Bölüm 15: Soru Cevap

Algoritmalar

Soru

- Verilen kod parçasının zaman karmaşıklığı nedir?

```
public void foo()
{
  System.out.println("Merhaba");
  System.out.println("Merhaba");
  System.out.println("Merhaba");
}
```

Cevap

- O(1)

Soru

- Verilen kod parçasının zaman karmaşıklığı nedir?

```
public void foo(int n)
{
  for (int i = 1; i <= n; i++) {
    System.out.println("Merhaba");
  }
}
```

Cevap

- O(n)

Soru

- Verilen kod parçasının zaman karmaşıklığı nedir?

```
public void foo(int n)
{
  for (int i = 1; i <= n; i = i * 2) {
    System.out.println("Merhaba");
  }
}
```

Cevap

- O(logn)

Soru

- Verilen kod parçasının zaman karmaşıklığı nedir?

```
public void foo(int[] n)
{
  for (int i = 0; i < n.length; i++) {
    System.out.println(n[i]);
  }
}
```

Cevap

- O(n)

Soru

- Verilen kod parçasının zaman karmaşıklığı nedir?

```
public void foo(int[][] n)
{
  for (int i = 0; i < n.length; i++) {
    for (int j = 0; j < n[0].length; j++) {
      System.out.println(n[i][j]);
    }
  }
}
```

Cevap

- $O(n^2)$
- $O(nm)$

Soru

- Verilen kod parçasının zaman karmaşıklığı nedir?

```
public void foo(int[] n)
{
  for (int i = 0; i < n.length; i++) {
    System.out.println(n[i]);
  }
  for (int i = 0; i < n.length; i++) {
    System.out.println(n[i]);
  }
}
```

Cevap

- O(n)

Soru

- Verilen kod parçasının zaman karmaşıklığı nedir?

```
public void foo(int m, int n) {
  for (int i = 0; i < m; i++) {
    System.out.println(i);
  }
  for (int i = 0; i < n; i++) {
    System.out.println(i);
  }
}
```

Cevap

- O(m+n)

Soru

- Verilen kod parçasının zaman karmaşıklığı nedir?

```
public void foo(int[] n)
{
  for (int i = 0; i < n.length / 2; i++) {
    System.out.println(n[i]);
  }
}
```

Cevap

- O(n)

Soru

- Verilen kod parçasının zaman karmaşıklığı nedir?

```
public int findMax(int[] arr) {
   int max = arr[0];
   for (int i = 1; i < arr.length; i++) {
      if (arr[i] > max) {
         max = arr[i];
      }
   }
   return max;
}
```

Cevap

- O(n)

Soru

- Verilen kod parçasının zaman karmaşıklığı nedir?

```
public boolean containsDuplicates(int[] arr) {
  HashSet<Integer> set = new HashSet<>();
  for (int num : arr) {
    if (set.contains(num)) {
      return true;
    }
    set.add(num);
  }
  return false;
}
```

Cevap

- O(n)
- set.contains(num) fonksiyonuna dikkat edilmeli!

Soru

- Verilen kod parçasının zaman karmaşıklığı nedir?

```
public void findPairsWithSum(int[] arr, int target) {
  for (int i = 0; i < arr.length - 1; i++) {
    for (int j = i + 1; j < arr.length; j++) {
      if (arr[i] + arr[j] == target) {
        System.out.println("Pair found.");
      }
    }
  }
}
```

Cevap

- $O(n^2)$

Soru

- Verilen kod parçasının zaman karmaşıklığı nedir?

```
public int binarySearch(int[] arr, int target) {
   int left = 0;
   int right = arr.length - 1;
   while (left <= right) {
      int mid = left + (right - left) / 2;
      if (arr[mid] == target) { return mid; }
      else if (arr[mid] < target) { left = mid + 1; }
      else { right = mid - 1; }
   }
   return -1; // Element not found
}
```

Cevap

- O(logn)

Soru

- Verilen kod parçasının zaman karmaşıklığı nedir?

```
public double findSquareRoot(int x) {
  if (x == 0 || x == 1) {return x; }
  double left = 0;
  double right = x;
  while (left <= right) {
    double mid = left + (right - left) / 2;
    double square = mid * mid;
    if (square == x) { return mid; }
    else if (square < x) { left = mid + 0.0001; }
    else { right = mid - 0.0001; }
  }
  return right; // Approximate result
}
```

Cevap

- O(logn)

Soru

- Verilen kod parçasının zaman karmaşıklığı nedir?

```
public void mergeSort(int[] arr, int left, int right) {
  if (left < right) {
    int mid = left + (right - left) / 2;
    // Sort first and second halves
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    // Merge the sorted halves
    merge(arr, left, mid, right);
  }
}
```

Cevap

- O(nlogn)
- merge(arr, left, mid, right) fonksiyonuna dikkat edilmeli!

