Resumo Didático da Column 12

A Sample Problem — Programming Pearls

Arthur Prates Pessoti
Teo Giambarra Roseiro

Universidade Vila Velha
Grupo 4 — Turma CC5Mb
Disciplina: Teoria da Computação
Professor: Abrantes Araújo Silva Filho

6 de março de 2026

Resumo

Este artigo apresenta um resumo didático e estruturado da *Column 12* do livro *Programming Pearls*, intitulada *A Sample Problem*. O objetivo é servir como material de estudo futuro, explicando de forma clara a ideia central do capítulo: programar bem não é apenas escrever código, mas reformular corretamente o problema, explorar diferentes soluções e somente depois implementar a alternativa mais adequada.

O texto discute a transformação de um problema prático de amostragem aleatória em uma especificação abstrata mais elegante, o uso do algoritmo de Knuth para gerar uma seleção ordenada sem repetição, a análise de outras abordagens possíveis e os princípios de projeto que o autor deseja ensinar. O foco está menos em decorar um algoritmo específico e mais em compreender o raciocínio por trás da solução.

Sumário

1 Introdução

A *Column 12* do livro *Programming Pearls* apresenta um pequeno problema prático que, à primeira vista, parece simples: selecionar aleatoriamente alguns itens entre muitos possíveis. No entanto, Jon Bentley usa esse exemplo para ensinar algo bem maior do que apenas um algoritmo de sorteio.

A principal lição do capítulo é que muitos problemas de programação ficam mais fáceis quando são **reformulados corretamente**. Em vez de aceitar imediatamente a primeira descrição feita pelo usuário, o programador deve compreender o contexto, separar a necessidade real da solução inicialmente imaginada e criar uma formulação mais limpa, mais abstrata e mais útil.

Por isso, este capítulo é importante não apenas pelo algoritmo apresentado, mas pelo processo de pensamento que ele demonstra. O autor mostra como sair de um pedido concreto e aparentemente limitado para chegar a uma especificação elegante, geral e mais fácil de resolver.

2 O problema original apresentado no capítulo

O caso relatado por Bentley ocorre no início da popularização dos computadores pessoais em uma empresa de pesquisa de opinião pública. Um funcionário sugeriu usar um programa para automatizar a escolha de uma amostra aleatória de *precincts* (distritos eleitorais) a partir de uma lista impressa.

A formulação inicial era a seguinte:

> Receber uma lista com os nomes dos distritos e um inteiro m, e então devolver m nomes escolhidos aleatoriamente.

Em termos práticos, parecia um pedido razoável. Entretanto, Bentley percebe que a solução proposta pelo usuário já traz um problema embutido: para usar o programa, seria necessário digitar centenas de nomes, mesmo que o programa fosse aproveitar apenas uma pequena fração dessa informação.

Isso mostra uma distinção muito importante em desenvolvimento de software:

- o usuário normalmente descreve o problema na forma como ele imagina a solução;
- o programador precisa descobrir qual é a necessidade real por trás dessa descrição.

3 A reformulação correta do problema

Bentley propõe abandonar a entrada com todos os nomes e substituir o problema por uma formulação mais abstrata e muito mais eficiente.

Em vez de digitar a lista inteira, basta fornecer:

- m: quantos elementos devem ser escolhidos;
- n: quantos elementos existem no total.

A saída passa a ser:

> Uma lista ordenada de m inteiros distintos no intervalo $0..n-1$.

Cada inteiro representa a posição de um item na lista impressa. Assim, se o programa produzir algo como $4, 15, 17$, a pessoa apenas percorre a lista e marca o 4^{o}, o 15^{o} e o 17^{o} item.

Essa mudança melhora o problema em vários aspectos:

1. elimina digitação desnecessária;
2. reduz a chance de erro humano;
3. torna a solução mais geral;
4. transforma o problema em algo matematicamente mais claro.

Além disso, o capítulo exige uma condição probabilística importante: não basta que cada número individual tenha chance de aparecer. É necessário que **cada subconjunto possível de tamanho m tenha a mesma probabilidade de ser escolhido**.

Essa exigência é mais forte do que simplesmente dizer que cada elemento tem probabilidade m/n de aparecer.

