \tag{1}$$

I became interested in this topic and especially in Fermat's Little Theorem [1], which states that

$$a^p \equiv 1 \pmod{p} \tag{2}$$

for every prime p and every integer a coprime to p. This is a particular case of Euler's Theorem [2]:

$$a^{\phi(n)} \equiv 1 \pmod{n}, \tag{3}$$

where a is any integer coprime to n, and $\phi(n)$ denotes Euler's Totient Function[1] [2]. The function ϕ returns the number of positive integers smaller than n that are coprime to n. For instance, $\phi(p) = p - 1$ if p is a prime, since all positive integers smaller than p are coprime to p. By using congruences I realized that by raising a number to consecutive powers one obtains certain cycles. For instance, the powers of 3 *modulo* 5 are

$$3^1 \equiv 3 \qquad 3^2 \equiv 4 \qquad 3^3 \equiv 2 \qquad 3^4 \equiv 1 \qquad 3^5 \equiv 3. \tag{4}$$

Thus 3 generates a cycle of length 4 *modulo* 5, since when we reach the fifth power it is congruent again to 3. This means that 3 is a *generator* of the multiplicative group of integers *modulo* 5. In general, we say that g is a *primitive root modulo* n if g generates a cycle of length $\phi(n)$ [3]. So if we keep raising g to all the powers from 1 to $n - 1$ we will obtain $\phi(n)$ different results. Using Euler's Totient Function, we can also say that if g is a primitive root[2] then $\phi(n)$ has to be the smallest exponent to which g is congruent to 1 *modulo* n. It is important to notice that not all integers have primitive roots: there exist primitive roots *modulo* n if and only if n is equal to 2, 4, p^k or $2p^k$ where p is an odd prime. Moreover, if a number n is of this form then there are $\phi(\phi(n))$ primitive roots [4].

However, given n there is no known formula or fast way to find its primitive roots. One way is to fix a certain base, which we call *possible primitive root*, keep raising it to increasing exponents from 1 to $n - 1$, and check if it yields $\phi(n)$ different results (in fact, there is one method that can be used to reduce the number of exponents that have to be checked, which will be discussed in the first section).

This made me think that it might be interesting to investigate the *distribution* of primitive roots for every integer n. If it turned out that in a certain interval there is

[1] Some properties of Euler's Totient Function can be found in the Annex.

[2] From now on we will refer to primitive roots *modulo* n as only "primitive roots" if there is no risk of confusion.

a bigger density of primitive roots, then this interval would be more suitable[3] to start checking possible primitive roots. Consequently, the aim of this work is to study the distribution of primitive roots and hence to determine which is the best way to try possible candidates.

In order to do so, we have written different versions of a C++ program to find primitive roots. We divided each n into a certain number of equal intervals and computed the number of primitive roots that fall into each interval, as well as the percentages of primitive roots in each interval. In addition, we carried out a statistical study to find out whether the differences observed among those percentages are statistically significant or not. Moreover, we computed the percentage of numbers n for which the smallest primitive root falls between 1 and 10.

After having found one primitive root for a number n, all the others can be obtained in a purely algebraic way using roots of unity in the complex plane. Since roots of unity can be represented geometrically as vertices of regular polygons, we were led to drawing primitive roots as points in a circle. By doing so, the statistical distribution discovered in our first calculations could be visualized. Moreover, roots of unity are given by cyclotomic polynomials [12]. We saw that cyclotomic polynomials can be used to find primitive roots for each n and realized that cyclotomic polynomials for numbers n of the form $2p$ and p^k can be computed without need of recursion. In conclusion, we quantified the effectiveness of different ways to find primitive roots and proposed our own method for optimally finding primitive roots according to the results of our study.

2 Methods for Finding Primitive Roots

The least effective way to find primitive roots for a given integer n is the one explained in the Introduction. The following is a better method [5]:

1. Compute $\phi(n)$.
2. Decompose $\phi(n)$ into its prime factors. Then $\phi(n) = p_1^{q_1} p_2^{q_2} \cdots p_k^{q_k}$.
3. Choose a possible primitive root g and keep raising it to $\frac{\phi(n)}{p_1}$, $\frac{\phi(n)}{p_2}$, etc., until $\frac{\phi(n)}{p_k}$.
4. If any of these values is congruent to 1 modulo n then g is not a primitive root. Otherwise it is.

This way we have reduced the number of exponents that we have to try from $n-1$ to the number of different prime factors of $\phi(n)$. However, the number of candidates that we have to try remains the same. Table 1 gives an example of the primitive roots for n smaller than 20.

Number	Primitive Roots	Number	Primitive Roots
2	1	10	3, 7
3	2	11	2, 6, 7, 8
4	3	13	2, 6, 7, 11
5	2, 3	14	3, 5
6	5	17	3, 5, 6, 7, 10, 11, 12, 14
7	3, 5	18	5, 11
9	2, 5	19	2, 3, 10, 13, 14, 15

Table 1: Primitive roots for integers smaller than 20

[3] When we say *more suitable* we mean that we have to check less possible primitive roots until we encounter one.

3 Collected Data

Since we wanted to see the distribution of primitive roots for a large amount of numbers, we programmed several C++ codes. In order to find the primitive roots for every number we first wrote a program using the least effective method; then we used the method explained in the previous section, and finally we improved this last program with some details: we realized that the code only needed to search for primitive roots for numbers of the form 2, 4, p^k and $2p^k$, hence we used the Sieve of Eratosthenes [6] in order to find prime numbers and decompose each number into its prime factors faster. Moreover, whilst writing the other programs we also noticed that $\phi(p^k)$ equals $\phi(2p^k)$ and consequently we only computed it once[4].

