1. Se dau n cuburi cu laturile **diferite două câte două**. Fiecare cub are o culoare, codificată cu un număr de la 1 la p (p dat). Să se construiască un turn de înălţime maximă de cuburi în care un cub nu poate fi aşezat pe un cub de aceeaşi culoare sau cu latură mai mică decât a sa

Soluţie

Strategia Greedy pentru rezolvarea acestei probleme este următoarea: La primul pas se selectează cubul cu latura cea mai pare. Ulterior, la fiecare pas este selectat cubul cu latura cea mai mare care se poate aşeza peste ultimul cub selectat**.** Pentru aceasta vom ordona cuburile descrescător după latură.

Corectitudine - Similar cu problema spectacolelor (v. curs Greedy)

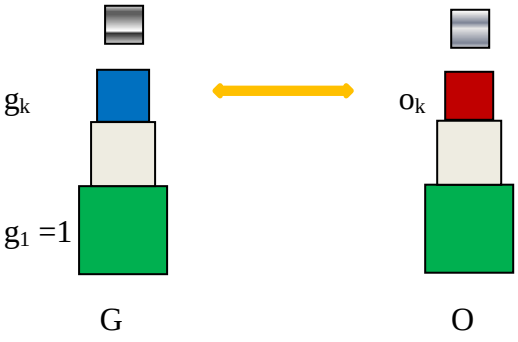
Notăm cu l_i, i=1..n laturile cuburilor şi cu c_i, i=1..n culorile cuburilor.

Renumerotăm cuburile astfel încât $l_1 > 1_2 > \ldots > l_n$. Astfel, primul cub selectat de algoritmul greedy este cubul 1.
Fie G = $\{g_1=1, \ldots, g_t\}$, $t \le p$ (cu $g_1 < \ldots < g_t$) soluţia Greedy.

Varianta 1 :
Considerăm o soluţie optimă O = $\{o_1, \ldots, o_p\}$ (cu cuburile notate astfel încât $o_1 < \ldots < o_p$) care are **un număr maxim de elemente iniţiale în comun cu soluţia Greedy** G. Presupunem O ≠ G.
Atunci există un indice $k \le t$ astfel încât $g_k \neq o_k$, altfel am avea $g_1 = o_1, \ldots, g_t = o_t$, adică $G \subseteq O$ şi $t < p$. Dar, deoarece cubul o_{t+1} este compatibil cu cubul $o_t = g_t$, algoritmul greedy ar mai fi avut cuburi compatibile cu g_t din care să selecteze, deci ar fi furnizat o soluţie cu mai multe elemente (un turn mai înalt).
Fie $k \le t$ cel mai mic indice astfel încât $g_k \neq o_k$ (prima poziţie pe care soluţia greedy G şi soluţia optimă O diferă). La pasul la care algoritmul a ales cubul g_k şi cubul o_k era neselectat şi putea fi aşezat peste cubul $g_{k-1} = o_{k-1}$. Deoarece cubul g_k a fost cel selectat, rezultă că $lg_k > lo_k$ (g_k a fost selectat deoarece avea latură mai mare). Avem atunci

$$lg_k > lo_k > lo_{k-1} > \ldots > lo_p$$

Dacă $c_{gk} = c_{ok}$ putem înlocui în soluţia optimă O cubul o_k cu g_k şi obţinem tot un turn corect O' = O – $\{o_k\} \cup \{g_k\}$, cu înălţime mai mare decât cel optim dat de O (deoarece $lg_k > lo_k$) , contradicţie.
Dacă $c_{gk} \neq c_{ok}$, atunci putem insera în turnul dat de O cubul g_k între $o_{k-1} = g_{k-1}$ şi o_k şi obţinem tot un turn corect O' = O $\cup \{g_k\}$, cu înălţime mai mare decât cel optim dat de O, contradicţie.



Varianta 2 (similar) – Demonstrăm prin inducţie după n că algoritmul greedy construieşte o soluţie optimă.

Pentru n = 0, 1 afirmaţia este evidentă.

Fie n ≥ 2. Presupunem că algoritmul greedy construieşte o soluţie optimă pentru orice mulţime de cel mult n - 1 cuburi

Fie S o mulţime de n cuburi.

