Algoritmalara Giriş Ekim 7, 2005
Massachusetts Institute of Technology 6.046J/18.410J
Professors Erik D. Demaine ve Charles E. Leiserson Dağıtım 12

Problem Seti 2 Çözümler

Problem 2-1. Bu (yaklaşık) sıralanmış mıdır?

Hogwarts sülalesinin çocuk büyücüsü Harry Potter'ı başı bir kez daha derttedir. Profesör Snape Harry'i tutuklamış ve ona son 200 yılın eski ev ödevlerini sıralama işini vermiş. Harry bir büyücü olduğundan elini söyle bir sallayıp —ordinatus sortitus" yani — düzenli sıra" demiş ve kağıtlar hızla kümelenmişler.
Ancak Profesör Snape Harry'nin büyüsünün kağıtları doğru olarak kümelediğini belirlemek istemektedir. Öte yandan n sayıda çok fazla kağıt olduğundan doğru sırada olup olmadıklarını belirlemek Ω(n) süresini alacaktır.
Bunun yerine Profesör Snapekağıtların — *yaklaşık-sıralanmış*" olduğunu kontrol etmeye karar verir. Kağıtların %90'ının sıralandığını bilmek ister: kağıtların %10'u kümeden çıkarılırsa elde edilen liste sıralanmış olabilir mi?
Bu problemde Profesör Snape'e n farklı elemanlı A giriş listesini işleyecek bir algoritma bularak yardım edeceğiz; algoritma aşağıdaki gibi davranacak:

- Eğer A listesi sıralıysa, algoritma her zaman **true/ doğru** çıkışını verecek.
- Eğer A listesi % 90 sıralı değilse, algoritma en az 2/3 olasılıkla **false/yanlış**çıkışını verecek.

(a) Profesör Snape önce şu algoritmayı düşünür:

k kere tekrarla:

1. Kağıtları bağımsız, tekdüze ve rastgele olarak seç ve (1,n) aralığını kullan. (Yani, 1 <i<n.)
2. A[i− 1] ve A[i]kağıtlarını karşılaştır. Eğer doğru sırada değillerse **yanlış** çıktısını ver ve dur.
3. A[i] ve A[i+ 1] kağıtlarını karşılaştır..Eğer doğru sırada değillerse **yanlış** çıktısını ver ve dur.

Doğru çıktısı ver.

Bu algoritmanın sıralama listesinin en az 2/3 olasılıkla yaklaşık-doğru olduğunu keşfedebilmesi için k = Ω(n) gerektiğini gösterin. *İpucu*: Yaklaşık sıralı olmayan bir dizi bulun, ama algoritmanın yanlış çıktı vereceği az sayıda eleman olsun.

Çözüm: Snape'in algoritmasının çalışmadığını, aşağıda belirtilen iki durumun olduğu ters-örnekle gösteririz:

- *A* 90% *sıralı değildir*.

• Snape'in algoritması sadece k = Ω (n) olduğunda, 2/3 olasılıkla yanlış verir. Özellikle aşağıdaki ters-örneği ele alırız:

$$A = [\lfloor n/2 \rfloor + 1, \ldots, n, 1, 2, 3, \ldots, \lfloor n/2 \rfloor] .$$

Ön kuram 1 *A* 90% *sıralı değildir.*

Kanıt. Çelişki yöntemiyle,listenin % 90 sıralı olduğunu varsayın. Öyleyse, bazı elemanlar birbirlerine göre % 90 doğru sıralanmış olmalıdır. Bu elemanların doğru sıralanmış bir kümesi listenin birinci yarısında olmalıdır, yani, *i ≤ [n/2]* dizininde. Ayrıca elemanların doğru sıralanmış bir kümesi listenin ikinci yarısında olmalıdır, yani, *j > [n/2]* dizininde. Ancak yapılandırma nedeniyle *A[i] > A[j]* 'dir ve bu bir çelişki oluşturur. Bu nedenle *A* % 90 sıralı değildir.

Lemma 2 Snape'in algoritması sadece k = Ω (n) olduğunda, 2/3 olasılıkla yanlış verir.

Kanıt. Algoritmanın her döngüsünde, listenin sıralı olmadığını belirleyecek sadece iki seçenek olduğuna dikkat edin: *i = [n/2]* veya *i = [n/2]* + 1.
Göstergesel rastgele değişkenleri aşağıdaki gibi tanımlayın:

$$X_\ell = \begin{cases} 1 & \text{if } i = \lfloor n/2 \rfloor \text{ or } i = \lfloor n/2 \rfloor + 1 \text{ on iteration } \ell, \\ 0 & \text{otherwise.} \end{cases}$$

Bu durumda, *Pr {X_l = 1} = 2/n ve Pr {X_l = 0}= (1 - 2/n)*. olduğuna dikkat edin.
Böylece ore, the probability that Snape'in algoritmasının tüm *k* döngüleri için yanlış çıkışını *vermemesi* (yani Snape'in algoritmasının çalışmaması) olasılığı şudur:

$$\begin{aligned} &= \prod_{\ell=1}^{k} \Pr\left(X_\ell = 0\right) \\ &= \left(1 - \frac{2}{n}\right)^k \end{aligned}$$

Biz Snape'in algoritmasının çalışması için gereken en küçük ***k*** değerini belirlemek istiyoruz, yani, başarısızlık olasılığını 1/3 den büyük yapmayacak en küçük ***k*** değerini:

$$\left(1 - \frac{2}{n}\right)^k \leq \frac{1}{3} .$$

k için çözdüğümüzde, şunu belirleriz:

$$k \geq \frac{\ln(1/3)}{\ln\left(1-\frac{2}{n}\right)}.$$

Şimdi kitabın 3.11 bölümündeki matemetiksel gerçeği hatırlarız:

$$\left(1-\frac{1}{x}\right)^{x} \leq \frac{1}{e}.$$

Buradan şunu hesaplarız:

$$\ln\left(1-\frac{1}{x}\right) \leq -\frac{1}{x}.$$

Sonuçta şu karara varırız:

$$\begin{aligned} k &\geq \frac{\ln(1/3)}{\ln\left(1-\frac{2}{n}\right)} \\ &\geq \frac{\ln(1/3)}{-2/n} \\ &\geq \frac{n\ln 3}{2}. \end{aligned}$$

Snape'in algoritmasının sadece eğer k = $\Omega(n)$ ise doğru olduğu kararına varırız.