Since in the beginning we did not know how this distribution would look like, we decided to try to divide each number into 3, 4, 5 and 6 intervals. We ran our program for integers up to 500, 1000 and 2000. Firstly we fixed 3 intervals and integers until 500. Then, for each number n of the form 2, 4, p^k and $2p^k$ from 2 to 500 we divided n into 3 intervals and we saw where the primitive roots fell. We kept adding the number of roots falling in a certain interval for every number, and finally we obtained for all the numbers until 500 how many primitive roots were in the first, second and third intervals. Then we repeated this procedure for the other number of intervals as described above. For instance, when checking 25 we would have obtained the following distribution with 5 intervals:

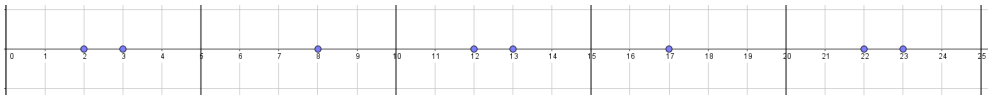


Figure 1: Example of distribution of the primitive roots for 25 in 5 intervals

In this case, we would add 2 to the number of primitive roots that lie in the first interval, 1 in the second one, etc. Using this method we collected data and observed that the primitive roots were not distributed equally among the different intervals. We also noticed that the bigger the maximum number, the more significant became the difference between the intervals. The following histograms represent the total number of primitive roots for each number until 500, 1000 and 2000 for 6 intervals:

[4]Further details on the Sieve of Eratosthenes and a comparison of the three computational times can be found on the Annex.

Figure 2: Primitive roots lying in 6 intervals for numbers until 500, 1000 and 2000

Moreover, we observed that the differences between the intervals became less relevant as the number of intervals increased. For instance, for numbers until 2000 we have the following distributions for 3, 4, 5 and 6 intervals:

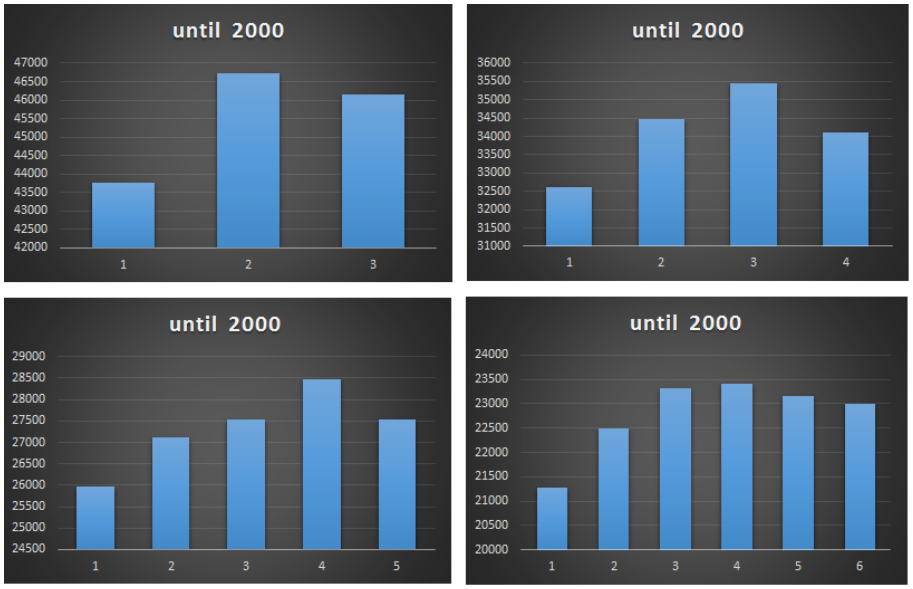


Figure 3: Amount of primitive roots until 2000 for 3, 4, 5 and 6 intervals

For these reasons we decided to keep working with 4 intervals and up to the first 2000 integers. Moreover, 4 intervals were adequate so that we could link primitive roots with roots of unity in the unit circle (see section 8) since we also have 4 quadrants. However, we realized that if we kept adding the number of primitive roots in each interval then the distribution of roots for small integers would become irrelevant, since numbers close to 2000 have a much larger number of primitive roots than numbers close to 1. Therefore, we decided to collect again these data but instead of adding the number of primitive roots computed for each number the percentage of primitive roots that lie in each of the 4

intervals. This way we would obtain the ratio of primitive roots falling in each interval. For instance, if a number had 5 roots out of 20 in the first interval, then there would be a 25% of the primitive roots in that interval. This way we were avoiding the inaccuracy of ignoring the primitive roots of small integers. We then computed the average percentage of primitive roots in each interval for all the numbers until 1000 and we obtained the data shown in Table 2:

	Interval 1	Interval 2	Interval 3	Interval 4
Percentages	22,57%	25,82%	27,67%	23,93%

Table 2: Average of the percentage of primitive roots that lie in each interval using ratios

It is also interesting to compare these percentages with the percentages obtained using the method of adding the primitive roots, to see until what extent ignoring the primitive roots of small integers would have an effect. For 4 intervals and up to the first 2000 numbers we obtained the following results:

	Interval 1	Interval 2	Interval 3	Interval 4
Percentages	23,86%	25,23%	25,94%	24,97%

Table 3: Average of the percentage of primitive roots that lie in each interval by adding the number of roots in each interval

We see that the percentages are quite different, which confirms that it was important to adjust our method in order to obtain non-biased data.

