# Guia prático para ensaístas de laboratórios na busca por evidências contra a imparcialidade de geradores de números aleatórios

Daniel Chicayban Bastos
Luis Antonio Brasil Kowada*

Maio 2020

# Apresentação

Este documento é formado por três partes: esta apresentação, uma primeira e uma segunda parte. Nesta apresentação, veremos a operação de programas de computador com o propósito de avaliar a qualidade da aleatoriedade de geradores de números aleatórios, incluindo tópicos que diretamente se relacionam com eles. Na primeira parte, fazemos uma pequena digressão em tópicos que um profissional dificilmente conseguiria evitar. É preciso ter um mínimo de intimidade com arquitetura de computadores, por exemplo. Já na segunda parte, investigamos brevemente as dependências teóricas para um melhor entendimento do vocabulário usado pelas ferramentas apresentadas. Cabe notar que os tópicos são objeto de estudo tipicamente lecionados por universidades ao longo de meses; não se propõe que a informação apresentada possa suprir tais disciplinas acadêmicas. Todavia, esforço é feito para que o vocabulário e os procedimentos efetuados pelas ferramentas tenham pelo menos suas formas compreendidas, ou seja, objetiva-se que o leitor perceba que os procedimentos utilizados vêm de métodos bem planejados.

Por depender de algumas ferramentas, descrevemos na Seção 1 as ferramentas necessárias para um maior acompanhamento do conteúdo. A Seção 2 descreve o que pensamos ser a provável forma da comunicação entre laboratórios e fornecedores. A Seção 3 efetivamente faz um passeio pelo uso das ferramentas como uma primeira exposição ao tema, encerrando a primeira parte deste documento.

A segunda parte é composta pelas Seções 4 e 5, preocupando-se aquela em apresentar definições relativas a geradores de números aleatórios que se envolvem com a arquitetura de computadores, enquanto esta visa a descrever procedimentos estatísticos análogos aos que são usados pelas ferramentas.

## Que relação existe entre segurança e números aleatórios?

Na década de noventa, um problema de segurança foi descoberto em certa versão do navegador da Netscape. A raiz do problema era uma indesejável semeadura do gerador de números aleatórios que o navegador usava [10]. É comum se usar geradores de números aleatórios cuja sequência produzida seja reproduzível usando-se uma informação que é sugestivamente chamada de *seed*, palavra inglesa que se traduz para "semente". Tendo a semente de um gerador, têm-se toda sequência produzida, fato que explica a escolha da palavra: não é estranha a ideia de que uma planta esteja — de certa forma — completamente codificada por sua semente.

Certa implementação do navegador semeava o gerador de números aleatórios usando três informações: a hora corrente (em precisão de microsegundos), o número de processo do navegador no sistema operacional e o número do processo que deu nascimento ao processo do navegador. Se o adversário

*Autores em ordem alfabética.

fosse capaz de prever — de qualquer forma — esses três números, ele descobriria a semente do gerador e, portanto, toda a sequência de números aleatórios produzidos pelo gerador.

Um usuário local no sistema em que o navegador executa consegue facilmente obter o número do processo do navegador bem como o número do processo que deu nascimento ao processo do navegador. Resta ao usuário local apenas descobrir a que hora exatamente o navegador semeou o gerador. Descobrir a hora exata em que o navegador faz a semeadura também não é muito difícil. O administrador do sistema, por exemplo, tem acesso privilegiado sobre a interface de rede e, assim, é capaz de analisar o tráfego relativo ao navegador. As ferramentas de análise de tráfego registram a hora exata em que elas veem cada pacote, o que dá ao administrador do sistema uma noção bem exata de que momento cada evento ocorre pelo menos na precisão de um segundo, restando a dificuldade de encontrar os microsegundos usados pelo navegador. Um segundo é um milhão de microsegundos. Assim, podemos dizer que o adversário tem um universo de busca de um milhão, o que certamente não é muito. Vejamos o quão rápido é recuperar uma informação num universo de busca de um milhão. Seja a semente um número secreto entre um e um milhão. Os primeiros números aleatórios produzidos por `random.random`, que é um procedimento incluído na distribuição padrão da linguagem Python, são 0,5845119624895062, 0,8112534844785401, ... Para descobrir a semente, só precisamos testar todas as sementes no intervalo $[1, 10^6]$, o que pode ser feito pelo seguinte procedimento.

```
def search():
  import random
  from timeit import default_timer as timer
  from datetime import timedelta
  start = timer()
  for c in range(1, 10**6 + 1):
    random.seed(c)
    if random.random() == 0.5845119624895062:
      end = timer()
      return c, str(timedelta(seconds = end - start))
  return None, str(timedelta(seconds = end - start))
```

A semente que produzir o número `0.5845119624895062` é a semente desejada. O procedimento custa alguns segundos num computador típico.

```
>>> search()
(267460, '0:00:02.325965')
>>> random.seed(267460)
>>> random.random()
0.5845119624895062
>>> random.random()
0.8112534844785401
```

O procedimento `search` fez a busca e encontrou corretamente a semente `267460`, o que é evidenciado pelos dois primeiros números aleatórios da sequência produzida pelo gerador padrão embutido na linguagem de programação.

Retornando à implementação do navegador, efetuada a primeira conexão com o servidor web, um número computado em função dos números aleatórios era passado sem criptografia para o servidor web. Esse número foi chamado de `challenge`. O adversário só precisava descobrir qual era a semente que gerava o `challenge`. Eis essencialmente o procedimento usado por essa versão do navegador para computar o `challenge`, o número que passava pela rede sem criptografia e era escolhido pelo gerador de números aleatórios [10, "Netscape's Implementation"].

```
procedure RNG_GenerateRandomBytes():
  x = MD5(seed)
  seed = seed + 1
```

```
  return x

global variable challenge, secret_key

procedure create_key():
  RNG_CreateContext()
  tmp = RNG_GenerateRandomBytes()
  tmp = RNG_GenerateRandomBytes()
  challenge = RNG_GenerateRandomBytes()
  secret_key = RNG_GenerateRandomBytes()
```

O procedimento `RNG_GenerateRandomBytes` representa o gerador de números aleatórios e o procedimento `create_key` usa o gerador para gerar a chave secreta. O gerador era essencialmente o algoritmo MD5, que na década de noventa ainda era uma função *hash* segura. O gerador em si é irrelevante aqui; não precisamos entender o que é uma função *hash*, por exemplo. O que importa é que a chave secreta era o próximo número após o `challenge`. Seja qual for o gerador, se conhecermos a semente, conheceremos a sequência. Logo, bastava ao adversário descobrir a semente que gerava o `challenge` que ele teria a chave secreta.

Vimos há pouco que todas as entradas usadas para computar a semente eram de fácil acesso a um adversário com acesso privilegiado ao sistema. A única informação desconhecida estava em um universo de um milhão de possibilidades. A segurança para esse adversário estava reduzida a um minúsculo universo.

Todos os detalhes da implementação dessa versão do Netscape foram descobertos sem acesso ao código-fonte; engenharia reversa foi feita. As evidências mostram que quando há um meio viável de se encontrar uma informação, ela será encontrada. Não se deve subestimar a viabilidade. Se aumentarmos o universo de busca de um milhão para $10^{617}$, que é o equivalente a 2048 bits de segurança, e escolhermos sementes verdadeiramente aleatórias nesse intervalo, acredita-se haver a inviabilidade de reproduzir a sequência de números aleatórios usada nos procedimentos. Isso hoje; não está clara qual será a capacidade de computação de um adversário no futuro.

Se a semente é obtida de forma verdadeiramente secreta, verdadeiramente aleatória, mas o gerador em si possui graves falhas e exibe um viés bem conhecido pelo adversário, a segurança que se pensaria existir no universo de busca é ilusória porque o viés do gerador reduz o universo de busca. Se reduzi-lo significamente, pode aparecer a viabilidade computacional que o adversário deseja. Por exemplo, suponha que um gerador seja capaz de produzir números aleatórios num certo intervalo. O universo de busca, portanto, é a extensão do intervalo. Mas suponha que se descubra que o gerador tem o viés de sempre gerar números pares e ímpares de forma alternada. Então, a cada número emitido pelo gerador, o universo de busca é metade do que seria se ele não tivesse o viés, afinal a cada número produzido sabemos a paridade do próximo.

## 1 As ferramentas necessárias

O leitor deve ter disponível um sistema de computador com as ferramentas que serão usadas ao longo do documento. O sistema pode ser qualquer um compatível com UNIX. Sistemas Windows e computadores recentes da Apple também servem uma vez que são todos suficientemente compatíveis com UNIX, mas as ferramentas que usamos neste documento nem sempre estão imediatamente disponíveis em todos os sistemas. Eis as ferramentas que usamos.

1. Um compilador C. O estado da arte em testes estatísticos disponíveis hoje são programas escritos em C. Usualmente esses programas não estão disponíveis em executáveis. Portanto, é necessário que o leitor consiga compilá-los em seu sistema.

2. Um compilador Python. Usamos a linguagem Python para ilustrar o uso de um gerador de números aleatórios. Em sentido estrito não é uma ferramenta necessária, mas é recomendável que

o leitor acompanhe em seu próprio computador os passos executados ao longo deste documento para um maior aproveitamento.

3. O programa `od`. É um programa tipicamente disponível em sistemas UNIX. Sua utilidade está em ler dados binários e exibi-los de foma textual para leitura e análise. O programa `od` é parte do pacote GNU `coreutils`[1]. Se seu sistema não tiver o `od` prontamente disponível, instale o pacote.

4. A ferramenta `crush` [2, 3], cuja instalação é descrita na Seção 4.7.

5. A biblioteca TestU01 [21, 19, 22], cuja instalação é descrita na Seção 4.7.

6. O programa `wget`. Usamos o `wget` para fazer a cópia de arquivos disponíveis na Internet. Outros programas como `curl` e um navegador gráfico também servirão ao mesmo propósito.

7. O programa `make`. Embora não estritamente necessário, usar o `make` nos permite dirigir o compilador padrão do sistema com suficiente conforto.

8. Um compilador C++. Necessário apenas se o leitor desejar compilar o *PractRand* [7]. Para sistemas Windows de 64 bits, há disponível um executável já compilado[2].

## 2 Um protocolo de comunicação entre laboratórios e fornecedores

A atividade laboratorial de um ensaísta provavelmente será a execução de testes estatísticos e a interpretação dos resultados. A execução de testes é tão simples quanto a execução de um programa. Entretanto, para que a operação se desenvolva com essa simplicidade, alguma harmonia entre entre laboratório e fornecedores é necessária.

Considere um fabricante que deseje submeter quatro roteadores sem fio à análise por um laboratório. Sejam eles os roteadores $A$, $B$, $C$ e $D$. Suponha que todos venham embutidos com geração de números aleatórios: os roteadores são capazes de comunicação criptografada, o que requer a geração aleatória de números com o objetivo de distribuir uniformemente a escolha de chaves secretas para a criptografia. A diferença relevante entre os produtos $A$, $B$, $C$ e $D$ é que ambos $A$ e $B$ usam um gerador de números aleatórios bem conhecido e são baseados em sistemas operacionais bem conhecidos, enquanto $C$ usa um gerador bem conhecido baseado num sistema operacional não popular e $D$ usa um gerador proprietário, baseado em *hardware* desenvolvido pelo próprio fabricante, contendo segredo empresarial.

**O caso $A$ e $B$.** O gerador embutido em $A$ é bem conhecido. O roteador $A$ é baseado num sistema operacional que usa o *kernel* Linux na versão 5.3.11 compilado para a arquitetura x86 de 64 bits. O gerador usado por $A$ é o `/dev/urandom`[3] provido pelo próprio *kernel*. Ao receber a especificação completa do gerador usado por $A$, o laboratório verifica que possui um sistema idêntico ao usado por $A$.

```
%uname -rs
Linux 5.3.11-x86_64-linode131
```

Sendo assim, nenhuma concessão de equipamento é necessária. O teste de $A$ se resume à execução de comandos num sistema já disponível no laboratório. O laboratório efetua os testes e produz um relatório final para o fornecedor contendo os resultados.

O roteador $B$ é muito parecido com $A$. A diferença é que $B$ é uma versão de $A$ mais nova, contando com uma atualização do sistema operacional, trazendo modificações inclusive no gerador de números aleatórios. O laboratório verifica que não possui um sistema idêntico ao especificado pelo fabricante. Em virtude disso, o laboratório requer ao fabricante a disponibilização de um sistema que seja equivalente ao utilizado por $B$. Sendo de fácil virtualização, o fabricante se propõe a enviar ao laboratório seu sistema em uma máquina virtual, tornando possível ao laboratório a instalação dos testes na máquina virtual e a realização dos testes.

[1] Homepage do `coreutils`: http://gnu.org/software/coreutils
[2] Copie-o de `https://dbastos0.github.io/crush/files/RNG_test.exe`.
[3] Código-fonte disponível em `https://bit.ly/3bvO8jD`.

**O caso $C$.** O gerador embutido em $C$ é um gerador bem conhecido, mas o roteador $C$ é baseado num sistema operacional proprietário cuja virtualização não é fácil de fazer. A dificuldade em reproduzir o sistema de $C$ é contornável porque $C$ usa um gerador bem conhecido. O comportamento do gerador de $C$ é facilmente implementável em um programa de computador que seja executável em algum sistema disponível no laboratório. Sendo assim, o fabricante fornece o código-fonte junto com instruções de preparação de um executável que permite ao laboratório a adequada experimentação do gerador. Por exemplo, se `gerador-c` é o nome do programa que implementa o gerador usado pelo roteador $C$, então o laboratório consegue testá-lo com o testador `crush` executando uma linha de comando.

```
%./gerador-c | crush --battery big --name gerador-c
[...]
```

**O caso $D$.** O gerador embutido em $D$ é proprietário e contém segredo empresarial desenvolvido pelo fabricante. O laboratório não tem qualquer informação sobre o gerador ou qualquer acesso ao produto. Os testes estatísticos são implementados por programas de computador, o que implica que o gerador a ser testado precisa estar disponível de alguma forma no sistema onde os testadores serão executados. O caso do gerador $D$ é consideravelmente mais elaborado porque demanda o fornecimento de um equipamento dedicado ao teste por parte do laboratório. É concebível, entretanto, que o *hardware* desenvolvido pela empresa para a geração dos números aleatórios possua um *driver* de controle do *hardware* também desenvolvido pela empresa. Assim, não é absurdo esperar que o fabricante tenha algum meio de comunicação entre o *hardware* projetado por ele e outros sistemas operacionais como os que estarão disponíveis no laboratório. É necessário algum meio de levar a informação aleatória (produzida pelo *hardware* do fabricante) aos programas que implementam os testes estatísticos.

O caso $D$ é o de um produto protegido por segredo empresarial e mesmo assim é passível de verificação estatística de sua reinvindicada aleatoriedade. Entretanto, verificações da entropia produzida pelo gerador não pode propriamente ser estimada sem uma descrição teórica do método implementado pelo gerador. A Seção 6 descreve o porquê.

**Um protocolo**. Em todos os casos concebidos, percebe-se a necessidade de uma especificação completa da produção dos números aleatórios e da disponibilização de equipamento capaz de produzir gerador equivalente ao usado pelo produto sob análise de forma que o laboratório consiga levar os dados aleatórios às ferramentas que implementam os testes estatísticos. Se o produto faz uso de geradores populares de acesso imediato ao laboratório, apenas a especificação do gerador usado pelo produto se faz necessária. Se o laboratório não tem acesso imediato ao gerador usado pelo produto, ao fabricante é requerido a produção de um programa que implemente o gerador. Se o gerador em questão é implementado em *hardware*, ao fabricante é requerido um meio de comunicação com os sistemas disponíveis no laboratório.

Sendo assim, é um protocolo mínimo aquele em que o fabricante fornece as informações necessárias para os testes e o laboratório provê um relatório final sobre os resultados do teste. A parte técnica do relatório é essencialmente provida pelas ferramentas de testes estatísticos, cabendo ao ensaísta do laboratório a organização e a interpretação dos resultados.

## 3 Sobre os testes de geradores

O objetivo deste guia é instruir o leitor minimamente sobre a análise da qualidade de geradores de números aleatórios, capacitando-o a executar uma avaliação de um produto com respeito ao gerador de números aleatórios utilizado, ou seja, este documento almeja prover a informação mínima necessária e as ferramentas mínimas suficientes para que um profissional possa investigar a aleatoriedade embutida em um produto.

Testar um gerador de números aleatórios que esteja implementado em um programa disponível no sistema produzindo números no formato adequado é tão simples quanto executar um comando como, por exemplo, a execução do programa `crush`.

```
%./gerador-x | crush --battery big --name gerador-x
```

```
[...]
========= Summary results of BigCrush =========
 Version:          TestU01 1.2.3
 Generator:        gerador-x
 Number of statistics:  [...]
 Total CPU time:   [...]
 All tests were passed
```

O comando executa o programa `gerador-x`, que implementa o gerador sob teste. A saída padrão do `gerador-x` é conectada à entrada do programa `crush`, cuja tarefa é ler os dados e aplicar seus testes estatísticos. A opção `battery` representa a escolha da bateria de testes, `BigCrush`[4], da biblioteca TestU01, o estado da arte em testes estatísticos para a avaliação da qualidade de geradores de números aleatórios. A opção `name` permite ao usuário identificar o relatório produzido pela ferramenta — em vez de colocar o nome do gerador, talvez o usuário queira escrever a data corrente, por exemplo. De acordo com o resultado do relatório acima, nenhuma evidência contra a aleatoriedade do `gerador-x` foi encontrada pela `BigCrush` da biblioteca TestU01. É o mínimo sucesso que se espera hoje de um gerador de números aleatórios sem restringir sua aplicação.