(b) Size içinde n sayıda top olan bir torba veriliyor. İçinde en az %10 oranında mavi top olduğu ve kırmızı topların sayısının da % 90'ı geçmediği söyleniyor. Asimptotik olarak (yani n'nin büyük değerleri için) en az 2/3 olasılıkla mavi topu bulanadek torbadan kaç kez top çekmeniz gerekir? (Topların torbadan çıkarıldıktan sonra yeniden torbaya konulduğunu varsayabilirsiniz.)

Çözüm: Soruda sadece çekilen topların asimptotik sayısı sorulduğundan, $\Theta(1)$ (artı bir miktar gerekçelendirme) yeterli bir yanıttır. Aşağıda daha tam bir yanıt veriliyor.
.

Torbadan k sayıda top çektiğinizi varsayın (her top incelendikten sonra torbaya atılıyor).

Ön kuram 3: *Yeterince büyük bir k sabiti için, bir topun mavi çıkma olasılığı en az* 2/3'tür.

Kanıt. Göstergesel rastgele değişkenleri şöyle tanımlayın:

$$X_i = \begin{cases} 1 & eğer\ top\ i\ mavi\ ise \\ 0 & eğer\ top\ i\ kırmızı\ ise \end{cases}$$

Bu durumda Pr X_i = 1 = 1/10 and Pr X_i = 0 = 9/10 olduğuna dikkat edin. Sonra en az bir topun mavi olma olasılığını hesaplarız:

$$\begin{aligned} &= 1 - \prod_{i=1}^{k} \Pr\left(X_i = 0\right) \\ &= 1 - \left(\frac{9}{10}\right)^k \\ &\geq \frac{2}{3}. \end{aligned}$$

Böylece eğer k = lg(1/3)/ lg 0.9 ise, en az bir mavi top çekme olasılığı da en az 2/3 'tür.

(c) Sıralanmamış bir listede bir —ikili arama" yaptığınızı varsayın:

İKİLİ_ARAMA(A, anahtar,sol,sağ) ⇨ Anahtarı A [sol,sağ] içinde Ara

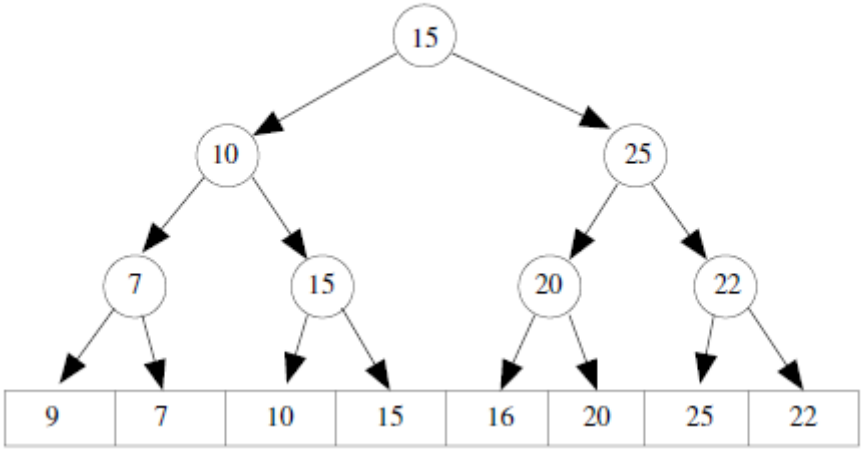
```
1 if left = right
2    then return left
3    else mid ← [(left + right) / 2 ]
4         if key < A [mid]
5            then return BINARY-SEARCH(A,key, left, mid −1)
6            else return BINARY-SEARCH(A, key, mid, right)
```

A'da (sıralı olmasa bile) key_1 için yapılan bir ikili aramanın ***i*** yuvasını çıktı olarak verdiğini farz edin. Benzer biçimde anahtar/key_2 için ikili arama da ***j*** yuvasını versin.

Aşağıdaki gerçeğin neden doğru olduğunu açıklayın:

Eğer i < j ise, $key_1 \leq key_2$ dir. Bir şekil çizin. *İpucu*: Öncelikle bunun A listesinin sıralı olması durumunda neden açıkça doğru olduğunu düşünün.

Çözüm:



Şekil 1: Sırasız bir dizilimin ikili-arama karar ağacının bir örneği .

Bu problemi anlamak amacıyla ikili arama algoritmasını karar-ağacı versiyonunu düşünün (sıralama algoritmalarının aksine ikili arama için karar ağacı oldukça küçüktür, yani O(n) düğüm vardır. Şekil1' deki örneği düşünün; bu örnekte ikili arama sırasız bir dizilimde [9 7 10 15 16 20 25 22] yapılıyor. key_1=*20 ve* key_2 = 25 olduğunu varsayın. Hem 20 hem de 25, ≥ 25 olduğundan kökten sağa doğru giden dalı seçin. Bu noktada iki ikili aramada ıraksar: 20 < 25 ve 25 ≥ 25. Böylece key_1 sol dalı ve key_2 de sağ dalı seçer. Bu, işin sonunda key_1 *ve* key_2 nin doğru sıralanmasını sağlar. Şimdi tartışmayı genelleyeceğiz. key_1 için, ikili aramanın dördüncü satırında (burada k = O(lg n)), x_1, x_2,….x_k , key_1 ile karşılaştırılan k sayıda eleman olsun.
(örnek x_1 = 15, x_2 = 25, ve x_3 = 20). y_1,y_2,…..y_t, key_2 ile karşılaştırılan t sayıda eleman olsun. $x_1 = y_1$ olduğunu biliyoruz. $X_\ell \neq y_\ell$ durumunu sağlayan en küçük sayı ℓ olsun. (özellikle $\ell > 1$ için). i < j olduğundan varsayım gereği key_2 sola dallanırken key_1 sağa dallanamaz. Böylece şu sonuca varırız.

$$key_1 < X_{\ell-1} = y_{\ell-1} \leq key_2 .$$

(Bu problem için daha az resmi bir çözüm yeterliydi çünkü sizden bunun 'neden doğru'' olduğunu açıklamanızı istemiştik. Yukardaki tartışma daha dikkatlice resmi hale getirilebilir).