4 Statistical Significance

When summarizing our data we realized that there was a larger number of primitive roots falling in the third interval. However, we needed to check if the difference between the percentages was statistically significant using the data shown in the previous section. For this reason we decided to use p-value hypothesis testing [7]. This testing method is used to check if a hypothesis is true or not. Hence, in this case, if the distribution of primitive roots were equal in the 4 intervals then the percentage in each of them should be 25%. We want to see if the percentage of roots in the third interval is significantly larger than 25%. Hence, this test was the only one that was defined for cases in which we know the value each interval should have and suitable to compare means. First we have to choose our null hypothesis H_0, which states that the mean in the third interval equals 25. That is: $\bar{x} = \mu_0$ where μ_0 equals 25. We want to see if we can reject this hypothesis. Our alternative hypothesis states that $\bar{x} > \mu_0$, and therefore we are using a right-tailed test. We then need to compute the z-value, which is defined as [8]

$$z = \frac{\bar{x} - \mu_0}{\frac{s}{\sqrt{n}}} \tag{5}$$

where $\bar{x}$ is the mean of our sample, μ_0 is the value used for our null hypothesis, s is the standard deviation and n is the size of the sample. The standard deviation is defined as

$$s = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}. \tag{6}$$

Along with the average of the percentage of primitive roots lying in each interval shown in Table 3, we also computed the standard deviation of each interval using the previous definition. Thus, the standard deviation of the third interval is 8,38. Also recall that our sample size is 278, since this is the number of integers having primitive roots smaller than 1000. Hence, our z-value is

$$z = \frac{27,67 - 25}{\frac{8,38}{\sqrt{278}}} = 5,32. \tag{7}$$

We have $n - 1 = 277$ degrees of freedom and since we do not have more information we will take a significance level of 0,05 because this is the one that is taken in statistics when it is not possible do determine the significance in any other way. By consulting a p-value table we have checked that for 100 degrees of freedom and a significance level of 0,05 the p-value is 2,625, and for 1000 degrees of freedom it is 2,581. The p-value is the probability associated with our z-value. If our z-value is bigger than the p-value, then we have to reject the null hypothesis. Since for both degrees of freedom our z-value is indeed larger than the p-value, we can conclude that there is sufficient evidence in our data to claim that the percentage of roots in the third interval is higher than if the primitive roots were distributed equally in the 4 intervals. However, it was not easy to find a statistical test that would be suitable for this study. In this case, we have considered that the p-value test is appropriate although it does have a flaw: the size of the sample is not the same for all the numbers. Each integer has a different number of primitive roots and hence 27,67 is the average computed from percentages that come from numbers with different sample sizes. We thought of computing a weighted mean so that we could take into account the sample size of each number. However, we considered that this is not a regular method and we could not ensure that we would obtain a correct result, so we decided to keep our previous average because we also tried other statistical tests that could be suitable such as percentage comparison tests, and we always obtained that our result was significant.

5 First Root

Even if we saw that in the third interval there is a larger density of primitive roots, when studying them we observed that many numbers had either 2, 3, or 5 as the first primitive root. Hence, we wanted to determine how many numbers had a number smaller than or equal to 10 as the first primitive root. To do so, we used our C++ programs to know which was the first primitive root for each number up to 1000. We computed the frequency of each first primitive root and its corresponding percentage:

Number	Frequency	Percentage
3	84	30,10%
2	70	25,09%
7	30	10,75%
5	53	19%
6	12	4,30%
>10	30	10,75%

Table 3: Frequency and percentage of the first primitive roots for numbers up to 1000

We can also represent this data with a pie chart:

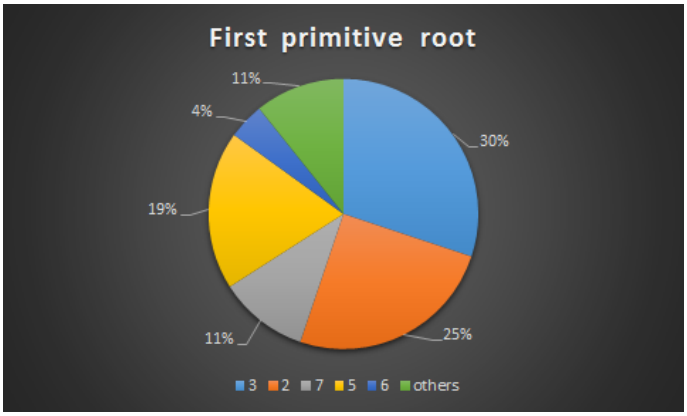


Figure 4: Frequency of each first primitive root of the numbers until 1000

Note that the sum of the frequencies is 279 because in the first 1000 integers only 279 actually have primitive roots. We then see that $89,24\%$ of the numbers have a number smaller than or equal to 10 as the first primitive root. It is also relevant that the highest first primitive root is 23 and it only appears once. We then conclude that the optimal intervals in which one should start looking for primitive roots are the first one because although it is the one with the lowest density of primitive roots we have observed that practically all numbers have a primitive root within the first 10 integers, and the third one because there is a higher density of primitive roots.