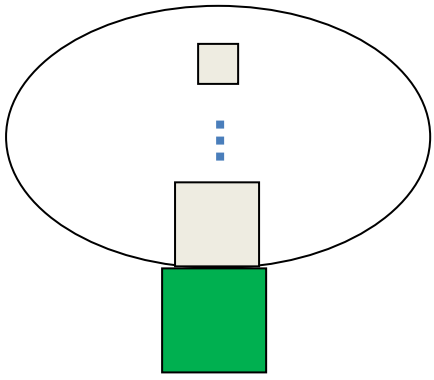
a) **Există o soluţie optimă pentru S care conţine cubul 1 (primul cub adăugat la soluţie de algoritmul greedy).**

Într-adevăr, fie O = $\{o_1,\ldots,o_p\}$ o soluţie optimă pentru S cu cuburile notate astfel încât $o_1 < \ldots < o_p$, deci $lo_1 > \ldots > lo_p$. Dacă cubul 1 aparţine lui O, atunci afirmaţia a) este adevărată. Presupunem că 1 nu aparţine lui O.

Dacă $c_1 = c_{o1}$ putem înlocui în soluţia optimă O cubul o_1 cu 1 şi obţinem tot un turn corect $O' = O - \{o_1\} \cup \{1\}$, cu înălţime mai mare decât cel optim dat de O (deoarece $l_1 > l_{o1}$) , contradicţie.

Dacă $c_1 \neq c_{o1}$ putem insera în turnul dat de O cubul 1 sub cubul o_1 şi obţinem tot un turn corect $O' = O \cup \{1\}$, cu înălţime mai mare decât cel optim dat de O, contradicţie.

b) Fie r primul cub având culoare diferită de cubul 1 (următorul cub ales de algoritmul greedy) şi S' = {r, r+1, …, n} (mulţimea cuburilor care pot aparţine unui turnul care are la bază cubul 1). Conform ipotezei de inducţie soluţia construită de algoritmul greedy pentru S', anume G' = G – {1}, este soluţie optimă pentru S'. Rezultă că $G = G' \cup \{1\}$ este soluţie optimă pentru S (altfel, conform punctului a) ar exista o soluţie optimă O pentru S care conţine cubul 1 cu înălţime mai mare decât G; dar atunci O - {1} este soluţie posibilă pentru S' cu înălţime mai mare decât G – {1} = G', ceea ce contrazice optimalitatea lui G')



2. Acoperire - Se dau un interval închis [a,b] si o mulţime de alte n intervale închise $[a_i,b_i]$, 1 ≤ i ≤ N. Scrieţi un program care determină şi afişează o submulţime de cardinal minim de intervale închise din mulţimea dată cu proprietatea că prin reuniunea acestora se obţine un interval care include pe [a,b]. Dacă prin reuniunea tuturor intervalelor nu putem obţine un interval care sa includă intervalul [a,b], se va afişa -1

Corectitudine - Similar cu problema spectacolelor (v. curs) şi problema 1 – cuburi

3. **(varianta 1.2) Planificare cu minimizarea întârzierii maxime** – Se consideră o mulţime de n activităţi care trebuie planificate pentru a folosi o aceeaşi resursă. Această resursă poate fi folosită de o singură activitate la un moment dat. Pentru fiecare activitate i se cunosc durata l_i şi termenul limită până la care se poate executa t_i (raportat la ora de început 0). Dorim să planificăm aceste activităţi astfel încât întârzierea fiecărei activităţi să fie cât mai mică. Mai exact, pentru o planificare a acestor activităţi astfel încât activitatea *i* este programată în intervalul de timp $[s_i, f_i)$, definim întârzierea activităţii *i* ca fiind durata cu care a depăşit termenul limită: $\boldsymbol{p_i = \max\{0, f_i - t_i\}}$.

