



**Adı – Soyadı – Numarası:**

**Soru 1:** Aşağıda verilen fonksiyonun en kötü durum algoritma karmaşıklığı nedir? Fonksiyon farklı boyutlarda ve rastgele elemanlar ile oluşturulan diziler ile tekrar tekrar çalıştırıldığında yürütme zamanı nasıl etkilenir?

```
int ara(int[] dizi, int hedef) {  
    for (int i = 0; i < dizi.length; i++) {  
        if (dizi[i] == hedef) {  
            return i;  
        }  
    }  
    return -1;  
}
```

Verilen fonksiyon, dizide belirli bir hedef değeri arayan doğrusal arama (linear search) algoritmasıdır. Fonksiyonun en kötü durum karmaşıklığını belirlemek için, en uzun süren yürütme senaryosunu düşünmek gerekir. En kötü senaryo hedef değerinin dizide bulunmamasıdır. Bu durumda, fonksiyon dizinin tüm elemanlarını tek tek kontrol edecektir. Dizinin boyutu arttıkça, fonksiyonun çalışma süresi de doğrusal olarak artacaktır. Bu da asimptotik olarak  $O(n)$  karmaşıklığına eşdeğerdir.

Rastgele elemanlarla oluşturulan dizilerde hedef değerinin konumu değişken olacağından, fonksiyonun çalışma süresi de değişken olacaktır.

**Soru 2:** Aşağıda verilen fonksiyonun en kötü durum algoritma karmaşıklığı nedir? Fonksiyon farklı boyutlarda diziler ile tekrar tekrar çalıştırıldığında yürütme zamanı nasıl etkilenir?

```
void doldur(int[] dizi) {  
    for (int i = 0; i < dizi.length; i++) {  
        dizi[i] = random.nextInt(100);  
    }  
}
```

Verilen fonksiyon, bir diziyi rastgele 0 ile 99 arasındaki sayılarla doldurur. Bu fonksiyonun en kötü durumu ve ortalama durumu aynıdır çünkü her durumda dizinin tüm elemanlarına tek tek erişilir. Döngü, dizinin tüm elemanlarını baştan sona dolaşır. Bu döngünün karmaşıklığı  $O(n)$ 'dir,  $n$  dizinin uzunluğudur.

