**Knapsack**

1.

a) Considerăm o matrice dp[k][n] pentru a forma sume parțiale, unde dp[i][j] reprezintă suma maximă <= j pe care o pot forma cu primele i elemente.

La pasul dp[i][j], vom decide dacă adăugăm obiectul s[i-1] la suma parțială curentă sau nu:

1. Dacă nu îl adăugăm, dp[i][j] va fi egal cu dp[i-1][j] (suma maximă mai mică sau egală cu j fără elementul i).
2. Dacă îl adăugăm (și nu depășim j), dp[i][j] va fi egal cu dp[i-1][j-s[i-1]] + s[i-1]. Alegem suma parțială maximă mai mică sau egală cu j-s[i-1] din sumele fără elementul i.

Complexitatea de timp a algoritmului este O(nk), unde n reprezintă numărul de elemente și k reprezintă suma maximă pe care vrem să o obținem. Algoritmul este pseudopolinomial deoarece depinde de k. Complexitatea de spațiu a algoritmului este de O(nk), deoarece folosim o matrice (n+1) x (K+1) pentru a stoca rezultatele intermediare, dar poate fi redus la O(2k) dacă păstrăm doar ultimele două rânduri ale matricei.

n = int(input())

K = int(input())

S = []

for i in range(n):

S.append(int(input()))

dp = [[0 for \_ in range (K+1)] for \_ in range(n+1)]

for i in range(n):

for suma in range(1, K+1):

if dp[i-1][suma-S[i-1]] + S[i-1] <= suma:

dp[i][suma] = max(dp[i-1][suma], dp[i-1][suma-S[i-1]] + S[i-1])

else:

dp[i][suma] = dp[i-1][suma]

print(dp[n-1][K])

b) Scopul algoritmului este de a calcula o suma cât mai apropiată de valoarea k prin parcurgerea unui șir de elemente și adunarea numărului curent la sumă în cazul în care se poate aduna fără depășirea valorii k. În timpul parcurgerii, algoritmul determină elementul cu valoare maximă din șir și în cazul în care este mai mare decât suma calculată până la momentul respectiv, schimbă suma cu acel element, cu scopul de a obține o soluție 1/2-aproximativă.

Știind că elementul curent din șir este mai mic decât k, pentru fiecare element se disting două situații:

1. Dacă valoarea elementului maxim din șir este mai mare sau egal decât jumătatea lui k, înseamnă că soluția curentă este deja mai bună decât jumătatea algoritmului optim, deci soluția este 1/2-aproximativă

2. Dacă valoarea elementului maxim din șir este mai mică decât jumătatea lui k, rezultă că toate elementele din șir sunt mai mici decât jumătatea lui k, deci prin parcurgerea întregului șir se va obține o sumă mai mare decât k/2, astfel algoritmul fiind 1/2-aproximativ

if \_\_name\_\_ == '\_\_main\_\_':

n = int(input())

k = int(input())

suma = 0

maxim = -1

while n > 0:

x = int(input())

if x + suma <= k:

suma += x

if maxim < x <= k:

maxim = x

n -= 1

print(max(suma, maxim))

**Load Balance**

2.

a) Presupunem prin reducere la absurd că nu există niciun I astfel încât ALG2(I) ≥ 2 ALG1(I). Altfel spus ALG2(I) < 2 ALG1(I) pentru orice I (1)

Știm că ALG1(I) ≤ 2 OPT(I) pentru orice I => 2 ALG1(I) ≤ 4 OPT(I) pentru orice I (2)

Din (1) și (2) => ALG2(I) < 2 ALG1(I) ≤ 4 OPT(I) pentru orice I => ALG2(I) < 4 OPT(I) pentru orice I (3)

Caz 1: Considerăm 2 și 4 tight bounds pentru ALG1 respectiv ALG2.

Deci există cu siguranță un I astfel încât ALG2(I) = 4 OPT(I) (4)

Din (3) și (4) rezultă contradicție, afirmația făcută este falsă, deci există cu siguranță un I astfel încât ALG2(I) ≥ 2 ALG1(I), deci afirmația din enunțul problemei este adevărată.

Caz 2: Considerăm 2 și 4 nu sunt tight bounds pentru ALG1 respectiv ALG2.

