İSTANBUL MEDENİYET ÜNİVERSİTESİ

BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

ALGORİTMA ANALİZİ VE TASARIMI ÖDEVİ

ÖDEVİN KONUSU: DÜŞÜK MALİYETLİ YOL HESAPLAMA

İSİM: FATİH DURSUN

SOYİSİM: ÜZER

DERSİ VEREN ÖĞRETİM ÜYESİ: AYŞE BETÜL OKTAY

Problem Tanımı:

Bir araç bir menzile doğru gidecektir ve bir müddet sonra benzin alması gerekecektir. Yol üzerinde uzaklıkları artan sırada m1, m2, ..,mn olan n adet benzin istasyonu bulunmaktadır. Her benzin istasyonunda benzin fiyatları değişkenlik göstermektedir ve her istasyondaki toplam depo benzin ücreti
p1,p2, …,pn dir. Her seferinde benzin tankı boşmuş gibi düşünülerek p'ler hesaplanmıştır, deponun içinde olan benzini hesaplamayınız. Benzin kaliteleri aynı olup, bir depo benzin ile maksimum f km yol gidebilmektedir. Araç benzin aldıktan sonra en az k kilometre durmadan gitmek zorundadır ve bir sonraki benzini en az k km sonraki istasyondan alabilir.

Aracın en az benzin fiyatıyla yolculuğu tamamlayabilmesi için hangi istasyonlarda durması gerektiğini bulmanız istenmektedir.

Problem Çözümleri:

Yukarıda tanımı yapılmış olan problemin çözümü için 4 farklı çözüm tekniği kullanılmıştır.
Bunlar: Greedy(Açgözlü) yaklaşım,Dinamik Programlama(Dynamic Programming),Kaba Kuvvet(Brute Force) ve Decrease And Conquer(Azalt ve Fethet) yöntemleridir.

1.GREEDY(AÇGÖZLÜ) YAKLAŞIM:

Genel olarak Açgözlü algoritma, karşısına çıkan seçenekler arasından seçim yapmak gerektiğinde problemin çözümü için o adımda ne gerekiyorsa onu yapar. Diğer adımlara bağlı olan sonuçlarla ilgilenmez.
Tanımlı problemin çözümü için Greedy yöntem olan Dijkstra's Shortest Path algoritmasına oldukça benzeyen Uniform Cost Search(UCS) algoritmasından esinlenilmiştir.
Algoritmanın genel olarak çalışma prensibi şu şekildedir:
Başlangıçta sadece içerisinde root node bulunan bir PriorityQueue oluşturulur.
(Algoritmada kullanılan Priority Queue'nın öncelik seçimi en düşük fiyatlı(ağırlıklı) vertex'e göredir. Yani kuyruğa eklenme sırası önemsiz olmaksızın en düşük fiyatlı vertex PriorityQueue'nun başında yer alır.)
PriorityQueue'ya kuyruğun başındaki elemanın komşuları(kuyruğun başındaki elemana en az k ve en fazla f km uzakta olan istasyonlar) eklenir ve kuyruğun başındaki eleman PriorityQueue'dan çıkartılır.Daha sonra eklenen elemanlardan yine fiyat olarak en düşüğü kuyruktan çıkarılırken, kuyruktan çıkarılan elemanın komşuları Priority Queue'ya dahil olur fakat bu elemanlar PriorityQueue'ya dahil edilirken kendi fiyatlarına parentlarının fiyatları da eklenir.
Bu adımlar hedef istasyon(Bu problem için bitiş noktası) bulunana kadar recursive şekilde devam eder.
Bitiş noktasına ulaşıldıktan sonra bu noktanın fiyatı return eder çünkü bu fiyat daha önceden durulan istasyonlarının fiyatları ve bitiş noktasının fiyatının toplamı olduğu için bize toplam maliyeti vermektedir.
Algoritma'da kullanılan bir başka veri yapısı ise ArrayList. Bu ArrayList gidilen durakların parentlarını tutarak aracın hangi istasyonlara uğradığını görmemize olanak sağlıyor.