Entretanto, mera execução de um programa não é conhecimento suficiente. No caso hipotético de se testar o `gerador-x`, assume-se no exemplo que o programa `gerador-x` produz perfeitamente os bytes necessários para teste. É concebível que seja demandado ao profissional que faça e saiba fazer a operação de produzir o programa `gerador-x` a partir da informação sobre que gerador um produto utiliza. Se o produto se assemelha ao caso $D$ exposta na Seção 2, espera-se que o fornecedor produza o programa. Se for outro caso, não é absurdo que seja esperado do ensaísta que ele saiba escrever um programa de computador que implemente o gerador e tenha as propriedades necessárias para que o gerador seja testado. Por exemplo, o `gerador-x` sob teste na ilustração da execução da ferramenta `crush` é um programa que adequadamente escreve os bytes produzidos pelo algoritmo do gerador de números aleatórios da forma esperada pela ferramenta `crush`. Veremos os detalhes típicos envolvidos na tarefa e entenderemos o que é preciso fazer para que um programa como `gerador-x` seja corretamente preparado para se submeter aos testes. Antes dos detalhes, entretanto, vejamos como executar as ferramentas.

Assumindo que você esteja operando um sistema Windows, obtenha o executável `crush`[5] e salve-o em algum diretório como, por exemplo, `c:\intro`. Feito isso, verifique que temos um programa operacional.

```
c:\intro>crush --help
Usage: crush [options]
 Tests your data for randomness against TestU01.

Examples:
cat /dev/urandom | crush -b small -n 'the local generator'
xorshift32 | crush -b small -n xorshift32

 The options are:
-b, --battery    Your choice of battery (small, medium, big)
-n, --name       The name of your generator
-h, --help       Display this information
```

Obtenha também o programa `xorshift.exe`[6], que será o exemplo de gerador de números aleatórios que usaremos neste momento. Diferente do teste que fizemos com o `crush.exe`, não execute o `xorshift.exe`

[4] A diferença entre as baterias é exposta na página 7.

[5] Disponível em `https://dbastos0.github.io/crush/files/crush.exe`.

[6] Disponível em `https://dbastos0.github.io/crush/files/xorshift.exe`. O código-fonte `xorshift.c` pode ser obtido em `https://bit.ly/3gYMysm`.

sem uma linha de comando adequada porque o programa só faz imprimir bytes que em sua maior probabilidade não são imprimíveis na tela, o que produziria uma saída incoerente sem utilidade.

```
c:\intro>xorshift.exe
kÂª³)¼xõ/tÇGöU,Sc:Ç³º.çË[...]
```

Para testar a qualidade desses bytes supostamente aleatórios que o programa `xorshift.exe` é capaz de produzir, execute a seguinte linha de comando.

```
c:\intro>xorshift.exe | crush.exe --battery small --name xorshift
[...]
========= Summary results of SmallCrush =========

 Version:          TestU01 1.2.3
 Generator:        xorshift
 Number of statistics:  15
 Total CPU time:   00:00:10.71
 The following tests gave p-values outside [0.001, 0.9990]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-015):

       Test                          p-value
 ----------------------------------------------
  1  BirthdaySpacings                   eps
  2  Collision                          eps
  6  MaxOft                             eps
  6  MaxOft AD                       1 -  3.6e-06
  7  WeightDistrib                   9.1e-15
  8  MatrixRank                         eps
 10  RandomWalk1 H                      eps
 10  RandomWalk1 M                   3.3e-16
 10  RandomWalk1 J                   1.4e-15
 ----------------------------------------------
 All other tests were passed
[...]
```

A bateria que usamos é a pequena, chamada de `SmallCrush` pela biblioteca TestU01, cujo tempo de espera é menor que 1 minuto se seu computador tiver o processamento médio de hoje.

A diferença entre o tamanho das baterias é a quantidade de testes aplicados em cada bateria disponível bem como a configuração de cada teste. A biblioteca TestU01 criou três baterias — chamadas de `SmallCrush`, `Crush`, `BigCrush`. Em certo experimento num processador AMD Athlon 64 de frequência 2.4 GHz, a bateria `SmallCrush` custou 14 segundos, computando um total de 15 estatísticas, consumindo aproximadamente $2^{28}$ números aleatórios. A bateria `Crush` levou 1 hora no mesmo equipamento, computando 96 estatísticas, consumindo aproximadamente $2^{35}$ números aleatórios. A `BigCrush` levou 5.5 horas, computando 160 estatísticas[7], consumindo aproximadamente $2^{38}$ números aleatórios [19, página 24]. Temos então uma noção do custo de cada bateria, mas note que o tempo depende de quão rápido é o gerador.

Em vez de "`All tests were passed`", agora temos um relatório de fracassos e a mensagem "`All other tests were passed`". O testador chamado `crush.exe` só reporta no sumário os testes em cujo gerador `xorshift.exe` não passou. O nome dos testes em que o gerador fracassou estão listados: `BirthdaySpacings`, `Collision`, ... Cada teste executado produz um $p$-valor, que é o número que

[7] Um teste pode produzir mais de uma estatística. Na bateria `BigCrush`, 106 testes são usados, mas a quantidade de estatísticas é 160 [19, página 24].

aparece à direita de cada teste. Como informa o próprio relatório, o nome `eps` representa o número `1.0e-300`, que é o número 1 dividido por $10^{300}$, ou seja,

$$0.\underbrace{000000000000000000...}_{299\text{ zeros}}01,$$

um número bem pequeno. De fato, todos os $p$-valores exibidos por esse relatório são números bem pequenos, exceto o $p$-valor do teste `MaxOft AD`, que é $1 - 3{,}6 \times 10^{-6} \approx 0{,}9999$, um número bem próximo de 1. Como também informa o relatório da ferramenta, os testes reportados são os que produzem $p$-valores que estejam fora do intervalo $[0{,}001, 0{,}9990]$, ou seja, são os testes que fracassam. Um $p$-valor é sempre um número entre zero e um porque, assumindo que o gerador seja de fato aleatório, o $p$-valor descreveria a probabilidade de que a amostra coletada do gerador seja tão atípica ou mais atípica que a amostra coletada. Quando essa probabilidade é muito pequena, deparamo-nos com o fato de que obtemos uma amostra que é muito improvável de ser obtida e por isso desconfiamos da hipótese de que o gerador seja aleatório. Quando o $p$-valor é muito grande, próximo de 1, deparamo-nos com o fato de que obtemos uma amostra cuja estatística coletada pelo teste está próxima demais do valor esperado da estatística, o que também levanta suspeitas sobre a hipótese do gerador ser aleatório. A prática tem sido rejeitar também esses $p$-valores. Veremos a ciência por trás dessas interpretações mais à frente, mas já podemos entender como testar e interpretar um gerador de números aleatórios.

Na exibição do resultado acima, omitimos os resultados individuais de cada teste. Se você olhar o resultado para o teste `BirthdaySpacings`, que é o primeiro, encontrará o seguinte relato.

```
smarsa_BirthdaySpacings test:
-------------------------------------------------
   N =  1,  n = 5000000,  r =  0,    d = 1073741824,    t = 2,    p = 1
      Number of cells = d^t = 1152921504606846976
      Lambda = Poisson mean =      27.1051
---------------------------------------------------
Total expected number = N*Lambda      :      27.11
Total observed number                 :   16227
p-value of test                       :   eps      *****
```

A documentação do teste nos informa que o parâmetro `n = 5000000` reflete o tamanho da amostra coletada do gerador, ou seja, uma amostra de tamanho cinco milhões. Assumindo que o gerador de fato produza uma sequência aleatória de números, o valor esperado para a estatística computada pelo teste `BirthdaySpacings` seria de 27.11, mas o valor observado na amostra foi de 16227, o que é muito improvável de ocorrer numa sequência verdadeiramente aleatória como assume a hipótese. O que isso significa para o gerador *Xorshift* implementado pelo `xorshift.exe`? Significa que por mais que ele possa oferecer algum benefício de aleatoriedade, uma amostra de tamanho cinco milhões é grande demais para ele. Se o testássemos com uma amostra bem menor, talvez não fôssemos capazes de reprová-lo.

Isso nos traz a um ponto importante sobre geradores de números *pseudoaleatórios*. Nenhum deles é de fato aleatório. O benefício que os geradores pseudoaleatórios nos trazem é apenas a aparência de aleatoriedade. Sendo assim, é interessante medir a partir de que ponto o gerador começa a fracassar nos testes estatísticos. Qualquer gerador pseudoaleatório eventualmente fracassará em algum teste estatístico. A questão é que tamanho de amostra é suficiente para revelar seu fracasso. Por isso quanto maior são os tamanhos das amostras coletadas, mais eficaz é o teste. Sendo assim, uma característica interessante para uma ferramenta de testes é de que ela seja capaz de coletar amostras cada vez maiores até que o gerador fracasse em algum teste. Isso nos dá um limite superior para o gerador onde sabemos que a aparente aleatoriedade do gerador não se sustenta.

O estado da arte em testes estatísticos para geradores de números aleatórios é a biblioteca TestU01, que é a biblioteca usada pelo programa `crush.exe`. Há ferramentas que concorrem com a TestU01 como o *PractRand* [7] e o *gjrand* [14].

O programa `RNG_test.exe`[8] é o testador do pacote *PractRand*. Copie o programa para seu diretório

[8] Disponível em `https://dbastos0.github.io/crush/files/RNG_test.exe`.

para continuarmos. Observe a seguir que usar o *PractRand* é análogo a usar o `crush.exe`. Eis o resultado de testar o `xorshift.exe`[9].

```
c:\intro>xorshift.exe | RNG_test.exe stdin32
RNG_test using PractRand version 0.94
RNG = RNG_stdin32, seed = unknown
test set = core, folding = standard (32 bit)

rng=RNG_stdin32, seed=unknown
length= 128 megabytes (2^27 bytes), time= 3.0 seconds
  Test Name                         Raw       Processed     Evaluation
[...]
  BRank(12):128(4)                  R=+606.9  p~=  1.3e-323   FAIL !!!!!!
  BRank(12):256(4)                  R= +4223  p~=  4e-2247    FAIL !!!!!!!!
  BRank(12):384(1)                  R= +3877  p~=  4e-1168    FAIL !!!!!!!!
  BRank(12):512(2)                  R= +8162  p~=  5e-2458    FAIL !!!!!!!!
  BRank(12):768(1)                  R= +8182  p~=  3e-2464    FAIL !!!!!!!!
  BRank(12):1K(2)                   R=+15332  p~=  2e-4616    FAIL !!!!!!!!
  BRank(12):1536(1)                 R=+12380  p~=  7e-3728    FAIL !!!!!!!!
[...]
  ...and 75 test result(s) without anomalies
```

O relatório lista várias estatísticas de vários testes apontando o fracasso do gerador em amostras de tamanho 128 megabytes, isto é, $2^{27}$ bytes. Apenas como ilustração, omitimos todas as estatísticas exceto as relativas ao teste estatístico chamado de *binary rank* [24, 26] [19, página 19].

O relatório afirma que $2^{27}$ bytes é uma amostra grande demais para as qualidades do *Xorshift* de 32 bits. Reduzindo o tamanho das amostras até que nada contra o gerador seja relatado, encontramos o limite superior de $2^{15}$ bytes, isto é, 32 KiB.

```
c:\intro>xorshift.exe | RNG_test.exe stdin32 -tlmax 32KB
RNG_test using PractRand version 0.94
RNG = RNG_stdin32, seed = unknown
test set = core, folding = standard (32 bit)

rng=RNG_stdin32, seed=unknown
length= 32 kilobytes (2^15 bytes), time= 0.1 seconds
  no anomalies in 34 test result(s)
```

Isso sugere que $2^{15}$ bytes é um limite superior para o *Xorshift* 32. Podemos afirmar com segurança que, acima desse limite, o comportamento do gerador é evidentemente não-aleatório, afinal somos capazes de coletar amostras que são raríssimas de serem obtidas sob a hipótese de que o gerador seja aleatório, o que sugere que a hipótese é falsa.

Um limite superior a partir do qual o gerador fracassa em algum teste estatístico é uma medida útil porque, através dela, geradores podem ser comparados [28, seção 3].

Um gerador melhor que o *Xorshift* de 32 bits se desempenha bem melhor na mesma bateria de testes efetuada pelo *PractRand*. O programa `RNG_output.exe`, incluído no pacote, implementa uma série de geradores de números aleatórios, dentre os quais encontramos o `jsf32`[10]. Pedindo por $2^{30}$ bytes ao `RNG_output.exe` e ao `RNG_test.exe`, observamos que nada contra o gerador é encontrado, mas note que 1 GiB é uma amostra bem pequena para geradores de números aleatórios.

[9] No comando exibido, a opção `stdin32` instrui o testador de que os dados supostamente aleatórios virão pela entrada padrão, produzidos pelo `xorshift.exe`.

[10] O gerador foi escrito por Bob Jenkins e publicado em sua própria homepage no endereço `https://bit.ly/3kAoo9P`.

```
c:\intro>RNG_output.exe jsf32 1073741824 | RNG_test.exe stdin32 -tlmax 1GB
RNG_test using PractRand version 0.94
RNG = RNG_stdin32, seed = unknown
test set = core, folding = standard (32 bit)

rng=RNG_stdin32, seed=unknown
length= 256 megabytes (2^28 bytes), time= 3.0 seconds
  no anomalies in 165 test result(s)

rng=RNG_stdin32, seed=unknown
length= 512 megabytes (2^29 bytes), time= 6.3 seconds
  Test Name                         Raw       Processed     Evaluation
  [Low8/32]Gap-16:A                 R=  -4.6  p =1-2.7e-4   unusual
  ...and 177 test result(s) without anomalies

rng=RNG_stdin32, seed=unknown
length= 1 gigabyte (2^30 bytes), time= 12.6 seconds
  no anomalies in 192 test result(s)
```

Note que aos $2^{29}$ bytes, o teste `Gap-16` chama nossa atenção para alguma amostra não muito usual, mas não considera o fato um fracasso. De fato, subsequentes execuções nos convencem de que não há fortes evidências contra o gerador — até 1 GiB, isto é.

```
c:\intro>RNG_output.exe jsf32 1073741824 | RNG_test.exe stdin32 -tlmax 1GB
RNG_test using PractRand version 0.94
RNG = RNG_stdin32, seed = unknown
test set = core, folding = standard (32 bit)

[...]

rng=RNG_stdin32, seed=unknown
length= 1 gigabyte (2^30 bytes), time= 12.6 seconds
  no anomalies in 192 test result(s)

c:\intro>RNG_output.exe jsf32 1073741824 | RNG_test.exe stdin32 -tlmax 1GB
RNG_test using PractRand version 0.94
RNG = RNG_stdin32, seed = unknown
test set = core, folding = standard (32 bit)

[...]

rng=RNG_stdin32, seed=unknown
length= 1 gigabyte (2^30 bytes), time= 12.6 seconds
  no anomalies in 192 test result(s)
```

Não é incomum que um gerador com suficiente comportamento aleatório mostre vez ou outra algum $p$-valor que chame a atenção: amostras atípicas podem ser obtidas mesmo de um gerador verdadeiramente aleatório [19, seção 3, página 6].

> When applying several tests to a given generator, $p$-values smaller than 0.01 or larger than 0.99 are often obtained by chance even if the RNG behaves correctly with respect to these tests (such values should normally appear approximately 2% of the time). In this case, suspicious values would not reappear systematically (unless we are extremely unlucky). [...] [W]hen a generator starts failing a test decisively, the $p$-value of the test usually converges to 0 or 1 exponentially fast as a function of the sample size of the test, when the sample size is increased further. Thus, suspicious $p$-values can easily be resolved by increasing the sample size.

Um gerador fracassa quando há alguma regularidade em seus fracassos. Um fracasso isolado não é suficiente razão para dizer que o gerador possui uma falha grave, mas um fracasso regular sim. Além disso, para resolver uma suspeição, aumente o tamanho da amostra.

Se houvesse uma receita geral para classificar um gerador de números aleatórios qualquer, então programas de computador por si só poderiam classificá-los como "ruim" ou "bom". Nenhuma das ferramentas apresentadas neste documento faz isso. Cabe ao profissional fazer o julgamento em função da aplicação intencionada para o gerador [19, section 3, página 5].

> [...] No universal test or battery of tests can guarantee, when passed, that a given generator is fully reliable for all kinds of simulations. But even if statistical tests can never prove that an RNG is foolproof, they can certainly improve our confidence in it. One can rightly argue that no RNG can pass every conceivable statistical test. The difference between the good and bad RNGs, in a nutshell, is that the bad ones fail very simple tests whereas the good ones fail only very complicated tests that are hard to figure out or impractical to run. Ideally, when testing an RNG for simulation, [the test statistic] should mimic the random variable of practical interest so that a bad structural interference between the RNG and the simulation problem will show up in the test. But this is rarely practical. This cannot be done, for example, for testing RNGs that are going to be used in general-purpose software packages.

