Time ve Space Complexity (Zaman ve Hafıza Karmaşıklığı)

Veri Yapıları ve Algoritmalar #1

Time Complexity (Zaman Karmaşıklığı) Nedir?

- Zaman karmaşıklığı, bir algoritmanın çalışma süresini veya işlem sayısını geliştirme veya karşılaştırma amacıyla ölçen bir kavramdır. Bir algoritmanın ne kadar sürede çalıştığı veya ne kadar işlem yaptığı genellikle girdi boyutuna bağlı olarak değişir.

- Bir programın çalışma süresi her zaman sizin vereceğiniz input ile doğru orantılı artmaz. Örneğin 10 büyüklüğünde bir input verdiğinde 1 saniyede çalışıyorsa, 100 büyüklüğünde input verdiğinde 10 saniyede çalışır gibi düşünmemelisiniz.

- Büyük O (Big O) notaşaması zaman karmaşıklığını ifade etmek için yaygın olarak kullanılan bir yöntemdir.

Space Complexity (Hafıza Karmaşıklığı) Nedir?

- Hafıza karmaşıklığı, bir algoritmanın çalıştırılması sırasında ne kadar bellek kullanacağını veya ne kadar hafıza gerektireceğini ölçen bir kavramdır. Bir algoritmanın hafıza karmaşıklığı, genellikle veri yapıları, değişkenler, yardımcı bellek ve diğer unsurlar aracılığıyla kullanılan bellek miktarını ifade eder.

- Hafıza karmaşıklığı, bir algoritmanın girdi boyutu arttığında bellek kullanımının nasıl değiştiğini değerlendirir. Örneğin, bazı algoritmalar sabit miktarda bellek kullanabilirken (sabit hafıza karmaşıklığı), bazıları girdi boyutuyla doğrusal olarak artan bellek kullanımına sahip olabilir (doğrusal hafıza karmaşıklığı).

Big O notasyonu nedir?

- Big O notasyonu, bir algoritmanın zaman veya hafıza karmaşıklığını büyük boyutlu problemlerdeki davranışını tanımlamak için kullanılan bir kavramdır. Algoritmanın performansının, özellikle girdi boyutu büyüdükçe nasıl değiştiğini belirtmek için kullanılır.

- Big O notasyonu, bir algoritmanın en kötü durumda ne kadar zaman veya bellek kullandığını ifade eder. O(n) gibi ifadelerle gösterilir, burada 'n', genellikle girdi boyutunu temsil eder. O notasyonunda, sabit faktörler ve düşük dereceli terimler genellikle ihmal edilir, çünkü Big O notasyonu, algoritmanın büyük girdi boyutlarında nasıl davrandığını anlamak için genel bir bakış sunar.

Best, Average, Worst Case Nedir?
En iyi , ortalama, en kötü durum?

En iyi, ortalama ve en kötü durum, algoritmaların performansını değerlendirmek için kullanılan terimlerdir. Bu terimler genellikle zaman karmaşıklığı veya diğer performans ölçümleriyle ilişkilendirilir.

- **En İyi Durum (Best Case):** Bir algoritmanın, en az kaynak kullanarak veya en hızlı şekilde çalıştığı durumdur. En iyi durum, genellikle algoritmanın belirli bir durumda optimal performans gösterdiği durumu ifade eder. Ancak, genellikle pratikte nadiren karşılaşılan bir durumdur.
- **Ortalama Durum (Average Case):** Bir algoritmanın, farklı girdi verileri üzerinde ortalama olarak ne kadar sürede veya kaynakta çalıştığı durumdur. Ortalama durum, algoritmanın genellikle hangi performansı sergilediğini daha genel bir açıdan ifade eder. Bunun için farklı girdi verileri üzerinde bir ortalama hesaplanır.
- **En Kötü Durum (Worst Case):** Bir algoritmanın, girdi boyutu veya özellikleri belirli bir kritik durumu tetikleyerek en fazla kaynak kullanarak veya en yavaş şekilde çalıştığı durumdur. En kötü durum, algoritmanın en fazla zamanda veya kaynakta ne kadar kötü performans gösterebileceğini belirtir.

