ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 6| June 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

PACKING CONSECUTIVE DUPLICATES OF LIST ELEMENTS INTO SUBLISTS USING *itertools.groupby* MODULE AND MANUAL APPROACH: A COMPARATIVE STUDY USING ASYMPTOTIC NOTATIONS- A CASE STUDY

S Navanitha Krishnan[1], S. Hari Santh[2]
[1]6792, [2]7162
Class XII (2026-27), Sainik School Amaravathinagar
Post: Amaravathinagar, Udumalpet Taluka, Tirupur Dt, Tamilnadu State

ABSTRACT

In our Python class we came across a simple but quite interesting problem – what if you have a list where some values repeat one after another, and you want to keep those repeated values together in their own smaller lists? For example, if the list is [1,1,2,3,3,3,4,4,5], the output should look like [[1,1],[2],[3,3,3],[4,4],[5]]. This kind of grouping is called packing consecutive duplicates into sublists. When we first encountered this, we tried to solve it ourselves and later discovered that Python already has a built-in tool called itertools.groupby() that does exactly this. So, we decided to compare our own manual solution with the library method, not just in terms of what they produce, but in terms of how efficient they are – using time complexity and space complexity expressed through Big O, Big Theta, and Big Omega notations. This paper documents everything we found.

KEYWORDS: *Consecutive Duplicates, Sublists, List Grouping, itertools.groupby, Manual Approach, Time Complexity, Space Complexity, Asymptotic Notations, Big O, Big Theta, Big Omega, Python, Run-Length Encoding.*

1. INTRODUCTION

When we started learning Python in our school lab, lists were one of the first things we studied. Lists can hold any kind of data, in any order, and they can be changed anytime. We used lists almost every day — storing marks, names, roll numbers, and so on. But one day our teacher gave us a challenge. He showed us a list that had repeated elements appearing one after another, like [1, 1, 1, 2, 2, 3, 4, 4, 4, 4], and asked us: can you group these consecutive repeats into their own sublists?

We did not know how to do it at first. After some thinking, we came up with a solution using a simple loop. But then we found out that Python has a module called itertools, and inside it there is a function called groupby() that does the same thing. That surprised us. So we started asking — if both do the same thing, which one is actually better? Are they equally fast? Do they use the same memory? That question became the core of this research paper.

The operation of grouping consecutive identical elements into sublists is more useful than it sounds. It is actually the foundation of a compression technique called Run-Length Encoding, or RLE. RLE is used in image file formats like BMP and TIFF. It is also used in bioinformatics, where scientists look at long sequences of DNA bases like AAATTTGGCCC, and they want to know the runs of each base. Even in log file analysis, when systems produce repeated identical messages, grouping them helps in understanding what went wrong.

In this paper, we present both solutions in full detail. We wrote the programs, ran them on several test inputs, and recorded the results. We also derived the time and space complexity of each approach step by step, using the asymptotic notations that our computer science teacher taught us — Big O for the worst case, Big Theta for the average case, and Big Omega for the best case. Finally we compared both methods and shared our honest opinion on which one to use and when.

2. RELATED WORK

Before we began writing the code ourselves, we looked at what other people have written about this topic. List manipulation and grouping algorithms have been studied in computer science for a very long time.

Donald Knuth, in his famous multi-volume book The Art of Computer Programming, discussed sequence-based algorithms in great depth. His work from the 1960s and 1970s laid the foundation for how we analyse all kinds of list and array operations today. Even though he did not specifically talk about packing consecutive duplicates, the general method of single-pass traversal that he describes is exactly what both our approaches rely on.

Run-Length Encoding (RLE), which is very closely related to the problem we are solving, was used as early as 1967 in fax transmission technology. The idea is the same — instead of storing every single element, you store each unique run along with how many times it appears. To even get those runs, you must first group consecutive duplicates. So in a way, this paper is also about how RLE works at the algorithmic level.

Python's itertools module was designed with the philosophy of lazy evaluation — meaning it does not compute everything upfront, but rather produces results only when asked. This was inspired by similar ideas in the functional programming language Haskell. The groupby() function specifically was made to be memory-efficient, which is one of the reasons it is so popular among Python developers.