Terminamos esta apresentação com um pequeno experimento sobre a qualidade de geradores de números aleatórios oferecidos por linguagens de programação populares. A Tabela 1 mostra a quantidade de testes em que cada gerador fracassa em cada bateria da TestU01. Se um gerador fracassa numa bateria menor, dispensamos os testes em baterias maiores.

Tabela 1: Resultado de testes contra geradores incluídos em linguagens de programação populares.

| linguagem | gerador | `SmallCrush` | `Crush` | `BigCrush` |
|---|---|---|---|---|
| Java | `drand48` | 5 | — | — |
| Python, PHP | `mt19937` | 0 | 2 | — |
| C++ | `minstd` | 9 | — | — |
| C++ | `ranlux24` | 9 | — | — |
| C++ | `ranlux48` | 0 | 0 | 0 |
| Racket | `mrg32k3a` | 0 | 0 | 0 |

Ambas linguagens Python e PHP incluem o `mt19937`. Apesar do sucesso na bateria `SmallCrush`, `mt19937` fracassa o *linear complexity test*, que é um teste rápido de ser efetuado e poderia fazer parte da pequena bateria da TestU01. Além disso, o `mt19937` tem seu estado interno totalmente determinado coletando-se 624 números [28, seção 2.2, página 8]. O gerador `mrg32k3a` passa pela `BigCrush` [19, seção 7, Table I, página 27] bem como o `ranlux48` [23, 13]. Contudo, enquanto o `mrg32k3a` termina a bateria em um dia, o `ranlux48` não termina antes de três dias no mesmo computador. Esse pequeno experimento sugere que a qualidade de geradores oferecidos por linguagens de programação populares não é alta.

# Primeira parte

## 4 Geradores de números aleatórios

É útil que conheçamos o vocabulário e as definições minimamente necessárias envolvidas com o tema de aleatoriedade, o que faremos de forma interativa ao longo de um passeio. Nosso primeiro passo é pedir um número aleatório ao computador.

```
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [...] on win32
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>> import random
>>> random.random()
0.13436424411240122
```

O número aleatório é 0,13436424411240122. Repetindo o mesmo pedido, receberemos outros.

```
>>> random.random()
0.8474337369372327
>>> random.random()
0.763774618976614
>>> random.random()
0.2550690257394217
```

## 4.1 Sobre números racionais e inteiros

É curioso que todos os números aleatórios que estamos vendo estejam entre 0 e 1, o que nos leva a discutir essa observação imediatamente porque é o que normalmente se encontra na prática e desejamos compreender o que encontraremos.

Sob certa ótica, podemos afirmar que computadores não trabalham com números racionais, mas com números inteiros. Sob outra ótica, números aparentemente inteiros podem ser apenas uma representação para números racionais. Por exemplo, escrever "1234" pode ser uma convenção para o número 0.1234, ou seja, convenciona-se que à esquerda de um número sempre há os símbolos "0.". Geradores de "números aleatórios" na verdade produzem dados aleatórios, a partir dos quais podemos produzir números inteiros, racionais ou outros elementos quaisquer desde que trabalhemos a representação dos dados até que atinjam a forma desejada. Geradores por padrão entregam números racionais ao usuário (e não inteiros) porque transformá-los em números inteiros é considerado uma implementação trivial. Transformar inteiros aleatórios em racionais aleatórios é considerado menos trivial; então o programador responsável pelo gerador faz o trabalho mais difícil na implementação original do gerador, a partir da qual ele implementa procedimentos triviais para produzir inteiros.

Por curiosidade, vejamos o número inteiro originalmente produzido pelo gerador de Python. O gerador que Python usa no módulo `random`[11] é o Mersenne Twister [27] e a implementação usa uma precisão de 53 bits. Obtemos o número inteiro correspondente multiplicando o resultado por $2^{53}$. Note que `**` é o operador Python para exponenciação.

```
>>> 0.8474337369372327 * 2**53
7633004523783416.0
```

Para saber que o fator de multiplicação é $2^{53}$ tivemos que olhar o código-fonte[12], onde se vê que, antes de produzir o número racional, uma divisão por `9007199254740992 = 2**53` é efetuada.

```
static PyObject *
_random_Random_random_impl(RandomObject *self)
/*[clinic end generated code: output=117ff99ee53d755c input=afb2a59cbbb00349]*/
{
    uint32_t a=genrand_uint32(self)>>5, b=genrand_uint32(self)>>6;
    return PyFloat_FromDouble((a*67108864.0+b)*(1.0/9007199254740992.0));
}
```

```
>>> 2**53
9007199254740992
```

[11] Documentação disponível em `https://bit.ly/3buoMTe`.
[12] Disponível em `https://bit.ly/3i4tuZW`.

O fator $2^{53}$ é específico do Mersenne Twister. Um outro gerador teria uma implementação diferente. Racket, uma linguagem da família Lisp, usa o gerador `mrg3k3a` [20], um gerador completamente diferente do Mersenne Twister. Em seu código-fonte[13], observamos a seguinte implementação, de onde extraímos a informação de que deveríamos dividir o número racional produzido pelo gerador por `2.328306549295728e-10` e adicionar o resultado por `1.0` para obter o número inteiro correspondente ao racional produzido pelo gerador.

```
#define m1 4294967087.0
[...]
/* normalization factor 1/(m1 + 1) */
#define norm 2.328306549295728e-10

double S_random_state_next_double(ptr s)
{
  double  x;
  x = mrg32k3a(s);
  return (x + 1.0) * norm;
}
```

Assim, podemos obter os inteiros relativos ao número racional produzido pelo `mrg3k3a`.

```
Welcome to Racket v7.6.
> (require random)
> (define norm 2.328306549295728e-10)

> (random)
> 0.5067308904603183
> (- (/ 0.5067308904603183 norm) 1.0)
2176392496.0

> (random)
0.8476066606357194
> (- (/ 0.8476066606357194 norm) 1.0)
3640442710.0
```

Um usuário típico de geradores de números aleatórios usualmente não se importa em obter o número inteiro do qual o número racional produzido pelo gerador foi obtido. Se tudo que ele deseja é obter um número inteiro aleatório entre 0 e 100, por exemplo, basta multiplicar o número racional por 100 e desprezar a parte fracionada do número.

```
>>> int(random.random() * 100)
47
>>> int(random.random() * 100)
3
```

## 4.2 Reproduzindo uma sequência aleatória

Ao executar esses passos no seu sistema, você deve ter notado que seus números aleatórios não são iguais aos nossos. Seu sistema está em um estado diferente do nosso e por isso gera números diferentes.

Um **gerador de números pseudoaleatórios** possui um estado interno que determina o próximo número a ser produzido. Se pudermos preparar o estado interno de um certo gerador, podemos fazê-lo reproduzir mais de uma vez a mesma sequência de números aleatórios.

Para obter os mesmos números que obtivemos acima, prepare o seu gerador com o valor inicial 1. Você obterá os exatos mesmos números.

[13] Disponível em `https://bit.ly/2Dfy9cR`.

```
>>> random.seed(1)
>>> random.random()
0.13436424411240122
>>> random.random()
0.8474337369372327
```

Como a semente determina toda a sequência de números, podemos então reproduzir uma mesma sequência desde que conheçamos sua semente, ou seja, uma sequência é nada aleatória para quem conhece sua semente.

Nem todo gerador de números aleatórios é assim, o que nos leva a distingui-los.

## 4.3 Pseudoaleatório *versus* verdadeiramente aleatório

O Mersenne Twister é um gerador **pseudoaleatório**, um procedimento aritmético dependente de um valor inicial. Esses geradores produzem sequências de números que podemos reproduzir desde que conheçamos a semente, isto é, o valor inicial dado ao gerador.

Há geradores em que não se sabe como reproduzir qualquer sequência que ele produza. Esses são chamados de **verdadeiramente aleatórios**. Eles obtêm os números de uma fonte de aleatoriedade que são fenômenos físicos como o decaimento de partículas atômicas em átomos ou oscilações de uma corrente elétrica, fenômenos em que a natureza é quem provê a aleatoriedade.

Ambos tipos de geradores são úteis, mas cada grupo possui suas características. Os geradores pseudoaletórios, por exemplo, tendem a ser mais rápidos que os verdadeiramente aleatórios, o que é desejável em aplicações que precisam de grandes volumes de números. Os verdadeiramente aleatórios são inviáveis de serem reproduzidos, o que é interessante para aplicações que demandam maior garantia quanto à irreprodutibilidade dos números.

Analisar os termos "pseudoaleatório" e "verdadeiramente aleatório" com profundidade nos levaria a profundidades teóricas de difícil tratamento [17, section 3.5] [4].

## 4.4 O que há dentro de um gerador?

No coração de um gerador verdadeiramente aleatório encontra-se provavelmente um detector relativo a algum fenômeno físico. O fenômeno deve intrinsicamente possuir aleatoriedade e, à medida que se desenvolve no tempo, o detector captura a informação tipicamente por um dispositivo eletrônico, levando a informação a algum sistema de computação. Normalmente um gerador verdadeiramente aleatório passa por várias etapas de tratamento da informação. O que testamos com as ferramentas e conteúdo discutidos neste documento é o produto final, ou seja, a sequência de dados produzida pelo gerador.

No coração de um gerador pseudoaleatório encontra-se apenas operações aritméticas como soma, subtração, divisão, multiplicação, dentre outras. Por exemplo, eis o que encontramos no gerador *Xorshift* [25] de George Marsaglia implementado com 32 bits e expressos usando a linguagem C.

```
unsigned int xorshift(void) {
  static unsigned int y = 2463534242U; /* a semente de Marsaglia */
  y = y ^ (y << 13);
  y = y ^ (y >> 17);
  y = y ^ (y << 5);
  return y;
}
```

Uma variável estática pede ao compilador C para inicializá-la uma única vez com o valor especificado; invocações subsequentes não reescrevem a variável `y`, mas usam seu último valor. Os símbolos `^`, `<<` e `>>` são os operadores da linguagem C que, respectivamente, representam adição módulo 2 bit a bit, multiplicação por 2 e divisão por 2. O operador `<<` é comumente chamado de *left shift* porque ele move os bits de um número para a esquerda. Por exemplo, `5 << 2` produz o número `20` porque 5 em

binário tem a representação `101` e se movermos esses bits duas casas para a esquerda preencheremos com zeros as casas vazias que sobrariam, obtendo `10100`, que é a expressão do número 20 em base dois. (A convenção é que se preenche com zeros.) Logo, `5 << 2` é o mesmo que `5 * 2 * 2` porque adicionar zeros à direita em base dois é o mesmo que multiplicar por dois duas vezes.

Eis um programa completo que ilustra o gerador.

```
#include <stdio.h>
unsigned int xorshift(void)
{
  static unsigned int y = 2463534242U; /* a semente de Marsaglia */
  y = y ^ (y << 13);
  y = y ^ (y >> 17);
  y = y ^ (y << 5);
  return y;
}

int main(void) {
  unsigned int r; int i;
  for (i = 0; i < 3; ++i) {
    r = xorshift();
    printf("%u\n", r);
  }
}
```

Obtemos os três primeiros números depois da semente `2463534242` fazendo os seguintes pedidos.

```
%make xorshift-text
gcc     xorshift-text.c   -o xorshift-text
%./xorshift-text.exe
723471715
2497366906
2064144800
```

Podemos implementar o mesmo gerador em Python, por exemplo, com as seguintes palavras.

```
from operator import xor
def xorshift():
  global y
  y = xor(y,  (y * 2**13) % 2**32 )
  y = xor(y, int(y / 2**17) % 2**32 )
  y = xor(y, (y * 2**5) % 2**32 )
  return y % 2**32
```

Reduzimos cada operação módulo $2^{32}$ para simularmos o comportamento de C89[14], afinal Python usa inteiros de tamanho arbitrário.

```
>>> y = 2463534242
>>> [xorshift() for i in range(3)]
[723471715, 2497366906, 2064144800]
```

[14] A linguagem C é definida por uma gramática e uma semântica, ou seja, a linguagem C é um livro contendo suas regras. Em 1989, uma definição da linguagem foi dada [12] e se tornou popularmente conhecida como "C89". A linguagem não considera que a multiplicação entre operandos do tipo `unsigned` possa produzir *undefined behavior*. Portanto, nossa assunção sobre o comportamento do programa é garantida por C89. *"A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting unsigned integer type."* [12, seção 3.1.2.5]

Essa breve introdução sobre o tema mostra que a atividade de testar geradores não livra o profissional de desenvolver uma certa intimidade com sistemas e arquiteturas de computação. O exemplo em C e em Python que mostramos anteriormente tem o propósito de exibir a operação de um gerador. Um programa para servir de produtor de informação para a análise dos testes estatísticos precisa colocar os números aleatórios em uma forma que o analisador entenda, que é o que veremos a seguir.

## 4.5 Produzindo um programa para ser testado

A forma mais fácil de testar um gerador é fazê-lo imprimir na saída padrão os bytes relativos a seus números aleatórios usando uma representação binária dos dados. A ilustração que vimos anteriormente exibe uma representação textual dos números.

```
%./xorshift-text.exe
723471715
2497366906
2064144800
```

Em particular, após cada número exibido acima na base decimal aparece uma quebra de linha, o que pode ser visto em alta precisão se interpretarmos os dados como uma sequência de bytes.

```
%./xorshift-text.exe | od -t x1
0000000 37 32 33 34 37 31 37 31 35 0a 32 34 39 37 33 36
0000020 36 39 30 36 0a 32 30 36 34 31 34 34 38 30 30 0a
0000040
```

O byte relativo a uma quebra de linha usa o valor 10 em base decimal, o que é expresso por `0a` em base hexadecimal, como exibido acima pelo programa `od`. Pedindo ao `od` que exiba os valores em base decimal, observamos que "o número" `10` aparece como o décimo byte.

```
%./xorshift-text.exe | od -t d1
0000000   55   50   51   52   55   49   55   49   53   10 [...]
```

De fato, o primeiro número impresso por `xorshift-text.exe` é o número `723471715`, que tem nove dígitos, sendo o décimo byte a quebra de linha. Concluímos então que os dados produzidos pelo programa `xorshift-text.exe` não são exatamente o número `723471715`, mas sim os números 55, 50, 51, 52, 55, 49, 55, ... Números que dificilmente se passariam como aleatórios.

O símbolo 0 tem valor decimal 48. Você pode memorizar esse valor como referência. O símbolo 1 tem valor 48 + 1 e assim sucessivamente. Por isso, o símbolo 7 tem valor $48 + 7 = 55$, como se observa.

Para testar o gerador de Marsaglia, precisamos que o programa produza os bytes que a arquitetura do computador tem em memória. A mudança que precisamos em nosso programa de ilustração é trocar a impressão textual para uma impressão binária, que é o trabalho de `fwrite` da biblioteca C padrão.

```
#include <stdio.h>
unsigned int xorshift(void)
{
  static unsigned int y = 2463534242U; /* a semente de Marsaglia */
  y = y ^ (y << 13);
  y = y ^ (y >> 17);
  y = y ^ (y << 5);
  return y;
}

int main(void) {
  unsigned int r; int i;
  for (i = 0; i < 3; ++i) {
```

```
    r = xorshift();
    fwrite(&r, sizeof r, 1, stdout);
  }
}
```

Compilando e executando o programa acima, obtemos 12 bytes como resposta.

```
%make xorshift-sample-3
gcc     xorshift-sample-3.c   -o xorshift-sample-3
%./xorshift-sample-3.exe | od -t x1
0000000 63 4d 1f 2b 7a cb da 94 a0 59 08 7b
0000014
```

São 12 bytes porque $12 = 3 \times 4$. O programa produz 3 números armazenados cada um em um `unsigned int`. Cada `unsigned int` ocupa 4 bytes na arquitetura que estamos usando.

Os primeiros 4 bytes são uma representação binária do número 723471715, que é o primeiro produzido pelo *Xorshift* que implementamos. Para verificar o fato, precisamos compreender como a arquitetura que estamos usando representa o número 723471715 na memória principal do computador, o que nos leva a investigar o que chamamos de ordem binária de um computador.

## 4.6 A ordem binária usada por um computador

Se um número ocupa quatro bytes e desejamos escrever esses bytes em algum lugar, precisamos escolher que byte primeiro será escrito. Representemos esses quatro bytes pela lista `b3 b2 b1 b0`. Se optarmos por escrever os bytes na ordem `b0 b1 b2 b3`, nossa decisão será chamada de ***little-endian*** porque optamos por começar pelo byte de menor significância. Se optarmos pela ordem `b3 b2 b1 b0`, nossa decisão será chamada de ***big-endian*** por começar pelo byte de maior significância[15].

