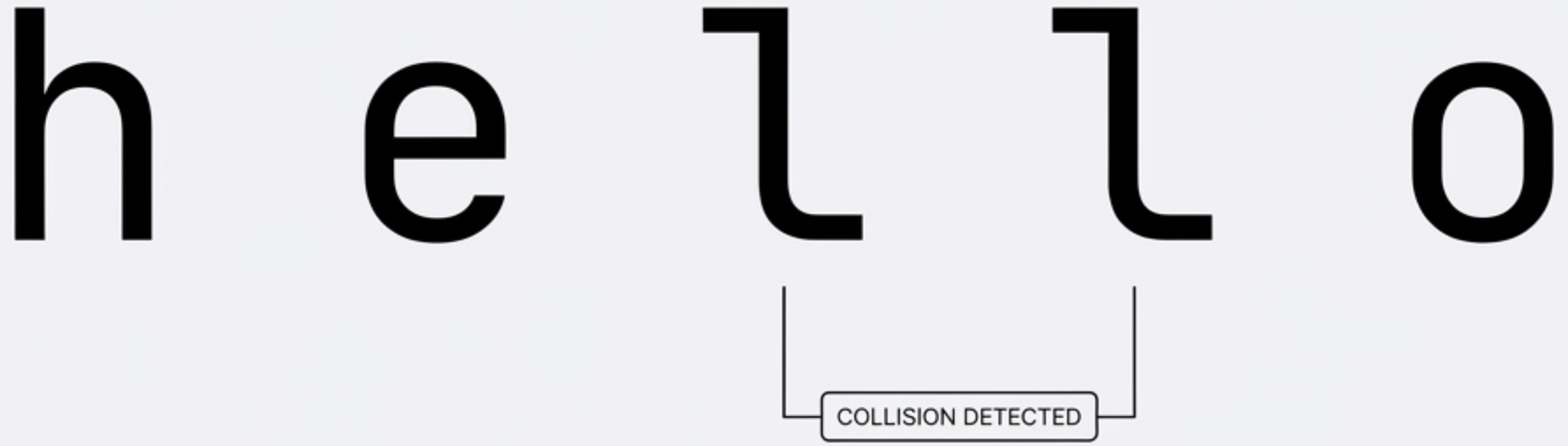
Duplicate Character Detection

Four Algorithmic Approaches in Python



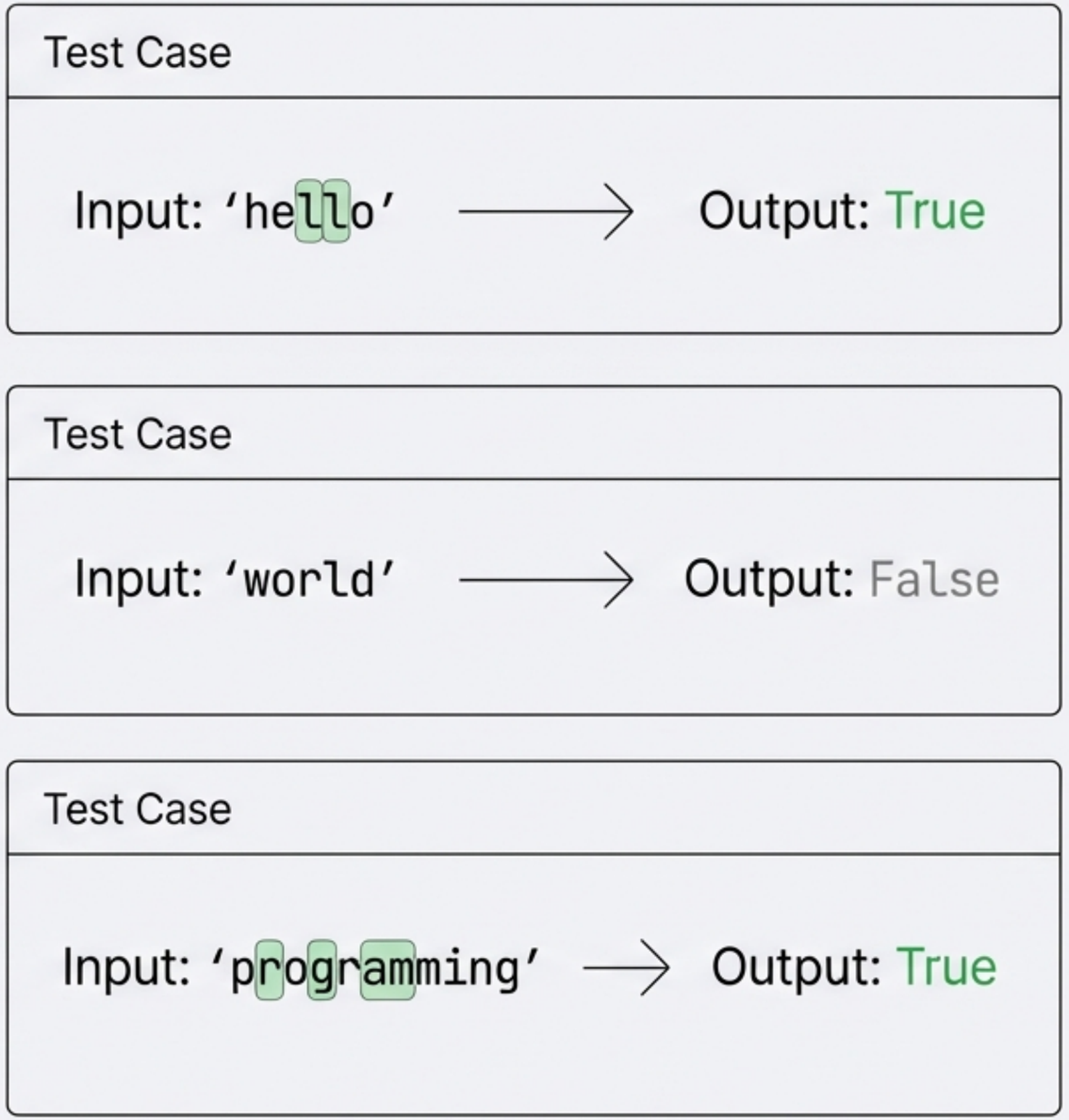
FROM BRUTE FORCE TO BIT MANIPULATION

The Challenge: Efficiency at Scale

The problem appears trivial, but the solution defines system performance.

Input: A `string s`.
Output: Boolean `True` if any character appears >1 times.

How do we detect the duplicate? There isn't just one way—there are four, and the right choice depends on your specific specific constraints.



1. The Hashmap Approach (Standard)

The General Purpose Winner

```python
def has_duplicate_hashmap(s: str) -> bool:
    char_count = {}
    for char in s:
        if char in char_count:
            return True
        char_count[char] = 1
    return False
```

TIME COMPLEXITY

O(n)

