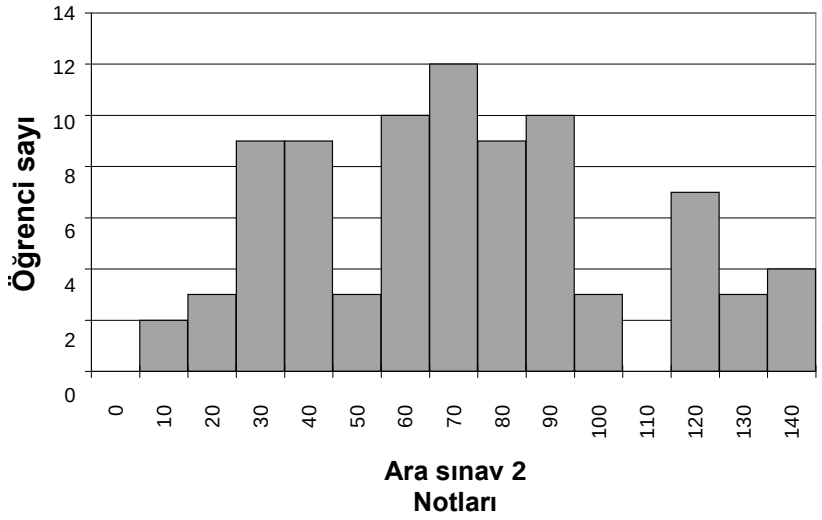
*Algoritmalara Giriş* 9 Aralık 2005
Massachusetts Institute of Technology 6.046J/18.410J
Profesörler Erik D. Demaine ve Charles E. Leiserson Kitapçık 33

# Ara Sınav 2 Çözümleri




## Problem 1. Yukarılar ve aşağılar

Akşam olunca, Prof. Silvermeadow Ulusal Teknoloji Üniversitesi'ndeki işinden uzaklaşıp, gece kulüplerinde sihir gösterileri yapmakta. Profesör, aşağıdaki kart numarasını geliştirmekte. 1'den n'ye kadar, n tane yüzü yukarı bakan kart var. İzleyicilerden biri,[i, j], arasında bir aralık vermekte. Profesör, $i \le k \le j$ olan her k kartını ters çevirmekte. Bu işlem sırasında izleyiciler profesörü hangi kartların yüzlerinin yukarı, hangilerinin aşağı olduğu konusunda sorguluyor ve işlem defalarca tekrarlanıyor. Aslında, gerçek kartlar yok. Profesör manipülasyonu kafasında yapmakta ve n oldukça büyük.
Profesör manipülasyonu yapabilmek için izleyenlere belli etmeden bir hesaplama aracı kullanıyor ancak , mevcut sürüm çok yavaş ve gerçek zamanlı olarak çalışmıyor. Profesöre verimli bir veri yapısı tasarlayarak yardımcı olun. Bu veri yapısı aşağıdaki işlemleri yapmalı:

- FLIP(i, j): [i, j] aralığındaki bütün kartları ters çevirsin.
- IS-FACE-UP (i): Eğer *i* kartı yukarı bakıyorsa DOĞRU, aşağı bakıyorsa YANLIŞ döndürsün.

**Çözüm:** F, izleyiciler tarafından istenen ters çevirme işlemlerinin sayısı olsun.
FLIP (i, j)'nin, FLIP (i, ∞) ve FLIP (j + 1, ∞) toplamına eşit olduğunu da unutmayın. Bu soruna hızlı bir çözüm, dinamik sıralı-istatistik ağacıdır; burada i anahtarlı eleman [i, ∞), aralığındaki döndürmeleri simgeler. x = i ve x = j + 1 için FLIP, eğer x, T'nin elemanı değilse; x'i T ağacına yerleştirir. Eğer x $\epsilon$ T ise x'i T ağacından siler. IS-FACE -UP ise, eğer ağaçtaki i elemanlarının sayısı ( buna z diyelim) çift ise "doğru", tek ise "yanlış" döner. Z'yi hesaplamanın bir yolu, i'yi T'ye eklemek, z'yi i'nin rütbesinden 1 eksik olarak belirlemek ve sonra T'den i'yi silmektir. Bu veri yapısı, O(min {F, n}) kadar yere gerek duyar ve FLIP ile IS-FACE -UP işlemlerinin her birini O(lg(min {F, n})) sürede gerçekleştirir. Tamamen bunun gibi doğru bir cevap, tam puan alır.
Soruya verilebilecek diğer cevaplar ve alınacak puanlar şöyle:

1. Bir başka çözüm; statik, 1 boyutlu ve anahtarları 1'den n'ye kadar olan bir menzil ağacı olan T'yi yaratmaktır. Ağacın her x düğümü, x köklü alt-ağaçta yer alan tüm elemanların aralığının ters çevrilmesine karşılık gelen bir bit'i depolar. FLIP(i, j), [i, j] üzerinde bir değer kümesi sorgulaması yapar ve sorgulama tarafından bulunan O(lg n) ayrışık alt-ağaçlarının bitini tersine çevirir. IS-FACE -UP(i), menzil ağacında aşağı doğru i düğümüne kadar yürür ve kökten i'ye kadar olan yoldaki düğümlerin bitlerinin toplamı çift sayıysa "doğru" döndürür. Bu veri yapısı, O(n) kadar boş alan ve her iki işlem için O(lg n) süre kullanır. Bu tür bir çözüm, 20 puan aldı.
2. Bazı öğrenciler F ≪ n olduğunu varsayıp, her [i, j] ters çevirme aralığını bir aralık ağacında depolamışlar. Bu durumda, IS-FACE -UP(i) işlemi, aralık ağacını sorgular ve eğer i ile çakışan aralıkların sayısı çift ise "doğru" döndürür. Eğer ağaç çakışan döndürme aralıkları içeriyorsa, FLIP O(lg F) ve IS-FACE -UP da O(F lg F) sürede koşarlar. Eğer ağaçta çakışan aralıklara izin vermezsek, FLIP O(F lg F ) süreye gerek duyarken, IS-FACE-UP O(lg F) süreye gereksinim duyar. Bu tür bir çözüm, 13 puan aldı.
3. Basit ama yavaş bir çözüm de hangi bitlerin ters çevrildiğini kaydetmek için n uzunluğunda bir bit-vektörü kullanmaktır. Bu çözüm, FLIP işlemini O(j − i) ve IS-FACE -UP işlemini de O(1) sürede koşturur. Yine, bir başka basit çözüm ise, bütün F ters çevirme aralıklarını bağlantılı bir listede saklamaktır. Burada FLIP'in O(1), IS-FACE-UP'ın ise O(F ) süreye gereksinimleri olur. Bu tür cevaplar 8 puan aldı.
4. Basit çözümlerden daha yavaş çalışan herhangi bir çözüm, (mesela, FLIP için O(lg n + j − i)) 5 puan aldı.

**Problem 2. Veri Merkezi**

Dünyaca ünlü mimar Gary O'Frank, Veri Merkezi adında yeni bir bina tasarlaması için görevlendirilir. Gary mimari tasarım asistanından, Veri Merkezinin modelini tasarlarken, hassas kesim çubuklar kullanmasını, ama istenmeyen dik açılardan da kaçınmasını ister. Gary, 1,2,...,n etiketli, i çubuğu $x_i$ uzunluğunda, n tane çubuktan oluşan bir küme üretir. Gary, çubukları asistanına vermeden size gösterir ve bunlardan herhangi üçünü kullanarak bir dik üçgen oluşturup oluşturamayacağınızı sorar. a, b, c çubuklarıyla böyle bir üçgen oluşturulup oluşturulamayacağını belirleyen verimli bir algoritma bulun. Bu üçgenin köşe uzunlukları; $x_a$, $x_b$ ve $x_c$ olsun. (Dolayısıyla $x_a^2 + x_b^2 = x_c^2$ olmalı.)

**Çözüm:** X [1..n] çubuk uzunluklarından oluşan dizilim olsun. Bizden istenen, dizinleri i, j ve k olan ve $X[i]^2 + X[j]^2 = X[k]^2$ eşitliğini sağlayan değerleri bulmamız.

Başlangıç adımı olarak, X [1..n] dizilimini artan şekilde sıralarız. Bu, örneğin yığın sıralaması ile en kötü durumda O(n lg n) süreye mal olur. Dizilim sıralanır sıralanmaz, $i < j < k$ olduğunda, i, j, ve k dizinleri için $X[i]^2 + X[j]^2 = X[k]^2$ olduğunu kontrol etmek yeterlidir. .

En kötü durumda $O(n^2)$ ile koşan bir algoritma bulmak için, k'yı 1'den n'ye kadar dış döngüde arttırırız, iç döngüde de $X[i]^2 + X[j]^2 = X[k]^2$ kullanarak, i ve j'leri, $i<j<k$ iken doğrusal sürede ararız.

```
1   X [1..n] artan sırada sırala
2   k ← 1
3   while k ≤ n
4       do  i, j ← 1, k − 1
5           while i < j
6                   if X [i]^2 + X [j]^2 = X [k]^2 ise doğru döndür
7                   if X [i]^2 + X [j]^2 < X [k]^2
8                     then  i ← i + 1
9                     else  j ← j - 1
10  yanlış döndür
```

Değişmezin;

$$[1 \le i' < j' \le k - 1 \text{ ve } X[i']^2 + X[j']^2 = X[k]^2] \Rightarrow [i \le i' < j' \le j].$$

olduğunu ispatlayacağız.

Başlangıçta, i = 1 ve j = k − 1 ve değişmez geçerli. Tümevarım adımı için, iç döngüye girdiğimizde ifadenin doğru olduğunu varsayalım. X [1..n]'nin artan düzende sıralı olduğuna dikkat edin. Eğer $X[i]^2 + X[j]^2 < X[k]^2$ , ise; $i < j' \le j$, için

$$X[i]^2 + X[j']^2 \le X[i]^2 + X[j]^2 < X[k]^2$$

ve değişmez i ← i+1 için geçerli. Eğer,

$X[i]^2 + X[j]^2 > X[k]^2$

$i \leq i' < j$ ise;

$X[i']^2 + X[j]^2 \geq X[i]^2 + X[j]^2 > X[k]^2$

ve değişmez j ← j-1 için de geçerli.

