Computing with Randomness

C. S. Calude

University of Auckland, NZ

Tutorial, UC'06, York, UK, 2006

Outline

Probabilistic primality tests

Three types of randomness

Using uncomputability for computation

Probabilistic understanding of the halting problem

Selected references

Outline

Probabilistic primality tests

Three types of randomness

Using uncomputability for computation

Probabilistic understanding of the halting problem

Selected references

Primality tests

Primality is known to be in P, but all known algorithms are inefficient for testing large numbers. Probabilistic primality tests are still the best practical choice. They are very efficient, but only “probably correct”.

Solovay-Strassen and Miller-Rabin probabilistic primality tests 1

To test whether n is prime, proceed as follows:

1. take k natural numbers uniformly distributed between 1 and $n-1$.
2. for each such a number i check whether some fixed predicate $W(i, n)$ holds.
3. if for some i, $W(i, n)$ is true, then **n is composite** and i is a *witness of n's compositeness*; otherwise, **n is prime**. In the second case, the probability that ''n is prime'' is greater than $1 - 2^{-k}$.

Solovay-Strassen and Miller-Rabin probabilistic primality tests 1

To test whether n is prime, proceed as follows:

1. take k natural numbers uniformly distributed between 1 and $n-1$.

2. for each such a number i check whether some fixed predicate $W(i,n)$ holds.

3. if for some i, $W(i,n)$ is true, then **n is composite** and i is a *witness of n's compositeness*; otherwise, **n is prime**. In the second case, the probability that ''n is prime'' is greater than $1-2^{-k}$.

Solovay-Strassen and Miller-Rabin probabilistic primality tests 1

To test whether n is prime, proceed as follows:

1. take k natural numbers uniformly distributed between 1 and $n-1$.

2. for each such a number i check whether some fixed predicate $W(i,n)$ holds.

3. if for some i, $W(i,n)$ is true, then **n is composite** and i is a *witness of n's compositeness*; otherwise, **n is prime**. In the second case, the probability that ''n is prime'' is greater than $1-2^{-k}$.

Solovay-Strassen and Miller-Rabin probabilistic primality tests 1

To test whether n is prime, proceed as follows:

1. take k natural numbers uniformly distributed between 1 and $n-1$.

2. for each such a number i check whether some fixed predicate $W(i,n)$ holds.

3. if for some i, $W(i,n)$ is true, then **n is composite** and i is a *witness of n's compositeness*; otherwise, **n is prime**. In the second case, the probability that ''n is prime'' is greater than $1-2^{-k}$.

Solovay-Strassen and Miller-Rabin probabilistic primality tests 2

"Witness of compositeness" exist because at least half of the numbers $i \in \{1, 2 \ldots, n-1\}$ are witnesses of n's compositeness—if n is composite—and none of them are—in case n is prime.

Furthermore, computing $W(i, n)$ is easy/fast.

The probabilistic primality tests are very efficient, but, remember, only probably correct. Can we make them error-free?

Solovay-Strassen and Miller-Rabin probabilistic primality tests 2

“Witness of compositeness” exist because at least half of the numbers $i \in \{1, 2 \ldots, n-1\}$ are witnesses of n’s compositeness—if n is composite—and none of them are—in case n is prime.

Furthermore, computing $W(i, n)$ is easy/fast.

The probabilistic primality tests are very efficient, but, remember, only probably correct. Can we make them error-free?

Solovay-Strassen and Miller-Rabin probabilistic primality tests 2

"Witness of compositeness" exist because at least half of the numbers $i \in \{1, 2 \ldots, n-1\}$ are witnesses of n's compositeness—if n is composite—and none of them are—in case n is prime.

Furthermore, computing $W(i, n)$ is easy/fast.

The probabilistic primality tests are very efficient, but, remember, only probably correct. Can we make them error-free?

Algorithmic randomness, primality and Monte-Carlo simulation

Theorem [Chaitin-Schwartz, 1978]

For almost all inputs, the probabilistic primality test is error-free in case it uses a long enough algorithmically random inputs.

Theorem [Calude-Zimand, 1984]

For almost all inputs, every "decent" Monte Carlo simulation algorithm is error-free in case it uses a long enough algorithmically random inputs.

What are these wonder algorithmically random inputs? Can we buy them? Are they expensive?

Algorithmic randomness, primality and Monte-Carlo simulation

Theorem [Chaitin-Schwartz, 1978]

For almost all inputs, the probabilistic primality test is error-free in case it uses a long enough algorithmically random inputs.

Theorem [Calude-Zimand, 1984]

For almost all inputs, every "decent" Monte Carlo simulation algorithm is error-free in case it uses a long enough algorithmically random inputs.

What are these wonder algorithmically random inputs? Can we buy them? Are they expensive?