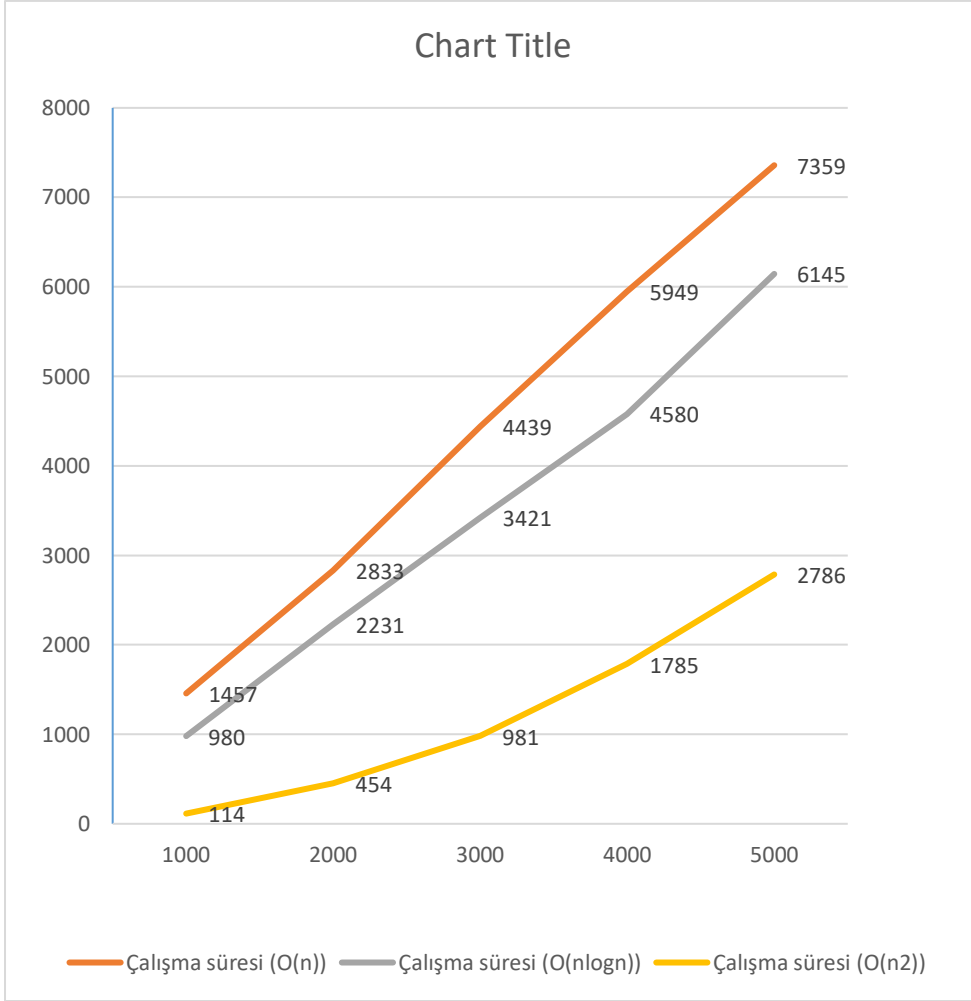
Dizinin boyutu arttıkça, fonksiyonun çalışma süresi de doğrusal olarak artar. Yani, dizinin boyutu iki katına çıkarsa, fonksiyonun yürütme süresi de yaklaşık iki katına çıkar. Bu,  $O(n)$  karmaşıklığının bir sonucudur.

**Soru 3:** Aşağıda girdi boyutlarına bağlı olarak farklı algoritma karmaşıklıklarına sahip fonksiyonların yürütme süreleri verilmiştir. Boş olan hücreleri yaklaşık değerler ile doldurunuz. Grafiğini çiziniz.

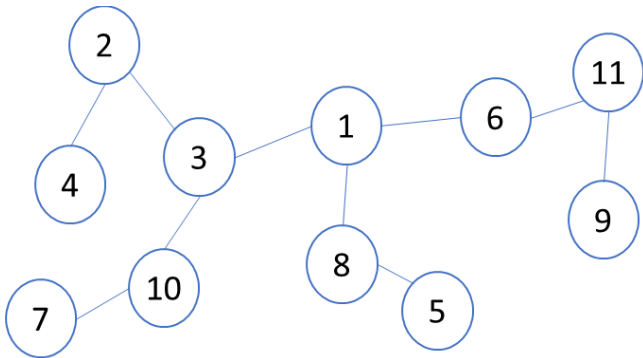
Girdi boyutu	Çalışma süresi ( $O(n)$ )	Çalışma süresi ( $O(n \log n)$ )	Çalışma süresi ( $O(n^2)$ )
1000	1457	980	114
2000	2833	2231	454
3000	4439	3421	981
4000	5949	4580	1785



5000	7359	6145	2786
------	------	------	------



**Soru 4:** Aşağıda verilen çizge 1 numaralı düğümden başlayarak derinlik öncelikli arama (DFS) ile gezildiğinde ziyaret edilen düğümleri sırasıyla yazınız. (gerekli ise numarası küçük olan düğüm önceliklidir)



1 → 3 → 2 → 4 → 10 → 7 → 6 → 11 → 9 → 8 → 5

**Soru 5:** Dijkstra algoritması bir kaynak düğümden diğer tüm düğümlere olan en kısa yolu bulur. Ancak, sadece pozitif ağırlıklı kenarlardan oluşan çizgelerde çalışır, negatif ağırlıklı kenarlar üzerinde çalışmaz.