$queueBaslatma(G, v)$
$//INPUTS:$
$//G$: PriorityQueue ve parent dizisi içeren Graph
$//v : Vertex$
$//end$:Hedef Vertex
for i to $mList.size$ **do**
  $G.parent[i] \leftarrow -1$
end for
$ENQUEUE(G.que, v)$
$UCS(G, end)$

$UCS(G, end)$
$//INPUTS:$
$//G$:PriorityQueue ve parent dizisi içeren Graph
$//end$:Hedef Vertex
if $G.que \neq \emptyset$ **then**
  $b \leftarrow peek(G.que)$
  $u \leftarrow DEQUEUE(G.que)$
  for $each$ $v \in G.adj(u)$ **do**
    $G.parent[v.label] \leftarrow b.label$
    $kontrol \leftarrow v$
    $v.fiyat \leftarrow v.fiyat + b.fiyat$
      if $a = end$ **then**
        $return$
      end if
      $ENQUEUE(G.que, v)$
    end for
    $UCS(G, end)$
  end if

Figure 1.1
Problemin çözümü için uygulanmış olan
Greedy Algoritma için pseudocode

Algoritmanın Asimptotik Analizi

QueueBaslatma fonksiyonuna bakacak olursak; Fonksiyonun içerisinde n defa tekrarlanan bir döngü bulunmaktadır. Bu yüzden bu fonksiyonun çalışma zamanı **O(n)** dir.
UCS fonksiyonuna bakacak olursak; Fonksiyonun içerisinde tek bir döngü bulunmakta. Bu döngü; Eğer bir istasyon kendisinden sonraki istasyonların bitiş noktası hariç hepsine komşuysa, yani o istasyondan bitiş noktası haricinde diğer tüm istasyonlara gidilebiliyorsa döngü n defa tekrarlanır.Fonksiyon recursive bir fonksiyon olduğu için şart sağlanana kadar
$T(n)=T(n)+n$
$T(n)=T(n)+n$
.

.
.

T(n)=n

Fonksiyonun en kötü durumda n defa kendisini çağıracağını varsaydığımızda en kötü durum **(worst case) O(n+n^2)** olur.

UCS fonksiyonuna bu sefer Best Case için bakacak olursak; Eğer aranan istasyon başlangıç istasyonunun komşularından birindeyse ve bu komşu diziside bir öğeden oluşuyorsa bu döngü O(1) sürede sona erer. Aradığımız eleman bulunduğu ve return ettiğimiz için fonksiyonun alt kısmına girmeyeceği yani recursive olarak kendini çağıracağı kısma gelmeyeceği için fonksiyon O(1) zamanda işini bitirmiş olur. T(n)=1 durumu meydana gelmiş olur yani fonksiyonun Best Case'i **O(1)**olmuş olur.

Algoritmanın input büyüklüklerine göre çalışması

Greedy algoritmanın çalışma hızı aracın benzin aldığı durakların sayısına göre sayısal olarak artış göstermektedir.

```
f:7.0
k:2.0
input size:7
istasyonların baslangic noktasina uzakliklari
0.0     5.0     7.0     9.0     11.0    13.0    15.0
İstasyon Fiyatları(baslangic noktasi ve bitis noktasinin fiyatları 0'dir.)
0.0     6.0     8.0     7.0     3.0     4.0     0.0
Bulundu9.0
Calisma Suresi 0.002 saniyedir
Aracin durma noktalari(bitis noktasindan baslangic noktasina)
4       2       1       0
```

```
f:120.0
k:30.0
input size:9
istasyonların baslangic noktasina uzakliklari
0.0     38.0    74.0    87.0    106.0   155.0   197.0   238.0   268.0
İstasyon Fiyatları(baslangic noktasi ve bitis noktasinin fiyatları 0'dir.)
0.0     14.0    11.0    15.0    7.0     20.0    10.0    8.0     0.0
Bulundu17.0
Calisma Suresi 0.002 saniyedir
Aracin durma noktalari(bitis noktasindan baslangic noktasina)
6       3       1       0
```

```
f:25.0
k:15.0
input size:10
istasyonların baslangic noktasina uzakliklari
0.0     10.4    18.3    28.9    39.41   55.1    57.25   64.5    68.8    70.5
İstasyon Fiyatları(baslangic noktasi ve bitis noktasinin fiyatları 0'dir.)
0.0     3.7     4.1     3.6     4.8     3.17    3.2     3.21    3.33    0.0
Bulundu12.069999999999999
Calisma Suresi 0.002 saniyedir
Aracin durma noktalari(bitis noktasindan baslangic noktasina)
5       4       2       0
```

```
f:15.0
k:5.0
input size:21
istasyonların baslangic noktasina uzakliklari
0.0     8.0     16.0    24.14   30.0    35.0    39.0    41.0    43.0    49.0    50.0    51.0    58.0    65.0    7
İstasyon Fiyatları(baslangic noktasi ve bitis noktasinin fiyatları 0'dir.)
0.0     1.2     1.3     2.15    1.9     2.8     1.54    2.0     1.13    2.0     1.88    1.74    1.56    1.44    1
Bulundu10.94
Calisma Suresi 0.006 saniyedir
Aracin durma noktalari(bitis noktasindan baslangic noktasina)
18      16      15      13      10      8       5       4       3       2       1       0
```

