



EVALUATION OF THE `character_vowel_check()` ALGORITHM VERSUS SET MEMBERSHIP TESTING FOR VOWEL IDENTIFICATION. AN EXAMINATION OF TIME COMPLEXITY AND PRACTICAL RUNTIME BEHAVIOUR OF BOTH APPROACHES - A CASE STUDY

Cadet D Nivaashan , Cadet E Egan

Class- XII 2026-27, Sainik School Amaravathinagar

Post: Amaravathinagar, Udumalpet Taluka, Tirupur Dt, Tamilnadu State

ABSTRACT

Deciding whether a single character belongs to a small, fixed category is one of the most frequently repeated operations in text processing, yet the implementation choice behind it is rarely examined closely. Vowel identification is a convenient, self-contained example of this larger problem: a program must decide, for every character it reads, whether that character belongs to the set {a, e, i, o, u}.

This paper studies two ways of performing that check. The first, `character_vowel_check()`, uses a sequential chain of `if/elif` comparisons. The second uses set membership testing, in which the character is looked up in a hash-based set through Python's ``in`` operator. Both are analysed theoretically through Big-O time and space complexity and then empirically, through direct benchmarking across string lengths ranging from one thousand to one million characters.

A second experiment extends the comparison beyond the fixed five-vowel case by scaling the size of the target category itself, from five members up to three hundred and twenty, to test whether the two approaches diverge further as the category grows. The results show that although both implementations share the same asymptotic order for this task, the constant-factor difference between them is large enough to matter in practice, and it grows sharply once the category is no longer as small as the vowel set.

KEYWORDS: Character Vowel Check (CVC), Set Membership Test (SMT), Hash Table (HT), Time Complexity (TC), Linear Scan (LS), Big-O Notation (BO).

1. INTRODUCTION

Character classification is a small operation, but it sits underneath a great deal of everyday software: a password validator checking for a digit, a syntax highlighter checking for a bracket, a tokenizer checking for whitespace, and a vowel counter checking for a, e, i, o, or u. Because the operation is so short, programmers rarely stop to compare implementation choices for it; the decision is usually made out of habit rather than measurement.

Two habits are common. The first is to write the check out explicitly, comparing the character against each member of the category in turn using `if` and `elif`. This is how many beginners are taught to write a vowel check, and it has the advantage of being immediately readable even to someone who has never seen the code before. The second habit, more common among experienced Python programmers, is to place the category members in a set and test membership with the `in` operator, relying on the interpreter's hash table implementation to do the searching.

Both approaches return the same answer for the same input, so correctness is not in question. What is in question is efficiency, and specifically whether the commonly repeated claim that “set lookup is $O(1)$ and therefore faster” actually holds up once real numbers are measured, and how much it matters for a category as small as five vowels. This paper examines that question directly, first by deriving the time and space complexity of both approaches formally, then by benchmarking real Python implementations of each, and finally by generalising the comparison to categories larger than five

members, since a classification routine written for vowels today is often extended tomorrow to cover accented letters, digits, or punctuation.

2. RELATED WORK

The formal treatment of asymptotic notation used throughout this paper follows the standard definitions of Big-O, Big-Omega and Big-Theta as bounds on an algorithm's growth rate with respect to input size [1][2]. These notations are well established for comparing algorithms whose costs scale with the number of elements being processed, and they are the basis for the complexity derivations in Sections 10 and 11.

Hash-based containers have been analysed extensively for large collections. The official CPython complexity reference documents membership testing on a set as an average-case $O(1)$ operation, with the caveat that this assumes the hash function distributes keys with few collisions, and that constant factors, not just algorithmic order, materially affect how quickly a real program finishes [3]. Independent write-ups of CPython's internal set implementation reach the same conclusion, noting that lookup, insertion, and deletion are each $O(1)$ on average but can degrade to $O(n)$ in a pathological, collision-heavy case [4][5].

Sequential comparison, by contrast, is covered under the well-known analysis of linear search: $O(1)$ in the best case, when the very first comparison succeeds, and $O(n)$ in the worst and average cases, where n is the number of items being compared against [6][7]. What is comparatively under-examined in this literature is the case where n , or in this paper's



terms k , is very small and fixed — five or ten items, not thousands. Most linear-search analysis assumes a search space large enough for the asymptotic terms to dominate; this paper instead asks what happens at the small end of that scale, where constant factors are not a footnote but the whole story, and how quickly that changes as the category grows.