Algorithmic randomness, primality and Monte-Carlo simulation

Theorem [Chaitin-Schwartz, 1978]

For almost all inputs, the probabilistic primality test is error-free in case it uses a long enough algorithmically random inputs.

Theorem [Calude-Zimand, 1984]

For almost all inputs, every "decent" Monte Carlo simulation algorithm is error-free in case it uses a long enough algorithmically random inputs.

What are these wonder algorithmically random inputs? Can we buy them? Are they expensive?

Algorithmic randomness, primality and Monte-Carlo simulation

Theorem [Chaitin-Schwartz, 1978]

For almost all inputs, the probabilistic primality test is error-free in case it uses a long enough algorithmically random inputs.

Theorem [Calude-Zimand, 1984]

For almost all inputs, every “decent” Monte Carlo simulation algorithm is error-free in case it uses a long enough algorithmically random inputs.

What are these wonder algorithmically random inputs? Can we buy them? Are they expensive?

Algorithmic randomness, primality and Monte-Carlo simulation

Theorem [Chaitin-Schwartz, 1978]

For almost all inputs, the probabilistic primality test is error-free in case it uses a long enough algorithmically random inputs.

Theorem [Calude-Zimand, 1984]

For almost all inputs, every “decent” Monte Carlo simulation algorithm is error-free in case it uses a long enough algorithmically random inputs.

What are these wonder algorithmically random inputs? Can we buy them? Are they expensive?

Outline

Probabilistic primality tests

Three types of randomness

Using uncomputability for computation

Probabilistic understanding of the halting problem

Selected references

Software-generated, quantum and algorithmic randomness

- Pseudo-randomness or software generated randomness.
  "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin". (John von Neumann)
- Quantum randomness.
- Algorithmic randomness.

Algorithmic/quantum randomness implies software generated randomness, but the converse is false.

Software-generated, quantum and algorithmic randomness

- Pseudo-randomness or software generated randomness. "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin". (John von Neumann)
- Quantum randomness.
- Algorithmic randomness.

Algorithmic/quantum randomness implies software generated randomness, but the converse is false.

Software-generated, quantum and algorithmic randomness

- Pseudo-randomness or software generated randomness. "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin". (John von Neumann)
- Quantum randomness.
- Algorithmic randomness.

Algorithmic/quantum randomness implies software generated randomness, but the converse is false.

Software-generated, quantum and algorithmic randomness

- Pseudo-randomness or software generated randomness. "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin". (John von Neumann)
- Quantum randomness.
- Algorithmic randomness.

Algorithmic/quantum randomness implies software generated randomness, but the converse is false.

Software-generated, quantum and algorithmic randomness

- Pseudo-randomness or software generated randomness. "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin". (John von Neumann)
- Quantum randomness.
- Algorithmic randomness.

Algorithmic/quantum randomness implies software generated randomness, but the converse is false.

Algorithmic randomness: Leibniz

Leibniz's 1686 dictum: A theory must be simpler than the data is explains.

Reason: a 'law' which is as complicated as the experimental data that it explains is meaningless, the 'law' explains nothing.

Understanding is compression.

Algorithmic randomness: Leibniz

Leibniz's 1686 dictum: A theory must be simpler than the data is explains.

Reason: a 'law' which is as complicated as the experimental data that it explains is meaningless, the 'law' explains nothing.

Understanding is compression.

Algorithmic randomness: Leibniz

Leibniz's 1686 dictum: A theory must be simpler than the data is explains.

Reason: a 'law' which is as complicated as the experimental data that it explains is meaningless, the 'law' explains nothing.

Understanding is compression.

Program-size complexity

Let $B = \{0, 1\}$ and let C be a prefix-free binary Turing machine. The program-size complexity $H_C(x)$ is the size in bits of the smallest program for C to compute x:

$$H_C(x) = \min\{|p| \mid x \in B^*, C(p) = x\},$$

with the convention that strings not produced by C have infinite complexity.

Universality

Theorem

We can effectively construct a prefix-free TM U such that for every prefix-free TM C there effectively exists a constant $c = c_{U,C}$ such that for each input string x:

$$H_U(x) \leq H_C(x) + c.$$

We pick a particular natural U and let H denote H_U.

Examples

- Low complexity strings: $H(0^n) \approx \log n + c$, $H(1^n) \approx \log n + c$.
- Intermediate complexity strings (borderline algorithmically random strings):
  if $x^* = \min\{p \mid U(p) = x\}$, then
  $H(x) = |x^*|, H(x^*) \geq |x^*| - c$.
- High complexity strings (algorithmically random strings):
  $\max\{H(x) \mid |x| = n\} \approx n + 2\log n + c$.

