3.QUİZ ÇALIŞMA SORULARI CEVAPLARI

1) Given a sorted array of distinct integers A[1, . . . , n], you want to find out whether there is

an index i for which A[i] = i.

a) Write a brute force algorithm. Specify time complexity.

b) Give a divide-and-conquer algorithm that runs in time O(log n). Prove that the time complexity is in O(logn).

A) BruteForce(A):

for i from 1 to n:

if A[i] == i:

return True

return False

B)

DivideAndConquer(A, start, end):

if start > end:

return False

mid = (start + end) // 2

if A[mid] == mid:

return True

elif A[mid] > mid:

return DivideAndConquer(A, start, mid - 1)

else:

return DivideAndConquer(A, mid + 1, end)

**2) a)** Write a brute force algorithm to find the maximum difference between any two elements of an n-dimensional array consisting of n numbers. Specify time complexity.

**b)** Write a divide and conquer algorithm that finds the maximum difference between any two

elements of an n-dimensional array consisting of n numbers in O(n) time. Prove that the time complexity is in O(n).

**A)**

BruteForceMaxDifference(arr):

max\_diff = -INFINITY

n = length(arr)

for i from 1 to n:

for j from i + 1 to n:

diff = abs(arr[j] - arr[i])

max\_diff = max(max\_diff, diff)

return max\_diff

Bu algoritmanın zaman karmaşıklığı O(n^2)'dir; burada n, dizinin boyutudur. Bunun nedeni, en kötü durumda dizideki tüm olası öğe çiftlerini yinelememizdir.

**B)**

DivideAndConquerMaxDifference(arr, start, end):

if start == end:

return 0

mid = (start + end) // 2

max\_left = DivideAndConquerMaxDifference(arr, start, mid)

max\_right = DivideAndConquerMaxDifference(arr, mid + 1, end)

min\_left = min(arr[start:mid+1])

max\_right = max(arr[mid+1:end+1])

return max(max\_left, max\_right, max\_right - min\_left)

Bu algoritma, diziyi yinelemeli olarak yarılara böler ve her yarıdaki öğeler arasındaki maksimum farkı bulur. Daha sonra genel maksimum farkı bulmak için sol yarı, sağ yarı ve ortadaki maksimum farkları karşılaştırır. Bu algoritmanın zaman karmaşıklığı O(n)'dir çünkü her özyineleme seviyesinde dizinin tüm elemanlarını bir kez işleriz.

**3)**

A k-wise merge takes as input k sorted arrays, and constructs a single sorted array containing all of the elements of the input arrays. Describe an efficient divide and conquer algorithm MultiMerge(k, A1, ..., Ak) which computes a k-wise merge of its input arrays. What is the run time of your algorithm with input of k arrays, each of length n.

MultiMerge(k, A1, ..., Ak):

if k == 1:

return A1

mid = k / 2

left\_half = MultiMerge(mid, A1, ..., A(mid))

right\_half = MultiMerge(k - mid, A(mid+1), ..., Ak)

return Merge(left\_half, right\_half)

Merge(arr1, arr2):

merged = []

i = 0

j = 0

while i < length(arr1) and j < length(arr2):

if arr1[i] < arr2[j]:

merged.append(arr1[i])

i += 1

else:

merged.append(arr2[j])

j += 1

while i < length(arr1):

merged.append(arr1[i])

i += 1

while j < length(arr2):

merged.append(arr2[j])

j += 1

return merged

Bu algoritma, her bölümde yalnızca bir dizi kalana kadar k giriş dizisini yinelemeli olarak yarıya böler. Daha sonra, tek bir sıralanmış dizi elde edilene kadar Birleştirme işlevini kullanarak bitişik dizi çiftlerini birleştirir. Bu algoritmanın her biri n uzunlukta olan k dizi girişiyle zaman karmaşıklığı O(n log k)'dir. Bunun nedeni, özyinelemenin her seviyesinde k-yollu bir birleştirme işlemi gerçekleştirmemiz ve birleştirme işleminin O(n) zaman almasıdır; burada n, her dizinin uzunluğudur. Log k düzeyinde özyineleme gerçekleştiriyoruz ve sonuçta O(n log k) toplam zaman karmaşıklığı ortaya çıkıyor.

