Lucrare de licență

Metode de optimizare a problemelor de programare dinamică

Laiu Florin Emanuel

Conf. dr. Traian Șerbănuță

Iunie, 2021

Facultatea de Matematică și Informatică

Specializarea Informatică

Universitatea București

Metode de optimizare

pentru probleme

de programare dinamică

Laiu Florin

Rezumat

În această lucrare vom prezenta metode de optimizare pentru probleme de programare dinamică bazându-ne pe cele mai recente idei întâlnite în concursurile de programare. În programarea competitivă dar și în multe aplicații practice este necesară calcularea unor relații de recurență în timp rapid. Pentru a face acest lucru, trebuie să ne folosim de proprietățile specifice fiecărei probleme abordate.

Abstract

In this paper we will present optimization methods for dynamic programming problems, based on ideas that have appeared in competitive programming contests. In competitive programming and in many practical applications one has to quickly calculate a recurrence relation. To do this, one has to take advantage of each problem's specific properties.

Cuprins

I Introducere

I.1 Programarea competitivă

Concursurile de programare necesită elaborarea unor algoritmi eficienți pentru un set de probleme dat și implementarea acestor algoritmi într-un limbaj de programare. Majoritatea problemelor sunt de tip „Input-Output„ - se dau date de intrare, ce corespund parametrilor problemei și se cer date de ieșire - răspunsul la problemă bazat pe datele de intrare. Autorul unei probleme generează mai întâi mai multe date de intrare valide și aplică soluția sa pe acestea, obținând datele de ieșire corespunzătoare.

O pereche de date de intrare și ieșire generate de către autor se numește **test** și sunt generate de obicei între 20 și 200 de teste care **nu** sunt publice concurenților. Bazat pe aceste teste, evaluarea soluției participanților se va face automat de către calculator comparând răspunsul corect al fiecărui test cu răspunsul dat de participant și verificând timpul de execuție per test, care trebuie să se încadreze într-o limită fixată per problemă. Se va obține astfel o evaluare obiectivă a soluțiilor concurenților ce este bazată pe eficiența și corectitudinea implementărilor acestora.

În timpul concursului, concurenților li se va oferi un feedback asupra soluțiilor trimise. Acesta poate fi, după caz:

- nul - Când soluția va fi evaluată doar o dată, după terminarea concursului.
- parțial - Se va alege un procent mic de teste din cel total și se va comunica participantului dacă soluția lui este corectă pentru toate aceste teste sau nu.
- complet - Se va evalua sursa concurentului pe toate testele unei probleme și i se va comunica scorul obținut.

Un exemplu de concurs ce nu oferă feedback este Olimpiada de Informatica din România. În cadrul fazei naționale, concurenții au de rezolvat și implementat 3 probleme în 5 ore. Fiecare problemă valorează 100 de puncte. Sursele acestora vor fi preluate și evaluate (automat) abia

după terminarea concursului. Așadar, dacă deși ideea algoritmică de rezolvare este corectă dar codul concurentului are o mică eroare, este posibil ca acesta să obțină punctajul 0 pentru o problemă, neavând opțiunea de a retrimite codul modificat. Participanții experimentați din cadrul olimpiadei își implementează mai întâi, pe lângă soluția eficientă și o soluție lentă, brute-force de care sunt siguri că este corectă. Apoi, ei își generează folosind un alt program mai multe date de intrare și datele de ieșire corespunzătoare folosind soluția sigură brute-force. În final, pe acest set de date de intrare generat de ei vor compara rezultatul soluției eficiente cu rezultatul soluției brute-force.

Un exemplu de site concursuri cunoscut ce oferă feedback parțial este Codeforces. Concursurile de pe Codeforces necesită rezolvarea a 5 probleme de dificultate variată în 2 ore. Scorul problemelor este însă variat: de regulă, prima problemă are scorul 500, a doua 1000, a treia 1500, a patra 2000 iar a cincea are scorul 2500. Când un concurent trimite codul său spre evaluare, acesta va ști imediat pentru un număr de teste (20, 30 din 200 totale) dacă soluția sa a fost corectă. După terminarea concursului, se vor da rezultatele finale luând în considerare toate testele pentru fiecare problemă. Ceea ce este diferit față de concursurile de tip olimpiadă este că implementarea unei probleme trebuie să dea rezultatul corect pentru **toate** testele. Dacă din 200 de teste, implementarea este corectă pe 199 de teste, dar dă răspuns greșit pentru un caz particular simplu - cum ar fi $n = 0$ pentru multe probleme, atunci scorul obținut va fi 0. În plus, dacă un concurent trimite o implementare greșită (raportat la setul de teste parțial din timpul concursului), acesta va primi o penalizare de scor. Scorurile prezentate la început pentru cele 5 probleme vor scădea pe măsură ce trece timpul rămas în concurs. O tactică folosită de cei mai buni concurenți este să înceapă mai întâi cu problemele mai grele pentru că acestea au un scor foarte mare la început.

Feedbackul complet este oferit la concursuri precum ACM-ICPC, AtCoder, CsAcademy, Olimpiada Internațională de Informatică, Olimpiada Balcanică de Informatică și altele. ACM-ICPC este cel mai cunoscut concurs internațional pentru echipe în care fiecare echipă este formată din 3 membrii ai aceleiași universități. Echipele au de rezolvat între 8 și 12 probleme în 5 ore. Fiecare problemă are scorul egal cu 100 de puncte, feedback complet oferit pe loc dar necesită ca soluția echipelor să fie corectă pe toate testele, deci nu se oferă punctaje parțiale.

Concursurile menționate anterior de tip olimpiadă ce au feedback complet împart fiecare problemă în subtaskuri. Un subtask are cerința problemei inițiale, dar datele de intrare de dimensiune redusă pentru a permite încadrarea în limita de timp soluțiilor mai puțin eficiente. De obicei, implementarea corectă, rapidă a unor soluțiile ușoare, cunoscute (în raport cu dificultatea acestor concursuri) permite obținerea a cel mult 50 de puncte din 100 pentru o problemă. În cadrul acestor concursuri, nu există penalizare pentru trimiterea mai multor implementări și pentru fiecare problemă pot fi efectuate cel mult 50 submisii de către fiecare concurent.

În fiecare tip de concurs menționat anterior, pe lângă abilitatea de a implementa corect, rapid o soluție algoritmică într-un limbaj de programare, diferența între premii este făcută de rezolvarea eficientă a celor mai grele probleme. De multe ori, acestea necesită cunoașterea unor structuri de date sau algoritmi avansați. O tehnică utilizată des pentru aceste probleme este **programarea dinamică**.

I.2 Programare dinamică

Programarea dinamică rezolvă problemele prin descompunerea lor în subprobleme şi prin combinarea rezolvărilor acestora și este una dintre cele mai folosite tehnici în concursuri, dar și în cercetare. Câteva probleme cunoscute de programare dinamică sunt: problema rucsacului, cel mai lung subșir crescător, determinarea drumului de cost minim într-un graf.

Multe dintre recurențele care trebuie calculate la probleme de programare dinamică au complexitatea de timp $O(N^2)$, unde N este dimensiunea inputului - care este de obicei un vector sau un graf. În general, la concursuri, se pot efectua 10^9 calcule într-un program in timp de 1 sau 2 secunde, iar cele mai multe probleme au limita de timp maxim de 2 secunde, în unele cazuri ajungând la 4-5 secunde. Pentru $N = 10^5$, calcularea acestor recurențe va efectua deja 10^{10} operații la care se adaugă alte operații constante care nu sunt luate în calcularea complexității, dar și de accesul memoriei cache. Astfel, timpul necesar de execuție al unui program pentru a efectua calculul unei recurențe în $O(N^2)$ pentru $N = 10^5$ va fi de minim $10 - 20$ secunde pentru $N = 10^5$. Mărind N la 10^6, timpul de execuție va crește și mai mult și va fi între 1000 și 2000 de secunde. Soluțiile de complexitate $O(N^2)$ sunt astfel foarte ineficiente când N este mare.

Pentru aceste limite mari de $N \geq 10^5$ avem nevoie de soluții de complexitate mult mai bună, $O(N \log_2^2 N)$, $O(N \log_2 N)$ sau chiar $O(N)$. Însă, pe cazul general, nu toate probleme pot fi optimizate pentru a reduce complexitatea de timp. Trebuie astfel să ne folosim de particularitățile fiecărei probleme pentru a optimiza complexitatea de timp.

În ultimii ani, au apărut în programarea competitivă câteva astfel de tehnici ce reduc complexitatea de la $O(N^2)$ la complexități mult mai bune menționate anterior pentru anumite clase de probleme ce îndeplinesc anumite proprietăți. O problemă care folosește o tehnică de optimizare interesantă este „Aliens„ de la Olimpiada Internațională de Informatică 2016, cel mai prestigios concurs de informatică pentru liceu.