(d) Profesör Snape listenin % 90 sıralı olduğunu belirlemek için rastgele bir algoritma öneriyor.. Algoritma bağımsız ve tekdüze bir tamsayıyı [1,n] kapalı aralığında rastgele seçmek için (Rastgele)/RANDOM (1,n) fonksiyonunu kullanıyor. Algoritma aşağıda verilmiş:

IS-ALMOST-SORTED(A, n, k) ⇨ Determine if A[1 ..n] is almost sorted.
(Yaklaşık–Sıralı) (*A[1,n]'nin Yaklaşık-Sıralı olduğunu belirleyin*)

```
1 for r ← 1 to k
2     do i ← RANDOM(1,n)      ⇨   Pick i uniformly and independently.
                                  (i'yi tekdüze ve bağımsız seçin)
3        j ← BINARY-SEARCH(A,A[i],1,n)
4        if  i ≠ j
5           then return false          (öyleyse,"yanlış" döndür)
6 return true                          ( "doğru" döndür )
```

Algoritmanın, k yeterince büyük bir sabitse geçerli olacağını gösterin. Yani k uygun seçildiğinde algoritma liste doğru sıralanmışsa her zaman **doğru**'yu çıktı olarak veriyor ve liste %90 doğru sıralanmamışsa, 2/3 olasılıkla **yanlış**ı çıktı olarak veriyor.

Çözüm: *Kısa açıklama*: Algoritmanın doğruluğunu göstermek için iki ana önkuramın kanıtlanması gerekir. (1) *A* listesi sıralıysa, algoritma her zaman **doğru** döndürür; (2) Eğer *A* listesi %90 *sıralı değilse*, algoritma en az 2/3 olasılıkla **yanlış** döndürür. Kolay olan önkuramdan başlayacağız ve bu önkuram temelde ikili aramanın doğru olduğunu savunur. Sonra da eğer liste %90 sıralı değilse %10 kadar elemanın _'ikili arama testini'' geçemeyeceğini göstereceğiz. Sonunda yeterince büyük bir *k* sabiti için, eğer liste %90 sıralı değilse, algoritmanın en az 2/3 olasılıkla **yanlış** çıkışını vereceği sonucuna varacağız.

Önkuram 4 *Eğer A listesi sıralıysa algoritma her zaman **doğru** döndürür.*

Kanıt. Etüt 1' de gösterildiği gibi, bu önkuram sıralı bir listede ikili aramanın doğruluğu ilkesine dayanır. Değişmez, *anahtarın sol* ile *sağ* arasındaki *A* diziliminde olduğudur.

Problemin geri kalan kısmında elemanları ikili sıralama testini geçip geçmediklerine bağlı olarak "iyi'' ve "kötü'' olarak etiketleyeceğiz.

$$label(i) = \begin{cases} iyi & eğer\ \ i = \ \ İKİLİ - ARAMA\ (A, A[i], 1, n) \\ kötü & eğer\ \ i \neq \ \ İKİLİ - ARAMA\ \ (A, A[i], 1, n) \end{cases}$$

Ilk bakışta hangi elemanların iyi hangilerinin kötü olduğunun anında anlaşılabilir olmadıklarına dikkat edin. Özellikle bazı elemanlar doğru sıralanmış görünmekle birlikte diğer elemanlar yanlış sıralandığı için kötü olabilirler. Benzer şekilde bazı elemanlar hiç yerlerinde değilmiş gibi görünür ama diğer yanlış yerleştirilmiş elemanlar nedeniyle iyi olabilirler. Kanıttaki kilit nokta kötü sıralanmış bir listenin bir sürü kötü elemanı olduğunu göstermektir.

Önkuram 5 *Eğer A listesi %90 sıralı değilse elemanların en az %10"u kötüdür.*

Kanıt. Çelişki yöntemiyle %10' dan daha az sayıda elemanın kötü olduğunu varsayın. Öyleyse en azından elemanların %90' ı iyidir. %90 sıralı bir listenin tanımını hatırlayın: eğer elemanların %10'u çıkarılırsa kalan elemanlar sıralı düzendedir. Onun için dizilimden bütün kötü elemanları çıkarın. Şimdi, kalan elemanların sıralı düzende olduğunu söyleyebiliriz. Kalan iyi elemanlardan herhangi iki tanesini, key_1 *ve* key_2"yi düşünün; key_1 dizin *i* de ve key_2 dizin *j*' dedir. Eğer i < j ise şık(c) $key_1 \leq key_2$ olduğunu gösterir. Benzer şekilde eğer *j* < *i* ise şık(c) $key_2 \leq key_1$. Yani 2 eleman da doğru sıralanmış düzendedir. Tüm ikili eleman çiftleri sıralanmış düzende olduğundan iyi elemanların dizilimi sıralanmış düzendedir.

Birçok kötü eleman olduğunu bir kez gösterdikten sonra geriye kalan, kötü bir elemanı rastgele örnekleme ile bulabileceğimizi göstermektir.

Önkuram 6 *A listesi %90 sıralı değilse algoritma en az 2/3 olasılıkla* ***yanlış*** *döndürür.*

Kanıt. Önkuram 5' den elemanların en az %10' unun kötü olduğunu biliyoruz. Şık(b)' den eğer *k* > lg(1/3) / lg 0.9 seçersek 2/3 olasılıkla bir kötü eleman bulacağımızı biliyoruz. Bu nedenle algoritmanın en az 2/3 olasılıkla **yanlış** döndüreceği sonucuna varırız.

