Veri Yapıları ve Algoritmalar (EBT512)

Ödev 1

Dinamik programlama ve Açgözlü Algoritmalar

Enver Arslan
1650E13113
enver.arslan@ogr.sakarya.edu.tr

Dinamik programlama ve Açgözlü Algoritmalar

1) Dinamik Programlama

Dinamik programlama bir problemi çözerken aynı alt-problemi birden fazla çözmemiz gereken durumlarda bu alt-problemi birden fazla kez çözmemizi engelleyen bir yaklaşımdır. Dinamik programlama, matematik ve bilgisayar bilimlerinde karmaşık problemleri çözmek için kullanılan bir metottur. Çakışan alt-problemler(Overlapping subproblems) ve En uygun altyapı(optimal substructure) problemlerine uygulanabiliyor.

Dinamik programlama kullanmak için problemimizin içermesi gereken üç prensip bulunmaktadır:

- Çözümü aynı olan alt-problemler.
- Büyük bir problemi küçük parçalara bölmek ve bu küçük parçaları kullanarak baştaki büyük problemimizin sonucuna ulaşmak. (Parçala, fethet)
- Çözdüğümüz her alt-problemin sonucunu bir yere not almak ve gerektiğinde bu sonucu kullanarak aynı problemi tekrar tekrar çözmeyi engellemek. (Hatırlama)

Eğer bir problem kendi içerisinde alt problemlere ayrılabiliyorsa bu çakışan alt-problemler prensibine uygundur. Buna en iyi örnek ise Fibonacci serisidir.

Fibonacci Serisi

Verilen bir sayının Fibonacci karşılığına fib(n) dersek;

fib(0) = fib(1) = 1 olmak üzere
fib(n) = fib(n-1) + fib(n-2)'dir.

Bu problemi çözümü aynı olan alt problemleri rekürsif yaklaşım kullanarak çözelim;

```
int fib(int n) {
 if (n < 2){
  return n;
 }
 return fib(n - 1) + fib(n - 2);
}
```

Bu kod kısa olmasına rağmen aynı değerler için **tekrar tekrar kendini çağıracak** şekilde çalışmaktadır.

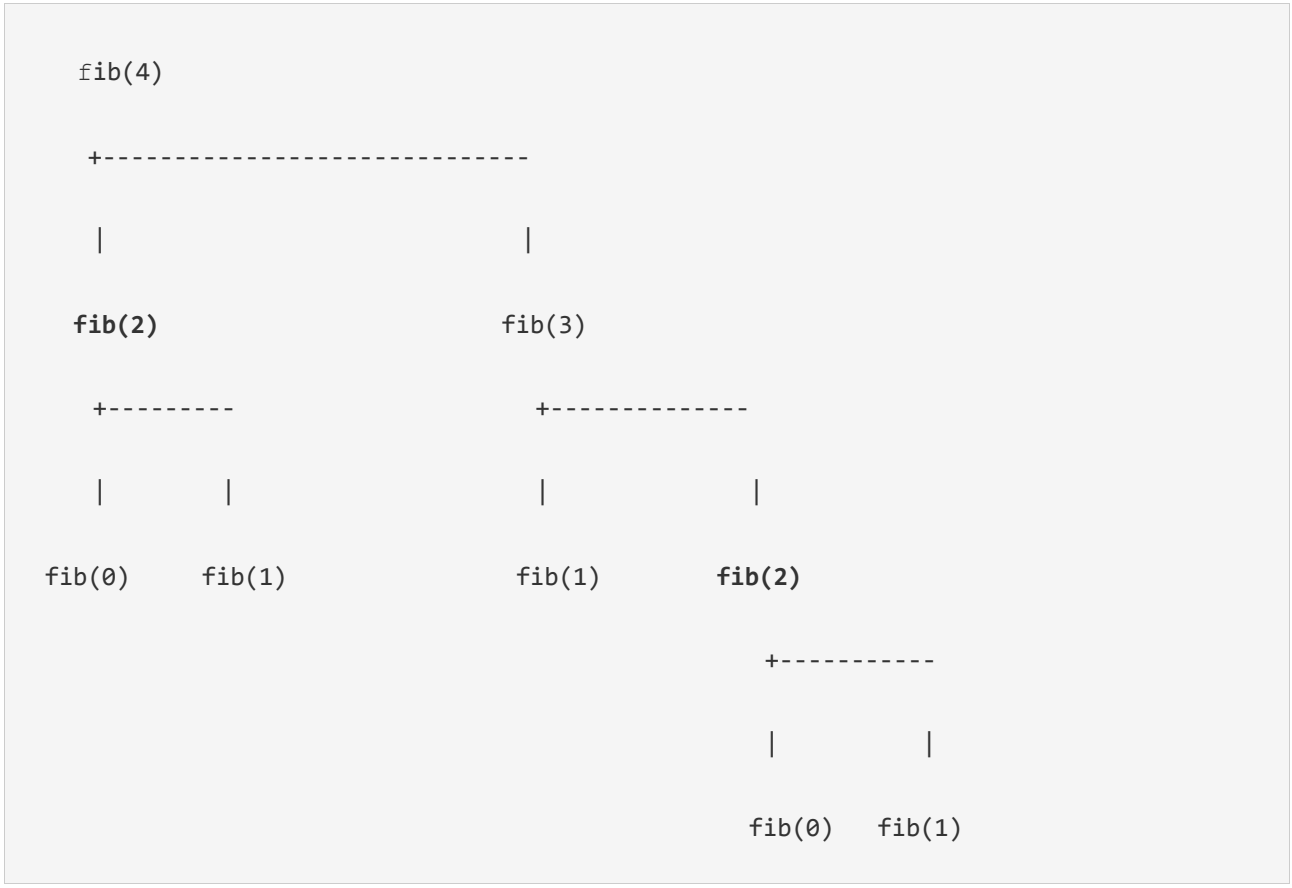
Fibonacci 4'ü hesaplamak için:

fib(4) = **fib(2)** + fib(3)

= (fib(1) + fib(0)) + fib(1) + **fib(2)**

= ((fib(1) +fib(0)) + fib(1))+ (fib(1) + fib(0))

Programımızın çağırımları görsel olarak bu şekilde ifade edilebilir:



Görüldüğü üzere fib(4) problemini çözmek için farklı dallarda iki kere fib(2) değeri hesaplanmak durumunda kaldı.

Fibonacci Serisine Dinamik Programlama Yaklaşımı

Dinamik programlama ile rekürsif olarak alt problemin her birini çözebilen genel çözümü aşağıdan yukarıya tekniğini (bottom-up) kullanarak daha önce çözdüğümüz problemin sonucunu bir yerde saklayacağız. Alt problemleri çözerken; bizden başka bir alt problemin çözümü istendiğinde genel çözümü tekrar çağırmaktansa zaten sakladığımız çözümü alt probleme sunacağız. Böylece *tekrar eden çağırımlara gerek olmayacak.* Eğer alt problem çözümümüz henüz saklanmadıysa çözülerek saklanacak.

Fibonacci değerlerini saklamak için f adında bir dizi oluşturalım, bu dizinin uzunluğu istenen n değeri kadar olacaktır. Çünkü fib(n) değerini bulmak için daha önce hesaplanmış n-1 uzunluğundaki sayının fibonacci değerlerine ihtiyacımız vardır, Fibonacci sayıları öncesinde gelen sayıların toplamından oluşur. Öyleyse bu değerleri tutmak için n elemanlı bir f dizisi oluşturmamız gerekiyor.

Ayrıca fib(1) ve fib(0) değerlerinin 1 olduğunu biliyoruz. Öyleyse bu f dizisinin ilk iki elemanını 1 değeri ile dolduracağız.

İstenen fibonacci değerini bulmak için o değerden 2 eksik kez dönen (çünkü fib(0) ve fib(1) değerlerini biliyoruz, tekrar hesaplamaya gerek yok.) bir döngü içerisinde fib(2), fib(3) .... fib(n) değerlerini hatırlatma dizisine yerleştireceğiz. Ardından istenen değeri hatırlatma dizisinden okuyarak değeri döndüreceğiz.

