Chapter 16:
Leave to Chance

Panos Louridas

Athens University of Economics and Business
Real World Algorithms
A Beginners Guide
The MIT Press

Outline

1. General
2. Random Numbers
3. Random Sampling
4. Power Games
5. The Monte Carlo Method
6. Monte Carlo Banzhaf
7. Searching for Primes

General

- Usually we try to minimize the effect of chance in our lives.
- We consider it good to be able to say that “we have left nothing to chance.”
- However, chance can be the solution to a whole range of important problems.

Randomness

- Something is *random* when it is *unpredictable*.
- A coin toss is random, as there is a 50–50 chance to come heads or tails.
- *Randomness* is the lack of any regularity in a series of events or data.

Random

- We say that something is random when there are not regularities or patterns in it, how much we try to find them.
- White noise is random.
- The throws of dice are random.
- Brownian motion is random.

Samples

- In a survey we take a *sample* of the surveyed population.
- We want this to be a *representative sample*.
- That is not always easy.
- For example, *survivorship bias* occurs when our sample includes only the members of the population that are still around.
- To avoid such problems, we try to work with a *random sample*.

Randomized Algorithms

- To find a random sample, we must use an algorithm that uses some kind of randomness.
- As the algorithm will work, to a degree, randomly, it will be a *randomized algorithm*.
- A randomized algorithm will not always follow the same set of steps for the same input.
- The question is: can we have randomized algorithms that work well, while also knowing the probability that they may not work well?
- Can we control the behavior of such algorithms by tweaking their parameters?

Outline

1. General
2. Random Numbers
3. Random Sampling
4. Power Games
5. The Monte Carlo Method
6. Monte Carlo Banzhaf
7. Searching for Primes

Randomness in Computers

- Randomized algorithms work with randomness, which comes from *random numbers*.
- But how can we produce random numbers in a computer?
- Obviously, a deterministic algorithm will never have a random result.

Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.—John von Neumann, 1951

Random Number Generators

- To produce random numbers we use *random number generators*.
- The only real random number generators, called *True Random Number Generators (TRNGs)*, use a random physical process as the source of their randomness.
- Such a process can be a Geiger counter, a photon detector behind a semi-transparent mirror, a detector of atmospheric radio noise, etc.
- A TRNG can be incorporated into a computer.
- However, it may not be always available, and it may not be able to produce random numbers with the required rate.

Pseudorandom Number Generators

- If we cannot use a TRNG, we have to resort to a *Pseudorandom Number Generator (PRNG)*.
- A PRNG produces numbers that *appear* random, even if in fact they are not.

Pseudorandom Numbers and the Uniform Distribution

- A common requirement is for the pseudorandom numbers to follow the *uniform distribution*.
- A set of numbers follows the uniform distribution if each one of them appears in the set with the same probability.
- However, this is not enough for our purposes.
- For example, the numbers:

$$1, 2, \ldots, 10, 1, 2, \ldots, 10, 1, 2, \ldots, 10, \ldots$$

follow the uniform distribution, but they do not appear random at all.

Statistical Randomness Checks

- For that purpose, we want the numbers produced by a PRNG to pass certain *statistical tests*.
- These tests indicate the numbers in a sequence deviate significantly from what a truly random sequence of numbers would be.

Linear Congruential Method

- A simple PRNG that has been used for a long time is the *linear congruential method*.
- That method produces a series of numbers with the formula:

$$X_{n+1} = \left(aX_n + c\right) \bmod m, \quad n \geq 0$$

- The sequence starts by feeding the formula with an initial value X_0, called *seed*.

Linear Congruential Algorithm

Algorithm: Linear Congruential Random Number Generator.

`LinearCongruential`$(x) \longrightarrow r$

Input: x, a number $0 \leq x < m$

Data: m the modulus, $m > 0$

a the multiplier, $0 < a < m$

c the increment, $0 < c < m$

Output: r, a number $0 \leq r < m$

1 $r \leftarrow (a \times x + c) \bmod m$

2 **return** r

Implementation

- When implementing the algorithm, we do not usually pass x in each call.
- Typically, there are two functions.
- There is a function `Seed(`s`)`, which feeds the seed to the method.
- Then there is a function `Random()` that carries out the calculation and keeps the result in a hidden variable, so that it can use it in the next call.

The Importance of the Seed

- The pseudorandom numbers that will be produced will depend on the initial value of x, the seed.
- That is not bad, because it allows us to convert a randomized program to a deterministic one.
- It is useful when checking and debugging the problem, when we want to be able to predict the outcome and we do not want to have different behavior every time.

The Parameters a, m, c

- The values a, m, c are very important.
- The method cannot produce more than m pseudorandom numbers.
- If at some point it produces a number it has already emitted before, it will start repeating the same values.
- So, the method has a *period*.
- We must select a, m, c so that the period is as long as possible (even equal to m) while the pseudorandom numbers must follow the uniform distribution.

The Parameters a, m, c (Contd.)

- For example, if we give $s = 0$, $m = 10$, $a = 3$, and $c = 3$, we will get::

$$3, 2, 9, 0, 3, 2, 9, 0, \ldots$$

- To obtain the maximum period possible, equal to m, the numbers a, m, c must meet the following conditions:
  1. m and c must be relatively prime.
  2. $a - 1$ must be divided by all the prime factors of m.
  3. $a - 1$ must be divisible by 4, if m is divisible by 4.
- Usually we pick some numbers that we already know that they meet the conditions, such as $s = 2^{32}$, $a = 32310901$, c is odd, and m is a power of 2.

Range of Values

- The linear congruential method returns numbers between 0 and $m - 1$, that is, in the range $[0, m - 1]$.
- If we want pseudorandom numbers in another range, for example, $[0, k]$, we can multiply the result by $k/(m - 1)$.
- If we want numbers from 0 to 1, we divide by $m - 1$.
- We often need numbers in the range $[0, 1)$, which we get by dividing with m.

xorshift64*

- Over the last few years, other PRNGs have been proposed as being better than the linear congruential method, because they pass more tests of randomness.
- One of them, with the advantage that it is very fast, is xorshift64* (read xor shift 64 star).
- Like the linear congruential method, it produces a new pseudorandom number each time it is called, getting as input the previous number it produced.
- The first time it must be fed with a non-zero seed.
- The xorshift64* algorithm produces 64 bits numbers.

The xorshift64* Algorithm

Algorithm: xorshift64*.

