*Algoritmalara Giriş* Kasım 4, 2005
Massachusetts Institute of Technology 6.046J/18.410J
Profesör Erik D. Demaine ve Charles E. Leiserson Çalışma notu 21

# Problem Seti 5 Çözümler

## Problem 5-1. Atlama Listeleri ve B-ağaçları

Sezgisel olarak önceden gördüğünüz bir elemanın yakınındaki bir elemanı görmek daha kolaydır. Bir dinamik veri yapısında ***–x' den- y' ye bir yoklama araması*** aşağıdaki sorgulamadır:

Bir veri yapısında *x* elemanını depolayan yuva verildiğinde, ve başka bir y elemanı verildiğinde veri yapısında *y'* yi depolayan yuvayı bulun. Atlama listeleri hızlı yoklama aramalarını aşağıdaki bağlamda destekler.

**(a)** Bir atlama listesinde *x*' den *y*' ye yoklama araması yapmak için gerekli algoritmayı verin. Algoritmanızın koşma süresi yüksek olasılıkla $O(\lg(2 + |rank(x) - rank(y)|))$ olmalıdır. Burada *x*' in rütbesi yani *rank (x*) dinamik setin sıralanmış düzeninde x elemanını o andaki rütbesini gösterir. ''Yüksek olasılıkla'' derken $m = 2 + |rank(x) - rank(y)|$ ' ye göre yüksek olasılığı kastederiz. Yani algoritmanızın koşma süresi, her $\alpha \geq 1$ için $1 - 1/m^{\alpha}$ …. olasılıkla $O(\lg m)$ olmalıdır.

Burada x elemanını depolayan atlama listesinin en altındaki listedeki yuvanın, yoklama- araması işlemine verildiğini düşünün.

**Çözüm:**
Bu problem çerçevesinde, atlama listesindeki her *x* yuvasının aşağıdaki alanları olduğunu düşünün:

| | |
|---|---|
| key*[x]* | *x* yuvasıyla ilintili anahtar |
| next*[x]* | linkli listede *x*' i içeren bir sonraki eleman |
| level*[x]* | *x*' i içeren linkli listede, linkli listenin düzeyi |
| up*[x]* | level*[x]*+1 düzeyindeki linkli listede *x* ile aynı anahtara sahip olan eleman |
| down*[x]* | level*[x]*-1 düzeyindeki linkli listede *x* ile aynı anahtara sahip olan eleman |

*Key[x] ≤k* durumu için kodu sunarız. *k≤ key[x]* durumu bunun simetriğidir. Aynı zamanda *level[x]=0* olduğunu varsayıyoruz. Yani arama, atlama listesinin en düşük düzeyinde başlıyor. (eğer arama *lg n* düzeyinde başlarsa yoklama aramasının koşma süresi *O(lg n)* olabilir.)

Algoritma iki evrede ilerler. İlk evrede düzeylerde mümkün olduğunca hızlı yukarı çıkarız, sonunda ileriye hareketin mümkün olmadığı düzeye kadar.

FINGER-SEARCH (x, *k)* ▷ Search for ***k*** in skip list containing node *x* .

(yoklama araması) (**x** düğümünü içeren atlama listesinde **k**' yı ara)

```
1 while key [next [x]] < k
2       do while up [x] ≠ NIL
3             do x← up[x]
4          x ←next [x]
5  while leve [x] ≥ 0
6          do while key [next [x]] ≤ k
7               do x ←next [x]
8           if level x > 0
9             then x ← down [x]
10 return x
```

( Aramanın yukarıya doğru gereğinden daha fazla gittiğine dikkat edin. Daha iyi bir çözüm her düzeydeki bir sonraki noktayı incelemek ve sadece bir sonraki anahtar, hedef anahtardan daha küçük ise yukarıya gitmeyi sürdürmektir. Öte yandan kanıtladığımız gibi, yukarıdaki basit algoritma da başarılı olur.)

Önce, yoklama araması sırasında, erişebileceğimiz en üst düzeyi belirleriz. Kanıtlaması, bir atlama listesinin en çok O(log *n*) düzeyi olduğundakinin aynısıdır.

**Önkuram 1** *Uygulama sırasında, yüksek olasılıkla, m=2+rank(key[x])-rank(k),* FINGER-SEARCH*, level[x] ≤ O(lg m).*

Kanıtlama: Atlama listesinin (kapalı) aralığı *M= [key[x], k]* ' de en çok *m-1* eleman vardır. Bu *m-1* elemanın olasılığını *c lg(m)* yüksekliğinden daha büyük olduğunu hesaplarız.

Pr [ M'deki bir eleman c lg(m+1) düzeyden fazlasında yer alır.]

$$\begin{aligned} &\leq (m-1)\cdot\frac{1}{2^{c\lg(m)}} \\ &\leq \frac{m}{(m)^c} \\ &\leq \frac{1}{(m)^{c-1}} \end{aligned}$$

*M*' deki hiçbir eleman *c lg(m)* düzeyinin ötesine terfi ettirilmediği için, *key[x]*' in her zaman *M* aralığında yer alacağı sonucu çıkar. Bu kanıtlamada yüksek olasılık çözümlemesinin *n*' ye göre değil, *m*' ye göre yapıldığını not edin. Şimdi algoritmanın 2 evresinin maliyetini ele alacağız: Önce yukarıya gitme maliyetini ve sonra aşağı gitme maliyetini. Sınıfta kanıtlanan ön kuramı hatırlayın:

**Önkuram 2** *Yazı-tura atışlarında, c lg m sayıda 'yazı' görme olasılığı yüksek olasılıkla O(lg m)' dir.*

Bu önkuramı kullanarak düzey yükselmelerini içeren birinci evrenin, yükselecek sadece *c lg m* kadar düzey olduğundan ve her 'yazı' bir düzey yükselmesi sonucunu verdiğinden, *O(lg m)* adımda tamamlandığını görebiliriz. Bu önkuram, aynı zamanda ikinci evreninde *O(lg m)* adımda tamamlandığını gösterir: Çünkü ARAMA çözümlemesinde, biz maliyeti tersten hesaplarız, *k*' yi saklayan yuvadan birinci evrede erişilen düzeye kadar yükseliriz; yukarı gidecek sadece *c lg m* düzeyi olduğundan ve her yazı gelişi bir düzey yukarı taşıdığından, ikinci evrenin maliyeti de *O(lg m)'* dir. Böylece toplam koşma süresi istendiği gibi *O(lg(m*+1)) olur.

B-ağaçlarından, hızlı yoklama aramalarını desteklemek için iki fikir gereklidir: $\mathrm{B}^+$ –ağaçları ve düzey linklemesi. Problem boyunca *B=O(1)* kabul edin.

Bir $\mathrm{B}^+$-ağacı öyle bir B-ağacıdır ki tüm anahtarlar yapraklarda depolanmıştır ve dahili düğümlerde bu anahtarların kopyaları depolanır. Daha kesin bir anlatımla bir $c_1, c_2 \ldots c_{k+1}$ gibi *k+1* ardılı olan bir p dahili düğümünde k sayıda anahtar depolanır. $c_1$in alt ağacındaki maksimum anahtar, $c_2$ nin alt ağacındaki maksimum anahtar, $c_k$ nın alt ağacındaki maksimum anahtar.