2.Kaba Kuvvet(Brute Force)

Kaba Kuvvet algoritması diğer algoritmalara göre maliyeti oldukça yüksek olan bir algoritmadır.Hedefi, maliyet ve alana bakmaksızın sonuca ulaşmaktır.Küçük boyutlu inputlarda diğer algoritmalarla benzer hızda çalışsa da, input büyüklüğü arttıkça algoritmanın hızı yavaşlamakta ve belli bir input büyüklüğünden sonra yüzbinlerce saat sürebilmektedir.
Tanımı verilmiş olan problemin çözümü için kullandığımız 2.yöntem olan Brute Force, problemin çözümü için çalışma prensibi şu şekildedir;
İnput olarak aldığı mList ya da pList'in size ı kadar olan sayıların alt kümelerini buluyor. Örneğin, mList'in size'ı 3 olsun bu durumda alt kümeler={{},{1},{2},{3},{1,2},{1,3},{2,3}}'dır.
Alt kümeleri bulduktan sonra, bu alt kümelerin boş küme ve tek bir sayı içermeyen kısımlarını yeni bir listeye ekliyor.
Bir sonraki aşamada ise, listenin içerisindeki alt kümelerin içerisinde başlangıç noktası ve bitiş noktası var ise ve yan yana olan iki eleman arasındaki fark k'ya eşit veya k den büyükse ve f ye küçük veya eşitse bunları başka bir yeni diziye aktarıyor. Birinci koşul sayesinde, hedefimiz başlangıç noktasından bitiş noktasına en ucuz şekilde gitmek olduğu için başlangıçtan başlamayan ve bitiş noktasında sonlanmayan bir yolu seçeneklerimiz arasından çıkarmış oluyor, ikinci koşul sayesinde ise problemin tanımına uygun kümeleri seçmiş oluyor.
Tüm uygun yolları bulduktan sonra yolcunun en az maliyetle yolculuğunu tamamlayama bilmesi için bu yolların toplam maliyetlerini hesaplayıp bu yollar arasındaki en az maliyetli yolu buluyor.
En az maliyetli yolu bulduktan sonra, en az maliyete sahip olan yolu ve en az maliyetli fiyatı kullanıcıya bildiriyor ve sonlanıyor.

```
bruteForce(mList, pList, k, f)
//INPUT : mList[1, , ....n]istasyonların baslangıç noktasına olan uzaklıklarını
tutan n elemanlı dizi
//pList[1, ......n]istasyonların fiyatlarını tutan n elemanlı dizi
//k : Aracın benzin aldıktan sonra gitmesi gereken en kısa mesafe
//f :Aracın benzin aldıktan sonra gidebileceği maksimum mesafe
n ← mList.size
list ← emptylist
for i ←1 to pow(2, n) do
  subset ← emptyList
  for j ←1 to n do
    if ((i&(1 << j)) > 0) then
      subset.append(j)
    end if
  end for
  if subset.size > 1 then
    list.append(subset)
  end if
end for
possiblePaths ← emptyList
for each a ∈ list do
  counter ← 0
  for i ←1 to a.size do
    if a.get(1) ≠ 0 or a.contains(n) ≠ 1 then
      counter ← counter + 1
      continue
        end if
        if i + 1 < a.size() then
          if (mList[a.get(i+1)] − mList[a.get(i)] < kormList[a.get(i+1)] −
          mList[a.get(i)] > f) then
            counter ← counter + 1
            continue
          end if
        end if
      end for
      if counter = 0 then
        possiblePaths.append(a)
      end if
    end for
    costs ← ∅
    i ← 0
    for each index ∈ possiblePaths do
      costs[i] ← 0.0
      for a ←1 to index.size do
        cost[i] ← cost[i] + pList[index.get(a)]
      end for
      i ← i + 1
    end for
    minimum ← cost[0]
    shortPathIndex ← 0
    for j ←1 to costs.size do
      if costs[j] ≤ minimum then
        minimum ← costs[j]
        shortPathIndex = j
      end if
    end for
```

for *short* $\leftarrow 1$ to *possiblePaths.get(shortPathIndex).size* **do**
  print(possiblePaths.get(shortPathIndex).get(short))
end for
return *possiblePaths*