Întârzierea planificării se defineşte ca fiind **maximul întârzierilor activităţilor**:
Exemplu. Pentru n = 3 şi $l_1 = 1$, $t_1 = 3$ / $l_2 = 2$, $t_2 = 2$ / $l_3 = 3$, $t_3 = 3$
o soluţie optimă se obţine dacă planificăm activităţile în ordinea 2, 3, 1; astfel:

- activitatea 2 în intervalul [0, 2) – întârziere 0
- activitatea 3 în intervalul [2, 5) – întârziere 5 – t_3 = 5 - 3 = 2
- activitatea 1 în intervalul [5, 6) – întârziere 6 – t_1 = 6 - 3 = 3

Întârzierea planificării este max{0, 2, 3}=3 $\boldsymbol{P = \max\{p_1, p_2, ..., p_n\}}$

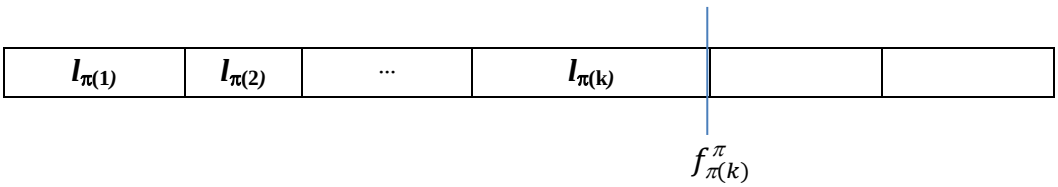
Soluţie
O *soluţie* (stabilirea ordinii în care se vor planifica activităţile) înseamnă o permutare $\pi \in S_n$

Putem considera doar soluţii în care activităţile sunt planificate fără a lăsa momente de timp neutilizate (libere) între activităţi, altfel activităţile programate după intervalele de timp libere ar putea fi programate mai devreme fără a lăsa astfel de intervale şi întârzierea noii planificării va fi cel puţin la fel de mică.

Astfel, corespunzător soluţiei π, timpul de finalizare pentru activitatea $\pi(k)$ (k=1,...,n) va fi

$$f^{\pi}_{\pi(k)} = l_{\pi(1)} + \ldots + l_{\pi(k)}$$ (v. desenul de mai jos)

Pentru k > 1 avem $f^{\pi}_{\pi(k)} = f^{\pi}_{\pi(k-1)} + l_{\pi(k)}$



Pentru u=1,..,n notăm cu $p^{\pi}_u = \max\{0, f^{\pi}_u - t_u\}$ întârzierea unei activităţi u în planificarea corespunzătoare permutării π şi cu $P^{\pi} = \max_{1 \le i \le n} p^{\pi}_i$ întârzierea planificării π.

Strategia Greedy pentru rezolvarea acestei probleme este următoarea: Se planifică activităţile în ordine crescătoare după termenul limită t_i.

Corectitudine-(v.şi curs Greedy-problema minimizării timpului mediu de acces/texte pe bandă)

Renumerotăm activităţile astfel încât $t_1 \le t_2 \le \ldots \le t_n$. Astfel, soluţia greedy corespunde permutării identice id. Intervalul de desfăşurare a activităţi k conform soluţiei greedy este $[l_1+\ldots+ l_{k-1}, l_1+\ldots+ l_k)$.

Presupunem prin absurd că soluţia greedy (permutarea id) nu este optimă.

Fie σ o permutare optimă cu număr minim de inversiuni (cea mai apropiată de soluţia greedy).

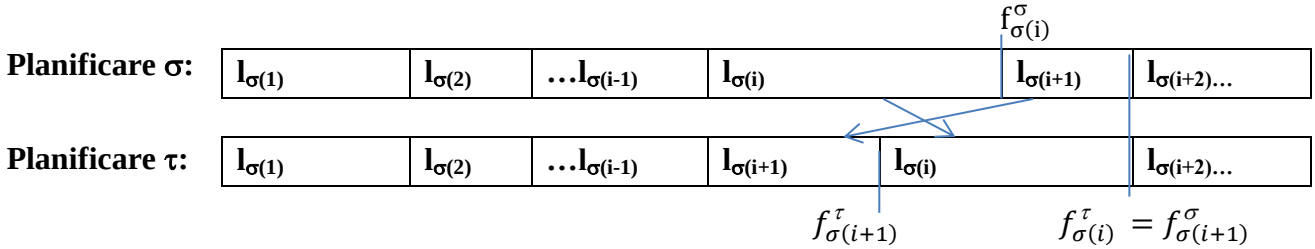
Deoarece id nu este optimă, rezultă că σ conţine o inversiune (i,j) cu $i < j$ şi $t_{\sigma(i)} > t_{\sigma(j)}$. Mai mult, există o astfel de inversiune în care j=i+1 (dacă $t_{\sigma(1)}, \ldots, t_{\sigma(n)}$ nu este ordonat crescător, există două elemente alăturate care nu sunt în această ordine).