`XORShift64Star`$(x) \longrightarrow r$

Input: x, a 64-bit integer different than 0

Output: r, a 64-bit number

1 $x \leftarrow x \oplus (x \gg 12)$

2 $x \leftarrow x \oplus (x \ll 25)$

3 $x \leftarrow x \oplus (x \gg 27)$

4 $r \leftarrow x \times 2685821657736338717$

5 **return** r

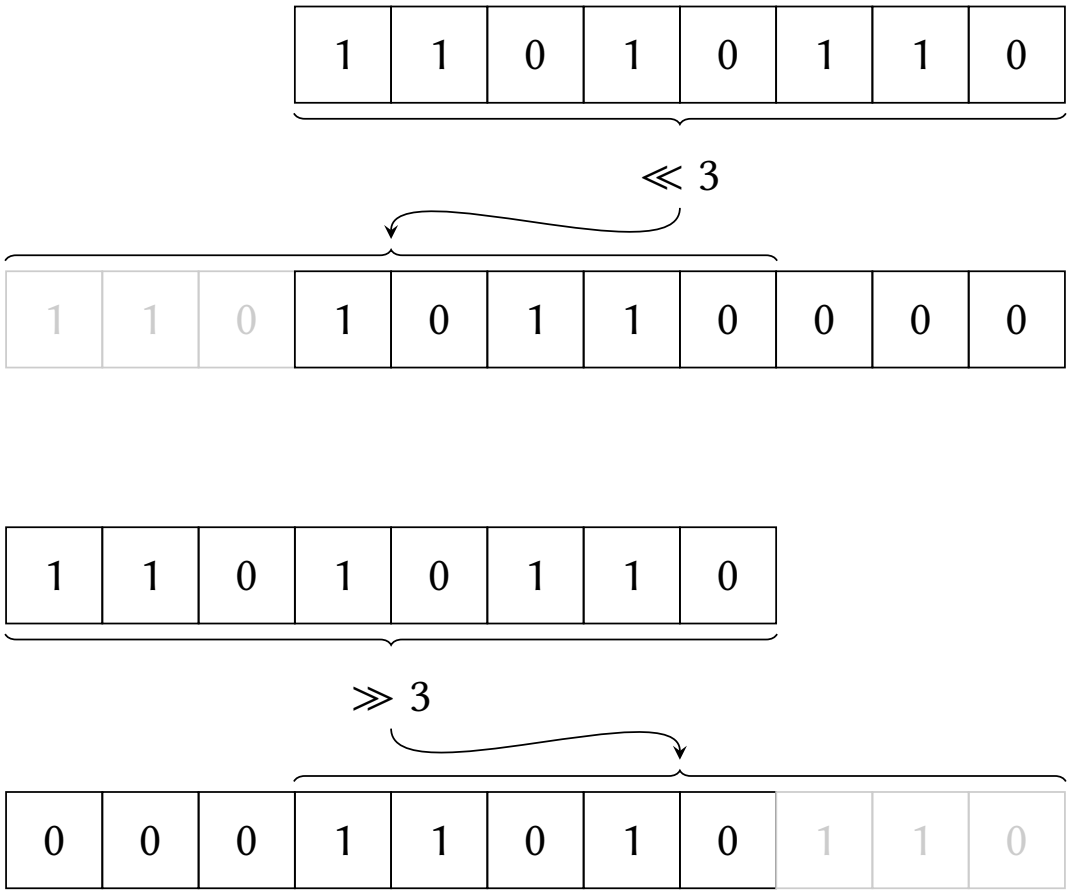
Magic Numbers

- The number 26858216577736338717 appearing in the algorithm is a *magic number*.
- A magic number in an algorithm or a program is a number without apparent meaning.
- In our case, the number is special because it ensures that the output does look random.

Shift Operators

- The xorshift64* algorithm uses the $\ll \gg$ *shift operators.*
- The expression $x \ll a$ means that we will shift the bits of x to the left by a positions, for example, $1110 \ll 2 = 1000$.
- The expression $x \gg a$ means that will shift the bits of x to the right by a positions, for example, $1101 \gg 2 = 0011$.

Shift Operators Illustrated



xorshift64* and xorshift1024*

- xorshift64* is very fast.
- Moreover, it produces numbers with good randomness characteristics.
- Its period is $2^{64} - 1$.
- If we want an even longer period, we can use its big sibling, xorshift1024*.

The xorshift1024* Algorithm

Algorithm: xorshift1024*.

`XORShift1024Star`$(S) \rightarrow r$

Input: S, an array of 16 unsigned 64-bit integers

Data: p, a number initially set to 0

Output: r, a 64-bit random number

1 $s_0 \leftarrow S[p]$

2 $p \leftarrow (p + 1)\ \&\ 15$

3 $s_1 \leftarrow S[p]$

4 $s_1 \leftarrow s_1 \oplus (s_1 \ll 31)$

5 $s_1 \leftarrow s1 \oplus (s_1 \gg 11)$

6 $s_0 \leftarrow s_0 \oplus (s_0 \gg 30)$

7 $S[p] \leftarrow s_0 \oplus s1$

8 $r \leftarrow S[p] \times 1181783497276652981$

9 **return** r

Explanation

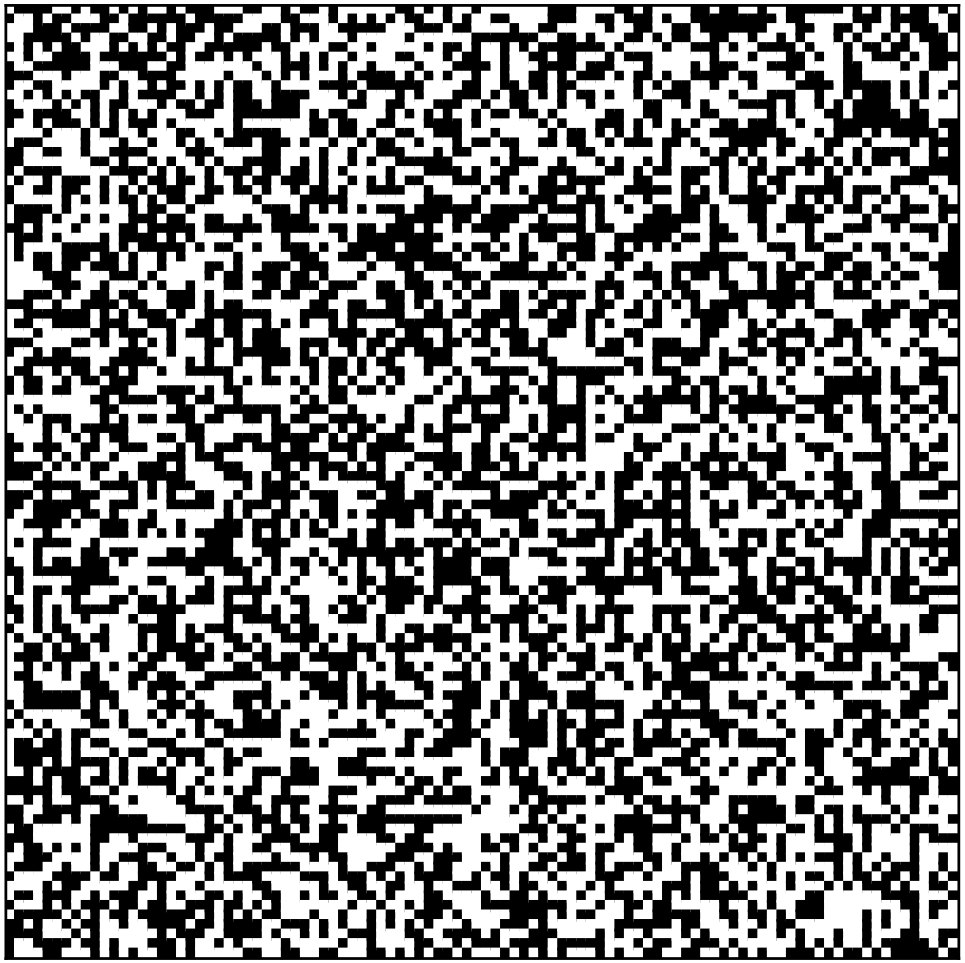
- Compared to xorshift64*, xorshift1024* has a much longer period: $2^{1024} - 1$.
- Moreover, the number it produces pass even more tests of statistical randomness.
- Its seed are the initial 16 numbers of S, produced from xorshift64*. The contents of S (in particular, $S[p]$) change with each call.
- The name of the algorithm is due to the fact that it uses an array of 16 places, each one of which has 64 bits, so $16 \times 64 = 1024$.
- Note line 2, which guarantees that p takes on the values from 0 up to and including 15.

Designing PRNGs

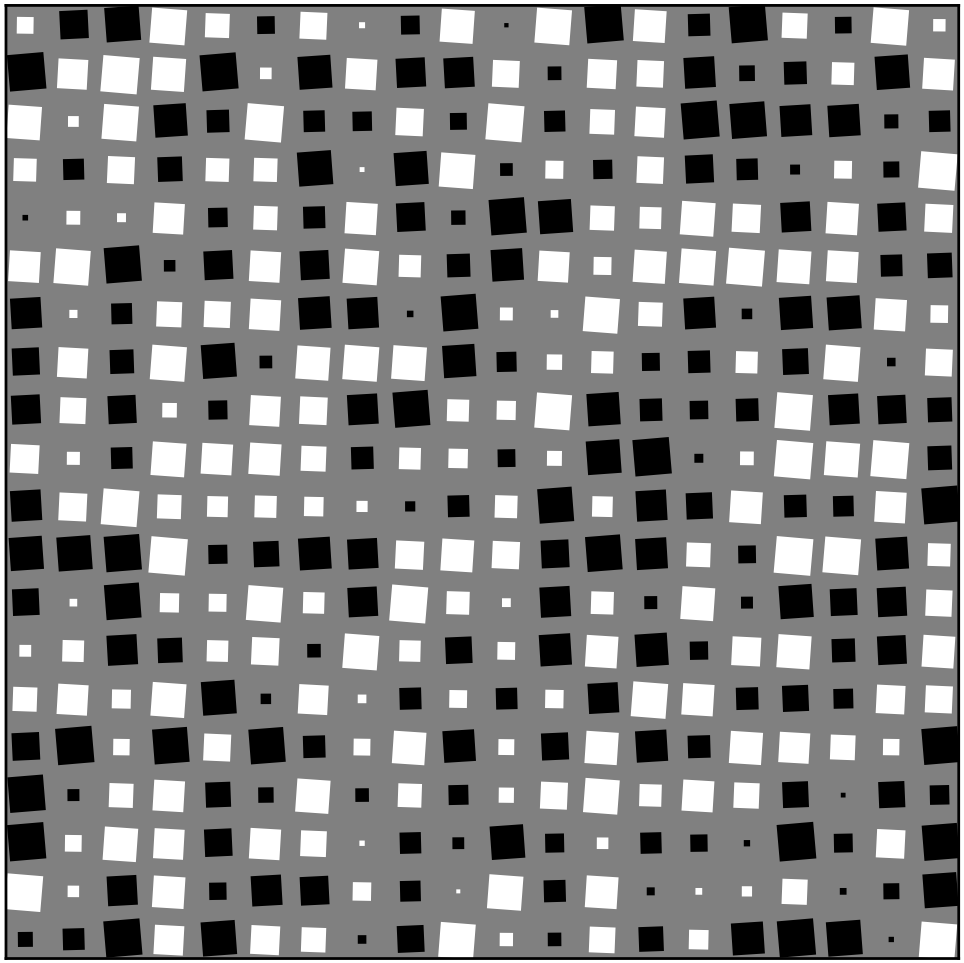
- As we can see, the design of a PRNG is not an easy task.
- It requires a lot of work and many tests.