4 A especificação abstrata

Depois da reformulação, o problema pode ser descrito de forma muito limpa:

> Gerar m inteiros distintos, em ordem crescente, no intervalo $0..n-1$, de modo que cada subconjunto possível de tamanho m seja equiprovável.

Essa é a verdadeira força do capítulo. Ao transformar um problema de escritório em um problema abstrato de seleção, o autor obtém uma especificação reutilizável em vários outros contextos, como:

- escolha de posições em vetores;
- geração de subconjuntos aleatórios;
- amostragem para testes;
- seleção de índices em estruturas de dados.

Ou seja, a solução encontrada deixa de servir apenas para um caso pontual e passa a ser útil como técnica geral de programação.

5 A solução principal do capítulo: o algoritmo de Knuth

Após definir corretamente o problema, Bentley recorre ao *Algorithm S*, de Donald Knuth. A ideia do algoritmo é simples e elegante:

- percorrer os números de 0 até $n - 1$ em ordem;
- decidir, para cada número, se ele entra ou não na amostra;
- ajustar a probabilidade de escolha conforme quantos elementos ainda faltam ser selecionados.

A grande vantagem desse método é que, como os candidatos são analisados em ordem, a saída já aparece ordenada naturalmente, sem necessidade de ordenação posterior.

5.1 Pseudocódigo central

```
select = m
remaining = n
for i = [0, n)
    if (bigrand() % remaining) < select
        print i
        select--
    remaining--
```

Nesse pseudocódigo:

- `select` representa quantos elementos ainda faltam ser escolhidos;
- `remaining` representa quantos candidatos ainda restam para examinar;
- `i` é o elemento atual sendo considerado.

6 Como entender a lógica do algoritmo

A regra do algoritmo é a seguinte:

> Se ainda faltam escolher s elementos entre r candidatos restantes, então o próximo elemento deve ser escolhido com probabilidade s/r.

Essa regra pode parecer estranha à primeira vista, mas ela garante equilíbrio ao longo de todo o processo.

6.1 Exemplo com $m = 2$ e $n = 5$

Queremos escolher 2 números entre os cinco valores:

$$\{0, 1, 2, 3, 4\}$$

Passo 1: decidir sobre o número 0

No início:

- faltam escolher 2 elementos;
- restam 5 candidatos.

Logo, o 0 deve ser escolhido com probabilidade:

$$\frac{2}{5}$$

Passo 2: decidir sobre o número 1

Agora existem dois casos.

Caso A: o 0 foi escolhido

- falta escolher apenas 1 elemento;
- restam 4 candidatos.

Então a chance de escolher 1 passa a ser:

$$\frac{1}{4}$$

Caso B: o 0 não foi escolhido

- ainda faltam escolher 2 elementos;
- restam 4 candidatos.

Então a chance de escolher 1 passa a ser:

$$\frac{2}{4}$$

Esse exemplo mostra a essência do algoritmo: a probabilidade não é fixa. Ela é recalculada dinamicamente conforme o estado atual da seleção.

7 Por que o algoritmo sempre funciona

O capítulo destaca três propriedades intuitivas muito importantes.

7.1 Ele nunca escolhe elementos demais

Quando `select` chega a zero, significa que a quantidade desejada já foi atingida. A partir desse momento, nenhuma nova escolha ocorre.

7.2 Ele nunca escolhe elementos de menos

Se em algum momento o número de elementos restantes for exatamente igual ao número ainda necessário, então todos os elementos restantes precisam ser escolhidos.

Em outras palavras, quando:

$$\texttt{select} = \texttt{remaining}$$

a probabilidade se torna 1, e a seleção desses elementos passa a ser obrigatória.

7.3 A saída já está ordenada

Como o algoritmo percorre os candidatos em ordem crescente e imprime imediatamente os escolhidos, o resultado final já sai ordenado. Não é necessário aplicar um *sort* depois.

8 Equiprobabilidade dos subconjuntos

Esse é um dos pontos mais importantes do capítulo. A exigência do problema não é apenas escolher números sem repetição. O algoritmo precisa garantir que qualquer subconjunto válido de tamanho m tenha a mesma chance de ocorrer.