**(b)** Bir $\mathrm{B}^+$ - ağacında, O(lg n) zamanında, verilen bir *x* anahtarını saklayan yaprağı bulmak için, B-ağacı ARAMA algoritmasının nasıl değiştirmeniz gerektiğini tarif edin.

**Çözüm:**
Yapılması gereken tek değişiklik, dahili bir düğümde anahtar bulunursa çıkışa gitmek yerine, aramayı her zaman yapraklara kadar sürdürmektir.

```
B+-TREE-SEARCH(x, k)
1  i ← 1
2  while i ≤ n[x] and k > key_i[x]
3        do  i ← i + 1
4  if leaf[x]
5     then if i ≤ n[x] and k = key_i[x]
6               then return (c_i[x], k)
7               else  return NIL
8     else  return B+-TREE-SEARCH(x, k)
```

**(c)** $B^+$ ağaçlarında koşma süresinin *O(lg n)* olması için, B-ağaçlarında ARAYA YERLEŞTİRME ve SİLME algoritmalarının nasıl değiştirilmeleri gerektiğini açıklayın.

**Çözüm:** Normalde bir araya yerleştirme sırasında B-TREE- SPLIT-CHILD ( B-AĞACI-YARMA-ARDIL) bir düğümü 3 parçaya böler:
Ortancadan daha küçük elemanlar, ortanca ve ortancadan daha büyük elemanlar. Bu ortancayı atanın içine yerleştirir ve diğerlerini yeni düğümlere koyar.
$B^+$ ağaçlarında yarma işlemi dahili düğümlerde değişmez, ama yapraklarda değişir. Ortancanın iki yaprak düğümünün birinde muhafaza edilmesi gereklidir –solda olanında – ve aynı zamanda bir ataya eklenmesi gereklidir. Aşağıdaki sözde kod yeni yarma yordamını uygular.

```
B+-TREE-SPLIT-CHILD(x, i, y)
 1  if leaf[y]
 2     then z ← ALLOCATE-NODE()
 3          leaf[z] ← FALSE
 4          n[z] ← B − 1
 5          for j ← 1 to B − 1
 6              do key_j[z] ← key_{j+B}[y]
 7                 c_j[z] ← c_{j+B}[y]
 8          c_t[z] ← c_{2B}[y]
 9          n[y] ← B                          ▷ Unlike a B-tree, the old node keeps B keys.
10          for j ← n[x] + 1 downto i + 1
11              do c_{j+1}[x] ← c_j[x]
12                 key_j[x] ← key_{j−1}[x]
13          c_{i+1}[x] ← z
14          key_i[x] ← key_B[y]
15          n[x] ← n[x] + 1
16     else return B-TREE-SPLIT-CHILD(x, i, y)
```

(B-ağacının aksine, eski düğüm B sayıda anahtar içerir)

Geri kalan araya yerleştirme yordamları, yeni yarma yordamını kullanarak değiştirilmelidir.

Bir $B^+$- ağacı silmesinin iki kısmı vardır: Anahtarı yapraktan silmek, ki bu ağacı düzeltmeyi gerektirebilir ve anahtarın bir kopyasının artık dahili düğümde görünmemesini sağlamak. Her kısım *O(lg n)* süre alır.

Silme kökten başlar ve o anahtarı saklayan yaprağa kadar inilir. Eğer dahili bir düğümde silinecek olan anahtar keşfedilirse bu görmezden gelinir. (Bu düğümleri hatırlayın; silmenin ikinci kısmında bu düğümlere yeniden döneceğiz). B-ağacında olduğu gibi aşağı iniş yolundaki her düğümün en az B sayıda anahtar içerdiğinden emin olmak istiyoruz. Bir B-ağacı silmesinin üç durumu olduğunu hatırlayın. Özellikle üçüncü durum aşağı iniş sırasında ağaçta bir silme yapılırken bir ardılda en az B sayıda anahtar vardır. Yol boyunca her düğümde en az B sayıda anahtar olmasını sağlamak için üçüncü durumu uygulayın. Buradaki tek değişiklik yaprakları

birleştirirken aşağıda yeni birleştirilen düğüme bir anahtarı göndermenin gereksiz olmasıdır; anahtar atadan sadece çıkarılır. Yaprakta anahtar bulunduğunda onu yapraktan çıkarırsınız.

Silmenin ikinci kısmında, silinen anahtarın dahili bir düğümde görülmediğinden emin olmak isteriz. Önce silinen anahtarın öncülünü arayın. Silinen anahtarın birinci kısımda bulduğumuz yuvalarını hatırlayın. Yol boyunca silinen anahtarın göründüğü her yerde anahtarı öncülüyle değiştirin. (silinen anahtarın kopyaları silme işlemi sırasında sadece kökten yaprağa gelinen yol boyunca görülebilir). Bir düzey linklemeli $B^+$ **- ağacı ,** öyle bir $B^+$ **- ağacı** dır ki, aynı derinlikteki düğümler arasında hemen sonundaki düğüme bir ek işaretleyicisi vardır ve aynı zamanda aynı derinlikteki düğümler arasında hemen sağındaki düğüme de bir ek işaretliyicisi vardır.

**(d)** (c) bölümündeki $B^+$ **- ağacı** ARAYA YERLEŞTİRME ve SİLME algoritmalarının işlem başına *O (log n)* süresinde düzey linklerini koruyabilmeleri için nasıl değiştirilebileceğini açıklayın.

**Çözüm:** Şimdi her *x* düğümünün, sağdaki kardeş düğümüne *link[x]* işaretleyicisi vardır. Yarma yordamı aşağıdaki iki satırı içerecek şekilde değiştirilir.

1 *link[z]* ← *link[y]*
2 *link[y]* ← *z*

Bu yarılan *y* düğümünün linkini yeni *z* düğümüne kopyalar ve *y*' nin linkini *z*' yi işaretleyecek şekilde günceller. Bunun dışında araya yerleştirme işlemi değişmez.

Bir silme işleminde iki düğüm birleştirildiğinde ilgili link işaretleyicileri güncellenmelidir: soldaki (silinen) düğümden olan link kalan birleşmiş düğümün link noktasına kopyalanmalıdır. Bunun dışında silme işlemi değişmez.

**(e)** Bir düzey-linkli $B^+$ ağacında *x*' den *y*' ye yoklama araması yapacak bir algoritma verin. Algoritmanızın koşma süresi *O(lg(2+|rank(x) – rank(y)| ))* olmalıdır.

**Çözüm:** Algoritma (a) kısmındaki çözümde olanın benzeridir. Arama, ağaç boyunca bir sonraki anahtar aranan anahtardan büyük oluncaya kadar yukarı doğru gider. Bu noktada arama, ağaçta, normalinde olduğu gibi aşağıya doğru devam eder ve gerektiğinde linkleri kullanır.