random numbers should not be generated with a method chosen at random.—Donald Knuth

10,000 Random Bits



Random Image



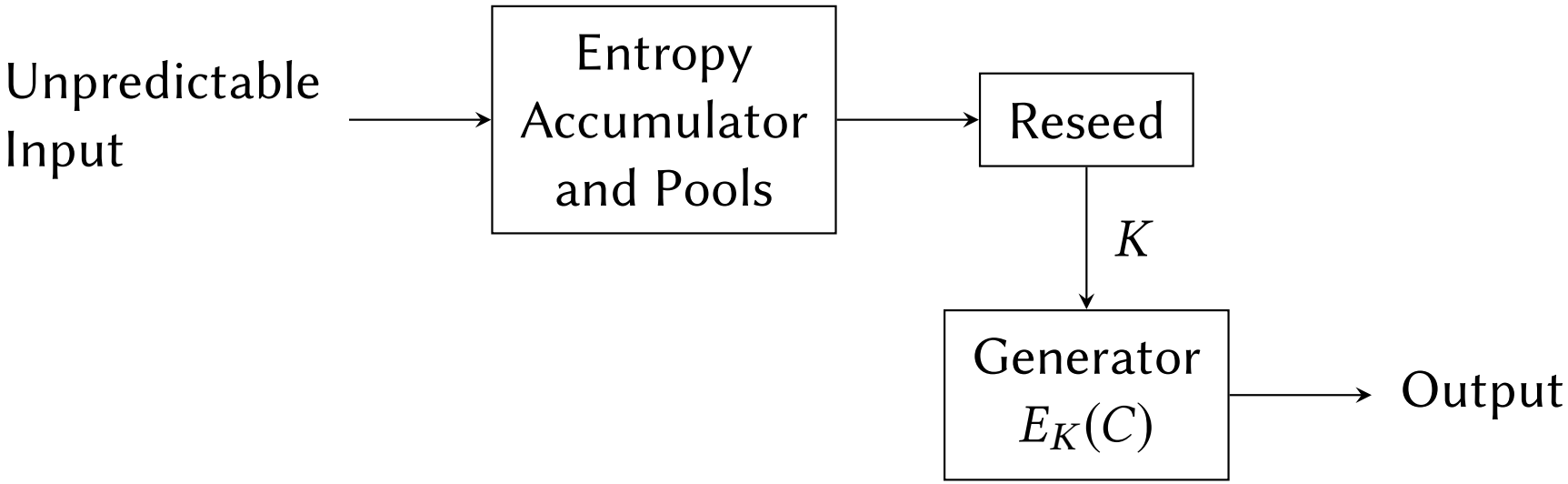
Random Numbers and Cryptography

- The PRNGs that we have seen till now *are not suitable for cryptographic use*.
- In many cryptographic protocols we need random numbers, either as keys, or as *nonces*, arbitrary numbers that are to be used only once.
- To be suitable for cryptographic use, pseudorandom numbers:
  - Must pass statistical randomness tests.
  - There must be no polynomial time algorithm that can start predicting the coming numbers.
  - There must be no way that, given the state of the algorithm, one can work backward and deduce the previously generated numbers.

Cryptographically Secure PRNGs

- In cryptography we use specially crafted PRNGs.
- These are called *Cryptographically Secure Pseudorandom Number Generators (CSPRNGs)*.

The Fortuna CSPRNGs



Outline

1 General

2 Random Numbers

3 Random Sampling

4 Power Games

5 The Monte Carlo Method

6 Monte Carlo Banzhaf

7 Searching for Primes

Simple Method (Wrong)

- Suppose that we have a population of size n and we want a random sample of size m.
- A simple idea is to take each member of the population and include it in our sample with probability m/n.
- That is *wrong*.
- In this way we will get a sample of size m *on average*, not *each time*.

Basic Idea

- Suppose that we have already selected some items.
- We now consider whether to select the next one.
- If we have already selected a lot of items and we still have a lot to examine, the probability of selecting the next one must be *low*.
- If we have selected few items and we only have few remaining ones to examine, the probability of selecting the next one must be *high*.
- So, we want to adjust the probability of selecting an item depending on the items we have selected and the remaining items.

Selection Sampling (1)

- Suppose we have examined t items and we have selected k of them for our sample.
- That means that there are $n - t$ items we have not yet examined.
- It also means that we have to add into our sample $m - k$ items.
- In how many ways can this happen?

Selection Sampling (2)

- The number of ways we can do that, w_1, is the number of ways that we can select $m - k$ out of $n - t$ items:

$$w_1 = \binom{n-t}{m-k}$$

- Similarly, if we have examined $t + 1$ items and selected $k + 1$ items, the process can continue in:

$$w_2 = \binom{n-t-1}{m-k-1}$$

different ways.

Selection Sampling (3)

- Therefore, the probability of going from t and k to $t+1$ and $k+1$ is:

$$\frac{w_2}{w_1} = \binom{n-t-1}{m-k-1} \Big/ \binom{n-t}{m-k} = \frac{m-k}{n-t}$$

- Therefore, if we have selected the first k items from the first t, the $(k+1)$th item will be selected with probability $(m-k)/(n-t)$.
- The algorithm we get by applying this rule is called *selection sampling*.

Selection Sampling Algorithm

Algorithm: Selection sampling.

`SelectionSampling`$(P, m) \rightarrow S$

Input: P, an array containing the items of the population
m the number of items to select

Output: S, an array with m randomly selected items from P

```
1  S ← CreateArray(m)
2  k ← 0
3  t ← 0
4  n ← |P|
5  while k < m do
6      u ← Random(0, 1)
7      if u × (n − t) < (m − k) then
8          S[k] ← P[t]
9          k ← k + 1
10     t ← t + 1
11 return S
```

Overall Probability of Selecting an Item

- We saw that the probability of selecting the $(k+1)$th item after selecting first k from the first t items is $(m-k)/(n-t)$.
- We can prove that the *overall probability* of selecting an item is exactly m/n.
- How do we get from $(m-k)/(n-t)$ to m/n?
- The value $(m-k)/(n-t)$ is a *conditional probability*: it is the probability of selecting the $(k+1)$th item after having selected k from t.
- The value m/n is the *unconditional probability* of selecting an item.

Number of Selected Items

- If we have yet to examine $n - t$ items and we also need to select $n - t$ items, then $m - k = n - t$.
- Therefore $u \times (n - t) < m - k$ will become $u < 1$, which means that we will definitely select the next item.
- The same will happen for the following item, and so on, therefore we will pick all the items till the end.
- So there is no way the algorithm can finish having selected less than m items.
- Moreover, if at some point we have selected m items, we won't select any other item, as $u \times (n - t) < m - k$ will become $u \times (n - t) < 0$.
- It follows that the algorithm will select at least m items and at most m items, therefore it will select exactly m items.

Number of Iterations

- The maximum number of repetitions is n, if we need to examine all elements.
- Frequently, however, we will select all the elements we need before examining all of them.
- The probability that an item will be selected is m/n.
- So the probability that the algorithm will stop *before* the last element is $1 - m/n$.
- It can be shown that the average number of elements that we consider before the algorithm stops is $(n + 1)m/(m + 1)$.