6 From an Algebraic Perspective

By using the results obtained in the previous sections we can find one primitive root in a quite effective way. But when we encounter one primitive root then we can infer the other ones. This is because there is a bijection [9] between the generators of the multiplicative group *modulo* n (the primitive roots) and the generators of the cyclic group of $\phi(n)$-th roots of unity [10]. These last generators, which we will call from now one *generators* in opposition to primitive roots, are very similar except that if a is a generator of $\mathbb{Z}_m$ then it means that we can obtain any other element of the group by repeatedly adding a to itself and working *modulo* m. For instance, 5 is a generator of $\mathbb{Z}_6$ because the multiples of 5 exhaust all the residues *modulo* 6:

$$\begin{aligned} &5+0\equiv 5 \qquad 5+5\equiv 4 \qquad 5+5+5\equiv 3 \qquad 5+5+5+5\equiv 2 \\ &\qquad\qquad\qquad\qquad 5+5+5+5+5\equiv 1 \qquad 5+5+5+5+5+5\equiv 0. \end{aligned} \tag{8}$$

We observe that the generators of $\mathbb{Z}_m$ are the numbers coprime to m, and so there will be $\phi(m)$ of them. We can represent this process geometrically:

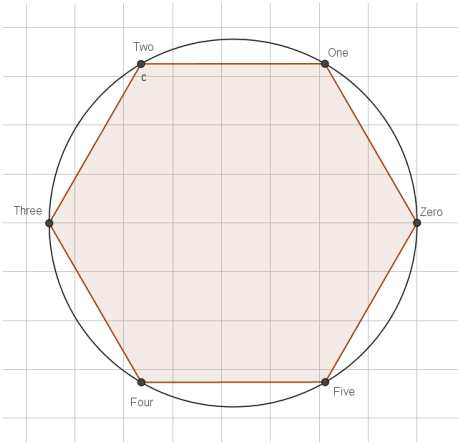


Figure 5: Geometric representation of 5 as a generator of $\mathbb{Z}_6$ (we start at vertex five, then we move to vertex four, and so on) in which we visit all vertices

Hence, $\phi(n)$ has $\phi(\phi(n))$ generators and therefore there are $\phi(\phi(n))$ primitive roots *modulo* n. We note that if a is a generator then $-a$ will also be one. This means that the distribution of the generators is symmetrical. Similarly, if g is a primitive root, then $1/g$ will also be one. The procedure is the following one:

1. We want to find all the primitive roots *modulo* n. We first find one, which we call g.
2. We compute $\phi(n)$ and find the numbers coprime to $\phi(n)$, namely k_1, k_2, etc.
3. The primitive roots of n are g^{k_1}, g^{k_2}, etc. *modulo* n.

Since we are studying the distribution of primitive roots and we know that the distribution of the generators is symmetrical, we thought it might be interesting to compare both distributions. For instance, when finding the primitive roots *modulo* 31 using the generators of $\mathbb{Z}_{30}$ (since $\phi(31) = 30$) we obtain the following bijection:

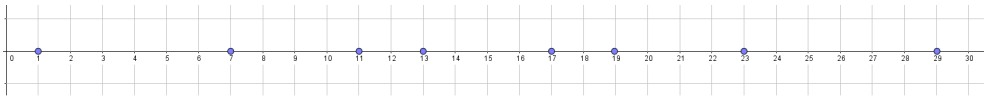


Figure 6: Distribution of the generators of 30 (numbers coprime to 30)

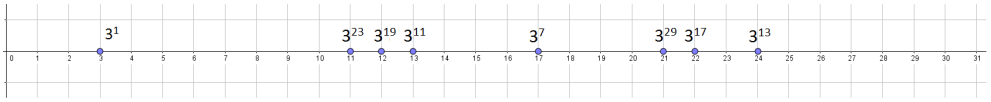


Figure 7: Distribution of the primitive roots of 31 using 3 as the first-found primitive root

Unfortunately, by trying several numbers we could not see any significant pattern on the map, except for the fact that in some intervals the primitive roots almost always appear to be consecutive (for instance, when n equals 31 we have that 11, 12 and 13 are consecutive roots and so are 21 and 22).

7 Relation with Roots of Unity in the Complex Numbers

In the previous section we saw that we can represent generators in regular polygons. This made us think that we could also represent primitive roots in the unit circle as this made us realize that there might be a connection with complex numbers that would give some clues about the distribution of primitive roots or how to find them. For instance, we can represent the primitive roots of 59 and 103 in the unit circle. We also see that they are distributed following the percentages that we found statistically:

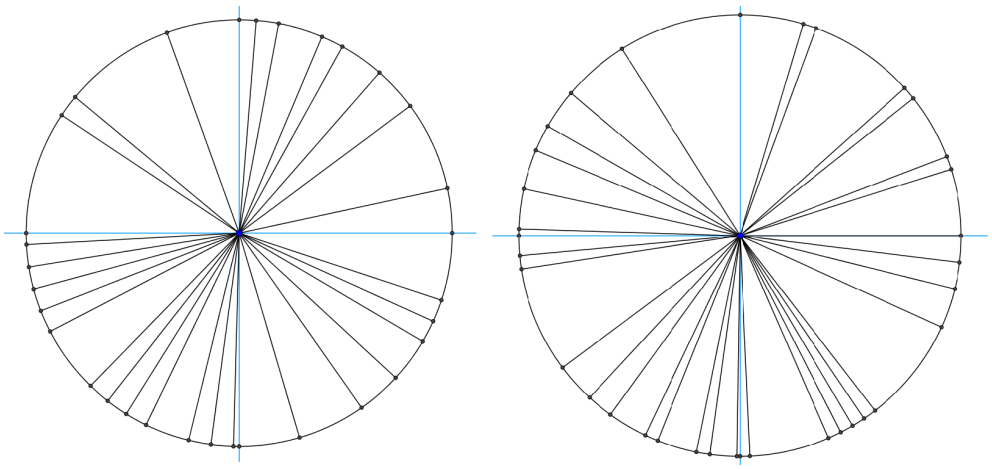


Figure 8: Distribution of the primitive roots of 59 and 103 in the unit circle

As we previously mentioned, we again observe that in some intervals many primitive roots appear to be consecutive. We also did a C++ program comparing the angles in which the generators and the primitive roots were placed, but again we did not spot any significant pattern.

