**1.** İki pozitif tamsayı bolunen ve bolen verildiğinde, bolunen / bolen işlemini **sadece çıkarma kullanarak** gerçekleştiren **recursive** bir C# fonksiyonu yazınız.

using System;

class Program

{

static void Main()

{

int bolunen = 17;

int bolen = 5;

int sonuc = Bol(bolunen, bolen);

Console.WriteLine("Sonuç: " + sonuc); }

static int Bol(int bolunen, int bolen)

{

// Bölünen, bölenden küçükse sonuç sıfırdır (base case)

if (bolunen < bolen)

return 0;

// Her çıkarma 1 bölme anlamına gelir

return 1 + Bol(bolunen - bolen, bolen);

}

}

**2.** İki adet 32 bitlik sayıyı bitwise operatörler kullanarak toplayan bir C# metodu yazınız.

Taşma (overflow) olup olmadığını da kontrol ediniz. Taşma olursa konsola "Taşma var" yazdırılacaktır. Olmazsa taşma yok yazdırılacaktır.

using System;

class Program

{

static void Main()

{

int x = 2147483647; // int.MaxValue

int y = 1;

// x ve y'yi toplayalım

int toplam = x + y;

// toplamdan x’i çıkarırsam sonuç y’ye eşit olmalı

int testY = toplam - x;

if (testY != y)

{

Console.WriteLine("Taşma var");

}

else

{

Console.WriteLine("Taşma yok");

}

Console.WriteLine("Sonuç: " + toplam);

}

}

**3.** Bir diyetisyen, toplamda en fazla 15 gramlık bir tatlı tabağı hazırlamak istiyor. Bu tabakta en fazla 3 adet farklı malzeme kullanılabilir. Elinde şu malzemeler var:

| **Malzeme** | **Kalori (her 1 gram için)** | **Şeker (her 1 gram için)** |
| --- | --- | --- |
| A | 5 | 2 |
| B | 4 | 3 |
| C | 7 | 1 |
| D | 3 | 4 |

**Toplam kaloriyi maksimum yaparken, toplam şeker miktarı 20’yi geçmemelidir.** Her malzemeden en fazla **5 gram** kullanılabilir. Her malzemenin gram miktarı **tam sayı** olmak zorundadır. Bu problemi çözecek bir algoritma yazınız. Brute-force ya da recursive bir yaklaşım kullanınız. Tüm kombinasyonları deneyen ve şartları kontrol eden yapıyı kurunuz. En uygun kombinasyonu bulup, toplam kalori miktarıyla birlikte yazdırınız. Yorum satırları zorunludur.

**Amaç:**

* En fazla 15 gram
* En fazla 3 malzeme
* Her malzemeden max 5 gram
* Şeker toplamı en fazla 20
* **Kalori maksimum olacak**

Deneme-yanılma (brute-force) yapacağız:  
Her malzemeyi 0-5 gram arası seçiyoruz (0 = hiç alma).  
Tüm kombinasyonları tek tek deniyoruz,  
**Kurallara uyuyorsa** kalorisine bakıyoruz,  
Şu ana kadarki **en yüksek kalori** mi diye kontrol ediyoruz.

**Denklemler:**

•x₁: A'dan kaç gram alındı (0 ≤ x₁ ≤ 5, tamsayı)

•x₂: B'den kaç gram alındı (0 ≤ x₂ ≤ 5, tamsayı)

•x₃: C'den kaç gram alındı (0 ≤ x₃ ≤ 5, tamsayı)

•x₄: D'den kaç gram alındı (0 ≤ x₄ ≤ 5, tamsayı)

•y₁ = 1 ise A seçilmiş, 0 ise seçilmemiş

•y₂ = 1 ise B seçilmiş

•y₃ = 1 ise C seçilmiş

•y₄ = 1 ise D seçilmiş

Z = 5x₁ + 4x₂ + 7x₃ + 3x₄ (Maksimize et)

x₁ + x₂ + x₃ + x₄ ≤ 15 (Gram limiti)

2x₁ + 3x₂ + 1x₃ + 4x₄ ≤ 20 (Şeker Limiti)

y₁ + y₂ + y₃ + y₄ ≤ 3 (En fazla 3 tane seçim)

**MALZEME BAŞI MAKSİMUM GRAM SINIRI**

Problemde her malzemeden maksimum 5 gram alınabilir denmişti.