**4)** a) Given two integers x and n, write a function to compute xn in O(n) time.

**b)**Given two integers x and n, write a function to compute xn in O(logn) time.

**A)**

PowerLinear(x, n):

result = 1

for i from 1 to n:

result = result \* x

return result

**B)**

PowerLogarithmic(x, n):

if n == 0:

return 1

temp = PowerLogarithmic(x, n / 2)

if n is even:

return temp \* temp

else:

return x \* temp \* temp

**5)** You are given two sorted lists of size m and n. Give an O(log m + log n) time algorithm for computing the kth smallest element in the union of the two lists.

FindKthSmallest(arr1, m, arr2, n, k):

if m > n:

return FindKthSmallest(arr2, n, arr1, m, k)

if m == 0:

return arr2[k - 1]

if k == 1:

return min(arr1[0], arr2[0])

idx1 = min(k // 2, m) // 2

idx2 = k - idx1 - 1

if arr1[idx1] < arr2[idx2]:

return FindKthSmallest(arr1[idx1:], m - idx1, arr2, n, k - idx1)

else:

return FindKthSmallest(arr1, m, arr2[idx2:], n - idx2, k - idx2)

**6)** Given a sequence of n elements where each element is an integer in [1,k], return the majority element (an element that appears more than n/2 times) or zero if no majority element is found. Give a divide-and-conquer algorithm that runs in time Θ (nlog n).

FindMajorityElement(arr, start, end):

if start == end:

return arr[start]

mid = (start + end) / 2

left\_majority = FindMajorityElement(arr, start, mid)

right\_majority = FindMajorityElement(arr, mid + 1, end)

if left\_majority == right\_majority:

return left\_majority

left\_count = CountOccurrence(arr, start, end, left\_majority)

right\_count = CountOccurrence(arr, start, end, right\_majority)

if left\_count > (end - start + 1) / 2:

return left\_majority

elif right\_count > (end - start + 1) / 2:

return right\_majority

else:

return 0

CountOccurrence(arr, start, end, num):

count = 0

for i from start to end:

if arr[i] == num:

count += 1

return count

algoritma, 1 boyutunda alt dizilere ulaşana kadar diziyi yinelemeli olarak yarılara böler. Daha sonra çoğunluk öğesini bulmak için sol ve sağ yarılardan elde edilen sonuçları birleştirir. Bu algoritmanın genel zaman karmaşıklığı Θ(n log n)'dir.

**7)** a) (1D closest pair problem) Given n points in 1-dimension, find two whose mutual distance is smallest. Write a divide and conquer algorithm. Specify time complexity

b) (2D closest pair problem) Given n points in 2-dimensions, find two whose mutual distance

is smallest. Write a divide and conquer algorithm. Specify time complexity

**A)** ClosestPair1D(points):

Sort points in non-decreasing order

min\_distance = INFINITY

closest\_pair = None

for i from 1 to length(points) - 1:

distance = points[i] - points[i-1]

if distance < min\_distance:

min\_distance = distance

closest\_pair = (points[i-1], points[i])

return closest\_pair

Bu algoritma, noktaları azalmayacak şekilde sıralar ve ardından karşılıklı en küçük mesafeye sahip nokta çiftini bulmak için sıralanan listeyi yineler. Bu algoritmanın zaman karmaşıklığı, sıralama adımından dolayı O(n log n)'dir.