Yukarıdaki açıklamayı C dilinde uygularsak fib fonksiyonumuz bu şekilde olacaktır.

```
int fib(int n)
{
 if(n < 2){
   return n;
 }
 // Hatırlanacak ficonacci değerlerini tutuyoruz.
 int f[n];
 f[0] = 1;
 f[1] = 1;
 // Verilen sayıya kadar olan fibonacci değerlerini hatırlatma dizimizden
yararlanarak hesaplayıp saklıyoruz.
 for(int i = 2; i < n; i++){
  f[i] = f[i-1] + f[i-2];
 }
 // İstenen fibonacci değerini sakladığımız diziden alıp döndürüyoruz.
 return f[n];
}
```

İlk uygulamamızda tekrar eden çağırımlardan dolayı kodumuzun zaman karmaşıklığı dinamik programlama yaklaşımına göre daha yüksek olacaktır.

Rekürsif uygulamanın karmaşıklığı:

$T(n) = T(n-1) + T(n-2) = T(n-2) + T(n-3) + T(n-3) + T(n-4) = T(n-3) + T(n-4) + T(n-4) + T(n-5) + T(n-4) + T(n-5) + T(n-5) + T(n-6)$

Her bir çağırım kendisini ikişer kez çağırmaktadır. n. değeri bulabilmek için algoritma $\mathbf{2^n}$ defa çalışmak durumunda kalmaktadır. Bu yüzden zaman karmaşıklığına $O(2^n)$ diyebiliriz. Yer karmaşıklığı ise ekstra bir alan kullanmadığımız için O(1) olacaktır.

Dinamik programlama yaklaşımının karmaşıklığı:

Verilen n sayısının fibonacci değerini bulmak için n-2 defa dönen bir döngü içerisinde tüm fibonacci değerlerini hesaplayarak bir diziye koyuyoruz. Bu yüzden;

$T(n) = T(n-2)$

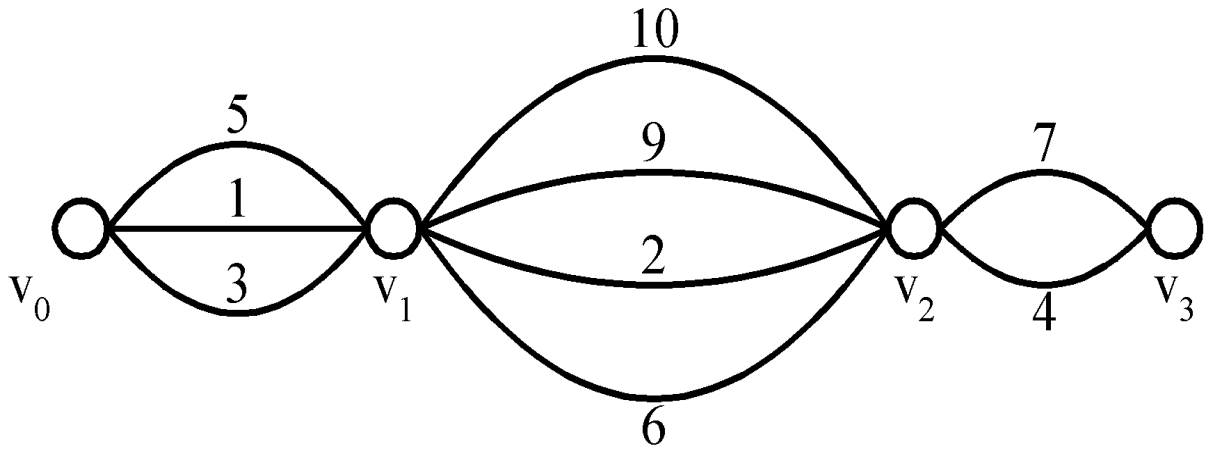
diyebilirliriz. Algoritmanin zaman karmaşıklığını sabit değeri yoksayarsak O(n) olacaktır. Ayrıca verilen sayıya kadar olan fibonacci değerlerini bir dizide tuttuğumuz için yer karmaşıklığına da O(n) diyebiliriz. Dizi veri yapısını kullandığımız için erişim karmaşıklığı da O(1) olacaktır.

2) Açgözlü Algoritmalar

Açgözlü yaklaşım bir problemin çözümünde her adımda en uygun seçeneği seçerek ilerlemek üzerine kuruludur. Yerel en uygun seçimlerin toplamı global en uygun seçimi oluşturacağını umar. Yaklaşım her adımda(alt-problem) en uygun çözümü bularak ilerlese de yapılan bu seçimler ana problemin çözümünde istenen başarımı sağlamayabilirler. Eğer bir problem açgözlü yaklaşımla mantıklı bir zamanda çözülebiliyorsa muhtemelen en iyi çözüm bu yaklaşımla olacaktır. Aç gözlü algoritmalar en uygun çözümü bulmak için tasarlansa da çok az sayıda optimizasyon problemine uygulanabilir.

