A geometria dos restos: Uma relação entre os números primos e as funções hash

Esteban Yepez (a116272)

Junho 2026

Resumo

Este estudo analisa a relação entre a ciência da computação e a teoria de números, destacando o papel do algoritmo da divisão na eficácia das estruturas de dados. É apresentada uma justificação matemática para explicar por que motivo a utilização de números primos como divisores nas funções hash tende a reduzir a formação de padrões e a melhorar a distribuição dos restos para muitas classes de dados.

1 Introdução

Na era atual da ciência da computação, são necessários algoritmos que garantam uma eficiência excecional para lidar com enormes volumes de dados. A tabela hash (*Hash table*), capaz de localizar, adicionar e remover elementos num tempo de computação ótimo de $O(1)$[1], é uma das estruturas mais básicas para atingir este objetivo.

O algoritmo da divisão, um dos teoremas mais antigos e elegantes da aritmética, ajuda-nos a resolver o problema da eficiência sem precisar de uma arquitetura de software muito complexa. Popularizado na década de 1950, o método *hash by division* converte uma determinada chave num endereço de memória específico. No entanto, a eficácia deste método não é incondicional pois depende essencialmente das características aritméticas do divisor.

Esta investigação demonstra como a utilização de números primos funciona como uma barreira estrutural contra o agrupamento de dados e como a forma dos restos produzidos pela operação modular é significativamente afetada se o divisor partilhar fatores comuns com a chave inserida.

2 O que é uma função hash

Em sentido estrito, uma função hash (também conhecida como função de mapeamento) é um método matemático que mapeia dados de comprimento variável, tais como uma palavra-passe ou um texto (uma *string*), para um valor inteiro de comprimento fixo. Este resultado é frequentemente designado por *hash code*, *hash value*, ou *hash*.

A criação de uma 'impressão digital' numérica dos dados originais é o principal objetivo deste processo. Este número é utilizado diretamente como índice em tabelas hash para aceder rapidamente à região da memória onde os dados estão ou serão armazenados.

[1] A notação $O(1)$, significa tempo constante e indica que o tempo de execução de um algoritmo permanece igual, independentemente do tamanho de entrada, habitualmente denotado por n.

2.1 A estrutura de uma Tabela Hash

Essencialmente, utiliza-se uma matriz contígua na memória, com um tamanho especificado, denotado por m, para implementar uma tabela hash. Com índices que variam de 0 a $m-1$, cada posição nesta matriz funciona como um contentor (comumente chamado de *bucket* ou *slot*) onde os dados propriamente ditos são armazenados.

O processo de armazenamento ou pesquisa ocorre em duas fases fundamentais:

1. **Conversão e dispersão:** A chave do elemento (por exemplo, um nome ou um texto) é passada à função hash, que a transforma num número inteiro bruto.

2. **Mapeamento de índice:** Esse número inteiro é então mapeado para caber perfeitamente dentro dos limites do *array* (entre 0 e $m-1$). O resultado final dita o índice exato de memória onde o dado será lido ou escrito.

2.2 Implementação prática

Para ilustrar a primeira fase deste processo (a conversão de dados textuais para o domínio dos inteiros), analisemos o mapeamento de *strings*. O trecho de código seguinte demonstra a estrutura clássica de uma função de dispersão baseada na soma de caracteres, uma abordagem fundamental no desenho de algoritmos:

```c
#include <stdio.h>

// Funcao que converte a chave num indice usando o metodo da divisao
int calcular_hash(const char *chave, int m) {
    int soma_ascii = 0;
    int i = 0;

    // Iteracao sobre cada caractere para acumular o seu valor numerico
    while (chave[i] != '\0') {
        soma_ascii += (int)chave[i];
        i++;
    }

    // Retorna a soma bruta (Ainda sem o mapeamento de indice)
    return soma_ascii;
}
```

Listing 1: Função clássica de dispersão para *strings* através da soma dos valores ASCII.

O Código 1 ilustra uma transformação direta[2]. Se inserirmos a palavra `'Mesa'`, a função somará os seus valores ASCII ($77 + 101 + 115 + 97$) e devolverá o inteiro 390.

Neste ponto, deparamo-nos com um problema: se a nossa tabela hash na memória RAM possuir um limite estrito $m = 8$ posições (índices de 0 a 7) e tentarmos aceder ao índice 390 num *array* de tamanho 8 resultará num erro fatal de memória (*Segmentation Fault* ou *Buffer Overflow*). Tornar a tabela tão grande quanto a maior soma possível é inviável e constitui um desperdício massivo de recursos.

Como podemos, então, confinar um universo numérico infinito a um espaço de memória finito e seguro?

[2]Esta função é utilizada para exemplificar mas na realidade, funções hash reais utilizam transformações mais sofisticadas para reduzir colisões.