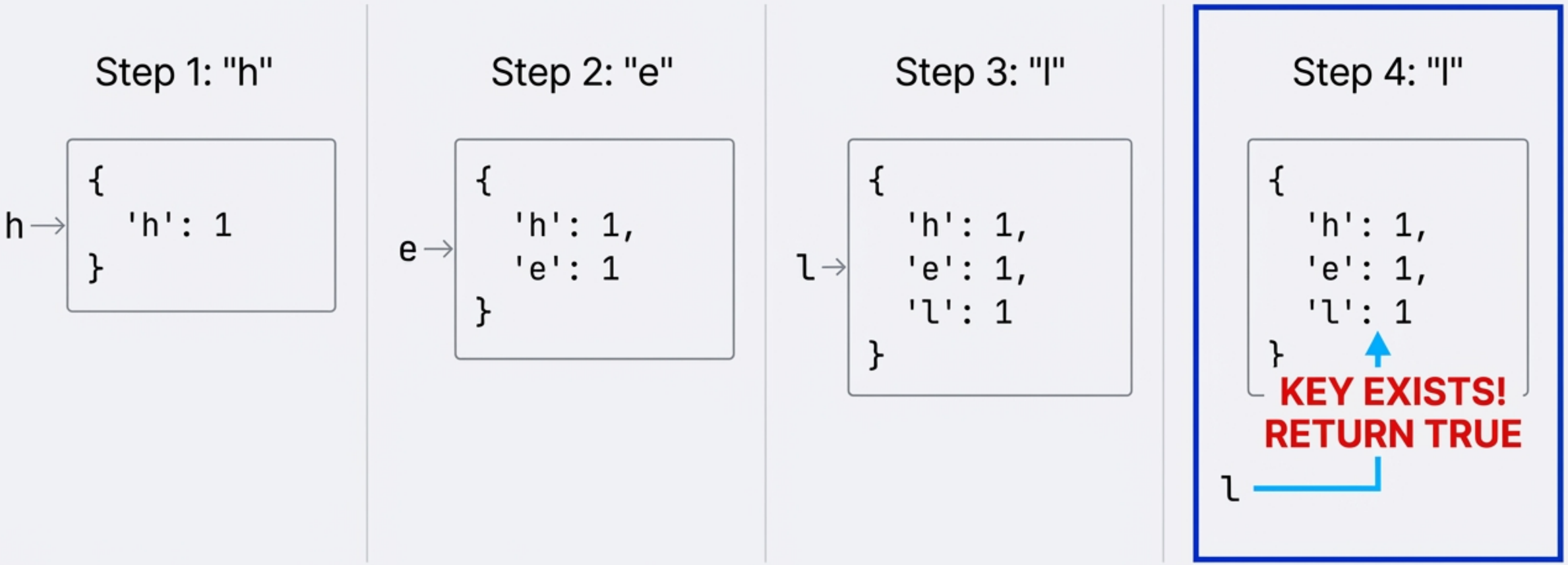
Linear time. We iterate through the string **exactly once.**

SPACE COMPLEXITY

O(k)

Linear space. We store up to k unique characters.

Execution Trace: "hello"



2. The Nested Loop Approach (Brute Force)

Intuitive, but performance prohibitive.

```
def has_duplicate_nested_loop(s: str) -> bool:
    n = len(s)
    for i in range(n):
        for j in range(i + 1, n):
            if s[i] == s[j]:
                return True
    return False
```

TIME COMPLEXITY

$O(n^2)$

Quadratic time. Comparisons grow exponentially.