Algoritmanın Asimptotik Analizi

Brute Force gerçekleştirilen Fonksiyon'da oldukça fazla döngü bulunmakta. İlk döngüye baktığımızda 2^n kere devam edecek bir döngü olduğu görülüyor. Bu döngünün içerisinde hiçbir koşula bakmaksızın n kere devam edecek başka bir döngü bulunmakta. Bu da demek oluyor ki içerideki döngü 2^n lik döngünün içerisinde olduğu için toplamda 2^n*n kere tekrarlanacak.
Diğer döngüler ise 2^n*n kere tekrar eden döngünün sonuçlarını koşullara bağlı olarak elediği için daha az ya da eşit süreceği aşikardır.Bundan dolayı;
Algoritma her ne olursa olsun 2^n*n şeklinde çalışacaktır. Çünkü her durumda tüm alt kümeleri bulmak zorundadır. Bu yüzden algoritmanın **Best,Average ve Worst Case'i O(2^n*n)** dir.

Algoritmanın input büyüklüklerine göre çalışması

Brute Force olarak çalışan algoritma her seferinde 2^n*n şeklinde çalışacağından dolayı input sayısı arttıkça fonksiyonun hızının azalması beklenmektedir.

```
mList
0.0    5.0    7.0    9.0    11.0    13.0    15.0
pList
0.0    6.0    8.0    7.0    3.0    4.0    0.0
k:2.0
f:7.0
input size : 7
Gidilecek Yol
0 1 4 6
Gidilecek yolun toplam ücreti
9.0
Toplam süre:0.017 saniye
```

```
mList
0.0    10.4    18.3    28.9    39.41    55.1    57.25    64.5    68.8    70.5
pList
0.0    3.7    4.1    3.6    4.8    3.17    3.2    3.21    3.33    0.0
k:15.0
f:25.0
input size : 10
Gidilecek Yol
0 2 4 5 9
Gidilecek yolun toplam ücreti
12.069999999999999
Toplam süre:0.016 saniye
```

```
mList
0.0    8.0    16.0    24.14    30.0    35.0    39.0    41.0    43.0    49.0    50.0    51.0    58.0    65.0    71.0    75.0    80.0    84.0    89.0
pList
0.0    1.2    1.3    2.15    1.9    2.8    1.54    2.0    1.13    2.0    1.88    1.74    1.56    1.44    1.1    1.25    1.37    1.36    1.39
k:5.0
f:10.0
input size : 21
Gidilecek Yol
0 1 2 3 4 6 9 12 13 14 16 18 19 20
Gidilecek yolun toplam ücreti
18.45
Toplam süre:2.594 saniye
```

```
mList
0.0     61.5     120.0    147.5    161.0    179.5    219.0    278.5    297.0    321.5    336.0    393.5    426.0    442.5    490.0    548.
pList
0.0     15.4     19.4     5.4      10.8     6.6      11.4     7.4      18.5     9.2      17.5     14.2     6.6      13.2     15.2     17.8
k:30.0
f:150.0
input size : 22
Gidilecek Yol
0 3 7 12 15 17 21
Gidilecek yolun toplam ücreti
52.5
Toplam süre:5.293 saniye
```

3.Dinamik Programlama(Dynamic Programming)

Dinamik Programlama’da önceki olaylara ait veriler kullanılır.
Tanımı verilmiş olan problemin çözümü için kullandığımız 2.yöntem olan Brute Force, problemin çözümü için çalışma prensibi şu şekildedir;
Bir istasyondan diğer istasyona gidilebiliyorsa, yani istasyonlar arası mesafe k ile f arasında ise bu iki istasyon birbirlerine komşudur ve bu komşuluk ikili dizide tutulur. Örneğin, 3. istasyondan 4.istasyona gidilebiliyor bu durum tabloda şu şekilde tutulur:
tablo[3][4]=4.istasyonun fiyatı.
Bir sonraki aşamada ise hiçbir şekilde ulaşılamayan bir istasyonun komşuluk ilişkilerini kesmektir.
Başlangıç noktasından x. istasyona hiçbir şekilde ulaşılmıyorsa o istasyonu devre dışı bırakıyoruz. Örneğin, 5.istasyona hiçbir şekilde gidemiyorsak tablo[5][6]=-1 durumuna getirilir ve bu da iki istasyon arası bir komşuluk bağı olmadığını gösterir.
Son aşamada ise bitiş noktasına ulaşmak için en düşük maliyeti bulmak için tabloda değişikler yapılır. Örneğin, tablo[2][3] 2’den 3’e gitmenin maliyetini tutar. Fakat 2 den 3e gitmek için ilk önce 2’ye ulaşmamız gerekir. Bu yüzden 2’ye ulaşmanın en düşük maliyetli yolu hesaplanır ve 2 den 3e gitmenin maliyetiyle birlikte toplam maliyete eklenir ve tablo boyunca aynı işlemler sürdükçe tablonun son elemanlarındaki sayıların en küçüğü (-1 hariç) en düşük maliyetli yolu bize vermektedir.