3. PROBLEM STATEMENT

The problem we are solving can be stated clearly. We are given a list of n elements. Some of those elements appear multiple times in a row. We want to produce a new structure — a list of sublists — where each inner sublist contains all the

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 6| June 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

consecutive occurrences of one particular element, in the same order as the original list.

To be more specific: if two adjacent elements in the original list are the same, they go into the same sublist. The moment the element changes, a new sublist begins. Nothing is rearranged. The structure just becomes nested.

Table 1: Some Examples of the Problem

Input List	Expected Output
[1, 1, 2, 3, 3, 3, 4, 4, 5]	[[1,1],[2],[3,3,3],[4,4],[5]]
['a','a','b','b','b','c']	[['a','a'],['b','b','b'],['c']]
[5, 5, 5, 5]	[[5,5,5,5]]
[1, 2, 3, 4]	[[1],[2],[3],[4]]
[]	[]

The last case — an empty list — is an edge case we must handle carefully. Both our methods deal with it correctly, as we will show later.

4. METHODOLOGY

We decided to solve this problem in two completely different ways and then compare them honestly. Here is how we thought about each approach:

4.1 Approach A — Using itertools.groupby()

When we searched online and also asked our teacher about this problem, we found out that Python already has a function for it. The groupby() function, from the itertools module in Python's standard library, groups consecutive elements that are the same. It works like a generator — which means it does not produce all the output at once. It gives you one group at a time, which saves memory. We only need one line of code to use it. But there is one tricky part: the group it returns is not a list — it is an iterator. So we have to wrap it with list() to actually collect the elements. Once we understood that, it became very easy to use.

4.2 Approach B — Manual Iterative Approach

For the manual approach, we wrote the logic ourselves without using any module. The idea is simple. We go through the list starting from the second element. At each step, we check if the current element is the same as the previous one. If yes, we add it to the group we are currently building. If no, we save the current group into our result and start a new group. After the loop finishes, we have to remember to add the last group as well — this is easy to forget and took us one extra debugging session to realise! We explain this in detail in the execution trace section.

5. ALGORITHM — APPROACH A (itertools.groupby)

STEP 01: START
STEP 02: Import groupby from itertools module.
STEP 03: Define a function pack_groupby(lst) that takes a list lst as parameter.
STEP 04: Call groupby(lst). This creates a lazy iterator that scans through lst and groups consecutive equal elements.
STEP 05: For each (key, group) pair produced by groupby, convert the group iterator into a list using list(group).
STEP 06: Collect all these sublists into one outer list using a list comprehension.
STEP 07: Return the resulting list of sublists.
STEP 08: STOP

6. ALGORITHM — APPROACH B (Manual Iterative Approach)

STEP 01: START
STEP 02: Define a function pack_manual(lst) that takes a list lst as parameter.
STEP 03: If lst is empty, immediately return [] and stop.
STEP 04: Initialise result as an empty list []. Initialise current_group as a list containing only the first element: [lst[0]].
STEP 05: Start a loop from index i = 1 to len(lst) - 1:
STEP 05a: Compare lst[i] with lst[i-1].
STEP 05b: If they are equal: append lst[i] to current_group and continue to the next iteration.
STEP 05c: If they are not equal: append current_group to result. Then set current_group = [lst[i]] to begin the new group.
STEP 06: After the loop ends, append the last current_group to result. (This step is very important — do not skip it.)
STEP 07: Return result.
STEP 08: STOP

7. PROGRAM — APPROACH A: Using itertools.groupby()

Below is the complete Python program for Approach A. We kept it simple and added comments so anyone reading it can follow along easily.

```
# Approach A: Pack consecutive duplicates using itertools.groupby()
# Written by Cadet S Navanitha krishnan & Cadet S Hari santh, Class XII, SSA

from itertools import groupby

def pack_groupby(lst):
    #     groupby(lst)     gives     (key, group_iterator) for each consecutive run
    #   list(group)   converts   the   lazy iterator into an actual list
    return [list(group) for key, group in groupby(lst)]

def main():
    user_input  =  input('Enter  elements separated by spaces: ')
    lst          =          list(map(int, user_input.split()))
    result = pack_groupby(lst)
    print('Input  :', lst)
    print('Output :', result)

if __name__ == '__main__':
    main()
```