3 O algoritmo da divisão

O Teorema da Divisão de Euclides (também conhecido como Algoritmo da Divisão), um conceito fundamental da teoria de números, pode ser aplicado para resolver o problema do confinamento de memória abordado na secção anterior, sem necessidade de uma arquitetura de software sofisticada. De acordo com este teorema, quaisquer dois números inteiros a e b (onde $b > 0$) têm números inteiros únicos q (quociente) e r (resto) tais que:

$$a = b \cdot q + r, \quad \text{onde } 0 \leq r < b$$

Quando este teorema é aplicado à computação, o divisor b corresponde ao tamanho finito da tabela m, e a variável a reflete o valor numérico bruto da chave (por exemplo, o valor 390). O índice de armazenamento exato é r, o resto da divisão. A definição da função hash completa $h(k)$ é:

$$h(k) = k \pmod{m}$$

Adicionámos o operador módulo (`%`), que realiza precisamente este cálculo aritmético e devolve o resto da divisão (neste caso, o índice exato onde o dado será armazenado), a fim de implementar esta correção no nosso código anterior:

```c
#include <stdio.h>

// Funcao corrigida que confina o indice ao tamanho m da tabela
int calcular_hash_seguro(const char *chave, int m) {
    int soma_ascii = 0;
    int i = 0;

    // Iteracao sobre cada caractere
    while (chave[i] != '\0') {
        soma_ascii += (int)chave[i];
        i++;
    }

    // Aplicacao do Algoritmo da Divisao para evitar Segmentation Fault
    return soma_ascii % m;
}
```

Listing 2: Função de dispersão corrigida com a aplicação do Algoritmo da Divisão.

Assim o índice resultante estará sempre compreendido entre 0 e $m - 1$. Voltemos ao nosso exemplo e insiramos as chaves que produziram os valores 390, 11 e 78 na tabela de tamanho $m = 8$:

- $h(390) = 390 \pmod{8} = 6$
- $h(11) = 11 \pmod{8} = 3$
- $h(78) = 78 \pmod{8} = 6$

O mapeamento resultante pode ser observado diretamente na representação da memória abaixo:

Índice	0	1	2	3	4	5	6	7
Memória	*Vazio*	*Vazio*	*Vazio*	**11**	*Vazio*	*Vazio*	**390, 78**	*Vazio*

Tabela 1: Representação do *array* linear de memória ($m = 8$), ilustrando uma colisão no índice 6.

O operador módulo funciona como um mecanismo geométrico circular, tal como se pode ver no Código 2 e na Tabela 1. O Princípio das Caixas de Pombos[3], no entanto, impõe uma dificuldade probabilística a este método: inúmeras chaves produzirão inevitavelmente o mesmo resto, uma vez que o conjunto de chaves originais é significativamente maior do que m. No nosso cenário, ocorre uma **colisão**, uma vez que ambos os valores brutos 390 e 78 reivindicam a posição 6. A seleção de m está indissociavelmente ligada à mitigação desta ocorrência, o que nos leva ao papel crucial dos números primos.

4 Escolha do divisor e números primos

O operador módulo limita os dados, mas não impede que várias chaves sejam mapeadas para o mesmo índice, tal como foi referido na secção anterior. De acordo com a intuição humana, as colisões deveriam ser pouco frequentes numa tabela praticamente vazia. No entanto, num fenómeno denominado **Paradoxo do Aniversário**, a probabilidade matemática sugere o contrário.

4.1 O paradoxo do aniversário e as colisões

De acordo com o Paradoxo do Aniversário, há uma probabilidade de 50 % de que, num grupo aleatório de apenas 23 pessoas, pelo menos duas tenham o mesmo dia de aniversário. Tendo em conta que um ano tem 365 dias, isto pode parecer ilógico, mas a matemática subjacente explica o fenómeno.

1. O número de pares: O nosso cérebro tende a comparar a nossa própria data com a dos restantes, mas o paradoxo avalia todas as combinações possíveis entre todos os indivíduos. Num grupo de 23 pessoas, o número de pares únicos gerados é dado pelo coeficiente binomial:

$$\binom{23}{2} = \frac{23!}{2!(23-2)!} = \frac{23 \times 22}{2} = 253 \text{ pares únicos para comparar} \tag{1}$$

2. O cálculo das probabilidades: A forma mais simples de calcular uma colisão é calcular primeiro a probabilidade de *nenhuma* colisão ocorrer. A primeira pessoa tem um aniversário único garantido (365/365). A segunda tem 364/365 de probabilidade de não coincidir com a primeira, a terceira 363/365, e assim sucessivamente.

$$P(\text{sem colisão}) = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \cdots \times \frac{343}{365} \approx 49.3\% \tag{2}$$

3. O resultado: Subtraindo a probabilidade de não haver colisões a 100%, obtemos a probabilidade de ocorrer **pelo menos uma** coincidência: $100\% - 49.3\% = 50.7\%$.