(e) Profesör Snape'in listenin 0 < ϵ < 1 arası bir epsilon değerinde 1 − ϵ oranında sıralanmış olduğunu kontrol etmek istediğini düşünün. (Önceki örneklerde ϵ = 0.10 idi. Büyük *n* değerleri için asimptotik olarak *k* değerini belirleyip algoritmanın doğru olduğunu gösterin. Toplam koşma süresi nedir?

Çözüm: Önkuram 4 Şık (d)'dekiyle aynıdır. Önkuram 5'te yapılan küçük bir değişiklikle, eğer dizilim *(1 — ϵ)-sıralı* değilse, o zaman en az *ϵn* kötü elemanın olduğu ve otherwise, kalan *(1 — ϵ)n* elemanın bir (1 — ϵ)-sıralı liste oluşturacağı gösterilir. En sona ***k*** için uygun bir değerin belirlenmesi kalır. Bu durumda ***k*** yı şöyle olacak şekilde seçmek isteriz:

$$(1-\epsilon)^k \leq \frac{1}{3}$$

$k = c / \epsilon$, seçeriz ve (kitaptaki 3.11'i kullanarak) şu sonuca varabiliriz:

$$\begin{aligned}(1-\epsilon)^k &\leq \left((1-\epsilon)^{1/\epsilon}\right)^c \\ &\leq \left(\frac{1}{e}\right)^c\end{aligned}$$

Böylece eğer $k = \Theta(1/\epsilon)$, ise, algoritmanın kötü bir elemanı en az 2/3 olasılıkla bulacağı sonucuna varırız. Algoritmanın koşma süresi $O(\lg n / \epsilon)$'dir.

Problem 2-2. Yaklaşık sıralanmış bir listeyi sıralamak.

Tutulduğu yerden dönerken Harry arkadaşı Hermione ile karşılaşır. Harry, sıralama büyüsünün tutmadığını Profesör keşfettiği için mutsuzdur. Kağıtları doğru sıralamak yerine her kağıt doğru sırada bir yuvanın içinde yerleşmiştir. Hermione hemen araya yerleştirme sıralamasının bu sorunu kolayca çözeceğini söyler. Bu problemdeHermione'nun (çoğunlukla olduğu gibi) haklı olduğunu göstereceğiz. Önceki gibi, n farklı elemandan oluşan A[1..n] diziliminde..

(a) Önce bir evirme (inversion) tanımlayacağız. Eğer i < j ve A[i] > A[j] ise, (i, j) ikilisine A'nın evirmesi denir. { 1,2,…n} diziliminin hangi devşiriminde (permutation) en çok evirme vardır? Bunlardan kaç tane vardır?

Çözüm: *{n,n* - 1,..., 2,1} permütasyonu en fazla evirme sayısına sahiptir. Bundaki evirme sayısı şudur:

$\binom{n}{2}$ = *n(n - 1) / 2*

(b) Eğer her kağıt başlangıçta doğru konumdaki k yuvada ise, araya yerleştirme sıralamasının koşma süresi O (nk) dır. *İpucu*: Önce ARAYA-YERLEŞTİRME (A)'nın koşma süresinin O (n+I) olduğunu gösterin; burada A'daki evirmelerin sayısıdır.

Çözüm: *Kısa açıklama*: Önce, araya yerleştirme sıralaması algoritmasını inceleyerek, INSERTİON-SORT (A) [Araya yerleştirme sıralaması] nın koşma süresinin *O(n+ I),* olduğunu gösteririz; burada ***I*** evirmelerin sayısıdır. Sonra içindeki her elemanın doğru konumdaki ***k*** yuvada olduğu bir dizilimde, olası evirmelerin sayısını buluruz. En çok *O(nk)* evirme olduğunu gösteririz.

Önkuram 7 INSERTİON-SORT (A)'nın koşma süresi *O(n + I); burada I, A'nın içindeki evirmelerin sayısıdır.*

Kanıt. Bir A diziliminde INSERTION-SORT uygulamasını düşünün.. Dış döngüde *O(n)* iş yapılır. İç döngünün her tekrarı tam olarak bir evirmeyi düzeltir. Algoritma sona erdiğinde hiç evirme kalmaz. Bu nedenle iç döngüde ***I*** sayıda tekrar olmalıdır ve bu da *O(I*) iş demektir. Böylece algoritmanın koşma süresi *O(n +I)* olur.

Sonra içindeki her elemanın doğru konumdaki ***k*** yuvada olduğu bir dizilimde evirmeleri sayarız.

Önkuram 8 *Eğer h*er eleman *doğru konumdaki **k** yuvada ise, o zaman en çok O(nk) evirme vardır.*

Kanıt. Evirmelerin sayısına bir üst sınır getiririz. Belirli bir *A[i]* elemanını düşünün. *A[i]* elemanı ile evrilebilecek, özellikle *A[i - 2k. . i + 2k]* aralığında en çok 4k eleman vardır . Bu nedenle, *i* en fazla *4k* evirmenin parçası olabilir be böylece en çok *4nk* evirme vardır.

Bundan yola çıkarak, her elemanının doğru konumdaki **k** yuvasında olduğu bir dizilimde araya yerleştirme sıralamasının koşma süresinin *O(nk)* olduğu sonucuna varırız..
Ek not olarak, bunu evirmeleri kullanmadan, araya yerleştirme sıralamasının iç döngüsünün hiçbir zaman bir elemanı 4k yuvadan daha öteye taşıyamayacağını göstererek, doğrudan kanıtlamak mümkün gibi görünür. Ancak bu göründüğü kadar kolay değildir: her ne kadar bir eleman son konumundan k yuva ötede işleme girese de, hiç bir zaman daha öteye gidemeyeceğini göstermek gereklidir. Örneğin, ya bir eleman önce *k* + 2 yuva geriye, sonra da 3 yuva ileriye giderse? Öte yandan, belki hiçbir zaman *4k* yuvadan daha öteye gidemeyeceği gösterilebilir.

(c) Her kağıdın doğru konumdaki *k* yuvada olduğu bir listede sıralama için Ω (n lg k) sayıda karşılaştırma gerektiğini gösterin. *İpucu*: Karar ağacı tekniğini kullanın.