O número 723471715 tem representação binária

`00101011 00011111 01001101 01100011`,

sendo que separamos os 32 bits em grupos de oito bits, isto é, um byte. Como a arquitetura que estamos usando é *little-endian*, começamos pela ponta direita porque começamos pelo byte menos significativo do número, que é o byte `01100011`. Representando cada byte em decimal, obtemos os bytes 43, 31, 77 e 99. Esses são os bytes que estão na memória do computador, ou seja, o número 723471715 é armazenado como `[99, 77, 31, 43]`. Convertendo esses bytes para hexadecimal, obtemos `63`, `4d`, `1f`, `2b`, exatamente na ordem que o programa `od` lê o que o programa `xorshift-sample-3.exe` produz.

```
%./xorshift-sample-3.exe | od -t x1
0000000 63 4d 1f 2b 7a cb da 94 a0 59 08 7b
0000014
```

São esses bytes que precisam ser analisados pelos testes estatísticos e não uma representação textual deles. A representação textual implica em uma conversão de dados aleatórios originais e consequente perda de informação uma vez que os números são todos convertidos para um intervalo específico. Em decimal, cada byte da representação textual está sempre entre 48 e 57, uma vez que só imprimimos algarismos de zero a nove. Em binário, o valor 48 é `00110000` e o valor 57 é `00111001`. Observe que os dois bits mais significativos são sempre zero, o que dificilmente seria visto como aleatório.

[15] O termo *endianess* se refere a como um computador ordena os bytes referentes a um número. A palavra *end* traduz-se não só para "fim", mas também para a "extrema ponta" de alguma coisa. O sufixo "ness" indica a substantivação da palavra, assim como o sufixo "idade" faz na língua portuguesa. Por exemplo, "maleável" é um adjetivo e "maleabilidade" é o substantivo que lhe é correspondente. Por qual ponta da lista de bytes de um número começa um certo computador? Se começa pela ponta onde está o byte de menor significância, então o computador é *little-endian*. Se pelo maior, *big-endian*. Quando escrevemos o número 123, o algarismo 1 é mais significativo que o 2 porque o 1 representa o número 100 enquanto o 2, o número 20. Isso mostra que nossa cultura é *big-endian*. Apesar de alguns árabes escreverem da direita para a esquerda, a cultura deles também é *big-endian* no papel, pelo menos com respeito à escrita de números. Computadores Intel tipicamente são *little-endian*. Os computadores Sparc da Sun Microsystems bem como a máquina virtual Java são *big-endian*. A ordem *little-endian* é mais popular que *big-endian*: provavelmente seu computador também é *little-endian*.

## 4.7 O programa `crush` para análise de geradores

Um gerador de números aleatórios tipicamente produz uma sequência de números infinita, por mais que seja cíclica. O *Xorshift* que ilustramos acima imprime apenas três números. Removamos a limitação e ele está pronto[16] para ser analisado.

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

unsigned int xorshift(void)
{
  static unsigned int y = 2463534242U; /* the seed */
  y = y ^ (y << 13);
  y = y ^ (y >> 17);
  y = y ^ (y << 5);
  return y;
}

int main(void) {
  unsigned int r;
  for (;;) {
    size_t n;
    r = xorshift();
    n = fwrite(&r, sizeof r, 1, stdout);
    if (n < 1) _exit(0);
  }
}
```

Compile e monte o programa.

```
%make xorshift
gcc     xorshift.c   -o xorshift
%file xorshift.exe
xorshift.exe: PE32 executable [...] Intel 80386 [...], for MS Windows
```

O estado da arte em testes estatísticos para geradores é a biblioteca TestU01. Para usá-la é usual a escrita de um programa C, o que muitas vezes é uma inconveniência. Como poderíamos testar um gerador implementado em Java, por exemplo? Seria preciso reescrevê-lo em C para tirar vantagem da biblioteca TestU01 ou nos contentar com outros mecanismos de comunicação entre a biblioteca e o gerador, incluindo a perda de performance decorrente, que é o que acontece quando usamos arquivos no disco como meio de fazer a comunicação. Por exemplo, para testar o *Xorshift*, basta fazer o seguinte pedido a seu *shell*.

[16] Adicionamos também a verificação do número de itens escritos por `fwrite`, terminando quando a quantidade for abaixo do esperado. A estratégia do `xorshift.c` é imprimir os bytes produzidos pelo *Xorshift* na saída padrão, isto é, na `stdout`. Nosso objetivo é rodar o programa tanto em sistemas UNIX como em sistemas Windows. Através do *shell*, o programa terá sua `stdout` conectada a `stdin` do `crush`. Com a modificação que fizemos, o `xorshift.c` produzirá uma sequência infinita de bytes e o `crush` consumirá uma quantidade finita deles. Quando o `crush` terminar, ele fechará a `stdin` e portanto fechará a `stdout` do `xorshift.c`. Quando isso acontece num sistema UNIX, o *kernel* entrega ao processo `xorshift` um aviso chamado `SIGPIPE`, notificando-o da tentativa de escrita em um *broken pipe*, ou seja, escrita em um canal de comunicação sem um leitor. Entregar um `SIGPIPE` significa interromper o programa e passar o controle do processador para uma rotina do programa responsável por reagir ao recebimento da ocorrência relativa ao aviso. Se o programa não reage à entrega do aviso, então ele é terminado pelo *kernel*. Sistemas Windows — sem configurações especiais — não implementam a entrega do aviso `SIGPIPE` e, logo, sem a verificação adicionada, o programa ficaria preso no laço infinito sem conseguir ter quem ler os bytes que ele escreveria eternamente. A solução é verificar pela escrita sem sucesso, o que serve como solução para ambos os sistemas.

```
%./xorshift.exe | crush --battery small --name xorshift | grep .
[...]
========= Summary results of SmallCrush =========
 Version:          TestU01 1.2.3
 Generator:        xorshift
 Number of statistics:  15
 Total CPU time:   00:00:10.93
 The following tests gave p-values outside [0.001, 0.9990]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-015):
       Test                          p-value
 ----------------------------------------------
  1  BirthdaySpacings                  eps
  2  Collision                         eps
  6  MaxOft                            eps
  6  MaxOft AD                      1 -  3.6e-06
  7  WeightDistrib                  9.1e-15
  8  MatrixRank                        eps
 10  RandomWalk1 H                     eps
 10  RandomWalk1 M                  3.3e-16
 10  RandomWalk1 J                  1.4e-15
 ----------------------------------------------
 All other tests were passed
```

O *Xorshift* de 32 bits de George Marsaglia não tem a menor condição de passar pela bateria pequena da TestU01. Cada linha do relatório acima especificando um teste estatístico apresenta um fracasso do gerador. Estudaremos o significado desses resultados em breve. Antes, vejamos a análise de um gerador que passa pela bateria pequena com facilidade, que é o caso do gerador `/dev/urandom` de um sistema GNU que utilize o *kernel* de Linus Torvalds.

```
%crush --battery small --name local < /dev/urandom | grep .
========= Summary results of SmallCrush =========
 Version:          TestU01 1.2.3
 Generator:        local
 Number of statistics:  15
 Total CPU time:   00:00:09.68
 All tests were passed
```

### 4.8 Instalando o programa `crush`

Na página[17] do `crush` você encontra uma versão Windows do programa. O resto desta seção se aplica a sistemas UNIX, onde você precisará compilar o programa bem como a biblioteca TestU01, que é com o que nos preocupamos agora.

As instruções para copiar e compilar a TestU01 são:

```
%wget http://simul.iro.umontreal.ca/testu01/TestU01.zip
[...]
%unzip TestU01.zip
[...]
%cd TestU01-1.2.3/
%./configure
[...]
```

[17] O endereço é `https://dbastos0.github.io/crush/`

```
%make
[...]
%sudo make install
[...]
```

Esses comandos instalam as bibliotecas do pacote TestU01 em seu sistema. Os comandos seguintes instalam o programa `crush`. Se você obtiver erros durante a montagem do programa ou durante a execução, consulte o Apêndice A para mais esclarecimentos.

```
%wget https://dbastos0.github.io/crush/files/crush-0.3.tar.gz
[...]
%tar zxf crush-0.3.tar.gz
%cd crush-0.3
%make
gcc -O2 -c crush.c
gcc -s -o crush crush.o -ltestu01 -lprobdist -lmylib -lm
%sudo make install
```

Feito isso, já podemos analisar o gerador local de um sistema UNIX.