Bu yüzden y1 = 1 olduğunda x1 ≤ 5 sınırı lazım.

Biz de bu sınırı 5 \* y1 şeklinde yazıyoruz ki:

y1 = 0 ise → x1 = 0 olur

y1 = 1 ise → x1 ≤ 5 olur

x₁ ≤ 5y₁

x₂ ≤ 5y₂

x₃ ≤ 5y₃

x₄ ≤ 5y₄

özetle:

x₁ + x₂ + x₃ + x₄ ≤ 15

2x₁ + 3x₂ + 1x₃ + 4x₄ ≤ 20

y₁ + y₂ + y₃ + y₄ ≤ 3

x₁ ≤ 5y₁

x₂ ≤ 5y₂

x₃ ≤ 5y₃

x₄ ≤ 5y₄

xᵢ ∈ {0,1,2,3,4,5}

yᵢ ∈ {0,1}

**Maksimize et: Z = 5x₁ + 4x₂ + 7x₃ + 3x₄**

**//Encapsulation yapacağız.** Encapsulation dediğimiz şey, İlgili verileri ve onlarla yapılan işlemleri bir class a koymak. Bu sayede hem kod dağınık durmaz, hem de dışarıdan biri bu sınıfı güvenle kullanabilir. Metod aşırı yükleme yapmak istemiyorum.

**using System;**

**class TatliSecici**

**{**

**private int[] kaloriler = { 5, 4, 7, 3 };**

**private int[] sekerler = { 2, 3, 1, 4 };**

**private string[] isimler = { "A", "B", "C", "D" };**

**private int[] secim = new int[4]; //**Her malzemeden kaç gram aldığımızı tutan dizi

**private int[] enIyiSecim = new int[4];//** finalde kazanan kombinasyonu tutan bellek  **private int enYuksekKalori = -1;**

**public void Hesapla()**

**{**

**Dene(0);**

**YazdirSonuc();**

**}**

**private void Dene(int index)**

**{**

**if (index == 4)**

**{**

**int toplamGram = 0;**

**int toplamKalori = 0;**

**int toplamSeker = 0;**

**int seciliCesit = 0;**

**for (int i = 0; i < 4; i++)**

**{**

**if (secim[i] > 0)**

**{**

**seciliCesit++;**

**toplamGram += secim[i];**

**toplamKalori += kaloriler[i] \* secim[i];**

**toplamSeker += sekerler[i] \* secim[i];**

**}**

**}**

**if (toplamGram <= 15 && toplamSeker <= 20 && seciliCesit <= 3)**

**{**

**if (toplamKalori > enYuksekKalori)**

**{**

**enYuksekKalori = toplamKalori;**

**for (int i = 0; i < 4; i++)**

**enIyiSecim[i] = secim[i];**

**}**

**}**

**return;**

**}**

**//** Bu malzemeden 0 gram alsam ne olur?  
//1 gram alsam ne olur?  
//5 gram alsam ne olur?”  
//Hepsini sırayla dene

**for (int g = 0; g <= 5; g++)**

**{**

**secim[index] = g;**

**Dene(index + 1);**

**}**

**}**

**private void YazdirSonuc()**

**{**

**Console.WriteLine("En yüksek kalori: " + enYuksekKalori);**

**Console.Write("Seçilenler: ");**

**for (int i = 0; i < 4; i++)**

**{**

**if (enIyiSecim[i] > 0)**

**Console.Write("{0}:{1}g ", isimler[i], enIyiSecim[i]);**

**}**

**}**

**}**

class Program

{

static void Main()

{

TatliSecici secici = new TatliSecici();

secici.Hesapla();

}

}

* Dene() fonksiyonu her malzeme için olası tüm gram miktarlarını deniyor.
* Her kombinasyon tamamlandığında, toplam gram, şeker ve çeşit sayısı kontrol ediliyor.
* Eğer kurallara uyuyorsa ve kalori daha yüksekse, bu kombinasyon kaydediliyor.
* En sonunda yazdırılan şey, en yüksek kaloriyi sağlayan geçerli kombinasyondur.