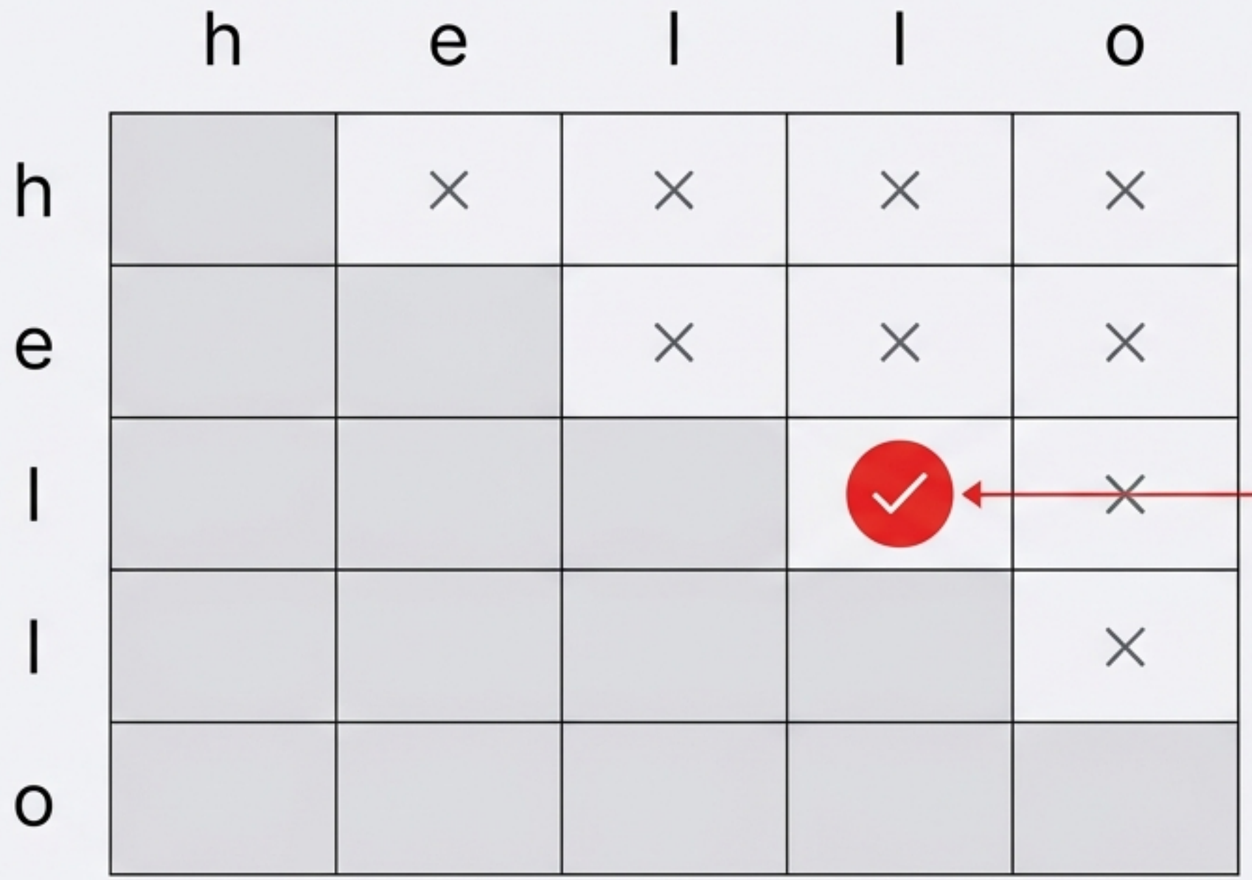
SPACE COMPLEXITY

O(1)

Constant space. No extra memory required.

The Cost of Quadratic Time

Visualising the comparison matrix for input "hello"



For small strings, this is negligible. For large inputs, the number of checks explodes.

3. The Sorting Approach (Pre-processing)

Restructuring the data to simplify detection.

```
def has_duplicate_sorting(s: str) -> bool:
    sorted_chars = sorted(s)
    for i in range(len(sorted_chars) - 1):
        if sorted_chars[i] == sorted_chars[i + 1]:
            return True
    return False
```

TIME COMPLEXITY

O(n log n)

Dominated by the sort operation.

SPACE COMPLEXITY

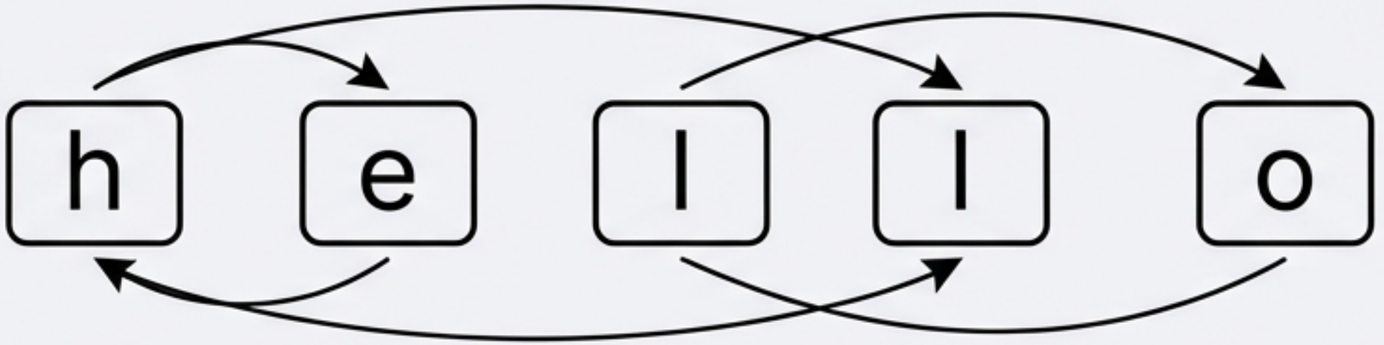
O(n)