Por exemplo, se queremos escolher 2 elementos entre 5, os subconjuntos possíveis são:

$$\{0,1\},\{0,2\},\{0,3\},\{0,4\},\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,4\},\{3,4\}$$

Todos esses subconjuntos devem ser igualmente prováveis.

Esse detalhe é essencial porque existem algoritmos que parecem aleatórios, mas na verdade introduzem viés. A solução de Knuth evita esse problema ao ajustar a probabilidade local com base no número de escolhas que ainda faltam.

9 Implementação prática e custo

Bentley comenta que a implementação final ficou extremamente pequena. Mesmo com entrada, saída e validações básicas, o programa ocupava poucas linhas e usava apenas algumas dezenas de bytes de memória.

Do ponto de vista de análise assintótica, a solução principal possui:

- **tempo:** $O(n)$;
- **memória extra:** $O(1)$.

Isso significa que o algoritmo é excelente em uso de memória, mas seu tempo depende linearmente de n. Para os problemas reais da empresa, isso era ótimo. Porém, para

valores gigantescos de n, a solução pode ficar lenta, porque todos os candidatos precisam ser visitados, mesmo que poucos sejam escolhidos.

10 O espaço de projeto: outras soluções possíveis

Após apresentar uma boa solução, Bentley não encerra a discussão. Pelo contrário: ele explora outras alternativas para mostrar que programar bem também significa conhecer o **espaço de soluções possíveis**.

Essa parte do capítulo é extremamente importante para estudo, porque ensina a não se apaixonar cedo demais pela primeira solução correta.

10.1 Solução baseada em conjunto (*set*)

Uma ideia simples é manter um conjunto inicialmente vazio. O programa gera números aleatórios no intervalo $0..n-1$ e os insere no conjunto apenas se ainda não estiverem presentes. Quando o conjunto atingir tamanho m, basta imprimir seus elementos em ordem.

Em alto nível:

```
initialize set S to empty
while size(S) < m
    t = random integer in 0..n-1
    if t not in S
        insert t into S
print S in sorted order
```

Essa abordagem tem pontos positivos:

- é conceitualmente simples;
- o código pode ficar muito curto, principalmente com bibliotecas prontas;
- funciona bem quando m é pequeno em relação a n.

Mas ela também tem limitações:

- usa memória proporcional a m;
- quando o conjunto cresce, surgem mais colisões e mais tentativas desperdiçadas;
- para m muito grande, o custo de memória pode se tornar pesado.

Em resumo, essa alternativa pode ser ótima em alguns cenários, mas não é universal.

10.2 Solução baseada em embaralhamento parcial

Outra possibilidade é criar um vetor com todos os números de 0 até $n-1$, realizar trocas aleatórias apenas na primeira parte do vetor, ordenar os m primeiros valores e imprimi-los.

A estrutura geral é:

1. criar um vetor com n elementos;
2. preencher o vetor com $0, 1, 2, \ldots, n-1$;
3. fazer trocas aleatórias para embaralhar parcialmente;
4. ordenar os m primeiros elementos;
5. imprimir a resposta.

Essa solução evita repetições de forma natural, mas tem custo alto de memória, pois precisa armazenar todo o universo de elementos.

Seu custo típico é:

- **tempo:** $O(n + m \log m)$;
- **memória:** $O(n)$.

Por isso, ela costuma perder para a solução de Knuth quando n é grande.

11 Comparando as abordagens

A comparação entre algoritmos pode ser resumida da seguinte forma:

Abordagem	**Custo principal**	**Memória**	**Observação**
Algoritmo de Knuth	$O(n)$	$O(1)$	Muito simples, elegante e econômico em memória; pode ficar lento se n for enorme.
Conjunto (set)	Aproximadamente $O(m \log m)$ quando $m \ll n$	$O(m)$	Bom quando m é pequeno, mas sofre com colisões e crescimento de memória.
Embaralhamento parcial	$O(n + m \log m)$	$O(n)$	Evita repetições naturalmente, porém exige memória alta para armazenar todo o universo.

Essa tabela deixa clara uma lição importante: **não existe melhor algoritmo no vácuo**. O melhor algoritmo depende dos parâmetros do problema e das restrições de ambiente.