Daha önce olduğu gibi *(x,i)* ağaçtaki bir anahtarı olan işaretçiyi temsil ettiğinde, aşağıdaki kod, başlangıçta *k*≥ $key_i$ *[x]* olduğunu varsayar. Ters durum simetriktir. Yoklama aramasını verimli yapabilmek için her x düğümünün bir *p[x]* ata işaretleyicisi ve bir *pi[x]* ata dizini olduğunu varsayarız.

```
B+-TREE-FINGER-SEARCH(x, i, k)
 1  while k > key_i[x] and link[x] ≠ NIL and k > key_1[link[x]]
 2        do while i ≤ n[x] and k > key_i[x]
 3              do i ← i + 1
 4           if i > n[x] and k > key_1[link[x]]
 5             then x ← p[x]
 6                  i ← pi[x]
 7  ▷ Begin downwards search.
 8  while not leaf[x]
 9        do if k ≥ key_1[link[x]]
10              then x ← link[x]
11                   i ← 1
12              else x ← c_i[x]
13                   i ← 1
14           while i ≤ n[x] and k > key_i[x]
15              do i ← i + 1
16  if k = key_i[x]
17     then return (x, i)
18     else return NIL
```

Aramanın verimliliğini göstermenin yolu birinci evrenin *O(lg m)* düzeyden daha fazla yukarı gitmemesidir. Bu gerçeği kanıtlamak için beşinci satırda sadece *k*> $key_1$*[link[x]]*.. olduğunda *x*' i *p[x]'* e eşit yaptığımıza dikkat edin. Bu *k*' nin, $c_1$ *[link[x]]* 'de kökü olan alt-ağacın her yaprağından daha büyük olduğunu ima eder. Dahası, arama başladığında, $c_1$ *[link[x]]*'de kökü olan alt-ağacın her yaprağı, ilk anahtar olan $key_i$*[x]* 'dan daha büyük olmalıdır. (çünkü arama sadece anahtarların artış yönünde hareket eder). Bu nedenle, eğer *x, l* düzeyinden *l+1* düzeyine çıkarsa (yaprakların sıfır düzeyinde olduğunu farzediyoruz), bu durumda $m \geq 2^{l-2}$ olur ve bu da *l=O(lg m)*' yı ima eder. Yani arama hiçbir zaman *O(lg m)* sayıda düzeyden daha fazla yukarı gitmez.

Öte yandan ağaçtan aşağıya doğru arama yaparken *x* en fazla bir yanal link geçişi yapar. Bu nedenle toplam arama maliyetinin *O(lg m)* olduğu sonucuna varabiliriz.

Bu fikirler, atlama listeleriyle düzey linkli 2-3-4 ağaçları arasında bir ilişki olduğunu işaret eder. Aslında bir atlama listesi düzey linkli $B^+$ağacının rastgele versiyonudur.

**(f)** Bir belirlenimci atlama listesinin, nasıl uygulanabileceğini açıklayın. Yani veri yapınız, bir atlama listesindeki genel işaretleyici yapısının aynısı olmalıdır: Aynı anahtarı depolayan komşu listelerdeki yuvalar arasındaki işaretleyicilerin olduğu bir veya daha fazla linkli liste dizisi. Arama algoritması bir atlama listesininkinin aynısı olmalıdır. Bir anahtarın terfi edileceğini belirlemek için ARAYA YERLEŞTİRME işlemini değiştirerek rastgeleliğin kullanımından kaçınmanız gerekir. Problemin bu kısmında SİLMEYİ görmezden gelebilirsiniz.

**Çözüm:**

Sonuçta çıkan belirlenimci atlama listesi, daha önceki bölümlerde geliştirilen düzey linkli $B^+$ağacının aynısıdır. Burada iki veri yapısı arasındaki bağlantıyı görmek için, aynı veri yapısını biraz farklı biçimde tekrarlarız. Veri yapısı tipik bir atlama listesinde olduğu gibi bir takım linkli listelerden oluşur. Tüm listeler - ∞ ile başlar. ARAMA algoritması, düzenli bir atlama listesi veri yapısındakinin tamamen aynısıdır. Linkli listedeki her düğüm, listede kendini takip eden terfi ettirilmemiş düğümlerin sayısı kadar genişletilir. Örneğin linkli listedeki sıradaki düğüm terfi ettirilirse bu sayı sıfırdır: Eğer bir sonraki düğüm terfi ettirilmez de, ondan sonraki düğüm yükseltilirse; bu sayı 1' dir. Burada hedef, bir terfi eden düğümün sayısının en az B-1 ve en çok 2B-1 olmasıdır. (B-ağacına oranla küçük farklı değerlerin olması, sayımın biraz farklı olan tanımından kaynaklanır).

Şimdi, ARAYA YERLEŞTİRMENİN nasıl devam edeceğini tanımlayalım ARAYA YERLEŞTİRME işlemi, atlama listesinin tepesinden - ∞ başlar ve normal atlama listesi aramasında olduğu gibi listede aşağı doğru iner. En alt düzeye ulaşıldığında linkli listeye yeni eleman eklenir ve sayılar güncellenir. (2B-1 sayımdan daha fazlasının güncellenmeyeceğini not edin). Eğer sayımların güncellenmesi bir önceki yükseltilmiş düğümün sayısını 2B-1' den daha büyük hale getirirse, B sayılı düğüm terfi ettirilir. Bu özyinelemeli olarak ağaçta yukarıya doğru sürdürülür ve gerekirse en üst düzeye bir yeni düzey daha eklenir. Bu yolla hiç rastgeleliği kullanmadan, atlama listesinin her zaman mükemmel bir atlama listesine yakın olmasını garanti ederiz.

**Problem 5-2. Bir Düzlemdeki Noktalarla Eğlence**

Saat sabahın 3'ü ve siz 6.046 derslerini videodan izlemeyi amaçlıyorsunuz ve problem seti 5 ile ilgili ipuçlarını arıyorsunuz. Garip bir nedenle belki de bilinciniz gelip-gittiğinden, odanızdaki beyaz duvarda, siyah noktalardan oluşan garip bir bulut fark ediyorsunuz. Bu nedenle dersi izlemek yerine, bilinçaltınız aşağıdaki problemi çözmeye çalışıyor. S= { $p_1$, $p_2$… $p_n$ }'nin *xy* düzleminde *n* noktadan oluşan bir küme olduğunu farz edin. Her $p_i$ noktasının koordinatları( $x_i$, $y_i$) ve **ağırlığı** da $m_i$.(noktanın boyutunu tanımlayan bir gerçek sayı). *F(p)= f(x,y,m)* bir rastgele fonksiyon olsun ve p noktasını *x,y* koordinatlarını m ağırlığında bir gerçek sayıya eşlemlesin ve bu *O(1)* süresinde hesaplanabilsin. Bir *S(T)* altkümesinde *F(T)* fonksiyonunu *T*' nin içindeki tüm noktalar için f( $p_i$)' lerin toplamı olarak tanımlayın. Yani

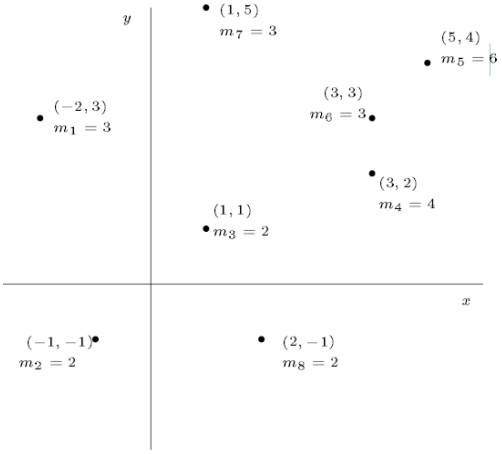
$$F(T) = \sum_{p \in T} f(p).$$

Örneğin, eğer f( $p_i$)=$m_i$ise *F(S)* tüm *n* noktalarının ağırlıklarının toplamıdır. Bu durum şekil 1' de gösterilmiştir.