Dene (index)

Recursive fonksiyon:

Her malzeme için 0 ile 5 gram arası deneme yapıyor.

4 malzeme tamamlandığında (index == 4):

Toplam gramı, şekeri, kaç çeşit malzeme seçildiğini hesaplıyor.

Eğer tüm kurallara uyuyorsa:

Bu kombinasyonun kalorisi önceki en iyiden büyükse yeni en iyi kabul ediliyor.

**Dallanma mantığı:**

Dene(0) → secim[0] = 3

 └── Dene(1) → secim[1] = 1

   └── Dene(2) → secim[2] = 2

     └── Dene(3) → secim[3] = 5

       → index == 4 → hesap yapılır

     ← geri dön

    ← diğer D seçenekleri denenir

  ← diğer C seçenekleri denenir

← diğer B seçenekleri denenir

**4. BELLMANFORD**

Bir ağırlıklı yönlü grafikte, bir başlangıç düğümünden tüm diğer düğümlere olan en kısa yolları bulmak için kullanılır.

En büyük farkı: Negatif ağırlıklı kenarları da doğru şekilde işler. Ama negatif ağırlıklı döngü varsa, bunu da fark eder ve uyarı verir. Dijkstra gibi her seferinde en kısa olanı seç demez. Onun yerine tüm kenarları tek tek deneyeyim, acaba daha kısa yol var mı? diye relax (gevşetme) işlemi yapar.

dist[] dizisi: Başlangıçtan her düğüme olan mesafeyi tutar.

İlk başta hepsi ∞ (sonsuz), sadece başlangıç düğümü 0.

Grafiğin tüm kenarlarını al.

Bu kenarları V-1 kere döngüyle kontrol et (V: düğüm sayısı)

Her kenar için: dist[u] + ağırlık < dist[v] ise → güncelle:

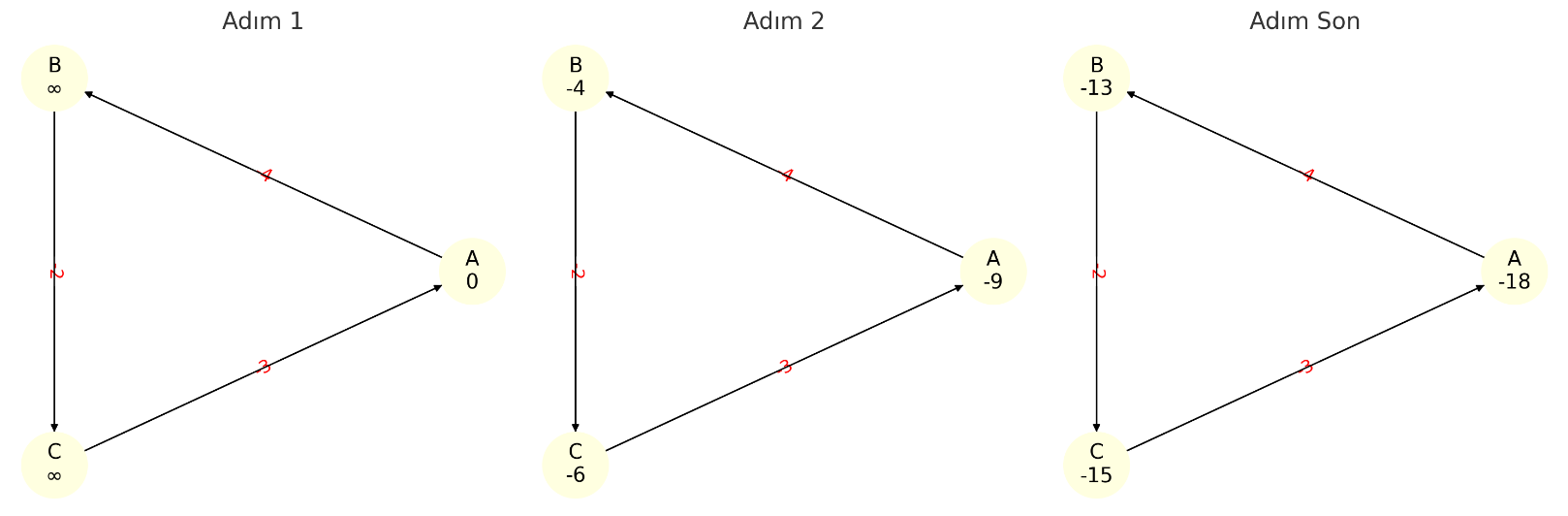
dist[v] = dist[u] + ağırlık;