12 Quando uma solução pode ser melhor que outra

Bentley também chama atenção para o fato de que o espaço de projeto vai além das três abordagens principais. Em alguns casos, por exemplo:

- se m estiver muito próximo de n, pode ser melhor gerar os poucos elementos que ficam de fora e depois tomar o complemento;
- outras soluções podem ser mais adequadas se a prioridade for reduzir o número de sorteios aleatórios;
- dependendo das bibliotecas disponíveis, uma abordagem de nível mais alto pode economizar muito tempo de implementação.

Esse ponto é valioso para estudos porque mostra uma mentalidade de engenharia: a análise não termina quando encontramos uma solução correta; ela continua até entender quando aquela solução é ou não a escolha ideal.

13 Os princípios centrais da Column 12

Na seção final do capítulo, Bentley transforma a experiência prática em princípios mais gerais de projeto de software.

13.1 Entender o problema percebido

O programador deve conversar com o usuário e compreender o contexto real. A primeira versão de um problema quase sempre vem misturada com uma sugestão de solução. Essa sugestão merece atenção, mas não deve limitar o pensamento.

13.2 Especificar um problema abstrato

Uma especificação limpa e precisa ajuda a resolver o problema atual e ainda permite reaproveitar a solução em outros contextos. Esse é um dos grandes ganhos de pensar em termos abstratos.

13.3 Explorar o espaço de projeto

Muitos programadores pensam por pouco tempo e codificam por tempo demais. Bentley sugere o oposto: pensar mais antes de implementar. Pseudocódigo, estruturas de dados abstratas e conhecimento prévio da literatura ajudam muito nessa etapa.

13.4 Implementar uma boa solução

Depois da análise, a implementação deve ser direta. O objetivo não é criar código artificialmente complexo, mas expressar a solução escolhida da forma mais simples e forte

possível.

13.5 Olhar para trás

Ao final, é importante refletir sobre o que foi aprendido, como a solução pode ser melhorada e quais ideias podem ser reutilizadas no futuro. Essa retrospectiva faz parte do amadurecimento técnico.

14 O que este capítulo ensina de verdade

Embora a Column 12 trate de amostragem aleatória, o ensino mais profundo do capítulo não é estatístico nem apenas algorítmico. O capítulo ensina, na prática, que:

- a formulação correta do problema influencia diretamente a qualidade da solução;
- uma boa abstração simplifica a implementação;
- comparar alternativas é parte essencial do trabalho do programador;
- simplicidade não é pobreza técnica, mas sinal de maturidade de projeto;
- conhecer literatura e técnicas clássicas acelera muito a resolução de problemas.

15 Resumo final para revisão rápida

Se fosse necessário condensar a Column 12 em poucas ideias para revisão antes de uma prova ou apresentação, o núcleo do capítulo seria este:

1. o problema original foi reformulado porque exigia entrada demais para pouco aproveitamento;
2. o problema abstrato correto é gerar m inteiros distintos e ordenados em $0..n-1$;
3. o algoritmo de Knuth percorre os números em ordem e escolhe cada um com probabilidade ajustada;
4. essa estratégia garante exatamente m escolhas e mantém a saída ordenada;
5. a solução principal usa pouca memória, mas pode ser lenta para n muito grande;
6. outras abordagens, como *set* e embaralhamento, podem ser melhores em cenários específicos;
7. o maior ensinamento do capítulo é sobre **processo de projeto**, e não apenas sobre código.

16 Conclusão

A Column 12 é um excelente exemplo de como um problema pequeno pode conter grandes lições de programação. O capítulo começa com uma tarefa de escritório aparentemente

simples e termina apresentando uma aula completa sobre definição de problemas, abstração, análise de alternativas e implementação elegante.

O algoritmo de Knuth é importante, mas o verdadeiro valor do capítulo está em mostrar como um bom programador pensa. Antes de escrever código, ele questiona a formulação inicial, busca uma especificação melhor, compara soluções e escolhe conscientemente a alternativa que melhor atende às restrições do caso.

Por isso, este capítulo é extremamente útil como material de estudo futuro: ele ajuda não apenas a resolver um exercício específico, mas a desenvolver uma forma mais madura de encarar problemas de programação.