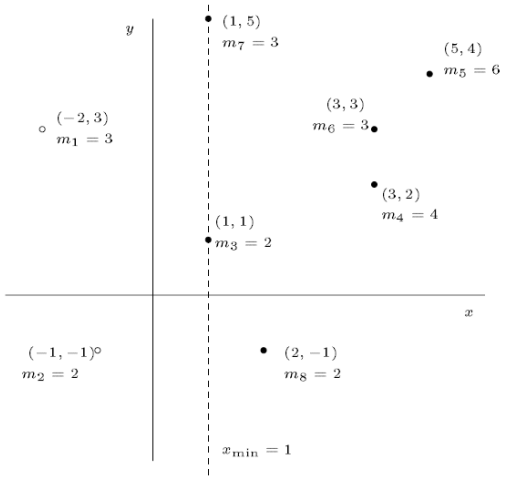
Hedefimiz, noktaların belli altkümeleri için *F* fonksiyonunu hesaplamaktır. Her alt kümeye bir *sorgulama* diyoruz ve her *T* sorgulaması için *F(T)*' yi hesaplamak istiyoruz. Çok sayıda sorgulama olabileceği için her sorgulamayı verimli yanıtlayabilecek bir veri yapısını tasarlamak istiyoruz.

Önce *x* koordinatını sınırlayan sorgulamaları ele alıyoruz. Özellikle *x* koordinatları en az $x_{min}$ olan noktaların kümesini ele alıyoruz. Resmi olarak *T(* $x_{min}$*)* aşağıdaki gibi noktaların kümesi olsun.

$$T(x_{\min}) = \{p_i \in S \mid x_i \geq x_{\min}\} .$$




Figür 1: 8 noktalı bu örnekte eğer *f(* $p_i$*)=* $m_i$ ise, *F(S)=25* olur.




Figür 2: $x_{min}$*= 1 ise, T(1)= {* $p_3$, $p_4$, $p_5$ $p_6$, $p_7$, $p_8$*} ve F(T(1))=20* olur.

Şu formdaki sorgulamalara yanıt bulmak istiyoruz: girişte herhangi bir $x_{min}$ değeri verildiğinde F(T) $x_{min}$' in değerini hesaplayın. Şekil 2, bu tür sorgulamaya bir örnektir. Bu durumda $x_{min}$=1 ve ilgi duyduğumuz noktalar *x* koordinatı en az *1* olanlardır.

**(a)** Bir dengeli ikili arama ağacının, böyle bir sorgulamayı *O(lg n)* sürede yapmak için nasıl değiştirilmesi gerektiğini gösterin. Daha açık bir deyişle F(T( $x_{min}$)) hesaplamasının ağaç boyunca aşağıya sadece bir yürüyüşte nasıl yapılacağını gösterin. Bu problem için güncelleme (araya yerleştirme ve silmeler) desteklerine ihtiyacınız yok.

**Çözüm:**

Problemin bu bölümü için ikili ağacı kullanarak iki değişik çözüm öneririz. İlk çözümde anahtarları ağacın sadece yapraklarında depolarız. İkinci çözümde ise anahtarları her düğümde depolarız. Çözümler yaklaşık birbirinin aynıdır.
Kökü *z* ' de olan alt-ağaçtaki tüm $p_i$ noktaları için f( $p_i$)' lerin toplamı *g(z)* olsun.

Eğer *z* bir yapraksa bu durumda *z*' de depolanmış noktanın f( $p_i$) değeri basitçe *g(z)*' dir. Her iki çözümde de ağaçtaki her *z* düğümünü *g(z)*' yi depolayacak şekilde genişletiriz. Sorgulamanın temelindeki fikir ikili ağaçta $x_{min}$ i aramak ve istenen toplamı yol boyunca hesaplamaktır.

İlk çözüm anahtarlar sadece yapraklarda depolandığında geçerli olur. Aramamız sırasında bir z düğümüne geldiğimizde *key[z]* < $x_{min}$ ise sağ alt-ağaçta özyineleme yapın. Diğer durumlarda *g(right[z])* değerini sol alt-ağaçta yapılan özyinelemeden çıkan sonuca ekleriz.

```
FWITHXMIN-V1(z, x_min)          ▷ Keys stored at leaves.
                                  (Anahtarlar yapraklarda depolanmış.)
1  if z = NIL
2     then return 0
3  if leaf[z]
4     then if key[z] ≥ x_min return g(z)
5     else  return 0
6  if key[z] < x_min
7     then return FWITHXMIN-V1(right[T], x_min)
8     else  return g(right[z]) + FWITHXMIN-V1(left[T], x_min)
```

Dikkat ederseniz son adımda z' nin sağ ardılını ziyaret etmekten, *g(right[z])=g(z)-g(left[z])* olması nedeniyle kaçınabiliriz.

Anahtarlar, ağacın dahili düğümlerinde depolandığında o an ziyaret ettiğimiz z düğümünün *f( $p_i$)* değerine bakmak, sorgulamada yapmamız gereken tek değişikliktir(simgelemi biraz bozarak buna f(z) deyin).

```
FWITHXMIN-V2(z, x_min)          ▷ Keys stored at every node.
                                  (anahtarlar her düğümde depolanmış olsun.)
1  if z = NIL
2     then return 0
3  if key[z] < x_min
4     then return FWITHXMIN-V2(right[T], x_min)
5     else  return f(z) + g(right[z]) + FWITHXMIN-V2(left[T], x_min)
```

Daha önce olduğu gibi son yineleme çağrısında *g(right[z])=g(z)-g(left[z])* olduğunu fark ederek z' nin sağ ardılını ziyaret etmek zorunda kalmayız.

Son olarak FWITHXMIN 'in doğruluğu ağacın yüksekliği üzerinde tümevarımla kanıtlanabilir.

**(b)** Tüm n noktalarının önceden bilindiği bir statik problemi düşünün. (a) şıkkındaki veri yapısını kurmak ne kadar zaman alır?

**Çözüm:** Veri yapısını kurmak *O(n lg n)* süresi alır çünkü aslında noktaları sıralıyoruz. Bunun yollarından biri noktaların *x* koordinatını baz alarak sıralama ve sonra bir ortanca elemanını kök olarak seçip dengeli bir ağaç yapılandırarak sol ve sağ alt-ağaçları özyinelemeli olarak kuramlamaktır. Bu yapı, noktaları sıralamak için *O(n lg n)* süresi ve ağacı kurmak için de *0(n)* süresi alır.

**(c)** Toplamda *n* sayıda nokta verilirse, veri yapısını kurmak ve *k* farklı sorgulamayı yanıtlamak ne kadar zaman alır? Öte yandan bir veri yapısı kullanmadan, saf algoritmayı *F(T( $x_{min}$))* her sorgulama için 0' dan başlayarak hesaplamak ve *k* farklı sorgulama yapmak ne kadar zaman alır? Veri yapısının asimptotik olarak daha verimli olması *k*' nin hangi değerleri için geçerlidir?