8 Cyclotomic Polynomials

The fact that there exists a link between complex numbers and primitive roots led us to the following question: if for each integer n there is a polynomial, namely $x^n - 1$, whose roots are the roots of unity of n [11], is it possible to find a polynomial whose roots are the generators of n? The answer is yes and they are called *cyclotomic polynomials* [12]. Hence, the roots of the nth cyclotomic polynomial are the numbers coprime to n. The cyclotomic polynomials are found recursively by applying the following property and using the fact that $\Phi_1(x) = x - 1$:

$$\prod_{d|n} \Phi_d(x) = x^n - 1. \tag{9}$$

This way we can find the first cyclotomic polynomials:

$$\Phi_2(x) = \frac{x^2 - 1}{\Phi_1(x)} = \frac{x^2 - 1}{x - 1} = x + 1 \tag{10}$$

$$\Phi_3(x) = \frac{x^3 - 1}{\Phi_1(x)} = \frac{x^3 - 1}{x - 1} = x^2 + x + 1 \tag{11}$$

$$\Phi_4(x) = \frac{x^4 - 1}{\Phi_1(x)\,\Phi_2(x)} = \frac{x^4 - 1}{x^2 - 1} = x^2 + 1 \tag{12}$$

Hence, we realize that to compute the nth cyclotomic polynomial we need to find all the divisors of n and the cyclotomic polynomial of each divisor. For this reason it is not fast to compute cyclotomic polynomials. By trying to find more cyclotomic polynomials by hand we observed that there are three cases in which the nth cyclotomic polynomial can be computed directly without recursion:

1. If n is a prime, then $\Phi_p(x) = 1 + x + \cdots + x^{p-2} + x^{p-1}$ since this is the result of dividing $x^p - 1$ by $x - 1$ (because prime numbers have no divisors except 1).
2. If n is of the form $2p$, then $\Phi_{2p}(x) = 1 - x + x^2 - \cdots + x^{p-1}$.
3. If n is of the form p^k, then $\Phi_{p^k}(x) = 1 + x + \cdots + x^{n-2p} + x^{n-p}$.

This made us think if there was any relationship between cyclotomic polynomials and primitive roots. The answer is again yes, and by using the aforementioned property of cyclotomic polynomials we see that if we evaluate the primitive roots of n in $\Phi_{\phi(n)}(x)$ *modulo* n the result will be zero. For instance, we know that 2 is a primitive root of 5. Then, $\phi(5) = 4$ and the 4th cyclotomic polynomial is $x^2 + 1$. By substituting x by 2 in this polynomial *modulo* 5 we obtain 0:

$$\Phi_4(2) = 2^2 + 1 = 5 \equiv 0 \pmod 5. \tag{13}$$

Therefore, we conclude that another way of finding the primitive roots of n is by computing $\Phi_{\phi(n)}(x)$, evaluate x from 2 to $n - 1$ *modulo* n and see for which integers this is equal to 0. However, as we have seen, finding a cyclotomic polynomial recursively is quite long, so we conclude that this method is only effective if $\phi(n)$ is of the form $2p$ or p^k, because then we can find the cyclotomic polynomial instantly.

9 Probabilities

Once we have described all the methods of finding primitive roots that we have encountered and how primitive roots are distributed, we will compute some probabilities and computational times:

1. For the first method for finding primitive roots (the least effective one), for each candidate we had to try $n - 1$ different exponents.
2. For the second method that used $\phi(n)$ (explained in section 2), for each candidate we had to try as many exponents as different prime factors the number had. Using a C++ program we have computed the average number of different prime factors of the first 1000 numbers, and the result has been 2,13 different factors. When we ran the program up to the first 2000 numbers, the result was 2,23 different factors, and hence the result is still low. Consequently, for big n this second method is clearly much faster, since we only have to raise the candidate to an average of 2 different factors instead of $n - 1$.
3. Another way of finding possible candidates would be to choose them randomly. In order to compute if this method would be effective we have used our C++ programs to calculate the ratio between the number of primitive roots of n and n (which is the ratio between $\phi(\phi(n))$ and n). For instance, 11 has 4 primitive roots and hence if we chose candidates randomly we would have a chance of $\frac{4}{11}$, that is a 36,36% probability of being successful. With our program we have obtained that in a 30,53% of the cases we would be successful with our choice.

4. However, as we mentioned in section 4, if we try a candidate smaller than or equal to 10 we have a probability of 89,24% of being successful.

5. If we start trying numbers in the third interval, we have a 5,1% higher probability of finding one root than in the first interval, 1,85% higher than in the second interval, and 3,75% higher than in the fourth interval.

10 Conclusions

After studying primitive roots from a wide variety of perspectives and carrying out some statistical and algebraic studies, we can conclude that the primitive roots are not distributed equally, but instead if we evaluate the distribution of primitive roots in 4 intervals we see that there is a higher density of primitive roots in the third interval. We have computed with C++ programs the primitive roots of each of the numbers of the form 2, 4, p^k and $2p^k$ up to 1000 using the method explained with Euler's function $\phi(n)$ and we have obtained the following percentages: 22,57% in the first interval, 25,82% in the second one, 27,67% in the third one, and 23,93% in the fourth one. Hence, there are less primitive roots lying in the first interval and more primitive roots falling in the third interval. Using a p-value test we have concluded that there is statistical significance to support this evidence. We have also explained that there is a bijection between the generators of the cyclic group $\mathbb{Z}_{\phi(n)}$ (which are the numbers coprime to $\phi(n)$) and the primitive roots *modulo* n.