Reservoir Sampling

- In selection sampling we must know the population size.
- What can we do if we do not know that?
- For example, we may want to select a random number of records from a file as we read it from the beginning to the end.
- Or we can be fed the elements as the algorithm is executing.
- For that purpose we can use the *reservoir sampling* algorithm.
- This is an online algorithm.

Basic Idea

- The basic idea is that if we want a sample of m elements, we start by filling up a reservoir with the first m elements we come across.
- Then, for every element that we examine, we may fix the contents of the reservoir so that the elements in the reservoir have a probability m/t of being there, where t is the number of elements we have examined.
- When we are done, the items in the reservoir will have a probability m/n of being there, where n is the population size.

Reservoir Sampling Proof Sketch

- We start by putting the first m elements in the reservoir. Therefore all the items have been selected with probability $m/t = m/m = 1$.
- Suppose that we have examined t elements and all items in the reservoir have the same probability of being there.
- We want to show that the same will happen after examining $t + 1$ elements.

Reservoir Sampling Proof

- When we examine the $(t+1)$th element, we add it in the reservoir with probability $m/(t+1)$, taking out one of the items already there.
- We pick the item to take out randomly, so each item in the reservoir has a $1/m$ probability of being taken out.
- So the probability of replacing an item in the reservoir is $m/(t+1) \times (1/m) = 1/(t+1)$.
- Conversely, the probability that an item will remain in the reservoir is $1 - 1(t+1) = t/(t+1)$.
- As the item was in the reservoir, the probability that it was there and it will be removed is $(m/t) \times t/(t+1) = m/(t+1)$.
- Therefore, both the new item and any item remaining in the reservoir have a probability $m/(t+1)$ of being there.

Basic Steps

- In short: if we are at item item $t \leq m$, we just add it in the reservoir.
- If we are at item $t > m$, we add it in the reservoir with probability m/t and we take out randomly an item already in the reservoir.

Reservoir Sampling Algorithm

Algorithm: Reservoir sampling.

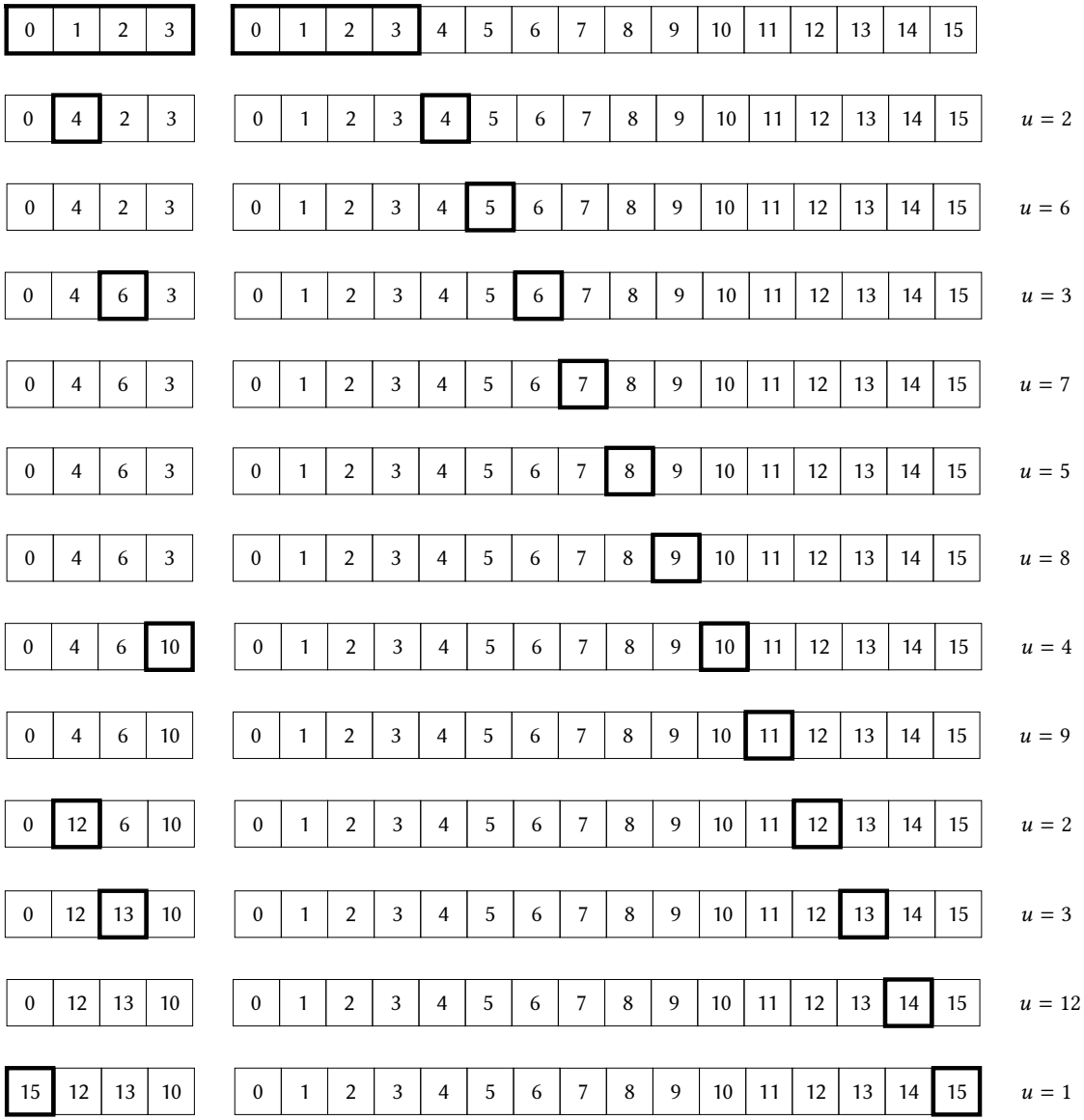
`ReservoirSampling`$(src, m) \rightarrow S$

Input: src, a source of items of the population
m the number of items to select

Output: S, an array with m randomly selected items from src

```
1  S ← CreateArray(m)
2  for i ← 0 to m do
3      S[i] ← GetItem(src)
4  t ← m
5  while (a ← GetItem(src)) ≠ NULL do
6      t ← t + 1
7      u ←RandomInt(1, t)
8      if u ≤ m then
9          S[u − 1] ← a
10 return S
```

Reservoir Sampling Example



Outline

1 General

2 Random Numbers

3 Random Sampling

4 Power Games

5 The Monte Carlo Method

6 Monte Carlo Banzhaf

7 Searching for Primes

General

- Does a vote count? How much?
- In general elections we follow the principle of One Person One Vote (OPOV).
- When millions of votes are cast, it is difficult to see that one of them will make a difference.
- Given that voting takes time and it is not clear how much a vote counts, the so-called “paradox of voting” describes our willingness to vote, even though it is not obvious that it matters.

European Economic Community, 1958

- In 1958 the precursor of today's European Union was set up as the European Economic Community (EEC).
- The founding members were six countries: France (FR), the Federal Republic of Germany (DE; Germany was divided back then), Italy (IT), Belgium (BE), The Netherlands (NL), and Luxembourg (LU).
- One of the governing bodies of the EEC was the Council of Ministers.
- Ministers of the member countries would convene and vote on matters of EEC policy.

Council of Ministers

- Because there were great differences among the countries (in terms of population, wealth, etc.), different countries had different weights in their votes in the Council of Ministers.
- France, Germany, and Italy had four units each.
- Belgium and the Netherlands had two units each.
- Luxembourg had one.
- In order for a vote to pass, it should gather at least 12 units.

Voting in the Council of Ministers

FR (4)	DE (4)	IT (4)	NL (2)	BE (2)	LU (1)	Sum
✓	✓	✓				12
✓	✓		✓	✓		12
✓		✓	✓	✓		12
	✓	✓	✓	✓		12

Possible Outcomes in the Council of Ministers

- We can see that the 12 votes threshold is always achieved without Luxembourg.
- That means that Luxembourg could vote, but that did not matter at all.
- Luxembourg's vote had no real power because it could not influence the outcome of any election.
- More generally, what do voting weights tell us? They imply that Germany was twice more powerful than Belgium. Was that true?