Examples

- Low complexity strings: $H(0^n) \approx \log n + c$, $H(1^n) \approx \log n + c$.
- Intermediate complexity strings (borderline algorithmically random strings):
  if $x^* = \min\{p \mid U(p) = x\}$, then
  $H(x) = |x^*|$, $H(x^*) \geq |x^*| - c$.
- High complexity strings (algorithmically random strings):
  $\max\{H(x) \mid |x| = n\} \approx n + 2\log n + c$.

Examples

- Low complexity strings: $H(0^n) \approx \log n + c$, $H(1^n) \approx \log n + c$.
- Intermediate complexity strings (borderline algorithmically random strings):
  if $x^* = \min\{p \mid U(p) = x\}$, then
  $H(x) = |x^*|$, $H(x^*) \geq |x^*| - c$.
- High complexity strings (algorithmically random strings):
  $\max\{H(x) \mid |x| = n\} \approx n + 2\log n + c$.

Examples

- Low complexity strings: $H(0^n) \approx \log n + c$, $H(1^n) \approx \log n + c$.
- Intermediate complexity strings (borderline algorithmically random strings):
  if $x^* = \min\{p \mid U(p) = x\}$, then
  $H(x) = |x^*|$, $H(x^*) \geq |x^*| - c$.
- High complexity strings (algorithmically random strings):
  $\max\{H(x) \mid |x| = n\} \approx n + 2\log n + c$.

Is program-size complexity computable?

Theorem
The program-size complexity H is not computable.

Theorem
The program-size complexity H is computable in the limit from above, i.e. the set

$$\{(x, n) \mid x \in B^*, n \geq 0, H(x) \leq n\}$$

is computably enumerable.

Is program-size complexity computable?

Theorem

The program-size complexity H is not computable.

Theorem

The program-size complexity H is computable in the limit from above, i.e. the set

$$\{(x, n) \mid x \in B^*, n \geq 0, H(x) \leq n\}$$

is computably enumerable.

Is program-size complexity computable?

Theorem

The program-size complexity H is not computable.

Theorem

The program-size complexity H is computable in the limit from above, i.e. the set

$$\{(x, n) \mid x \in B^*, n \geq 0, H(x) \leq n\}$$

is computably enumerable.

Algorithmically random strings

- Satisfy all computable enumerable statistical properties of randomness.
- Are unpredictable.
- Every "decent" Monte Carlo simulation algorithm powered with algorithmic randomness produces the result not only true with high probability, but rigourously correct.
- Are more than Turing uncomputable.

Algorithmically random strings

- Satisfy all computable enumerable statistical properties of randomness.
- Are unpredictable.
- Every "decent" Monte Carlo simulation algorithm powered with algorithmic randomness produces the result not only true with high probability, but rigourously correct.
- Are more than Turing uncomputable.

Algorithmically random strings

- Satisfy all computable enumerable statistical properties of randomness.
- Are unpredictable.
- Every "decent" Monte Carlo simulation algorithm powered with algorithmic randomness produces the result not only true with high probability, but rigourously correct.
- Are more than Turing uncomputable.

Algorithmically random strings

- Satisfy all computable enumerable statistical properties of randomness.
- Are unpredictable.
- Every “decent” Monte Carlo simulation algorithm powered with algorithmic randomness produces the result not only true with high probability, but rigourously correct.
- Are more than Turing uncomputable.

Characteristics of quantum randomness

- It is postulated, not derived.
- It has been confirmed by theoretical and experimental research.
- It passes all reasonable statistical properties of randomness.
- Can be easily and reliably produced: A photon generated by a source beamed to a semitransparent mirror is reflected or transmitted with 50 per cent chance, and these measurements can be translated into a string of quantum random bits.
- It is Turing uncomputable.

Characteristics of quantum randomness

- It is postulated, not derived.
- It has been confirmed by theoretical and experimental research.
- It passes all reasonable statistical properties of randomness.
- Can be easily and reliably produced: A photon generated by a source beamed to a semitransparent mirror is reflected or transmitted with 50 per cent chance, and these measurements can be translated into a string of quantum random bits.
- It is Turing uncomputable.

Characteristics of quantum randomness

- It is postulated, not derived.
- It has been confirmed by theoretical and experimental research.
- It passes all reasonable statistical properties of randomness.
- Can be easily and reliably produced: A photon generated by a source beamed to a semitransparent mirror is reflected or transmitted with 50 per cent chance, and these measurements can be translated into a string of quantum random bits.
- It is Turing uncomputable.

Characteristics of quantum randomness

- It is postulated, not derived.
- It has been confirmed by theoretical and experimental research.
- It passes all reasonable statistical properties of randomness.
- Can be easily and reliably produced: A photon generated by a source beamed to a semitransparent mirror is reflected or transmitted with 50 per cent chance, and these measurements can be translated into a string of quantum random bits. JPict
- It is Turing uncomputable.

