Äpplen och päron

Här gäller det bara att göra det som står i beskrivningen: läs in antalet äpplen och päron, multiplicera med kostnaden, jämför svaren. Lösning i python:

```
A = int(input())
P = int(input())
price_A = A*7
price_P = P*13

if price_A > price_P:
  print("Axel")
elif price_A < price_P:
  print("Petra")
elif price_A == price_P:
  print("lika")
```

Personnummer

Detta är också en ren implementationsuppgift – beskrivningen säger vad som ska göras.

Såhär kan man skriva det i python:

```
s = input()
year = 1900 + int(s[0]+s[1])
if year < 1920:
    year += 100
if s[6] == '+':
    year -= 100
print(str(year) + s[2:6] + s[7:])
```

Hundraelva

Problemet frågar efter det minsta antalet sedlar på formen 1, 11, 111, etc som måste användas för att betala exakt N kronor.

Idén för lösningen är att en girig strategi fungerar: *det är alltid optimalt att välja minst en sedel med största valören <= N.* (För en bevis-sketch, se nedan).

Med denna insikt är lösningen relativt enkel. Man hittar den största sedeln som är maximalt N, och så kan man dra av värdet på denna sedel från N och fortsätta på liknande sätt.

```
n = int(input())

ans = 0
while n > 0:
    sedel = 1
    while 10*sedel+1<=n:
        sedel = 10*sedel+1
    # sedel är nu största talet <= n på formen 11...1
    n -= sedel
    ans += 1

print(ans)
```

Varför är detta snabbt nog? I svaret behövs maximalt 10 st av varje sorts sedel (om N är strikt större än 10 gånger en sedels värde använder man först en större sedel istället), och det finns bara 8 stycken sedlar som är relevanta då det är garanterade att N <= 10^9.

Bevisidé för att giriga strategin fungerar.

Låt $a_1 = 1$, $a_2 = 11$, $a_3 = 111$, $a_4 = 1111$ osv. Det gäller att $10a_i + a_j = a_{i+1} + 10a_{j-1}$ för alla $i >= j >= 2$ och $10a_i + a_1 = a_{i+1}$ för alla $i >= 1$. Detta betyder att om man har sedlar som summerar till minst a_{i+1}, så är det aldrig sämre att byta sedlarna till att använda minst en till sedel av valören a_{i+1}.

Lavapaddling

Säg att du har P stycken paddar, varje paddel kan man ta K paddeltag med och det är M meter till nästa ö. Vad är det bästa sättet att använda paddlarna på för att komma till nästa ö? Eftersom du kan laga alla paddlar som inte gått sönder helt när du kommer till nästa ö är det enda som spelar roll hur många paddlar du har kvar. Därför är det optimalt att först använda alla paddlar så mycket som möjligt utan att någon går sönder, det vill säga paddla K-1 paddeltag med varje paddel. Först därefter bör man använda det sista paddeltaget på några paddlar, om det behövs. Med denna insikt kan vi skapa en funktion som givet P, K, M kan räkna ut hur många paddlar vi har kvar när vi kommer till nästa ö:

```
def paddlar_kvar(P,K,M):
    #Paddla (K-1) tag med varje paddel
    M -= (K-1)*P

    #Har vi redan kommit fram?
    if M<=0:
        return P

    #Kan vi komma fram genom att
    #utnyttja sista paddeltaget på
    #varje paddel?
    if M<=P:
        return P-M

    #Annars omöjligt
    return -1
```

Lösning 1: Binärsökning

Ifall vi gissar ett visst antal paddlar att börja med, så kan vi sedan använda funktionen `paddlar_kvar` för att simulera hela resan från första ön, och se hur långt vi kommer. En första tanken kan vara att testa simulera alla olika antal paddlar, från 0 och uppåt, tills vi kommer fram hela vägen till sista ön. Problemet är att, som man kan se i Körningsexempel 3, att svaret kan bli mycket stort, och vi kommer då inte hinna inom tidsgränsen.