**Çözüm:** (a) ve (b) şıklarını kullanarak veri yapısını kurmak ve k sayıda sorguyu yanıtlamak *Θ(n lg n + k lg n )* süresini gerektirir. Saf algoritmayı, her sorgu için 0' dan başlayıp *F(T( $x_{min}$))* ' yi hesaplamak için kullanmak, tüm n noktalarını taramak ve bunları x koordinatları bazında en az $x_{min}$ kere toplamayı gerektirir. Bu ikinci algoritma *Θ (kn)* süresi gerektirir. Bu terimlere bakarak sezgiyle eğer *k*=$\omega$ *(lgn)* ise veri yapısını kurmanın *F*' yi 0' dan hesaplamaktan daha verimli olduğuna karar verebiliriz.

Bu bölüm için resmi bir kanıtlama peşinde değiliz ama bu konuyu tartışmanın yolu *k* sayıda sorgusunun gerektirdiği zamanlar olsun. Bu durumda asimptotik olarak $T_1$ *(n,k)* ≤ $T_2$ *(n,k)* karşılayan yeterli koşulları vermek istiyoruz.

**Önkuram 3** *Eğer k*=$\omega$ *(nlgn).. ise tüm n* ≥ $n_0$ *değerleri için* $T_1$ *(n,k)*< $T_2$ *(n,k)' yı sağlayan bir* $n_0$ *sabiti vardır.*

*Kanıt:* $c_1$ ve $n_1$ *O(n lg n+ k lg n)* ile çalışan birinci algoritmanın sabitleri ve $c_2$ ve $n_2$ *Ω(kn)* ile çalışan ikinci algoritmanın sabitleri olsun. Bu durumda *k* ve *n* için aşağıdaki durumu karşılayan yeterli şartları sağlarız.

$$T_1(n,k) \le c_1(n \lg n + k \lg n) < c_2 kn < T_2(k,n)$$

Eğer $n \geq max\{ n_1, n_2, \}$ ise

$$k > \frac{c_1 n \lg n}{c_2 n - c_1 \lg n}.$$

olduğunu biliriz.

$n_3$ öyle bir sabit olsun ki tüm $n \geq n_3$ değerleri için, $c_1 \lg n \leq c_2 \lg n / 2$ olsun. Bu durumda tüm $n \geq max\{ n_1, n_2, n_3\}$ için

$$k > \left(\frac{2c_1}{c_2}\right) \lg n.$$

olduğunu biliriz.

Eğer $k = \omega(n)$ ise tanım gereği, c, $2 c_1 / c_2$ olarak seçildiğinde öyle bir $n_4$ değeri bulabiliriz ki tüm $n \geq n_4$ değerleri için $k$, yukarıdaki koşulu karşılayacak kadar büyük olur.

Bu nedenle tüm $n > n_0$ ' lar için eğer $n_0 = max\{ n_1, n_2, n_3, n_4\}$ ise, $T_1(n,k) < T_2(n,k)$ olur.

**(d)** Bu veri yapısını bir kırmızı-siyah ağaç kullanarak dinamik hale getirebiliriz. (a) şıkkındaki çözümünüz genişletildiğinde bunun bir kırmızı-siyah ağaç tarafından verimli şekilde destekleneceğini tartışın; yani noktalar *O(lg n)* süresinde araya yerleştirilebilir veya silinebilir.

**Çözüm:** Anahtarlar her düğümde depolandığında, bir kırmızı-siyah ağaçtaki *g(z)*' nin her düğümde genişletilerek korunması, dinamik düzen istatistiği ağacının, alt-ağaçlarının boyutlarının genişletilerek korunmasıyla tamamen aynıdır.

Ağaca bir *p* noktası eklediğimizde, ağaçtan aşağıya doğru yürürken uygun z düğümlerinde *g(z)*' yi *f(p)* kadar arttırırız. Silme işleminde ağaçtan bir *p* noktası çıkarıldığında, *p*' den köke doğru yukarı yürürüz ve *g(z)*' yi *f(p)* kadar azaltırız. Bir *p* düğümünü onun atası ya da ardılı olan *p'* ile değiştirip düğümü sildiğimiz durumlarda, *p '* den yukarıya doğru yürürüz ve *g(z)*' yi *f(p')-f(p)* kadar arttırırız, ve sonra da kökü *p* olan ağaçtan *p'* düğümünü özyinelemeli olarak sileriz.

Döndürmeler sırasında *g(z)*' nin korunması da benzerdir. Örneğin, *x*' in kök ve *y*' nin de sağ-ardıl olduğu bir ağaçta, sol döndürme yaptığımızda (sayfa 306 şekil 14.2' ye bakın) *g(x) ve g(y)'* yi, *g(y) ← g(x) ve g(x) ← g(left[x])+ g(right[x])+f(x)* olacak şekilde güncelleriz.

Birinci türde çözümde anahtarlar kırmızı-siyah ağacın yapraklarında depolandığında, *z*' nin sol alt-ağacının maksimum değerini de korumamız gerektiğine dikkat edin. Bu özelliğin de korunduğunu tartışmamız gerekir. Normal ağaç araya yerleştirmeleri, silmeleri ve döndürmeleri için benzer argümanlar geçerli olacaktır.

Bundan sonra, girişi $x_{min}$ gibi tek sayı olmayan, bir $X=[ x_{min}, x_{max}]$ $( x_{min} \leq x_{max})$ aralığını girişinde kabul eden sorgulamalar üzerinde düşüneceğiz. *T(X), x* koordinatları, aşağıdaki aralıkta olan noktalar kümesi olsun.

$$T(X) = \{p_i \mid x_i \in [x_{\min}, x_{\max}]\}$$

Bu tür sorgulamanın bir örneği için şekil 3' e bakalım.

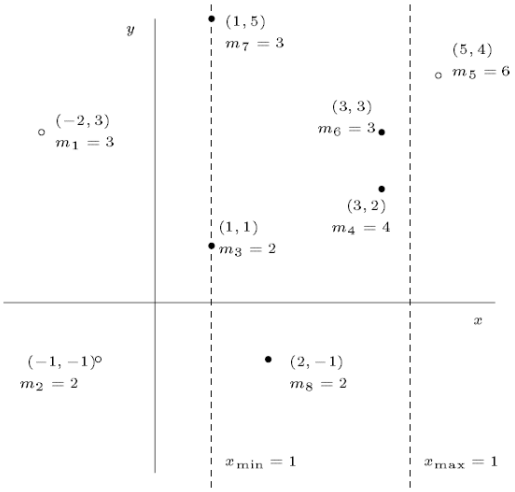
İddiamız *F(T(X))*' i hesaplamak için (d) şıkkındaki dinamik veri yapısını kullanabileceğimizdir.

**(e)** *F(T(X))'* i, *O(lg n)* sürede hesaplamak için (a) şıkkındaki algoritmanızın nasıl değiştirilmesi gerektiğini gösterin. *İpucu: x* koordinatı $x_{min}$ ile $x_{max}$ arasında olan ağaçtaki en sığ düğümü bulun.