Moreover, we explained that it is possible to represent generators and primitive roots geometrically, but we did not find any relevant pattern between these figures except for the fact that the generators are symmetrical and the primitive roots appear to be consecutive in some intervals. Also, their similarities with the roots of unity in the complex numbers led us to the use of cyclotomic polynomials for finding primitive roots, because if we evaluate the primitive roots for n in $\Phi_{\phi(n)}(x)$ *modulo* n we obtain zero. Nonetheless, cyclotomic polynomials are found recursively, and so it is only possible to obtain directly a cyclotomic polynomial if n is of the form p, $2p$ or p^k. Finally, we saw that there is a 30,53% probability of having chosen an actual primitive root when trying possible candidates randomly, and a 89,24% probability of finding a primitive root between 1 and 10. For all these reasons and taking into account the distribution of the primitive roots that we have encountered, if we had to prepare a method for finding primitive roots of a given integer n effectively, it would be the following:

1. Compute $\phi(n)$. If $\phi(n)$ is of the form p, $2p$ or p^k write directly the $\Phi_{\phi(n)}(x)$ cyclotomic polynomial and evaluate it for all the numbers between 2 and $n-1$. If when evaluating a certain integer the polynomial returns 0 *modulo* n, then this integer is a primitive root. After finding $\phi(n)$ primitive roots, stop and these are them all.

2. If $\phi(n)$ is not of the form described before, start trying possible candidates to primitive roots either with 2 (since between 1 and 10 the probabilities of finding a primitive root are very high) or in the interval between $n/2$ and $3n/4$ (since we have seen that there is a higher density of primitive roots). To check if the number is a primitive root apply the method explained in section 2. In average it will only be needed to raise the candidate to two exponents to see if it is a primitive root or not (since the average number of different prime factors of integers below 2000 is 2,29).

3. After having found one primitive root g, compute $\phi(n)$ and find the numbers that are coprime to it. Then keep raising g to all the numbers coprime to $\phi(n)$ *modulo* n and this provides all the other primitive roots.

Although I believe that the aim of this project has been accomplished, some aspects of it could have been carried out differently. Firstly, although the C++ programs were improved in successive versions, they were still slow and so I could only test results up to the first 2000 integers. Since we wanted to find the distribution of primitive roots in general, it would have been better to obtain results for much larger integers since the results would have been more reliable. We could also have applied the statistical test to the percentages obtained by adding the primitive roots lying in each interval, although we considered that these results were not optimal since this method did not take into account the primitive roots of small integers.

Also, I would have investigated in greater detail other types of cyclotomic polynomials that can be found without recursion, since these are useful for finding primitive roots. Moreover, I should also take into consideration that if $\phi(n)$ is very large then in order to evaluate the $\phi(n)$-th cyclotomic polynomial too large numbers are needed. It would also have been interesting to continue comparing the bijection between generators and primitive roots using their geometrical representation, since it might have given us a clue on the reason why there seem to be more primitive roots in the third interval.

11 Annex

11.1 Euler's Totient Function

Euler's Totient Function, which has been used a lot during this work, counts the number of integers up to n that are coprime to n and it is denoted by $\phi(n)$. Coprime means that they do not share any prime factor. For instance, $\phi(10)$ equals 4 because there are only 4 numbers smaller than 10 and coprime with 10: 1, 3, 7 and 9. In order to compute $\phi(n)$ it is important to take into account the following properties:

1. $\phi(ab) = \phi(a)\phi(b)$ if a and b are coprime.
2. $\phi(p^k) = (p-1)p^{k-1}$.

Hence, by decomposing n into its prime factors it is relatively easy to compute $\phi(n)$. Table 4 shows the computation of $\phi(n)$ for the first natural numbers.

Number	Φ(n)	Number	Φ(n)
1	1	8	4
2	1	9	6
3	2	10	4
4	2	11	10
5	4	12	4
6	2	13	12
7	6	14	6

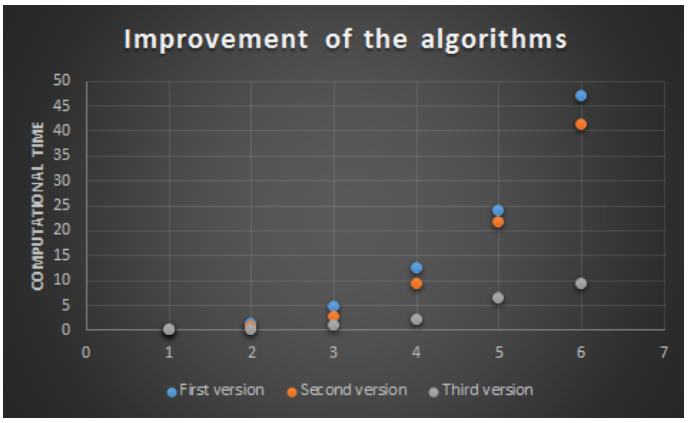
Table 4: Example of the value of $\phi(n)$ for the first natural numbers

11.2 Sieve of Eratosthenes

The Sieve of Eratosthenes is an algorithm that is used in order to find primitive numbers in an effective way. The regular way to find if a number is prime or not is by checking it is divisible by some integer smaller than the square root of this number. However, the Sieve of Eratosthenes finds prime numbers until n in a much faster way. First it erases all the multiples of 2 until n; then it erases all the multiples of 3 until n; then it erases all the multiples of 5 (because 4 is already gone) until n. It keeps repeating this process and so the numbers that remain are prime numbers because they do not have any divisor that could have erased them.