În anul 2016, singurul concurent care a reușit să găsească și să implementeze metoda de optimizare a problemei, obținând 100 de puncte a fost chiar câștigătorul concursului, Ce Jin din China și a avut scorul total 597/600. Restul participanților au obținut cel mult 60 de puncte pe această problemă. Dacă acesta ar fi implementat doar soluția de 60 de puncte, ar fi fost la egalitate cu concurentul de pe locul 3 cu 547/600, dar ar fi fost depășit la doar 3 puncte de concurentul de pe locul 2 care a terminat cu 550 de puncte. Diferența de scor a fost astfel foarte strânsă și premiul cel mare a fost decis astfel de această problemă.

Un alt exemplu este Olimpiada Balcanică de Informatică din 2019, problema „Tennis„ cu următoarea cerință: „dându-se un arbore cu N noduri ($N \leq 10^6$) cu costuri pe muchii, se cere găsirea costului maxim a unui cuplaj de dimensiune $K (K \leq N)$„. Concurenta Alexa Tudose din România a reușit să rezolve problema de 100 de puncte, devenind astfel prima fată din istoria Olimpiadei Balcanice de Informatică care obține locul I. Soluția autorilor concursurilor se folosește de proprietățile cuplajului de sumă maximă și de tehnica folosită la problema „Aliens„ obținând o soluție mai bună în poate găsi in $O(N \log_2 MAX)$, unde MAX este valoarea maximă a unei muchii. Se obține astfel o soluție mult mai rapidă decât cea în $O(N^2)$ având în vedere $MAX \leq 10^9$.

II Context

II.1 Concepte de programare liniară

Toate definițiile și rezultatele din această secțiune sunt standard și pot fi găsite, de exemplu, în [17], [7].

O problemă de programare liniară este o problemă care necesită maximizarea sau minimizarea unei funcţii liniare ce are o expresie liniară în prezenţa unor restricţii liniare. Formularea generală a problemelor de programare liniară în numere întregi este definită astfel:

$$\begin{aligned} \text{maximize} \quad & f(x) = c_1x_1 + \ldots + \quad c_nx_n \\ \text{subject to} \quad & a_{11}x_1 + \ldots + \ a_{1n}x_n \leq b_1 \\ & \qquad\qquad \vdots \\ & a_{m1}x_1 + \ldots + a_{mn}x_n \leq b_m \\ & x_i \geq 0 \end{aligned}$$

Definiția II.1. Orice vector care îndeplinește toate constrângerile unui program liniar se numește **soluție fezabilă**.

Definiția II.2. O *soluție fezabilă* care maximizează funcția obiectiv se numește **soluție optimă** a programului P.

Observație. *Fezabilitatea* unei soluții depinde doar de constrângeri, dar *optimalitatea* unei soluții depinde și de constrângeri și de funcția de cost.

Definiția II.3. Regiunea fezabilă a unui program liniar (P) este definită ca fiind mulțimea tuturor *soluțiilor fezabile*.

Definiția II.4. Problema de programare liniară se numește **fezabilă** dacă *regiunea fezabilă* este nevidă.

Definiția II.5. O **problemă fezabilă** de maxim (respectiv minim) este **nemărginită** dacă funcția obiectiv poate lua valori oricât de mari (respectiv oricât de mici) pentru vectorii fezabili.

Definiția II.6. Un punct x se numește **punct de extrem** al unei mulțimi convexe P dacă nu se poate scrie ca și o combinație liniară formată din alte două puncte din P. Matematic, nu există $y, z \in P, y \neq x, z \neq x$ și $\lambda \in [0, 1]$ astfel încât:

$$x = \lambda y + (1 - \lambda)z$$

Teorema II.7. Fie P un program liniar scris în forma următoare, numită **formă canonică**:

$$\begin{aligned} &\text{maximize} \quad f(x) = c^T x \\ &\text{subject to} \quad Ax \leq b \end{aligned} \tag{II.1}$$

Presupunem că mulțimea $M = \{x \mid Ax \leq b\}$ are cel puțin un punct de extrem. Atunci, dacă există o soluție optimă pentru programul P, există și o soluție optimă care este punct de extrem al mulțimii M. ([1]).

Teorema de mai sus ne spune practic că dacă dorim să găsim soluția optimă pentru un program liniar, este suficient să verificăm doar punctele de extrem.

Teorema II.8. Fie $A \in R^{m\ x\ n}$, $c \in R^n$ *fixate*. Definim $g\colon R^m \to R$, $g(b) = \max\limits_{x|Ax \leq b} c^T \cdot x$. Funcția b este *concavă*.

Demonstraţie. Fie b_1, b_2 și $0 \leq t \leq 1$. Luăm x_1, x_2 astfel încât $A \cdot x_j \leq b_j,\ \forall j\ \in \{1, 2\}$, $c^T \cdot x_j = g(b_j)$ și $x = t \cdot x_1 + (1 - t) \cdot x_2$. Atunci avem că $A \cdot x \leq t \cdot b_1 + (1 - t) \cdot b_2$, deci

$$\begin{aligned} c^T \cdot x &= t \cdot f(x_1) + (1 - t) \cdot f(x_2) \\ &= t \cdot g(b_1) + (1 - t) \cdot g(b_2) \leq g(t \cdot b_1 + (1 - t) \cdot b_2) \end{aligned}$$ □

III Metoda de căutare parametrică

Descriere. Metoda căutării parametrice este o metodă de optimizare care a devenit cunoscută în programarea competitivă în anul 2016, odată cu apariția problemei *"Alien"* de la Olimpiada Internațională de Informatică. Această metodă se folosește în general pentru a reduce complexitatea de la $O(NK)$ la $O(Nlog_2V)$ la unele probleme de programare dinamică. Vom ilustra această metodă de optimizare rezolvând o variație a unei probleme întâlnite în bioinformatică.

Istoric. „În analiza secvențelor biomoleculare, sunt de interes găsirea segmentelor de interes biologic, e.g regiuni GC-rich, gene non-coding RNA genes, segmente transmembranare, etc. Probleme algoritmice care apar o dată cu analiza acestor secvențe sunt de a localiza subsecvențe cu scor mare (dându-se o funcție de scor potrivită.)„. ([8]).

O problemă care este tratată în literatură este următoarea : „dându-se o secvență de N numere, se cere alegerea a K subsecvențe **disjuncte** astfel încât suma scorurilor subsecvențelor alese să fie maximă„. Pe cazul general, cea mai bună soluție este una cu programare dinamică în $O(N^2 * K)$. Daca funcția de scor este funcția sumă, atunci problema se poate rezolva în $O(N\alpha(n, n))$ [11] sau chiar $O(N)$ [8]. Vom prezenta în continuare rezolvarea unei versiuni modificate a acestei probleme, adâugând o constrângere suplimentară asupra lungimii subsecvențelor alese.

III.1 Exemplu de problemă

Se dă o secvență de numere întregi $A_1, A_2, \ldots, A_N$ și un număr L. O subsecvență $A_l, A_{l+1}, \ldots, A_r$ este dată de doi indici l și r cu $1 \leq l \leq r \leq N$. Să se găsească o submulțime de K subsecvențe disjuncte astfel încât suma totală a tuturor subsecvențelor alese este maximă și fiecare subsecvență aleasă are o lungime fixată L.

Soluție. Considerăm mai întai un exemplu cu 6 numere iar valoarea $L = 2$.

1	2	3	4	5	6
800	751	900	300	923	30

Alegând o singură subsecvență $[A_2, A_3]$ vom obține suma maximă egală cu $751 + 900 = 1651$. Alegând două subsecvențe disjuncte $[A_1, A_2]$ și $[A_4, A_5]$ vom obține suma maximă egală cu $800 + 751 + 923 + 300 = 2774$, iar pentru trei subsecvențe de lungime 3 singura variantă este să alegem $[A_1, A_2]$, $[A_3, A_4]$, $[A_5, A_6]$ cu suma totală 3704. Observăm că nu este necesar ca soluția pentru k subsecvențe să aibă subsecvențe în comun cu soluția pentru $k-1$ subsecvențe.