Eğer iç döngü biterse, i=j demektir ve X [i']$^2$ + X [j' ]$^2$ = X [k]$^2$ ifadesini, 1≤ i' < j' ≤ k – 1 olacak şekilde karşılayan i' ve j' dizinleri bulunmaz.
Bu algoritmamızın doğruluğunu kanıtlar. İç döngü, en kötü durumda O(n)'de koşar ve böylece dış döngü ile beraber en kötü koşma süresi O(n$^2$) olur.

**Problem 3. Bir matrisi, pivot kullanarak negatiflikten kurtarma**

Bir M [1 . . n, 1 . . n] matrisi, R∪ (∞ )'dan alınan girdiler içerir. Her satır en fazla 10 adet sınırlı, ama bazıları negatif olabilecek değer içerir. Amacımız, pivot kullanarak M'yi dönüştürerek, her girdisini negatif olmaktan kurtarmaktır.

```
PIVOT(M, i, x)
1  for j ← 1 to n
2       do M [i, j] ←M [i, j] + x
3            M [j, i] ← M [j, i] − x
```

i ve x'in çeşitli değerlerinde bir dizi pivotlama işlemi kullanarak sonuçta i, j= 1,2,...., n için, M[i, j] ≥ 0 olup olamayacağını belirlemek için verimli bir algoritma bulun.

**Çözüm:**

PIVOT 〈(M, $i_1$, $x_1$ ), PIVOT (M, $i_2$, $x_2$ ), . . . , PIVOT(M, $i_k$ , $x_k$ )〉 gibi bir pivot dizimiz olduğunu düşünelim. Her bir pivot işlemi sütunlara ekleyip, satırlardan çıkarma yaptığına göre, toplama işleminin birleşme ve değişme özelliklerinden dolayı pivot işlemlerinin sırasının önemli yoktur. Ayrıca, PIVOT (M, $i_1$ , $x_1$), PIVOT (M, $i_2$ , $x_2$) tarafından takip edildiğinden, bu PIVOT (M, $i_1$ , $x_1 + x_2$ )'e eşittir ve herhangi bir dizinde birden fazla pivot işlemi yapılmasına gerek yoktur. Bu nedenle harhangi bir pivot dizisi tam olarak 〈PIVOT (M, 1, $x_1$ ), PIVOT(M, 2, $x_2$ ), . . . , PIVOT (M, n, $x_n$ )〉 gibi n sayıda pivot işlemine eşittir; burada bazı $x_i$'ler 0 olabilir.

Herhangi bir M [i, j] matris girdisini etkileyen pivot işlemleri, sadece PIVOT (M, i, $x_i$ ) ve PIVOT (M, j, $x_j$ )'dir. Bu nedenle, tüm pivotların işlenmesinden sonra oluşan M ', M'[i, j] = M[i, j] + $x_i$ − $x_j$ ögelerine sahiptir. Hiç bir M'[i, j] öğelesinin 0'dan küçük olmamasını istiyoruz ve bu da M [i, j] + $x_i$ − $x_j$ ≥ 0, veya $x_j$ − $x_i$ ≤ M [i, j] olması anlamına geliyor. Dolayısıyla, sadece bir küme fark kısıtını çözmemiz gerekiyor. n değişkende en fazla 10n kadar fark kısıtı bırakmamızı sağlayacak M [i, j] = ∞ fark kısıtlarını görmezden geliriz. 10n tane sınırlı ögeyi bulmak bize O($n^2$ ) süreye mal olur ve Bellman-Ford algoritmasını kullanarak bunları O(n · (10n)) = O($n^2$ ) sürede çözebiliriz.

**Problem 4. Queueinator$^{TM}$ 'yi geliştirmek**

Profesör Uriah'ın şirketi, profesörün sıcak fizyon konusundaki araştırmalarıyla geliştirdiği Queueinator$^{TM}$ 'yi üretmeye ve satmaya başladı. Queueinator$^{TM}$ öncelikli bir kuyruk donanımı ve normal bir bilgisayara bağlanarak verimli şekilde öncelikli kuyruk işlemleri olan INSERT(araya yerleştirme) EXTRACT-MIN(en küçüğü çıkarma) işlemlerini O(1) sürede yapabiliyor. Profesörün şirketinin bir müşterisi var, ancak müşteri "iki uçlu" bir öncelikli kuyruk uygulaması donanımı istiyor. Sadece INSERT ve EXTRACT-MIN işlemlerini değil ayrıca EXTRACT-MAX işlemini de yapan bir cihaz gerekmekte. Queueinator$^{TM}$'ın yeniden tasarımı profesörün şirketinin yaklaşık 1 yılını alacak. Profesöre verimli bir "iki-uçlu öncelikli kuyruk" yazılımı ve bir ya da daha fazla Queueinator$^{TM}$ cihazı tasarlaması konusunda yardımcı olun.