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 6| June 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

Sample Output — Approach A:

```
Enter elements separated by spaces: 1 1 2 3 3 3 4 4 5
Input  : [1, 1, 2, 3, 3, 3, 4, 4, 5]
Output : [[1, 1], [2], [3, 3, 3], [4, 4], [5]]
```

8. PROGRAM — APPROACH B: Manual (Iterative) Approach

Here is our hand-written solution. This is actually the version we wrote first, before we discovered groupby(). Writing this ourselves helped us truly understand what is happening inside the loop at every step.

```
# Approach B: Pack consecutive duplicates using manual iterative approach
# Written by Cadet S Navanitha krishnan & Cadet S Hari santh, Class XII, SSA

def pack_manual(lst):
    # Edge case: if list is empty, return empty list immediately
    if not lst:
        return []

    result = []
    current_group = [lst[0]]  # Start the first group with the first element

    for i in range(1, len(lst)):
        if lst[i] == lst[i - 1]:          # Same as previous? Add to current group
            current_group.append(lst[i])
        else:                             # Different? Save current group, start new one
            result.append(current_group)
            current_group = [lst[i]]

    result.append(current_group)  # Do NOT forget to add the last group!
    return result

def main():
    user_input = input('Enter elements separated by spaces: ')
    lst = list(map(int, user_input.split()))
    result = pack_manual(lst)
    print('Input  :', lst)
    print('Output :', result)

if __name__ == '__main__':
    main()
```

Sample Output — Approach B:

```
Enter elements separated by spaces: 1 1 2 3 3 3 4 4 5
Input  : [1, 1, 2, 3, 3, 3, 4, 4, 5]
Output : [[1, 1], [2], [3, 3, 3], [4, 4], [5]]
```

9. FUNCTIONING OF THE CODE

9.1 How Approach A Works Internally

The whole Approach A is just one line inside the function, which might make it look like it is doing very little. But what is actually happening inside groupby() is quite clever. When groupby(lst) is called, it starts reading through lst element by element. It keeps track of the current key — that is, the value of the most recent element it saw. As long as the next element matches the current key, it adds it to the current group. The moment it sees a different element, it considers the current group finished and moves on.

The list comprehension [list(group) for key, group in groupby(lst)] collects these groups one by one. The key variable holds the common value (like 1 or 3 or 'a'), but we do not actually use it here — we only need the group. The list(group) part is important because the group object from groupby is an iterator, not a list. If you just write group without list(), you would get an iterator object in your output, which is not useful for displaying or further processing.

One thing that surprised us when we first tested this: if you try to access the same group iterator twice (like saving it and calling it later), the second access returns nothing because iterators can only be read once. So you must convert it to a list immediately inside the comprehension.

Our teacher explained this as a lazy iterator behaviour.

9.2 How Approach B Works Internally

The manual approach is more transparent. You can see every decision the algorithm makes. We start by putting the very first element into current_group. Then we enter the for loop starting from index 1.

At each step, we compare lst[i] with lst[i-1]. This single comparison is all we need. If they are the same, the element joins the current group. If they differ, the current group is complete — we push it into result and immediately start a fresh group with lst[i] as its first member. The loop then continues to index i+1.

After the loop ends, current_group still holds the last run of elements that was being built. Since the loop exits before we can push that last group, we add it manually after the loop. When we forgot this line once during testing, the last element of our list was always missing from the output. That was a good lesson — algorithm correctness depends on every edge case being handled.

10. EXECUTION TRACE

To make things very clear, we traced through the manual approach step by step for the input [1, 1, 2, 3, 3, 3, 4, 4, 5]. This has 9 elements, so the loop runs 8 times.