3. METHODOLOGY

Two functions were implemented in Python. `character_vowel_check(ch)` compares the input character against 'a', 'e', 'i', 'o', 'u' and their uppercase equivalents using a chain of if/elif statements, returning True on the first match and False if the chain is exhausted. The set-based alternative, referred to here as `vowel_check_set(ch)`, tests membership of the character in a precomputed frozenset containing the same ten characters. Both functions were wrapped in a counting loop, `count_vowels_ifelif(s)` and `count_vowels_set(s)`, that iterates over an input string s and tallies how many characters satisfy the check, so that the per-character cost could be measured across realistic string lengths rather than in isolation.

Test strings were generated using Python's random module with a fixed seed (42) so that the experiment is reproducible, drawing uniformly from the 52 upper- and lower-case ASCII letters. Seven string lengths were tested: 1,000, 5,000, 10,000, 50,000, 100,000, 500,000 and 1,000,000 characters. For each length, both counting functions were run seven times and the median execution time, in milliseconds, was recorded using `time.perf_counter()`; the median rather than the mean was used deliberately, to reduce the influence of any single anomalously slow run caused by operating-system scheduling rather than the algorithm itself. Before timing began, both functions were confirmed to return an identical vowel count for every test string, ruling out a correctness difference as an explanation for any timing difference. All measurements were taken on CPython 3.12.3 running on a 64-bit Linux host.

A second experiment was designed to move beyond the fixed five-vowel case. Here, the category being tested against was scaled from 5 up to 320 members, drawn from the printable ASCII range, while the input string length was held constant at 200,000 characters. The if/elif chain was represented, for this experiment, as membership testing against a tuple rather than writing out hundreds of elif branches by hand; in CPython, ``ch in a_tuple`` and an equivalent chain of ``ch == x or ch == y or ...`` compile to the same underlying pattern of sequential equality comparisons, so the tuple form is a faithful and far more practical stand-in for a long if/elif chain. This was compared against membership testing on a frozenset built from the same characters. The purpose of this second experiment is to show what the vowel-specific case, with its small and unchanging category of five or ten letters, does not reveal on its own: how the two approaches scale once the category grows past a handful of members.

4. ALGORITHM: `character_vowel_check()`

STEP 01: START

STEP 02: RECEIVE A SINGLE CHARACTER, ch , AS INPUT.

STEP 03: IF ch EQUALS 'a' OR 'A', RETURN TRUE.

STEP 04: ELSE IF ch EQUALS 'e' OR 'E', RETURN TRUE.

STEP 05: ELSE IF ch EQUALS 'i' OR 'I', RETURN TRUE.

STEP 06: ELSE IF ch EQUALS 'o' OR 'O', RETURN TRUE.

STEP 07: ELSE IF ch EQUALS 'u' OR 'U', RETURN TRUE.

STEP 08: ELSE RETURN FALSE.

STEP 09: STOP

5. ALGORITHM: SET MEMBERSHIP CHECK

STEP 01: START

STEP 02: BEFORE ANY CHARACTER IS PROCESSED, BUILD A SET, VOWELS, CONTAINING

'a','e','i','o','u','A','E','I','O','U'.

STEP 03: RECEIVE A SINGLE CHARACTER, ch , AS INPUT.

STEP 04: COMPUTE THE HASH OF ch AND LOCATE ITS BUCKET IN VOWELS.

STEP 05: IF ch IS FOUND IN THAT BUCKET, RETURN TRUE.

STEP 06: ELSE RETURN FALSE.

STEP 07: STOP

6. PROGRAM: PYTHON IMPLEMENTATION

`character_vowel_check()` -- explicit comparison chain

def `character_vowel_check(ch)`:

if $ch == 'a'$ or $ch == 'A'$:

return True

elif $ch == 'e'$ or $ch == 'E'$:

return True

elif $ch == 'i'$ or $ch == 'I'$:

return True

elif $ch == 'o'$ or $ch == 'O'$:

return True

elif $ch == 'u'$ or $ch == 'U'$:

return True

else:

return False

set membership test -- hash-based lookup

VOWELS = frozenset('aeiouAEIOU')

def `vowel_check_set(ch)`:

return ch in VOWELS

counting wrappers used for benchmarking

def `count_vowels_ifelif(s)`:

count = 0

for ch in s :

if `character_vowel_check(ch)`:

count += 1

return count

def `count_vowels_set(s)`:

count = 0

for ch in s :

if `vowel_check_set(ch)`:

count += 1

return count

Both counting functions traverse the input string exactly once, and both call their respective character-level check exactly



n times for a string of length n. The two implementations therefore differ only in the constant-time cost of the check itself, which is precisely what the complexity analysis and the benchmarks that follow are designed to isolate.