În acest caz nu putem zice cu siguranță că există un I pentru care ALG2(I) ≥ 2 ALG1(I), deci afirmația din enunț e falsă.

b) Presupunem prin reducere la absurd că există I astfel încât ALG1(I) ≥ 2 ALG2(I) (1)

ALG1 e 2-aproximativ => ALG1(I) ≤ 2 OPT(I) (2)

Din (1) și (2) => 2 OPT(I) ≥ ALG1(I) ≥ 2 ALG2(I) (3)

Știm că OPT(I) ≤ ALG2(I) => 2 OPT(I) ≤ 2 ALG2(I) (4)

Din (3) și (4) rezultă că 2 ALG2(I) = 2 OPT(I) = ALG1(I) <=> ALG2(I) = OPT(I) = ALG1(I) <=> ALG2(I) = OPT(I) si ALG1(I) = OPT(I)

Cele două condiții pot fi îndeplinite => presupunerea făcută este adevărata => afirmația că nu există niciun I care sa îndeplinească condiția ALG1(I) ≥ 2 ALG2(I) este falsă

3. Considerăm următoarele notații:

k = indicele mașinii cu load-ul maxim în urma rulării algoritmului

q = ultimul job asignat mașinii k

load’(M) = load-ul mașinii M după asignarea primelor q-1 job-uri, dar nu și job-ul q

ALG = load(k) = load’(k) + tq (1)

load’(k) ≤ ≤ (2)

Din (1) si (2) => load’(k) + tq ≤ + tq = + tq = + tq ≤ + (tm + tm+1) ≤ OPT + OPT = OPT = ( - ) OPT

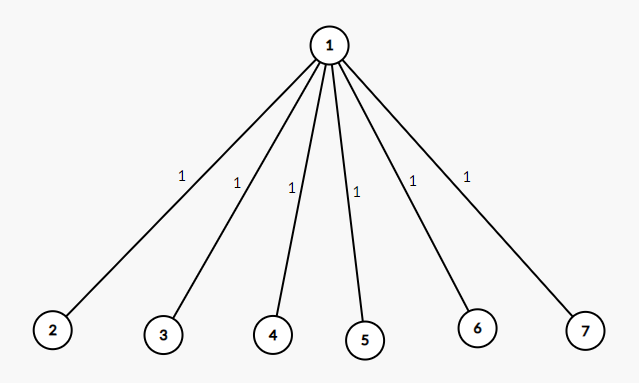
**Travelling Salesman Problem**

1.

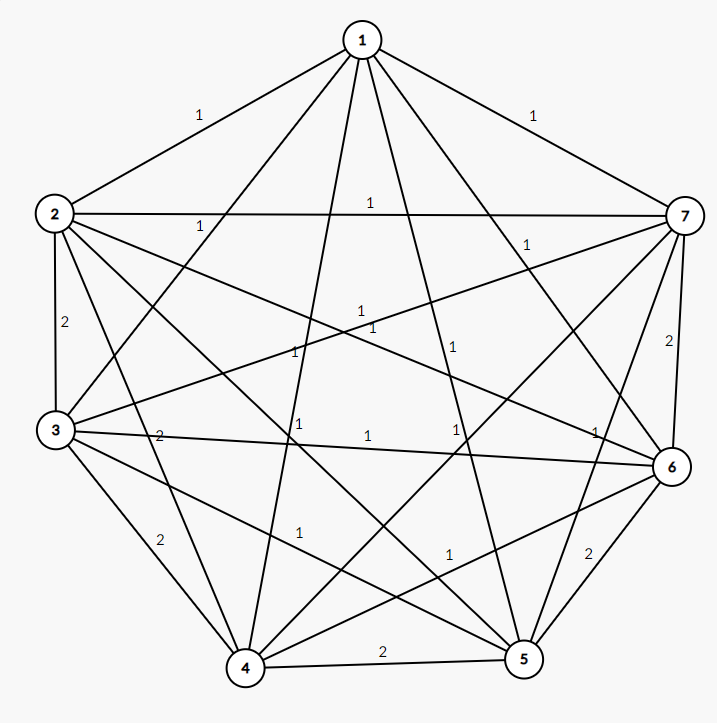
a) Presupunem prin reducere la absurd că problema nu rămâne NP-hard, deci admite un algoritm polinomial. Consideram un graf G neorientat și neponderat și construim un graf G’ pe baza lui G astfel încât ponderile muchilor să fie egale cu 1 daca exista în G sau egale cu 2 altfel (G’ respecta condițiile din cerința).