Characteristics of quantum randomness

- It is postulated, not derived.
- It has been confirmed by theoretical and experimental research.
- It passes all reasonable statistical properties of randomness.
- Can be easily and reliably produced: A photon generated by a source beamed to a semitransparent mirror is reflected or transmitted with 50 per cent chance, and these measurements can be translated into a string of quantum random bits. ▸ JPict
- It is Turing uncomputable.

Quantis

Quantis: quantum mechanical random number generator produced and sold by *id Quantique* of the University of Geneva

“PC plus Quantis”

The hybrid computer “PC plus Quantis” exists and was and it is used.
For example, Calude, Câmpeanu and Dumitrescu (2005) showed, via a statistical analysis, that with a high degree of accuracy (i.e., with precision higher than 99% and level of confidence 0.9973), the probability that a finite automaton recognizes no word tends to zero when the number of states and the number of letters in the alphabet tend to infinity.

“PC plus Quantis”

The hybrid computer “PC plus Quantis” exists and was and it is used. For example, Calude, Câmpeanu and Dumitrescu (2005) showed, via a statistical analysis, that with a high degree of accuracy (i.e., with precision higher than 99% and level of confidence 0.9973), the probability that a finite automaton recognizes no word tends to zero when the number of states and the number of letters in the alphabet tend to infinity.

Open questions

1. How random is quantum randomness? More precisely, to what extent is quantum randomness algorithmic randomness?
2. How accurate are Monte Carlo simulations powered by quantum random bits?
3. What is the computational power of the hybrid machine “PC plus Quantis”?

Open questions

1. How random is quantum randomness? More precisely, to what extent is quantum randomness algorithmic randomness?
2. How accurate are Monte Carlo simulations powered by quantum random bits?
3. What is the computational power of the hybrid machine "PC plus Quantis"?

Open questions

1. How random is quantum randomness? More precisely, to what extent is quantum randomness algorithmic randomness?
2. How accurate are Monte Carlo simulations powered by quantum random bits?
3. What is the computational power of the hybrid machine “PC plus Quantis”?

Open questions

1. How random is quantum randomness? More precisely, to what extent is quantum randomness algorithmic randomness?
2. How accurate are Monte Carlo simulations powered by quantum random bits?
3. What is the computational power of the hybrid machine “PC plus Quantis”?

Outline

Probabilistic primality tests

Three types of randomness

Using uncomputability for computation

Probabilistic understanding of the halting problem

Selected references

The halting problem

The halting problem for Turing machines is the problem to decide whether an arbitrary Turing machine T (coded by a string, $code(T)$) eventually halts on an arbitrary input x.

Does there exist a Turing machine T_{halt} which given $code(T)$ and x, eventually stops and produces 1 if $T(x)$ stops and 0 if $T(x)$ does not stop?

Turing's famous result states that this problem cannot be solved by any Turing machine, i.e. there is no such T_{halt}.

All uncomputability results previously discussed can be reduced to the halting problem.

The halting problem

The halting problem for Turing machines is the problem to decide whether an arbitrary Turing machine T (coded by a string, $code(T)$) eventually halts on an arbitrary input x.

Does there exist a Turing machine T_{halt} which given $code(T)$ and x, eventually stops and produces 1 if $T(x)$ stops and 0 if $T(x)$ does not stop?

Turing's famous result states that this problem cannot be solved by any Turing machine, i.e. there is no such T_{halt}.

All uncomputability results previously discussed can be reduced to the halting problem.

The halting problem

The halting problem for Turing machines is the problem to decide whether an arbitrary Turing machine T (coded by a string, $code(T)$) eventually halts on an arbitrary input x.

Does there exist a Turing machine T_{halt} which given $code(T)$ and x, eventually stops and produces 1 if $T(x)$ stops and 0 if $T(x)$ does not stop?

Turing's famous result states that this problem cannot be solved by any Turing machine, i.e. there is no such T_{halt}.

All uncomputability results previously discussed can be reduced to the halting problem.

The halting problem

The halting problem for Turing machines is the problem to decide whether an arbitrary Turing machine T (coded by a string, $code(T)$) eventually halts on an arbitrary input x.

Does there exist a Turing machine T_{halt} which given $code(T)$ and x, eventually stops and produces 1 if $T(x)$ stops and 0 if $T(x)$ does not stop?

Turing's famous result states that this problem cannot be solved by any Turing machine, i.e. there is no such T_{halt}.

All uncomputability results previously discussed can be reduced to the halting problem.

Fermat Last Theorem

It is perhaps surprising that many problems in mathematics can be reformulated in terms of the halting/non-halting status of appropriately constructed Turing machines.

Fermat's Last Theorem, stating that there is no integers $x, y, z, n > 3$ such that $x^n + y^n = z^n$, is an example.

We can construct a Turing machine T_{Fermat} which enumerates systematically all possible integers (for example, written in binary) $x, y, z, n > 3$, checks whether $x^n + y^n = z^n$, and stops if for some values x, y, z, n the relation is true; otherwise, T generates a new 4-tuple x, y, z, n and repeats the above procedure.