Soru

- Verilen kod parçasının zaman karmaşıklığı nedir?

```
void generateSubsets(int[] nums,int i,int[] subset,int size)
{
  if (i == nums.length) {
    return;
  }
  subset[size] = nums[i];
  generateSubsets(nums, i + 1, subset, size + 1);
  generateSubsets(nums, i + 1, subset, size);
}
```

Cevap

- $O(2^n)$

Soru

- Hızlı Sıralama algoritmasının en kötü durumda zaman karmaşıklığı nedir?

- A) O(n)
- B) O(n log n)
- C) $O(n^2)$
- D) O(log n)

Cevap

- C) $O(n^2)$

- Hızlı Sıralama algoritmasının en kötü durumda zaman karmaşıklığı $O(n^2)$ dir. Bu, algoritmanın en kötü durumda, her seçilen pivot elemanı için $n-1$ elemanın karşılaştırıldığı ve bölündüğü durumu ifade eder.

Soru

- Ortalama durumda arama, ekleme ve silme işlemlerinde O(1) zaman karmaşıklığını elde etmek için hangi veri yapısı kullanılır?

- A) İkili Arama Ağacı
- B) Hash Tablosu
- C) Yığın (Heap)
- D) AVL Ağacı

Cevap

- B) Hash Tablosu

- Ortalama durumda *O(1)* zaman karmaşıklığı elde etmek için hash tablosu kullanılır. Hash tablosu, verileri hızlı bir şekilde aramak, ekleme yapmak ve silmek için kullanılan etkili bir veri yapısıdır. Bu, hash fonksiyonunun iyi bir dağıtım sağladığı ve çakışma (collision) sayısının minimum olduğu durumda geçerlidir.

Soru

- En kötü durumda en iyi zaman karmaşıklığına sahip olan sıralama algoritması hangisidir?

- A) Kabarcık Sıralaması (Bubble Sort)
- B) Birleştirme Sıralaması (Merge Sort)
- C) Hızlı Sıralama (Quick Sort)
- D) Ekleme Sıralaması (Insertion Sort)

Cevap

- B) Birleştirme Sıralaması (Merge Sort)

- Birleştirme Sıralaması (Merge Sort), en iyi en kötü durum zaman karmaşıklığına sahiptir. Birleştirme sıralamasının en kötü durum zaman karmaşıklığı her zaman $O(n \log n)$ olarak sabittir. Bu, algoritmanın her durumda etkili ve hızlı bir şekilde çalışmasını sağlar. Diğer sıralama algoritmaları, özellikle en kötü durumda daha yavaş olabilir veya $O(n^2)$ gibi daha kötü zaman karmaşıklıklarına sahip olabilir.

Soru

- Boyutu n olan sıralı bir dizide bir öğeyi aramak için ikili arama algoritmasının zaman karmaşıklığı nedir?

- A) O(1)
- B) O(log n)
- C) O(n)
- D) O(n log n)

Cevap

- B) O(log n)

- İkili arama algoritması, sıralı bir dizide bir öğeyi aramak için kullanılır ve *O(log n)* zaman karmaşıklığına sahiptir. Burada n, dizinin boyutunu temsil eder. İkili arama algoritması, her adımda diziyi yarıya böler ve hedef öğeyi bulana kadar arama alanını daraltır. Bu nedenle, ikili arama algoritmasının zaman karmaşıklığı, aramanın her adımında veri kümesini yarıya böldüğü için logaritmik olarak büyür.

Soru

- Hızlı Sıralama (Quick Sort) için en iyi durum senaryosunun zaman karmaşıklığı hakkında hangisi doğrudur?

- A) O(n log n)
- B) O(n)
- C) $O(n^2)$
- D) O(log n)

Cevap

- A) O(n log n)

- Hızlı Sıralama (Quick Sort) algoritması için en iyi durum senaryosunda, her bölme işlemi dengeli bir şekilde gerçekleşir ve her seferinde dizinin orta elemanı pivot olarak seçilir. Bu durumda, algoritmanın her adımda veriyi iki eşit parçaya böldüğü ve her parçayı ayrı ayrı sıraladığı kabul edilir. Böylece, her seviyede *O(n)* zaman karmaşıklığı olur ve algoritmanın derinliği *O(logn)* olur.