Requires creating a new sorted list.

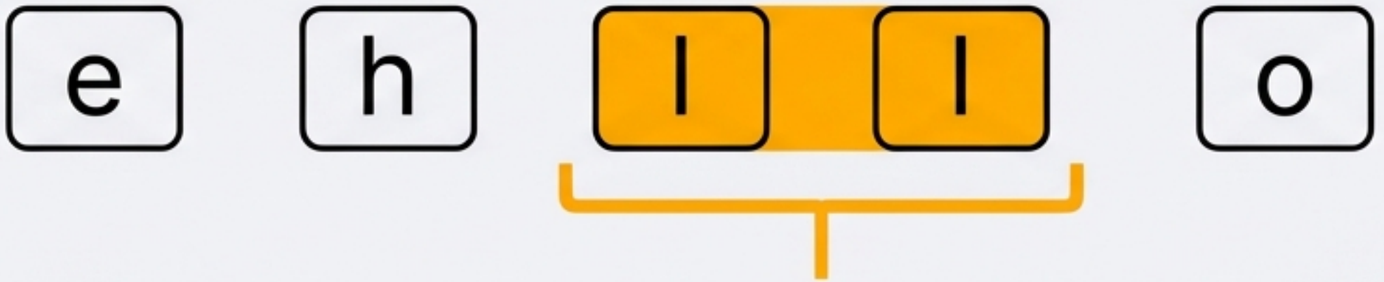
Bringing Duplicates Together

Visualising the transformation of the string

BEFORE SORT



AFTER SORT



COMPARISON LOGIC

```
[ e ] == [ h ] ? → No
[ h ] == [ l ] ? → No
[ l ] == [ l ] ? → YES ✓
```

4. The Bitmask Approach (The Specialist)

High performance for strictly lowercase inputs.

```
def has_duplicate_bitmask(s: str) -> bool:
    bitmask = 0
    for char in s:
        position = ord(char) - ord('a')
        if bitmask & (1 << position):
            return True
        bitmask |= (1 << position)
    return False
```

TIME COMPLEXITY

O(n)

Linear time.