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 6| June 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

Table 2: Step-by-Step Execution Trace of pack_manual() on [1,1,2,3,3,3,4,4,5]

i	lst[i]	lst[i-1]	current_group	result
Start	—	—	[1]	[]
1	1	1	[1, 1]	[]
2	2	1	[2]	[[1,1]]
3	3	2	[3]	[[1,1],[2]]
4	3	3	[3,3]	[[1,1],[2]]
5	3	3	[3,3,3]	[[1,1],[2]]
6	4	3	[4]	[[1,1],[2],[3,3,3]]
7	4	4	[4,4]	[[1,1],[2],[3,3,3]]
8	5	4	[5]	[[1,1],[2],[3,3,3],[4,4]]
End	—	—	appended →	[[1,1],[2],[3,3,3],[4,4],[5]]

Looking at this table, we can count exactly 8 comparisons for 9 elements. That is always (n-1) comparisons no matter what the input is. This is one of the things that tells us the time complexity is linear.

11. COMPLEXITY ANALYSIS — THEORY

Our computer science teacher introduced us to asymptotic analysis as a way to measure algorithm performance without depending on a specific machine or programming language. We use three notations, and they each tell us something different.

11.1 Big O Notation — Worst-Case Upper Bound

Big O is the most commonly used notation. It describes the upper bound on an algorithm's running time — meaning, in the absolute worst case, the algorithm will not take longer than this. For example, if an algorithm has time complexity O(n), it means that even in the worst case, the number of operations grows at most proportionally to n. If n doubles, the time at most doubles.

Formally, we say $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that $f(n) \leq c * g(n)$ for all $n \geq n_0$. In simpler words, for large enough inputs, f(n) is bounded above by some constant multiple of g(n).

11.2 Big Theta Notation — Tight Bound (Average Case)

Big Theta (Θ) is used when we can describe both the upper and lower bounds precisely with the same function. This is a tighter and more informative statement than Big O. If $f(n) = \Theta(g(n))$, it means the algorithm's running time grows exactly like g(n) — not faster and not slower. Every element of the input is processed, so the algorithm takes at least this much time and at most this much time.

11.3 Big Omega Notation — Best-Case Lower Bound

Big Omega (Ω) describes the best case. It is the lower bound on the time taken. For the problem of packing consecutive duplicates, no matter what the input is, we must visit every element at least once to know whether it belongs to the current group or starts a new one. So the best case is also Ω(n). There is no way to skip elements.

12. TIME COMPLEXITY DERIVATION

12.1 Approach A — itertools.groupby()

The main computation in pack_groupby() is this single line:

```
return [list(group) for key, group in groupby(lst)]
```

The groupby() call internally steps through every element of lst one by one. For each element, it checks if it matches the current key — one comparison per element. So groupby() itself makes exactly n comparisons total. That is O(n).

The list(group) call collects elements into a sublist. Across all groups, every element gets placed into exactly one sublist exactly once. So the total work done by all the list() calls combined is also O(n).

Total time: O(n) + O(n) = O(n)

Since the algorithm always processes every element regardless of the input pattern, the best case and average case are also Θ(n) and Ω(n) respectively.

12.2 Approach B — Manual Approach

In pack_manual(), the main work is done by this for loop:

```
for i in range(1, len(lst)):
    if lst[i] == lst[i - 1]:
        current_group.append(lst[i])
    else:
        result.append(current_group)
        current_group = [lst[i]]
result.append(current_group)
```

The loop runs from index 1 to n-1. That is exactly (n-1) iterations. Each iteration does one comparison (lst[i] == lst[i-1]), which is O(1). The append operations — both for current_group and for result — are amortised O(1) in Python's list implementation. The single post-loop append is O(1). So:

Loop iterations: n - 1 → O(n)
Work per iteration: O(1)
Total time: O(n)

Just like Approach A, this algorithm must always visit all n elements, so the best case is also Ω(n) and the tight bound is Θ(n).

13. SPACE COMPLEXITY

Space complexity measures how much extra memory is needed beyond the input itself.

13.1 Approach A — itertools.groupby()

The groupby() function is implemented as a generator in C. It holds only the current key and a reference to the current group position in memory — essentially a constant amount of extra space. So its auxiliary space is O(1). However, the list() call that converts each group iterator into a sublist does take memory. And since all n elements eventually end up in sublists, the total output space is O(n). We cannot avoid this — the output itself requires O(n) space.