Soru

- Kabarcık Sıralama (Bubble Sort) algoritmasının en kötü durum zaman karmaşıklığı nedir?

- A) O(n)
- B) O(n log n)
- C) $O(n^2)$
- D) O(log n)

Cevap

- C) $O(n^2)$

- Kabarcık Sıralama (Bubble Sort) algoritmasının en kötü durum zaman karmaşıklığı $O(n^2)$ dir. Bu, algoritmanın her elemanı karşılaştırarak ve gerektiğinde yer değiştirerek sıralama işlemi yapmasından kaynaklanır. Dizinin her elemanı diğer tüm elemanlarla karşılaştırılmalıdır, bu nedenle her bir elemanın sıralanması için $O(n)$ zaman gereklidir. Bu işlem, algoritmanın tüm elemanları sıralayana kadar her bir elemanın $O(n)$ zaman karmaşıklığıyla sıralanması gerektiği anlamına gelir.

Soru

- Merge Sort'in Quick Sort'a göre avantajı nedir?

- A) Merge Sort, yerinde sıralama algoritmasıdır
- B) Merge Sort, daha iyi ortalama durum zaman karmaşıklığına sahiptir
- C) Merge Sort, daha iyi en kötü durum zaman karmaşıklığına sahiptir
- D) Merge Sort, daha iyi yer karmaşıklığına sahiptir

Cevap

- C) Merge Sort, daha iyi en kötü durum zaman karmaşıklığına sahiptir

- Merge Sort ile Quick Sort arasındaki temel fark, en kötü durum zaman karmaşıklığıdır. Merge Sort, her zaman $O(nlogn)$ zaman karmaşıklığına sahipken, Quick Sort'un en kötü durum zaman karmaşıklığı $O(n^2)$ olabilir.

Soru

- Kıyaslama tabanlı olmayan ve *O(n log n)* zaman karmaşıklığına sahip olan sıralama algoritması hangisidir?

- A) Kabarcık Sıralaması (Bubble Sort)
- B) Radix Sıralaması (Radix Sort)
- C) Hızlı Sıralama (Quick Sort)
- D) Birleştirme Sıralaması (Merge Sort)

Cevap

- B) Radix Sıralaması (Radix Sort)

- Radix Sıralaması, kıyaslama tabanlı olmayan bir algoritmadır ve *O(n log n)* zaman karmaşıklığına sahiptir. Diğer seçenekler, kıyaslama tabanlı sıralama algoritmalarıdır. Ancak Radix Sıralaması, her bir sıralama geçişinde sıralanacak elemanların boyutuna bağlı olarak, *O(n log n)* zaman karmaşıklığına sahip olabilir.

Soru

- Sıralı olmayan dizilerde arama yapmak için genellikle hangi arama algoritması kullanılır?

- A) İkili Arama (Binary Search)
- B) Derinlik-Öncelikli Arama (Depth-First Search)
- C) Doğrusal Arama (Linear Search)
- D) Genişlik-Öncelikli Arama (Breadth-First Search)

Cevap

- C) Doğrusal Arama (Linear Search)

- Doğrusal Arama (Linear Search), sıralanmamış bir dizide veya listede belirli bir öğeyi aramak için kullanılır. Bu algoritma, diziyi veya listeyi baştan sona tarar ve aranan öğeyi bulana kadar her bir öğeyi sırayla kontrol eder. Doğrusal Arama, sıralı olmayan veri yapılarında arama yapmak için etkili ve basit bir yöntemdir.

Soru

- Zaman karmaşıklığı açısından, sıralı bir dizide arama yapmak için hangi arama algoritması en verimlidir?

- A) İkili Arama (Binary Search)
- B) Doğrusal Arama (Linear Search)
- C) Derinlik-Öncelikli Arama (Depth-First Search)
- D) Genişlik-Öncelikli Arama (Breadth-First Search)

Cevap

- A) İkili Arama (Binary Search)

- İkili Arama (Binary Search) algoritması, sıralı bir dizide bir öğeyi bulmak için en verimli algoritmadır. İkili arama, her adımda aranan öğenin dizinin ortasına bakarak arama alanını yarıya böler. Bu nedenle, sıralı bir dizide arama yapmak için İkili Arama algoritması, *O(log n)* zaman karmaşıklığına sahiptir.