Voting Games

Definition

- Suppose we have a set of voters $V = \{v_1, v_2, \ldots, v_n\}$, and a set of weights $W = \{w_1, w_2, \ldots, w_m\}$.
- Voter v_i votes with a weight w_j according to a mapping $f : V \rightarrow W$.
- For a decision to be taken it must reach a *quota* Q.
- The setup of V, W, f, and Q is called a *voting game*.

In the EEC example we have $Q = 12$.

Coalitions

Definition

- Each subset of voters is called a *coalition*.
- A coalition that reaches the quota is called a *winning coalition*.
- A coalition that fails to reach the quota is called a *losing coalition*.

More on Coalitions

- If we add more voters in a winning coalition, then we get another winning coalition.
- For example, if in the winning coalition {DE, FR, IT} we add Belgium, the resulting coalition {DE, FR, IT, BE} is also winning.
- It is more interesting to see what happens when we take out a voter from a winning coalition.
- For example, if from the coalition {DE, FR, IT} we remove a country, the resulting coalition is not a willing one.

Minimal Winning Coalition

Definition

- A *minimal winning coalition* is a winning coalition such that the removal of any of its members results in obtaining a losing coalition.
- A voter is *critical* in a winning coalition, also called a *swinger* or a *pivot*, if its removal from the coalition makes the coalition a losing coalition.

- In a minimal winning coalition all voters are critical.
- A voter may be critical in a coalition that is not minimal.
- For example, consider DE in {DE, FR, IT, BE}.

Minimal Winning Coalition (Contd.)

- A critical voter can affect the outcome of an election.
- A voter that is not critical in any coalition is a voter that is not able to affect the outcome of an election at all.
- Such a voter is called a *dummy*.
- In the EEC example, Luxembourg was a dummy voter.

Banzhaf Score

Definition

The *Banzhaf score* of a voter v_i is the number of coalitions in which the v_i is critical. We denote the Banzhaf score by $\eta(v_i)$.

- The name comes from John F. Banzhaf III who proposed the score.
- The Banzhaf score by itself does not provide much information.
- That is because it gives no indication of the importance of the number of coalitions in which voter v_i is critical.
- There may be many more critical coalitions in which v_i is not critical.

Banzhaf Index

Definition

The *Banzhaf index of voting power*, or *Banzhaf index*, is the number of coalitions in which v_i is critical divided by the total number of critical coalitions for all voters. We denote the Banzhaf index by $\beta(v_i)$, so we have:

$$\beta(v_i) = \frac{\eta(v_i)}{\eta(v_1) + \eta(v_2) + \cdots + \eta(v_n)}$$

If we think of the total voting influence as a pie, then the Banzhaf index is the part of the pie, the ratio of the influence, that goes to each voter.

Banzhaf Index Example

- Suppose we have four voters A, B, C, D with corresponding weights 4, 2, 1, 3 and quota $Q = 6$.
- The critical coalitions are the following (we underling the critical voters): $\{\underline{A}, \underline{B}\}$, $\{\underline{A}, \underline{D}\}$, $\{\underline{A}, \underline{B}, C\}$, $\{\underline{A}, B, D\}$, $\{\underline{A}, C, \underline{D}\}$, $\{\underline{B}, \underline{C}, \underline{D}\}$.
- So we have $\eta(v_A) = 5$, $\eta(v_B) = 3$, $\eta(v_C) = 1$, and $\eta(v_D) = 3$.
- Therefore, $\beta(v_A) = 5/12$, $\beta(v_B) = 3/12$, $\beta(v_C) = 1/12$, $\beta(v_D) = 3/12$.
- Note that D has a greater weight than B, but this is not translated into more voting power.
- Note also that it is not easy to calculate the Banzhaf index in elections with many voters.

Relative and Absolute Voting Power

- The Banzhaf index is a *relative measure*.
- It shows us how a value compares to others.
- It is like the the proportion of the total income of a group that accrues to each particular member.
- We are also interested in an *absolute measure*.
- That is similar to the step from the relative income to the actual income of the person.

Coalitions and Power Sets

- Each coalition corresponds to a subset of the set S of voters.
- The set of all possible subsets of S is called *power set* and its symbol is 2^S.
- If the size of S is n, the size of 2^S is 2^n.
- To see that, observe that each subset of S corresponds to a binary number with n bits.

Power Sets and Binary Numbers

	x	y	z
$\varnothing$	0	0	0
$\{z\}$	0	0	1
$\{y\}$	0	1	0
$\{y, z\}$	0	1	1
$\{x\}$	1	0	0
$\{x, z\}$	1	0	1
$\{x, y\}$	1	1	0
$\{x, y, z\}$	1	1	1

Banzhaf Measure

- If each coalition is equally likely, its probability is $1/2^n$.
- If we take out a voter v_i we have $n-1$ voters, so 2^{n-1} coalitions.
- The probability that a voter is critical is the probability that one of the 2^{n-1} coalitions becomes critical when adding v_i.
- That probability is equal to the number of critical coalitions involving v_i divided by the number of all coalitions without v_i.

Definition

The Banzhaf score $\eta(v_i)$ divided by 2^{n-1} is the *Banzhaf measure of voting power*, $\beta'(v_i)$:

$$\beta'(v_i) = \frac{\eta(v_i)}{2^{n-1}}$$

Banzhaf Measure (Contd.)

- That is the absolute measure we were looking for.
- The Banzhaf measure is the probability, if we don't know how v_i will vote, when the votes are counted, if v_i were to switch its preference, then the outcome of the vote would change as well.
- Alternatively, it is the probability that, if we know how v_i will vote, the outcome of the voting would change if v_i were to change opinion.

Banzhaf Measure Example

Coalitions without A	Coalitions with A	Votes	Winning	Critical
$\emptyset$	$\{A\}$	4		
$\{B\}$	$\{A, B\}$	6	✓	✓
$\{C\}$	$\{A, C\}$	5		
$\{D\}$	$\{A, D\}$	7	✓	✓
$\{B, C\}$	$\{A, B, C\}$	7	✓	✓
$\{B, D\}$	$\{A, B, D\}$	9	✓	✓
$\{C, D\}$	$\{A, C, D\}$	8	✓	✓
$\{B, C, D\}$	$\{A, B, C, D\}$	10	✓	

Banzhaf Measure Example (Contd.)

- The Banzhaf measure for A is $5/8$.
- For the others we have $B = 3/8$, $C = 1/8$, $D = 3/8$.
- The measure is not normalized, so it does not add up to one.
- That happens because it is an absolute measure.
- We can use it to compare voters' influence across different elections, which we cannot do with the Banzhaf index.

Banzhaf Measure and Banzhaf Index

- We can go from the Banzhaf measure to the Banzhaf index with the following:

$$\beta(v_i) = \beta'(v_i) \times \frac{2^{n-1}}{\eta(v_1) + \eta(v_2) + \cdots + \eta(v_n)}$$

- Therefore we can treat the Banzhaf measure as the primary concept and the Banzhaf index as a derivative concept.

Calculating the Banzhaf Measure

- The enumeration procedure we used to calculate $\beta'(v_i) = \eta(v_i)/2^{n-1}$ can work only when the number of voters is small.
- Calculating $\beta'(v_i)$ for a large number of voters is difficult.
- The denominator is not a problem—2^{n-1} is a very big number, but we can calculate it directly.
- The problem is the numerator. There is no way to calculate $\eta(v_i)$ efficiently.

Randomized Calculation of the Banzhaf Measure

- However, we can calculate $\beta'(v_i)$ using a randomized approach.
- Suppose we take coalitions at random.
- Some of these will be critical.
- If we take them randomly, and we have a large enough sample, then the ratio of the critical coalitions in our sample over all coalitions in our sample will approach the true ratio of the critical coalitions over all possible coalitions.