Biz genellikle uygulamamızın çalışacağı worst case'i dikkate alırız.

Örnekler

```python
n = int(1e5)
liste = [0] * n

for i in range(n):
    print(liste[i])
```

```python
import math
n = int(1e5)
liste = [0] * n

for i in range(int(math.sqrt(n))):
    print(liste[i])
```

```python
import math
n = int(1e5)
liste = [0] * n

for i in range(int(math.log(n))):
    print(liste[i])
```

```python
n = int(1e5)
liste = [0] * n

for i in range(n):
    for j in range(n):
        print(liste[i]+liste[j])
```

Örnekler

```python
n = int(1e5)
liste = [0] * n

for i in range(n):
    print(liste[i])
```

Time - O(N)

Space - O(N)

```python
import math
n = int(1e5)
liste = [0] * n

for i in range(int(math.sqrt(n))):
    print(liste[i])
```

Time - O(sqrt(

Space - O(N)

```python
import math
n = int(1e5)
liste = [0] * n

for i in range(int(math.log(n))):
    print(liste[i])
```

Time - O(log(N))

Space - O(N)

```python
n = int(1e5)
liste = [0] * n

for i in range(n):
    for j in range(n):
        print(liste[i]+liste[j])
```

Time - O(N^2)

Space - O(N)

Örnekler

```python
n = int(1e5)
while n>0:
    print(n)
    n = n//2
```

```python
n = int(1e5)
liste1 = [10] * n

for i in range(n):
    print(liste1[i])
for i in range(n):
    print(-liste1[i])
```

```python
n = int(1e5)
liste1 = [10] * n
liste2 = [10] * n

for i in range(n):

    for j in range(n):
        print(liste1[i]+liste2[j])
    for j in range(n):
        print(liste1[i]-liste2[j])
```

Örnekler

```python
n = int(1e5)
while n>0:
    print(n)
    n = n//2
```

Time : O(log(N))

Space : O(1)

```python
n = int(1e5)
liste1 = [10] * n

for i in range(n):
    print(liste1[i])
for i in range(n):
    print(-liste1[i])
```

Time: O(N)

Space: O(N)

```python
n = int(1e5)
liste1 = [10] * n
liste2 = [10] * n

for i in range(n):

    for j in range(n):
        print(liste1[i]+liste2[j])
    for j in range(n):
        print(liste1[i]-liste2[j])
```

Time: O(N^2)

Space: O(N)

Örnekler

```python
n = int(4)
root = int(math.sqrt(n)) + 1
bolen_sayisi = 0
for i in range(1,root):
    if(n%i==0):
        bolen_sayisi+=2
        if(n//i == i):
            bolen_sayisi -=1
print(bolen_sayisi)
```

```python
def binary_search(arr, target):
    sol = 0
    sag = len(arr) - 1

    while sol <= sag:
        orta = (sol + sag) // 2

        # Hedef değeri ortada bulduysak
        if arr[orta] == target:
            return orta
        # Eğer hedef değer ortadaki değerden küçükse, sol tarafı kontrol et
        elif arr[orta] > target:
            sag = orta - 1
        # Eğer hedef değer ortadaki değerden büyükse, sağ tarafı kontrol et
        else:
            sol = orta + 1
    # Hedef değeri bulamazsak
    return -1
liste = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
hedef = 10

result = binary_search(liste, hedef)
```

arrayin boyutunu n uzunluğunda düşünün

Örnekler

```python
n = int(4)
root = int(math.sqrt(n)) + 1
bolen_sayisi = 0
for i in range(1,root):
    if(n%i==0):
        bolen_sayisi+=2
        if(n//i == i):
            bolen_sayisi -=1
print(bolen_sayisi)
```

Time: O(sqrt(N))