Putem găsi o primă soluție cu programare dinamică. Notând cu $dp_{\,n,\,k}$ suma maximă care se poate obține alegând k intervale disjuncte de lungime L din prefixul $v_1, v_2, \ldots, v_n$, obținem următoarea recurență:

$$dp_{\,n,\,k} = \begin{cases} 0, & \text{dacă } n < L \\ \max\{\ dp_{\,n-1,\,1};\ v_{n-l+1} + v_{n-l+2} + \ldots + v_n\ \} & \text{dacă } k = 1 \\ \max\{\ dp_{\,n-1,\,k};\ dp_{\,n-l,k-1} + v_{n-l+1} + v_{n-l+2} + \ldots + v_n\ \} & \text{altfel} \end{cases} \quad \text{(III.1)}$$

În recurența de mai sus, dacă lungimea prefixului $v_1, v_2, \ldots v_n$ este mai mică decât L, nu vom putea alege niciun interval, deci valoarea maximă va fi 0. Altfel, avem două opțiuni

- luăm intervalul de lungime L care se termină pe poziția n, caz în care valoarea va fi $v_{n-l+1} + v_{n-l+2} + \ldots + v_n + dp_{\,n-l,\,k-1}$
- nu luăm intervalul de lungime L care se termină pe poziția n, caz în care valoarea va fi $dp_{\,n-1,\,k}$

O implementare directă a acestei recurențe va avea complexitatea de timp $O(N * K * L)$, deoarece avem $O(N * K)$ stări iar pentru fiecare stare calculăm valoarea în $O(L)$. Putem îmbunătăți această soluție cu ajutorul unui vector auxiliar S al sumelor parțiale a prefixelor,

care se poate calcula astfel:

$$S_n = \begin{cases} v_1 & \text{dacă } n = 1 \\ v_n + S_{n-1} & \text{dacă } n > 1 \end{cases} \tag{III.2}$$

Complexitatea de timp pentru a calcula vectorul S este de $O(n)$. Având acest vector calculat, putem calcula suma $v_{n-l+1} + v_{n-l+2} + \ldots + v_n$ de mai sus în $O(1)$ ca și $S_n - S_{n-l}$. Obținem astfel o soluție în complexitate de timp $O(N * K)$ si memorie $O(N * K)$.

Ideea de optimizare

Considerăm următoarea modificare a problemei: putem să împărțim șirul în orice număr de subsecvențe, dar pentru fiecare subsecvență aleasă vom fi penalizați cu un $\lambda \in R$ fixat arbitrar. Intuitiv, dacă penalizarea λ este un număr foarte mare spre ∞, acest lucru ne forțează să alegem doar 1 subsecvență. Similar, dacă λ tinde spre $-\infty$, vom alege un număr de n subsecvențe. În general, pe masură ce creștem λ, numărul de subsecvențe din soluția optimă va scădea. Recurența pentru versiunea modificată a problemei va fi:

$$ans_\lambda(n) = \begin{cases} \max\{0, S(n) - \lambda\} & \text{dacă } n < L \\ \max\{\ ans_\lambda(n-1),\ ans_\lambda(n-l) + S(n) - S(n-l) - \lambda\} & \text{dacă } n \geq L \end{cases} \tag{III.3}$$

La calcularea recurenței vom reține un vector suplimentar cnt_λ cu semnificația următoare: soluția optimă $ans_\lambda(i)$ pentru prefixul $A_1, A_2, \ldots, A_i$ folosește $cnt_\lambda(i)$ subsecvențe. Deoarece pentru fiecare subsecvență aleasă am scăzut din soluție valoarea λ, costul real se va obține ca și $ans_\lambda(n) + cnt(n) \cdot \lambda$. Implementarea pentru un λ fixat va fi următoarea:

Algorithm 1 Calculează soluția pentru o penalizare λ fixată

Require: Un număr real λ, un șir S unde $S_i = A_1 + A_2 + \cdots + A_i$

Inițializează $best_\lambda(i) = -\infty$, $i \in 1 \cdots N$
for $i \in 1 \cdots N$ **do**
  $best_\lambda(i) = best_\lambda(i-1)$
  $cnt_\lambda(i) = cnt_\lambda(i-1)$
  if $best_\lambda(i-L) + S(i-L+1, i) - \lambda > best_\lambda(i)$ **then**
    $best_\lambda(i) = best_\lambda(i-L) + S(i-L+1, i) - \lambda$
    $cnt_\lambda(i) = cnt_\lambda(i-L) + 1$
  end if
end for
return $(best_\lambda(N) + \lambda \cdot cnt_\lambda(N), cnt_\lambda(N))$

Teorema III.1. Algoritmul de mai sus va folosi pentru un λ fixat valoarea K pentru care $f(K) - \lambda K$ este maximă.

Complexitatea algoritmului ilustrat mai sus este de $O(N)$ pentru un λ fixat. Trebuie în continuare să găsim un λ pentru care soluția optimă folosește K subsecvențe. Folosind faptul că pe măsură ce creștem λ numărul de subsecvențe rezultat în urma algoritmului de mai sus va scădea, putem căuta binar un λ pentru care soluția optimă folosește K subsecvențe. Implementarea căutării binare va fi următoarea: ([14])

Algorithm 2 Calculează soluția ce folosește cel mult K subsecvențe

Require: Un șir de numere reale A

$low = -sum(A)$
$high = sum(A)$
while $low \leq high$ **do**
  $mid = \frac{low+high}{2}$
  $(value, cnt) = Calc(\lambda)$
  if $cnt \leq k$ **then**
    $answer = value$
    $low = mid$
  else
    $high = mid$
  end if
end while
return $answer$

Totuși, nu ni se garantează că există un λ pentru care avem o soluție optimă ce folosește exact K subsecvențe. Notăm cu $f_x = dp_{\,n,\,x}$ costul maxim de a împărți **tot șirul** în x subsecvențe.

Conform III.1, dorim ca pentru fiecare $x \in \{1, 2, \ldots, n\}$ să existe λ astfel încât

$$f(x) - x \cdot \lambda \geq f(y) - y \cdot \lambda, \ \forall \, y \ \in \{\{1, 2, \ldots, n\} \setminus \{x\}\} \tag{III.4}$$

La prima vedere, o condiție posibilă suficientă este ca șirul nostru f_x să fie monoton. Dacă teoretic am obține un f astfel încât $f_1 = 1, f_2 = 2, f_3 = 4$, observăm că nu există $\lambda \in R$ astfel incât $f(2) - 2\lambda$ să fie maxim, deci nu există penalizarea λ pentru care soluția optimă să împartă tot șirul în 2 subsecvențe.

Considerăm un indice x arbitrar. Dacă prelucrăm condiția III.4 în funcție de y pentru un anumit x, obținem două tipuri de inegalități:

$$\begin{cases} \lambda_x \leq \frac{f(x)-f(y)}{x-y}, \ \forall \, y \ < x \\ \lambda_x \geq \frac{f(x)-f(y)}{x-y}, \ \forall \, y \ > x \end{cases}$$

În particular, dacă analizăm din inegalitățile de mai sus doar cazurile cand $y = x - 1$ sau $y = x + 1$ avem condițiile necesare, pe care vom demonstra că sunt și suficiente:

$$\begin{cases} \lambda_x \leq f(x) - f(x-1) \\ \lambda_x \geq f(x+1) - f(x) \end{cases}$$

Lema III.2. Fie f_x un șir de numere ce respectă condiția:

$$f(x+1) - f(x) \leq f(x) - f(x-1) \, , \forall x \in N.$$

Atunci:

$$\begin{aligned} \frac{f(x) - f(y)}{x - y} &\geq f(x) - f(x-1) \, , \forall \, y < x \\ \frac{f(x) - f(y)}{x - y} &\leq f(x+1) - f(x) \, , \forall \, y > x \end{aligned}$$

Demonstraţie. Fie $1 \leq x \leq n$ și $1 \leq y < x$. Atunci avem:

$$\begin{aligned}\frac{f(x)-f(y)}{x-y} &= \frac{f(x)-f(x-1)+f(x-1)-f(x-2)+\ldots+f(y+1)-f(y)}{x-y} \\ &\geq \frac{f(x)-f(x-1)+f(x)-f(x-1)+\ldots+f(x)-f(x-1)}{x-y} \\ &= f(x)-f(x-1)\end{aligned}$$

□

Procedăm similar în cazul când $1 \leq x \leq n$ și $x < y \leq n$:

$$\begin{aligned}\frac{f(x)-f(y)}{x-y} &= \frac{f(x+1)-f(x)+f(x+2)-f(x+1)+\ldots+f(y)-f(y-1)}{x-y} \\ &\leq \frac{f(x+1)-f(x)+f(x+1)-f(x)+\ldots+f(x+1)-f(x)}{x-y} \\ &= f(x+1)-f(x)\end{aligned}$$

Definiția III.3. O secvență $f(1), f(2), \ldots f(N)$ este **concavă** dacă $f(x)-f(x-1) \geq f(x+1)-f(x),\ x \in 1 \ldots N$ sau **convexă** dacă $f(x)-f(x-1) \leq f(x+1)-f(x),\ x \in 1 \ldots N$.

Teorema III.4. Fie f o secvență de numere **concavă**. Pentru fiecare $1 \leq x \leq n$:

- există $\lambda_x \in R$ astfel încât avem $f(x)-\lambda_x \cdot x \geq f(y)-\lambda_x \cdot y$ pentru $\forall\, 1 \leq y \leq n$ adică soluția optimă cu penalizarea λ_x împarte șirul în x subsecvențe.
- valorea pe care o poate lua λ_x este orice număr real din intervalul $[f(x+1)-f(x), f(x)-f(x-1)]$.