Considerăm τ permutarea obţinută din σ interschimbând elementele de pe poziţiile i şi i+1 (permutarea τ va avea astfel mai puţine inversiuni, deci va fi mai asemănătoare cu soluţia greedy id decât σ). Avem atunci:

$\tau(k) = \sigma(k) \quad \forall k \neq i, i+1$

$\tau(i) = \sigma(i+1)$

$\tau(i+1) = \sigma(i)$



Comparăm întârzierea corespunzătoare permutării τ cu cea a soluţiei optime σ (pentru a arăta că τ are întârzierea mai mică sau egală cu a lui σ, dar are mai puţine inversiuni decât σ, ceea ce contrazice alegerea lui σ)

Deoarece $\tau(k) = \sigma(k) \quad \forall k \neq i, i+1$, singurele activităţi care pot avea întârzieri diferite în σ şi τ sunt σ(i) şi σ(i+1).

Pentru simplitate vom nota $u := \sigma(i)$ şi $v := \sigma(i+1)$. Avem (v. si desen)

$f_u^{\sigma} = l_{\sigma(1)} + \cdots + l_{\sigma(i-1)} + l_{\sigma(i)}$

$f_u^{\tau} = l_{\sigma(1)} + \cdots + l_{\sigma(i-1)} + l_{\sigma(i+1)} + l_{\sigma(i)}$

$f_v^{\sigma} = l_{\sigma(1)} + \cdots + l_{\sigma(i-1)} + l_{\sigma(i)} + l_{\sigma(i+1)} = f_u^{\tau}$

$f_v^{\tau} = l_{\sigma(1)} + \cdots + l_{\sigma(i-1)} + l_{\sigma(i+1)}$

Deoarece $f_v^{\tau} = f_u^{\tau} - l_{\sigma(i)} = f_v^{\sigma} - l_{\sigma(i)} < f_v^{\tau}$ (activitatea $v = \sigma(i+1)$ este programată în τ mai devreme decât este în σ) este suficient să mai comparăm întârzierile activităţii $u = \sigma(i)$ în

permutările σ respectiv τ, în cazul în care activitatea u are întârziere în τ (altfel în τ nu creşte întârzierea niciunei activităţi faţă de σ, deci τ este tot o planificare optimă).
Întârzierea activităţii *u* în τ este

$$p_u^\tau = f_u^\tau - t_u = f_v^\sigma - t_u$$

Dar $t_{\sigma(i)} > t_{\sigma(i+1)}$ (sau, cu notatiile u si v, $t_u > t_v$), de unde rezultă

$$p_u^\tau = f_u^\tau - t_u = f_v^\sigma - t_u < f_v^\sigma - t_v \leq p_v^\sigma \leq \mathrm{P}^\sigma = \max_{1 \leq i \leq n} p_i^\sigma$$

(întârzierea lui $u = \sigma(i)$ în τ este mai mică decât întârzierea lui $v = \sigma(i+1)$ în σ deoarece se termină la acelaşi moment, dar u are termenul limită mai mare). Deci

$$\mathrm{P}^\pi = \max_{1 \leq i \leq n} p_i^\pi < \mathrm{P}^\sigma$$

ceea ce contrazice optimalitatea lui σ, deci presupunerea este falsă. Rezultă că id = σ este permutare optimă.

4. (varianta 3.3) Problema partiţionării intervalelor – Se consideră n intervale închise. Se cere să se împartă (partiţioneze) această mulţime de intervale într-un **număr minim de submulţimi** cu proprietatea că oricare două intervale dintr-o submulţime nu se intersectează şi să se afişeze aceste submulţimi

Soluţie

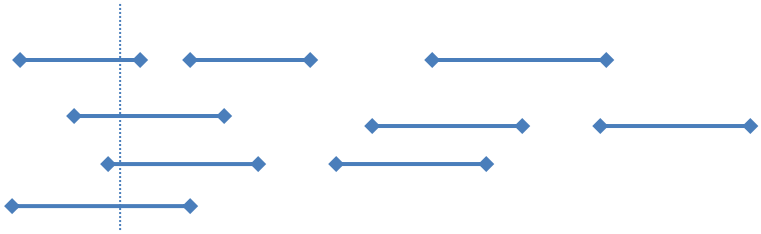
Algoritm greedy: Se sortează intervalele crescător după extremitatea iniţială. Pentru fiecare interval I în această ordine execută: se adaugă I la o submulţime deja construită, dacă se poate (nu se intersectează cu niciun interval din ea), altfel se creează o nouă submulţime cu intervalul I.