Space: O(1)

```python
def binary_search(arr, target):
    sol = 0
    sag = len(arr) - 1

    while sol <= sag:
        orta = (sol + sag) // 2

        # Hedef değeri ortada bulduysak
        if arr[orta] == target:
            return orta
        # Eğer hedef değer ortadaki değerden küçükse, sol tarafı kontrol et
        elif arr[orta] > target:
            sag = orta - 1
        # Eğer hedef değer ortadaki değerden büyükse, sağ tarafı kontrol et
        else:
            sol = orta + 1
    # Hedef değeri bulamazsak
    return -1
liste = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
hedef = 10

result = binary_search(liste, hedef)
```

arrayin boyutunu n uzunluğunda düşünün

Time: O(log(N))

Space: O(N)

Örnekler

```
n = int(1e5)
liste = [20] * n

q = int(1e5)
sorgular = [10] * q
for hedef in sorgular:
    result = binary_search(liste, hedef)
```

```
N=100000
liste = [3]* N
liste.sort()
print(liste)
```

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

# Fibonacci dizisinin ilk birkaç elemanını göstermek için örnek kullanım:
n = 10000
print(fibonacci(n))
```

Örnekler

```
n = int(1e5)
liste = [20] * n

q = int(1e5)
sorgular = [10] * q
for hedef in sorgular:
    result = binary_search(liste, hedef)
```

Time: O(Q*log(N))

Space: O(N)

```
N=100000
liste = [3]* N
liste.sort()
print(liste)
```

Time:O(NlogN)

Space: O(N)

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

# Fibonacci dizisinin ilk birkaç elemanını göstermek için örnek kullanım:
n = 10000
print(fibonacci(n))
```

Time: O(2^N)

Space: O(N)

Örnekler

```
n = int(1e5)
liste = [10] * n

q = int(1e5)
sorgular = [10] * q

for sorgu in sorgular:
    if(sorgu not in liste):
        print(sorgu)
```

```
n = int(1e5)
liste = [10] * n

q = int(1e5)
sorgular = [10] * q

for sorgu in sorgular:
    if(sorgu in liste):
        print(sorgu)
```

```
n = int(1e5)
liste = [10] * n

q = int(1e5)
sorgular = [10] * q

for sorgu in sorgular:
    if(liste.count(sorgu)):
        print(sorgu)
```

Örnekler

Time: O(Q*N)

Space: O(max(N,q))

```
n = int(1e5)
liste = [10] * n

q = int(1e5)
sorgular = [10] * q

for sorgu in sorgular:
    if(sorgu not in liste):
        print(sorgu)
```

Time: O(Q*N)

Space: O(max(N,q))

```
n = int(1e5)
liste = [10] * n

q = int(1e5)
sorgular = [10] * q

for sorgu in sorgular:
    if(sorgu in liste):
        print(sorgu)
```

Time: O(Q*N)

Space: O(max(N,q))

```
n = int(1e5)
liste = [10] * n

q = int(1e5)
sorgular = [10] * q

for sorgu in sorgular:
    if(liste.count(sorgu)):
        print(sorgu)
```

Örnekler

```python
n = int(1e5)

for i in range(n):
    for j in range(i,n):
        print(i+j)
```

```python
n = int(1e5)
k = 5
liste = [10] * n
for i in range(n):
    print(liste[i:i+k])
```

```python
n = int(1e5)
k = 5
liste = [10] * n

for j in range(3):
    for i in range(n):
        print(liste[i:i+k])
```

Örnekler

Time: O(N^2)

Space: O(1)

```
n = int(1e5)

for i in range(n):
    for j in range(i,n):
        print(i+j)
```

Time: O(k*N)

Space: O(N)

```
n = int(1e5)
k = 5
liste = [10] * n
for i in range(n):
    print(liste[i:i+k])
```

Time: O(k*N)

Space: O(N)

```
n = int(1e5)
k = 5
liste = [10] * n

for j in range(3):
    for i in range(n):
        print(liste[i:i+k])
```

Bonus Soru

```python
import random
n = int(1e5)
arr = []
for i in range(n):
    num = random.randint(1, int(1e5))
    arr.append(num)
arr.sort()

target = 10
index = binary_search(target)

print(index)
```

Constraints - Kısıtlamalar ve Big-O karşılaştırmaları

input size	required time complexity
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ or $O(n)$
n is large	$O(1)$ or $O(\log n)$