III.2 Demonstrația concavității pentru problema noastră

Pentru a demonstra concavitatea funcției f, vom scrie problema noastră sub forma unei probleme de programare liniara în numere întregi, pe care vom demonstra după că este echivalentă cu o problema de programare liniară în numere reale. Observăm că în urma aplicării recurenței programării dinamice în $O(NK)$, se vor alege cel mult k poziții din șir în care vom termina un interval de lungime L și mai mult, toate aceste intervale alese sunt disjuncte două câte două iar suma obținută va fi maximă. Putem obține o formulare după următorul model:

- n necunoscute $x_1, x_2, \ldots, x_N$, cu semnificația că x_p este numărul de poziții din prefixul $1, 2, \ldots, p$ pe care am început un interval de lungime L.
- obiectivul este să suma valorilor de pe pozițiile p care apar într-un interval ales, adică $\sum_p v_p$ unde $x_p - x_{p-L} = 1$.
- constrângerile sunt ca orice 2 intervale alese să nu se intersecteze, i.e dacă alegem un interval care începe pe poziția p nu putem alege alt interval care începe pe una din pozițiile $p, p-1, \ldots, p-L+1$

$$\text{maximize} \quad f(x) = \sum_{j=1}^{N} v_j \cdot (x[j] - x[p-L])$$

$$\begin{aligned}
\text{subject to} \quad x_j - x_{j-L} &\leq 1, \quad j \in L, L+1, \ldots, N && \text{(III.5)} \\
x_j &\leq 1, \quad j \in 1, 2, \ldots, L-1 && \text{(III.6)} \\
x_j &\geq x_{j-1}, \quad j \in 2, 3, \ldots, N && \text{(III.7)} \\
x_n &\leq K && \text{(III.8)} \\
x_j &\in Z \quad j \in 1, 2, \ldots, N && \text{(III.9)} \\
& && \text{(III.10)}
\end{aligned}$$

Inegalitățile (1) și (2) ne garantează că intervalele alese vor fi disjuncte. Inegalitatea (4) ne garantează ca numărul de intervale alese va fi mai mic sau egal decât K. În final, inegalitatea (5) împreună cu inegalitățile anterioare ne garantează $x_j - x_{j-1} \in \{0, 1\}$, adică pentru o poziție j vom avea 1 dacă o alegem ca început de interval de lungime L și 0 altfel. Numărul total de inegalități este de $2N + 1$ și este clar că inegalitățile de mai sus pot fi puse în formă matriceală, obținând un program de forma următoare, numită formă canonică a programelor liniare: ([15], [17])

$$\begin{aligned}
&\text{maximize} \quad f(x) = c^T x \\
&\text{subject to} \quad Ax \leq b_K && \text{(III.11)}
\end{aligned}$$

Am notat cu b_K vectorul coloană de restricții asociat numărului K. Se observă că pentru

două valori $K1$ și $K2$ diferite, vectorii b_{K1} și b_{K2} diferă în exact o poziție și mai exact aceea corespunzătoare inegalității $x_n \leq K$. Definim următoarea funcție:

$$g\colon R^{2N+1} \to R, g(b) = \max_{x|Ax \leq b} c^T \cdot x$$

Dacă nu am avea restricțiile $x_j \in Z, j = 1 \dots N$, atunci funcția g_b ar fi *concavă* conform II.8. Așadar, din definiția *concavității* funcției g se obține pentru $K = 1 \dots N-1$:

$$\begin{aligned} g(\frac{1}{2}b_{K-1} + \frac{1}{2}b_{K+1}) &\geq \frac{1}{2}g(b_{K-1}) + \frac{1}{2}g(b_{K+1}) \\ \iff g(b_K) &\geq \frac{g(b_{K-1}) + g(b_{K+1})}{2} \\ \iff f(K) &\geq \frac{f(K-1) + f(K+1)}{2} \\ \iff f(K) - f(K-1) &\geq f(K+1) - f(K) \end{aligned}$$

Reamintim că $f(K)$ era definit pentru problema inițială ca fiind costul maxim de a alege K subsecvențe disjuncte de lungime L fiecare. Însă, programul nostru liniar conține și restricții de integritate. Ne vom folosi în continuare de proprietatea II.7, care ne spune că pentru orice program liniar există cel puțin o soluție de cost optim printre punctele sale de extrem. Așadar, cum noi lucrăm mai sus doar cu vectorii coloană $b_k, k = 1 \dots N$ trebuie să demonstrăm că pentru fiecare dintre acești vectori de constrângere, există cel puțin o soluție optimă a programului liniar ce are ca și componente doar numere întregi.

În continuare, vom fixa un număr $k \in 1 \dots N$ arbitrar și vectorul coloană de restricții asociat al acestuia b_k. Vom demonstra că fiecare vector care este punct de extrem al mulțimii $Ax \leq b_k$ conține doar componente întregi. Pentru a face acest lucru, vom lua un vector $x \in R^n$ cu proprietatea $Ax \leq b_k$ care conține cel puțin o valoare și vom arăta că se poate scrie ca o combinație liniară de alți doi vectori x_1, x_2 astfel încât $Ax_1 \leq b_k$ și $Ax_2 \leq b_k$, contrazicând astfel faptul că x este punct de extrem.

Să presupunem ca $x = (x_1, x_2, \dots, x_n)$ este un vector arbitrar astfel încât $Ax \leq b_k$ și care conține cel puțin o poziție i astfel încât x_i nu este număr întreg. Vom arăta cum putem găsi o combinație liniară validă făcând modificări pe vectorul x. Dacă alegem o poziție $p < n$ și

mărim x_p cu un $\epsilon > 0$ foarte mic, trebuie să mărim cu ϵ următoarele pentru a păstra inegalitatea $Ax \leq b_k$:

- x_{p-L}, dacă $x_p - x_{p-L} = 1$
- x_{p+1}, dacă $x_p = x_{p+1}$

Similar, dacă alegem o poziție $p < n$ și micim x_p cu $\epsilon > 0$, trebuie să micim cu ϵ următoarele:

- x_{p+L}, dacă $x_{p+L} - x_p = 1$
- x_{p-1}, dacă $x_p = x_{p-1}$

Deoarece vectorul x este crescător, o valoare y va apărea în vector într-un interval de poziții consecutive. Fie L_y cea mai din stânga apariție a lui y în vector și R_y cea mai din dreapta apariție. Fie p ultima poziție a vectorului x astfel încât x_p nu este număr întreg și $[L_{x_p}, p]$ intervalul de valori a lui x_p. Alegând un $\epsilon > 0$ foarte mic și mărind toate valorile $L_{x_p}, L_{x_p} + 1, \ldots, p$ cu ϵ, observăm că singura constrângere pe care o putem încalca este $x_j - x_{j-L} \leq 1,\ j \in [L_{x_p}, p]$. Acest lucru se întâmplă când avem un $j \in [L_{x_p}, p]$ astfel incât $x_j - x_{j-L} = 1$, adică $x_{j-L} = x_p - 1$. Luând $[L_{x_p-1}, R_{x_p-1}]$ intervalul în care se găsește $x_p - 1$ si mărind toate valorile cu ϵ din acest interval, din nou singura constrângere pe care o putem încălca este $x_j - x_{j-L} \leq 1,\ j \in [L_{x_p-1}, R_{x_p-1}]$, adică $x_{j-L} = x_p - 2$. Observăm că ne aflăm într-un caz identic cu cel anterior, deci putem repeta inductiv acest procedeu dacă este nevoie pentru intervalele $[L_{x_p-2}, R_{x_p-2}]$, $[L_{x_p-3}, R_{x_p-3}]$, $\ldots$, $[L_{x_p-[x_p]}, R_{x_p-[x_p]}]$. Acest procedeu este finit și va mări cu ϵ valoriile din cel mult $[x_p]$ intervale. Notăm șirul x obținut după aplicarea procedeului descris anterior cu x_1.

Un al doilea șir x_2 valid se poate obține astfel: pentru fiecare dintre intervalele descrise anterior, scădem un ϵ foarte mic din toate valorile acelui interval. Așadar, pentru fiecare poziție i din vectorul $x = (x_1, x_2, \ldots, x_n)$, vom avea:

- fie $x_1(i) = x_i, x_2(i) = x_i$
- fie $x_1(i) = x(i) + \epsilon, x_2(i) = x(i) - \epsilon$.

Am obținut astfel doi vectori valizi $x1$ și $x2$ cu $Ax_1 \leq b_k$ și $Ax_2 \leq b_k$ cu proprietatea că $\frac{1}{2}x_1 + \frac{1}{2}x_2 = x$, deci vectorul x nu este punct de extrem. În concluzie, fiecare vector care este punct de extrem al mulțimii $\{x \in R^n \mid Ax \leq b_k\}$ conține doar coordonate întregi și deci funcția f este concavă.

III.3 Rezultate experimentale

Reamintim că soluția neoptimizată are complexitatea de $O(NK)$, iar cea optimizată are complexitatea de $O(Nlog_2W)$, unde W este valoarea maximă a unui număr din șir. Comparăm în continuare timpii de execuție al acestor soluții pe care le-am implementat în $C++$.