Så istället kan vi använda följande insikt: Om simulerar med ett visst antal paddlar att börja med och vi inte kommer fram så vet vi att vi måste börja med fler paddlar. Om vi däremot kommer fram så skulle vi kunna testa att börja med färre paddlar istället. Alltså kan vi "binärsöka" efter svaret. Att binärsöka innebär att vi håller koll på det minsta talet som vi vet är tillräckligt många paddlar (kalla det för R) och det största talet som vi vet är inte tillräckligt (kalla det för L). Sedan gissar vi att vi börjar med ett antal paddlar P som är mitt mellan L och R. Vi simulerar, och om P visar sig vara tillräckligt så uppdaterar sätter vi R=P, och om P inte var tillräckligt sätter vi L=P. På så sätt halverar

vi vårt intervall av möjliga värden. Vi upprepar detta tills R-L=1. Då vet vi att det är möjligt med R paddlar, men inte möjligt med R-1=L paddlar. Alltså är R det minsta antalet paddlar som behövs, och vi kan svara med det.

```
def paddlar_kvar(P,K,M):
    M -= (K-1)*P
    if M<=0:
        return P
    if M<=P:
        return P-M
    return -1

N = int(input())
K = int(input())
H = int(input())

D = []
for i in range(N-1):
    D.append(H*int(input()))


#Vi vet att det inte går med 0 paddlar
l=0
#Detta tal är tillräckligt stort så att det alltid kommer gå
r=10**18

while r-l!=1:
    #Gissa mitt mellan r och l
    start_paddlar = (r+l)//2

    P = start_paddlar
    #Simulera
    for i in range(N-1):
        P = paddlar_kvar(P, K, D[i])
        if P<0:
            break

    #Ifall vi har negativt antal paddlar så
    #har vi inte kommit hela vägen fram

    if P<0:
        l = start_paddlar
    else:
        r = start_paddlar

print(r)
```

Lösning 2: Räkna baklänges

Det finns också en annan lösning, som kan vara lättare att koda men kanske svårare att komma på. Ifall vi vänder på problemet, och ställer oss frågan: "Om vi vill ha kvar P' paddlar när vi kommer till ö X, vad är det minsta antalet paddlar P vi behöver ha på ö X-1?". Genom att utgå från `paddlar_kvar` ovan går det att komma fram till formeln P=max(P', ⌈(D[X]+P')/K⌉) där ⌈x⌉ betyder x avrundat uppåt till närmsta heltal. Om man börjar på sista ön med P'=0, så kan man gå baklänges genom alla öar, och med hjälp av formeln hela tiden räkna ut vad det minsta antalet paddlar man behöver är:

```
P = 0
for i in range(N-2,-1,-1):
    P = max(P, math.ceil((D[i]+P)/K))
print(P)
```

Klockan

n <= 23

För testfallet där n <= 23 kan tjuven inte ha varit inne i huset i mer än en sekund (även om klockan skulle vara 11:11:11 två gånger i rad krävs minst 24 energienheter för att klockan ska vara igång i två sekunder). Vi kan gå igenom varje sekund under dygnet och räkna hur många av dem som drar exakt n energienheter.

n <= 200

För testfallet med n <= 200 kan vi gå igenom varje sekund under dygnet och testa om klockan kan ha startat på den sekunden. Som exempel kan vi säga att n=166 och vi vill kolla om klockan kan ha startat 20:00:00. Vi börjar med att räkna ut att klockan drar 35 energienheter klockan 20:00:00. Det är för lite, så vi adderar på nästa sekund, 20:00:01, och får totalt 66 energienheter. Det är fortfarande för lite, så vi fortsätter med att addera på fler sekunder. När vi kommit fram till 20:00:04 är summan uppe i 167, vilket är mer än 166, så vi kan dra slutsatsen att klockan inte kan ha startat 20:00:00.

Sedan testar vi om 20:00:01 är en giltig starttid. Vi adderar på fler sekunder på samma sätt som innan tills vi kommit fram till 20:00:05. Då är summan precis 166 och vi kan dra slutsatsen att tjuven kan ha brutit sig in 20:00:01. Vi testar på samma sätt alla starttider mellan 00:00:00 och 23:59:59 och räknar hur många som funkar.