7. COMPLEXITY OF ALGORITHM

Analysing an algorithm's complexity means describing how its resource use, whether time or memory, grows as the size of its input grows. It is rarely enough to say that one algorithm "feels faster" than another; a formal notion of growth is needed so that the comparison holds regardless of the machine it is run on [1]. Asymptotic notation supplies that formal notion, and it is applied here to both the per-character check in isolation and to the full counting loop that calls it n times.

8. SPACE COMPLEXITY

The space complexity of an algorithm is the amount of memory it requires as a function of its input size, separate from the space taken by the input itself [1][3].

`character_vowel_check()` allocates nothing beyond a handful of temporary comparison results, so its auxiliary space complexity is $O(1)$.

The set-based approach requires a frozenset of ten characters, which is built once, outside the per-character check, and reused for every subsequent lookup. Because that set does not grow with n, its contribution to the space complexity of the counting loop is also $O(1)$ with respect to n, though it is fair to note that the constant itself, $O(k)$ for a category of k members, is slightly larger than the near-zero footprint of the `if/elif` version. For the five/ten-vowel case this difference is negligible; it becomes more relevant, though still modest, in the scaled-category experiment discussed in Section 14.

Space complexity of `character_vowel_check()`: $O(1)$

Space complexity of set membership testing: $O(1)$ with respect to n, $O(k)$ to build the set

9. TIME COMPLEXITY

The time complexity of an algorithm is the amount of time it takes to run, expressed as a function of its input length n, and it is commonly described using three asymptotic notations [1][2]:

Big O – $O(n)$: the upper bound, describing the worst case.

Big Theta – $\Theta(n)$: the tight bound, describing the average case when upper and lower bounds coincide.

Big Omega – $\Omega(n)$: the lower bound, describing the best case.

These three notations are not tied to any one case by definition — Big O can describe a best case just as Big Omega can describe a worst case — but in conventional usage O is applied to the worst case, Θ to the average case, and Ω to the best case, and that convention is followed here [2][22].

10. DERIVATION OF TIME COMPLEXITY FOR `character_vowel_check()`

For a single call, `character_vowel_check()` performs at most five comparisons — one against each vowel pair — before returning. Because that number is fixed and does not depend on how large the surrounding string is, the per-call complexity is

$O(1)$ in terms of n, with a constant bounded by the size of the vowel category, $k = 5$ pairs.

This constant-time classification hides an asymmetry that matters in practice. The chain checks vowels in order and returns as soon as it finds a match, so its best case, a string that is entirely vowels, requires only one comparison per character. Its worst case, a character that is not a vowel at all, requires all five comparisons before falling through to the else branch. Because roughly 80 percent of English-language characters are consonants rather than vowels, a realistic input string causes the chain to hit its worst case far more often than its best case, which means the practical average case for ordinary text sits close to the theoretical worst case rather than comfortably below it, even though both remain $O(1)$ asymptotically.

Applied across the full counting loop, which calls the check once per character in a string of length n, the total cost is n calls at $O(1)$ each, giving an overall time complexity of $O(n)$ for `count_vowels_ifelif()`.

11. DERIVATION OF TIME COMPLEXITY FOR SET MEMBERSHIP TESTING

A set membership test computes a hash of the character and inspects the corresponding bucket, an operation whose average-case cost does not depend on how many elements are already in the set, provided the hash function spreads keys evenly [3][4]. For single-character strings, CPython's hash function is fast and well distributed, so this holds comfortably in practice. The average-case per-call complexity is therefore $O(1)$, and unlike the `if/elif` chain, that $O(1)$ cost is essentially symmetric: a hit and a miss cost the same, because both require exactly one hash computation and one bucket inspection, not a variable number of comparisons.

The worst case for a hash set is $O(n)$, reached only when many keys collide into the same bucket and the implementation falls back to scanning a chain within it [3][5]. For a set of ten single-character keys drawn from a well-behaved hash function, this degradation is not something that arises in ordinary use, and it did not arise in any of the trials reported in Section 12.

Applied across the counting loop, the total cost is n calls at $O(1)$ average-case each, giving the same overall asymptotic order, $O(n)$, as the `if/elif` version. On paper, then, the two implementations of `count_vowels()` are equivalent; Sections 12 through 14 examine whether that equivalence survives contact with an actual interpreter.