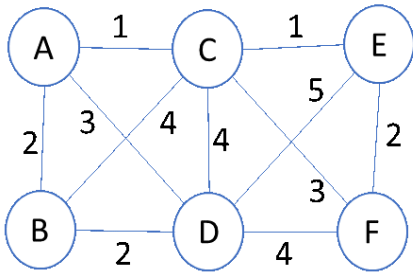
Bellman-Ford algoritması, Dijkstra algoritmasından farklı olarak negatif ağırlıklı çizgeler üzerinde de çalışır. Bu durum algoritma karmaşıklığını nasıl etkiler?

Bellman-Ford algoritmasının negatif ağırlıklı kenarlar üzerinde çalışabilme yeteneği, onun Dijkstra algoritmasından farklı ve daha yüksek bir zaman karmaşıklığına sahip olmasına neden olur. Bellman-Ford algoritması her düğüm için tüm kenarları  $V-1$  kez tarar, bu da karmaşıklığının  $O(VE)$  olmasına neden olur.

Dijkstra:  $O((V+E)\log V)$

Bellman-Ford:  $O(VE)$

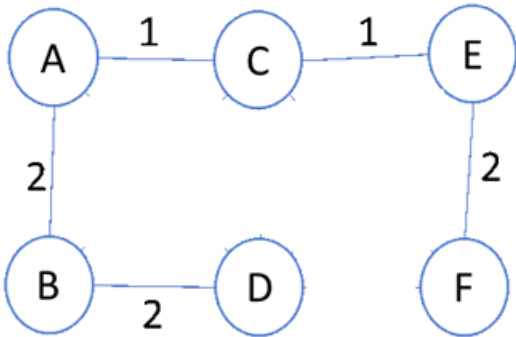
**Soru 6:** Prim algoritması bir çizgede tüm düğümleri birbirine bağlayan en kısa ağırlıklı ağacı oluşturur. Algoritmayı aşağıda verilen çizge üzerinde A düğümünden başlatarak çalıştırınız. Eklenen kenarları sırayla yazınız. Bulduğunuz minimum kapsayan ağacı (MST) çiziniz.



Aynı ağırlığa sahip kenarlar olduğunda kenarların MST'ye eklenme sıraları değişebilir. Bu örnek için sonuçta oluşan MST değişmeyecektir.

Kruskal algoritmasından farklı olarak, A – C kenarı eklenmeden C – E kenarı eklenemez. C – E kenarı eklenmeden E – F kenarı eklenemez. A – B kenarı eklenmeden B – D kenarı eklenemez.

A – C, C – E, A – B, B – D, E – F, D – F



**Soru 7:** Herhangi bir çizgede tüm kenarların ağırlıkları 1 arttırılırsa herhangi iki düğüm arasındaki en kısa yol değişir mi? Açıklayınız.

Herhangi bir çizgede tüm kenarların ağırlıkları 1 arttırılırsa minimum kapsayan ağaç (MST) değişmez. Çizgedeki tüm kenar ağırlıklarının 1 artırılması, her kenar ağırlığının aynı sabit miktarda artması anlamına gelir. Örneğin, başlangıçta bir kenarın ağırlığı  $w$  ise, ağırlığı  $w+1$  olacaktır. MST'yi oluştururken önemli olan,



iki düğüm arasındaki nispi ağırlık farklarıdır, yani hangi kenarın diğerlerinden daha hafif olduğu önemlidir. Tüm kenar ağırlıkları aynı miktarda artırıldığında, kenarların nispi ağırlık farkları değişmez.