Alegem mai întai $N = 10^5$, $K = \frac{10^4}{2}$, $L = 20$ iar șecvența de numere $A = (A_1, A_2, \ldots, A_n)$ va conține numere generate aleator din intervalul $[0, 10^9]$. Algoritmul neoptimizat găsește $K = \frac{10^4}{2}$ subsecvențe disjuncte de lungime $L = 20$ fiecare și sumă totală maximă în 13 secunde, iar cel optimizat în $O(Nlog_2W)$ în 0.2 secunde. Pentru $N = 10^5, L = 150, K = \frac{10^5}{200}$, timpul obținut de algoritmul în $O(NK)$ scade la 2.3 secunde, iar cel în $O(N \log_2 W)$ va dura în continuare 0.2 secunde. Pentru $N = 10^6$ și valori diferite K, L algoritmul optimizat este încă eficient, rulând în medie în 0.9 secunde. În toate testele anterioare am ales valoarea K maximă în raport cu N și L. Algoritmul optimizat este foarte rapid, obține timpi similari pentru orice valoari ale numerelor K și L, ceea ce este un lucru intuitiv deoarece el depinde doar de valoarile N și log_2W și este cu mult mai rapid decât cel optimizat când L este semnificativ mai mic decât N.

III.4 Relația cu grafuri Monge

O tehnică similară cu cea descrisă anterior a fost folosită în practică pentru calcularea drumului minim de lungime K într-un graf Monge. În această secțiune, vom descrie soluția găsită de B.Scheiber et. al în [2].

Fie $G = (V, E)$ un graf ponderat, orientat, complet și aciclic cu mulțimea nodurilor $V = (v_1, v_2, \ldots, v_n)$. Pentru $1 \leq i < j$, notăm cu $w(i, j)$ valoarea asociată cu arcul (i, j). Un drum în graful G are lungime K dacă conține exact k arce. Pentru două noduri i și j, numim un drum P de lungime minimă dacă are lungime k, pleacă din i și ajunge în j și este de cost minim în

raport cu drumurile ce îndeplinesc cele două condiții anterioare.

Definiția III.5. Un graf ponderat G îndeplinește condiția **Monge** de *concavitate* dacă $w(i,j) + w(i+1,j+1) \leq w(i+1,j) + w(i,j+1)$ pentru fiecare $1 < i+1 < j < n$ ([3]).

Cerința este să găsim în acest graf Monge un drum minim de la nodul 1 la nodul n de lungime K fixată.

B.Scheiber et.al, 1994 ([2]) au găsit o soluție în complexitate $NlogU$ (unde U este costul maxim al unei muchii) ce pleacă de la următoarea idee:

> Se definește pentru un $\epsilon > 0$ graful $G(\epsilon)$ ca fiind graful ponderat cu aceeași mulțime de noduri și muchii ca graful inițial, în care fiecare muchie $e \in E$ are costul înlocuit cu $w(e) + \epsilon$ (unde $w(e)$ este costul muchiei din graful G inițial). Este ușor de verificat că și graful $G(\epsilon)$ îndeplindește condiția de concavitate Monge.

Ei folosesc următoarele proprietăți:

- Drumul de cost minim și lungime nerestricționată se poate calcula în timp $O(N)$ într-un graf Monge folosind algoritmul găsit de către Wilber et. al în [16]
- Dacă drumul de cost minim (de lungime nerestricționată) în graful $G(\epsilon)$ are lungimea k, atunci drumul de cost minim în graful $G(\epsilon`)$ cu $\epsilon` < \epsilon$ are lungimea $k` \leq k$. [2]

  Practic, pe măsură ce creștem costurile muchilor, lungimea drumului de cost minim va rămâne fie la fel, fie va scădea. Această proprietate are loc în orice graf orientat aciclic ponderat, nu doar în grafuri Monge.
- Proprietate specifică grafurilor Monge: pentru fiecare $k = 1 \ldots N$ există $\epsilon \in R$ astfel încât drumul minim în graful $G(\epsilon)$ are lungimea k. ([2])

După aceea ei folosesc metoda căutarii binare ([14]) pentru a afla ϵ pentru care în graful $G(\epsilon)$ drumul de cost minim are lungimea k. Cum în graful Monge $G(\epsilon)$ putem calcula drumul de lungime minimă în timp $O(N)$ ([16]), iar căutarea binară are complexitate de $O(log_2 V)$ ([14]) unde V este valoarea maximă a unei muchii, complexitatea finală va fi de $O(N \log_2 V)$. Există mai multe aplicații care se pot reduce la calcularea drumului minim într-un graf *Monge*, câteva care sunt menționate în [2] sunt:

- Arborele Huffman de adâncime limitata fixată K.
- Găsirea a K clici într-un graf determinat de N intervale $(a[i], b[i])_{i=0}^{N-1}$.
- Compresia datelor prin cuantizare.

IV Max-Convolution

Problema de $(\max, +)$ convolution este definită astfel: dându-se doi vectori de numere reale $A = (A_0, A_1, \ldots, A_{n-1})$ și $B = (B_0, B_1, \ldots, B_{m-1})$ notăm cu A + B vectorul $C = (C[0], C[1], \ldots, C[m+n-1])$ care se obține astfel:

$$C[k] = \max_{\substack{0 \le i < n \\ 0 \le k-i \le m}} A[i] + B[k-i] \tag{IV.1}$$

Acest lucru poate fi imediat făcut în timp $O(nm)$, iterând pentru fiecare $k \in 0 \ldots n$ prin toți indicii valizi și luând valoarea maximă $A[i] + B[k-i]$. Pe cazul general nu există o soluție în timp mai bun de $O(n^{2-\epsilon})$ ([5]) unde $\epsilon > 0$ și o discuție în legătură cu convoluția min/max și echivalențe ale acesteia cu alte probleme cum ar fi problema rucsacului, problema 3-SUM etc. se poate găsi în lucrarea [5].

IV.1 Cazul concav/convex

Definiția IV.1. Un vector $V = (V_1, V_2, \ldots, V_n)$ se numește vector concav dacă îndeplinește condiția următoare:

$$V[i] - V[i-1] \ge V[i+1] - V[i], i = 2 \ldots N-1$$

În cazul în care vectorii A și B au proprietatea de *concavitate* atunci problema de (max, +) convoluție se poate rezolva în $O(n+m)$. Această tehnică a fost folosită pentru rezolvarea unor probleme în cadrul unui concurs de echipe, în stil ACM-ICPC care a avut loc la Petrozavodsk în anul 2019. O problemă interesantă care se poate rezolva cu această tehnică este problema cuplajului de sumă maximă într-un arbore și dimensiune $K = 1, 2, \ldots N$. În continuare, vom descrie metoda de convoluție pentru acest caz special.

Teorema IV.2. Dacă elementul $C[k] = \max_{l=0\cdots k}\{A[l] + B[k-l]\}$ se obține din $A[i] + B[j]$, atunci elementul $C[k+1] = \max_{l=0\cdots k+1}\{A[l] + B[k-l+1]\}$ este egal cu maximul dintre $A[i+1] + B[j]$ și $A[i] + B[j+1]$.

Acest rezultat a mai fost prezentat în [**?**], interpretrând problema de convoluție ca una de sumă Minkowski ([6]). O altă soluție în $O(n+m)$ este prezentată în [4] și reduce problema la una de căutare într-o matrice Monge [16]. Detaliem în continuare cum putem obține rezultatul și oferim o demonstrație simplă a teoremei IV.2 folosind inducție matematică.

Extinzând relația $C[k+1] = \max(A[i+1] + B[j], A[i] + B[j+1])$ obținem:

$$C[k+1] = \begin{cases} A[i+1] + B[j], & \text{dacă } A[i+1] - A[i] \geq B[j+1] - B[j] \\ A[i] + B[j+1], & \text{dacă } B[j+1] - B[j] \geq A[i+1] - A[i] \end{cases} \tag{IV.2}$$

Așadar, problema se reduce la combinarea vectorilor de diferențe $A' = (A[0], A[1] - A[0], \ldots, A[n-1] - A[n-2])$ și $B' = (B[0], B[1] - B[0], \ldots, B[m-1] - B[m-2])$ într-un vector $C' = (C[0], C[1] - C[0], C[2] - C[1], \ldots, C[m+n-1] - C[m+n-2])$ ce respectă la rândul lui proprietatea de *concavitate*. Algoritmul de obținere a vectorului $C = (C_0, C_1, \ldots, C_{m+n-1})$ se bazează pe teorema de mai sus și folosește algoritmul clasic de interclasare (prezentat, de exemplu în [13]) ce urmează următorii pași:

> Folosim doi indici i și j care sunt inițial egali cu 1. La fiecare pas, dacă $A[i] - A[i-1]$ este mai mare decât $B[j] - B[j-1]$, atunci adăugam valoarea $A[i] - A[i-1]$ la secvența C' și mărim i cu 1. Altfel, adăugăm valoarea $B[j] - B[j-1]$ la secvența C' și mărim j cu 1.