**Çözüm:**

İki tane Queueinator ile *MIN* ve *MAX* için ayırın; *MAX*'ta anahtarları eksi olacak şekilde düzenleyin ve yaklaşık aynı boyutta olsun ve *MAX*'ın en küçük elemanını *mid* olarak saklayın. Anahtarı *k* olan bir elemanı araya yerleştirirken, *mid* ile karşılaştırın ve eğer *k<mid* ise onu *MIN*'e yerleştirin; aksi halde *MAX*'a ekleyin. EXTRACT-MIN için MIN'den, EXTRACT-MAX için *MAX*'dan özütleyin (elemanı çıkarın). Eğer biri boşalırsa, diğerindeki bütün elemanları sıralı bir dizilim olarak çıkarın. Dizilimi ortadan ikiye bölün ve küçük yarıyı *MIN*'e, büyük yarıyı da *MAX*'a yerleştirin. Bu çözüm tüm işlemler için O(1) amortize edilmiş maliyet verir.

**Doğruluğun İspatı**: Değişmez, *MIN*'deki bütün elemanların mid'den küçük olduğu ve *MAX*'taki bütün elemanlarin da *mid*'e büyük-eşit olduğudur. Diğer değişmez de *MIN* ve *MAX*'ın araya yerleştirilmiş ve daha özütlenmemiş bütün elemanları içerdiğidir. Bu değişmezler tüm işlemlerde korunur. Dolayısıyla özütlemeler doğru sonuçları verir.

**Koşma süresi çözümlemesi**: Potansiyel fonksiyon $\Phi(i) = 2\,||MIN| - |MAX||$'tır. Burada |M I N | ve |M AX |, sırasıyla *MIN* ve *MAX* QUEUEINATOR'larındaki eleman sayılarını gösterir. *MIN* ve *MAX* başta boştur ve potansiyelleri 0'dır. Potansiyel hiçbir zaman 0'dan küçük olamaz. Amortize edilmiş maliyetlere bakalım.

1. INSERT: M I N veya M A X'ta araya yerleştirme yapar. Bunun gerçek maliyeti 1'dir ve potansiyel eğer büyük kuyruğa yerleştiriyorsak 2 artar veya eğer küçük kuyruğa yerleştiriyorsak 1 azalır. Dolayısıyla maliyet her zaman O(1)'dir.
2. EXTRACT-MIN: Eğer MIN boş değilse gerçek maliyeti 1'dir, potansiyel ise 2 artar veya azalır. Eğer MIN boş ise, kuyrukları tekrar dengelemeniz gerekir. Dengelemeden önce potansiyel 2 çarpı MAX'taki eleman sayısı kadardır. Dengelemeden sonraki potansiyel ise ya 0'dır ya da 2'dir ( eğer elemanların sayısı tek sayı ise, ama bu fazla şey değiştirmez). Bütün elemanlar özütlendiği veya araya yerleştirildiği için, dengelemenin gerçek maliyeti 2*MAX'ın eleman sayısıdır. Yani, potansiyeldeki değişim, yeniden dengelemeyi karşılar ve amortize edilmiş maliyet O(1) olur.
3. EXTRACT-MAX da EXTRACT-MIN'in benzeridir.
   Bu nedenle işlemlerin amortize edilmiş maliyeti O(1)'dir.

Bazı başka güzel çözümler de aşağıda verilmiştir.

1. 4 Queueinator kullanın ve 2 tanesi silinen elemanların kaydını tutsun. Bu çözümü tercih edenlerden birkaç puan kırıldı çünkü gereksiz donanım kullanılıyor.

2.Öğelerin içinde ek alanlar kullanarak, bu alanlarda silinen elemanlar takip edilir. Bu çözümü tercih edenlerden, var olan öğelerin değiştirilebileceğine dair varsayımda bulundukları için birkaç puan kırıldı.

3.Silinen elemanları tutmak için kıyım tabloları kullanın. Bu beklenen amortize edilmiş süre olarak O(1) süresini verir ve anahtarların bir aralıktaki tamsayılar olduğu varsayımını kullanır. Bu cevabı verenler 5 puan kaybettiler.

4.Direkt erişim tabloları kullanarak silinen elemanları saklayanlardan, ek yer gereksinimi nedeniyle 6-7 puan kırıldı.

Çözümler için çözümlemeler çok önemli olduğundan, yanlış yada eksik koşma süresi çözümlemesi olan yanıtlardan puan kırıldı.

**Problem 5. Spam ( mesaj yağanağı ) dağıtımı**

Professor Hormel spam dağıtımı için bir ağ tasarlamakta. Bu ağ, $T = (V, E)$ ile simgelenen ve kökü $r \in V$ olan bir ağaç yapısında; ağacın köşe ağırlık fonksiyonu $w: E \rightarrow R$ ve negatif değil. Her $v \in V$ olan köşe bir milyon e-posta adresini barındıran bir sunucuyu temsil ediyor ve her $e \in E$ olan kenar ise maliyeti w(e) dolar olan bir iletişim kanalını temsil ediyor. Kök r'den v'ye giden tüm yol satın alınırsa spam $v \in V$ sunucusuna ulaşıyor. Profesör, kök $r$'den $k \leq |V|$ sunucularına (kök dahil) ucuza spam göndermek istiyor. Profesöre en az ağırlıklı ve bağlantılı, kök dahil k köşesi olan $T$ alt ağacını bulacak bir algoritma tasarlayarak yardımcı olun. ( Kısmi not için, problemi her $v \in V$ köşesinin ***T*** içinde 2 ardılının olması durumunda çözün.)

