1. Σταθερή Πολυπλοκότητα (O(1))

Αυτός ο αλγόριθμος έχει την ίδια διάρκεια εκτέλεσης ανεξαρτήτως του μεγέθους της εισόδου.

```
def get_first_element(arr):
    return arr[0]
```

Ανάλυση: Ο αλγόριθμος επιστρέφει πάντα το πρώτο στοιχείο ενός πίνακα, ανεξάρτητα από το μέγεθος του πίνακα. Επομένως, η πολυπλοκότητα είναι **O(1)**.

---

2. Γραμμική Πολυπλοκότητα (O(n))

Ο χρόνος εκτέλεσης αυξάνεται αναλογικά με το μέγεθος της εισόδου.

```
def print_elements(arr):
    for element in arr:
        print(element)
```

Ανάλυση: Ο αλγόριθμος εκτυπώνει κάθε στοιχείο του πίνακα `arr`. Επομένως, εκτελείται μία φορά για κάθε στοιχείο. Άρα η πολυπλοκότητα είναι **O(n)**, όπου n είναι το μέγεθος του πίνακα.

---

3. Τετραγωνική Πολυπλοκότητα (O(n^2))

Οι εμφωλευμένοι βρόχοι οδηγούν σε αύξηση του χρόνου εκτέλεσης με βάση το τετράγωνο του μεγέθους της εισόδου.

```
def print_pairs(arr):
    for i in range(len(arr)):
        for j in range(len(arr)):
            print(arr[i], arr[j])
```

Ανάλυση: Ο πρώτος βρόχος εκτελείται n φορές και για κάθε τιμή του `i`, ο δεύτερος βρόχος εκτελείται επίσης n φορές. Συνολικά έχουμε $n * n = n^2$ εκτελέσεις, άρα η πολυπλοκότητα είναι **$O(n^2)$**.

---

4. Λογαριθμική Πολυπλοκότητα (O(log n))

Αυτό το είδος πολυπλοκότητας εμφανίζεται σε αλγόριθμους που μειώνουν το μέγεθος της εισόδου σταδιακά, όπως η δυαδική αναζήτηση.

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
```

```
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Ανάλυση: Σε κάθε βήμα της αναζήτησης, ο αλγόριθμος χωρίζει τον πίνακα στα δύο. Αυτό σημαίνει ότι ο αριθμός των στοιχείων μειώνεται στο μισό κάθε φορά, οδηγώντας σε πολυπλοκότητα **O(log n)**.

---

5. Γραμμική-Λογαριθμική Πολυπλοκότητα (O(n log n))

Η συγχώνευση ή η τακτοποίηση ενός πίνακα με αλγόριθμο διαίρει και βασίλευε, όπως το Merge Sort ή το Quick Sort, έχει αυτή την πολυπλοκότητα.

Παράδειγμα Merge Sort:

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
```

Ανάλυση: Το Merge Sort διαιρεί τον πίνακα στα δύο μέχρι να φτάσει σε υποπίνακες μήκους 1, που είναι το λογαριθμικό μέρος (log n). Η συγχώνευση των υποπινάκων χρειάζεται γραμμικό χρόνο O(n). Άρα η συνολική πολυπλοκότητα είναι **O(n log n)**.

---

6. Εκθετική Πολυπλοκότητα (O(2^n))

Αυτή η πολυπλοκότητα εμφανίζεται σε προβλήματα όπου ο αλγόριθμος εξερευνά όλους τους δυνατούς συνδυασμούς εισόδου, όπως στην αναδρομική λύση του προβλήματος Fibonacci.

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Ανάλυση: Σε κάθε κλήση, ο αλγόριθμος κάνει δύο επιπλέον αναδρομικές κλήσεις. Αυτό σημαίνει ότι ο αριθμός των κλήσεων αυξάνεται εκθετικά, με πολυπλοκότητα **O(2^n)**.

---

7. Πολυπλοκότητα O(n!)

Πολυπλοκότητα αυτού του τύπου εμφανίζεται σε αλγόριθμους που αναζητούν όλους τους δυνατούς συνδυασμούς μιας σειράς στοιχείων, όπως στην εύρεση όλων των πιθανών διατάξεων (permutations).

```
import itertools

def generate_permutations(arr):
    return list(itertools.permutations(arr))
```

Ανάλυση: Αν έχεις n στοιχεία, υπάρχουν n! πιθανοί τρόποι να τα διατάξεις. Η πολυπλοκότητα είναι **O(n!)**.