**20.05.2020**

# BIL 206 ALGORİTMA ANALİZİ VE TASARIMI

# DÖNEM PROJESİ

**Semih Demir**

**18120205005**

## 1. Giriş

Bu projede amaç, bir aracın belli bir mesafeyi farklı aralıklardaki benzin istasyonlarına uğrayarak en optimum şekilde gitmesini sağlamaktır. Problemin tam tanımı ise şu şekildedir;

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.

Bu veriler ile aracın en az benzin fiyatıyla yolculuğu tamamlaması için hangi benzin istasyonlarında durması gerektiğini bulmamız istenmektedir.

Değişkenlerimiz ise su şekilde olmalıdır ;

| | |
|---|---|
| mList[1,..,n] | İstasyonların başlangıç noktasına uzaklıklarını içeren array |
| pList[1,….,n] | İstanyonların bir depo benzin maliyeti |
| Limit | K değişkeni olarak tanımlanan minimum durma uzaklığı |
| F | Bir depo ile gidilebilecek maksimum uzaklık |

Bu bilgiler eşliğinde çözümlerimizi bruteforce, dynamic programming, greedy ve divide and conquer metotlarıyla yapmamız ve çalışma zamanlarımızı karşılaştırmamız isteniyor.

## 2.Yöntemler

### a) Greedy Çözüm

Bir problemin çözümünde ileride doğabilecek sonuçlar göze alınmadan, mevcut şartlar altında en iyi olan seçimin yapılması gerektiğini savunan çözüm yöntemidir. Temel ilkesi o adımda bulunan çözümün genelde de işe yarayabileceğinin düşünülmesidir. Yani kısaca karşılaşılan seçenekler arasından seçim yapmak gerektiğinde problemin çözümü için gerekilenin yapılmasıdır.

Problemimize gelirsek bu yöntemi problemimize uyarlarsak sözde kodumuz şu şekilde olur;

```
curr_refil = 0
price_refill = 0

while limit < distance:
    temp = curr_refil;
    if curr_refil >= n || mlist[curr_refil] > limit
       return -1;

    while curr_refil < n - 1 && mlist[curr_refil + 1] <= limit
       curr_refil += 1;

    limit = mlist[curr_refil] + f;
    if mlist[curr_refil] - mlist[temp] >= k
       price_refill += plist[curr_refil];
return price_refill;
```

Asimptitok çalışma zamanımız ise burada her her bir döngümüz n kere çalıştığın için **$O(n^2)$** olur.

**b) Bruteforce Çözüm**

Bu yöntemi kısaca anlatmak gerekirse dilimizde kaba kuvvet algoritması olarak da anılır. Burada amaç problemin çözümünü en basit şekilde düşünüp uygulamaktır. Bir nevi tüm yolların denenmesi de denebilir. Sonuç odaklıdır ve performans aranmaz.

Problemimize gelirsek bruteforce yöntemi kullanılarak şu şekilde bir çözümleme yapabiliriz;

```
    add_cost = 0;

    if point == count
        if mlist_temp[0] > f
            return 1;
        //0(n)
        For i (0, count):
            add_cost += stop[i]

        costs.push_back(add_cost);

      // O(n)
    For i (begin, count-point <= (finish-i)+1 && i<=finish)
        mlist_temp[point] = mlist[i];
        stop[point] = plist[i];
        // 0(nlogn)
        brute_solve(mlist,plist,k, f, mlist_temp, begin+1, finish,
                    point+1, count--,stop,costs);
        if(mlist_temp[i+1] - mlist_temp[i] > f || mlist_temp[i+1] -
           mlist_temp[i] <= k )
                break;
       sort(costs.begin(),costs.end())

       return costs;
```

Asimptitok çalışma zamanımız ise burada **O($2^n$.logn)** olur.