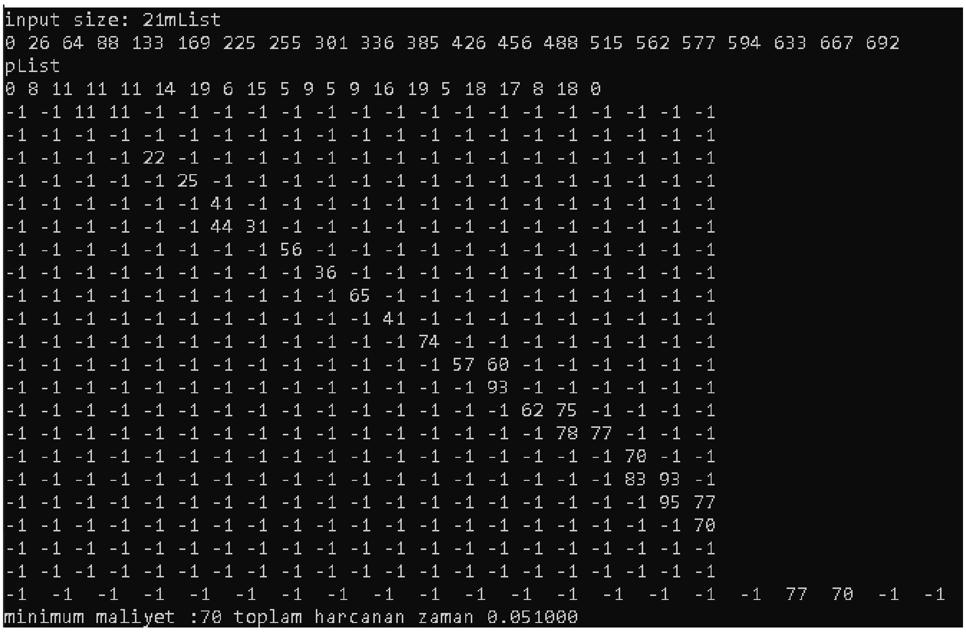
```
dynamicProgramming(mList, pList, size, k, f)
M ← size
N ← size
for i ←1 to M do
  for j ←1 to N do
    tablo[i][j] ← −1
  end for
end for
for i ←1 to M do
  for j ←i+1 to N do
    fark ← mList[j] − mList[i]
    if fark ≥k and fark ≤f then
      tablo[i][j] ← pList[j]
    end if
  end for
end for
for i ←2 to M do
  sayac ←0
  for a ←1 to i do
    if tablo[a][i] ≠ −1 then
      sayac ← sayac + 1
    end if
  end for
  for j ←i+1 to N  do
    if sayac = 0 then
      tablo[i][j] ← −1
    end if
  end for
end for
for i ←1 to M do
  for j ←1 to N do
    if tablo[i][j] > −1 then
      minimum ← 0
      for c ←1 to i do
        if minimum = 0 and tablo[c][i] ≠ −1 then
          minimum ← tablo[c][i]
          continue

        end if
        if tablo[i][j] ≤ minimum and tablo[c][i] ≠ −1 then
          minimum ← tablo[c][i]
        end if
      end for
      tablo[i][j] ← tablo[i][j] + minimum
    end if
  end for
end for
```

