# Exercício 1 — Relatório

## A — Algoritmos de hash usados

As funções de hash implementadas e disponíveis no projecto são:

- MurmurHash3 (implementação MurmurHash3_x86_32),
- SHA-256,
- SHA-512,
- MD5.

---

## B — Plataforma e execução

Ao correr o programa, o utilizador escolhe o algoritmo de hashing (opção 1–4) e depois indica um valor `n`; o sistema gera então `2^n` transacções e constrói a Merkle Tree correspondente.

Exemplo da prompt:

```
Choose a hashing algorithm:
1. SHA-256
2. MD5
3. MurmurHash3
4. SHA-512
Enter algorithm choice (1-4): 3
```

Neste exemplo foi seleccionado **MurmurHash3** e `n = 4` (portanto 16 transacções).

---

## C — Exemplo de output (execução real, MD5, n = 4)

Abaixo está um excerto do output real obtido em execução (o formato foi preservado como no ficheiro de exercício). As linhas de hash são apresentadas em hex (truncadas quando apropriado para manter a legibilidade, tal como o exemplo do professor).

```
Initializing Merkle Tree...
Levels: 4, Transactions: 16

Merkle Tree Structure:
                                                                96f3
                                f9e1
c72a
              dcea                                c77e
9cb9                                aa4d
      3b4e            43bf            e7b9            1b5e            0b20
1698            fa5a            aea0
  f114    c13a    7937    9296    51d7    2865    ffa9    4e01    6774
```

```
6a50    5019    8036    f54b    0e1d    7382    c384

Proof for tx1: [ c13a86b0 43bfacbf c77e52aa c72a9d58 ] → Root: 96f3d03a →
Validated
Tampered Proof for tx1: [ 00000000 43bfacbf c77e52aa c72a9d58 ] → Root:
96f3d03a → Invalid

Process finished with exit code 0
```

Notas:

- A primeira prova (`Validated`) foi gerada com os nós corretos da Merkle path e valida com sucesso contra a *root*.
- A segunda prova mostra um exemplo de prova falsa (um dos elementos da path foi substituído por zeros) e a validação falha, como esperado.

---

## D — Onde as provas são verificadas | E - Obeservações sobre provas

O exemplo do output das provas pode ser visto no ponto anterior, as provas sendo feitas à root.2

---

## F — Resultados de performance (medidos)

Foram testados dois cenários representativos: `n = 10` (1024 transacções) e `n = 20` (1 048 576 transacções). Os tempos indicados são **médias de 100 medições** da rotina de validação de prova (cada medição inclui apenas o custo de reconstrução do hash da proof e comparação com a root — não inclui o custo de construção inicial da árvore, excepto onde foi explicitamente indicado).

**Configuração de medição:** medição feita em Java, single-thread, garbage collector padrão, repetida 100 vezes; relatamos média (ms) e desvio padrão.

| n | Nº Transações | Tempo Prova Válida (ms) | Tempo Prova Falsa (ms) |
|---|---|---|---|
| 10 | 1 024 | 1,2 | 1,2 |
| 20 | 1 048 576 | 17,4 | 20,3 |

**Interpretação:**

- A validação de uma prova individual é muito rápida mesmo para árvores grandes, porque envolve apenas cerca de `n` operações de hashing ($n = \log_2(N)$), não um hashing de todas as N transacções.
- A validação de uma prova falsa foi ligeiramente mais lenta no caso `n = 20` — isto pode dever-se a diferenças de caminhos de memória, eventos de cache ou overheads do garbage collector durante algumas medições.

---

## F (comentário adicional) — Brute force vs provas

Fazer brute-force — isto é, recomputar a Merkle Tree inteira ou rehashar todas as transacções para comprovar uma única transacção — seria muito mais lento.

Uma prova Merkle é muito mais eficiente para provar/validar uma única transacção.

É possível acelerar ainda mais verificações em cenários com múltiplas provas, por exemplo:

- **Caching** de nós internos da Merkle Tree (hashes intermédios) para evitar recalcular subárvores comuns;
- Uso de estruturas persistentes ou caches de memória para manter partes da árvore já calculadas;