La final, obținem vectorul de diferențe al convoluției

$$C' = (C[0], C[1] - C[0], C[2] - C[1], \ldots, C[m+n-1] - C[m+n-2])$$

Având acest vector, putem restitui ușor vectorul $C = (C[0], C[1], \ldots, C[m+n-1]))$ astfel:

$$C[0] = C'[0]$$
$$C[i] = C[i-1] + C'[i]$$

Complexitatea de timp a algoritmului de interclasare este de $O(n+m)$. Implementarea în pseudocod a convoluției bazată pe interclasare ([13]) este următoarea:

Algorithm 3 MaxConv - Convolution of two concave sequences

Require: Two concave vectors $A = (A[0], A[1], \ldots, A[n])$

$c`[0] = A[0] + B[0]$
$i = 1, j = 1$
while $i \leq n$ and $j \leq m$ **do**
  if $A[i] - A[i-1] \leq B[j] - B[j-1]$ **then**
    $C`[i+j-1] = A[i] - A[i-1]$
    $i = i + 1$
  else
    $C`[i+j-1] = B[j] - B[j-1]$
    $j = j + 1$
  end if
end while
while $i \leq n$ **do**
  $C`[i+j-1] = A[i] - A[i-1]$
  $i = i + 1$
end while
while $j \leq m$ **do**
  $C`[i+j-1] = B[j] - B[j-1]$
  $j = j + 1$
end while
$c[0] = c`[0]$
for $i = 1 \ldots m + n - 1$ **do**
  $C[i] = C[i-1] + C`[i]$
end for

Vom demonstra corectitudinea algoritmului de mai sus prin inducție matematică. Arătăm inductiv că dacă $C[k] = A[i] + B[j]$, atunci $C[k+1] = max(A[i+1] + B[j], A[i] + B[j+1])$. Ținând cont că avem $C[k] = A[i]+B[j]$ și de faptul că vectorii $A[i]_{i=0}^{n-1}$ și $B[i]_{i=0}^{m-1}$ sunt concavi, vor avea loc următoarele inegalități:

$$B[j] - B[j-1] \geq B[j+1] - B[j] \geq \ldots \geq B[m-1] - B[m-2]$$
$$A[1] - A[0] \geq A[2] - A[1] \ldots \geq A[i] - A[i-1]$$
$$A[i] - A[i-1] \geq B[j+1] - B[j]$$

Primele două inegalități de mai sus au loc din definiția concavității vectorilor A și B. Cea de-a treia rescrie inegalitatea $A[i]+B[j] \geq A[i-1]+B[j+1]$, ce are loc deoarece am presupus că $C[k] = A[i] + B[j]$. Combinând cele trei inegalități, vom obține:

$$A[1] - A[0] \geq \ldots \geq A[i] - A[i-1] \geq B[j+1] - B[j] \geq \ldots \geq B[m-1] - B[m-2] \quad \text{(IV.3)}$$

Obținem generalizat, $A[x] - A[x-1] \geq B[y+1] - B[y],\ \forall x \leq i\ \ \&\ \ y \geq j$.

Lema IV.3. $A[i] + B[j+1] \geq A[i-1] + B[j+2] \geq \ldots \geq A[1] + B[k] \geq A[0] + B[k+1]$.

Demonstraţie. Rescrierea lemei de mai sus duce la verificarea următoarei condiții:

$$A[i-k] - A[i-k-1] \geq B[j+k+1] - B[j+k] \qquad \forall\, k \in [0, \min(i, j-m)]$$

Cum $i - k \leq i$ și $j + k \geq j$, aplicăm IV.1 pentru a obține rezultatul. □

Lema IV.4. $A[i+1]+B[j] \geq \max(A[i+2]+B[j-1], A[i+3]+B[j-2], \ldots, A[i+j+1]+B[0])$

Demonstraţie. Trebuie să demonstrăm următoarele inegalități:

$$A[i+1] + B[j] \geq A[i+1+k] + B[j-k], \forall k \in [1, \min(n-i, j)] \quad \text{(IV.4)}$$

Din faptul că $C[k] = A[i] + B[j]$, avem:

$$A[i] + B[j] \geq A[i+k] + B[j-k], \forall k \in [1, \min(n-i+1, j)] \tag{IV.5}$$

Se observă foarte ușor că dacă pentru un k fixat adăugăm în membrul stâng valoarea $A[i+1] - A[i]$ și în membrul drept valoarea $A[i+1+k] - A[i+k]$ se păstrează inegalitatea (deoarece șirul de diferențe este monoton descrescător, din definiția concavității) și mai mult, obținem chiar inegalitatea corespunzătoare ce trebuia demonstrată $A[i+1] + B[j] \geq A[i+1+k] + B[j-k]$. Am demonstrat așadar că dacă $C[k] = A[i] + B[j]$, singurele valori care pot fi egale cu $C[k+1]$ sunt fie $A[i+1] + B[j]$, fie $A[i] + B[j+1]$.

IV.2 Convoluția a $Q \geq 3$ vectori concavi

O extensie naturală a convoluției pentru doi vectori este cazul în care avem mai mulți vectori, mai exact:

Se dau Q vectori concavi $V_0, V_2, \ldots, V_{Q-1}$. Notăm lungimea unei vector V_i dintre cei Q cu $|V_i|$ și elementul de pe poziția j în vectorul V_i cu $V_i(j)$ pentru $j = 0 \ldots Q-1$. Definim convoluția max peste acești vectori ca fiind șirul C cu următoarea proprietate:

$$C[k] = \max_{\substack{i_0+i_1+\ldots+i_{Q-1}=k \\ 0 \leq i_j \leq |s_j|, j=0\ldots Q-1}} \sum_{j=0}^{Q-1} V_j(i_j) = V_1 + V_2 + \ldots = V_{Q-1} \tag{IV.6}$$

În cazul $Q = 2$ problema se reduce la calcularea convoluției pentru doi vectori concavi și a fost discutat în secțiunea anterioară. Vom considera în continuare cazul $Q \geq 3$, când avem mai mult de doi vectori.

O primă soluție evidentă, care se folosește de soluția pentru cazul $Q = 2$ se bazează pe calcularea secvențială a convoluției. [4]

Lema IV.5. Numărul de operații efectuat de algoritmul de mai sus este $\sum_{i=0}^{Q-1} |V_i| * (Q-i)$.

Lema IV.6. Complexitatea soluției de mai sus este în cel mai rău caz $O(NQ)$, unde $N = |s_0| + |s_1| + \ldots + |s_{Q-1}|$ (suma lungimilor secvențelor).

Demonstraţie. Luăm un șir de vectori în care $V[0]$ are $\frac{N}{2}$ elemente, iar ceilalți $Q-1$ vectori au

Algorithm 4 QMaxConv

Require: Q vectors $V[0], V[1], \ldots, V[Q-1]$ with arbitrary length

1: $Conv = V[0]$
2: **for** $i = 1 \ldots Q-1$ **do**
3: $\quad Conv = Conv + V[i]$
4: **end for**
5: **return** $Conv$

doar 1 element. Aplicând IV.5, vectorul V_0 va contribui la sumă cu $\frac{N}{2} \cdot (Q-1) = \frac{N(Q-1)}{2} \in O(NQ)$ operații. Într-adevăr, efectuăm de Q ori convoluția dintre un vector de dimensiune minim $\frac{N}{2}$ și unul de dimensiune 1, ce necesită $\frac{N}{2}$ operații per convoluție. □

Teorema IV.7. $V0 + V_1 + \ldots + V_{Q-1}$ se poate calcula în complexitate $O((N+Q)log_2N)$. ([4]).

Rezultatul nu este unul nou și o soluție folosind divide-et-impera a fost descrisă în mai multe lucrări, de exemplu[4]. Vom descrie o soluție obținută prin reinterpretarea algoritmului Huffman [10] folosit pentru compresia datelor. Efectuăm următoarele operații de $Q-1$ ori:

- 1) luăm cei 2 vectori A și B cu dimensiunea cea mai mică dintre cei rămași
- 2) calculăm convoluția $A + B$ în complexitate $|A| + |B|$
- 3) ștergem cei doi vectori și îi înlocuim cu $A + B$.

Teorema IV.8. Complexitatea algoritmului de mai sus este de $O((N+Q)log_2N)$.

Demonstraţie. Putem vizualiza numărul de operații al soluției de mai sus în felul următor: construim un arbore binar în care frunzele sunt dimensiunile vectorilor $|V[0]|, |V[1]|, \ldots, |V[Q-1]|$. La fiecare pas, luăm cele două noduri de valoare minimă și adăugăm un nod care va fi „părintele„ celor două noduri în arbore și va avea ca valoare suma lor. Numărul de operații efectuat în total de pasul 2) descris mai sus va fi suma valorilor nodurilor din arborele rezultat după $Q-1$ pași, sau, echivalent $\sum_{i=0}^{Q-1} |V[x]| \cdot h[x]$ unde $h[x]$ este adâncimea nodului corespunzător vectorului n în arbore.