Örnek 1:

Aşağıdaki şekilde $\mathbf{V_0}$ noktasından $\mathbf{V_3}$ noktasına gidebilmek için ***tercih edilmesi gereken en kısa yolu*** açgözlü yaklaşımı uygulayarak bulalım.



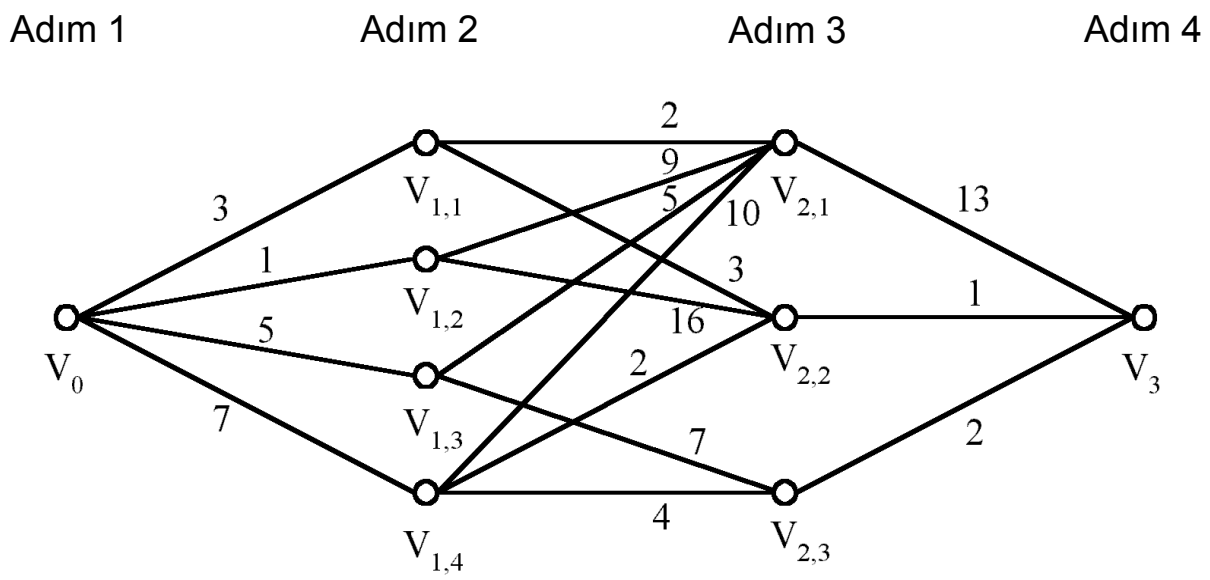
Problem çözme adımları:

1. $\mathbf{V_0}$ noktasından $\mathbf{V_1}$ noktasına gitmek için en kısa yolu seç. (1)
2. $\mathbf{V_1}$ noktasından $\mathbf{V_2}$ noktasına gitmek için en kısa yolu seç. (2)
3. $\mathbf{V_2}$ noktasından $\mathbf{V_3}$ noktasına gitmek için en kısa yolu seç. (4)
4. Her adımda çözülen alt problemler ile ana problemi çöz: Toplam($\mathbf{V_0}$, $\mathbf{V_3}$)= 1 + 2 + 4 = 7

Bu problem açgözlü algoritma kullanarak mantıklı bir zamanda çözülebilir. Çünkü her adım sonrasında kat edilen yol uzun da olsa aynı noktada buluşarak yeni bir seçim yapılıyor.

Örnek 2:

Aşağıdaki şekilde $\mathbf{V_0}$ noktasından $\mathbf{V_3}$ noktasına gidebilmek için ***tercih edilmesi gereken en kısa yolu*** bulalım.