**Çözüm:**

**Genel Bakış.** Bu problemin çözümü dinamik programlamaya dayanmakta. Her $v \in V$ köşesinin, $k' \leq k$ sayısıyla kombinasyonu (birleşimi) için $k|V|$ tane alt problem tanımlarız. Her alt problem, "v kökü ve k' tane köşesi olan en az ağırlıklı bağlantılı alt ağacı bulun." şeklindedir. Her alt problem O(k) süresinde çözülebilir, bu da toplamda $O(k^2|V|)$ kadar koşma süresine mal olur.

**Simgelem.** Bu çözümde aşağıdaki simgelemi kullanacağız.

- MWCT(v,i): ***i*** adet düğümlü, kökü ***v*** olan en az ağırlıklı bağlantılı ağaç
- *child*(v,i): ***v*** düğümünün ***i*** çocuğu/ardılı.
- w(v, i): ***v***'den ardılı ***i***'ye giden kenarın ağırlığı
- W (S): S ağacındaki bütün kenarların ağırlıkları toplamı
- *deg*(v): ***v*** düğümünün ardıllarının sayısı.

**En uygun altyapı.** Bu problem hem uygun altyapıyı hem de çakışan alt problemleri kapsamına alır. Sezgi için, bir ***T*** ikili ağacını düşünün. ***v*** bu ağaçtaki bir düğüm olsun, MWCT (*T*) 'nin kökü ***v***'de olsun, $\ell$ tane düğümü olsun. Eğer $T_1$, $v$'nin $\ell_1$ düğümlü sol ağacı ise, $T_1$, v'nin sol ardılında köklenen, $\ell_1$ düğümlü MWCT'sidir. Benzer şekilde, eğer $T_2$, $v$'nin sağ alt ağacı ve $\ell_2$ düğümlüyse, $T_2$, ***v***'nin sağ ardılında köklenen $\ell_2$ düğümlü MWCT'sidir. İspatı, kes-yapıştır yöntemiyle aşağıdaki gibi yapılır. $T_1$'in $v$'nin sol ardılının MWCT'si *olmadığını* farzedelim. Bu, başka bir ***S*** ağacının olduğunu ve bunun $v$'nin sol ardılında köklü ve $T_1$'den daha az ağırlıklı olduğu anlamına gelir. v köklü ağacın ağırlığını $T_1$'i S ile değiştirerek azaltabiliriz ve bu da $v$ köklü ve $k$ düğümlü ağacın en az ağırlıklı bağlantılı alt ağaç olduğu varsayımımız ile çelişir.

Şimdi çakışan alt problemleri nasıl kullanacağımıza bakabiliriz. Bir kez MWCT( $v$, $\ell$)'yi, bütün $v \in V$ ve $\ell \in k'$ ler için hesapladığımızda, (v, k' + 1)'i belirleyebiliriz. Özellikle, $v$'nin ardıllarından köklenen alt ağaçlardan toplamda $k'$ düğüme sahip olanları seçeriz. Yani v'nin sol alt ağacında $\ell$ tane düğüm varsa, sağdaki alt ağaçta ( $k' - \ell$) düğüm olmalıdır. $\ell$'yi; ağacı, sağ ve sol ardıllarına toplam ağırlığı en aza indirecek şekilde seçeriz. Şimdi bunu ikiden farklı boyutlu ağaçlara genelleyelim ve algoritmayı daha detaylı açıklayalım.

**Algoritmanın Açıklaması.** *T[v]*, *T*'nin *v*'de köklü alt ağacı olsun. *T [v, c]* de *T*'nin *v*'de köklü ve *v* ile *1...c*'ye kadarki ardıllarında köklü alt ağaçlarını ifade etsin.