```
%crush -b small -n 'nosso gerador local' < /dev/urandom | grep .
[...]
========= Summary results of SmallCrush =========
 Version:          TestU01 1.2.3
 Generator:        nosso gerador local
 Number of statistics:  15
 Total CPU time:   00:00:10.96
 All tests were passed
```

Note que o programa `crush` é instalado no diretório `/usr/local/bin`.

# Segunda parte

## 5 Fazendo a leitura de um teste estatístico

Teoria estatística nos fornece uma estratégia de verificação de hipóteses que é chamada de "teste de hipótese" [8, capítulo 26]. O procedimento nos remete a uma prova por contradição, onde supomos uma hipótese e mostramos que ela nos leva a resultados sabidamente falsos. No teste de hipótese, assume-se uma hipótese a respeito de um fenômeno e coleta-se amostras de dados relativas ao fenômeno. Se as amostras contradizem a hipótese relativa ao fenômeno, temos então razão para negá-la, de outra forma as amostras corroboram a hipótese.

Um exemplo de fenômeno é a produção de números aleatórios. Uma hipótese que podemos propor relativa a um gerador é: "o gerador produz números aleatórios". Ao coletarmos certas amostras de números, podemos argumentar que o gerador supostamente aleatório não produziria tais amostras com a alta probabilidade com a qual elas foram obtidas, encorajando-nos a desconfiar de que o gerador não produza números aleatórios como a hipótese afirma.

Os relatórios de testes estatísticos contra geradores de números aleatórios usualmente respondem um número chamado $p$-valor [8, seção 26.3]. Esse número consolida a evidência obtida contra o gerador. O estudo que faremos nesta seção nos dará esse vocabulário, ou seja, o significado de um $p$-valor.

## 5.1 O teste de hipótese

Consideremos um dado honesto como gerador de números aleatórios. A cada lançamento, um dado de seis lados oferece um dos números 1, 2, 3, 4, 5, 6. Sendo honesto, a distribuição dos valores é uniforme, o que implica que em média cada lançamento resulta no valor $3{,}5 = 21/6 = (1/6)\times(1+2+3+4+5+6)$. Se lançamos o dado 100 vezes e computamos a soma dos valores, o valor esperado da soma é $100\times 3{,}5 = 350$. Isso não significa que obteremos o exato valor 350. É provável, por exemplo, que obtenhamos um valor ao redor de 350, como $x \in [335, 365]$. Isso porque o dado é honesto.

Um gerador de números aleatórios é como um dado com muitos lados. A atividade de um ensaísta de laboratório é exatamente escrutinizar o "dado" buscando evidências da existência de algum viés que o dado possa ter. A busca é tipicamente feita por algum teste de hipótese, o que ilustraremos a seguir. Todavia, o propósito de exibirmos um exemplo agora é só para que observemos a forma do raciocínio. Uma visão de todo o percurso deve nos ajudar a compreender por que precisamos estudar certos outros tópicos da teoria.

Alguém nos fornece um dado de seis lados e afirma que ele é honesto. Vamos testá-lo! Lançamos o dado 100 vezes e computamos a soma dos números obtidos. Obtemos a soma 368. Daí raciocinamos da seguinte forma. O *desvio padrão* [8, seção 4.5] de um dado honesto é aproximadamente 1,7. Com essa informação, podemos computar o *erro padrão* da amostra obtida, que é aproximadamente 17, visto que a amostra tem *tamanho* 100. Logo, a soma 368 está aproximadamente *1 erro padrão acima do valor esperado*, visto que $1 \approx (368 - 350)/17$. A probabilidade de obtermos uma soma que esteja 1 erro padrão ou mais acima da média é de aproximadamente 30%, que é o $p$-valor obtido para a amostra. É uma probabilidade suficientemente alta. Logo, o experimento não indica que o dado seja desonesto; ao contrário, o experimento sugere que o dado é de fato honesto, e assim corrobora a hipótese sob escrutínio.

Vejamos agora um caso em que a evidência sugere a negação da hipótese de que o dado seja honesto. Tomemos um outro dado de seis lados. Suponha que o dado seja honesto. Dessa vez lançamos o dado 1000 vezes. A soma obtida foi de 3680, sendo o valor esperado 3500. Como com o outro dado, o desvio padrão deste é aproximadamente 1,7, o que nos dá um erro padrão aproximado de 54. A amostra, portanto, exibe uma soma que está mais de 3 erros padrão acima da média, visto que

$$3 < \frac{3680 - 3500}{1{,}7\sqrt{1000}}.$$

A probabilidade de ocorrência de tal evento é menor que 1%, o que não sugere que esse dado seja honesto. A amostra obtida é suficientemente atípica; ela exibiu uma soma significativamente acima do valor esperado.

Perceba as informações que usamos nesses dois exemplos. Note que a hipótese em análise afirma que o dado é honesto. Sendo honesto e seguindo uma distribuição de probabilidade conhecida, somos capazes de obter o valor esperado da distribuição, o que nos permite prever o erro obtido em cada amostra, ou seja, esperamos que o erro apresentado por cada amostra esteja próximo do erro padrão. O erro padrão, por sua vez, segue a distribuição normal, que também conhecemos. Sendo assim, somos capazes de computar a probabilidade de ocorrência da amostra coletada ou de uma ainda mais atípica que ela, que foi o $p$-valor computado em cada exemplo acima. O que faremos nas próximas seções é fazer um passeio para conhecer a existência dessas ferramentas estatísticas usadas no cálculo de um $p$-valor, o que nos dará um entendimento mínimo sobre o tema.

## 5.2 O que é uma distribuição de probabilidade?

Uma distribuição de probabilidade é uma função que responde a probabilidade de eventos "elementares". Por exemplo, a probabilidade de se obter o número 5 ao lançar um dado honesto é 1/6. De fato, a probabilidade de se obter 1, 2, 3, 4, 5 ou 6 é também 1/6. A distribuição de probabilidade desse dado honesto é uma função $d(x)$ que responde 1/6 para todo $x$ em seu domínio. O domínio é o conjunto {1, 2, 3, 4, 5, 6}. O mapeamento da função em si pode ser descrito pelo fato de que $d(1) = d(2) = \cdots = d(6) = 1/6$. Se desenhássemos um gráfico dessa função $d(x)$, veríamos os pontos (1, 1/6),(2, 1/6), ..., (6, 1/6), ou seja, pontos isolados na altura 1/6 do eixo-$y$. Assim, "por distribuição de probabilidade", imagine

uma função tal que, ao ser solicitada relativa a certo evento elementar, ela responde um número entre 0 e 1 que representa a probabilidade do evento. No exemplo de lançar um dado honesto que acabamos de ver, os eventos "elementares" são "obter o lado 1" ou "obter o lado 2", ou de uma forma geral "obter o lado $x$", sendo $x \in \{1, 2, 3, 4, 5, 6\}$. Uma distribuição de probabilidade responde facilmente a probabilidade de um evento elementar: basta aplicar a função. Por exemplo, qual a probabilidade de obtermos o lado 1? Basta aplicarmos $d$ a 1, isto é, $d(1) = 1/6$. Logo, a probabilidade de obtermos o lado 1 é $1/6$. Essa facilidade é neste contexto e neste tipo de fenômeno.

O tipo de fenômeno envolvido no exemplo do dado é tal que sua distribuição de probabilidade é dita discreta. "Discreta" é uma palavra que vem do Latim "*discretus*", que significa "separado". O lançamento de um dado tem resultados distintos um do outro: o resultado é 1 ou 2 ou 3, ... O dado não oferece a possibilidade de um resultado tal que não consigamos distinguir se seria o lado 2 ou outro lado. "Mesmo que o dado termine o lançamento encostado contra a parede?" Em matemática nunca de fato falamos de um dado que poderia se encostar contra a parede. Falamos em "lançamento de dado", mas nos referimos à ideia abstrata de um processo que não precisa ter uma representação no mundo físico e que resulte *sempre* em um dos elementos do conjunto $\{1, 2, 3, 4, 5, 6\}$, ou seja, a palavra "dado" não faz referência a um cubo de seis lados feito de matéria, mas sim à função $d(x)$ que descrevemos anteriormente.

Agora considere medir a altura de uma pessoa. Poderíamos determinar medições discretas. Podemos arbitrar que a altura de uma pessoa será 1, 1,5 ou 2 metros. Se uma pessoa tiver mais de 1,5 e menos de 2 metros, teremos que determinar de alguma forma se lhe atribuímos 1,5 ou 2 metros. Se a altura dela estiver mais próxima de 1,5, diremos 1,5; caso contrário, 2 metros. Mas e se a medição resultar em 1,75 metros, quando então estaremos igualmente distantes de 1,5 e 2 metros? Podemos arbitrar que 1,75 metros exatamente será considerado 1,5 metros; qualquer altura acima de 1,75 metros é atribuída a 2 metros. O que acabamos de fazer é discretizar a medição de altura, o que evidentemente pode ser feito. Entretanto, outra operação seria considerar a altura de uma pessoa como uma medida que possa cair em qualquer número real, mesmo os negativos e os irracionais. Quando os eventos elementares de um fenômeno podem, num certo intervalo, tomar um número infinito de possibilidades, sendo este infinito da mesma natureza daquele que se encontra na completude dos números reais, o fenômeno é chamado de contínuo e, assim, a distribuição de probabilidade relativa ao fenômeno é chamada de contínua.

Quando a distribuição de probabilidade é contínua, é comum usarmos os métodos de Cálculo, que é a disciplina matemática que provê soluções para as dificuldades oriundas dessa característica da linha real, isto é, da natureza contínua da linha real. Quando se quer a probabilidade de que uma pessoa *qualquer* tenha certa altura, o que se pergunta de fato é a probabilidade de uma pessoa qualquer ter uma altura pertencente a um intervalo. Por exemplo, qual a probabilidade de que um homem brasileiro adulto tenha menos de 1,60 metros? Por "menos de 1,60 metros" entende-se o intervalo da linha real $x < 1{,}60$. Podemos dizer com confiança que a probabilidade de um homem adulto brasileiro ter menos de 1,60 metros é 9% desde que se assuma que a média de altura de uma pessoa nessa população seja 1,70 metros e que o desvio padrão na população seja de 7,42 centímetros. (O desvio padrão é uma medida de quão variável é a amostra coletada da população; pessoas de uma certa idade tendem a ter alturas muito próximas, então a variabilidade de alturas é pequena, como sugere a medida de 7,42 centímetros.) A forma como respondemos a essa pergunta é consultar a distribuição de probabilidade relativa a alturas. Como se espera, é provável que uma pessoa numa certa idade tenha altura próxima da média e improvável encontrarmos alturas muito distantes da média. Por exemplo, mantendo-nos na distribuição de probabilidade da população de homens adultos com média de 1,70 metros e desvio padrão de 7,42 centímetros, qual a probabilidade de que um homem adulto tenha até 1,50 metros? Essa probabilidade é menor que 0,004, isto é, menor que 0,4% (ou, digamos, 40% de 1%). É bem pequena. Já se perguntamos a probabilidade de um homem adulto ter até 1,80 metros, obtemos um número maior que 91%.

A Figura 1 mostra a distribuição de probabilidade relativa a homens adultos no Brasil. O eixo-$y$ descreve a densidade de probabilidade e o eixo-$x$ descreve a altura em centímetros. Note como o gráfico sobe rapidamente quando a altura se aproxima de 1,70 metros, illustrando o fato de que a maior parte dos homens tem altura próxima da média. Obter a probabilidade de um homem adulto ter entre 1,50 e 1,80 metros significa calcular a área embaixo do gráfico entre os pontos 150 e 180 do eixo-$x$, o que é

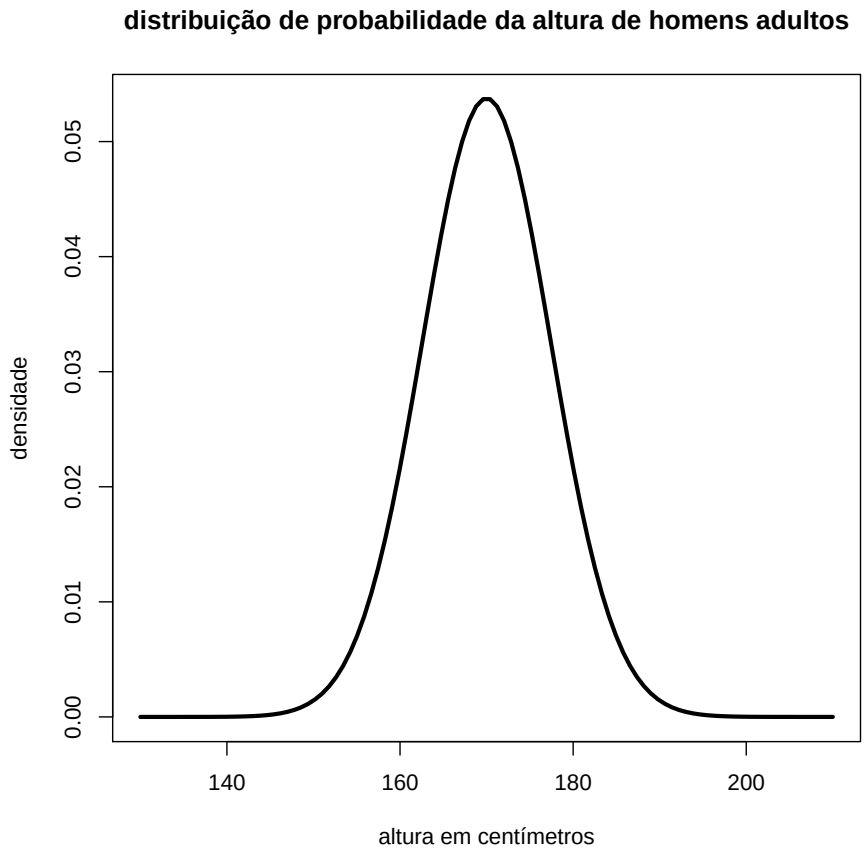


Figura 1: A distribuição de probabilidade da altura de um homem adulto brasileiro.

um problema que se estuda na disciplina chamada "Cálculo".

Essa forma do gráfico da distribuição de probabilidade relativa a alturas ocorre não só com alturas, mas com tantos outros fenômenos na natureza e no mundo. Assim, esses gráficos, essas funções que descrevem esse tipo de distribuição de probabilidade vieram a ser chamadas de "distribuição normal". O matemático Carl Friedrich Gauss foi uma das pessoas que modelou essa distribuição e, por isso, ela é às vezes chamada de distribuição gaussiana.

Nem tudo se distribui de forma "normal". O lançamento de um dado, como vimos, se distribui de forma uniforme, ou seja, todos os eventos elementares têm a mesma probabilidade de ocorrência e, por isso, em vez de "normal", a distribuição de probabilidade de um dado honesto é chamada de "uniforme".

Nas seções à frente, haverá momentos em que diremos que certo fenômeno tem distribuição normal e usaremos nossos computadores para calcular essas probabilidades. Mesmo que você não tenha estudado a distribuição normal e outras ainda, mantenha em mente que essas probabilidades são perfeitamente calculáveis e que você aprenderia a fazer os cálculos por si só cursando uma disciplina de probabilidade.

## 5.3 O desvio padrão

A palavra "desvio" significa o quão distante um valor está da média dos valores no conjunto a que o valor pertence. O desvio padrão [8, seção 4.5] é a raiz quadrada da variância de um conjunto. A variância [6, seção 3.2] é uma medida de quão variável um conjunto de dados é ao redor da média do conjunto. Se o conjunto de dados é $\{1, 11/10, 11/10, 8/10, 1, 1\}$, então a média do conjunto é 1 e a variância é

$$\sigma^2 = \frac{(1-1)^2 + (1{,}1-1)^2 + (1{,}1-1)^2 + (0{,}8-1)^2 + (1-1)^2 + (1-1)^2}{6} \approx 0{,}06.$$

O desvio padrão é $\sigma = \sqrt{0.06} = 0{,}24$.

A variância reflete a média dos *quadrados dos desvios* de cada elemento do conjunto de dados. Uma pergunta frequente em cursos de estatística é por que se escolhe medir a variabilidade dos dados através dos *quadrados* dos desvios. O caso realmente merece um escrutínio cuidadoso.

Uma outra forma de medir a variabilidade de um conjunto de dados é usar o chamado *desvio absoluto* [5, "mean absolute deviation", páginas 79–80]. O conjunto {1, 11/10, 11/10, 8/10, 1, 1} tem, por definição, desvio absoluto

$$\frac{|1-1|+|1{,}1-1|+|1{,}1-1|+|0{,}8-1|+|1-1|+|1-1|}{6} \approx 0{,}07.$$

Eis, pois, duas formas diferentes de medir a variabilidade dos dados. Para compreendermos melhor essas medidas, precisamos de um exemplo concreto. Suponha que uma empresa terceirize o corte de uma peça de madeira usada em sua produção. A especificação é de que a peça tenha sempre 1 metro de comprimento, mas muitas peças não saem exatamente como manda a especificação. Um certo lote continha seis peças com os tamanhos exibidos pelo conjunto {1, 11/10, 11/10, 8/10, 1, 1} e, usando o desvio absoluto, medimos a variabilidade desse lote em 0,07. Uma interpretação justa da medida é que, em média, a empresa erra em 0,07 metros ao produzir uma peça. Por exemplo, quando a empresa produz uma peça de exatamente 1 metro, a produção é perfeita; quando produz uma peça de 1,1 metros, ela erra em 0,1 metros. Quando produz uma peça de 0,8 metros, ela erra em 0,2 metros. Considerando as seis peças, o erro médio cometido é de 0,07 metros. O desvio absoluto é perfeitamente razoável. Se computarmos o desvio padrão do conjunto {1, 11/10, 11/10, 8/10, 1, 1}, obteremos um valor aproximadamente 0,24. É "folgadamente" maior que 0,07. O que ocorre no desvio padrão é que valores no conjunto de dados que estejam a menos de uma unidade da média participam em menor peso no cálculo da média dos desvios do que valores que estejam a mais de uma unidade da média, afinal elevamos esses desvios ao quadrado[18]. Se olharmos para a medida como uma espécie de nota que se dá ao lote produzido pela empresa, o desvio padrão representa o desejo de dar menor peso ao erro quando o erro é próximo do tamanho desejado do comprimento da peça e maior peso ao erro quando o erro é menos próximo, aumentando-se a punição à medida que o erro cresce. No caso em tela, a empresa terceirizada poderia ingenuamente preferir o desvio absoluto, já que nele a pontuação de seus erros é menor. Se medirmos um lote pelo desvio absoluto e outro pelo desvio padrão, é certo que uma comparação entre os dois não seria justa. Mas o fato é que se medirmos todos os lotes pela mesma medida, seja pelo desvio padrão, seja pelo desvio absoluto, ambas as medidas são perfeitamente justas, afinal a regra é igual para todos os lotes — e seria igual para todas as empresas terceirizadas que viessem a competir por contratos.

A introdução do uso dos quadrados dos desvios foi feita por Gauss, que afirmou ter encontrado medida mais vantajosa que a usada anteriormente por Pierre-Simon Laplace [9, páginas 5–6].

> A determinação do tamanho do erro de uma observação pode ser comparada, não inapropriadamente, a um jogo de azar em que se pode apenas fracassar, nunca ganhar, e em que cada erro corresponde a uma perda. O risco envolvido nesse jogo é medido pela perda provável, ou seja, pela soma dos produtos das possíveis perdas individuais associadas a suas probabilidades. Não está claro, de forma alguma, que perda deve ser equacionada a que erro. De fato, a determinação dessa perda depende, pelo menos parcialmente, de nosso próprio julgamento. Igualar a perda com o próprio erro é obviamente proibido; se erros positivos forem tratados como perda, então erros negativos serão considerados como lucro. O tamanho da perda tem que ser expresso por uma função do erro e que seja sempre positiva. Dentre as infinitas escolhas de funções, a função quadrática parece ser a mais simples e isso parece ser vantajoso. [...] Laplace tratou o problema de forma similar, mas ele escolheu o valor absoluto do erro como a medida de perda. Entretanto, a não ser que estejamos incorretos, essa escolha é certamente não menos arbitrária que a nossa [...]"

"Jogo de azar" porque o erro é produzido por uma fonte aleatória. Nunca se ganha porque o melhor resultado que se pode obter é erro zero, resultado onde apenas se evita uma punição, diferente de quando há um erro para cima ou para baixo, o que ele chama de perda. O "risco do jogo" é a média ponderada dos erros, o que se obtém somando todos os erros prováveis ponderados por suas probabilidades. Não

[18] Um número $x \in (0,1)$ satisfaz $x^2 < x$, enquanto $|x| > 1$ satisfaz $|x| < x^2$.

está claro, ele afirma, se a um erro de tamanho 1,5 devemos dar uma punição de 1,5 pontos ou se devemos, por exemplo, aplicar peso dois ao erro, punindo-o a perda de 3 pontos. A aplicação dessa punição depende do nosso próprio julgamento. Igualar a perda com o próprio erro é absurdo porque um erro de uma unidade para cima cancelaria um erro de uma unidade para baixo — duas coisas erradas não fazem uma coisa certa. Logo, o tamanho da punição tem que ser sempre positivo. Dados esses requerimentos, há infinitas escolhas. A vantagem a que Gauss se refere é que a função quadrática é de tratamento matemático muito mais fácil que a do valor absoluto. Por exemplo, computar a derivada ou a integral de uma função quadrática — tarefa típica do estudante de Cálculo — é tarefa bem menos laboriosa que a do valor absoluto, mesmo em uma única variável.

Outra pergunta que se faz é por que da raiz quadrada no desvio padrão. O fato da variância ser definida com o quadrado de outras medidas produz uma medida expressa no quadrado da unidade. Por exemplo, se um conjunto de dados representa medidas em metros, a variância representa metros quadrados. Por isso, o desvio padrão usa a raiz quadrada da variância, reduzindo a unidade de medida de volta à unidade original. O fato de computarmos a raiz quadrada modifica a medida, mas isso não é um problema porque, como Gauss nos ensinou, a medida da variabilidade é arbitrária desde que se satisfaça algumas propriedades; há infinitas escolhas. Usando a raiz quadrada, obtemos uma unidade de medida mais adequada do que sem ela.

## 5.4 A lei das médias, o crescimento do erro observacional e o erro padrão

Se lançamos uma moeda honesta muitas vezes, a proporção de vezes em que um certo lado é obtido aproxima-se de 50%, mas se compararmos a quantidade absoluta de cara contra a de coroa observamos que a diferença entre as duas quantidades tende a aumentar à medida que lançamos a moeda. É isso que diz a lei das médias [8, capítulo 16]: a proporção tende à igualdade, mas o valor absoluto das quantidades tende a crescer.

> **Lei das médias**. *À medida que o número de lançamentos cresce, cresce a diferença entre o número de cara e a metade do número de lançamentos, mas a diferença entre o percentual de cara e 50% dos lançamentos diminui.*

Por exemplo, suponha que lancemos uma moeda 10 mil vezes. Esperamos obter aproximadamente 5000 caras, mas dificilmente obteremos 5000 caras exatamente. Usualmente há um desvio do valor esperado. A esse desvio dá-se o nome de *erro observacional*. Em símbolos,

$$\delta = C - L/2,$$

sendo $\delta$ o erro observacional, $C$ o número de caras obtidas e $L$, o número de lançamentos da moeda. A lei das médias afirma que $\delta$ cresce à medida que $L$ cresce, mas a razão $\delta/L$ diminui.

A teoria estatística nos dá ainda mais informação sobre o fenômeno. Quanto tende o erro observacional a crescer? Por exemplo, com cem lançamentos, o erro observacional fica provavelmente ao redor de cinco. Com dez mil lançamentos, o erro observacional fica ao redor de cinquenta.

Experimente isso no computador. Faça sorteios de dez, cem, mil, dez mil, cem mil e um milhão de lançamentos. Em cada experimento, conte a quantidade de caras obtidas nos lançamentos e veja o quão distante da metade a quantidade de caras se encontra. O seguinte programa simula $L$ lançamentos de uma moeda honesta, produzindo o erro observacional $\delta$ bem como a razão $\delta/L$.