11.3 Comparison of computational times

As we previously explained, we have programmed three different codes in order to compute primitive roots. Each one incorporated something that made the code more effective. In the first version, the C++ code computed the primitive roots with the very slow way: in order to find the primitive roots of n it chose as possible candidates all numbers between 1 and $n-1$ and raised each candidate to the $n-1$ possible exponents *modulo* n and checked if there were $\phi(n)$ different remainders. In the second version we applied the faster method explained in section 2: instead of raising each candidate to $n-1$ exponents, we only raised each candidate to the number of different primitive factors of n (although the number of candidates needed is the same). In order to divide each number into prime factors we did it in the classic way: trying which numbers smaller than n actually divided n. Finally, for the third version we realized that only the numbers of the form 2, 4, p^k and $2p^k$ have primitive roots, and hence it was not needed to find the primitive roots of the other numbers because they do not have any. We also incorporated the Sieve of Eratosthenes in order to know which numbers are prime and to decompose each number into its prime factors. These two improvements were very effective. The following plot represents computational time of each version:



We observe that the improvement between versions one and two is not as remarkable as the difference between versions two and three. With this fact we can deduce that only finding the primitive roots of the numbers of the form 2, 4, p^k and $2p^k$ and using the Sieve of Eratosthenes is much better. However, all the versions have exponential computational times, and this is something that we should consider and improve in possible subsequent algorithms, because exponential means that the code is still slow.

11.4 C++ Programs

11.4.1 First program

```cpp
#include<iostream>
#include<vector>
#include<cmath>
#include<ctime>
using namespace std;

int main(){
    cout<<"Introdueix fins quin nombre: ";
    int m;
    cin>>m;
    for(int n=1; n<m; n++){
        int res;
        int cot = 0;
        for(int i=1; i<n; i++){
                vector<int> v(n, 0);
                v[0]=1;
            for(int j=1; j<n; j++){
                    res = 1;
                for(int q=0; q<j; q++){
                    res = (res*i)%n;
                }
            v[res] = 1;
            int tru = 0;
            for(int w=0; w<v.size(); w++){
                if(v[w]==0){
                    tru = 1;
                }
            }
            if((tru == 0) and (cot==0)){
                cout<<endl<<"numero: "<<n<<endl;
                cot++;
            }
            if((tru == 0)and(cot!=0)){
                cout<<i<<"   ";
            }
            }
    }
```

11.4.2 Second program

```cpp
#include<iostream>
#include<vector>
#include<cmath>
#include<set>
using namespace std;

vector<int> fact(int n, int k, vector<int> factors){
     if(n==1){
        return factors;
    }
    else{
        if(n!=1){
            if(n%k==0){
            n = n/k;
            factors.push_back(k);
            return fact(n, k, factors);
        }
        else{
            k++;
            return fact(n, k, factors);
        }
    }
    }
}

int fi(int n, int k, int phi, int exp, vector<int> factors){
    if(n==1){
        for(int i=1; i<exp; i++){
                phi = phi*k;
            }
        if(exp!=0){
                phi = phi*(k-1);
            }
        return phi;
    }
    else{
        if(n!=1){
```

```
            if(n%k==0){
            n = n/k;
            factors.push_back(k);
            exp++;
            return fi(n, k, phi, exp, factors);
        }
        else{
            for(int i=1; i<exp; i++){
                phi = phi*k;
            }
            if(exp!=0){
                phi = phi*(k-1);
            }
            k++;
            return fi(n, k, phi, 0, factors);
        }
    }
    }
}

int main(){
    cout<<"Introdueix fins quin nombre: ";
    int m;
    cin>>m;
    for(int n=1; n<m; n++){
        int primer = 0;
        for(int t=2; t*t<=n; t++){
            if(n%t==0){
                primer = 1;
                break;
            }
        }
        if(primer==0){
             int res;
        int comp = 0;
        int cot = 0;
        vector<int> factors;
```

```
        vector<int> factors;
        int euler = fi(n, 2, 1, 0, factors);
        factors = fact(euler, 2, factors);
        for(int i=1; i<n; i++){
             comp = 0;
            for(int j=0; j<factors.size(); j++){
                 res = 1;
                for(int q=1; q<=(euler/factors[j]); q++){
                    res = (res*i)%n;
                }
            if(res==1){
                comp = 1;
                break;
            }
            }
            if((cot==0)and(comp==0)){
                cout<<endl<<"numero: "<<n<<endl;
                cot++;
            }
            if(comp==0){
                cout<<i<<"   ";

            }
        }
        }
        if(primer==1){
            int cot = 0;
            for(int i=1; i<n; i++){
                set<int> s;
            for(int j=1; j<n; j++){
                    int res = 1;
                for(int q=0; q<j; q++){
                    res = (res*i)%n;
                }
            int fal = 0;
            for(int y=2; y<=i; y++){
                if((n%y==0)and(i%y==0)){
```

```
                    fal = 1;
                }
            }
            if(fal==0){
                s.insert(res);
            }
            }
            vector<int> factors;
            int euler = fi(n, 2, 1, 0, factors);
            if((s.size()==euler) and (cot==0)){
                cout<<endl<<"numero: "<<n<<endl;
                cot++;
            }
            if((s.size()==euler)and(cot!=0)){
               cout<<i<<"   ";
            }
        }

    }
}
}
```