$$T[v, c] = v \cup \{T[w] : w = childv, i, 1 \leq i \leq c\}$$

İki |*V* | × *k* × |*V* | dizilimi yaratalım: *C [1..*|*V* |*, 1..k, 1.. |V |]* ve *B [1..*|*V* |*, 1..k, 1..*|*V* |]. C[v, k', c] ise k' düğümü olan minimum ağırlıklı bağlantılı T[v,c] alt ağacının maliyetini saklasın. B dizilimi dinamik programlama sona erdirildikten sonra, ağacı tekrar oluşturmak için kullanılır. B[v, k', c], T[v, c]'nin MWCT'sinin ardıl c'de k' düğümü olan alt ağacındaki ardıllarının sayısını depolar.
Sonuçta oluşan dizilimin k|V |$^2$ boyutlarında 3 boyutlu tablolar olduğuna dikkat edin. Şansımıza, girdilerin sadece k|V | kadarını kullanacağız, geri kalanını başlatılmamış olarak veya boş bırakacağız; algoritmayı çözümlerken göreceğimiz gibi. (Basit olması için, rastgele boyuttaki bellek bloklarına O(1) sürede tahsis yapılabildiğini varsayacağız. Eğer bellek tahsisleri daha maliyetli ise, bellek kullanımını O(k|V |)'ye indirebiliriz.
MWCT(V)'yi, k düğüm durumunda hesaplamak için ana yöntemimiz aşağıdaki gibi:

```
MWCT(V, k)
 1  for all v ∈ V
 2      do for c = 1 to deg(v)
 3             do C[v, 1, c] ← 0
 4                B[v, 1, c] ← 0
 5  for ℓ = 2 to k
 6      do for all v ∈ V
 7             do w ← child(v, 1)
 8                C[v, ℓ, 1] ← w(v, w) + C[w, ℓ − 1, deg(w)]
 9                B[v, ℓ, 1] ← ℓ − 1
10                for c = 2 to deg(v)
11                    do i ← FIND-NUM-CHILDREN(v, ℓ, c)
12                       B[v, ℓ, c] ← i
13                       if i = 0 then C[v, ℓ, c] ← C[v, ℓ, c − 1]
14                       if i = ℓ then C[v, ℓ, c] ← w(v, w) + C[w, ℓ − 1, deg(w)]
15                       if 0 < i < ℓ then C[v, ℓ, c] ← C[v, ℓ − i] + w(v, w) + C[w, i, deg(w)]
```

1'den 4. satıra kadar olan kısımda, bütün *v* Є *V* 'ler ve *c* ≤ |*V* | 'ler için C [v, 1, c] = 0 ilklendirmesiyle başlıyoruz. Bunlar dikkate alacağımız en küçük alt problemlerdir. Tek düğümlü, en az ağırlıklı bağlantılı *v* köklü alt ağaç, v'nin kendisini içerir ve maliyeti 0'dır.
Bundan sonra, 3 iç içe döngüyle dizilimlerimizi doldururuz. Dış döngüde (satır 5), $\ell$'yi 2'den k'ya kadar döngüye sokarız. 2. döngüde (satır 6), *v*'yi V'deki bütün düğümlerde döngüye sokarız. İçerideki döngüde (satır 10) *c*'yi 2'den ***deg***(*v*)'ye kadar döngüleriz.

Döngünün içinde, *T[v, c]* alt ağacını ve *w=child(v,c)* düğümünü inceleriz. S, $\ell$ tane düğüm içeren T[v,c]'nin MWTC'si olsun. *C[v,*$\ell$*, c]*'yi, yani S'nin maliyetini hesaplamak istiyoruz.

İlk olarak, *c=1* olan durumu inceleyelim. Bu durumda, *S*'deki bütün düğümler, *w*'da köklenen altağaçtır. Bu nedenle *S, v* düğümünü ve *w*'nun $\ell$*-1* düğümlü MWCT'sini içerir.

Şimdi, *c ≥ 2* durumunu inceleyelim. *S*'deki bazı düğümlerin, *v*'nin ilk *c-1* ardılından köklenen altağaçlarda olabileceğini ve bazılarının da *w*'da köklenen altağaçlarda olabileceğini unutmayın. FIND-NUM-CHILDREN (v, $\ell$, c) fonksiyonu, *S*'nin *w* 'de köklü altağaçlarındaki düğümlerin sayısını hesaplar; Geri kalan $\ell$ *- i – 1* düğüm ise, *v*'nin ilk *c-1* ardılından köklenen altağaçlardır.

Bu durumda, *S*'nin maliyetini belirlemek kolaydır. Eğer S'deki herhangi bir düğüm w köklü alt ağaçta yer almıyorsa, *S* sadece, *T[v,c-1]*'in MWCT'sidir ve $\ell$ düğüm vardır (satır 11). Eğer bütün düğümler w köklü altağaçta ise, ***S***, *v* düğümü ile $\ell$ *-1* düğümlü *w*'nun MWCT'si olur (satır 12). Son olarak eğer i, 0 ile $\ell$ arasında ise, ***S****,* $\ell$ *–i düğümlü T[v, c-1]*'in MWCT'si artı *i* düğümlü *w*'nin MWCT'si kadardır(satır 13)

Geriye FIND-NUM -CHILDREN altprogramını tartışmak kalır:

```
FIND-NUM-CHILDREN(v, ℓ, c)
 1  w ← child(v, c)
 2  min-weight ← C[v, ℓ, c − 1]
 3  num ← 0
 4  for i = 1 to ℓ − 2
 5       do wt ← C[v, ℓ − i, c − 1] + w(v, c) + C[w, i, deg(w)]
 6          if wt < min-weight
 7            then min-weight ← wt
 8                 num ← i
 9  if min-weight > w(v, w) + C[w, ℓ − 1, deg(w)]
10    then num ← ℓ
11  return num
```