**B)** ClosestPair2D(points):

Sort points by x-coordinate

Split the points into two halves, left and right

left\_pair = ClosestPair2D(left\_half)

right\_pair = ClosestPair2D(right\_half)

min\_distance = min(distance(left\_pair), distance(right\_pair))

middle\_line = points[length(left\_half) - 1].x-coordinate

Create a list strip[] containing points within min\_distance distance from middle\_line

Sort strip[] by y-coordinate

for i from 0 to length(strip) - 1:

for j from i + 1 to min(i + 7, length(strip) - 1):

distance = distance(strip[i], strip[j])

if distance < min\_distance:

min\_distance = distance

return min(min\_distance, distance(left\_pair), distance(right\_pair))

Bu algoritma noktaları x koordinatlarına göre sıralıyor ve ardından bunları iki yarıya bölüyor. Her iki yarıdaki en yakın çiftleri yinelemeli olarak bulur ve ardından orta çizginin etrafındaki nokta şeritlerini kontrol eder. Ayrıca sol yarı, sağ yarı ve şeritten olan minimum mesafeyi birleştirir. Bu algoritmanın zaman karmaşıklığı, sıralama adımı ve özyinelemeli böl ve yönet yaklaşımı nedeniyle O(n log n)'dir. Şeritteki mesafe hesaplaması O(n) zaman alır ancak şerit sınırlı sayıda nokta içerdiğinden genel zaman karmaşıklığına hakim değildir.

**8)**Atama problem için bir greedy algoritma tasarlayın greedy algoritma her zaman en optimum çözümü sağlar mı ?

Atama problem için greedy algoritma şöyle tasarlanır. Öncelikle boş bir atama listesi başlatır,ardından her sayı için aracıyı kalan elemanlar arasından en az maliyetli olanıyla atarız.tüm aracılara atama yapılana kadar işleme devam edilir. Greedy algoritma her zaman en optimal çözümü garanti edemez. Çünkü algoritma aç gözlü yaklaşımla sadece o andaki en optimal çözümü bulabilir. Atama işlemini her adımda yerel optimum olarak yapan bu algoritma genel optimum seviyesine çıkmayabilir. Çünkü sadece o adımdaki en optimal çözüme odaklanmıştır , bu atama genel optimum seviyenin üzerinde olabilir.

**9)** Design an algorithm for finding a maximum spanning tree–a spanning tree with the largest possible

edge weight–of a weighted connected graph.

**Answer)**

MaxSpanningTree(Graph G):

Initialize an empty MST

Initialize a set of visited vertices to keep track of vertices already in the MST

Add any vertex to the MST

While MST does not include all vertices:

Select a vertex v not in the MST with the maximum weight edge connecting it to a vertex in the MST

Add v to the MST

Add the edge with maximum weight to the MST

Return the MST

**10)** (Minimum Coin Change Problem) Given a set of coins and a value, find the minimum number of coins which satisfies the value. Write a greedy algorithm. Does your greedy algorithm always yield an optimal solution? Discuss.

For example: coins[] = {5,10,20,25}, value = 50.

Solutions:

{5 \* 10} = 50 [10 coins]

{5 \* 8 + 10 \* 1} = 50 [9 coins] …

Best solution{25 \* 2} = 50 [2 coins]

**MinimumCoinChange(coins[], value):**

**Sort coins[] in non-decreasing order**

**Initialize a counter to 0**

**While value is not zero:**

**Find the largest coin denomination c such that c <= value**

**Subtract c from value**

**Increment the counter by 1**

**Return the counter**

Açgözlü algoritma her zaman en uygun çözümü vermeyebilir. Madeni para birimlerinin setine bağlıdır. Madeni para birimleri, her madeni para biriminin önceki değerin tam katı olduğu "kanonik madeni para sistemi" olarak bilinen bir madeni para sistemi oluşturursa, o zaman açgözlü algoritma her zaman en uygun çözümü üretecektir. Kanonik madeni para sistemlerinin örnekleri arasında ABD madeni para sistemi (1, 5, 10, 25) ve İngiliz madeni para sistemi (1, 2, 5, 10, 20, 50, ...) yer alır. Ancak madeni para birimleri kanonik bir madeni para sistemi oluşturmuyorsa açgözlü algoritma optimal bir çözüm üretemeyebilir. Bu gibi durumlarda, en uygun çözümü garanti etmek için dinamik programlama kullanan madeni para değiştirme problemi gibi dinamik programlama algoritmaları kullanılmalıdır.