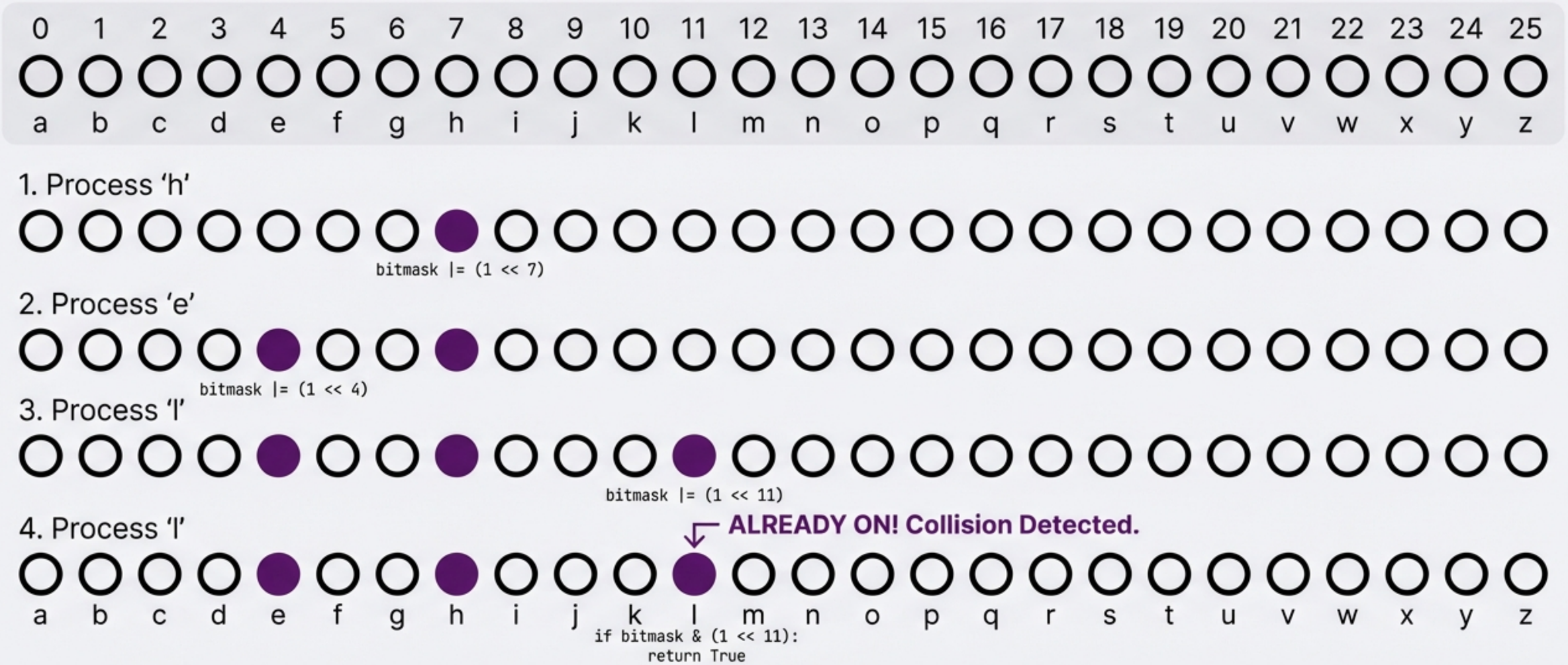
SPACE COMPLEXITY

O(1)

Constant space. Uses a single integer.