Bu altprogram, $\ell$ sayıda boğumun *T[v, c-1]* ile *w=child(v,c)* köklü alt ağaç arasındaki bölünmesinin tüm yollarını kıyaslar. Önce, tüm $\ell$ sayıda ardılın *T[v, c-1]*'de olduğu ve *w*'de köklenen altağaçta hiç ardıl olmayan durumu ele alır (satır 2-3). Sonra, $\ell$ ***- 2*** kez döngüye girerek $\ell$ ***– i*** düğümü *T[v, c-1]* 'in içine ve ***i*** düğümü de *w*'de kökü olan altağacın içine yerleştirir ( satırlar 4-8). Son olarak da $\ell$ ***- 1*** ardılın *w*'de kökü olan altağacın içinde olduğu durumu ele alır (satırlar 9-10).

Algoritmanın son basamağında ağacı basar ve çıkan ağaçta derinliğine gezinme yapar.

```
PRINT-TREE(v, k)
1  PRINT(v)
2  ℓ ← k
3  for c = deg(v) downto 1
4       do if ℓ > 0 and B[v, ℓ, c] ≠ 0
5            then w ← child(v, c)
6                 PRINT-TREE(v, B[v, ℓ, c])
7                 ℓ ← ℓ − B[v, ℓ, c]
```

PRINT-TREE altprogramı, v'nin son ardılından başlayarak c ardılının kök olduğu alt ağaçtaki boğumların sayısını belirlemek için, *B[v,* $\ell$*, c]*'yi değerlendirir. Daha sonra, alt ağaçta özyinelemeli olarak MWCT'de kalan düğümlerin sayısını günceller.

**Koşma Süresi.** Önce MWCT'nin 1-4. satırlarını değerlendirelim. 2 döngü her düğümdeki her kenarı işler, bu da $O(|E|) = O(|V|)$ işleme denk gelir. Daha sonra 5. satırdaki *for*-döngüsü $O(k)$ süresince tekrarlanır. Şimdi 6-15 satırları arasını değerlendirelim. En iç döngü her köşenin her kenarı için bir kez olmak üzere $O(|E|) = O(|V|)$ kez çalışır.. Döngünün her çalışması FIND-NUM -CHILDREN (v, l, c)'ı çağırır ve bu da $O(\ell) = O(k)$ süresi alır. Bu nedenle 6-15. satırlar arasındaki her bir döngü, $O(k|V|)$'ye mal olur. Böylece MWCT'nin toplam maliyeti, $O(k^2|V|)$'dır. Son olarak çıkış baskısı maliyetimiz, $O(|V|)$'dir.

**Doğruluk.** Doğruluk, *k*'da kes yapıştır tartışması kullanarak tümevarım yöntemi ile yapılır. Döngü değişmezi, en içteki döngü tarafından sağlanır ve bütün $c' \le c, C[v, \ell, c']$'ler için $T[v, c']$'nin MWCT'sine eşittir. Dış döngünün değişmezi, tüm v', tüm $\ell' \le \ell$, tüm $c' \le deg(v)$'ler için, $C[v, \ell', c'], \ell'$ düğüm için $T[v', c']$'nin MWCT'sine eşittir. Bu iki değişmez tümevarımla birlikte ispatlanmalıdır. ( Burada *B* diziliminden bahsetmedik ama onun da doğruluğu aşağıdakine benzer şekilde çıkarılabilir.)

Bazı $v, \ell$ ve *c*'ler için en iç döngünün bittiği anı düşünün. $C[v, \ell, c]$'nin $T[v, c]$'nin MWCT'sinin maliyeti olduğunu söyleriz. Bunu kabul etmezseniz, o zaman $T[v,c]$'nin *S*'si gibi başka bir alt ağaç vardır ve bunun $\ell$ düğümü olup maliyeti $C[v, \ell, c]$'den küçüktür. Şimdi, *v*'nin ardılı olan *c*'de köklü S altağacını göz önüne alalım.(S boş ağaç olabilir.) Bu altağacın maliyetinin en az $x = C[child(v,c), i, \mathbf{deg}(child(v,c))]$ olması gerektiğini biliyoruz, çünkü tümevarımla, ***x***, *i* düğümlü *v*'nin *c* ardılının MWCT'sidir. Aynı şekilde $T[v,c-1]$'deki bütün düğümleri içeren S'nin alt ağacını değerlendirelim. Maliyet en az $y = C[v, \ell - i, c-1]$ olmalıdır çünkü, tümevarımla ***y***, $\ell - i$ düğümlü $T[v, c-1]$'in MWCT'sidir. Ama, FIND-MIN-CHILDREN, $C[v, \ell, c]$'nin $x + y + w(v, c)$'den daha büyük olmadığını garanti ediyor ve bu varsayımımızla çelişiyor. Sonuçta dış döngü değişmezinin tümevarım adımı, iç döngü değişmezinden çıkıyor, bu da ispatımızı sona erdiriyor.