```
def lei_das_medias(L):
  from random import randint
  from functools import reduce
  ls = [randint(0,1) for i in range(L)]
  C = reduce(lambda x, y: x + y, ls) # número de [C]aras
  R = abs(C - L) # número de co[R]oas
  d = C - (L/2)
  return d, d/L
```

Ao fazer o experimento, obtivemos os desvios (da média) 0, 4, −3, 25, 56, −729, ou seja, em um milhão de lançamentos, apareceram 729 coroas a mais que caras; com dez lançamentos, o número de caras foi exatamente o mesmo de coroas. O erro observacional claramente cresce. Mas e a proporção $d/L$? Obtivemos 0, 0,04, −0,003, 0,0025, 0,00056, −0,000729, uma sequência finita que oscila ao redor de zero, excetuando-se o primeiro elemento da sequência que aparentemente por sorte equilibrou-se entre o número de caras e coroas.

Há algum padrão de crescimento no tamanho de $\delta$, ou seja, no tamanho do valor absoluto do erro? A teoria estatística nos garante que à medida que o número de lançamentos cresce, o erro observacional é proporcional à raiz quadrada do número de lançamentos bem como do desvio padrão da amostra [8, "square root law", seção 17.2, página 294]. Sendo assim, se fizermos 1 trilhão de lançamentos, que é $10^{12}$, o desvio esperado é proporcional a $\sqrt{10^{12}} = 10^6$, além de proporcional ao desvio padrão da amostra. Quanto maior é o tamanho da amostra, mais seu desvio padrão se aproxima do desvio padrão da distribuição de probabilidade da variável aleatória em questão. A variável aleatória em questão toma valores cara e coroa, o que segue uma distribuição de probabilidade de Bernoulli [6, seção 6.6.2, página 142]. Como nossa moeda é honesta, $p = 1/2$ e, logo, o desvio padrão é $\sigma = 1/2$. Assim, o tamanho do erro observacional será próximo de $(1/2)\sqrt{L}$, que é, por definição, o erro padrão [8, seção 17.2, páginas 290–291].

> **O erro padrão**. *Numa amostra aleatória com reposição, o erro padrão para a soma dos números da amostra é $\sigma\sqrt{L}$, sendo $\sigma$ o desvio padrão da distribuição de probabilidade a que o fenômeno obedece e $L$, o tamanho da amostra aleatória.*

Observe que é intuitivo que o crescimento dependa do desvio padrão da distribuição de probabilidade. Quanto maior é a variabilidade possível numa amostra, maior pode ser o erro obtido relativo ao valor que se espera encontrar na amostra. Como o desvio padrão é uma medida da variabilidade, é coerente que o erro padrão lhe seja proporcional.

## 5.5 A estatística $z$

Uma estatística é qualquer função de uma amostra aleatória [6, seção 10.6, página 271]. Quando brevemente ilustramos o teste de hipótese, usamos a soma dos números obtidos na amostra coletada de vários lançamentos de um dado de seis lados. A soma é uma estatística porque é uma função da amostra aleatória. Quando comparamos a soma com o valor esperado da soma, temos outra estatística. Quando computamos quantos erros padrão acima ou abaixo do valor esperado a amostra apresentava, computamos uma outra estatística, sendo esta última usualmente chamada de estatística $z$. Ela mede quão distante a soma da amostra está relativa à média da distribuição de probabilidade [8, seção 26.3, páginas 478–479]. O teste de hipótese demanda uma estatística, que é chamada de *estatística de teste*. Para calcular o $p$-valor de uma estatística de teste, precisamos da distribuição de probabilidade da estatística de teste. A estatística $z$ convenientemente segue a distribuição normal, que conhecemos bem. A letra $z$ é tipicamente usada como variável independente da distribuição normal.

Em símbolos, a definição da estatística $z$ é

$$z = \frac{x - \mu}{\sigma\sqrt{L}},$$

onde $x$ é o valor observado na amostra e $\mu$, o valor esperado. O denominador de $z$ é o erro padrão, o que já conhecemos.

Quando coletamos uma amostra, podemos computar alguma estatística a seu respeito como, por exemplo, a soma dos valores obtidos no dado lançado $L$ vezes. Chamaremos de $x$ o valor observado na amostra. A diferença entre $x$ e o valor esperado $\mu$ mede o quão distante a amostra está do valor esperado. Se a diferença é positiva, o valor observado está acima da média; se negativa, abaixo.

Uma técnica aritmética é dividir uma quantidade por uma outra quantidade de referência. Por exemplo, suponha que a altura média de um brasileiro seja 1,60m, o que tomaremos como referência. Se uma certa pessoa tem 1,70m, então essa pessoa está 6.25% acima da média porque 1,70/1,60 = 1,0625. O número 1,0625 compara a altura 1,70m relativamente à altura 1,60m. Se o numerador fosse a mesma

quantidade que o denominador, a divisão resultante seria 1. Uma divisão maior que 1 significa que o numerador está acima da referência e quando menor que 1, abaixo da referência. Acima quanto? Se a divisão resultou em 1,0625 e 0,0625 é o quanto o numerador está acima da referência, então o numerador está 0,10 metros de 1,60m, visto que $0{,}10 = 0{,}0625 \times 1{,}60\text{m}$.

A divisão serve não só para distribuir uma quantidade, mas também para comparar uma quantidade relativamente a uma referência. Quando dividimos o desvio (do valor observado relativo ao valor esperado) pelo erro padrão relativo ao tamanho da amostra, obtemos uma medida que descreve o quão distante do erro padrão o valor observado está relativamente ao valor esperado. Em outras palavras, a estatística $z$ converte unidades. Em vez de falar em tantos "números reais" distantes de uma referência, a estatística $z$ fala em tantos "erros padrão" da referência adotada, que é 1 erro padrão. O tamanho de um erro padrão depende da amostra e do tamanho dela, como veremos a seguir ao revisitarmos os exemplos de cálculo do $p$-valor de um teste de hipótese.

A distribuição normal nos informa que mais de 95% de todas as amostras distribuem-se em até 2 erros padrão ao redor do valor esperado. Mais de 99,73% de todas as amostras distribuem-se em até 3 erros padrão ao redor do valor esperado. Se uma amostra está acima ou abaixo de 3 erros padrão do valor esperado, temos um amostra significativamente rara.

## 5.6 Revisitando os testes

Revisitemos o teste de hipótese que computamos anteriormente. O primeiro exemplo que demos envolvia um dado de seis lados. Um gerador de números aleatórios tipicamente usado em criptografia é um dado com muitos lados. Sendo uniforme, é fácil computar o valor esperado.

O valor esperado de um dado uniforme de seis lados é 3,5. Lançando o dado 100 vezes, a soma esperada dos pontos obtidos no dado é $\mu = 350 = 3{,}5 \times 100$. Depois de 100 lançamentos não esperamos obter exatamente o valor 350. De fato, a amostra que obtivemos foi $x = 368$. A questão é se essa variação do valor esperado é um comportamento típico do dado ou se evidencia um evento raro. Para fazer essa avaliação, definimos uma estatística da qual conhecemos a distribuição de probabilidade. A estatística $z$ serve bem a esse propósito. Ao computá-la relativa à amostra coletada, obtemos

$$z = \frac{368 - 350}{1{,}7 \times \sqrt{100}} \approx 1{,}05,$$

que descreve o erro padrão relativo à amostra[19]. Isso significa que a amostra coletada está aproximadamente 1 erro padrão acima do valor esperado. Quão provável é a obtenção de uma amostra que esteja a aproximadamente 1 erro padrão ou maior acima da média? Para responder, recorremos à distribuição de probabilidade a que estatística $z$ obedece, que é a distribuição normal. A probabilidade de se obter estatística $z \geq 1.05$ é

$$P(z \geq 1.05) = 1 - 2P(z < 1.05) = 1 - 0{,}7063 \approx 30\%,$$

que não é uma probabilidade implicando rara ocorrência. A amostra de soma 368 pontos é típica e não nos encoraja a negar a hipótese de que o dado seja honesto. Isso não significa que o dado seja honesto, mas a amostra de 100 números obtidas não produz evidência para que suspeitemos de qualquer viés do dado.

É importante notar que esse teste verifica a soma dos dados. Não sabemos, por exemplo, se o lado de número 3 apareceu alguma vez qualquer durante os lançamentos. Talvez não tenha aparecido. Talvez o dado nunca apresente o número 3, o que evidenciaria um claro viés particular ao número 3. É por isso que se diz que um único teste de hipótese, uma única estatística pode prover forte evidência contra um gerador de números aleatórios, mas nunca provaria que o gerador é de fato aleatório. Nenhum conjunto finito de estatísticas pode jamais determinar que um gerador seja verdadeiramente aleatório. Uma estatística é uma visão muito específica de um fenômeno.

Para finalizar nossa excursão estatística, revisitemos o último exemplo. Recebemos para teste um dado de seis lados que não conhecíamos até então. Para testá-lo, supomos que ele era honesto e, logo,

[19] A quantidade 1,7 é o desvio padrão da distribuição de probabilidade a que o gerador obedece e não o desvio padrão da amostra coletada.

obedecia à distribuição de probabilidade uniforme. Assim, o valor esperado dado pela distribuição é 3,5. Decidimos coletar uma amostra de tamanho 1000. A soma dos pontos produzidos pela amostra foi de 3680, sendo que o valor esperado é $3500 = 3{,}5 \times 1000$. O desvio padrão da distribuição de probabilidade de um dado de seis lados é aproximadamente 1,7. Computamos a estatística $z$ e obtivemos

$$z = \frac{3680 - 3500}{1{,}7 \times \sqrt{1000}} \approx 3{,}34.$$

Qual a probabilidade de uma amostra neste contexto apresentar soma maior ou igual a 3 erros padrão acima da média? Já sabemos. A probabilidade é folgadamente menor que 1%. Uma amostra de soma 3680 é significativamente atípica em 1000 lançamentos de um dado honesto. Eis, pois, evidência contra a uniformidade do dado.

## 5.7 Interpretando um relatório de testes

Recordemos o teste da bateria `SmallCrush` da biblioteca TestU01 executada pelo programa `crush` contra o gerador *Xorshift*, o que fizemos na Seção 3.

```
%./xorshift.exe | crush --battery small --name xorshift | grep .
[...]
========= Summary results of SmallCrush =========
 Version:          TestU01 1.2.3
 Generator:        xorshift
 Number of statistics:  15
 Total CPU time:   00:00:10.93
 The following tests gave p-values outside [0.001, 0.9990]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-015):
       Test                          p-value
 ----------------------------------------------
  1  BirthdaySpacings                   eps
  2  Collision                          eps
  6  MaxOft                             eps
  6  MaxOft AD                       1 -  3.6e-06
  7  WeightDistrib                   9.1e-15
  8  MatrixRank                         eps
 10  RandomWalk1 H                      eps
 10  RandomWalk1 M                   3.3e-16
 10  RandomWalk1 J                   1.4e-15
 ----------------------------------------------
 All other tests were passed
```

Note que por `eps`, a biblioteca quer dizer [18, página 33] o número $1.0 \times 10^{-300}$ e por `eps1`, $1.0 \times 10^{-15}$. O nome "eps" vem de *epsilon*, a letra grega tipicamente usada por matemáticos para representar um número bem pequeno.

Agora que sabemos o significado de um $p$-valor, podemos entender que o teste estatístico chamado de `BirthdaySpacings` [21, página 133] é um procedimento que extraiu do gerador uma amostra de certo tamanho, computou uma estatística e, conhecendo a distribuição de probabilidade da estatística, obteve a probabilidade de uma amostra ser tão ou mais atípica que a coletada pelo teste. Como a estatística se mostrou extremamente atípica, da ordem de $10^{-300}$, o teste sugere a rejeição da hipótese de que o gerador é aleatório. Situação similar ocorreu com outros testes. Essa versão do gerador *Xorshift* fracassou nos nove testes estatísticos reportados acima pela `SmallCrush` da TestU01. Dado o flagrante fracasso do gerador, é dispensável a execução de baterias maiores.

Os detalhes particulares de cada teste estatístico foram omitidos acima, mas são emitidos pela ferramenta à medida que os testes executam. A especificação de cada informação relativa a cada teste

precisa ser obtida pela documentação da ferramenta. Por ilustração, podemos apontar que a `SmallCrush` quando executa o `BirthdaySpacings`, coleta uma amostra de tamanho $5 \times 10^6$, ou seja, cinco milhões, o que é verificável pelo parâmetro $n = 5000000$.

```
[...]
smarsa_BirthdaySpacings test:
-----------------------------------------------
   N =  1,  n = 5000000,  r =  0,    d = 1073741824,    t = 2,    p = 1
      Number of cells = d^t = 1152921504606846976
      Lambda = Poisson mean =      27.1051
-------------------------------------------------------
Total expected number = N*Lambda      :      27.11
Total observed number                 :   16227
p-value of test                       :   eps      *****
-----------------------------------------------
[...]
```

Tivesse o gerador passado pelo teste, encontraríamos um $p$-valor não extremo, como neste caso abaixo produzido por outro gerador de números aleatórios diferente do *Xorshift*.

```
[...]
smarsa_BirthdaySpacings test:
-----------------------------------------------
   N =  1,  n = 5000000,  r =  0,    d = 1073741824,    t = 2,    p = 1
      Number of cells = d^t = 1152921504606846976
      Lambda = Poisson mean =      27.1051
-------------------------------------------------------
Total expected number = N*Lambda      :      27.11
Total observed number                 :      31
p-value of test                       :    0.25
-----------------------------------------------
[...]
```

# 6 Entropia

Entropia é um tema envolvido com a noção de quantidade de informação e tem conexões com aleatoriedade. A ideia tradicional de quantidade de informação não se coaduna com as noções que apresentaremos nesta seção. Por exemplo, é comum dizer que copiamos uma grande quantidade de informação ao copiar um grande arquivo de computador. A sugestão de Shannon, publicada em 1948, é relativa a uma fonte que produz informação de acordo com uma distribuição de probabilidade; sem uma distribuição de probabilidade, o que não se pode associar a um arquivo de computador contendo informações fixas, a definição de Shannon não se aplica. Em 1963, Andrey Kolmogorov sugeriu a noção de entropia algorítmica, também chamada de complexidade de Kolmogorov, capaz de trabalhar com sequências fixas de dados, o que nos dá uma nova definição de quantidade de informação. Todavia, mesmo pela ótica de Kolmogorov, há grandes arquivos de computador que não contêm grande quantidade de informação. Por exemplo, um arquivo repleto de zeros perfazendo 1 GiB é resumível a menos de 800 bytes — e esse resumo certamente não é o menor —, implicando uma baixa complexidade de Kolmogorov, ou seja, um arquivo grande contendo pouca informação[20].

Quando falamos em quantidade de informação, podemos nos referir a dois números: (1) um relativo a um evento cuja probabilidade de ocorrência é determinada por uma distribuição de probabilidade; (2) outro relativo a toda distribuição de probabilidade. Quando computamos a entropia de uma distribuição inteira, o número representa a quantidade média de informação transmitida por um evento cuja probabilidade de ocorrência é determinada pela distribuição. Veremos exemplos e detalhes em breve.

[20] O apêndice C descreve brevemente a noção de entropia de Kolmogorov em mais detalhes.

A entropia de Shannon é uma medida que é função de uma distribuição de probabilidade [11, seção 6.1, página 103]: é um número computado a partir da distribuição de probabilidade que modela o comportamento aleatório da fonte. Se temos a distribuição, temos a medida exata de entropia. Se não temos a distribuição de probabilidade, não temos a entropia. Shannon estipulou requerimentos para sua medida de entropia e concluiu [29, teorema 2, seção 6, página 10] que funções $H$ capazes de satisfazer esses requerimentos tem a forma[21]

$$H = -K \sum_{i=1}^{n} p_i \log p_i.$$

Shannon explica que a constante $K$ se refere à unidade de medida adotada. Tipicamente essa unidade é "bits de informação". Tomamos $K$ como sendo "1 bit de informação" e escolhemos a base dois para o uso do logaritmo na fórmula. Assim, obtemos a fórmula $H = -\sum_{i=1}^{n} p_i \lg p_i$ bits de informação, que é equivalente a

$$H = \sum_{i=1}^{n} p_i \lg(1/p_i)$$

bits de informação.

Vejamos um exemplo de uso da definição. Considere um dado $d$ não honesto com a distribuição de probabilidade dada pela Tabela 2.

Tabela 2: A distribuição de probabilidade de um certo dado de seis lados.

| Lado | Probabilidade |
|---|---|
| 1 | 0,1 |
| 2 | 0,2 |
| 3 | 0,3 |
| 4 | 0,2 |
| 5 | 0,1 |
| 6 | 0,1 |

Aplicando a fórmula de Shannon, obtemos

$$H(d) = 0{,}1 \lg(1/0{,}1) + 0{,}2 \lg(1/0{,}2) + 0{,}3 \lg(1/0{,}3) + 0{,}1 \lg(1/0{,}1) + 0{,}1 \lg(1/0{,}1) \approx 2{,}4456$$

bits de informação. Se o dado fosse honesto, a probabilidade de obtenção de qualquer um de seus lados seria $1/6$, o que nos daria a entropia de $6 \times (1/6) \lg 6 \approx 2{,}5850$, que é maior que a entropia do dado $d$. Quanto mais uniforme é a fonte, maior é a entropia. Se computarmos, por exemplo, a entropia de um dado em que a probabilidade de obter os lados 1 e 2 são 0,6 e 0,4, respectivamente, enquantos os outros lados têm probabilidade zero, encontraremos um número próximo de 0,97 bits de informação. Se computarmos a entropia de outro dado com probabilidade 1 para um certo lado e probabilidade zero para todos os outros, encontraremos entropia $H = 1 \times \lg 1 = 1 \times 0 = 0$, como esperado. Esses resultados nos sugerem que à medida que a distribuição de probabilidade de um fenômeno se aproxima de uma distribuição uniforme, a entropia aumenta a seu máximo.

Podemos observar esse fato com maior clareza se considerarmos um fenômeno de dois estados, em vez dos seis estados de um dado, que é o caso de uma moeda. Considere uma moeda com probabilidade $p$ de dar cara e probabilidade $1 - p$ de dar coroa. A entropia dessa moeda é expressa por

$$H(p) = p \lg(1/p) + (1 - p) \lg(1/(1 - p)).$$

Observando o gráfico da função $H(p)$, exposto na Figura 2, percebemos imediatamente que a entropia é máxima quando a moeda é honesta, isto é, quando $p = 0{,}5$.

[21] Shannon usou a letra $H$ em homenagem ao "Teorema H" da mecânica estatística, provado por Ludwig Boltzmann no século dezenove. A palavra "entropia" também decorre do mesmo tema: a física já usava a palavra para descrever uma propriedade de um sistema termodinâmico. Existe uma analogia entre esses sistemas e a noção dada de entropia de Shannon, o que explica sua escolha.

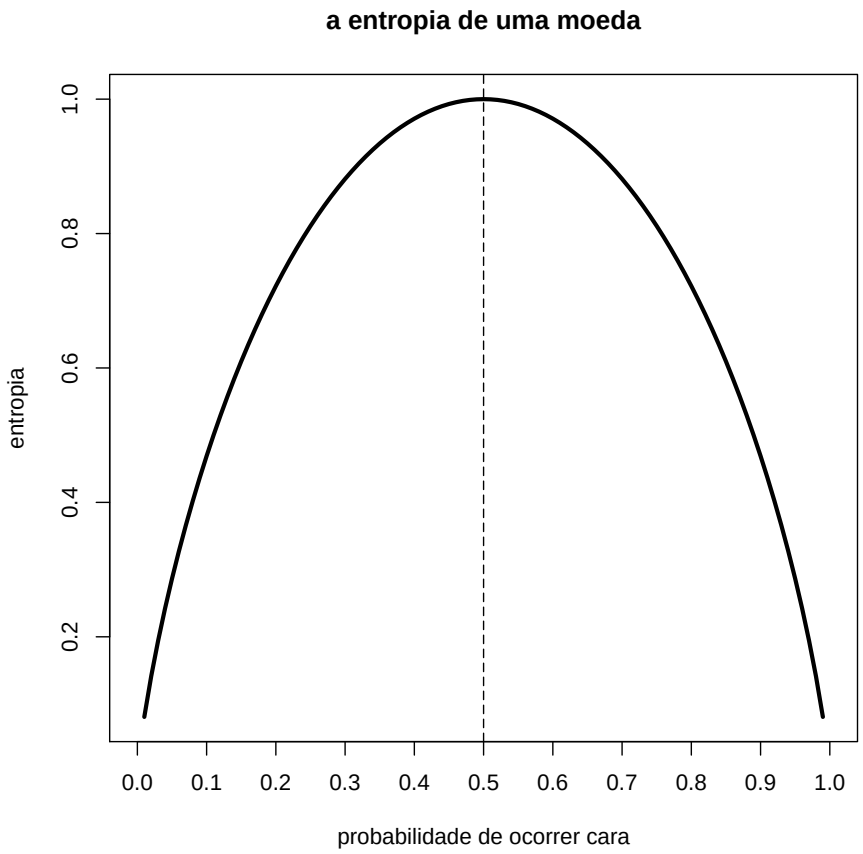


Figura 2: A entropia de uma moeda.

Similarmente, para um dado de seis lados, a entropia máxima é o número $6 \times (1/6) \lg 6$, um valor teórico que pode ser usado para comparar a entropia de um dado material, desses que encontramos em jogos de tabuleiro. Quando nos fornecem um dado material que não conhecemos, não sabemos qual é a sua distribuição de probabilidade, consequentemente não conhecemos sua entropia, mas sabemos qual é sua máxima entropia. Pode-se assim vislumbrar uma série de testes que possam estimar a entropia do dado desconhecido, levando-se em consideração que sua entropia mínima possível é zero e sua entropia máxima possível é próxima de 2,58 bits de informação.

Suponha, por exemplo, que nos forneçam um gerador de números aleatórios e nos apresentem um argumento teórico afirmando que o gerador produz $x$ bits de informação a cada unidade de tempo, sendo $x$ um valor próximo do máximo teórico. Em seguida, usando estimativas de entropia, suponha que obtenhamos fortes evidências de que o gerador produza metade da entropia reivindicada. Temos, portanto, informações conflitantes. Se as estimativas de entropia são confiáveis, temos então evidências de que a implementação do gerador não é fiel ao projeto. Se, por outro lado, as estimativas não são confiáveis, não temos como confirmar a implementação do gerador relativa a seu projeto. Confirmar ou refutar o argumento teórico do fornecedor é uma das propostas do documento NIST SP 800-90B [1, requerimento 3, página 19]:

> Documentation shall provide an explicit statement of the expected entropy provided by the noise source outputs and provide a technical argument for why the noise source can support that entropy rate. To support this, documentation may include a stochastic model[22] of the noise source outputs, and an entropy estimation based on this stochastic model may be included.

O documento especifica princípios e requerimentos para fontes de entropia que sejam usados como elementos de verdadeira aleatoriedade em geradores de bits aleatórios. Além disso, oferece estratégias

[22] *Stochastic model* ou modelo estocástico é sinônimo de modelo probabilístico.

estatísticas para estimar a entropia de uma fonte. Paralelo ao documento, há uma implementação[23] dessas estratégias com a qual podemos investigar uma fonte de entropia.

O documento faz uma distinção entre fontes de entropia. Uma fonte é IID — *independent and identically distributed* — se cada amostra produzida pela fonte tem a mesma distribuição de probabilidade que qualquer outra amostra e quaisquer amostras são independentes duas a duas. Isso significa que cada informação produzida por uma fonte IID é independente de qualquer informação produzida anteriormente. Por exemplo, um dado (honesto ou não honesto) é uma fonte IID: o dado "não lembra" de seus últimos lançamentos, logo os resultados prévios não afetam o próximo lançamento. O mesmo não pode ser dito sobre um PRNG porque o algoritmo mantém um estado interno, uma memória interna, fazendo com que toda informação prévia seja responsável pela próxima informação produzida.

O fornecedor de uma fonte de entropia deve apresentar a reivindicação de que sua fonte seja IID ou não-IID por via de um argumento teórico. Se a reivindicação é de que seja IID, então o documento provê estratégias para se verificar se há evidências que corroboram a reivindicação. Se a reinvindicação aponta uma fonte não-IID, outras estratégias são providas. O fornecedor também deve prover uma estimativa inicial da quantidade de informação que seja provida por sua fonte.

Vejamos um exemplo de se usar as estratégias do NIST SP 800-90B para uma fonte IID. Seja `random.dat` um arquivo contendo uma amostra em formato binário coletada por um laboratório diretamente do gerador de certo fabricante que reivindica uma entropia de 7,99 bits de informação por cada byte. Usando o programa `ea_iid`, provido pela implementação do NIST SP 800-90B, testamos a amostra:

```
%./ea_iid random.dat | grep .
Calculating baseline statistics...
H_original: 7.899807
H_bitstring: 0.998377
min(H_original, 8 X H_bitstring): 7.899807
** Passed chi square tests
** Passed length of longest repeated substring test
Beginning initial tests...
Beginning permutation tests... these may take some time
** Passed IID permutation tests
```

O programa afirma que a amostra contém dados que corroboram a reivindicação de que a fonte é IID, afinal 7,89 não é muito distante de 7.99. Entretanto, não será fácil determinar um ponto de corte em que a conclusão seria negativa.

Se a reivindicação fosse tal que a fonte seja não-IID, um outro conjunto de testes é oferecido pelo NIST SP 800-90B e implementado pelo programa `ea_non_iid`, cujo uso é análogo:

```
%./ea_non_iid ./random.niid.bin | grep .
Running non-IID tests...
Running Most Common Value Estimate...
Running Entropic Statistic Estimates (bit strings only)...
Running Tuple Estimates...
Running Predictor Estimates...
H_original: 7.295390
H_bitstring: 0.897841
min(H_original, 8 X H_bitstring): 7.182727
```

Como outro exemplo, consideremos uma fonte sem qualquer entropia, como este procedimento:

```
def entropy_sha256_numeric_strings():
  import hashlib
```

[23] Disponível em `https://github.com/usnistgov/SP800-90B_EntropyAssessment`.

```
f = open("random-sha256.bin", "wb")
for i in range(0, 31250):
  bs = bytes("{}".format(i), "utf-8")
  xs = hashlib.sha256(bs).digest()
  f.write(xs)
f.close()
```

O procedimento computa 31250 *hashes* SHA-256, cada um produzindo 32 bytes. Assim, no total obtemos $31250 \times 32 = 1000000$, tamanho mínimo da amostra desejada pelas ferramentas NIST SP 800-90B. Os *hashes* são computados a partir das cadeias de caracteres `0`, `1`, `2`, ..., `31249`. Como não há qualquer aleatoriedade nessa produção de dados, não há qualquer entropia a ser calculada. Mesmo assim, a forma como o algoritmo SHA-256 embaralha os dados faz com a ferramenta considere os dados IID e estima a entropia em aproximadamente 7,87 bits de informação por byte.

```
%./ea_iid random-sha256.bin | grep .
Calculating baseline statistics...
H_original: 7.875447
H_bitstring: 0.998517
min(H_original, 8 X H_bitstring): 7.875447
** Passed chi square tests
** Passed length of longest repeated substring test
Beginning initial tests...
Beginning permutation tests... these may take some time
** Passed IID permutation tests
```

Esse resultado ilustra a importância de um argumento teórico que afirme a entropia da uma fonte. Se a fonte não tem entropia, não há entropia a ser estimada e as ferramentas providas pelo NIST SP 800-90B serão enganadas com essa facilidade.

Apesar de dificuldades [16, 31, 15] apontadas pela literatura, hoje não parecer haver alternativas ao NIST SP 800-90B.

Cabe também notar que o fundamento do NIST SP 800-90B não é inferir a entropia de uma fonte por via de suas amostras, mas sim verificar se resultados absurdos aparecem oriundos de uma fonte que não poderia produzi-los, indicando assim um mal funcionamento na engenharia do gerador: é uma verificação de sanidade e não uma estimativa de entropia. Assim, um fabricante que não revele seu projeto, que não exponha seu argumento teórico a respeito da entropia de sua fonte, não satisfaz o mínimo necessário para ter a entropia de sua fonte verificada por um laboratório. É fácil enganar esses testes e, consequentemente, a evidência produzida em laboratório não serviria como qualquer argumento a favor do gerador sob análise.

Parte da dificuldade do problema é que entropia depende do modelo teórico usado pela fonte [11, seção 6.3, página 109].

> It is important to realize that a remark like "Consider the entropy of the source" can have no meaning unless a model of the source is included. Using random numbers as an example, you have formulas for generating pseudorandom numbers. Suppose that we had a table of such numbers. If you do not recognize that they are pseudorandom numbers, then you would probably compute the entropy based on the frequencies of occurrence of the individual numbers. Since pseudorandom-number generators do a good job of simulating random numbers, you would also find that each new number came as a complete surprise. But if you knew the structure of the formula used to generate the table, you would, after a few numbers, be able to predict perfectly the next number — there would be no surprise. Your estimate of the entropy of a source of symbols therefore depends on the model you adopt of the structure of the symbols.

Se o modelo apresentado pela fonte não incluir uma distribuição de probabilidade, a fórmula de Shannon não serve ao propósito de expressar a entropia e, nesse caso, será preciso avaliar especificamente o que o modelo apresenta, ou seja, uma solução específica para o caso é necessária.

# Apêndices

## A Problemas de montagem e de execução do `crush`

Para montar programas usando essas bibliotecas, seu compilador precisa estar programado para olhar o diretório `/usr/local/lib`.

```
%ls -l /usr/local/lib/libtestu01.a
-rw-r--r-- 1 root root 5340988 Nov  8  2018 /usr/local/lib/libtestu01.a
%ls -l /usr/local/lib/libtestu01.so.0.0.1
-rwxr-xr-x 1 root root 3130936 Nov  8  2018 /usr/local/lib/libtestu01.so.0.0.1
```

Se ele não estiver programado para olhar o diretório `/usr/local/lib`, ele não encontra as bibliotecas. Similarmente, para executar um programa montado com essas bibliotecas, seu sistema precisa saber onde encontrá-las. Provavelmente tanto seu compilador quanto seu sistema já estão programados para isso, mas se não o estiverem, você encontrará problemas e pode resolvê-los instruindo seu compilador e seu sistema adequadamente.

Se você consegue compilar o programa, mas não consegue executá-lo, há alguma probabilidade de que seu sistema não saiba onde encontrar as bibliotecas da TestU01. Se você disser `ldd crush` ao seu *shell*, ele provavelmente responderá "not found" relativo à libtestu01.so. Por exemplo,

```
%ldd crush
        linux-vdso.so.1 (0x00007ffe34947000)
        libtestu01.so.0 => not found
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fb115070000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fb11526a000)
```

Isso significa que a `libtestu01.so.0` não está num diretório em que o montador do seu sistema considera na hora de montar o executável em memória. Para produzir esse exemplo em que obtivemos o *not found* acima, propositalmente movemos a biblioteca para o diretório `/tmp`. Mas, sabendo que ela está lá, podemos usar a variável de ambiente `LD_LIBRARY_PATH` para instruir o montador a encontrá-la:

```
%LD_LIBRARY_PATH=/tmp ldd crush
        linux-vdso.so.1 (0x00007ffee01f2000)
        libtestu01.so.0 => /tmp/libtestu01.so.0 (0x00007f1dfac15000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f1dfaa24000)
        libprobdist.so.0 => /usr/local/lib/libprobdist.so.0 (0x00007f1dfa9f8000)
        libmylib.so.0 => /usr/local/lib/libmylib.so.0 (0x00007f1dfa9eb000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f1dfa85e000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f1dfaced000)
```

Agora o montador tem tudo que precisa e você poderia executar o `crush` de forma similar:

```
%LD_LIBRARY_PATH=/tmp ./crush --help
Usage: ./crush [options]
 Tests your data for randomness against TestU01.

Examples:
cat /dev/urandom | crush -b small -n 'the local generator'
xorshift32 | crush -b small -n xorshift32
```

```
 The options are:
-b, --battery    Your choice of battery (small, medium, big)
-n, --name       The name of your generator
-h, --help       Display this information
```

Para instruir seu sistema a olhar esses diretórios de forma permanente, é preciso consultar a documentação do seu sistema. Um sistema típico UNIX costuma ter um programa chamado `ldconfig` que compila uma base de dados usualmente armazenada no arquivo `libc.conf`, que no nosso sistema reside em `/etc/ld.so.conf.d/libc.conf`.

# B Compilando o *PractRand* em sistemas UNIX

Duas ferramentas que concorrem com a TestU01 são o pacote *PractRand*[24] e o *gjrand*[25]. Como vimos anteriormente, o *PractRand* tem a interessante característica de aumentar gradativamente o tamanho das amostras até que o gerador sob teste fracasse em algum teste estatístico, o que nos permite ter uma medida do quão evidentemente inseguro é o gerador. Nesta seção descrevemos como instalar a ferramenta em sistemas UNIX e contornamos as dificuldades de fazê-lo.

A documentação[26] relativa à instalação nos fornece os comandos necessários para compilar o pacote, mas não informa que o código-fonte incluído no pacote não compilará sem alguns ajustes.

Como primeiro passo, copie o pacote para seu diretório local. Certifique-se de que a opção dada ao `wget` usa a letra maiúscula; o descuido de digitar uma letra minúscula produzirá um comando bem sucedido, mas incorreto. Descompacte o pacote em seguida.

```
%wget https://bit.ly/31UxwxL -O PractRand_0.94.zip
[...]
%cd PractRand_094/
%unzip PractRand_0.94.zip
[...]
%ls
bin  doc  include  lib  src  tools  unix  VisualC_net
```

Antes de compilar o código-fonte em `src/`, precisamos fazer correções. São elas:

```
%cd include/
%ln -s Tests tests
%cd Tests/
%ln -s nearseq.h NearSeq.h
%ln -s coup16.h Coup16.h
%ln -s birthday.h Birthday.h
```

Essas correções sugerem que o autor é usuário de sistemas Windows, onde o sistema de arquivos não diferencia letra maiúscula de minúscula. Enquanto o código-fonte faz referência a arquivos como `NearSeq.h`, o arquivo em si é chamado originalmente de `nearseq.h`. Os comandos acima criam nomes simbólicos para os novos arquivos, ou seja, quando o compilador pedir pelo arquivo `NearSeq.h`, nome que originalmente não existe no pacote, o sistema de arquivos entregará o arquivo `nearseq.h`, que é o nome de fato dado pelo autor. Assim, o compilador recebe o arquivo `NearSeq.h` que o código-fonte deseja e a compilação é bem sucedida.

Compile o código-fonte. O comando abaixo leva menos de 2 minutos em nossos sistemas. Alguns avisos podem ser emitidos por seu compilador.

[24] Homepage: `http://pracrand.sourceforge.net`
[25] Homepage: `http://gjrand.sourceforge.net`
[26] Disponível em `https://bit.ly/2Y5N8xc`.

```
%g++ -c src/*.cpp src/RNGs/*.cpp src/RNGs/other/*.cpp -O3 -Iinclude -pthread
[...]
```

A compilação produzirá uma série de arquivos chamados de *object files*.

```
%ls *.o
arbee.o        isaac.o        platform_specifics.o  simple.o
chacha.o       jsf.o          rand.o                test_batteries.o
efiix.o        math.o         salsa.o               tests.o
fibonacci.o    mt19937.o      sfc.o                 transform.o
hc256.o        mult.o         sha2_based_pool.o     trivium.o
indirection.o  non_uniform.o  sha2.o                xsm.o
```

Crie a biblioteca `libPractRand.a`:

```
%ar rcs libPractRand.a *.o
```

Compile e monte os executáveis do *PractRand*. Os programas dependem da biblioteca chamada `pthread`, mas seu sistema deve tê-la instalado em sua maior probabilidade.

```
%g++ -o RNG_test tools/RNG_test.cpp libPractRand.a -O3 -Iinclude -pthread
%file RNG_test
RNG_test: ELF 64-bit LSB shared object, x86-64 [...] not stripped
%g++ -o RNG_output tools/RNG_output.cpp libPractRand.a -O3 -Iinclude -pthread
%file RNG_output
RNG_ouxtput: ELF 64-bit LSB shared object, x86-64 [...] not stripped
```

Faça um teste contra seu gerador local.