Esta probabilidade escala de forma drástica à medida que mais elementos entram no sistema:

- **23 elementos:** $\sim 50\%$ de probabilidade de colisão.
- **50 elementos:** $\sim 97\%$ de probabilidade de colisão.
- **75 elementos:** $\sim 99\%$ de probabilidade de colisão.

[3]O Princípio da Casa dos Pombos (ou Princípio das Gavetas de Dirichlet) afirma que se n elementos forem distribuídos por m recipientes, com $n > m$, então pelo menos um recipiente tem de conter obrigatoriamente mais do que um elemento.

- **366 elementos:** 100% de probabilidade (uma garantia matemática absoluta suportada pelo Princípio da Casa dos Pombos).

Esta aritmética constitui a base dos *birthday attacks* (ataques do aniversário), não sendo apenas uma curiosidade estatística. Esta abordagem probabilística é utilizada por *hackers* na criptografia para detetar colisões em algoritmos de segurança ou para decifrar palavras-passe mais rapidamente do que através de técnicas de força bruta.

No contexto do nosso estudo, isto significa que ocorrerão colisões em grande escala numa tabela hash muito antes de o *array* ficar cheio. Uma vez que estas são estatisticamente inevitáveis, a única opção do engenheiro é garantir que a distribuição geométrica dos dados minimize o impacto, o que nos leva à escolha do divisor.

4.2 A falha geométrica dos números compostos

No mundo da informática, os dados de entrada raramente são aleatórios. Frequentemente, seguem sequências aritméticas ou padrões (por exemplo, endereços de memória que aumentam em 4 bytes).

Suponhamos que escolhemos um divisor composto, tal como $m = 8$ (uma potência de 2), e que as nossas chaves seguem o padrão $k \in \{4, 8, 12, 16, 20\}$. Os restos resultantes têm a seguinte distribuição:

- $4 \pmod 8 = 4$
- $8 \pmod 8 = 0$
- $12 \pmod 8 = 4$
- $16 \pmod 8 = 0$
- $20 \pmod 8 = 4$

O resultado é desastroso para a estrutura de dados. Os restos entram em «ressonância» geométrica porque o divisor (8) e o padrão de dados (4) têm fatores comuns (cujo máximo divisor comum é $\text{mdc}(4, 8) = 4 \neq 1$). Apenas nos índices 0 e 4 é que as chaves ficam presas num subgrupo cíclico, deixando 75% da tabela desocupada, o que pode degradar o tempo de acesso para $O(n)$[4] no pior caso.

4.3 A solução: Os números primos

Para quebrar este padrão, a Teoria de Números oferece uma solução: escolher como divisor m um número primo.

Se alterarmos o tamanho da nossa tabela para o número primo mais próximo, $m = 7$, e inserirmos exatamente a mesma sequência de dados $(4, 8, 12, 16, 20)$, a nova distribuição será:

- $4 \pmod 7 = 4$
- $8 \pmod 7 = 1$

[4]Ao contrário de $O(1)$, a notação $O(n)$ indica que o tempo de execução cresce linearmente com o número de elementos n armazenados na tabela.

- $12 \pmod 7 = 5$
- $16 \pmod 7 = 2$
- $20 \pmod 7 = 6$

Seja a o padrão de espaçamento entre as chaves inseridas (o 'salto', que no nosso caso é 4) e m o tamanho da tabela. A sequência gerada pelos dados na memória assume a forma:

$$0, a, 2a, 3a, \ldots, (m-1)a \pmod m \tag{3}$$

A matemática estabelece que se a e m forem coprimos, ou seja, se o seu máximo divisor comum (mdc) for $\text{mdc}(a, m) = 1$, esta sequência percorrerá todos os m restos possíveis exatamente uma vez antes de se repetir, formando uma permutação completa do espaço de memória.

Demonstração: Suponhamos, por absurdo, que dois múltiplos distintos da sequência resultam no mesmo índice na tabela (uma colisão prematura). Sejam eles $k_1 a$ e $k_2 a$, com $0 \leq k_1 < k_2 < m$. Teríamos então:

$$k_1 a \equiv k_2 a \pmod m \tag{4}$$

Isto implica que m divide a diferença, ou seja, $m \mid a(k_2 - k_1)$. Como assumimos que $\text{mdc}(a, m) = 1$, m não partilha fatores com a. Logo, pelo Lema de Euclides[5], m tem obrigatoriamente de dividir $(k_2 - k_1)$.

No entanto, como k_1 e k_2 são ambos estritamente menores que m, a sua diferença $(k_2 - k_1)$ é um número positivo estritamente menor que m. É matematicamente impossível que um número inteiro divida um número positivo menor que ele próprio.