11.4.3 Third program

```
#include<iostream>
#include<vector>
#include<cmath>
#include<set>

using namespace std;

vector <bool> v;
vector <int> primers;
vector <vector <int> > factors;
vector <vector <int> > factorsrepet;

void garbell(){
    v[0]=false;
    v[1]=false;
    for(int i=2; i<v.size(); i++){
        if(v[i]){
            primers.push_back(i);
            for(int h=2; h*i<v.size(); h++){
                v[i*h] = 0;
                factors[i*h].push_back(i);
                int repet = i*h;
                while(repet%i==0){
                    repet = repet/i;
                    factorsrepet[i*h].push_back(i);
                }
            }
        }
    }
}

int main(){
    int n;
    cin >> n;
    v = vector <bool> (n, true);
    primers = vector <int> ();
    factors = vector <vector <int>> (n);
```

```
    factorsrepet = vector <vector <int>> (n);
    garbell();
    for(int i=1; i<primers.size(); i++){
        int pot = primers[i];
        int k = 1;
        for(int q=1; q<k; q++){
            pot = pot*pot;
        }
        while(pot<=n){
            cout<<endl<<"numero: "<<pot<<"    ";
            if(k==1){
                //cout<<endl;
                int euler=pot-1;
            for (int arr=2; arr<pot; arr++){
                int comp=0;
                for(int s=0; s<factors[euler].size(); s++){
                    int res = 1;
                    for(int m=1; m<=(euler/factors[euler][s]); m++){
                        res = (res*arr)%pot;
                    }
                    if(res==1){
                        comp = 1;
                        break;
                    }
                }
                    if(comp==0){
                        cout<<arr<<"   ";
                    }
            }
            if(2*pot<=n){
                cout<<endl;
                int dospot = 2*pot;
                cout<<"numero: "<<dospot<<"    ";
                int euler = primers[i]-1;
                for(int q=1; q<k; q++){
                    euler = euler*primers[i];
                }
```

```
                int cot = 0;
                for(int i=2; i<dospot; i++){
                    set<int> s;
                    for(int j=1; j<dospot; j++){
                            int res = 1;
                        for(int q=0; q<j; q++){
                            res = (res*i)%(dospot);
                        }
                    int fal = 0;
                    for(int y=2; y<=i; y++){
                        if(((dospot)%y==0)and(i%y==0)){
                            fal = 1;
                        }
                    }
                    if(fal==0){
                        s.insert(res);
                    }
                    }
                    if((s.size()==euler)){
                        cout<<i<<"    ";
                    }
            }
        }
            }
            if(k!=1){
                //cout<<endl;
                int euler = primers[i]-1;
                for(int q=1; q<k; q++){
                    euler = euler*primers[i];
                }
                int cot = 0;
                for(int i=2; i<pot; i++){
                    set<int> s;
                    for(int j=1; j<pot; j++){
                            int res = 1;
                        for(int q=0; q<j; q++){
                            res = (res*i)%pot;
```

```
                    }
                int fal = 0;
                for(int y=2; y<=i; y++){
                    if((pot%y==0)and(i%y==0)){
                        fal = 1;
                    }
                }
                if(fal==0){
                    s.insert(res);
                }
                }
                if((s.size()==euler)){
                    cout<<i<<"    ";
                }
            }
             if(2*pot<=n){
                cout<<endl;
                int dospot = 2*pot;
                cout<<"numero: "<<dospot<<"    ";
                int euler = primers[i]-1;
                for(int q=1; q<k; q++){
                    euler = euler*primers[i];
                }
                int cot = 0;
                for(int i=2; i<dospot; i++){
                    set<int> s;
                    for(int j=1; j<dospot; j++){
                            int res = 1;
                        for(int q=0; q<j; q++){
                            res = (res*i)%(dospot);
                        }
                    int fal = 0;
                    for(int y=2; y<=i; y++){
                        if(((dospot)%y==0)and(i%y==0)){
                            fal = 1;
                        }
                    }
```

```
                    if(fal==0){
                        s.insert(res);
                    }
                    }
                    if((s.size()==euler)){
                        cout<<i<<"    ";
                    }
                }
            }
        }
            k++;
            for(int q=1; q<k; q++){
                pot = pot*pot;
            }
        }
    }

    }
```

11.4.4 Other codes

The codes for computing the first primitive roots and the distribution of primitive roots in each interval are modifications of the third version of the program and hence we do not think it is necessary to include them.

References

[1] P. Fannon, B. Woolley, S. Ward, V. Kadelburg, *Mathematics Higher Level for the IB Diploma. Discrete Mathematics*, Cambridge, 2013.

[2] J. Grané, *Sessions de preparació per a l'olimpíada matemàtica*, Societat Catalana de Matemàtiques, 2004.

[3] M. Hunter, *The Least Square-free Primitive Root Modulo a Prime*, Australian National University, 2016.

[4] T. M. Apostol, *Introduction to Analytic Number Theory*, Undergraduate Texts in Mathematics, Springer-Verlag, New York, Heidelberg, 1976.

[5] K. D. Crisman, *Number Theory: In Context and Interactive*, 2017.

[6] D. N. Cox, *Visualizing the Sieve of Eratosthenes*, Notices of the AMS, vol. 55, no. 5, May 2008.

[7] P. Fannon, B. Woolley, S. Ward, V. Kadelburg, *Mathematics Higher Level for the IB Diploma. Statistics and Probability*, Cambridge, 2013.

[8] J. H. Abramson, *Making Sense of Data*, Oxford, 1994.

[9] P. Fannon, B. Woolley, S. Ward, V. Kadelburg, *Mathematics Higher Level for the IB Diploma. Sets, Relations and Groups*, Cambridge, 2013.

[10] H. W. Lenstra, *Finding Isomorphisms Between Finite Fields*, Mathematics of Computation, vol. 56, no. 193, American Mathematical Society, 1991.

[11] I. Wazir, T. Garry et al., *Mathematics Higher Level for the IB Diploma*, Chapter 10, Pearson, 2012.

[12] A. Al-Kateeb, H. Hong, E. Lee, *Block Structure of Cyclotomic Polynomials*, 2017.