Outline

1 General

2 Random Numbers

3 Random Sampling

4 Power Games

5 The Monte Carlo Method

6 Monte Carlo Banzhaf

7 Searching for Primes

The Monte Carlo Method

Definition

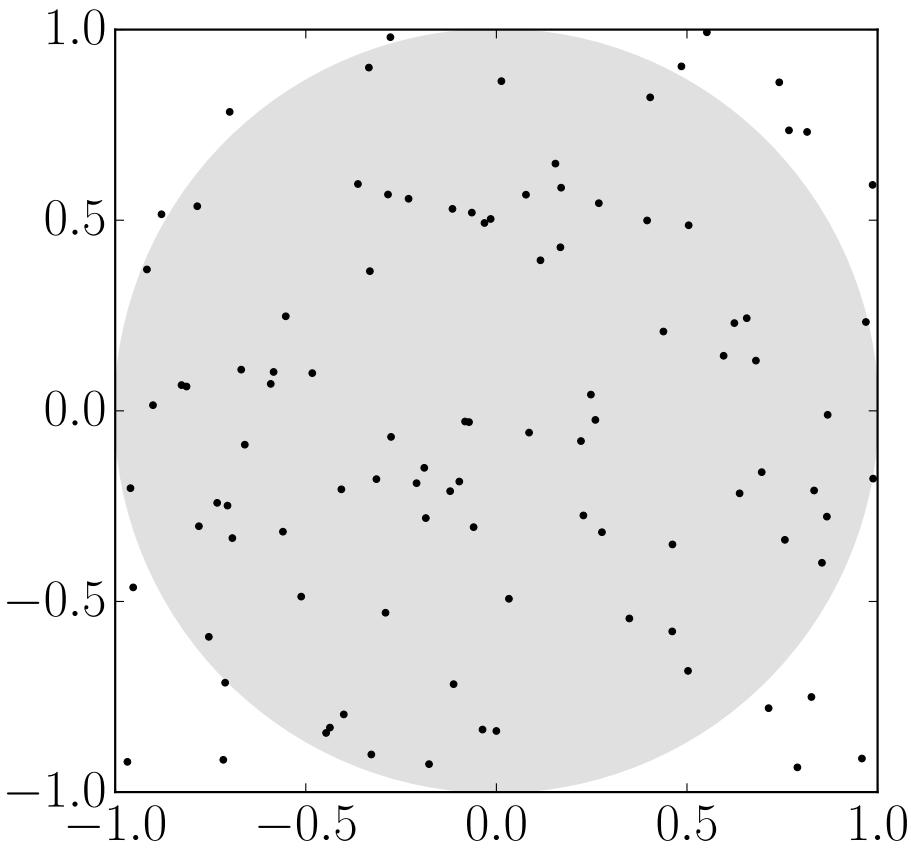
A computational method that uses random sampling to arrive at its results is called a *Monte Carlo method*.

- The Monte Carlo method is named after the famous casino.
- It was invented by digital computers pioneers.
- It has a wide range of applications, from engineering to finance.

The Monte Carlo Method for π

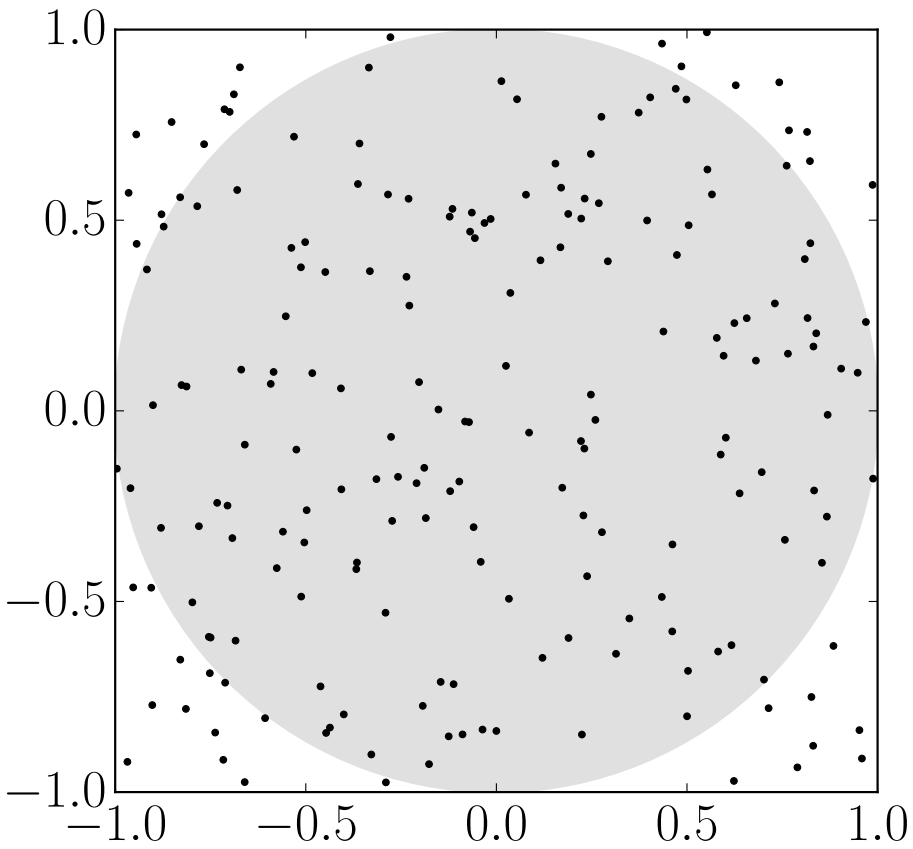
- Suppose we have a square whose sides are two units long. The area will be four square units.
- If we inscribe a circle inside the square, its diameter will be two units and its radius one unit.
- Therefore the circle's area will be π.
- If we scatter some small objects randomly over the square (say, grains of rice), some of them will land inside the circle, some of them outside.
- If we scatter enough objects, the ratio of those inside the circle to those outside will be $\pi/4$.

The Monte Carlo Method for π Example (1)



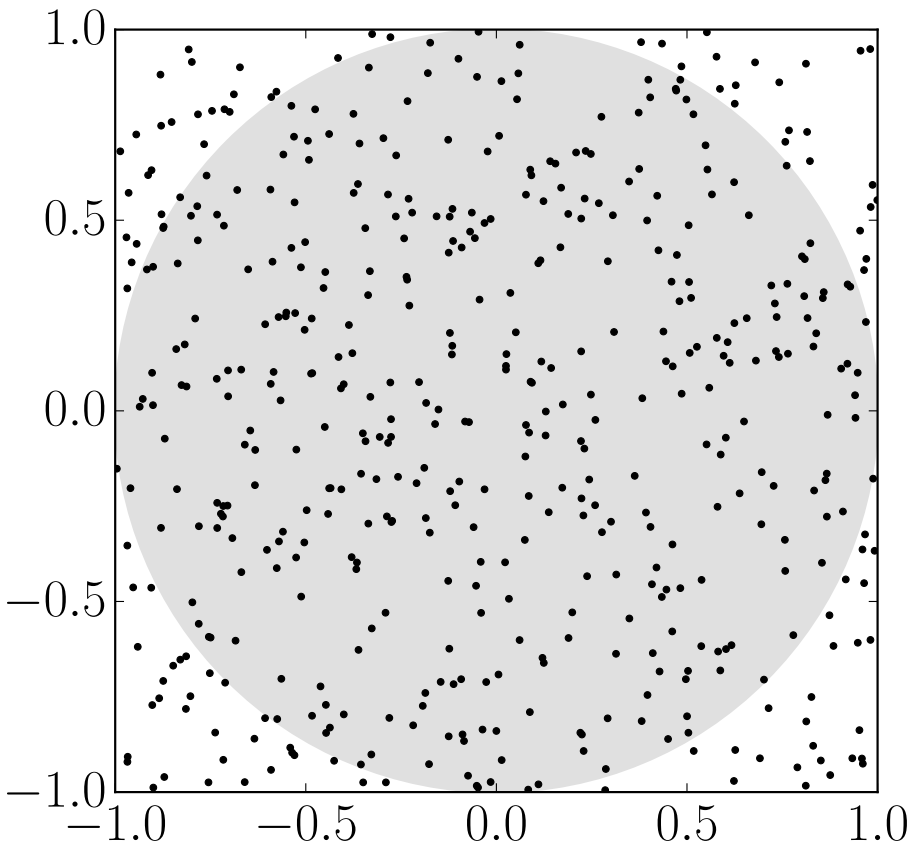
100 points, $\pi = 3.2$, $s_e = 0.017$.