Bu gevşetme işlemiyle tüm kısa yollar ortaya çıkar.

Son turda bir daha gevşeme oluyorsa → negatif döngü var, çünkü en kısa yol hala kısalıyorsa bu döngüdür.

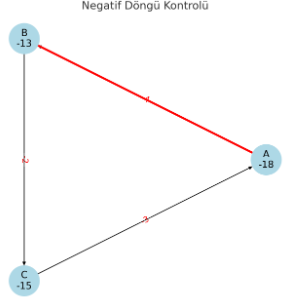
**BELLMAN FORD NE ZAMAN KULLANILIR**

| **Durum** | **Kullanılır mı?** |
| --- | --- |
| Kenar ağırlıkları pozitif | Kullanılabilir ama Dijkstra daha hızlıdır |
| Kenar ağırlıkları negatif | Bellman-Ford tercih edilir |
| Grafikte negatif döngü olabilir | Saptayabilir |
| Grafikte negatif döngü varsa | En kısa yol bulunamaz |
| Düğümler: A(0), B(1), C(2)  Kenarlar:  A → B, ağırlık: -4  B → C, ağırlık: -2  C → A, ağırlık: -3 |  |



Yukarıdaki görselde Bellman-Ford algoritmasının adım adım nasıl çalıştığını gösterdik.

* Başlangıçta yalnızca A düğümünün mesafesi 0, diğerleri ∞ (ulaşılmaz).
* Her adımda kenarlar tek tek kontrol edilip daha kısa yol varsa mesafeler güncellenir.
* Son adımda tüm düğümlere giden yollar güncellenmiş olur.  
  Her adımda mesafeler azalıyor ve sonrasında hâlâ daha azaltılabiliyor , yani neg döngü var.



 u: Kenarın başladığı düğüm

 v: Kenarın bittiği düğüm

 w: O kenarın ağırlığı

using System;

class Program

{

static void Main()

{

// Dugum sayisi

//Grafimizde **3 tane düğüm** var.  
//Yani **A, B, C** (bunları sırasıyla 0, 1, 2 diye kullanacağız).

int N = 3;

// Kenar bilgileri: [baslangic, bitis, agirlik]

int[,] kenarlar = new int[3, 3]

{

{ 0, 1, -4 }, // A -> B

{ 1, 2, -2 }, // B -> C

{ 2, 0, -3 } // C -> A

};

// Kenar sayisi

int kenarSayisi = 3;

// Mesafe dizisinde her düğüme olan mesafeleri tutacağız. Yani A’ya kaç birimle ulaşılır, B’ye kaçla, C’ye

int[] mesafe = new int[N];

for (int i = 0; i < N; i++)

{

mesafe[i] = 2147483647; // int.MaxValue

}

mesafe[0] = 0; // Baslangic dugumu A

// N-1 kere tüm kenarları dene (Bellman-Ford kuralı)

for (int i = 0; i < N - 1; i++)