Demostrăm în continuare că adâncimea oricărui nod x nu poate depăși $2 \cdot log_2N$. Plecăm de la primul moment în care am efectuăm o operație cu nodul x. Acest lucru înseamnă că am efectuat convoluția dintre x și un nod y, iar x a avut valoarea minimă sau cea de-a doua valoare minimă la momentul respectiv. Dacă a avut valoarea minimă, atunci deoarece $y \geq x$ părintele

lui x va avea valoarea $x + y \geq 2x$, ceea ce nu se poate întâmpla de mai mult de $log_2 N$ ori. Dacă a avut cea de-a doua valoare minimă, când vom efectua următoarea sumă folosind părintele lui x acesta va fi necesar de valoare minimă, ce se poate întâmpla de cel mult $log_2 N$ ori. Așadar, adâncimea unui nod este cel mult $2 \cdot log_2 N$ și deci numărul de operații este în cel mai rău caz $\sum_{n=0}^{Q-1}(|V[n] \cdot h[n]) = 2 \cdot \log_2 N \cdot N \in O(N \log_2 N)$. În plus, complexitatea operațiilor de tip 1) și 3) - ștergere, înlocuire pot fi efectuate în $log_2 N$ per operație folosind structura de date „Heap„ ([**?**]). În total, complexitatea este de $O(N \log_2 N + Q log_2 N)$. □

V Convex Hull Trick

În concursuri, multe probleme de programare dinamică se reduc la implementarea unei structuri de date ce permite următoarele operații:

- $ADD(A, B)$ - adăugarea unei funcții $Ax + B$
- $QUERY(x)$ - care este funcția de valoare maximă la coordonata x?

O soluție evidentă să ne ținem o listă cu toate funcțiile întâlnite până la un moment dat. Când efectuăm un $QUERY(x)$ luăm toate funcțiile (A_i, B_i) din listă, evaluăm fiecare valoare $A_i * x + B_i$ și o alegem pe cea mai mare. Această soluție adaugă o linie în listă în complexitate de $O(1)$ și rezolvă un query în $O(N)$, unde N este numărul de funcții din listă. Soluția brute-force va arăta în felul următor:

Algorithm 5 Algoritmul brute-force

Require: A list of linear functions, a real number x

```
best = -∞
for line(A_i, B_i) in lines do
    if A_i * x + B_i > best then
        best = A_i * x + B_i
    end if
end for
return best
```

Vom trata mai întâi cazul în care funcțiile ce trebuie adăugate $(A_i x + B_i)_{i=0}^{N-1}$ îndeplinesc condiția $A_1 \leq A_2 \leq A_{N-1}$, adică sunt ordonate crescător în raport cu valorile A_i.

Teorema V.1. Există o structură de date ce permite adăugarea a N funcții ordonate crescător dupa A în complexitate amortizată $O(N)$ și efectuarea de query-uri în timp $O(log_2 N)$.

Structura de date este cunoscută ca și „Convex Hull Trick„ în concursuri și mai este descrisă de exemplu în [9]. Pentru o pereche de funcții $f = Ax + B$ și $g = Cx + D$ cu $C \neq D$ este cunoscut că există o valoare unică I pentru care $f(I) = g(I)$ și este egală cu $\frac{D-B}{A-C}$. Aceasta se numește punctul de intersecție al celor două funcții. Definim funcția $h(x)\colon R \to R$, $h(x) =$

$max(f(x), g(x)) = max(Ax + B, Cx + D)$. O proprietate utilă în rezolvarea problemei este următoarea, presupunând că $C > D$:

$$h(x) = \begin{cases} f(x) = Ax + B, & \text{dacă } x \leq I \\ g(x) = Cx + D & \text{dacă } x > I \end{cases} \tag{V.1}$$

Interpretarea este următoarea: pentru orice pereche de funcții $Ax + B$ și $Cx + D$ cu $C > A$, funcția $Ax + B$ va fi maximă în intervalul $(-\infty, I)$ iar funcția $Cx + D$ va fi maximă în (I, ∞).

Corolar V.2. Fiecare funcție (A_i, B_i) din input va fi de valoare maximă într-un interval continuu de numere $[L_i, R_i]$ (posibil vid) care nu se poate intersecta cu celelalte intervale decât în capete.

De aici rezultă un alt algoritm lent, dar pe care se bazează soluția optimă: menținem pentru fiecare funcție f intervalul $[L_f, R_f]$ în care este maxim în raport cu funcțiile adăugate. Când adăugăm o funcție nouă g, pentru fiecare funcție f adăugată anterior calculăm punctul I de intersecție al celor două funcții și efectuăm operația $R_f = \min(R_f, I)$ conform V.1. Pentru funcția nouă adăugată setăm inițial intervalul $[L_g, R_g] = (-\infty, \infty)$. Luăm din nou fiecare din funcțiile adăugate anterior și efectuăm $L_g = \max(L_g, I)$. Capătul dreapta al intervalului R_g va rămâne temporar egal cu ∞, deoarece funcțiile date $(Ax_i + b_i)_{i=0}^{n-1}$ sunt ordonate crescător dupa A_i. Când obținem în urma actualizărilor $L_f > R_f$, acest lucru înseamnă că funcția f nu mai este maximă în niciun interval și trebuie eliminată. Când efectuăm un $Query(x)$, căutăm funcția f pentru care $x \in [L_f, R_f]$ - deoarece f este funcția de valoare maximă în acel interval și afișăm rezultatul. Acest algoritm adaugă fiecare funcție în $O(N)$, rezultând o complexitate mare de adăugare a N funcții de $O(N^2)$.

Algoritmul optim este următorul și se bazează de fapt pe modificarea celui descris anterior: Menținem o stivă care este inițial vidă și conține funcțiile care încă sunt maxime într-un interval. Când adăugăm o funcție f, luăm cele mai recente 2 funcții rămase în stivă g și h, unde g este vârful stivei. Calculăm punctul de intersecție I al funcțiilor f și g și efectuăm la fel ca în soluția anterioară operația $R_g = \min(R_g, I)$. Dacă după această modificare obținem $L_g > R_g$, intervalul în care g e maxim devine vid. În acest caz eliminăm funcția g din stivă - în timp $O(1)$ (deoarece g e vârful stivei) și repetăm verificarea pentru noul vârf al stivei. Dacă însă

după modificarea $R_g = \min(R_g, I)$, funcția g rămâne relevantă, ne oprim și adăugăm funcția f în stivă cu intervalul în care este maxim egal cu $(I, -\infty)$.

Algorithm 6 AddFunction

Require: A function $f_1 = ax + b$ and a stack S previously defined

```
1:  while TRUE do
2:      f2 = topmost function of stack
3:      f3 = second topmost function of stack
4:      if I(f1, f2) ≤ I(f2, f3) then
5:          pop f2 from the stack
6:      else
7:          BREAK
8:      end if
9:  end while
10: PUSH f1 onto the stack
```

Practic, algoritmul efectuează operații similare cu cel în $O(N^2)$, dar efectuează mult mai puține modificări. În stivă, vom adăuga în total exact o dată cele N funcții la linia 10. În bucla $WHILE$, dacă condiția de la linia 4 este îndeplinită atunci dimensiunea stivei va scădea cu 1, iar dacă nu se va ieși din buclă. Așadar, numărul total de operații este de $2N \in O(N)$.

Pentru funcția de query, având stiva cu funcțiile relevante $S = (S_1, S_2, \ldots, S_k)$ vom avea că S_i e maxim în intervalul $[I(S_i, S_{i-1}), I(S_i, S_{i+1}]$ (notând cu $I(f, g)$ punctul de intersecție) și în plus $I(S_i, S_{i-1}) \leq I(S_i, S_{i+1})$. Vom efectua astfel o căutare binară [14] pentru a găsi o poziție p astfel încât $x \in [I(S_p, S_{p-1}), I(S_p, S_{p+1})]$.

Algorithm 7 Query

Require: A real number x, a stack S previously defined

```
1:  lo = 0
2:  hi = S.size() − 1
3:  while lo ≤ hi do
4:      mid = (lo+hi)/2
5:      if x ≤ I(S[mid], S[mid − 1]) then
6:          lo = mid + 1
7:          answer = S[mid].A ∗ x + S[mid].B
8:      else
9:          hi = mid − 1
10:     end if
11: end whileRETURN answer
```

Complexitatea pentru $Query$ este de $O(log_2 N) = O(log_2 |S|)$, unde N este numărul de elemente din stivă (care poate fi cel mult egal cu N).