Fermat's Last Theorem is equivalent with the statement "T_{Fermat} never halts".

Fermat Last Theorem

It is perhaps surprising that many problems in mathematics can be reformulated in terms of the halting/non-halting status of appropriately constructed Turing machines.

Fermat's Last Theorem, stating that there is no integers $x, y, z, n > 3$ such that $x^n + y^n = z^n$, is an example.

We can construct a Turing machine T_{Fermat} which enumerates systematically all possible integers (for example, written in binary) $x, y, z, n > 3$, checks whether $x^n + y^n = z^n$, and stops if for some values x, y, z, n the relation is true; otherwise, T generates a new 4-tuple x, y, z, n and repeats the above procedure.

Fermat's Last Theorem is equivalent with the statement "T_{Fermat} never halts".

Fermat Last Theorem

It is perhaps surprising that many problems in mathematics can be reformulated in terms of the halting/non-halting status of appropriately constructed Turing machines.

Fermat's Last Theorem, stating that there is no integers $x, y, z, n > 3$ such that $x^n + y^n = z^n$, is an example.

We can construct a Turing machine T_{Fermat} which enumerates systematically all possible integers (for example, written in binary) $x, y, z, n > 3$, checks whether $x^n + y^n = z^n$, and stops if for some values x, y, z, n the relation is true; otherwise, T generates a new 4-tuple x, y, z, n and repeats the above procedure.

Fermat's Last Theorem is equivalent with the statement "T_{Fermat} never halts".

Fermat Last Theorem

It is perhaps surprising that many problems in mathematics can be reformulated in terms of the halting/non-halting status of appropriately constructed Turing machines.

Fermat's Last Theorem, stating that there is no integers $x, y, z, n > 3$ such that $x^n + y^n = z^n$, is an example.

We can construct a Turing machine T_{Fermat} which enumerates systematically all possible integers (for example, written in binary) $x, y, z, n > 3$, checks whether $x^n + y^n = z^n$, and stops if for some values x, y, z, n the relation is true; otherwise, T generates a new 4-tuple x, y, z, n and repeats the above procedure.

Fermat's Last Theorem is equivalent with the statement "T_{Fermat} never halts".

A universal prefix-free machine 1

A register machine has a finite number of registers, each of which may contain an arbitrarily large non-negative binary integer. The register machine *U* (labelled) instructions are:

L: EQ R1 R2 R3
L: SET R1 R2
L: ADD R1 R2
L: READ R1
L: HALT

A universal prefix-free machine 2

A register machine program consists of a finite list of labelled instructions from the above list, with the restriction that the HALT instruction appears only once, as the last instruction of the list.

The input data (a binary string) follows immediately after the HALT instruction.

A program not reading the whole data or attempting to read past the last data-bit results in a runtime error. Some programs have no input data.

Theorem

The register machine U is prefix-free and universal.

A universal prefix-free machine 2

A register machine program consists of a finite list of labelled instructions from the above list, with the restriction that the HALT instruction appears only once, as the last instruction of the list.

The input data (a binary string) follows immediately after the HALT instruction.

A program not reading the whole data or attempting to read past the last data-bit results in a runtime error. Some programs have no input data.

Theorem

The register machine U is prefix-free and universal.

A universal prefix-free machine 2

A register machine program consists of a finite list of labelled instructions from the above list, with the restriction that the HALT instruction appears only once, as the last instruction of the list.

The input data (a binary string) follows immediately after the HALT instruction.

A program not reading the whole data or attempting to read past the last data-bit results in a runtime error. Some programs have no input data.

Theorem

The register machine U is prefix-free and universal.

A universal prefix-free machine 2

A register machine program consists of a finite list of labelled instructions from the above list, with the restriction that the HALT instruction appears only once, as the last instruction of the list.

The input data (a binary string) follows immediately after the HALT instruction.

A program not reading the whole data or attempting to read past the last data-bit results in a runtime error. Some programs have no input data.

Theorem

The register machine U is prefix-free and universal.

Goldbach's Conjecture, Riemann Hypothesis, Collatz's Conjecture

The program T_{Goldbach}, which never stops if Goldbach's Conjecture is true, has 135 instructions totalling 3,484 bits.

The program T_{Riemann}, which never stops if the Riemann Hypothesis is true, consists of 290 instructions totalling 7,780 bits.

There is a *non-constructive* way to prove that there exists a program T_{Collatz} which never stops iff the Collatz' Conjecture is true.

Goldbach's Conjecture, Riemann Hypothesis, Collatz's Conjecture

The program T_{Goldbach}, which never stops if Goldbach's Conjecture is true, has 135 instructions totalling 3,484 bits.

The program T_{Riemann}, which never stops if the Riemann Hypothesis is true, consists of 290 instructions totalling 7,780 bits.