Çözüm: Dizilimi sıralamanın Ω *(n lg n)* karşılaştırma gerektirdiğini zaten biliyoruz. Eğer *k > n / 2* ise, o zaman *n lg n* = Ω *(n lg k)* olur ve kanıt tamamlanır. Kanıtlamanın geri kalan kısmı için *k ≤ n / 2* olduğunu varsayın.

Hedefimiz, her elemanının doğru konumdaki ***k*** yuvasında olduğu bir dizilimi sıralayan algoritmanın karar ağacındaki yapraklarının sayısına bir alt-sınır getirmektir. Bu nedenle, bu koşulu sağlayan olası permütasyonlara bir alt sınır belirleriz.

Önce, *n* boyutlu dizilimi, her biri *k* boyutunda *[n / k]* bloğa parçalayın ve kalan *n(mod k)* boyutunda olsun. Her blok için ***k!*** permütasyon vardır ve bu da tüm dizilim için en az toplam $(k!)^{[n/k]}$ permütasyon sonucunu verir. Bu permütasyonlardan hiç biri bir elemanı *k* yuvadan fazla değiştirmez.

Bunun permütasyonların toplam sayısını olduğundan az gösterdiğini dikkate alın çünkü, hiçbir eleman bir *k-elemanlı* bloktan diğerine gitmez ve kalan bloktaki elemanların permütasyonunu görmezden geliriz.

Böylece karar ağacının en az $(k!)^{[n/k]}$ yaprağı olduğu sonucuna varırız. Karar ağacı bir ikili ağaç olduğundan, ağacın yüksekliğinin aşağıdaki gibi olduğuna karar veririz:

$$
\begin{aligned}
&\geq \lg\left((k!)^{\lfloor n/k \rfloor}\right) \\
&\geq \left\lfloor \frac{n}{k} \right\rfloor \lg(k!) \\
&\geq \left\lfloor \frac{n}{k} \right\rfloor (c_1 k \lg k) \\
&\geq c_1(n-k)\lg k \\
&\geq \frac{c_1 n \lg k}{2} \\
&= \Omega(n \lg k)
\end{aligned}
$$

(Son adımın gerekçesi $k \leq n/2$ olduğu varsayımımızdır.*)*

(d) Alt sınırı karşılayacak bir algoritma tasarlayın; yani her kağıdın doğru konumdaki k yuvada olduğu bir listede sıralamayı Θ(n lg k) sürede yapsın. *İpucu*: Kitabın 142. sayfasındaki 6.5-8 no.lu probleme bakın.

Çözüm: Bu problemin çözümünde bir yığın kullanacağız. Aşağıdaki altyordamları olan bir yığınımız olduğunu varsayıyoruz:

- MAKE-HEAP () yeni bir boş yığın döndürür.
- INSERT *(H, key, value)* anahtar/değer çiftini yığında araya yerleştirir.
- EXTRACT-MIN(H) en küçük anahtarlı anahtar/ değer çiftini yığından çıkarır, değrini döndürür.

Önce, *t* sıralı listeyi birleştirme problemini düşünün. Her biri sıralı *A1,...*, A_t, listemiz olduğunu farz edin. Aşağıdaki stratejiyi kullanırız (sözde kodu daha aşağıda):

1. Yeni bir *H* yığını yarat.
2. Her *t* listesi için, ilk elemanı listed araya yerleştir. Liste *i* için INSERT(H, A_t[1], i) uygula.
3. *n* kere tekrarla:

   (a) EXTRACT-MIN *(H)* kullanarak en küçük elemanı yığından çıkar. Listenin kimliği olan ***v,*** döndürülen değer olsun.

   (b) Liste ***v***'den çıkarılan elemanı yeni bir dizilimde sıraya koy.

   (c) Liste ***v***'deki sonraki elemanı araya yerleştir.

Anahtar değişmez, döngünün her tekrarından sonra, yığının her listedeki en küçük elemanıa sahip olduğunun gösterilmesidir. (Resmi bir tümevarım kanıtı vermiyoruz çünkü soruda sadece bir algoritma tasarlamanız istenmişti.) Her EXTRACT-MIN ve INSERT işleminin *O(lg k)* süre gerektirdiğine dikkat edin, çünkü yığında hiç bir zaman *2k*'dan fazla eleman olmaz. Döngü sadece sabit bir miktarda başka iş yaptığından ve *n* kere tekrarlandığından, sonuçta koşma süresi *O(n* lg *k)* olur.

Bunu problemimize uyarlamak için *A*'yı bir sıralı listeler kümesi olarak düşünürüz. Özellikle tüm *i* ≤ *n - 2k - 1*'ler için, *A[i] < A[i + 2k* + 1] olduğuna dikkat edin: *i* yuvasındaki eleman, sıralama sırasında en fazla *i + k* yuvasına kadar ileri gidebilir ve *i + 2k* + 1 yuvasındaki eleman en fazla

i + *k* + 1 yuvasına kadar geri gidebilir.

Şu an için, *n* 'nin *2k „ya* bölünebileceğini farz edin. Aşağıdaki gibi tanımlanan *t* = *2k* listelerini düşünelim:

$$A_1 = \left[A[1], A[2k+1], A[4k+1], \ldots A[n-(2k-1)]\right]$$
$$A_2 = \left[A[2], A[2k+2], A[4k+2], \ldots A[n-(2k-2)]\right]$$

… A_t = [A[2k], A[4k], A[6k],…, A[n]]

Bu listelerin her biri sıralıdır ve herbirinin boyutu n'ye küçük-eşittir. Bu nedenle bu listeleri yukarıdaki yöntemi kullanarak *O(n* lg *k)* sürede sıralayabiliriz. Şimdi daha detaylı sözde kodunu veriyoruz:

SORT-ALMOST-SORTED *(A, n, k)* ⇨ Eğer her eleman doğru konumundan k yuvadan daha fazla uzak değilse, A'yı sırala.

```
1  H ← MAKE-HEAP ()
2  for i ← 1 to 2k
3       do  INSERT(H, A[i], i)
4  for i ← 1 to n
5       do j ← EXTRACT-MIN(H)
6          B[i] ← A[j]
7          If j + 2 k ≤ n
8            then INSERT(H, A[j + 2k], j)
9  return B
```

Hatırlamaniz gereken şey, yığın işlemlerini açıklarken her ne kadar ~~-d~~değer"I ihmal etsek de, bir yığın genelde bir anahtarı ve onun ilişkili değerini depolamak için kullanılır. Bu durumda, anahtar *A [j]* elemanıyken, değer bir *j* dizinidir. Bunun sonucunda, yığın dizilimdeki sonraki en küçük elemanın dizinini döndürür.