{

for (int j = 0; j < kenarSayisi; j++)

{

int u = kenarlar[j, 0]; // baslangic

int v = kenarlar[j, 1]; // bitis

int w = kenarlar[j, 2]; // agirlik

// Eğer u'ya zaten gidebiliyorsan, ve oradan v'ye geçmek **daha kısa yol** sağlıyorsa, hemen v'nin mesafesini **daha kısa //olanla** güncelle. Yani hep daha ucuz yolu seçiyoruz.

if (mesafe[u] != 2147483647 && mesafe[u] + w < mesafe[v])

{

mesafe[v] = mesafe[u] + w;

}

}

}

// Negatif dongu kontrolu

bool negatifDongu = false;

for (int j = 0; j < kenarSayisi; j++)

{

int u = kenarlar[j, 0];

int v = kenarlar[j, 1];

int w = kenarlar[j, 2];

//tur bitti ama hâlâ bir yolu **daha da kısaltabiliyorsak**, negatif döngü var.  
 if (mesafe[u] != 2147483647 && mesafe[u] + w < mesafe[v])

{

negatifDongu = true;

break;

}

}

if (negatifDongu)

{

Console.WriteLine("Negatif agirlikli bir dongu bulundu!");

}

else

{

Console.WriteLine("A noktasindan en kisa mesafeler:");

Console.WriteLine("A: " + mesafe[0]);

Console.WriteLine("B: " + mesafe[1]);

Console.WriteLine("C: " + mesafe[2]);

}

}

}

| **Aşamalar** | **Ne Yapıyor** |
| --- | --- |
| Başlangıçta herkes sonsuz | Yola çıkmadan kimseye ulaşamıyoruz |
| A’ya mesafe 0 | Kendine gitmek 0 maliyet |
| Her yol kontrol edilir | Daha kısa yol varsa güncellenir |
| En sonda ekstra kontrol | Hala kısalıyorsa → Negatif döngü |

**Dijkstra ve Bellman-Ford Algoritmalarının Karşılaştırması**

1. **Negatif Ağırlık Desteği**  
   Dijkstra algoritması negatif kenar ağırlıklarını desteklemez.  
   Bellman-Ford algoritması ise negatif ağırlıkları destekler ve negatif döngüleri tespit edebilir.
2. **Zaman Karmaşıklığı**  
   Dijkstra'nın zaman karmaşıklığı, min-heap kullanıldığında O((V + E) log V)'dir.  
   Bellman-Ford’un zaman karmaşıklığı ise O(V \* E) olarak bilinir.
3. **Yöntem**  
   Dijkstra, en kısa yolu bulmak için açgözlü (greedy) bir seçim yapar.  
   Bellman-Ford, her kenarı tekrar tekrar kontrol ederek mesafeleri iyileştirir.
4. **Kullanım Alanı**  
   Dijkstra algoritması, genellikle tüm kenar ağırlıklarının pozitif olduğu grafiklerde tercih edilir.  
   Bellman-Ford algoritması ise negatif ağırlıklı kenarların bulunduğu grafiklerde kullanılır.
5. **Aşamalar**  
   Dijkstra, her adımda en kısa mesafeye sahip düğümü işleyerek ilerler.  
   Bellman-Ford, her kenarı toplam V−1 kez tarar ve günceller.
6. **Graf Tipi**  
   Dijkstra algoritması, yönlü veya yönsüz grafiklerde (pozitif ağırlıklı) çalışabilir.  
   Bellman-Ford ise yalnızca yönlü grafiklerde kullanılır; çünkü negatif döngü varsa yönsüzlük mantık hatası doğurabilir.
7. **Kodda Güncelleme Şartı**  
   Dijkstra’da güncelleme yapılması için yeni yolun mevcut yoldan daha kısa olması gerekir (yeni yol < bilinen yol).  
   Bellman-Ford’da da aynı mantık geçerlidir, ancak ek olarak negatif döngü kontrolü yapılır.