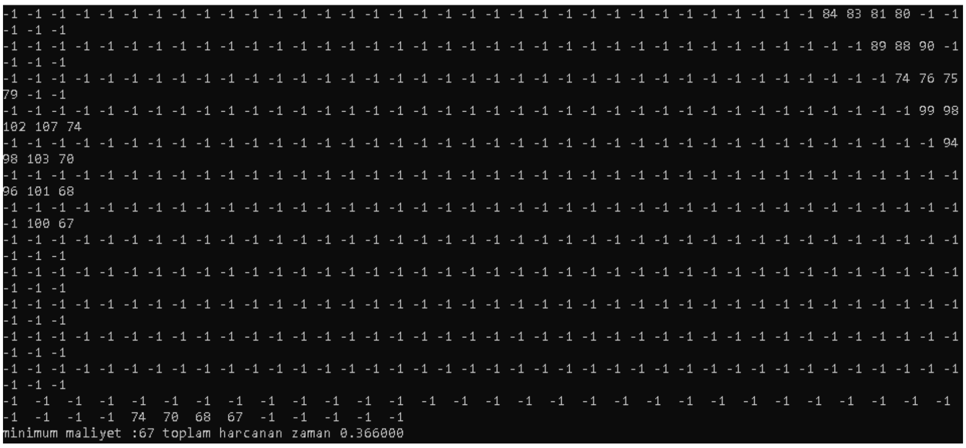
Algoritmanın Asimptotik Analizi

Algoritmanın birçok bölümünde iç içe iki tane döngü varken bir yerde iç içe 3 döngü bulunmaktadır. Iç içe iki tane döngü olan döngüler tablo doldurmaya yönelik döngülerdir yani input büyüklüğüne bağlıdırlar. İç içe 3 döngü olan kısımda ise tabloda geri gitme,eski verileri kullanma var. Bu eski veriler ne kadar uzaktan gelirse çalışma zamanı o kadar artar. En kötü durumda dizinin son elemanında minimum elemanı bulacağımızı varsayarsak **Worst Case O(n^3)** dür. En iyi durum ise döngünün içerisindeki koşulu sağlayan bir ifadenin olmaması durumudur. Çünkü o ifade sağlanırsa eğer 3.bir döngüye girmiş olur. Bu durumdan ötürü algoritmanın **Best Case'i O(n^2)** dir.

Algoritmanın input büyüklüklerine göre çalışması



```
input size: 21mList
0 26 64 88 133 169 225 255 301 336 385 426 456 488 515 562 577 594 633 667 692
pList
0 8 11 11 11 14 19 6 15 5 9 5 9 16 19 5 18 17 8 18 0
-1 -1 11 11 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 22 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 25 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 41 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 44 31 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 56 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 36 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 65 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 41 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 74 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 57 60 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 93 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 62 75 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 78 77 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 70 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 83 93 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 95 77
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 70
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  77  70  -1  -1
minimum maliyet :70 toplam harcanan zaman 0.051000
```



```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 84 83 81 80 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 89 88 90 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 74 76 75
79 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 99 98
102 107 74
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 94
98 103 70
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
96 101 68
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 100 67
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
-1  -1  -1  -1  74  70  68  67  -1  -1  -1  -1  -1
minimum maliyet :67 toplam harcanan zaman 0.366000
```

Yukarıdaki sonuç 43 inputlu diziye aittir.

```
input size: 10mList
0 10 18 29 39 55 57 65 69 71
pList
0 4 4 4 5 3 5 6 4 0
-1 -1 4 4 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 9 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 7 9 -1 -1 -1
-1 -1 -1 -1 -1 12 14 15 13 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 7
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1  -1  -1  -1  -1  7  -1  -1  -1  -1
minimum maliyet :7 toplam harcanan zaman 0.029000
```

```
input size: 7mList
0 5 7 9 11 13 15
pList
0 6 8 7 3 4 0
-1 6 8 -1 -1 -1 -1
-1 -1 14 13 9 -1 -1
-1 -1 -1 15 11 12 -1
-1 -1 -1 -1 16 17 13
-1 -1 -1 -1 -1 13 9
-1 -1 -1 -1 -1 -1 12
-1 -1 -1 -1 -1 -1 -1
-1  -1  -1  13  9  12  -1
minimum maliyet :9 toplam harcanan zaman 0.017000
```

İnput büyüklükleri yakın olduğu takdirde aralarında pek fark görülmeyen bu algoritmalar arasındaki fark input büyüklüğü artınca meydana çıkmaktadır. Brute force 25 ve sonrası için bilgisayarımda hesaplanamadı. Diğer algoritmalarda input büyüklüğü arttıkça çalışma zamanlarını 2 katına çıkarmaya başladı. Verimsiz bir algortima yazıldığında ufak inputlarda bile bilgisayarın zorlanabileceğini söyleyebiliriz.

Algoritmaları çalışma zamanlarının karmaşıklığına göre sıralayacak olursak

Brute Force>Dynamic>Greedy>DecreaseAndConquer