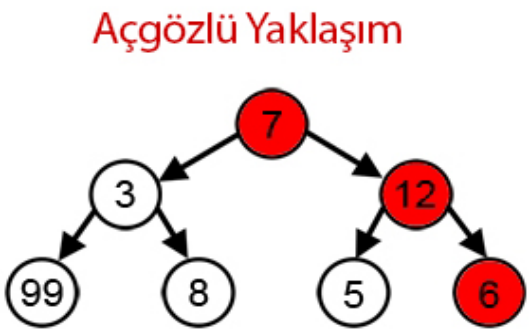
Böyle bir problemde açgözlü yaklaşımı uygulamaya çalışırsak 1. adımda yapacağımız en kısa yol seçimi diğer adımlarda yapacağımız seçimleri etkileyecektir. Çünkü 1. adımda seçtiğimiz en kısa yol 2. adımda yapabileceğimiz seçimleri kısıtlar ve seçemediğimiz daha iyi seçenekleri elemiş oluruz.

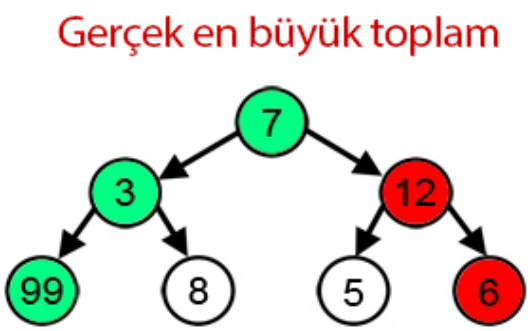
Açgözlü yaklaşıma göre problem çözme adımları:

1. Adım 1'den Adım 2'e gitmek için en kısa yolu seç. (1) $\mathbf{V_{1,2}}$
2. Adım 2'den Adım 3'e gitmek için en kısa yolu seç. (9) $\mathbf{V_{2,1}}$
3. Adım 3'den Adım 4'e gitmek için en kısa yolu seç. (13) $\mathbf{V_3}$
4. Her adımda çözülen alt problemler ile ana problemi çöz: Toplam($\mathbf{V_0}$,$\mathbf{V_3}$)= 1 + 9 + 13 = 23

Fakat göz ucu ile bakıldığında bile daha kısa yollar bulunabilir. Örneğin $\mathbf{V_0} - \mathbf{V_{1,1}} - \mathbf{V_{2,2}} - \mathbf{V_3}$ yolu daha kısa yol olarak bulunabilir. Sadece birinci adımda en kısa yolu seçmeyerek aç gözlü algoritmadan daha iyi bir seçim yapabildik. Görüldüğü üzere bu problem açgözlü algoritma ile en uygun seçimler yapılarak çözülecek bir problem değildir.

Aynı durum bir ağaç yapısı üzerinde en büyük toplamı açgözlü algoritma ile bulmaya çalıştığımızda da karşımıza çıkacaktır.





Dinamik Programlama ile Çözümü

Eğer dinamik programlamayı bu probleme uygularsak ve her alt problem için en kısa yolu hesaplayarak bu alt problemlerin çözümünden ana problemin en uygun çözümünü bulabiliriz.

$\mathbf{Y_{min}(i,j)}$: **i** ve **j** arasındaki en kısa uzaklık olsun.

$\mathbf{V_0}$ ile $\mathbf{V_3}$ arasındaki en kısa yol her noktanın $\mathbf{V_3}$ noktasına olan uzaklıklarının en kısası hesaplanarak bu hesaplamalardan en az olanın seçilmesiyle bulunur.

$$\mathbf{Y_{min}(V_0,\ V_3)} = \mathbf{min}\begin{cases} 3+\mathbf{Y_{min}}(V_{1,1},\ V_3) \\ 1+\mathbf{Y_{min}}(V_{1,2},\ V_3) \\ 5+\mathbf{Y_{min}}(V_{1,3},\ V_3) \\ 7+\mathbf{Y_{min}}(V_{1,4},\ V_3) \end{cases}$$

Kaynaklar:

- *Introduction to Algorithms* (Cormen, Leiserson, Rivest, Stein) 2001, Bölüm 15 "Dynamic Programming"
- *Introduction to Algorithms* (Cormen, Leiserson, Rivest, Stein) 2001, Bölüm 16 "Greedy Algorithms"
- http://en.wikipedia.org/wiki/Greedy_algorithm
- http://en.wikipedia.org/wiki/Dynamic_programming