**Problem 6. Günde bir domates**

Profesör Kerry domateslere bayılır! Profesör, hergün bir domates yer. Çünkü profesör, domatesin içindeki antioksidan likopenin sağlığa faydaları hakkında takıntılıdır ve onları yemeyi çok sever. Domates fiyatları yıl içinde artar ve azalır ve doğal olarak fiyatlar düşükken, profesör alabildiği kadar çok domates almak ister. Ancak domatesin belli bir raf ömrü olduğundan (d gün), profesör, ***i*** gününde aldığı domatesi $i \le j < i + d$ aralığında bir ***j*** gününde yemelidir veya domates boşa gider ve çöpe atılır. Profesör, herhangi bir gün ne kadar çok domates alırsa alsın, hergün sadece bir domates yediğinden, fiyat çok düşük olsa da çok fazla domates almamaya dikkat etmelidir.

Profesörün takıntısı, onu domatese çok para harcadığı konusunda bir endişeye itti. Profesör, geçmiş *n* günün domates fiyat verilerini edindi ve domatese ne kadar para harcadığını biliyor. Geçmiş fiyat verileri *C[i],* ***i*** günündeki fiyatı temsil eten *C [1 . . n]* diziliminde saklanıyor. Profesör, geçmiş verilere bakarak, en az ne kadar para harcayarak günde bir domates yiyebileceğini öğrenmek istiyor. Bunu kendi yaptığı harcama ile karşılaştıracak.

Geçmiş verilere dayanarak en iyi çevrimdışı (20/20 sonradan belirlenimci) satınalma stratejisini belirleyebilmek için verimli bir algoritma yazın. *d, n* ve *C[1,2,...,n]* verildiğinde, algoritmanızın çıktısı *B[1,...,n]*'yi vermeli ve *B[i]*, ***i***'inci günde satın alınacak domateslerin sayısı olmalı.

**Çözüm:**

Bu problemi $\Theta(n)$ sürede çözebilirsiniz. Bu olabilecek en hızlı asimptotik çözümdür, çünkü maliyet dizilimini taramak için $\Omega(n)$ süreye ihtiyaç vardır. Her i gününde, o gün için, $W(i) \equiv [i-d+1, i] \cap [1,n]$ aralığında bir domates satın almalısınız.. Burada aç gözlü seçimin çalıştığını not edin: en uygun çözümde, ***i*** günü için en az maliyetli $T_i$ gününde domatesi satın almalısınız; $T_i$ *W(i)*'nin içinde olmalı. Bir kere bütün ***i***'ler için, $T_i$'yi hesapladıktan sonra, *B*'yi $\Theta(n)$ sürede kolaylıkla bulabilirsiniz.

Bütün $T_i$'leri, kayan pencere hesabı ile *O(n)* sürede hesaplayabilirsiniz; her ***i*** için $T_i$ hesabının amortize süresi $O(1)$'dir. Günlerin 2-uçlu kuyruğu (deque) $Q = \langle q_1, q_2, ..., q_m \rangle$ kullanın; burada $q_1$, *W(i)* içindeki en az maliyetli gündür ve her $q_{k>1}$ girdisi, $q_{k-1}$ 'i takip eden *W(i)* içindeki en az maliyetli günlerdir. Bu durumda, $q_1 = T_i$ ve bu değişmez $O(1)$ amortize süresinde aşağıdaki algoritma ile sağlanabilir.

```
    BUYTOMATOES(d, C[1..n])
1   Q ← ∅
2   B[1..n] ← 0
3   for i ← 1 to n
4       do if not EMPTY(Q) and HEAD(Q) ≤ i − d
5             then POP-FRONT(Q)          ▷ Remove old day
6          while not EMPTY and C[i] ≤ C[TAIL(Q)]
7             do POP-BACK(Q)             ▷ Remove non-minimum-cost days
8          PUSH-BACK(Q, i)               ▷ Add current day
9          B[HEAD(Q)] ← B[HEAD(Q)] + 1
10  return B
```

Her i günü, Q'ya bir kez eklenir ve en fazla bir kere Q'dan çıkarılır. Dolayısıyla 2-uçlu kuyruk işlemleri *O(n)*'e mal olur. Her 2-uçlu kuyruk işlemi *O*(1) sürede tamamlanır. Algoritmadaki döngü *n* kere işlediğinden, toplam koşma süresi Θ(n)'dir. Geçmişteki *d* gün için en fazla bir öge olabildiğinden, 2-uçlu kuyruk için *O(min(n, d))* kadar boş yere ihtiyaç olur. Sonuç dizilimi B, Θ(n) kadar yere ihtiyaç duyduğundan toplam kullanılacak alan Θ(n)'dir.

$T_i$ hesaplamalarında amortize sürenin *O*(1) olduğunu göstererek tam puan alacak başka yollar da vardır. Buna karşın, $T_i$'yi *O(log d)* veya *O(d)* gibi sürelerde hesaplayan daha yavaş çözümler gidiş yolundan kısmi puan aldılar; bunlar B'yi her adımda *d* gün ileri bakarak hesaplayan çözümler, dinamik programlama çözümleri ve domates fiyatlarını sıralayıp, düşük fiyattan alınan domates miktarını en fazlaya çıkaran çözümlerdi.