Coin[] = {5, 10, 20, 25} ve değer = 50 ile verilen örnekte, açgözlü algoritma optimal olan {25 \* 2} = 50 [2 coin] çözümünü verebilir. Ancak diğer madeni para birimleri için durum her zaman böyle olmayabilir.

**11)** Suppose we are given n ropes of different lengths, and we want to tie these ropes into a single rope. The cost to connect two ropes is equal to sum of their lengths. We want to connect all the ropes with the minimum cost.

For example, suppose we have 4 ropes of lengths 7, 3, 5, and 1. One (not optimal!) solution would be

to combine the 7 and 3 rope for a rope of size 10, then combine this new size 10 rope with the size 5 rope for a rope of size 15, then combine the rope of size 15 with the rope of size 1 for a final rope of size 16. The total cost would be 10 + 15 + 16 = 41. (Note: the optimal cost for this problem is 29. How might you combine the ropes for that cost?) Find a greedy algorithm for the minimum cost and prove the correctness of your algorithm.

Diyelim ki bize farklı uzunluklarda n tane ip veriliyor ve biz bu ipleri tek bir ip halinde bağlamak istiyoruz. İki halatı bağlamanın maliyeti uzunluklarının toplamına eşittir. Tüm halatları minimum maliyetle bağlamak istiyoruz.

Örneğin, uzunlukları 7, 3, 5 ve 1 olan 4 ipimiz olduğunu varsayalım. Optimal olmayan çözüm şu olabilir:7 ve 3 numaralı ipi birleştirip 10’u, ardından bu ipi 5 ile birleştirerek 15 ,son olarak 1ile birleştirerek toplam maliyeti 16 olan ip bulabiliriz. Toplam maliyet 10 + 15 + 16 = 41 olacaktır. (Not: Bu problem için en uygun maliyet 29'dur. Bu maliyet için ipleri nasıl birleştirebilirsiniz?) Minimum maliyet için açgözlü bir algoritma bulun ve algoritmanızın doğruluğunu kanıtlayın.

**MinimumCostToConnectRopes(ropeLengths):**

**Initialize a priority queue (min-heap) to store rope lengths**

**Add all rope lengths to the priority queue**

**totalCost = 0**

**while size of priority queue > 1:**

**shortestRope1 = pop shortest rope from priority queue**

**shortestRope2 = pop shortest rope from priority queue**

**combinedLength = shortestRope1 + shortestRope2**

**totalCost += combinedLength**

**push combinedLength back into priority queue**

**return totalCost**

**İSPAT:**

Her adımda birleştirilecek en kısa iki ipi seçiyoruz. Bu seçim en uygunudur çünkü en kısa iki halatın birleştirilmesi mevcut işlemin maliyetini en aza indirir ve gelecekteki kombinasyonlar için daha uzun halatlar bırakır. Daima en kısa iki halatı birleştirerek, tüm halatları bağlamanın toplam maliyetinin en aza indirilmesini sağlıyoruz. Bu problemde her ip yalnızca bir kez birleştirilebildiğinden üst üste binme alt problemi yoktur. Bu nedenle açgözlü algoritmayı uygularken örtüşen alt problemleri dikkate almamıza gerek yoktur. Bundan ötürü, tasarlanan açgözlü algoritmanın tüm halatları bağlamak için her zaman minimum maliyeti sağlayacağı sonucuna varabiliriz.

**12)** A graph coloring problem is a coloring of the graph vertices s.t. no pair of adjacent vertices share

the same color. The chromatic number χ(G) of a graph G is the smallest number of colors needed to

color the graph. Write a greedy algorithm to find the minumum number of colors. Does your

greedy algorithm always yield an optimal solution? Discuss.