```
%./RNG_test stdin -tlmax 128MB < /dev/urandom
RNG_test using PractRand version 0.94
RNG = RNG_stdin, seed = unknown
test set = core, folding = standard(unknown format)

rng=RNG_stdin, seed=unknown
length= 32 megabytes (2^25 bytes), time= 2.2 seconds
  no anomalies in 167 test result(s)

rng=RNG_stdin, seed=unknown
length= 64 megabytes (2^26 bytes), time= 5.9 seconds
  no anomalies in 179 test result(s)

rng=RNG_stdin, seed=unknown
length= 128 megabytes (2^27 bytes), time= 11.7 seconds
  no anomalies in 196 test result(s)
```

# C A entropia algorítmica de Kolmogorov

Considere duas sequências binárias [30, seção 6.4, página 262]

$$A = 01010101010101010101010101010101010101,$$
$$B = 11100101101000111010100001110100110101111.$$

Podemos afirmar que $A$ contém pouca informação porque ela é a repetição de zero seguido de um, vinte vezes. Com relação a $B$, dizer uma regra que possa resumi-la não parece tão fácil.

A sugestão de Kolmorogov é dizer que a quantidade de informação contida numa sequência é o tamanho de sua menor descrição. Para evitar a vagueza da palavra "descrição", podemos recorrer à Teoria da Computação, sugerindo que por "descrição" se entenda uma máquina de Turing capaz de produzir a sequência, ou seja, um algoritmo que produza a sequência. Sendo assim, a quantidade de informação contida numa sequência é o tamanho do menor algoritmo capaz de produzi-la. Se não conseguimos encontrar uma forma de resumir $B$, teríamos que recorrer a um algoritmo que simplesmente repetisse $B$ símbolo por símbolo, o que tomaria pelo menos tanto espaço quanto a própria sequência, seja qual for o meio de descrição dos algoritmos [30, teorema 6.27, página 266].

Uma máquina de Turing é uma noção matemática cujo objetivo é sugerir-se como a definição de um procedimento que seja "efetivamente calculável", termo usado por Alan Turing para descrever a ideia das coisas que uma máquina seja capaz de fazer. Pode-se descrever uma máquina de Turing por uma sequência de símbolos. Em particular, pode-se descrever uma máquina de Turing por uma sequência de zeros e uns. Assim, a máquina de Turing mais curta que descreva uma sequência nos dá a quantidade de informação contida na sequência. Podemos dizer "a máquina de Turing mais curta" porque se houver mais de uma com mesmo tamanho, resolvemos a ambiguidade por enumerá-las em ordem alfabética e *arbitrar* que a primeira da lista é a menor [30, definição 6.23, página 264].

# D O papel do logaritmo na fórmula de Shannon

Antes de discutirmos a função do logaritmo, vejamos o fato de que a fórmula descreve uma média ponderada. Dada uma distribuição de probabilidade $f$ relativa a uma fonte de aleatoriedade que emite símbolos pertencentes a um conjunto finito de símbolos $s_1, s_2, ..., s_n$, a entropia de $f$ é, por definição,

$$\sum_{i=1}^{n} -p_i \lg(p_i) = \sum_{i=1}^{n} p_i \lg(1/p_i)$$

bits de informação, sendo $p_i = f(s_i)$ a probabilidade da fonte emitir o símbolo $s_i$.

Para entendermos que essa fórmula expressa uma média, olhemos o caso concreto de uma média qualquer. Usualmente, calcula-se uma média somando-se os valores que temos e dividimos a soma pela quantidade de valores somados. Por exemplo, qual a média de pontos obtidos no lançamento de um dado honesto? Para responder, computamos

$$\frac{1}{6}(1+2+3+4+5+6) = 3{,}5$$

porque, sendo honesto, qualquer lado tem a mesma probabilidade. Quando o dado não é honesto, as probabilidades de cada lado não são iguais. Se um dado de seis lados tem probabilidades $p(1) = 0{,}1$, $p(2) = 0{,}2$, $p(3) = 0{,}3$, $p(4) = 0{,}2$, $p(5) = 0{,}1$ e $p(6) = 0{,}1$, então a cada lançamento obtemos, em média, os pontos

$$0{,}1 + 2 \times 0{,}2 + 3 \times 0{,}3 + 4 \times 0{,}2 + 5 \times 0{,}1 + 6 \times 0{,}1 = 3{,}3$$

Nessa expressão, diz-se que a quantidade de pontos é ponderada pela probabilidade de ocorrência. Similarmente, a fórmula de Shannon também computa uma média porque o termo $\lg(1/f(s_i))$ é ponderado pela probabilidade $f(s_i)$. A fórmula expressa a quantidade de informação que se obtém — em média — a cada ocorrência de um dos símbolos $s_i$ emitidos pela fonte.

Agora nos resta compreender por que o logaritmo é usado. Faremos isso tomando o lugar de Shannon. Suponha que não tenhamos a fórmula e vejamos como obter uma fórmula qualquer que sirva ao propósito de medir a quantidade de informação produzida por uma fonte de informação que produz símbolos $s_i$ de um conjunto finito de símbolos $\{s_1, ..., s_n\}$.

Consideremos primeiro a quantidade de informação de um evento apenas. Seja $s_i$ esse evento e seja $H$ a função que expressa a quantidade de informação relativa a $s_i$. Arbitremos que $H$ descreverá a quantidade de informação em função da probabilidade $p_i$ de $s_i$. Sendo assim, $H(p_i)$ é a quantidade de informação transmitida na ocorrência do evento $s_i$. Essa primeira decisão sugere o domínio de $H$: o conjunto das probabilidades $p_i$ dos eventos $s_i$ emitidos pela fonte. Desejamos um número não-negativo

para representar "quantidade de informação" porque não nos parece razoável medir informação por quantidades negativas. Assim, arbitremos que $H(p) \geq 0$.

Para dois eventos independentes $e_1, e_2$ de probabilidades $p_1, p_2$, o número $p_1 p_2$ pertence ao domínio de $H$ e, portanto, $H(p_1 p_2)$ é bem-definido. Sendo eventos independentes, é razoável que requisitemos que $H$ satisfaça

$$H(p_1 p_2) = H(p_1) + H(p_2)$$

porque a ocorrência do evento $e_1$ não nos dá qualquer informação sobre a subsequente (ou prévia) ocorrência de $e_2$. Logo, se a fonte emite $e_1$ seguida de $e_2$, é razoável que arbitraremos que temos a soma da informação transmitida pelos dois eventos. Isso significa que a função de que estamos em busca satisfaz esse propriedade. Assim, se $p_1 = p = p_2$ são as probabilidades dos eventos independentes $e_1, e_2$, então

$$H(p_1 p_2) = H(p^2) = H(p) + H(p) = 2H(p).$$

Similarmente, se $p_1 = p$, $p_2 = p^2$, deduzimos

$$H(p_1 p_2) = H(pp^2) = H(p) + H(p^2) = H(p) + 2H(p) = 3H(p).$$

Requisitamos, portanto, que $H$ tenha a propriedade

$$H(p^n) = nH(p)$$

para todo número natural $n$. Essa propriedade sugere que $H$ descreve expoentes. Como expoentes são logaritmos e logaritmos são expoentes, então $H$ descreve uma função logarítmica.

Vejamos em detalhes. Sabemos que, por definição, $\log_x y$ é o número $t$ tal que $y = x^t$, sendo $x \neq 0$. Dessa definição é imediata a observação de que $\log y$ é um expoente: o definimos assim. Se temos a expressão $x^t = y$ e aplicamos $\log_x$ a ambos lados da equação, obtemos $\log_x(x^t) = \log_x(y)$. Certamente $\log_x(x^t) = t$ porque a definição de $\log_x$ demanda que encontremos o número que seja o expoente de $x$ que resulte em $x^t$. Logo, $t = \log_x(y)$, o que nos remete ao teorema que nos afirma que $\log_x(x^t) = t \log_x(x)$, precisamente como arbitramos como requerimento para $H$. Logo, uma função logarítmica numa base qualquer serve como solução.

Precisamos escolher uma base. Seja $y = \log_a x$. Sabemos que $a^y = x$. Aplique $\log_b$ a ambos lados desta última equação e obtenha $y \log_b a = \log_b x$. Logo,

$$\log_a x = y = \frac{\log_b x}{\log_b a} = (\log_a b) \log_b x,$$

mostrando que qualquer logaritmo pode ser computado em função de um outro logaritmo. Se escolhemos $b = 2$, obtemos

$$\log_a x = (\log_a 2) \lg x$$

e a expressão de $H$ passa a ser $H(p) = \log_a 2 \lg p$. Se declaramos $k = \log_a 2$, obtemos $H = k \lg p$. Se $p \in (0, 1)$, então $\lg p < 0$, mas precisamos satisfazer $H(p) \geq 0$, o que nos sugere a tomar $k = -1 = \log_a 2$, ou seja, escolher $k = -1$ é o mesmo que escolher $a = 1/2$. Assim, finalmente, obtemos

$$H(p) = -\lg p = \lg(1/p),$$

como desejado.

Incluir expoentes racionais na definição de $H$ requer alguma manipulação simbólica. Se $y = p^n$, então $y^{1/n} = p$ e

$$H(y) = H(p^n) = nH(p) = nH(y^{1/n}).$$

Esse resultado nos mostra que para qualquer argumento de $H$, podemos anexar um fator $n$ "fora da função $H$" desde que incluamos "dentro da função $H$" um expoente $1/n$. Essa estratégia nos dá o direito de tratar probabilidades $p^{1/n}$. Para obter um racional qualquer $m/n$, declare $y = p^{m/n}$, obtenha $y^{m/n} = p$ e deduza

$$H(y) = H(p^{m/n})$$

$$
\begin{aligned}
&= mH\Big(p^{(m/n)(1/m)}\Big) \\
&= mH\Big(p^{1/n}\Big) \\
&= \frac{1}{n}mH\Big(p^{(1/n)[1/(1/n)]}\Big) \\
&= \frac{m}{n}H\Big(p^{(1/n)n}\Big) \\
&= \frac{m}{n}H(p),
\end{aligned}
$$

como desejado.

Para expoentes irracionais, entretanto, precisamos da continuidade de $H$. Se $H$ é uma função contínua, então por definição $\lim H(p^x) = H(p^y)$ com $x \to y$. Seja $X_n$ uma sequência de números racionais que convirja para $y$. Se $x$ é um elemento da sequência $X_n$, então $H(p^x) = xH(p)$. Daí

$$
\begin{aligned}
H(p^y) &= \lim_{x\to y} H(p^x) \\
&= \lim_{x\to y} xH(p) \\
&= \Big(\lim_{x\to y} x\Big)H(p) \\
&= yH(p),
\end{aligned}
$$

como desejado. Assim definimos $H$ para todos os argumentos $p^r$, sendo $r$ um número real.

Assumimos que $H(p) \geq 0$ para todo $p \in [0,1]$, assumimos $H(p_1p_2) = H(p_1) + H(p_2)$ para quaisquer eventos independentes $s_1, s_2$ cujas probabilidades de ocorrência são $p_1, p_2$ respectivamente bem como $H$ como função contínua. A propriedade aditiva de $H$ nos apontou que uma função logarítmica satisfaz os requisitos de $H$. Resta-nos saber se existem outras. Suponha que exista uma outra e chame-a de $J$. Então $J(p^x) = xJ(p)$. Além disso, já temos $H(p^x) = xH(p)$. Subtraindo esta última da primeira, obtemos

$$J(p^x) - H(p^x) = x(J(p) - H(p)).$$

Como $H(p^x) = k\lg(1/p^x)$, reescrevemos a equação anterior como

$$J(p^x) - k\lg(1/p^x) = x(J(p) - k\lg(1/p)). \tag{1}$$

Escolhendo

$$k = \frac{J(p)}{\lg(1/p)},$$

obtemos zero no lado direito da Equação (1), ou seja,

$$
\begin{aligned}
J(p^x) - k\lg(1/p^x) &= x\Big(J(p) - \frac{J(p)}{\lg(1/p)}\lg(1/p)\Big) \\
&= x(J(p) - J(p)) \\
&= 0.
\end{aligned}
$$

Para terminarmos, declare $z = p^x$. Para todo $z$, existe número real $x$ tal que $p^x = z$. Assim, substituindo $z$ na equação anterior, obtemos

$$
\begin{aligned}
J(p^x) - k\lg(1/p^x) &= 0 \\
J(z) &= k\lg(1/z) \\
&= H(z),
\end{aligned}
$$

mostrando que não há outras funções como suposto inicialmente; a função logarítmica é a única. Assim, a função $H(p_i)$ que desejamos tem essencialmente a forma $\lg(1/p_i)$, ou seja, uma forma logarítmica.

Então o papel do logaritmo na fórmula de Shannon é "centro das atenções" porque a noção de quantidade de informação concebida por Shannon é expressa quantitativamente por uma função logarítmica.

# Referências


[1] Elaine Barker, John Kelsey e John Bryson Secretary. *NIST DRAFT Special Publication 800-90B Recommendation for the Entropy Sources Used for Random Bit Generation*. 2012.

[2] Daniel Chicayban Bastos e Luis Antonio Brasil Kowada. "Medindo a qualidade de geradores de números aleatórios". Em: *IV Workshop sobre Regulação, Avaliação da Conformidade, Testes e Padrões de Segurança*. Galoá. Rio de Janeiro, Brasil., 2018.

[3] Daniel Chicayban Bastos, Luis Antonio Brasil Kowada e Raphael C. S. Machado. "Measuring randomness in IoT products". Em: *2019 II Workshop on Metrology for Industry 4.0 and IoT*. IEEE. 2019, pp. 466–470.

[4] Daniel Chicayban Bastos, Luis Antonio Brasil Kowada e Raphael CS Machado. "On pseudorandom number generators". Em: *ACTA IMEKO* 9.4 (2020), pp. 128–135. ISSN: 2221-870X.

[5] Norman Blaikie. *Analyzing Quantitative Data: From Description to Explanation*. Sage, 2003. ISBN: 0-7619-6758-3.

[6] Wilton Bussab e Pedro A. Morettin. *Estatística Básica*. 6ª edição. Saraiva, 2010. ISBN: 978-85-02-08177-2.

[7] Chris Doty-Humphrey. *PractRand*. 2014. URL: http://pracrand.sourceforge.net.

[8] David Freedman, Robert Pisani e Roger Purves. *Statistics*. 4ª ed. Norton, 1998.

[9] Carl Friedrich Gauss. *Abhandlungen zur Methode der kleinsten Quadrate*. P. Stankiewicz, 1887.

[10] Ian Goldberg e David Wagner. "Randomness and the Netscape browser". Em: *Dr Dobb's Journal-Software Tools for the Professional Programmer* 21.1 (1996), pp. 66–71.

[11] Richard W. Hamming. *Coding and information theory*. Englewood Cliffs, NJ, Prentice Hall, 1986. ISBN: 0-13-139072-4.

[12] American National Standards Institute. *ANSI X3.159-1989*. 1989.

[13] Fred James. "RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Lüscher". Em: *Computer Physics Communications* 79.1 (1994), pp. 111–114.

[14] Gary Johnson. *gjrand*. 2014. URL: http://gjrand.sourceforge.net/.

[15] Anna Johnston. "Comments on Cryptographic Entropy Measurement." Em: *IACR Cryptol. ePrint Arch.* 2019 (2019), p. 1263.

[16] John Kelsey, Kerry A. McKay e Meltem Sönmez Turan. "Predictive Models for Min-entropy Estimation". Em: *Cryptographic Hardware and Embedded Systems – CHES 2015*. Ed. por Tim Güneysu e Helena Handschuh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 373–392. ISBN: 978-3-662-48324-4.

[17] Donald Ervin Knuth. *The Art of Computer Programming, volume 2, seminumerical algorithms*. 3ª ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 978-0-201-89684-8.

[18] Pierre L'Ecuyer e Richard Simard. *A Software Library of Probability Distributions and Goodness-of-Fit Statistics in ANSI C*. Rel. técn. Département d'Informatique et de Recherche Opérationnelle, Université e de Montréal, 2002.

[19] Pierre L'Ecuyer e Richard Simard. "TestU01: a C library for empirical testing of random number generators". Em: *ACM Transactions on Mathematical Software (TOMS)* 33.4 (2007).

[20] Pierre L'ecuyer et al. "An object-oriented random-number package with many long streams and substreams". Em: *Operations research* 50.6 (2002), pp. 1073–1075.

[21] Pierre L'Ecuyer e Richard Simard. "TestU01: a software library in ANSI C for empirical testing of random number generators: User's guide, detailed version". Em: *Département d'Informatique et de Recherche Opérationnelle Université de Montréal* (2005).

[22] Pierre L'Ecuyer e Richard Simard. *TestU01: A software library in ANSI C for empirical testing of random number generators. Users guide, compact version. (Version: August 17, 2009).*

[23] Martin Lüscher. "A portable high-quality random number generator for lattice field theory simulations". Em: *Computer physics communications* 79.1 (1994), pp. 100–110.

[24] George Margasalia. "A Current View of Random Number Generators". Em: *The Proceedings of the 16th Symposium on the Interface, Atlanta 1984*. 1984.

[25] George Marsaglia. "Xorshift rngs". Em: *Journal of Statistical Software* 8.14 (2003), pp. 1–6.

[26] George Marsaglia e Liang-Huei Tsay. "Matrices and the structure of random number sequences". Em: *Linear algebra and its applications* 67 (1985), pp. 147–156.

[27] Makoto Matsumoto e Takuji Nishimura. "Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". Em: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1 (1998), pp. 3–30.

[28] Melissa E. O'Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Rel. técn. HMC-CS-2014-0905. Claremont, CA: Harvey Mudd College, set. de 2014.

[29] Claude E. Shannon. "A mathematical theory of communication". Em: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423.

[30] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012. ISBN: 978-1-133-18781-3.

[31] Shuangyi Zhu et al. "Analysis and improvement of entropy estimators in NIST SP 800-90B for non-IID entropy sources". Em: *IACR Transactions on Symmetric Cryptology* (2017), pp. 151–168.