Soru

- f(n) = O(n) ve g(n) = O(nlogn) olsun. f(n) + g(n) = ?

- a) Θ(1)
- b) Θ(n)
- c) Θ(n log n)
- d) Θ(n^2)

Cevap

- Cevap C

- $f(n) \leq c_1 \cdot n$
- $g(n) \leq c_2 \cdot n\log n$
- $f(n) + g(n) \leq c_1 \cdot n + c_2 \cdot n\log n$
- $f(n) + g(n) = O(n\log n)$

Soru

- Aşağıdakilerden hangisi 1 + 1/2 + 1/3 + · · · + 1/n toplamının büyümesini en iyi tanımlar?

- a) Θ(1/n)
- b) Θ(1)
- c) Θ(log n)
- d) Θ(n log n)

Cevap

- Cevap: C

- Verilen toplam, harmonik seri olarak bilinir ve logn büyüme hızına sahiptir. Bu yüzden, 1+1/2+1/3+⋯+1/n ifadesinin büyümesi Θ(logn) ile tanımlanır.

Soru

- Başlangıçta boş olan bir yığına n eleman eklemek için en kötü durum çalışma süresi T(n) nedir?

- a) $T(n) = O(n^2)$.
- b) $T(n) = O(\sqrt{n}\log n)$.
- c) $T(n) = O(n)$.
- d) $T(n) = O(n\log n)$.

Cevap

- Cevap: D

- Yığına eleman eklenmesi sırasında yığının yüksekliği kadar karşılaştırma ve yer değiştirme gerekir.
- Yığının yüksekliği, eleman sayısının logaritması ile orantılıdır, O(logn).
- n eleman eklemek için bu işlem n kere tekrarlanır.

Soru

- n boyutunda bir ikili yığından (binary heap) bir elemanı çıkarmak için en kötü durum çalışma süresi T(n) nedir?

- a) T(n) = O(1).
- b) T(n) = O(log n).
- c) T(n) = O(n).
- d) T(n) = O(n log n).

Cevap

- Cevap: B

- İkili yığından (binary heap) eleman çıkarmak (pop) iki adımdan oluşur: İkili yığının kök düğümünü çıkarma işlemi O(1) zaman alır, çünkü bu adım yalnızca en üstteki elemanın alınmasını içerir. Yığının özelliğini (heap property) korumak için kök düğümden itibaren yığının yeniden düzenlenmesi gerekir. Bu işlem yığının yüksekliği kadar zaman alır. Bir ikili yığının yüksekliği *O*(logn) seviyesindedir.

Soru

- Başlangıçta boş olan bir ikili arama ağacına (binary search tree) n eleman eklemek için en kötü durum çalışma süresi T(n) nedir?

- a) $T(n) = O(n^2)$.
- b) $T(n) = O(\sqrt{n} \log n)$.
- c) $T(n) = O(n)$.
- d) $T(n) = O(n \log n)$.

Cevap

- Cevap: A

- En kötü durum, elemanların ağaca sıralı bir şekilde (ya tamamen artan ya da tamamen azalan) eklenmesi durumunda ortaya çıkar. Bu durumda, ağaç dengeli bir şekilde büyümez. İkili arama ağacı bir bağlı listeye (linked list) dönüşür. İlk elemanı eklemek O(1) zaman alır. İkinci elemanı eklemek O(1) zaman alır. Üçüncü elemanı eklemek O(2) zaman alır. n'inci elemanı eklemek O(n-1) zaman alır.
- $T(n) = 1+2+3+...+(n-1)$
- $T(n) = O(n^2)$.

Soru

- Bir tam sayı sıralama algoritmasının en iyi durum çalışma süresinin Θ(n log n) olduğunu varsayalım. Algoritmanın en iyi durum çalışma süresini Θ(n) yapacak değişikliği açıklayın.

Cevap

- Mevcut algoritmalar en iyi durum senaryosunda bile Θ(nlogn) zaman karmaşıklığına sahiptir. Örneğin; quicksort veya mergesort gibi. En iyi durum senaryosunu Θ(n) zaman karmaşıklığına indirgemek için yapılabilecek bir değişiklik, verinin zaten sıralı olduğunu kontrol etmek için bir ön kontrol eklemektir.

SON