Chegamos a uma contradição. Logo, todos os elementos da sequência produzem restos distintos.

Ao escolher um número primo para m, garantimos que $\text{mdc}(a, m) = 1$ para muitos padrões de dados a (exceto quando a é um múltiplo de m). Fica assim demonstrado que, para sequências aritméticas cujo passo seja coprimo com m, os restos percorrem todos os índices possíveis antes de se repetirem. Esta propriedade explica por que razão divisores primos tendem a reduzir padrões periódicos e a favorecer uma distribuição mais homogénea dos dados em muitas aplicações práticas.

5 Resolução de colisões

O Paradoxo do Aniversário demonstra que as colisões não podem ser totalmente evitadas, apesar de a utilização de números primos melhorar a distribuição geométrica dos restos. A fim de manter a acessibilidade e a integridade dos dados, os engenheiros de software criaram planos de contingência para lidar com chaves que entram em colisão no mesmo índice.

As duas abordagens algorítmicas fundamentais são:

- **Encadeamento (*Chaining*):** A tabela hash não armazena o dado diretamente no índice, mas sim um ponteiro para uma Lista Ligada (*Linked List*). Se ocorrer uma colisão, o novo elemento é simplesmente adicionado ao nó final da lista pertencente a esse índice.

[5]Lema de Euclides: Sejam $a, b, c \in \mathbb{Z}$, com a e b não simultaneamente nulos. Se $a \mid bc$ e $\text{mdc}(a, b) = 1$, então, $a \mid c$.

- **Endereçamento Aberto (*Open Addressing*):** Se o índice alvo $h(k)$ estiver ocupado, o algoritmo procura sistematicamente a próxima posição vazia no *array* (por exemplo, através de exploração linear ou *Linear Probing*, verificando $h(k) + 1, h(k) + 2$, etc.). Neste cenário, a aritmética modular volta a demonstrar a sua utilidade: o cálculo $(h(k) + i) \pmod{m}$ garante que a procura volta circularmente ao início do *array* caso atinja o limite máximo da memória.

Para evitar a formação de aglomerados de dados e manter a complexidade temporal da pesquisa o mais próxima possível do valor ótimo $O(1)$, ambas as abordagens baseiam-se principalmente numa distribuição inicial uniforme, o que é garantido pela seleção de um divisor primo.

6 Conclusão

Conceitos matemáticos constituem frequentemente a base para a eficácia das estruturas de dados básicas. Um excelente exemplo desta sinergia é o funcionamento de uma tabela de dispersão. O Algoritmo de Divisão fornece um método preciso para armazenar dados infinitos numa região de memória estritamente finita. Além disso, as propriedades especiais dos números primos servem como uma salvaguarda estrutural contra os enviesamentos e as anomalias encontrados nos dados do mundo real.

Por último, os engenheiros e cientistas da computação podem antecipar falhas sistémicas inevitáveis e desenvolver algoritmos fiáveis de resolução de conflitos através da compreensão de regras probabilísticas, em particular o Paradoxo do Aniversário. Ficou demonstrado que a utilização correta da teoria dos números e da matemática discreta é tão importante para a excelência e a otimização do desenvolvimento de software como a própria escrita de código.

Referências

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4th ed.). MIT Press. (Capítulo 11: Hash Tables).

[2] Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley Professional. (Secção 6.4: Hashing).

[3] Jones, G. A., & Jones, J. M. (1998). *Elementary Number Theory* (Springer Undergraduate Mathematics Series). Springer.

[4] Smith, P. M., & Martins, P. M. (2009). *Matemática Discreta* (1ª ed.). Departamento de Matemática da Universidade do Minho.

[5] Yousuf, M. (2024). *Repositório youtube-tutorials (Branch: CD_006_hashing)* [Código fonte]. GitHub. Disponível em: `https://github.com/monisyousuf/youtube-tutorials`

[6] MDN Web Docs (s.d.). *Glossary: Hash function*. Mozilla Foundation. Disponível em: `https://developer.mozilla.org/en-US/docs/Glossary/Hash_function`

[7] *Hash Tables and Hash Functions* [Vídeo]. YouTube. Disponível em: `https://youtu.be/pMM9cIAFAug`

[8] Computerphile (2013). *Hashing Algorithms and Security* [Vídeo]. YouTube. Disponível em: https://youtu.be/b4b8ktEV4Bg

[9] Computerphile (2025). *Hash Collisions & The Birthday Paradox* [Vídeo]. YouTube. Disponível em: https://youtu.be/jsraR-el8_o

[10] Wikimedia Commons (2010). *File:Hash table simchk 999.svg* [Imagem]. Disponível em: https://commons.wikimedia.org/wiki/File:Hash_table_simchk_999.svg