Doğruluk ve performans yukarıdaki tartışmadan çıkar.

Bu problem çözmenin ikinci bir yolu olduğunu fark edin. *n* (ortak) elemanı olan iki sıralı listeyi *O(n)* sürede birleştirmeyi zaten bildiğimizi hatırlayın. Öyleyse listeleri bir turnuva" ile birleştirmek mümkündür. *k* = 8 durumu için bir örnek veriyoruz, burada *A* + *B,* *A* ve *B* listelerini birleştirmek anlamında kullanılıyor:

$$\begin{aligned}
\text{Round 1:} &\quad (A_1 + A_2),\ (A_3 + A_4),\ (A_5 + A_6),\ (A_7 + A_8) \\
\text{Round 2:} &\quad (A_1 + A_2 + A_3 + A_4),\ (A_5 + A_6 + A_7 + A_8) \\
\text{Round 3:} &\quad (A_1 + A_2 + A_3 + A_4 + A_5 + A_6 + A_7 + A_8)
\end{aligned}$$

Dikkat ederseniz lg *k* birleştirme adımı var ve her biri *n* elemanı (*k* listeye dağıtılmış) birleştiriyor ve böylece maliyet *O(n)* oluyor. Bu da istenen *O(n lg k)* koşma süresini sağlıyor .

Problem 2-3. Ağırlıklı Ortanca

n sayıda farklı $x_1, x_2,..., x_n$elemanı var ve pozitif ağırlıkları $w_1, w_2,..., w_n$; $\sum_{i=1}^{n} w_i=1$, teriminde ağırlıklı (alt) ortanca aşağıdaki durumları karşılayan x_k dır.

$$\sum_{x_i<x_k} w_i < \frac{1}{2}$$

$$\sum_{x_i>x_k} w_i \leq \frac{1}{2}.$$

(a) $x_1, x_2,..., x_n$ 'nin ortancasının, w_i =1/n, i =1,2,...,n için ağırlıkları olan $x_1, x_2,..., x_n$'nin ağırlıklı ortancası olduğunu savunun..

Çözüm: x_k ; $x_1, x_2,..., x_n$'nin ortancası olsun. Ortancanın tanımı gereği x_k , x_i 'nin tam olarak diğer $\left\lfloor \frac{n+1}{2} \right\rfloor - 1$ elemanından daha büyüktür. Öyleyse x_k „dan küçük olan elemanların ağırlıklarının toplamı:

$$\begin{aligned}
\sum_{x_i<x_k} w_i &= \frac{1}{n} \cdot \left(\left\lfloor \frac{n+1}{2} \right\rfloor - 1 \right) \\
&= \frac{1}{n} \cdot \left\lfloor \frac{n-1}{2} \right\rfloor \\
&\leq \frac{n-1}{2n} \\
&< \frac{n}{2n} \\
&< \frac{1}{2}
\end{aligned}$$

olur. Tüm elemanlar birbirinden farklı olduğundan, x_k aynı zamanda tam olarak başka $n - \left\lfloor \frac{n+1}{2} \right\rfloor$ elemandan daha küçüktür. Bu nedenle;

$$\begin{aligned}\sum_{x_i > x_k} w_i &= \frac{1}{n} \cdot \left(n - \left\lfloor \frac{n+1}{2} \right\rfloor\right) \\ &= 1 - \frac{1}{n} \cdot \left\lfloor \frac{n+1}{2} \right\rfloor \\ &\leq 1 - \left(\frac{1}{n}\right)\left(\frac{n}{2}\right) \\ &\leq \frac{1}{2}\end{aligned}$$

olur. Sonuçta ağırlıklı ortancanın tanımı gereği, x_k da ağırlıklı ortancadır.

(b) En kötü durum O (n lg n) süresinde ağırlıklı ortancanın sıralama kullanarak nasıl hesaplanacağını gösterin.

Çözüm: *n* elemanın ağırlıklı ortancasını hesaplamak için, elemanları sıralarız ve sonra da ortancayı bulana kadar elemanların ağırlıklarını toplarız.

```
1  WEIGHTED-MEDIAN(A)
2  k ← 1
3  s ← 0
4  while s + w_k < 1/2
5        do s ← s + w_k
6           k ← k + 1
7  return x_k
```

⇨ s = Tüm $x_i < x_k$'*ların toplam ağırlığı*

Bu algoritmanın döngüsünün değişmezi, s'nin x_k 'dan küçük olan tüm elemanların ağırlıklarının toplamı olduğudur *:*

$$s = \sum_{x_i < x_k} w_i$$

Bunun doğru olduğunu tümevarımla kanıtlarız. Taban durumu doğrudur çünkü ilk döngüde *s* = 0'dır. Liste sıralı olduğundan tüm $i < k$ için, $x_i < x_k$ 'dir. Tümevarımla, *s* doğrudur çünkü, döngüdeki her tekrarda *s* bir sonraki elemanın ağırlığı kadar artar. Döngünün sona ereceği garantilidir, çünkü tüm elemanların ağırlıklarının toplamı 1'dir. Ağırlıklı ortancanın tanımını kullanarak, döngü sona erdiğinde x_k 'nın ağırlıklı ortanca olacağını kanıtlarız.

s'nin döngünün sondan bir önceki tekrarının başlangıcındaki değeri s' olsun: $s = s' + W_{k-1}$. Sondan bir önceki döngü sonlanma koşulunu karşılamadığından, biliriz ki:

$$
\begin{aligned}
s' + w_{k-1} &< \frac{1}{2} \\
\sum_{x_i < x_k} w_i = s &< \frac{1}{2}
\end{aligned}
$$