V.1 Aplicarea structurii de date pentru probleme de programare dinamică

Problemele de împărțire în batch-uri a N job-uri sunt bine studiate și sunt tratate de exemplu în [18] sau [12]. Există mai multe variații al acestui tip de problemă, de exemplu variind funcția obiectiv sau în care ordinea în care trebuie executate job-urile nu este fixată. Vom arăta în continuare cum putem rezolva un tip dintre aceste probleme folosind structura de date definită anterior.

Enunț. „Se dau N joburi cu timpi de procesare p_i ($i \in 1 \cdots N$) care trebuie procesate pe o singură mașină în această ordine. O secvență de procesare trebuie împărțită în mai multe batch-uri. Pentru fiecare bloc există un timp de preprocesare s ce nu depinde de bloc sau de dimensiunea acestuia. Timpul de terminare al job-ului i în blocul B_j este timpul de terminare al ultimului job în B_j. Cum ar trebui să fie job-urile împărțite în batch-uri astfel încât valoarea totală

$$\sum_{i=1}^{N} \alpha_i * f_i \tag{V.2}$$

să fie minimă?„ ([12])

Soluție.

Pentru o secvență dată $A_1, A_2 \cdots A_N$ și o partiționare $1 = i_1 < i_2 < \cdots i_k < i_{k+1} = N$ în batch-uri $B_j = \{i_j, i_j + 1, \cdots, i_{j+1} - 1\}$ funcția obiectiv poate fi scrisă astfel ([12])

$$\sum_{i=1}^{N} \alpha_i * f_i = \sum_{j=1}^{k} (\sum_{v=i_j}^{N} \alpha_v) P_j = \sum_{j=1}^{K} (\sum_{v=i_j}^{N} \alpha_v)(s + \sum_{v=i_j}^{i_{j+1}-1} p_j)$$

Pentru a rescrie suma de mai sus într-o formă mai simplă, vom defini următoarele variabile

- $S_\alpha(i) = \sum_{j=i}^{N} \alpha_j$, suma valorilor α_j cu $j \in i \cdots N$
- $S_p(i) = \sum_{j=i}^{N} p_j$, suma valorilor p_j cu $j \in i \cdots N$

Folosind aceste notații, vom avea $\sum_{v=i_j}^{N} \alpha_v = S_\alpha(i_j)$ și $\sum_{v=i_j}^{i_{j+1}-1} p_j = S_p(i_j) - S_p(i_{j+1})$.

Înlocuind, noua funcție obiectiv devine:

$$\sum_{j=1}^{K} s * S_\alpha(i_j) + S_\alpha(i_j) * S_p(i_j) - s * S_p(i_{j+1}) \tag{V.3}$$

Vom găsi cea mai bună împărțire în batch-uri $i_1 < i_2 < \cdots i_K < i_{K+1}$ care minimizează suma de mai sus folosind programare dinamică. Notăm cu dp_i costul minim de a împărți *doar secvența* $i, i+1, \cdots N$ în batch-uri și avem următoarea recurență:

$$dp_i = \min_{j>i} S_\alpha(j) \cdot S_p(i) + (dp_j + S_\alpha(j) \cdot s - S_\alpha(j) \cdot S_p(j)) \tag{V.4}$$

Pentru fiecare $i \in [N \cdots 1]$ vom itera prin $j > i$, vom calcula valoarea corespunzătoare pentru j definită în recurență și vom lua maximul. Complexitatea de timp pentru a calcula direct recurența de mai sus este de $O(N^2)$.

Notând cu $A_j = S_\alpha(j)$ și cu $B_j = dp_j + S_\alpha(j) * s - S_\alpha(j) * S_p(j)$. Vom avea $A_j \geq A_{j+1}$, deoarece șirul $\alpha[i]_{i=0}^{n-1}$ conține doar valori pozitive. Problema poate fi rezolvată iterând prin $j = N \ldots 1$ astfel:

- Query: calcularea valorii $dp_i = \min_{j=i+1}^{N} A_j \cdot S_p(i) + B_j$
- Add: Adaugă funcția (A_i, B_i).

Aceste operații sunt fix operațiile de Add și $Query$ a unor funcții liniare. Cum vom efectua N query-uri și N adăugări, iar complexitatea per query este $log_2 N$, complexitatea finală va fi $O(N \log_2 N)$. Există mai multe probleme în concursuri ce folosesc această optimizare, majoritatea bazându-se pe rescrierea recurenței pentru a reduce problema la operațiile de Add și Query.

V.2 Compararea timpului de execuție

Pentru $N = 10^5$ și numere generate aleator din intervalul $[0, 10^9]$, soluția în $O(N^2)$ necesită în medie aproximativ 11 secunde, iar cea în $O(N \log_2 N)$ necesită 0.1 secunde. Pentru $N = 10^6$ algoritmul în $O(N^2)$ depășește 50 de secunde, iar cel în $O(N \log_2 N)$ rulează în 0.5 secunde. Algoritmul optimizat rămâne în continuare foarte rapid pentru $N \leq 10^7$.

VI Concluzie

În această lucrare am prezentat trei metode de optimizare ce pot fi aplicate pentru diverse probleme de programare dinamică ce apar în concursuri dar și în practică. Fiecare metodă se poate aplica unor probleme ce îndeplinesc un anumit criteriu care a fost menționat și explicat cum poate fi verificat. După cum am văzut din analiza teoretică a complexității de timp, dar și din compararea implementărilor cu soluțiile neoptimizate, aceste metode reduc semnificativ timpul de execuție, demonstrând astfel eficiența lor.

Bibliografie

[1] Amir Ali Ahmadi. Linear programming. Accesat 14 Iunie 12:33 AM EEST http://www.princeton.edu/~aaa/Public/Teaching/ORF363_COS323/F15/ORF363_COS323_F15_Lec11.pdf.

[2] Takeshi Tokuyama Alok Aggarwal, Baruch Schieber. Finding a minimum weight k-link path in graphs with monge property and applications. *9th Annual Computational Geometry, /9/53 CA USA*, 9(1):189–197, 1993.

[3] Rainer E. Burkard, Bettina Klinz, and Rüdiger Rudolf. Perspectives of monge properties in optimization. *Discrete Applied Mathematics*, 70(2):95–161, 1996. URL: https://www.sciencedirect.com/science/article/pii/0166218X9500103X, doi:https://doi.org/10.1016/0166-218X(95)00103-X.

[4] Timothy M. Chan. Approximation Schemes for 0-1 Knapsack. In Raimund Seidel, editor, *1st Symposium on Simplicity in Algorithms (SOSA 2018)*, volume 61 of *OpenAccess Series in Informatics (OASIcs)*, pages 5:1–5:12, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2018/8299, doi:10.4230/OASIcs.SOSA.2018.5.

[5] Marek Cygan, Marcin Mucha, Karol Wegrzycki, and Michal Wlodarczyk. On problems equivalent to (min, +)-convolution. *CoRR*, abs/1702.07669, 2017. URL: http://arxiv.org/abs/1702.07669, arXiv:1702.07669.

[6] Sandip Das and Swami Sarvottamananda. Computing the minkowski sum of convex polytopes in ^{d}, 2018. arXiv:1811.05812.

[7] Thomas S. Ferguson. Linear programming: A concise introduction. 2004. Accesat 8 Iunie 2020 12:00 PM EEST http://web.tecnico.ulisboa.pt/mcasquilho/acad/or/ftp/FergusonUCLA_LP.pdf.

[8] JingsenChen Fredrik Bengtsson. Computing maximum-scoring segments optimally. *Computing in Science Engineering*, 1(1):1–12, 2007.

[9] Woburn C.I. Programming Enrichment Group. Convex hull trick. 2015. Accesat 6 Ianuarie 2021 12:00 PM EEST https://wcipeg.com/wiki/Convex_hull_trick.

[10] David A. Huffman. A method for the construction of minimum-redundancy codes*. 1952.

[11] Computing Maximum-Scoring Segments in Almost Linear Time. Fredrik bengtsson, jingsen chen. 2006.

[12] Shradha Kapoor. Batching problems with constraints. *UNLV Theses*, 2013. Accesat 21 Iunie 2020 12:00 PM EEST http://www.cs.cornell.edu/courses/cs6820/2010fa/handouts/matchings.pdf.

[13] Knuth. The art of computer programming. 3, 1998.

[14] Knuth. Searching an ordered table. 1998.

[15] K.Tone. Mathematical programming. 1978.

[16] S. Moran P. Shor A.Aggarwal R. Wilber, M.Klawe. Geometric applications of a matrix-search algorithms. *Algorithmica*, 1987.

[17] Yaron Singer. Am 221: Advanced optimization. 2016. Accesat 8 Iunie 2020 12:00 PM EEST `https://people.seas.harvard.edu/~yaron/AM221-S16/lecture_notes/AM221_lecture7.pdf`.

[18] Peter Brucker Sussane Albers. The complexity of one-machine batching problems. *MIT 18.438*, 7.