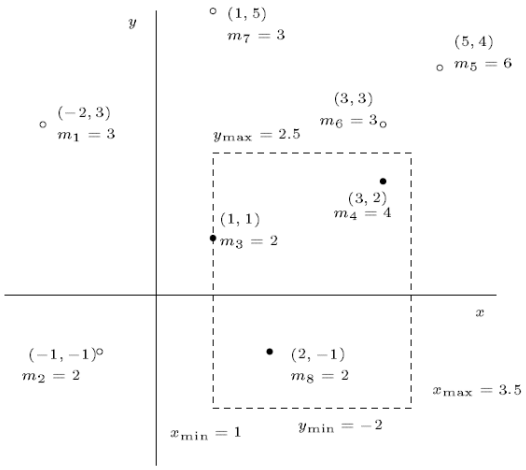
**Çözüm:**

Önce $x_{min}$ ve $x_{max}$ için ikiye bölünmüş bir düğüm bulun. *w* bölünmüş düğümü en derin düğüm olma özelliğini karşılar ve alt-ağacı, anahtarları *[* $x_{min}$, $x_{max}$*]* aralığında olan tüm düğümleri içerir.

Bir bölüntü düğümünü, $x_{min}$ ve $x_{max}$ ' ı yolları ayrılana kadar uygulanacak normal bir arama algoritmasıyla buluruz.




Figür3 ; *X=[1,3,5]* iken, *T(X)={* $p_3$ , $p_4$ , $p_6$ , $p_7$ , $p_8$ *}* ve *F(T(X))=14*




*F*igür 4; *X[1,3,5]* ve *Y=[-2,2.5]* için *T(X,Y)= {* $p_3$, $p_4$, $p_8$*}* ve *F(T(X,Y))=8*

```
FINDSPLITNODE-V1 (z, x_min, x_max)        ▷ Keys stored at leaves.
 1  if z = NIL                              (anahtarlar yapraklarda depolanmış.)
 2     then return NIL
 3  if leaf[z]
 4     if (x_min ≤ key[z] ≤ x_max) return z
 5     else  return NIL
 6  if (x_max ≤ key[z])
 7     then return FINDSPLITNODE-V1(left[T], x_min, x_max)
 8  if (key[z] < x_min)
 9     then return FINDSPLITNODE-V1(right[T], x_min, x_max)
10  return z
```

```
FINDSPLITNODE-V2 (z, x_min, x_max)        ▷ Keys stored at every node.
                                            (anahtarlar düğümlerde depolanmış.)
 1  if z = NIL
 2     then return NIL
 3  if (x_max < key[z])
 4     then return FINDSPLITNODE-V2(left[T], x_min, x_max)
 5  if (key[z] < x_min)
 6     then return FINDSPLITNODE-V2(right[T], x_min, x_max)
 7  return z                               ▷ If we get here, x_min ≤ key[z] ≤ x_max.
                                            (buraya gelindiğinde)
```

Eğer FINDSPLITNODE ,NIL dönüşü verirse, o zaman F(T(X))= 0 olur çünkü aralıkta hiç nokta yoktur. Diğer durumlarda F(T(X))' i hesaplamak için (a) şıkkındaki FWITHXMIN fonksiyonunu ve ilgili FWITHXMAX fonksiyonunu kullanabiliriz (bu fonksiyon F(T(I -∞, $x_{max}$I))) 'ın hesaplar.

```
FWITHXMAX-V1(z, x_max)        ▷ Keys stored at leaves.
1  if z = NIL                 (anahtarlar yapraklarda depolanmış)
2     then return 0
3  if leaf[z]
4     then if key[z] ≤ x_max return g(z)
5     else  return 0
6  if key[z] < x_max
7     else  return g(left[z])+ FWITHXMAX-V1(right[T], x_max)
8     then return FWITHXMAX-V1(left[T], x_max)
```

```
FWITHXMAX-V2(z, x_max)        ▷ Keys stored at every node.
                              (anahtarlar düğümlerde depolanmış.)
1  if z = NIL
2     then return 0
3  if key[z] > x_max
4     then return FWITHXMAX-V2(left[T], x_max)
5     else  return f(z) + g(left[z])+ FWITHXMAX-V2(right[T], x_max)
```

Eğer bir *w* bölünmüş düğümümüz varsa, $\omega$ ' nin sol alt-ağacındaki tüm düğümlerin anahtarlarını en çok *xmax* olduğunu ve sağ alt-ağacındaki tüm düğümlerin anahtarların da en az *xmin* olduğunu biliriz.

Eğer tüm anahtarlar yapraklarda depolanmışsa;

F(T(X))= FWITHXMIN-V1 left[ $\omega$],$x_{min}$) + FWITHXMAX-V1 (right[ $\omega$ ], $x_{max}$ )

Anahtarlar hem yapraklarda hem de dahili düğümlerde depolanmışsa;

F(T(X))= f($\omega$ )+FWITHXMIN-V2 (left[$\omega$], $x_{min}$) + FWITHXMAX-V2 (right[ $\omega$ ], $x_{max}$ )

```
FWITHX-V2(z, x_min, x_max)
1  if z = NIL
2     then return 0
3  w ← FINDSPLITNODE-V2(z, x_min, x_max)
4  if w = NIL
5     then return 0
6     else  return f(w) + FWITHXMIN-V2(left[z], x_min) + FWITHXMAX-V2(right[z], x_max)
```

Son olarak statik problemi 2 boyutta genelliyoruz. Farz edin ki bize, *X=[* $x_{min}$, $x_{max}$*]* ve *Y=[* $y_{min}$, $y_{max}$*]* gibi iki aralık verilmiş olsun. *T(X,Y)* bu dikdörtgendeki tüm noktaların kümesi olsun. Yani;

$$T(X,Y) = \{p_i \mid x_i \in X \text{ and } y_i \in Y\}$$

2 boyutlu bir sorgulamanın örneği için şekil 4' e bakın.

**(f)** *X* ve *Y* aralıklarında, *F(T(X,Y))* sorgusunu verimli olarak hesaplamayı destekleyen bir veri yapısını açıklayın. Bir sorgu $O(lg^2\ n)$ sürede koşmalıdır.
*İpucu:* Bir değer kümesi ağacını genişletin.

**Çözüm:** Birincil ağacı *x* koordinatına anahtarlanmış, 2-D (2 boyutlu) bir değer kümesi ağacı kullanırız. Bu *x*-ağacındaki her *z* düğümünün 1 boyutlu değer kümesi ağacının, *y* koordinatına anahtarlanmış bir işaretleyicisi vardır. Şık (a)' daki aynı *g(z)* ile y-ağaçlarında bu düğümlerin her birini genişletiriz. Bir sorgulama yapmak için, öncelikle birincil *x*-ağacının $[\ x_{min},\ x_{max}\ ]$ aralığında *x* koordinatları olan tüm düğüm/alt-ağaçlarını ararız. Sonra bu düğüm/alt-ağaçların her biri için *y*-ağacında sorgulama yaparız ve *y* koordinatı $[\ y_{min},\ y_{max}\ ]$ aralığında olan tüm noktalar için *F*' yi hesaplarız.