Bir graf boyama problemi, bitişik köşelerin aynı rengi paylaşmaması koşuluyla grafin köşelerinin bir renklendirmesidir. Bir grafın kromatik sayısı χ(G), grafi renklendirmek için gereken en küçük renk sayısıdır. Minimum renk sayısını bulmak için bir açgözlü algoritma yazın. Açgözlü algoritmanız her zaman optimal bir çözüm mü verir? Tartışın.

Bu kod, yukarıdaki örnekte verilen grafBoya fonksiyonunun özelleştirilmiş halidir. Fonksiyon, bir Graf yapısını ve köşelerin renklerini içeren bir int dizisini döndürür.

Fonksiyonun işleyişi şu şekildedir:

1. **Köşelere Renk Ataması:**
   * renkler adında her köşe için -1 (renk atanmamış) değeri içeren bir dizi oluşturulur.
   * İlk köşeye 0 rengi atanır.
2. **Kalan Köşeleri Boyama:**
   * Her bir köşe (i) için:
     + kullanilabilirRenkler adında her renk için 1 değeri içeren bir dizi oluşturulur.
     + i köşesinin bitişik köşelerinin renklerini kontrol ederek kullanilabilirRenkler dizisinde kullanılabilecek renkleri 0 olarak işaretlenir.
     + kullanilabilirRenkler dizisindeki ilk 1 değeri (renk) seçilir.
     + Seçilen renk (renk), i köşesine atanır.
3. **Sonuç Dönüşü:**
   * Boyanan köşelerin renklerini içeren renkler dizisi fonksiyonun sonucu olarak döndürülür.

int \*grafBoya(Graf \*graf) {

int numVertices = graf->numVertices;

// Her köşeye -1 (renk atanmamış) değeri atanır

int \*renkler = malloc(numVertices \* sizeof(int));

for (int i = 0; i < numVertices; i++) {

renkler[i] = -1; }

// İlk köşeye ilk renk (0) atanır

renkler[0] = 0;

// Kalan köşelerde gezdirilerek boyama işlemi gerçekleştirilir

for (int i = 1; i < numVertices; i++) {

// Kullanılabilir renklerin listesini oluşturma

int kullanilabilirRenkler[numVertices];

for (int j = 0; j < numVertices; j++) {

kullanilabilirRenkler[j] = 1; }

// Bitişik köşelerin renklerini kontrol ederek kullanılabilir renkleri filtreleme

for (int j = 0; j < numVertices; j++) {

if (graf->adjMatrix[i][j] == 1 && renkler[j] != -1) {

kullanilabilirRenkler[renkler[j]] = 0; } }

// Kullanılabilir renklerden ilkini seçme

int renk = -1;

for (int j = 0; j < numVertices; j++) {

if (kullanilabilirRenkler[j] == 1) {

renk = j;

break;} }

// Seçilen rengi i. köşeye atama

renkler[i] = renk; }

return renkler; }  
**1. Yerel Optimizasyon:**

Fonksiyon, her köşe için en uygun rengi seçmeye odaklanır, ancak gelecekteki renk atamalarının genel etkilerini göz önünde bulundurmaz. Bu, bazı durumlarda, daha az renk kullanarak grafi renklendirmenin mümkün olduğu senaryolara yol açabilir.

**2. Renk Seçimi Stratejisi:**

Fonksiyon, kullanılabilir renklerden ilk 1 değeri seçer. Bu, her zaman en iyi seçeneği sunmayabilir. Bazı durumlarda, farklı bir renk seçimi daha az renk kullanarak grafi renklendirmeye imkan sağlayabilir.

**3. Graf Yapısı:**

Bazı graflar, doğaları gereği minimum renk sayısıyla renklendirmek için özel bir renklendirme sırası gerektirir. grafBoya fonksiyonu bu tür graflara özel bir yaklaşım sunmaz ve bu nedenle optimal bir çözüm elde edemeyebilir.