**Soru 8:** Kruskal algoritmasının karmaşıklığı  $O(E \log E)$ 'dir. Algoritmanın hangi adımı bu duruma sebep olur?

Kruskal algoritmasının zaman karmaşıklığının  $O(E \log E)$  olmasının nedeni, algoritmanın kenarları ağırlıklarına göre sıralama adımıdır. Kullanılan sıralama algoritmasına göre Kruskal algoritmasının zaman karmaşıklığı değişir. Kenarlar sıralandıktan sonra, kenarlar teker teker işlenerek döngü oluşturup oluşturmadıkları kontrol edilir. Bu adımda kullanılan Birleştir-Bul (Union-Find) veri yapısı, path compression (yol sıkıştırma) ve union by rank (seviyeye göre birleştirme) gibi teknikler kullanılarak optimize edildiğinde, her bir işlem neredeyse sabit zaman alır, amortize edilmiş zaman karmaşıklığı  $O(\alpha(n))$  olur. Burada  $\alpha$ , çok yavaş büyüyen ters Ackermann fonksiyonudur ve pratikte sabit olarak kabul edilir.

**Soru 9:** Yönlü bir çizgede döngü olup olmadığı nasıl bulunur?

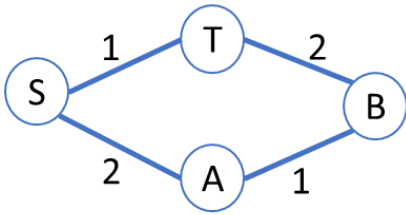
DFS, yönlü bir çizgede döngü tespiti için kullanılabilir. DFS sırasında, bir düğüm yeniden ziyaret edilirse (yani, bir düğüm DFS çağrısı yığınınına tekrar girerse), bu bir döngünün varlığını gösterir.

**Soru 10:** Bir çizgenin birden fazla minimum kapsayan ağacı olabilir mi? Açıklayınız.

Evet, bir çizgenin birden fazla minimum kapsayan ağacı (MST) olabilir. Eğer çizgede birden fazla kenar aynı ağırlığa sahipse, bu kenarlar arasında seçim yaparken farklı kombinasyonlar mümkün olabilir ve bu da farklı MST'lerin oluşmasına yol açabilir. Aşağıdaki örnek için;

MST 1: S – T, A – B, T – B

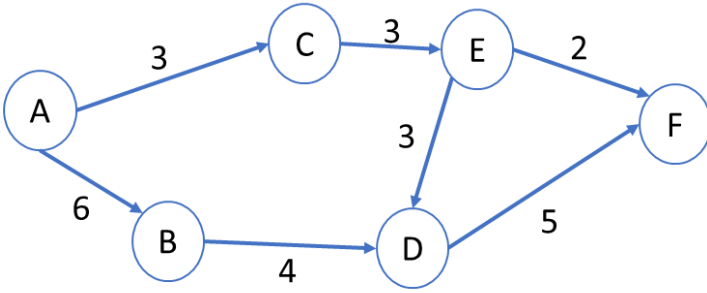
MST 2: S – T, A – B, S – A



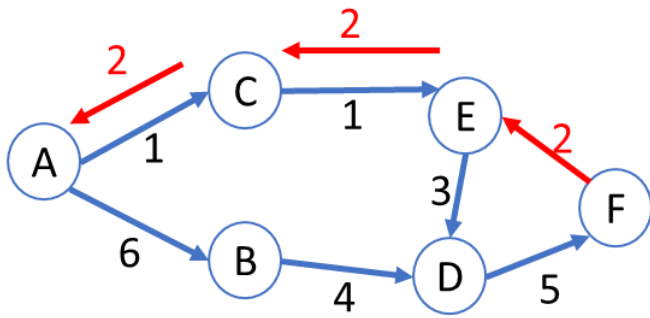
**Soru 11:** Maksimum akış bulma probleminde artık yolları (augmenting paths) bulmak için Ford-Fulkerson algoritması DFS, Edmonds-Karp algoritması BFS yaklaşımını kullanır. Bu iki farklı yaklaşım nasıl bir sonuca neden olur?