Dikkat ederseniz, döngünün o tekrarı olsa bile bu doğrudur, çünkü $s = 0 < 1/2$. Bu ağırlıklı ortanca olmanın ilk koşulunu kanıtlar. Sonra ikinci koşulu kanıtlarız. x_k 'dan büyük elemanların ağırlıklarının toplamı şudur:

$$
\sum_{x_i > x_k} w_i = 1 - \left(\sum_{x_i < x_k} w_i \right) - w_k = 1 - s - w_k
$$

Döngünün sonlanması koşulu nedeniyle,

$$
\begin{aligned}
\sum_{x_i < x_k} w_i = s &\geq \frac{1}{2} - w_k \\
-s &\leq -\frac{1}{2} + w_k
\end{aligned}
$$

$$
\begin{aligned}
1 - s - w_i &\leq \frac{1}{2} \\
\sum_{x_i > x_k} w_i &\leq \frac{1}{2}
\end{aligned}
$$

olur. Böylece x_k ağırlıklı ortanca olmanın ikinci koşulunu da karşılar. Bu nedenle x_k ortancadır ve algoritma doğrudur.

Algoritmanın koşma süresi, dizilimi sıralamak için gerekli zaman artı ortancayı bulmak için gereken zamandır. Dizilimi *O(n lg n)* sürede sıralayabiliriz. WEİGHTED-MEDİAN 'daki döngünün *O(n)* sayıda tekrarının her biri *O(1)* süre gerektirdiğinden toplam koşma süresi *O(n* lg *n)*'dir.

(c) Kitabın Bölüm 9.3'ündeki SELECT (SEÇ) gibi bir komutu bir doğrusal-zaman ortanca algoritması kullanarak ağırlıklı ortancayı Θ(n) en kötü sürede nasıl hesaplayacağınızı gösterin.

Çözüm: Ağırlıklı ortanca, Θ(n) en kötü süresinde, bir Θ(n)-süreli ortanca algoritmasıyla hesaplanabilir. Temel strateji ikili aramanın benzeridir: algoritma ortancayı hesaplar ve girişin ağırlıklı ortancayı barındıran yarımında özyineleme yapar.

```
1  LINEAR-TIME-WEIGHTED-MEDIAN(A, l)
2  n ← length[A]
3  m ← MEDIAN(A)
4  B ← ∅                    ▷ B = {A[i] < m}
5  C ← ∅                    ▷ C = {A[i] ≥ m}
6  w_B ← 0                  ▷ w_B = total weight of B
7  if length[A] = 1
8     then return A[1]
9  for i ← 1 to n
10       do if A[i] < m
11             then w_B ← w_B + w_i
12                  Append A[i] to array B
13             else Append A[i] to array C
14 if l + w_B > 1/2         ▷ Weighted median ∈ B
15    then LINEAR-TIME-WEIGHTED-MEDIAN(B, l)
16    else LINEAR-TIME-WEIGHTED-MEDIAN(C, w_B)
```

Bu algoritmadaki ilk çağrı LINEAR-TIME-WEIGHTED-MEDIAN*(A, 0).* Bu algoritmada, *A* başlangıç girdisinin ortancasını içeren bir dizilimdir ve *l, A*‟nın tüm elemanlarından daha küçük olan başlangıç girdisinin elemanlarının toplam ağırlığıdır. *B*, ortancadan küçük tüm elemanları içerir, *C* ortancaya büyük ya da eşit tüm elemanları içerir, ve *wB, B*‟ deki elemanların toplam ağırlığıdır.

Bu algoritmanın doğruluğunu kanıtlamak için, aşağıdaki önkoşulun her özyinelemeli arama için geçerli olduğunu gösteririz: başlangıçtaki *A*‟ nın, ağırlıklı ortancası olan *y*, *A*‟ nın özyinelemeli aramalarında her zaman vardır ve *l, A*‟nın tüm elemanlarından daha küçük olan tüm x_i elemanlarının toplam ağırlığıdır. Bu önkoşul başlangıç araması için açık bir şekilde doğrudur. Bu önkoşulun her özyinelemeli arama için de doğru olduğunu tümevarım yöntemiyle kanıtlayabiliriz. Tümevarım yaklaşımıyla önkoşulun doğru olduğunu farzedelim.

Öncelikle; $l + w_B > ½$ olduğu durumu ele alalım. *y*‟ nin *A*‟ da olması gerektiğinden, 14. satırda *y,* ya *B* ya da *C*‟ de olmalıdır. C‟ deki bir elemandan küçk olan tüm elemanların toplam ağırlığı 1/2‟ den büyük olduğundan, tanım gereği, ağırlık ortancası C‟ de olamaz yani B‟ de olmalıdır. Ayrıca B‟ deki bir elemandan daha küçük hiçbir elemanı atmadık, o halde l doğrudur ve önkoşul karşılanır. Eğer $l + w_B \leq ½$ 14. satırdaysa ise, *y C*‟ de olmalıdır. *C* nin tüm elemanları, *B*‟ nin tüm elemanlarından büyük olduğundan *C*‟ nin elemanlarından

küçük olan elemanların toplam ağırlığı $l + w_B$"dir ve özyinelemeli çağrının önkoşulu yine karşılanır. Bu nedenle tümevarımsal olarak önkoşul her zaman doğrudur.

Bu algoritma her zaman sonlanır çünkü *A* 'nın boyutu her özyineleme çağrısında küçülür. Algoritma sonlandığında, sonuç doğrudur. Ağırlıklı ortanca her zaman *A*" nın içinde olduğundan, tek eleman kaldığında onun ağırlıklı ortanca olması gerekir. Algoritmanın koşma süresi $\Theta(n)$' dir. Ortancayı hesaplamak ve *A* 'yı *B* ile *C* 'ye bölmek $\Theta(n)$ süresi alır. Her özyinelemeli çağrı dizilimin boyutunu *n*'den $[n/2]$ 'ye düşürür. Böylece yineleme $T(n) = T(n/2) + \Theta(n) = \Theta(n)$ olur.