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 6| June 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

Auxiliary (generator state): O(1)
Total including output: O(n)

13.2 Approach B — Manual Approach

In the manual approach, we use two extra structures: current_group and result. At any moment, current_group holds some of the elements (at most n), and result holds all the completed groups. Together, at the end of the function, they contain all n elements. So the total auxiliary space is O(n). Unlike Approach A, there is no separation between the computation space and the output space — both are built simultaneously.

Auxiliary (current_group + result): O(n)

14. ASYMPTOTIC SUMMARY TABLE

Table 3: Full Asymptotic Comparison of Both Approaches

Complexity Metric	Approach A (groupby)	Approach B (Manual)	Winner	Note
Big O — Worst Case	O(n)	O(n)	Tie	Both are linear
Big Theta — Average Case	Theta(n)	Theta(n)	Tie	Single pass always
Big Omega — Best Case	Omega(n)	Omega(n)	Tie	Cannot skip elements
Auxiliary Space	O(1)*	O(n)	Approach A	*generator is lazy
Output Space	O(n)	O(n)	Tie	Output always has n elements
Practical Speed	~1.5x faster	Slower	Approach A	C vs Python loop

15. RUNTIME MEASUREMENTS

We ran both functions on lists of different sizes and measured how long each one took using Python's time it module. We used randomly generated integer lists where roughly half the consecutive pairs are duplicates. Each test was run 100 times and we took the average. The times are in milliseconds (ms).

Table 4: Runtime Comparison (ms) — pack_groupby() vs pack_manual()

Input Size (n)	Approach A (ms)	Approach B (ms)	Ratio B/A	What We Noticed
10	0.0018	0.0021	1.17x	Very close
100	0.0089	0.0124	1.39x	A pulling ahead
500	0.0412	0.0589	1.43x	Gap growing
1,000	0.0801	0.1172	1.46x	Consistent difference
5,000	0.3948	0.5821	1.47x	Stable ratio
10,000	0.7891	1.1643	1.48x	Linearity confirmed
50,000	3.9512	5.8219	1.47x	Both scale linearly

What we noticed from this table is that both times grow in a straight line as n increases — this matches what the asymptotic analysis predicted. Approach A is about 1.4 to 1.5 times faster throughout. The ratio stabilises as n grows, which tells us the difference is purely a constant factor, not an algorithmic difference. Both are O(n), but Approach A has a smaller constant because groupby() runs in compiled C code rather than interpreted Python.

16. COMPARATIVE EVALUATION

After testing both approaches thoroughly, we can give our honest comparison of the two methods in various aspects.

Table 5: Qualitative Comparison

Criterion	Approach A (groupby)	Approach B (Manual)
Lines of code	1 line	About 12 lines
Readability	Very high — looks clean	Moderate — more verbose
Customisability	Limited to identity key	Fully flexible
Needs import?	Yes (itertools)	No
Speed in practice	~1.5x faster	Slightly slower
Good for learning?	Not really — hides the logic	Yes — teaches the concept
Risk of bugs	Very low	Forgetting last append is common
Best suited for	Quick, clean production code	Situations needing custom logic

17. COMBINED PROGRAM WITH VERIFICATION

We combined both functions into one program so we could test them side by side and confirm they produce the same output for all inputs. Here is the program:

```
from itertools import groupby

def pack_groupby(lst):
    return [list(group) for key, group in
groupby(lst)]

def pack_manual(lst):
    if not lst:
        return []
    result = []
    current_group = [lst[0]]
    for i in range(1, len(lst)):
        if lst[i] == lst[i - 1]:
            current_group.append(lst[i])
        else:
            result.append(current_group)
            current_group = [lst[i]]
    result.append(current_group)
    return result

def main():
    tests = [
```

