টাইম কমপ্লেক্সিটি কী?

টাইম কমপ্লেক্সিটি (Time Complexity) হলো একটি অ্যালগরিদমের কার্যকারিতা বা দক্ষতা মাপার একটি উপায়, যা বলে দেয় যে ইনপুটের আকার (n) বাড়ার সাথে সাথে অ্যালগরিদমটি চালাতে কত সময় লাগবে। আমরা এটাকে সাধারণত **Big O নোটেশন** (O) দিয়ে প্রকাশ করি, যা শুধুমাত্র অ্যালগরিদমের সবচেয়ে প্রভাবশালী অংশটির দিকে ফোকাস করে।

টাইম কমপ্লেক্সিটি আমাদের বুঝতে সাহায্য করে:

- অ্যালগরিদম কত দ্রুত কাজ করে।
- বড় ইনপুটের ক্ষেত্রে এটি কতটা স্কেল করে।

Big O নোটেশনের সহজ ব্যাখ্যা

Big O নোটেশন আমাদের বলে অ্যালগরিদমের সময় বা স্টেপ সংখ্যা কীভাবে ইনপুটের আকারের (n) উপর নির্ভর করে বাড়ে। এখানে কিছু সাধারণ টাইম কমপ্লেক্সিটি:

1. **O(1)** - ধ্রুবক সময় (Constant Time): ইনপুট যত বড়ই হোক, সময় একই থাকে।
2. **O(n)** - রৈখিক সময় (Linear Time): ইনপুটের আকারের সাথে সময় সমানুপাতে বাড়ে।
3. **$O(n^2)$** - বর্গাকার সময় (Quadratic Time): ইনপুটের আকারের বর্গের সমানুপাতে সময় বাড়ে।
4. **O(log n)** - লগারিদমিক সময় (Logarithmic Time): ইনপুট বাড়লেও সময় খুব ধীরে বাড়ে।
5. **O(n log n)** - লিনিয়ার-লগারিদমিক সময়: কিছু দ্রুত সর্টিং অ্যালগরিদমে দেখা যায়।

উদাহরণ দিয়ে বোঝা যাক

আমরা পাইথনে কয়েকটি উদাহরণ দেখবো, যাতে টাইম কমপ্লেক্সিটি কীভাবে কাজ করে তা পরিষ্কার হয়।

১. O(1) - ধ্রুবক সময়

ধরো, আমরা একটি লিস্টের প্রথম উপাদান প্রিন্ট করতে চাই।

```python
def get_first_item(lst):
    return lst[0]  # শুধুমাত্র প্রথম উপাদান অ্যাক্সেস করা হচ্ছে

my_list = [1, 2, 3, 4, 5]
print(get_first_item(my_list))  # আউটপুট: 1
```

ব্যাখ্যা:

- এখানে লিস্ট যত বড়ই হোক (১০ বা ১০০০ উপাদান), আমরা শুধু প্রথম উপাদান অ্যাক্সেস করছি।
- এটি একটি ধ্রুবক অপারেশন, তাই টাইম কমপ্লেক্সিটি **O(1)**।
- অর্থাৎ, ইনপুটের আকার (n) বাড়লেও সময় একই থাকে।

---

২. O(n) - রৈখিক সময়

ধরো, আমরা একটি লিস্টে একটি নির্দিষ্ট সংখ্যা খুঁজতে চাই (লিনিয়ার সার্চ)।

```python
def linear_search(lst, target):
    for i in range(len(lst)):
        if lst[i] == target:
            return i
    return -1

my_list = [1, 3, 5, 7, 9]
print(linear_search(my_list, 7))  # আউটপুট: 3
```

ব্যাখ্যা:

- এখানে আমরা লিস্টের প্রতিটি উপাদান চেক করছি।
- সবচেয়ে খারাপ ক্ষেত্রে (target না পাওয়া গেলে), আমাদের পুরো লিস্ট (n উপাদান) স্ক্যান করতে হবে।
- তাই টাইম কমপ্লেক্সিটি **O(n)**, কারণ সময় ইনপুটের আকারের সমানুপাতে বাড়ে।

---

৩. $O(n^2)$ - বর্গাকার সময়

ধরো, আমরা একটি লিস্টে বাবল সর্ট (Bubble Sort) অ্যালগরিদম ব্যবহার করে উপাদানগুলো সাজাতে চাই।

```python
def bubble_sort(lst):
    n = len(lst)
    for i in range(n):
        for j in range(0, n - i - 1):
            if lst[j] > lst[j + 1]:
                lst[j], lst[j + 1] = lst[j + 1], lst[j]
    return lst

my_list = [64, 34, 25, 12, 22]
print(bubble_sort(my_list))  # আউটপুট: [12, 22, 25, 34, 64]
```

ব্যাখ্যা:

- বাবল সর্টে আমরা প্রতিটি উপাদানের জন্য (n) আবার প্রতিটি উপাদানের সাথে তুলনা করি (n-1, n-2, ...), যা মোটামুটি $n \times n = n^2$ অপারেশন।
- তাই টাইম কমপ্লেক্সিটি **$O(n^2)$**।
- ইনপুটের আকার বাড়লে সময় বর্গাকার হারে বাড়ে, যা বড় লিস্টের জন্য ধীর।

---

৪. O(log n) - লগারিদমিক সময়

ধরো, আমরা একটি সাজানো লিস্টে বাইনারি সার্চ (Binary Search) ব্যবহার করে একটি সংখ্যা খুঁজতে চাই।

```python
def binary_search(lst, target):
    left, right = 0, len(lst) - 1
    while left <= right:
        mid = (left + right) // 2
        if lst[mid] == target:
            return mid
        elif lst[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

my_list = [1, 3, 5, 7, 9, 11, 13]
print(binary_search(my_list, 7))  # আউটপুট: 3
```

ব্যাখ্যা:

- বাইনারি সার্চ প্রতিবার লিস্টের অর্ধেক অংশ বাদ দেয়।
- যেমন, ১০০০ উপাদানের লিস্টে প্রথমে ৫০০, তারপর ২৫০, তারপর ১২৫... এভাবে ক্রমাগত অর্ধেক হয়ে যায়।
- এটি লগারিদমিক হারে কাজ করে, তাই টাইম কমপ্লেক্সিটি **O(log n)**।
- বড় ইনপুটের জন্য এটি খুব দ্রুত।

---

৫. O(n log n) - লিনিয়ার-লগারিদমিক সময়

অনেক দক্ষ সর্টিং অ্যালগরিদম, যেমন মার্জ সর্ট (Merge Sort) বা পাইথনের ডিফল্ট `sorted()` ফাংশন, এই কমপ্লেক্সিটির। এখানে একটি সরল উদাহরণ:

```python
def merge_sort(lst):
    if len(lst) <= 1:
        return lst
    mid = len(lst) // 2
    left = merge_sort(lst[:mid])
    right = merge_sort(lst[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result

my_list = [64, 34, 25, 12, 22]
print(merge_sort(my_list))  # আউটপুট: [12, 22, 25, 34, 64]
```

ব্যাখ্যা:

- মার্জ সর্ট লিস্টকে অর্ধেক করে (log n) এবং প্রতিটি অর্ধেকে n অপারেশন করে মার্জ করে।
- তাই টাইম কমপ্লেক্সিটি **O(n log n)**, যা বাবল সর্ট ($O(n^2)$) থেকে অনেক দ্রুত।

তুলনামূলক ধারণা

একটি টেবিল দিয়ে বিভিন্ন টাইম কমপ্লেক্সিটির তুলনা দেখি:

ইনপুটের আকার (n)	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$
10	1	~3	10	~30	100
100	1	~7	100	~700	10,000
1000	1	~10	1000	~10,000	1,000,000

- **O(1)** সবসময় একই সময় নেয়।
- **O(log n)** খুব দ্রুত, বড় ইনপুটেও।
- **O(n)** ইনপুটের সমানুপাতে বাড়ে।
- **O(n log n)** দ্রুত সর্টিংয়ের জন্য ভালো।
- **$O(n^2)$** বড় ইনপুটে ধীর।

কীভাবে টাইম কমপ্লেক্সিটি গণনা করবেন?

1. **লুপ গণনা করুন**:
   - একটি একক লুপ: O(n)
   - নেস্টেড লুপ: $O(n^2)$
2. **রিকারসন বিশ্লেষণ করুন**:
   - যেমন, বাইনারি সার্চে প্রতিবার অর্ধেক হয়, তাই O(log n)।
3. **সবচেয়ে প্রভাবশালী অংশ দেখুন**:
   - যদি একটি ফাংশনে O(n) এবং $O(n^2)$ থাকে, তাহলে টাইম কমপ্লেক্সিটি $O(n^2)$।

উপসংহার

টাইম কমপ্লেক্সিটি বোঝা একজন প্রোগ্রামারের জন্য অত্যন্ত গুরুত্বপূর্ণ, কারণ এটি আপনাকে দক্ষ অ্যালগরিদম বেছে নিতে সাহায্য করে। উপরের উদাহরণগুলো (লিনিয়ার সার্চ, বাইনারি সার্চ, বাবল সর্ট, মার্জ সর্ট) দিয়ে আশা করি বিষয়টা পরিষ্কার হয়েছে। যদি আরও কোনো প্রশ্ন থাকে বা কোনো নির্দিষ্ট অ্যালগরিদমের টাইম কমপ্লেক্সিটি বোঝাতে হয়, তাহলে বলো! 😊