Algoritmul nostru va genera pentru G’ o soluție egală cu n (numărul de noduri) dacă graful G este hamiltonian, deoarece costul total minim se obține traversând chiar muchiile din G având fiecare costul 1. Daca G nu este hamiltonian, soluția data de algoritm ar fi cel puțin egala cu n + 1 (cazul cel mai bun e când traseul conține n – 1 muchii egale cu 1 și încă una egala cu 2). Deci, ar însemnă ca cu acest algoritm putem determina în funcție de rezultat (=n sau ≥ n+1) daca graful parțial G este sau nu hamiltonian. Dar știm ca problema determinării unui ciclu hamiltonian într-un graf neorientat și neponderat este NP-completă, deci și NP-hard (proprietate adevărată și pentru un graf ponderat cu toate ponderile egale cu 1). Deci avem contradicție cu presupunerea inițială, rezultă ca problema rămâne NP-hard.

b) Știm ca inegalitatea triunghiului este L2 + L3 ≥ L1 cu L1 ≥ L2 ≥ L3. Verificăm toate cazurile posibile pentru graful din problemă:

1. Dacă L1 = L2 = L3 = 1 avem L2 + L3 = 2 ≥ 1 = L1 => se respecta inegalitatea
2. Dacă L2 = L3 = 1 si L1 = 2 avem L2 + L3 = 2 ≥ 2 = L1 => se respecta inegalitatea
3. ****Dacă L3 = 1 si L1 = L2 = 2 avem L2 + L3 = 3 ≥ 2 = L1 => se respecta inegalitatea
4. Dacă L1 = L2 = L3 = 2 avem L2 + L3 = 4 ≥ 2 = L1 => se respecta inegalitatea

Rezultă că se respecta inegalitatea triunghiului pentru graful din problemă.

****c) Presupunem prin reducere la absurd că algoritmul descris este – aproximativ, deci ALG(I) ≤ OPT(I) pentru oricare I – intrare. Pentru a arata contrariul este suficient să găsim un exemplu în care nu se respectă presupunerea făcută.

De exemplu, pentru un graf complet cu n=7 noduri, muchiile de cost 1 sau 2 și nodul de plecare 1, algoritmul va găsi un ciclu hamiltonian 1-2-3-4-5-6-7-1 cu ALG = 1+2+2+2+2+2+1 = 12, dar un ciclul hamiltonian de cost minim (1-3-7-5-2-6-4-1) are costul OPT = 1+1+1+1+1+1+1 = 7. Deci \* 7 = 10.5 ≤ 12 = ALG, rezultă contradicție cu presupunerea făcută, deci algoritmul descris nu este unul – aproximativ.

**Vertex Cover**

1. Worst case este când algoritmul selectează la fiecare pas o variabilă care să nu se

regăsească decât în clauza din care a fost selectată și nu printre alte clauze. Deci la fiecare pas va fi eliminată doar clauza care e selectată, deci doar câte o singură clauză la fiecare iterație. Se ajunge la ALG = m, deci algoritmul este m-aproximativ.

1. Un algoritm 3-aproximativ pentru problema inițială este:

1. Fie 𝐶 = {𝐶1,…,𝐶𝑚} mulțimea de predicate, 𝑋 = { 𝑥1,…, 𝑥𝑛 } mulțimea de variabile.

2. Cât timp 𝐶 ≠ ∅ execută:

(a) Alegem aleator 𝐶𝑗 ∈ 𝐶.

(b) 𝑥𝑖 ← 𝚝𝚛𝚞𝚎 pentru cele 3 variabile din Cj (i=1-3)

(d) Eliminăm din 𝐶 toate predicatele care conțin pe cel puțin o variabilă 𝑥𝑖 (i=1-3)

3. Soluția constă din variabilele pe care le-am setat ca true pe parcursul

execuției algoritmului

Dem:

Fie C’ ⊂ C mulțimea de clauze cu variabile distincte două câte două. Se poate observa că e nevoie de cate o variabilă din soluție setată cu true pentru fiecare clauză din C’. Fie S o soluție => |S| ≥ |C'| <=> OPT ≥ |C'|

Clauzele selectate la pasul 2a aparțin mulțimii C’ deoarece când selectez o clauză elimin și clauzele care conțin măcar una din cele trei variabile din clauza selectată, deci nu vor fi incluse în soluție. Fie mulțimea acestor clauze S’ => OPT ≥ |S'| => 3 OPT ≥ 3 |S'| = |S| => algoritmul este 3-aproximativ

1. Să considerăm problema dată ca o problemă liniară cu următoarele specificații:

Fie X = {x1, ..., xn} și C = {C1, ..., Cm}, unde:

1. Fiecare variabilă xi, pentru i ∈ {1, ..., n}, este restricționată astfel încât 0 ≤ xi ≤ 1.
2. Pentru fiecare clauză Ci din mulțimea C și pentru fiecare triplet de variabile xi1, xi2, xi3 ∈ Ci, condiția impusă este xi1 + xi2 + xi3 ≥ 1.

Fie f:X->N, f(x) = .

Dorim să determinăm costul minim pentru care putem evalua expresia ca fiind True, adică să minimizăm suma .

1. Consider ca o variabilă are valoarea true daca xi ≥ .

ALG = ≤ ≤ ≤ 3 OPT => algoritmul e 3-aproximativ