There is a *non-constructive* way to prove that there exists a program T_{Collatz} which never stops iff the Collatz' Conjecture is true.

Goldbach’s Conjecture, Riemann Hypothesis, Collatz’s Conjecture

The program T_{Goldbach}, which never stops if Goldbach’s Conjecture is true, has 135 instructions totalling 3,484 bits.

The program T_{Riemann}, which never stops if the Riemann Hypothesis is true, consists of 290 instructions totalling 7,780 bits.

There is a *non-constructive* way to prove that there exists a program T_{Collatz} which never stops iff the Collatz’ Conjecture is true.

What's the point?

Given that the halting problem is undecidable . . .

- First, we have a measure of the difficulty of problems which can be used for finitely refutable conjectures.
- Secondly, can we "attack" the halting problem, maybe in a different way?

What's the point?

Given that the halting problem is undecidable . . .

- First, we have a measure of the difficulty of problems which can be used for finitely refutable conjectures.
- Secondly, can we "attack" the halting problem, maybe in a different way?

What's the point?

Given that the halting problem is undecidable . . .

- First, we have a measure of the difficulty of problems which can be used for finitely refutable conjectures.
- Secondly, can we “attack” the halting problem, maybe in a different way?

Outline

Probabilistic primality tests

Three types of randomness

Using uncomputability for computation

Probabilistic understanding of the halting problem

Selected references

The Omega number 1

Recall: The register machine U is prefix-free and universal.

Assume that programs are uniformly distributed. Chaitin's Omega, the "halting probability of U", is:

$$\Omega_U = \sum_{p \in \mathrm{dom}(U)} 2^{-|p|}.$$

Theorem

The number Ω_U is algorithmically random, i.e. if $\Omega_U = 0.\omega_1\omega_2\cdots$, then for all $n \geq 1$, $H_U(\omega_1\omega_2\cdots\omega_n) \geq n -$ constant.

Theorem

With the first n bits of Ω_U we can solve the halting problem for every program of length less than or equal to n.

The Omega number 1

Recall: The register machine U is prefix-free and universal. Assume that programs are uniformly distributed. Chaitin's Omega, the "halting probability of U", is:

$$\Omega_U = \sum_{p \in \mathrm{dom}(U)} 2^{-|p|}.$$

Theorem

The number Ω_U is algorithmically random, i.e. if $\Omega_U = 0.\omega_1\omega_2\cdots$, then for all $n \geq 1$, $H_U(\omega_1\omega_2\cdots\omega_n) \geq n -$ constant.

Theorem

With the first n bits of Ω_U we can solve the halting problem for every program of length less than or equal to n.

The Omega number 1

Recall: The register machine U is prefix-free and universal. Assume that programs are uniformly distributed. Chaitin's Omega, the "halting probability of U", is:

$$\Omega_U = \sum_{p \in \mathrm{dom}(U)} 2^{-|p|}.$$

Theorem

The number Ω_U is algorithmically random, i.e. if $\Omega_U = 0.\omega_1\omega_2\cdots$, then for all $n \geq 1$, $H_U(\omega_1\omega_2\cdots\omega_n) \geq n -$ constant.

Theorem

With the first n bits of Ω_U we can solve the halting problem for every program of length less than or equal to n.

The Omega number 1

Recall: The register machine U is prefix-free and universal.
Assume that programs are uniformly distributed. Chaitin's Omega, the "halting probability of U", is:

$$\Omega_U = \sum_{p \in \mathrm{dom}(U)} 2^{-|p|}.$$

Theorem
The number Ω_U is algorithmically random, i.e. if $\Omega_U = 0.\omega_1\omega_2\cdots$, then for all $n \geq 1$, $H_U(\omega_1\omega_2\cdots\omega_n) \geq n -$ constant.

Theorem
With the first n bits of Ω_U we can solve the halting problem for every program of length less than or equal to n.

The Omega number 2

Solving the halting problem for all programs up to 80 bits one can get:

Theorem [Calude, Dinneen, 2006]

The first 43 bits of Ω_U are:

0001000000010000101001110111000100000101110

Still, a long way till 3,484 bits for Goldbach's Conjecture!

Even worse,

Theorem

One can compute an integer N such that no Turing machine can compute any digit of Ω_U, ω_n, with $n > N$.

The Omega number 2

Solving the halting problem for all programs up to 80 bits one can get:

Theorem [Calude, Dinneen, 2006]

The first 43 bits of Ω_U are:

$$0001000000010000101001110111000100000101110$$

Still, a long way till 3,484 bits for Goldbach's Conjecture!

Even worse,

Theorem

One can compute an integer N such that no Turing machine can compute any digit of Ω_U, ω_n, with $n > N$.