**c) Dinamik Çözüm**

Dinamik programlama karışık problemlerin daha basit düzeylere indirilerek çözülmesini esas alan bir optimizasyon yöntemidir. Optimizasyondaki amaç, problemdeki kısıtlayıcı koşullar altında bu problemle ilgili en iyi karara varmaktır. Bir problem üzerinde dinamik programlama uygulayabilmek için o problemin alt problemlere parçalanabilir veya bir önceki problemin karakteristiğini koruyacak şekilde çözümü daha kolay başka probleme dönüştürülebilir olması yeterlidir

Yönteme göre optimum çözüm *başlangıç durumundan bağımsız olarak diğer çözümler ile çözüm sonuçlarına göre optimum çözümler ardışıklığı*dır. Yani, başlangıçta alt problemlerin çözümü bulunup buradan elde edilen verilerle daha büyük alt problemler çözüldüğünde problemin kendisi de çözülmüş olmaktadır.

Çözümümüz ise şu şekildedir;

```
for i (0,n)
       stations[i].pos = mlist[i];
        stations[i].cost = plist[i];

    // find next cheaper station for each station
    int stacklen = 0;
    for i (n -1, 0)

     while (stacklen > 0 && stations[s[stacklen - 1]].cost >= statio
ns[i].cost)
       stacklen--;

        nextSmall[i] = (stacklen == 0 ? -1 : s[stacklen - 1]);
        s[stacklen] = i;
        stacklen++;
```

```
curGas -= stations[0].pos; // move to station 1
    cost = 0;
    for i(0,n)
        // gas is less than 0 means it is impossible to reach the st
ation
        if curGas < 0
            return 0;

        gasNeeded = min(maxGas, (nextSmall[i] == 1 ? dist :
stations[nextSmall[i]].pos) - stations[i].pos);
        if gasNeeded > curGas
            cost += (long long)stations[i].cost;
            curGas = gasNeeded;

        curGas = (i == n - 1 ? dist : stations[i + 1].pos) - station
s[i].pos;
```

Asimptitok çalışma zamanımız **O(nlog)**'dir.

**d) Divide Çözüm**

Parçala ve fethet , problemi daha küçük alt problemlere ayırıp, her alt problemin çözülmesine ve daha sonra bütün çözümlerin birleştirilmesine dayanır.

Parçala ve fethet yönteminde alt problemlerin çözümü birbirinden bağımsızdır. Alt problemler ve bunların çözümü arasında etkileşim yoktur. Dinamik programlamada ise bir alt problemin çözümü başka alt problemlerin çözümünde de kullanılır. Yani alt problemlerin çözümü birbiri ile etkileşim halindedir.

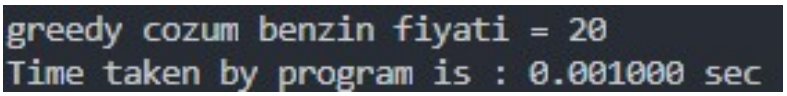
Çözümü ise bu yöntem yaklaşımıyla yapabiliriz. Alt problemleri tek tek birleştirerek bütüne varmamız mümkündür.

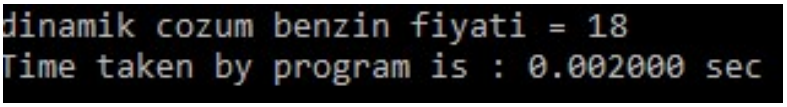
## 3. Sonuç

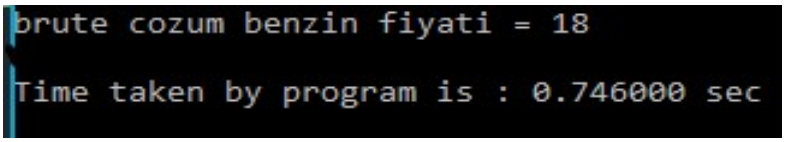
Tüm yöntemlerimize baktığımızda sonuç olarak görüyoruz ki hepsi çözüme farklı şekilde yaklaştığı için sonuçları ve geçen zamanları birbirinden farklı olabilmektedir. Bir problem için tek bir optimum çözüm yolu olmadığı gibi, problemlere göre de yaklaşımların çözüm için gerekli zamanı birbirinden farklı olabilmektedir. Bizim problemimiz de ise en hızlı çözüme sırasıyla;

1) **Divide**
2) **Dinamik Programlama**
3) **Greedy**
4) **Bruteforce**

Yöntemleri ile ulaşabiliyoruz.




Çalışma sürelerini ise, farklı inputlar ile programımızı çalıştırdığımızda şu şekilde görmemiz mümkün.