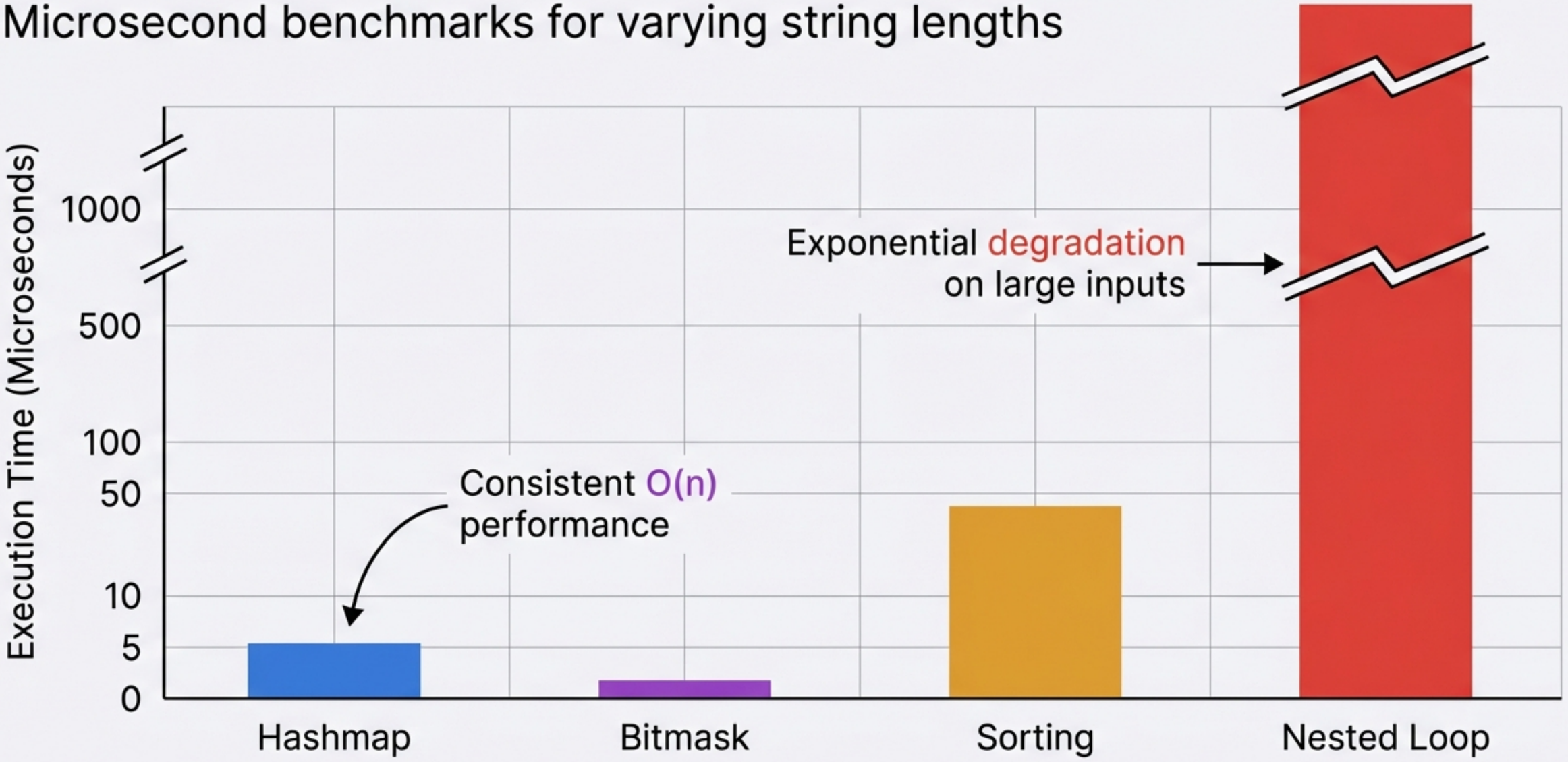
Tracking with Bits

Visualising the integer bits as a row of switches.



Performance Showdown

Microsecond benchmarks for varying string lengths



The Algorithmic Toolbox

Approach	Time	Space	Best Use Case
Hashmap	$O(n)$	$O(k)$	General purpose standard.
Nested Loop	$O(n^2)$	$O(1)$	Educational / Small strings only.
Sorting	$O(n \log n)$	$O(1)$	When output must be sorted anyway.
Bitmask	$O(n)$	$O(1)$	Memory constrained + Lowercase only.

Code Cheat Sheet

Hashmap

```
seen = set()
for c in s:
    if c in seen: return True
    seen.add(c)
```

Nested Loop

```
for i in range(len(s)):
    for j in range(i+1, len(s)):
        if s[i] == s[j]: return True
```

Sorting

```
s = sorted(s)
for i in range(len(s)-1):
    if s[i] == s[i+1]: return True
```

Bitmask

```
mask = 0
for c in s:
    pos = ord(c) - 97
    if mask & (1<<pos): return True
    mask |= (1<<pos)
```

The Final Verdict

For 99% of cases, use the Hashmap.

It is readable, fast, and robust across all character types.

Use Bitmask only for extreme constraints.

If memory is tight and inputs are guaranteed lowercase.

Algorithm choice is not just about speed—it is about matching the tool to the data.