Eğer g(z)' nin her olagelmesini FWITHY( *ytree[z]*, $y_{min}$, $y_{max}$ ) ile değiştirirsek, şık(a) ve (e)' deki sözde kodun aynısını kullanabiliriz; burada FWITHY bir boyutlu değer kümesi ağacındaki sorgulamadır.

Sorgulamalar için algoritma, sabit zamanlı *g(z)* fonksiyonu çağrılarının *ytree[z]* sorgu çağrılarıyla değiştirilmesi koşuluyla, şık(a)' dakinin aynıdır. Bir *y*-ağacındaki her sorgu *O(lg n)* süre alacağından, sorgu için toplam koşma süresi $O(lg^2\ n)$ olur.

**(g)** Veri yapınızı kurmak ne kadar süre alır? Ne kadar yer tutar?

**Çözüm:** Bir *y*-ağacı veya *x*-ağacının her düğümüne sabit miktarda bilgi yüklediğimizde, kullanılan alan 2 boyutlu değer kümesi ağacıyla asimptotik olarak aynıdır; yani *O(n lg n)*' dir. Bir 2 boyutlu değer kümesi ağacında kullanılan alan *O(n lg n*)' dir çünkü her nokta *O(lg n) y*-ağacında görünür.

Bir 2 boyutlu değer kümesi ağacını kurmak için gereken basit bir algoritma $O(nlg^2\ n)$ süre alır. Önce şık (c)' deki gibi, noktaları *x* koordinatına dönük sıralayarak *x*-ağacını *O(n lg n)* sürede kurun. Sonra *x*-ağacındaki her *z* düğümü için ilgili *y*-ağacını kurun ve bu amaçla (c) şıkkındaki algoritmayı *y*-koordinatlarını sıralamada kullanın. Tüm y-ağaçlarını oluşturacak yineleme için gereken süre *T(n)=2T(n/2)+O(n lg n)* ya da $T(n)= O(n\ lg^2\ n)$ olur.

Daha akıllı bir çözümle 2 boyutlu değer kümesi ağacını *O(nlgn)*' de kurmak mümkündür. Dikkat ederseniz, 2 boyutlu değer kümesi ağacında, *x*-ağacındaki her *z* düğümü için, *z*' nin sol ve sağ alt-ağaçlarındaki *y*-ağaçları ayrışıktır. Bu gözlem aşağıdaki algoritmayı düşündürür.

Önce her noktayı *x* koordinatına göre sıralayın ve önceki gibi birincil değer kümesi ağacını kurun. Sonra noktları *y* koordinatına göre sıralayın ve *z* kök düğümü için, aynı sıralamayı kullanarak en büyük *y*-ağacını kurun. *Left[z]* ve *right[z]* için *y*-ağaçlarını elde etmek amacıyla, dizilimin *x-key[z]* (*x*-ağacında *z*' nin anahtarı) etrafındaki *x*-koordinatlarında dengeli bir bölüntüleme yapın. Bu bölüntüleme, bize *y*-koordinatında tekrar sıralama yapmaksızın, *z*' nin sol ve sağ alt-ağaçları için, y-ağaçlarını oluşturmamıza izin verir. Bu nedenle, koşma süresi için yineleme *T(n)=2T(n/2)+O(n)* ya da *O(n lg n)* olur.

Maalesef bu veri yapısını dinamik hale getirmede sorunlar vardır.

**(h)** Şık (d)' deki argümanınızın 2 boyutlu duruma genelleştirilip genelleştirilemeyeceğini açıklayın. Şık (f)' deki veri yapısına yeni bir noktanın araya yerleştirilmesi için gerekli en kötü durum süresi nedir?

**Çözüm:** Şık (d)' deki argümanı, 2 boyutlu değer kümesi ağaçlarına genelleştiremeyiz çünkü, *x*-ağacını kolaylıkla güncelleyemeyiz. Tek bir *x* veya *y*-ağacında normal bir ağaç araya yerleştirmesi yapmak *O(lg n)* zaman alır. Öte yandan *x*-ağacındaki bir düğümde, bir döndürme yapmak tüm *y*-ağacını yeni baştan kurmayı gerektirir. Örneğin, kitabınızın 278. sayfasındaki şekil 13.2' deki sola döndürme diyagramını düşünün. Döndürmeden sonra, *y* için kurulan *y*-ağacı, döndürmeden önce *x* için oluşan *y*-ağacıyla tamamen aynı düğümleri içerir. Öte yandan, döndürmeden sonra *x* kökü olan *y*-ağacını düzeltmek için *y*'nin orijinal *y* ağacında $\gamma$' deki tüm noktaları çıkarmamız ve $\propto$'daki tüm noktaları bulmamız gerekir.

En kötü durumda, eğer *x* kök ise $\propto$ ve $\gamma$ alt-ağaçlarının *O(n)* sayıda düğümü olacaktır. Böylece *x*-ağacında bir döndürme yapmak Ω*(n)* süresi gerektirir. Bir kırmızı-siyah ağaçta güncelleme yapmak, sadece sabit sayıda döndürme gerektirdiğinden, bir güncelleme için gereken zaman *O(n)*' dir.

**(i)** Başlangıçta *(n)* noktası olan bir veri yapısını kurduktan sonra *O(lg n)* sayıda güncelleme uygulayacağımızı varsayalım. Bu durumda hem sorguları hem de güncellemeleri verimli şekilde desteklemek için, veri yapısını nasıl değiştirebiliriz?

**Çözüm:**
Eğer *x*-ağaçları ve *y*-ağaçlarındaki dengenin korunmasıyla ilgilenmezsek, o zaman *x*--ağacına ve ilgili *y*-ağaçlarına, yeni noktaları yerleştirmek için, normal bir ağaç araya yerleştirme işlemini kullanabiliriz. Algoritmanın, güncelleme başına *O($lg^2$ n)* süreye ihtiyacı olur çünkü *O(lg n)* sayıda *y*-ağacına bir nokta eklemek zorundayız.

Ek bir not olarak, küçük sayıda güncelleme yapıyorsak, sorgulamaları *O($lg^2$ n)* sürede ve güncellemeleri de *O*(1) sürede yapmak mümkündür! Bu

yöntemde değer kümesi ağacını $O(lg^2\ n)$ boyutlu bir ek ara bellek boyutunda genişletiriz. Araya yerleştirme ve silme işlemleri tembeldir: sadece ara bellekte kuyruk oluştururlar. Bir sorgulama olduğunda orijinal 2 boyutlu değer kümesi ağacını arar ve eski yanıtı $O(lg^2\ n)$ sürede bulur. Sonra yanıtını ara bellekteki her noktayı tarayarak günceller. Böylece O($\lg^2$ n) sayıda güncellemeyi, sorguların asimptotik koşma süresini değiştirmeden destekleyebiliriz.

**Tamamiyle isteğe bağlı kısımlar**

Bu problemin geri kalan bölümünde, daha önceki bölümlerde tanımladığımız veri yapılarını kullanarak gerçek bir uygulamada, verimli hesaplama yapan bir *F* fonksiyonu örneği verilmekte. İlgili $f(p_i)$ fonksiyonunun türetilmesi şık (j) ve (l)' de ana hatlarıyla açıklanıyor.