Alla testfall

För att lösa alla testfall behöver vi göra lösningen snabbare. Det kan vi göra genom att inse att när vi går vidare till att kolla om 20:00:01 funkar som starttid kan vi behålla summan som vi räknade ut mellan 20:00:00 och 20:00:04, men subtrahera bort energin som 20:00:00 drar, för att direkt få summan mellan 20:00:01 och 20:00:04. Vi ser då att summan är för liten och adderar på 20:00:05. Då blir summan precis 166 och vi noterar att 20:00:01 är en giltig starttid. Sedan kan vi gå vidare till att testa om 20:00:02 är en giltig starttid genom att subtrahera bort 20:00:01 och fortsätta likadant.

Den här lösningen använder två pekare (two-pointer technique) eftersom man kan säga att vi har två pekare som pekar ut starten och slutet av tidsintervallet som vi har summan för. Om intervallet drar för mycket energi stegar vi fram startpekaren en sekund (och minskar summan), och om det drar för lite energi stegar vi fram slutpekaren (och ökar summan).

```
# digit innehåller hur mycket energi varje siffra drar per sekund. Exempelvis är digit[7]=3
digit = [6, 2, 5, 5, 4, 5, 6, 3, 7, 6]

# beräknar hur mycket energi som används för att visa ett visst klockslag, t sekunder efter
```

```
def energy(t):

    # timtalet, minuttalet och sekundtalet som visas på klockans display.
    seconds = t%60
    minutes = t//60%60
    hours = t//3600

    # för varje tal visas tiotalet som en siffra och entalet som en siffra.
    return(digit[hours // 10] + digit[hours % 10] +
           digit[minutes//10] + digit[minutes%10] +
           digit[seconds//10] + digit[seconds%10])



n = int(input())

ans = 0 #antalet möjliga starttider som hittats

i = 0 # intervallets starttid (mätt i sekunder efter 00:00:00)
j = 0 # intervallets sluttid (också mätt i sekunder efter 00:00:00)
s = energy(0) # summan av hur mycket energi som går åt under tidsintervallet mellan i och j

while(j < 24*60*60):
    if(s == n):
        ans+=1

    # om summan är för liten stegar vi fram intervallets slut
    if(s < n):
        j += 1
        s += energy(j)
    # annars stegar vi fram intervallets start
    else:
        s -= energy(i)
        i += 1

print(ans)
```

Kodlås

För delpoängen där det gäller att N,M <= 5 räcker det med att prova alla N^M möjligheter, för en lösning som kör i O(N^M * NM) tid.

För nästa delpoäng (där det finns maximalt tre hål i varje skiva) kan man göra en liknande lösning, fast man kan undvika de flesta möjligheter ("pruning"). Idén är att om efter man har fixerat de första t raderna inser att inga kolumner kan vara öppna finns det ingen poäng med att prova alla resterande möjligheter för de sista N-t raderna.

För full poäng behöver man göra något lite smartare. Vi presenterar här två olika full-poängs-lösningar.

Dynamisk programmering

Vi kan tänka oss att vi gör en rekursiv lösning där vi går igenom raderna i ordning och provar varje inställning av skivorna. Vi kan också hålla reda på vilken delmängd C av kolumnerna som har hål i sig i all skivor vi ställt in än så länge.

Så vår funktion ser ut ungefär:

```
# f(r, C) beräknar antalet sätt att ställa in skivorna r, r+1, ... N
# så att minst en kolumn är öppen, givet att skivorna 1,2,...,r-1 är
# inställda på ett sådant sätt att exakt kolumnerna i C är öppna.
f(r, C):
    ans = 0
    För varje inställning av skivan på rad r:
        räkna ut vad nya mängden öppna columner K är
        ans += f(r+1, K)
    return ans
```

Denna funktion går utmärkt att använda dynamisk programmering (memoization) på. Istället för att räkna ut f(r,C) flera gånger för samma r och C så kan man spara svaren. Totalt finns det maximalt N * 2^M olika inputs vi kommer anropa functionen med (det finns exakt 2^M delmängder av {1,2,...,M}), så vi behöver bara köra for-loopen i funktionen O(N 2^M) gånger.