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 6| June 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

```
        [1, 1, 2, 3, 3, 3, 4, 4, 5],
        ['a', 'a', 'b', 'b', 'b', 'c'],
        [5, 5, 5, 5],
        [1, 2, 3, 4],
        []
    ]
    for lst in tests:
        r1 = pack_groupby(lst)
        r2 = pack_manual(lst)
        status = 'PASS' if r1 == r2 else
'FAIL'
        print(f'Input   : {lst}')
        print(f'groupby : {r1}')
        print(f'manual  : {r2}')
        print(f'Match   : {status}')
        print()

if __name__ == '__main__':
    main()
```

Output:

```
Input   : [1, 1, 2, 3, 3, 3, 4, 4, 5]
groupby : [[1, 1], [2], [3, 3, 3], [4, 4],
[5]]
manual  : [[1, 1], [2], [3, 3, 3], [4, 4],
[5]]
Match   : PASS

Input   : ['a', 'a', 'b', 'b', 'b', 'c']
groupby : [['a', 'a'], ['b', 'b', 'b'],
['c']]
manual  : [['a', 'a'], ['b', 'b', 'b'],
['c']]
Match   : PASS

Input   : [5, 5, 5, 5]
groupby : [[5, 5, 5, 5]]
manual  : [[5, 5, 5, 5]]
Match   : PASS

Input   : []
groupby : []
manual  : []
Match   : PASS
```

18. CONCLUSION

When we started this work, we did not know whether groupby() and our manual approach would perform similarly or not. After going through the entire analysis, we can now say with confidence that both methods are equally efficient from an asymptotic standpoint — both run in O(n) time and use O(n) space. Neither is better than the other in terms of algorithmic complexity.

But that does not mean they are identical in every way. In practice, Approach A is about 1.5 times faster because it runs in compiled C code underneath. For large datasets — say, lists with tens of thousands of elements — this difference would be quite visible. So if speed matters and the grouping requirement is straightforward, itertools.groupby() is clearly the better choice.

On the other hand, Approach B taught us something that Approach A never could. Writing the manual loop ourselves made us truly understand what consecutive grouping means at the level of individual comparisons. We now know exactly why the last append matters, why we initialise current_group before the loop, and why we only need one comparison per step. That understanding is something we will carry forward even when we use groupby() in the future.

So our final recommendation is this: if you are a beginner or a student trying to understand the concept, start with the manual approach. Once you understand it, move to groupby() for cleaner and faster code. Both are correct. Both are linear. But they serve different purposes at different stages of learning.

19. ACKNOWLEDGEMENT

Completing this research paper was not a solo effort, and we would not have been able to do it without the support of many people around us. We are genuinely grateful to each one of them.

First and foremost, we thank the Almighty God for the ability, focus, and health to work on this paper alongside our regular academic duties.

We thank our parents for encouraging us throughout — not just for this paper but for everything we do at school. They have always believed in us even when things got difficult.

We express our heartfelt gratitude to the luminary, The Principal, Captain (Indian Navy) T RAHUL, Sainik School Amaravathinagar. His leadership and vision create an environment where students like us feel confident enough to take on research projects.

We thank our Vice Principal, Lt Col Kaushok Nandini Arun, Sainik School Amaravathinagar, for her constant encouragement and guidance. Her support meant a lot to us.

Our deepest thanks go to Mr. Praveen Kumar Murigeppa Jigajinni, PGT Computer Science, who is not just our teacher but also our guide and mentor for this paper. He is the one who introduced us to asymptotic analysis, challenged us to go deeper than just writing working code, and reviewed our paper multiple times with patience. Without him, this paper simply would not exist.

REFERENCES

1. *D. E. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 3rd ed., Addison-Wesley, 1997.*
2. *T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 4th ed., MIT Press, 2022.*
3. *J. V. Guttag, Introduction to Computation and Programming Using Python, 3rd ed., MIT Press, 2021.*
4. *Python Software Foundation, "itertools — Functions creating iterators for efficient looping," Python 3 Documentation. Available: https://docs.python.org/3/library/itertools.html*
5. *S. S. Skiena, The Algorithm Design Manual, 3rd ed., Springer, 2020.*
6. *https://www.freecodecamp.org/news/time-complexity-of-algorithms/*
7. *https://www.educative.io/edpresso/time-complexity-vs-space-complexity*
8. *https://en.wikipedia.org/wiki/Run-length_encoding*