12. EMPIRICAL RUNTIME COMPARISON

Table 1 reports the median execution time, in milliseconds, of `count_vowels_ifelif()` and `count_vowels_set()` across the seven tested string lengths, together with the ratio between them.

Input (n)	if/elif (ms)	Set (ms)	Ratio
1,000	0.144	0.049	2.96x
5,000	0.624	0.267	2.34x
10,000	1.338	0.506	2.64x
50,000	6.855	2.600	2.64x
100,000	13.450	5.231	2.57x
500,000	72.388	27.611	2.62x
1,000,000	134.660	53.422	2.52x

Both columns grow linearly with n, exactly as the $O(n)$ derivation in Sections 10 and 11 predicts. What the asymptotic

analysis alone does not predict is the ratio column: across every tested size, the if/elif chain took between roughly 2.4 and 2.9 times as long as the set-based version, with the gap holding fairly steady rather than narrowing or widening as n increases. This is the constant-factor difference the theory allows for but does not quantify.

13. GRAPHICAL REPRESENTATION

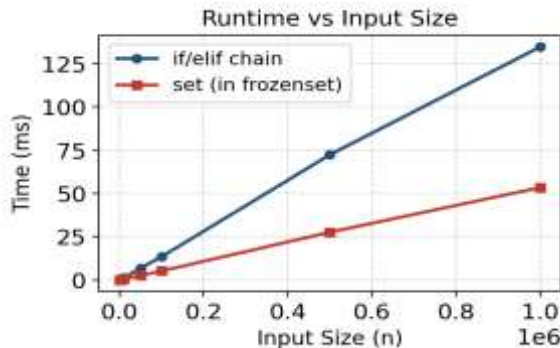


Figure 1. Measured runtime of character_vowel_check() (if/elif) and set membership testing against input size n .

Both lines rise in an almost straight, proportional path from the origin, which is the visual signature of linear, $O(n)$, growth. The vertical gap between the two lines widens in absolute terms as n increases, since a fixed ratio applied to a larger number produces a larger absolute difference, but the two lines remain parallel in shape — confirming that the two approaches are separated by a constant factor rather than by a difference in asymptotic order.

14. EFFECT OF TARGET SET SIZE ON PERFORMANCE

The fixed five-vowel case, by design, keeps k small and unchanging, which limits how much the if/elif chain is disadvantaged — five comparisons is not a large constant. To test how the two approaches behave once that assumption is relaxed, the second experiment (described in Section 3) held the input string length constant at 200,000 characters and scaled the size of the target category, k , from 5 to 320 members.

Size (k)	Linear (ms)	Hash (ms)	Ratio
5	13.473	4.719	2.86x
10	21.733	6.147	3.54x
20	38.329	5.601	6.84x
40	66.543	7.368	9.03x
80	94.482	7.138	13.24x
160	98.826	7.638	12.94x
320	98.558	7.167	13.75x

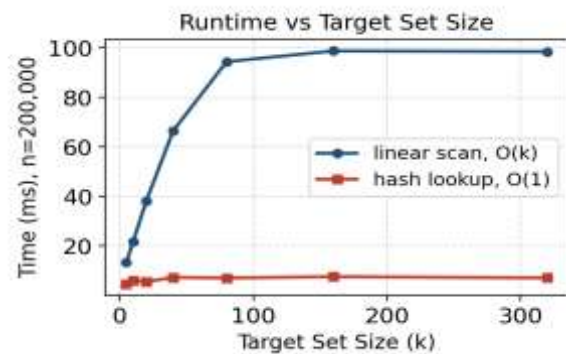


Figure 2. Measured runtime of linear (tuple) scanning versus hash-set lookup as the target category size k grows, for a fixed input of 200,000 characters.

The hash-set line stays essentially flat across the entire range of k , exactly as the $O(1)$ average-case derivation in Section 11 predicts: adding more members to the target set does not make a single lookup any slower. The linear-scan line, by contrast, climbs steeply as k grows from 5 to roughly 80, then levels off as k approaches the size of the alphabet being tested, since at that point most characters are found within the first few comparisons rather than requiring a full scan. The key result is the shape of the gap: at $k = 5$, the two approaches are separated by a ratio of roughly three; by $k = 40$, that ratio has grown past nine. A category that starts out as small as the vowel set does not stay small forever in real software — a vowel check is often extended to cover accented characters, or generalised into a broader character-classification routine — and this experiment shows how quickly the cost of the naive approach compounds once that happens.