Maksimum akış problemini çözmek için kullanılan Ford-Fulkerson ve Edmonds-Karp algoritmaları, artık yolları (augmenting paths) bulmak için farklı arama stratejileri kullanır: Ford-Fulkerson derinlik öncelikli arama (DFS) kullanırken, Edmonds-Karp genişlik öncelikli arama (BFS) kullanır. DFS, rastgele ve derinlemesine seçimler yaparak artık yolları bulur. Bu nedenle, seçilen yollar her zaman en kısa veya en etkili yollar olmayabilir. Bu, algoritmanın daha fazla adımda sonuca ulaşmasına neden olabilir. BFS, kaynak düğümden başlayarak en kısa artık yolu (en az sayıda kenar içeren) bulur. BFS ile en kısa yollar bulunarak yapılan artırımlar, genel olarak daha az adımda maksimum akışa ulaşmayı sağlar.

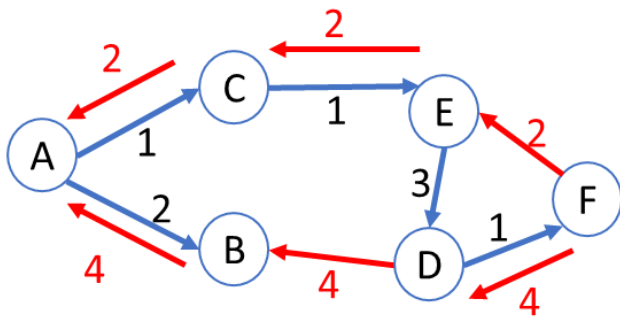
**Soru 12:** Aşağıda verilen çizgede seçtiğiniz bir algoritmaya göre maksimum akışı bulunuz. Her bir adımda bulduğunuz yolu ve kapasitesini yazınız ve çizgenin güncel durumunu çiziniz.



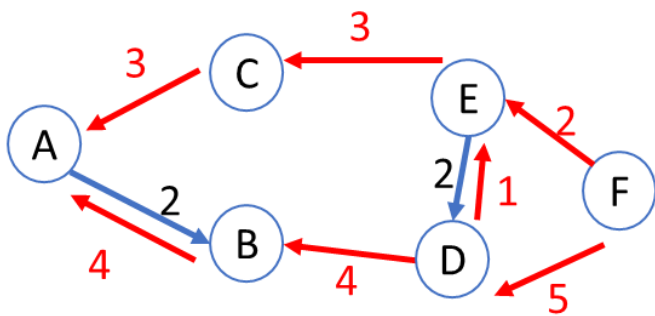
$A \rightarrow C \rightarrow E \rightarrow F$  kapasite 2



$A \rightarrow B \rightarrow D \rightarrow F$  kapasite 4



$A \rightarrow C \rightarrow E \rightarrow D \rightarrow F$  kapasite 1



**Soru 13:** Dizge eşleme (string matching) için kullanılan String Naive algoritması metindeki her karakter için örüntünün ilk karakterinin eşleşip eşleşmediğini kontrol eder. İlk karakterler eşleşiyor ise sırayla sonraki



GİRESUN ÜNİVERSİTESİ MÜHENDİSLİK FAKÜLTESİ  
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ  
ALGORİTMALAR DERSİ FİNAL SINAVI

karakterler tek tek kontrol edilir. Metin içerisinde örüntü tamamen eşleşmiyor ise, metin bir karakter kaydırılarak arama yeniden başlatılır. "abracacadabra" içerisinde "cad" örüntüsü arandığında kaç adet karşılaştırma yapılır?

text[0] != pattern[0] a?c  
text[1] != pattern[0] b?c  
text[2] != pattern[0] r?c  
text[3] != pattern[0] a?c  
text[4] != pattern[0] c?c  
text[5] != pattern[1] a?a  
text[6] != pattern[2] c?d  
text[5] != pattern[0] a?c  
text[6] != pattern[0] c?c  
text[7] != pattern[1] a?a  
text[8] != pattern[2] d?d

11 adet karşılaştırma