Totala tidskomplexiteten blir O(N * M * 2^M).

Alternativ lösning: Inklusion-Exklusion.

Betrakta följande (felaktiga) lösningsidé:

> För varje kolumn, beräkna hur många inställningar av skivorna som leder till att denna kolumn är öppen (kan enkelt göras med att räkna ut detta för varje skiva för sig och multiplicera talen). Sedan summerar vi svaren för de olika kolumnerna.

Det som går fel är att vi kan råka dubbelräkna inställningar av skivorna som leder till att minst två kolumner är öppna. För att fixa detta kan man tänka sig att man provar alla par av kolumner, och räknar ut hur många gånger som båda dessa kolumner är öppna (igen, detta är bara lösa för skivorna separat och multiplicera), och subtraherar det från svaret. Tyvärr fungerar inte detta heller nu har man råkat räkna bort de inställningar där minst tre olika kolumner är öppna för många gånger. Detta kan igen fixas med att man nu för varje tre-tupel av kolumner räknar ut svaret, men då har vi dubbelräknat fyr-tupler osv osv.

Med Principen om Inklusion/Exklusion kan detta formaliserats. För en delmängd kolumner C, låt f(C) beteckna antalet inställningar av skivorna som gör att alla kolumner i C bara har hål i sig. För ett givet C kan man beräkna f(C) i O(NM) tid (för varje skiva räkna ut antalet sätt som den kan ställas in på så att det är hål i kolumnerna C, och multiplicera dessa tal).

Inklusions/Exklusions-principen säger nu att svaret är summan av odd(C) * f(C) över alla 2^M delmängder C; där odd(C) = 1 om C har ett udda antal kolumner, och -1 annars.

Totala tidskomplexiteten blir O(N * M * 2^M).

Implementationstips

I båda lösningarna vill man kunna behandla delmängder av kolumnerna. En sådan delmängd C kan representeras av en bitmask (tänk en sträng av 0or och 1or) där den t:e siffran är 1 om och endast om kolumn t finns i C. Då M är ganska litet här kan man representera en bitmask med en integer i de flesta programmeringsspråk, genom att tänka sig integern i binära talsystemet. Då kan man använda bitwise-operatorer så som `&`, `<<` etc för att förenkla vissa operationer.

```python
# Dynamisk programmering i python
n, m = [int(i) for i in input().split()]

def dp(i, op):
    if i == n:
        return 1 if op > 0 else 0
    if mem[i][op] != -1:
        return mem[i][op]
    res = 0
    for j in codes[i]:
        res += dp(i+1, op&j)
    mem[i][op] = res
    return res

mem = [[-1 for _ in range(1<<m)] for _ in range(n)]
codes = []
for i in range(n):
```

```
    rotation = [0 for i in range(m)]
    cur = input()
    for i in range(m):
        for j in range(m):
            rotation[j]+=((cur[i] == '.')<<((i+j)%m))
    codes.append(rotation)

print(dp(0, (1<<m)-1))
```

```
// Inklusion/Exklusion i c++
int main() {
  int n, m;
  cin >> n >> m;
  std::vector<int> v(n);
  for(int i = 0; i < n; ++i) {
    std::string s;
    cin>>s;
    for(int j = 0; i < m; ++i)
      if(s[j] == '.')
        v[i] ^= (1<<j);
    v[i] = v[i] + (v[i]<<m);
  }

  long long ans = 0;
  for(int msk = 0; msk < (1<<m); ++msk)
    long long ans_for_C = 1;
    for(int i = 0; i < n; ++i) {
      long long disk = 0;
      for(int j = 0; i < m; ++i)
        if(((v[i]>>j)&msk) == msk)
          ++disk;
      ans_for_C *= cnt;
    }
    long long odd_C = (__builtin_popcount(msk)%2 ? 1 : -1)
    ans += odd_C * ans_for_C;
  }
  std::cout << ans << std::endl;
}
```