The Monte Carlo for π Example (2)



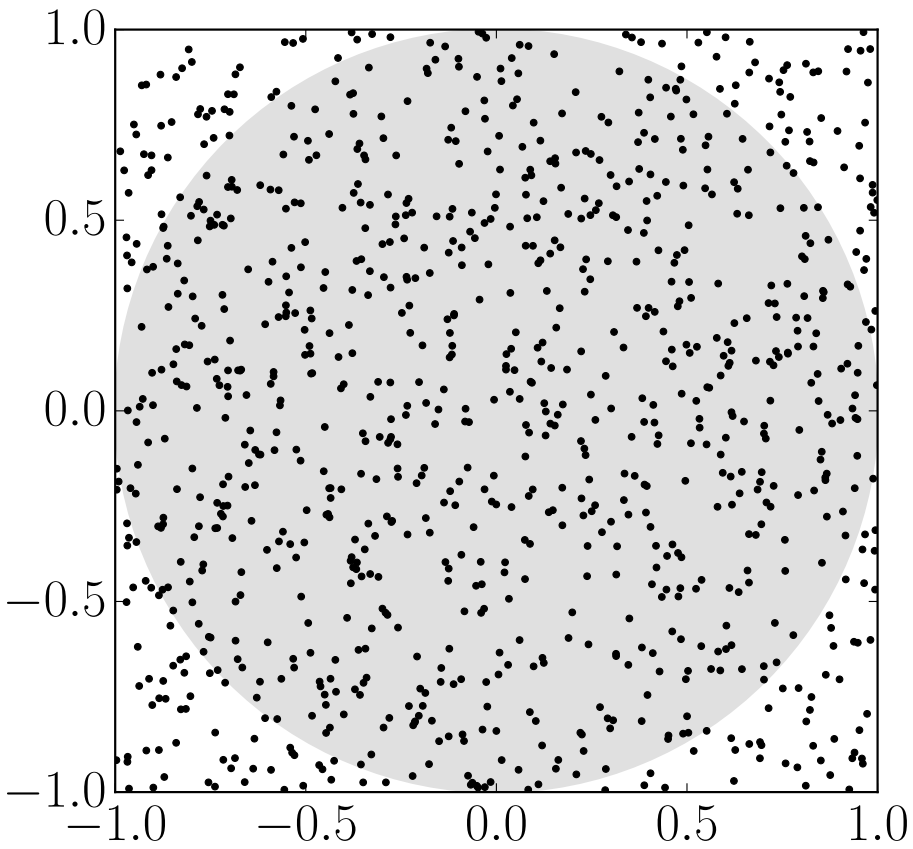
200 points, $\pi = 3.12$, $s_e = 0.012$.

The Monte Carlo Method for π Example (3)



500 points, $\pi = 3.12$, $s_e = 0.008$.

The Monte Carlo Method for π Example (4)



1000 points, $\pi = 3.14$, $s_e = 0.005$.

The Monte Carlo Method for π—Standard Error

- We define a variable X equal to one when the point is inside the circle and equal to zero when it is outside.
- The *expected value* of X is $E[X] = \pi/4 \times 1 + (1 - \pi/4) \times 0 = \pi/4$.
- The *variance* of X is $\sigma^2 = E[X^2] - (E[X])^2$.
- We have $E[X^2] = (\pi/4) \times 1^2 + [1 - (\pi/4)] \times 0^2 = \pi/4$. Therefore $\sigma^2 = \pi/4 - (\pi/4)^2 = (\pi/4)(1 - \pi/4)$.
- The *standard error* then is $s_e = \sigma/\sqrt{n}$.
- The square root of the variance, σ, is the *standard deviation*.
- A standard error s_e means that we have a 95% probability that the true value of π is within a margin of $\pm 1.96 s_e$ from the calculated value.

The Monte Carlo Method for π—Only an Example!

Careful, this was nice example to introduce Monte Carlo methods in general, it is not a particularly good method for calculating π. There are much more efficient methods for that.

Outline

1 General

2 Random Numbers

3 Random Sampling

4 Power Games

5 The Monte Carlo Method

6 Monte Carlo Banzhaf

7 Searching for Primes

Monte Carlo Banzhaf Measure Calculation

- We can use the Monte Carlo method to calculate the Banzhaf measure.
- We will be generating coalitions in random and checking whether they are critical.
- The ratio of those that are critical to all those that we generate will give us an approximation of the Banzhaf measure.
- A random coalition is a random subset, so we need to find a way to generate random subsets.

Random Subset Generation Algorithm

Algorithm: Random subset generation.

RandomSubset(S) $\rightarrow$ RS

Input: S, a set

Output: RS, a random subset of S

```
1  RS ← CreateList()
2  foreach m in S do
3      r ← Random(0, 1)
4      if r < 0.5 then
5          InsertInList(RS, NULL, m)
6  return RS
```

Random Subset Generation

- The input is a set S.
- The output RS is a random subset of S.
- We examine each element of S and we decide at random whether to include it in RS.
- To do that, we take a random number in the range $[0, 1)$ and we compare it with 0.5. If it is smaller than 0.5 we include it in RS, otherwise we leave it out.

Monte Carlo Banzhaf Measure Algorithm

Algorithm: Monte Carlo Banzhaf measure.

BanzhafMeasure$(v, ov, q, w, t) \rightarrow b$

Input: v, a voter
ov, a list containing the other voters
q, the quota required
w an associative array containing the weight of each voter
t, the number of tries

Output: b, the Banzhaf measure for voter v

```
1  k ← 0
2  nc ← 0
3  while k < t do
4      coalition ← RandomSubset(ov)
5      votes ← 0
6      foreach m in coalition do
7          votes ← votes + Lookup(w, m)
8      if votes < q and votes + Lookup(w, v) ≥ q then
9          nc ← nc + 1
10     k ← k + 1
11 b ← nc/k
12 return b
```

Monte Carlo Banzhaf Measure

- The algorithm calculates the Banzhaf measure for a voter in t iterations.
- In each iteration, we take a random subset, i.e., a coalition, that does not contain the voter that we are considering.
- We add the votes of the subset.
- If by adding the votes of the voter we pass the quota, the coalition is critical.
- At the end we divide the critical coalitions over all coalitions that we considered.

Accuracy of Monte Carlo Banzhaf Measure

- We want to know how many times iterations we need in order to achieve a target accuracy.
- It can be shown that if we want our result to be within ϵ with probability δ, the required number of samples is:

$$k \geq \frac{\ln \frac{2}{1-\delta}}{2\epsilon^2}$$

Banzhaf Measures of U.S. States

- Now we can calculate the Banzhaf measures of U.S. states.
- The U.S. president is elected from the electoral college.
- The electoral college consists of a number of electors for each state and the District of Columbia.
- There are 538 electors, so the president is elected with a quota of 270 electors.
- The number of electors per state changes depending on the latest census.
- The results of the following table were calculated with $\epsilon = 0.001$ and $\delta = 0.95$.
- That required 1,844,440 samples for each Banzhaf measure.

Table of Banzhaf Measures of U.S. States

CA	55	0.471	MN	10	0.076	NM	5	0.038
TX	38	0.298	MO	10	0.075	WV	5	0.038
FL	29	0.223	WI	10	0.076	HI	4	0.03
NY	29	0.224	AL	9	0.068	ID	4	0.03
IL	20	0.153	CO	9	0.068	ME	4	0.03
PA	20	0.153	SC	9	0.068	NH	4	0.03
OH	18	0.136	KY	8	0.06	RI	4	0.03
GA	16	0.121	LA	8	0.061	AK	3	0.023
MI	16	0.121	CT	7	0.053	DC	3	0.023
NC	15	0.114	OK	7	0.052	DE	3	0.023
NJ	14	0.106	OR	7	0.053	MT	3	0.023
VA	13	0.098	AR	6	0.045	ND	3	0.023
WA	12	0.091	IA	6	0.045	SD	3	0.023
AZ	11	0.083	KS	6	0.045	VT	3	0.023
IN	11	0.083	MS	6	0.045	WY	3	0.023
MA	11	0.083	NV	6	0.045			
TN	11	0.083	UT	6	0.046			
MD	10	0.076	NE	5	0.038			

Outline

1. General
2. Random Numbers
3. Random Sampling
4. Power Games
5. The Monte Carlo Method
6. Monte Carlo Banzhaf
7. Searching for Primes

General

- In many cryptographic applications we need to find large prime numbers.
- These numbers typically have m bits, where m is a large power of two (1024, 2048, 4096, ...).
- Finding primes is not an easy task.
- According to the Prime Number Theorem, if n is large, the number of primes less than or equal to n is about $n/\ln n$.
- But how can we find them?