(d) Postane yeri problemi şöyle tanımlanır. Bize ilintili ağırlıkları $w_1, w_2, ..., w_n$ olan *n* sayıda $p_1, p_2, ..., p_n$ noktası veriliyor. Biz *a* ile *b* noktalarının arasındaki uzaklığın $d(a,b)$ olarak gösterildiği durumda $\sum_{i=1}^{n} w_i d(p,p_i)$ toplamını en aza indirecek bir *p* noktası bulmak istiyoruz. (giriş noktalarından biri olması şart değil).

Ağırlıklı ortancanın tek-boyutlu postane yeri problemi için en iyi çözüm olduğunu savunun; burada noktalar basit gerçek sayılar ve *a* ile *b* noktası arasındaki uzaklık $d(a,b)=|a-b|$.

Çözüm:

Tek-boyutlu postane yeri probleminin çözümünü, noktaların ağırlıklı ortancası olduğunu savunuruz. Postane yeri probleminin hedefi, maliyeti en aza indirecek *p* noktasını seçmektir;

$$c(p) = \sum_{i=1}^{n} w_i d(p, p_i)$$

$C(p)$' yi *p*"den küçük olan noktaların ve *p*' den büyük olan noktaların maliyetlerinin toplam katkısı olarak yeniden yazabiliriz:

$$c(p) = \left(\sum_{p_i<p} w_i(p - p_i) \right) + \left(\sum_{p_i>p} w_i(p_i - p) \right)$$

Dikkat ederseniz, eğer bir k için $p=p_k$ ise o nokta maliyete katkıda bulunmaz. Bu maliyet fonksiyonu süreklidir çünkü tüm *x*' ler için, $lim_{p \to x} c(p) = c(x)$ olur. Bu fonksiyonun en küçük değerini bulmak için *p*"ye göre türevini alırız:

$$\frac{dc}{dp} = \left(\sum_{p_i<p} w_i\right) - \left(\sum_{p_i>p} w_i\right)$$

Bu türevin $p=p_i$ olduğu bir i değerinde tanımsız olduğuna dikkat edin çünkü $c(p)$' nin sol ve sağ taraftaki limitleri farklıdır. Ayrıca $\frac{dc}{dp}$ nin azalmayan bir fonksiyon olduğuna dikkat edin çünkü p arttıkça $p_i < p$ noktalarının sayısı azalamaz.
$p < min(p_1, p_2,\ldots, p_n,)$ için $\frac{dc}{dp} < 0$ *ve* $p > max(p_1, p_2,\ldots,p_n)$için $\frac{dc}{dp} > 0$ olur. Bu nedenle öyle bir p^* noktası vardır ki $p<p^*$ noktaları için $\frac{dc}{dp} \leq 0$ ve $p>p^*$ noktaları için $\frac{dc}{dp} \geq 0$ olur ve bu nokta evrensel minimumdur. Biz ağırlıklı ortanca olan y' nin böyle bir nokta olduğunu gösteririz. p' nin ağırlıklı ortanca olmadığı tüm $p<y$ noktaları için ve bir i değerinde $p \neq p_i$ değilse:

$$\sum_{p_i<p} w_i < \sum_{p_i>p} w_i$$

Bunun anlamı $\frac{dc}{dp} < 0$ olduğudur. Benzer şekilde p' nin ağırlıklı ortanca olmadığı $p>y$ noktalarında ve bir i için $p \neq p_i$ olduğunda;

$$\sum_{p_i<p} w_i > \sum_{p_i>p} w_i$$

olur.

Bunun anlamı $\frac{dc}{dp} > 0$ olduğudur. Bir i değeri için $p = p_i$ ve $p \neq y$, durumlarında $\frac{dc}{dp}$ nin hem sol hem de sağ taraf limitlerinin işareti her zaman aynıdır; Böylece aynı sav geçerlidir. Bu nedenle ağırlıklı ortanca olmayan tüm p' ler için $c(p) > c(y)$ olur ve böylece ağırlıklı ortanca y bir evrensel minimumdur.

(e) İki boyutlu postane yeri problemi için en iyi çözümü bulun.; burada noktalar (x,y) koordinat çiftleri ve $a = (x_1,y_1)$ ve $b = (x_2, y_2)$ noktaları arasındaki uzaklık, $d(a,b)=|x_1 - x_2| + |y_1 - y_2|$ ile verilen Manhattan distance (uzaklığı)dır.

Çözüm:

Iki boyutlu postane yeri problemini Manhattan uzaklığı yöntemiyle çözmek, tek-boyutlu postane yeri problemini her boyut için ayrı ayrı çözmekle aynıdır.
Çözüm ; $p = (p_x, p_y)$ olsun. Dikkat ederseniz Manhattan uzaklığı yöntemindeki maliyet fonksiyonunu, iki tane tek boyutlu postane yeri maliyet fonksiyonu olarak aşağıdaki gibi yazabiliriz;

$$g(p) = \left(\sum_{i=1}^{n} w_i |x_i - p_x|\right) + \left(\sum_{i=1}^{n} w_i |y_i - p_y|\right)$$

Dikkat ederseniz $\frac{\delta g}{\delta p_x}$, giriş noktalarının *y* koordinatlarına bağımlı değildir ve önceki şıktaki gibi girişin sadece *x* koordinatlarına bağlı olan $\frac{dc}{dp}$ formundadır. Benzer şekilde $\frac{\delta g}{\delta p_x}$, sadece *y* koordinatına bağımlıdır. Bu nedenle g(p)‘ yi en aza indirmek için iki boyuttaki maliyetleri bağımsız olarak en aza indirgeyebiliriz. İki boyutlu problemin en iyi çözümü; p_x‘ in $x_1, x_2, \ldots, x_n$ girişleri için tek boyutlu postane yeri probleminin çözümü olması ve p_y nin de $y_1, y_2, \ldots, y_n$ girişleri için tek boyutlu postane yeri probleminin çözümü olmasıdır.