The Omega number 2

Solving the halting problem for all programs up to 80 bits one can get:

Theorem [Calude, Dinneen, 2006]

The first 43 bits of Ω_U are:

0001000000010000101001110111000100000101110

Still, a long way till 3,484 bits for Goldbach's Conjecture!

Even worse,

Theorem

One can compute an integer N such that no Turing machine can compute any digit of Ω_U, ω_n, with $n > N$.

Can a program stop at an algorithmically random time? 1

Let $\text{bin}(i)$ be the ith non-empty binary string.

Theorem [Chaitin, 1987]

There is a constant c such that if $U(\text{bin}(i))$ halts exactly in time t, then $H(\text{bin}(t)) \leq |\text{bin}(i)| + c$.

Theorem [Calude, Stay, 2006]

Assume that an N-bit program $U(p)$ has not stopped by time $2^{2N+2c+1}$, where $N \geq 2$ and c comes from the above theorem. Then, $U(p)$ cannot stop exactly at any algorithmically random time $t \geq 2^{2N+2c+1}$.

Can a program stop at an algorithmically random time? 1

Let bin(i) be the ith non-empty binary string.

Theorem [Chaitin, 1987]

There is a constant c such that if $U(\text{bin}(i))$ halts exactly in time t, then $H(\text{bin}(t)) \leq |\text{bin}(i)| + c$.

Theorem [Calude, Stay, 2006]

Assume that an N-bit program $U(p)$ has not stopped by time $2^{2N+2c+1}$, where $N \geq 2$ and c comes from the above theorem. Then, $U(p)$ cannot stop exactly at any algorithmically random time $t \geq 2^{2N+2c+1}$.

Can a program stop at an algorithmically random time? 1

Let bin(i) be the ith non-empty binary string.

Theorem [Chaitin, 1987]

There is a constant c such that if $U(\text{bin}(i))$ halts exactly in time t, then $H(\text{bin}(t)) \leq |\text{bin}(i)| + c$.

Theorem [Calude, Stay, 2006]

Assume that an N-bit program $U(p)$ has not stopped by time $2^{2N+2c+1}$, where $N \geq 2$ and c comes from the above theorem. Then, $U(p)$ cannot stop exactly at any algorithmically random time $t \geq 2^{2N+2c+1}$.

Can a program stop at an algorithmically random time? 2

A time t is called "exponential stopping time" if there is a program p which stops on U exactly at $t_p > 2^{2|p|+2c+1}$.

Theorem [Calude, Stay, 2006]

The set of exponential stopping times has effectively zero density.

Can a program stop at an algorithmically random time? 2

A time t is called "exponential stopping time" if there is a program p which stops on U exactly at $t_p > 2^{2|p|+2c+1}$.

Theorem [Calude, Stay, 2006]

The set of exponential stopping times has effectively zero density.

Halting according to a computable time distribution 1

Postulate *an a priori computable probability distribution on all possible runtimes*. Consequently, the probability space is the product of the space of programs and the time space.

More precisely, the probability space is

$$Space_{\{\rho_i\}} = B^* \times \{1, 2, \ldots\},$$

where N-bit programs are assumed to be uniformly distributed, and the observer time flows according to the computable probability distribution $\{\rho_i\}$.

Halting according to a computable time distribution 1

Postulate *an a priori computable probability distribution on all possible runtimes*. Consequently, the probability space is the product of the space of programs and the time space.

More precisely, the probability space is

$$Space_{\{\rho_i\}} = B^* \times \{1, 2, \ldots\},$$

where N-bit programs are assumed to be uniformly distributed, and the observer time flows according to the computable probability distribution $\{\rho_i\}$.

Halting according to a computable time distribution 3

Theorem [Calude, Stay, 2006]

Consider that the observer time flows from 1 to ∞ according to a probability distribution ρ_i which effectively converges to 1. Then, the probability that an N-bit program which hasn't stopped on U by time t_k (which can be effectively computed) will eventually halt effectively converges to zero.

Halting according to a computable time distribution 3

Here is an example.
For every program bin(i) let $t_{\mathrm{bin}(i)}$ be the exact time $U(\mathrm{bin}(i))$ stops, if $U(\mathrm{bin}(i))$ halts, or $t_{\mathrm{bin}(i)} = \infty$, otherwise.

Define the computable number:

$$0 < \Upsilon_U = \sum_{i \geq 1} 2^{-i} / t_{\mathrm{bin}(i)} < 1.$$

Then, define the computable probability distribution

$$\rho_i = \frac{2^{-i}}{t_{\mathrm{bin}(i)} \cdot \Upsilon_U}.$$

Halting according to a computable time distribution 3

Here is an example.
For every program $\text{bin}(i)$ let $t_{\text{bin}(i)}$ be the exact time $U(\text{bin}(i))$ stops, if $U(\text{bin}(i))$ halts, or $t_{\text{bin}(i)} = \infty$, otherwise.