The Sieve of Eratosthenes

- One way to pick a random prime is to find all prime numbers with m bits and select one of them at random.
- To find all prime numbers we can use the *Sieve of Eratosthenes*.
- We start with two, which is prime.
- We mark all multiples of two up to our limit—these numbers are composites.
- We go back to two and look for the next prime number after two. That will be the first number we have not marked as composite.
- We mark again all the number's multiples as composites.
- We continue until we exhaust all numbers.

Sieve of Eratosthenes Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
2	F	F	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
3	F	F	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T
5	F	F	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	F	F	F	F	T	F	T

The Sieve of Eratosthenes Algorithm

Algorithm: The Sieve of Eratosthenes.

`SieveEratosthenes`$(n) \rightarrow isprime$

Input: n, a natural number greater than 1

Output: $isprime$, a boolean array of size $n+1$ such that if $p \leq n$ is a prime, $isprime[p]$ is TRUE, otherwise it is FALSE

```
1  isprime ← CreateArray(n + 1)
2  isprime[0] ← FALSE
3  isprime[1] ← FALSE
4  for i ← 2 to n + 1 do
5      isprime[i] ← TRUE
6  p ← 2
7  while p^2 ≤ n do
8      if isprime[p] = TRUE then
9          j ← p
10         while j ≤ ⌊n/p⌋ do
11             isprime[j × p] ← FALSE
12             j ← j + 1
13     p ← p + 1
14 return isprime
```

Analysis of the Algorithm

- In each iteration of the outer loop we mark as composites all multiples of the prime numbers $p \leq \sqrt{n}$.
- The square root comes into play because every composite c such that $\sqrt{n} \leq c \leq n$ can be written as the product of two factors $c = f_1 \times f_2$ where $f_1 \leq \sqrt{n}$ and $f_2 \leq \sqrt{n}$.
- In the first iteration we strike off all multiples of 2, $\lfloor n/2 \rfloor$; in the second iteration all multiples of 3, $\lfloor n/3 \rfloor$; then all multiples of 5, $\lfloor n/5 \rfloor$, and so on until the largest prime $k < \sqrt{n}$.

Analysis of the Algorithm (Contd.)

- In total we strike off at most $n/2 + n/3 + n/5 + \cdots + n/k$ composites, which is equal to $n(1/2 + 1/3 + 1/5 + \cdots + 1/k)$.
- The sum $(1/2 + 1/3 + 1/5 + \cdots + 1/k)$ is the sum of the reciprocal of primes not greater than $\sqrt{n}$.
- It can be proven that the sum of the reciprocal of primes not greater than a number m is $O(\log \log m)$.
- Therefore, the total time we spend striking off composites is $O(n \log \log \sqrt{n}) = O(n \log \log n)$, which is the complexity of the algorithm.

Efficiency of Finding Primes

- There are better algorithms that have a complexity of $O(n)$.
- It therefore appears that we do not have a problem with efficiency.
- But it is not so.
- We said that $n = 2^m$, therefore in reality the complexity is $O(2^m)$.
- For a number with 4096 bits we get $O(2^{4096})$, which is not practical.

Randomized Primes Finding

- From the Prime Numbers Theorem, if we examine the first n numbers, we expect to find about $n/\ln n$ primes.
- If we pick a number at random, the probability that it will be prime will be $1/\ln n$.
- Conversely, it can be shown that in order to find a prime we must try on average $\ln n$ numbers.
- If we are interested in primes with 4096 bits, we will have to try on average $\ln(2^{4096}) \approx 2840$ candidate primes.
- So, if we have a fast way to check primality, this approach will be practical.

Primality Checks

- To check if a number n is prime, we can try to divide it with all numbers from 2 up to $\sqrt{n}$.
- That is because a number greater than $\lceil \sqrt{n} \rceil$ can produce n only if it is multiplied by a number no greater than $\lfloor \sqrt{n} \rfloor$.
- In fact, we can halve the checks, by trying only with odd numbers.
- In any case, this approach will require $O\left((1/2)\sqrt{n}\right) = O(\sqrt{n})$ steps.
- That is again prohibitive, because $O\left(\sqrt{2^m}\right) = O\left((2^m)^{1/2}\right) = O(2^{m/2})$.

A Randomized Primality Check

- We can use a randomized approach instead.
- We'll present an algorithm that will be able to report very fast a prime number.
- However, the algorithm may be wrong!
- That is, it may give us a number asserting that it is prime, while in reality it is composite.
- But we can control that, if the probability of error is extremely low.

A Randomized Primality Check

- To do that, we'll need an auxiliary algorithm, which we'll call a *witness*.
- This algorithm will take a number as input.
- If it reports that the number is composite, it is telling the truth.
- If it reports that the number is prime, it may be wrong.

A Witness for Composite Numbers

- The witness does not always tell the truth.
- If the witness says “The number is composite,” this we know is true.
- If the witness says “The number is prime,” we cannot be sure—but we’ll see that it is more probable that the witness is telling the truth.

Composite Witness Operation

1. To do that, we start with a random odd number p.
2. This number can always be written in the form $p = 1 + 2^r q$ with q odd.
3. Then we take another random number x in the interval $[2, p-1]$.
4. We calculate $y = x^q \bmod p$
5. If $y = 1$ or $y = p - 1$ we stop and we say that p is probably prime.
6. Otherwise, we calculate $y \leftarrow y^2 \bmod p$.
7. If $y = p - 1$ we stop and say that p is probably prime.
8. If $y = 1$ we stop and say that p is definitely composite.
9. Otherwise, we go back to step 6, doing up to r iterations.
10. After r iterations, we stop and say that p is definitely composite.

Witness for Composite Numbers Algorithm

Algorithm: A witness for composite numbers.

`WitnessComposite`$(p) \longrightarrow$ TRUE or FALSE

Input: p, an odd integer

Output: a boolean value that is TRUE if the number is definitely composite, FALSE otherwise

```
1  (r, q) ← FactorTwo(p − 1)
2  x ← RandomInt(2, p − 1)
3  y ← x^q mod p
4  if y = 1 then
5      return FALSE
6  for j ← 0 to r do
7      if y = p − 1 then
8          return FALSE
9      y ← y^2 mod p
10     if y = 1 then
11         return TRUE
12 return TRUE
```

How Reliable is the Witness?

- What is the probability that the witness is wrong?
- It can be proved that the probability that the witness will report falsely that a number is prime is $1/4$.
- Therefore, if we use the witness twice, the probability that it will report falsely that a number is prime is $(1/4)^2$.
- If we use it t times, the probability of a wrong answer is $(1/4)^t$.
- For $t = 50$ the probability that the witness is wrong is $(1/4)^{50}$, which is enough for practical purposes.

Miller-Rabin Primality Test

- Following what we've seen, we arrive at the Miller-Rabin primality test.
- That algorithm takes a random odd number as input.
- It checks, with the witness, whether it is composite.
- If it is, it returns immediately, saying that it is not a prime.
- Otherwise, it calls the witness again. We repeat the process as many times as we want.
- The overall complexity of the witness algorithm is $O((\lg p)^3)$.
- We call the witness algorithm t times, so for t iterations the overall complexity of the Miller-Rabin test is $O(t \cdot (\lg p)^3)$.
- We expect to try about $\ln p$ numbers to find a prime, where each check takes $O(t \cdot (\lg p)^3)$.

Miller-Rabin Primality Test Algorithm

Algorithm: Miller-Rabin primality test.

`MillerRabinPrimalityTest`$(p, t) \longrightarrow$ TRUE or FALSE

Input: p, an odd integer
t, the number of times the witness primality function will be applied

Output: TRUE if the number is prime with probability $(1/4)^t$, FALSE if the number is definitely composite

```
1  for i ← 0 to t do
2      if WitnessComposite(p) then
3          return FALSE
4  return TRUE
```

Finding $p = 1 + 2^r q$

- In the witness algorithm we stated that an odd number p can be written as $p = 1 + 2^r q$, where q is odd.
- To do that we used the `FactorTwo` function.
- It is time to see how this is implemented.
- If $p = 1 + 2^r q$, then $2^r q$ is even.
- The following algorithm implements the factoring of an even number n as $2^r q$ with q odd.

Factor n as $2^r q$, with q Odd

Algorithm: Factor n as $2^r q$, with q odd.

FactorTwo$(n) \rightarrow (r, q)$

Input: n, an even integer

Output: (r, q), such that $n = 2^r q$ with q odd

```
1  q ← n
2  r ← 0
3  while q mod 2 = 0 do
4      r ← r + 1
5      q ← q/2
6  return (r, q)
```