Bu problemin geri kalan kısmı, tamamiyle isteğe bağlıdır. Lütfen bu bölümleri yanıtınıza eklemeyin.

Daha önceki gibi, bir düzlemde *n* sayıdaki noktaların kümesi $S\{p_1, p_2, \ldots\ldots, p_n\}$ olsun; bunların koordinatları $(x_i, y_i)$ ve ağırlığı $m_i$. Biz kümedeki noktaların eylemsizlik momentini en aza indirecek ekseni hesaplamak istiyoruz. Resmi olarak alttaki değeri en aza indiren düzlemdeki *L* çizgisini hesaplamak istiyoruz.

$$\sum_{i=1}^{n} m_i \Big(d(L, p_i)\Big)^2.$$

Burada $d(L, p_i)$, $p_i$noktasından, *L* çizgisine olan mesafedir. Eğer tüm *i* ' ler için $m_i$=1 ise, bu ekseni kümenin oryantasyonu yani yönelimi olarak düşünebiliriz.

**(j)** *xy* düzlemindeki bir çizginin bir parametreleme şekli de, onu bir $(\rho, \theta)$ çifti olarak tanımlamaktır; burada $\rho$, kaynaktan çizgiye olan uzaklık ve $\theta$' da çizginin *x* ekseniyle yaptığı açıdır. Bir *(x,y)* noktası ile bir *L* çizgisinin arasındaki uzaklık, $(\rho, \theta)$ cinsinden

$$|x \sin\theta - y \cos\theta + \rho|$$

parametrelendirilebilir.

Biz S noktalar kümesinin yönelimini $L = (\rho, \theta)$ olarak tanımladık ve bu aşağıdaki fonksiyonu en küçük değerine indirgiyor.

$$f(\rho, \theta) = \sum_{i=1}^{n} m_i \left(x_i \sin\theta - y_i \cos\theta + \rho\right)^2$$

$\frac{\partial f}{\partial \rho} = 0$ ayarlamasının;

$$M_{x1} \sin\theta - M_{y1} \cos\theta + M_0 \rho = 0,$$

kısıtını getireceğini gösterin, burada;

$$M_0 = \sum_{i=1}^{n} m_i, \qquad M_{x1} = \sum_{i=1}^{n} m_i x_i, \qquad M_{y1} = \sum_{i=1}^{n} m_i y_i$$

**Çözüm:**

$$\begin{aligned} f(\rho,\theta) &= \sum_{i=1}^{n} m_i (x_i \sin\theta - y_i \cos\theta + \rho)^2 \\ \frac{\partial f}{\partial \rho} &= \sum_{i=1}^{n} (2m_i (x_i \sin\theta - y_i \cos\theta + \rho)) \\ &= 2\sin\theta \left(\sum_{i=1}^{n} m_i x_i\right) - 2\cos\theta \left(\sum_{i=1}^{n} m_i y_i\right) + 2\rho \left(\sum_{i=1}^{n} m_i\right) \end{aligned}$$

*∂ f / ∂* ρ yu *0'* a eşitleyerek ρ* *ve* Θ* yu bulun. Bunun:

$$M_{x1} \sin\theta^* - M_{y1} \cos\theta + M_0 \rho^* = 0$$

**(k)** *∂ f / ∂ ρ* =0 ayarlamasının ve (j) şıkkından gelen kısıtın kullanılmasının aşağıdaki denklemi vereceğini gösterin.

$$\tan 2\theta = \frac{2\left(M_0 M_{xy} - M_{x1} M_{y1}\right)}{M_0\left(M_{x2} - M_{y2}\right) + M_{y1}^2 - M_{x1}^2}$$

burada

$$M_{xy} = \sum_{i=1}^{n} m_i x_i y_i, \qquad M_{x2} = \sum_{i=1}^{n} m_i x_i^2, \qquad M_{y2} = \sum_{i=1}^{n} m_i y_i^2$$

**Çözüm:** Yukardaki çözümün doğru olduğu varsayıldığında bu sorunun da çözümü mümkündür. Bu çözümün matematiğini iki kez kontrol etmek isteyebilirsiniz.

$\partial f/\partial\theta$ ' yı hesaplarız ve bunu 0' a eşitleriz.

$$\begin{aligned}
\frac{\partial f}{\partial\theta} &= \sum_{i=1}^{n} \left(2m_i(x_i\sin\theta - y_i\cos\theta + \rho)(x_i\cos\theta + y_i\sin\theta)\right) \\
&= \sum_{i=1}^{n} \left[2m_i\left((x_i^2+y_i^2)\sin\theta\cos\theta - x_iy_i(\cos^2\theta - \sin^2\theta) + \rho(x_i\cos\theta + y_i\sin\theta)\right)\right] \\
&= \sum_{i=1}^{n} \left[m_i\left((x_i^2+y_i^2)\sin 2\theta - x_iy_i\cos 2\theta + 2\rho(x_i\cos\theta + y_i\sin\theta)\right)\right] \\
&= (M_{x2}+M_{y2})\sin 2\theta - 2M_{xy}\cos 2\theta + 2\rho\left(M_{x1}\cos\theta + M_{y1}\sin\theta\right).
\end{aligned}$$

Her iki kısmi türevin de 0 olduğu durumda (ρ* , Θ*) uç noktalarını bulmak için, (ρ*)' nü ortadan kaldırmak amacıyla bir önceki kısımdaki ifadeyi kullanırız.

$$\rho^* = -\frac{M_{x1}\sin\theta^* - M_{y1}\cos\theta^*}{M_0}$$

$$\begin{aligned}
(M_{x2}+M_{y2})\sin 2\theta^* - 2M_{xy}\cos 2\theta^* - \\
\frac{2\left(M_{x1}\sin\theta^* - M_{y1}\cos\theta^*\right)\left(M_{x1}\cos\theta^* + M_{y1}\sin\theta^*\right)}{M_0} &= 0 \\
(M_{x2}+M_{y2})\sin 2\theta^* - 2M_{xy}\cos 2\theta^* - \\
\frac{2(M_{x1}^2 - M_{y1}^2)\sin\theta^*\cos\theta^* - 2M_{x1}M_{y1}(\cos^2\theta^* - \sin^2\theta^*)}{M_0} &= 0
\end{aligned}$$

$$\begin{aligned}
\left(M_{x2}+M_{y2} - \frac{M_{x1}^2 - M_{y1}^2}{M_0}\right)\sin 2\theta^* &= 2\left(M_{xy} - \frac{M_{x1}M_{y1}}{M_0}\right)\cos 2\theta^* \\
\tan 2\theta^* &= \frac{2\left(M_0M_{xy} - M_{x1}M_{y1}\right)}{M_0\left(M_{x2}+M_{y2}\right) - \left(M_{x1}^2 - M_{y1}^2\right)}
\end{aligned}$$

**(I)** Yönelim probleminin biraz önce çözdüğümüz problemin özel bir durumu haline getiren *F($p_i$)* fonksiyonunu verin. *İpucu: F($p_i$)* fonksiyonu, bir vektör-değerli fonksiyondur.

**Çözüm:**

$$f(p_i) = m_i(1, x_i, y_i, x_i y_i, x_i^2, y_i^2)^T$$

denklemini kullanın.