Define the computable number:

$$0 < \Upsilon_U = \sum_{i \geq 1} 2^{-i} / t_{\text{bin}(i)} < 1.$$

Then, define the computable probability distribution

$$\rho_i = \frac{2^{-i}}{t_{\text{bin}(i)} \cdot \Upsilon_U}.$$

Halting according to a computable time distribution 3

Here is an example.
For every program $\mathrm{bin}(i)$ let $t_{\mathrm{bin}(i)}$ be the exact time $U(\mathrm{bin}(i))$ stops, if $U(\mathrm{bin}(i))$ halts, or $t_{\mathrm{bin}(i)} = \infty$, otherwise.

Define the computable number:

$$0 < \Upsilon_U = \sum_{i \geq 1} 2^{-i} / t_{\mathrm{bin}(i)} < 1.$$

Then, define the computable probability distribution

$$\rho_i = \frac{2^{-i}}{t_{\mathrm{bin}(i)} \cdot \Upsilon_U}.$$

Halting according to a computable time distribution 4

Theorem [Calude, Stay, 2006]

Assume that $U(p)$ has not stopped by time $T > k - \lfloor \log \Upsilon_U \rfloor$. Then, the probability (according to the probability space $Space_{\{\rho_i\}}$) that $U(p)$ eventually halts is smaller than 2^{-k}.

Theorem [Calude, Stay, 2006]

Assume that U and $Space_{\{\rho(i)\}}$ have been fixed. For every integer $k > 0$, the set of halting programs for U can be written as a disjoint union of a computable set and a set of probability effectively smaller than 2^{-k}.

Halting according to a computable time distribution 4

Theorem [Calude, Stay, 2006]

Assume that $U(p)$ has not stopped by time $T > k - \lfloor \log \Upsilon_U \rfloor$. Then, the probability (according to the probability space $Space_{\{\rho_i\}}$) that $U(p)$ eventually halts is smaller than 2^{-k}.

Theorem [Calude, Stay, 2006]

Assume that U and $Space_{\{\rho(i)\}}$ have been fixed. For every integer $k > 0$, the set of halting programs for U can be written as a disjoint union of a computable set and a set of probability effectively smaller than 2^{-k}.

Grand open problem

Open problem

Can the above analysis be used to develop a probabilistic solution for the halting problem?

Outline

Probabilistic primality tests

Three types of randomness

Using uncomputability for computation

Probabilistic understanding of the halting problem

Selected references

Books

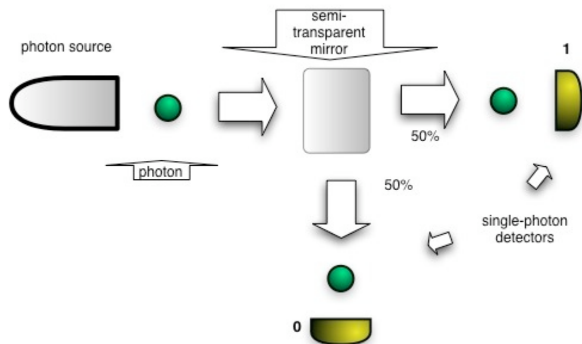
- C. S. Calude. *Information and Randomness – An Algorithmic Perspective*, Springer-Verlag 2002.
- G. J. Chaitin. *Meta Math!*, Pantheon, 2005.
- M. Sipser. *Introduction to the Theory of Computation*, PWS 1997.

Papers 1

- C. S. Calude. Algorithmic randomness, quantum physics, and incompleteness, LNCS 3354, Springer, Berlin, 2005, 1–17.
- C. S. Calude, C. Câmpeanu, Monica Dumitrescu. Automata recognizing no words: A statistical approach, *Fundamenta Informaticae* 72 (2006), 1–18.
- C. S. Calude, Elena Calude, M. J. Dinneen. A new measure of the difficulty of problems, *Journal for Multiple-Valued Logic and Soft Computing* 10 (2006), 1–21.
- C. S. Calude, J. Casti. The jumble cruncher, *New Scientist* 25 September 2004, 36–37.

Papers 2

- C. S. Calude, M. J. Dinneen and C.-K. Shu. Computing a glimpse of randomness, *Experimental Mathematics* 11, 2 (2002), 369–378.
- C. S. Calude, B. Pavlov. Coins, quantum measurements, and Turing's barrier, *Quantum Information Processing* 1, 1–2 (2002), 107–127.
- C. S. Calude, M. A. Stay. De-Quantising Non-Halting Programs, *CDMTCS Research Report* 284, 2006, 17 pp.
- G. J. Chaitin. Computing the busy beaver function, in T. M. Cover, B. Gopinath (eds.). *Open Problems in Communication and Computation*, Springer-Verlag, Heidelberg, 1987, 108–112.



Optical system for generating quantum random bits