15. DISCUSSION

Three findings stand out. First, the asymptotic complexity of the two approaches is identical for this task: both `count_vowels_ifelif()` and `count_vowels_set()` run in $O(n)$ time, and neither derivation in Sections 10 and 11 is wrong. Big- O notation was never intended to predict wall-clock time exactly; it predicts the shape of growth, and both implementations have the same shape. Anyone relying on asymptotic complexity alone to choose between them would find no difference worth reporting.

Second, the constant factor that asymptotic notation deliberately ignores turned out to be large enough to matter here — consistently more than double across every tested input size. This lines up with the observation, made directly in the CPython complexity reference, that constant factors “can significantly affect the constant factors: how quickly a typical program finishes” even when they do not change the algorithmic order [3]. The if/elif chain's disadvantage is not a fluke of this particular benchmark; it follows directly from Section 10's observation that ordinary English text triggers the chain's worst case, all five comparisons, far more often than its best case.

Third, and this is the finding most likely to be missed by an analysis that only looks at the five-vowel case, the two approaches do not stay equally matched once the category grows. Section 14 shows the gap widening from roughly threefold to ninefold as k grows from 5 to 40 members. A vowel checker is a narrow, fixed problem, but the underlying pattern — testing a character against a set of possibilities — recurs



throughout software in forms that are not nearly so narrow: validating characters against an allowed set, checking file extensions against a supported list, or matching tokens against a set of keywords. For any of these, the choice made when the category has five members is not necessarily the right choice once it has fifty.

The results here are naturally limited to CPython on a single machine; a compiled language, a JIT-optimising interpreter such as PyPy, or a different CPU architecture could narrow or widen the measured gap without changing its direction. The experiments also do not cover Python's own built-in alternatives, such as `str.isalpha()` combined with a manual vowel check, or regular expressions, both of which trade a different set of constant factors for the same underlying task. Those comparisons are a natural extension of this case study rather than a gap in it.

16. CONCLUSION

For the specific, narrow problem of identifying a vowel among five possibilities, `character_vowel_check()` and set membership testing share the same asymptotic time complexity, $O(n)$ over a string of length n , and the same $O(1)$ space complexity per call. Anyone judging the two implementations by Big-O notation alone would be entitled to call them equivalent. The benchmarks in this paper show that equivalence is only half the picture: the set-based version was consistently more than twice as fast in absolute terms, a gap traced directly to the `if/elif` chain's tendency to run its full comparison budget on ordinary, consonant-heavy text rather than exiting early.

That gap is modest while the category stays as small as five or ten vowels, but it is not stable. Once the category is allowed to grow, as classification categories in real software so often do, the linear-scan approach degrades in a way the hash-based approach simply does not, and the two stop being close substitutes for one another. The practical recommendation that follows is not that `if/elif` chains are wrong — for a genuinely fixed, tiny category, their readability is a real advantage — but that set membership testing is the more defensible default whenever there is any chance the category will not stay tiny, and it costs nothing in this case study's results to choose it even when the category does stay small.

17. ACKNOWLEDGEMENT

Apart from my own effort, the completion of this research paper owes a great deal to the guidance and encouragement of others. I would like to thank [Name of Guide / Mentor], [Designation], [Institution Name], for the direction and critical feedback provided throughout the implementation and writing of this study, and [Name of Principal / Head of Institution], [Institution Name], for the encouragement extended during its preparation. I am also grateful to my family for their continued support.

18. REFERENCES

1. *GeeksforGeeks – Asymptotic Notations for Analysis of Algorithms*. <https://www.geeksforgeeks.org/dsa/types-of-asymptotic-notations-in-complexity-analysis-of-algorithms/>
2. *Educative – Big O Notation: A primer for beginning devs*. <https://www.educative.io/blog/a-big-o-primer-for-beginning-devs>
3. *Python Wiki – TimeComplexity (CPython operation complexities)*. <https://wiki.python.org/moin/TimeComplexity>
4. *GeeksforGeeks – Internal working of Set in Python*. <https://www.geeksforgeeks.org/python/internal-working-of-set-in-python/>
5. *GeeksforGeeks – Complexity Cheat Sheet for Python Operations*. <https://www.geeksforgeeks.org/python/complexity-cheat-sheet-for-python-operations/>
6. *GeeksforGeeks – Linear Search Algorithm*. <https://www.geeksforgeeks.org/dsa/linear-search/>
7. *Wikipedia – Linear search*. https://en.wikipedia.org/wiki/Linear_search