Corectitudine

Fie S mulţimea de intervale dată.

Definiţie: Numim *adâncimea lui S* numărul maxim de intervale din *S* care se suprapun peste un acelaşi punct (care au toate un acelaşi punct)

Exemplu



Adâncimea mulţimii intervalelor de mai sus este 4 (există cel mult 4 intervale care se suprapun peste un acelaşi punct).
Pentru simplificarea prezentării vom asocia fiecărei submulţimi o etichetă (un număr). Astfel, problema se reduce la a eticheta intervalele cu un număr minim de etichete astfel încât două intervale cu aceeaşi etichetă să fie disjuncte.

Vom demonstra corectitudinea în 3 etape:

1) Soluţia construită de algoritm este corectă (posibilă), deoarece toate intervalele sunt etichetate (adăugate într-o submulţime) şi prin modul de alegere al etichetelor, intervalele cu aceeași etichetă sunt disjuncte două câte două (un interval este adăugat la o submulțime doar dacă nu se intersectează cu alt interval din submulţime)

2) **Propoziție**: Numărul minim de etichete necesare etichetării intervalelor din S este mai mare sau egal cu adâncimea lui S.

Demonstrație:

Notăm cu *d* adâncimea lui *S*. Din definiție rezultă că există o mulțime de intervale I_1, I_2,...,I_d ce se suprapun peste un același punct. Deoarece nu putem folosi aceeaşi etichetă pentru două sau mai multe intervale care se suprapun, rezultă că cele *d* intervale anterior menționate trebuie să aibă etichete distincte. Astfel, pentru etichetarea corectă a *tuturor* intervalelor din *S* trebuie să folosim minim *d* etichete.

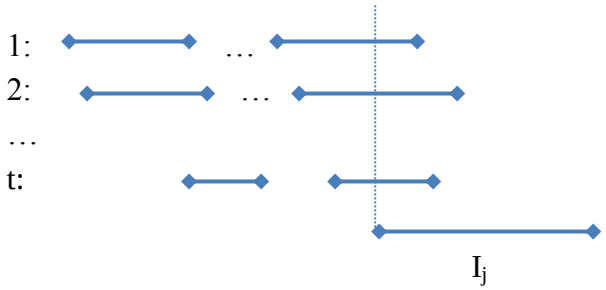
3) Vom demonstra că algoritmul Greedy folosește exact d etichete, de unde va rezulta folosind propoziția că numărul de etichete folosite de algoritm (deci de submulţimi generate) este minim.

Fie t numărul de etichete folosite de algoritm la pasul curent. Trebuie să demonstrăm că la fiecare pas $t \leq d$ (în final va avea loc chiar egalitatea). Iniţial t=0.

Fie I_j intervalul care este etichetat la pasul curent.

Dacă lui I_j i se poate asocia o etichetă deja existentă (se poate adăuga la o submulţime existentă) atunci t nu se modifică şi inegalitatea rămâne valabilă.

Dacă lui I_j nu i se poate asocia o etichetă deja existentă (nu se poate adăuga la o submulţime existentă), atunci există *t* intervale cu etichete distincte două câte două (câte unul din fiecare submulţime deja construită) cu extremitatea iniţială mai mică sau egală decât cea a lui I_j ce se intersectează cu intervalul curent.



Din faptul că toate cele *t* intervale au extremitatea iniţială mai mică sau egală decât cea a lui I_j, rezultă că există cel puţin un punct în care toate cele t+1 intervale (incluzând intervalul I_j) se suprapun, şi anume acest punct este chiar extremitatea iniţială a lui I_j. Rezultă că $t+1 \leq d$. Deci, chiar dacă pentru I_j este folosită o a *t+1* etichetă (este creată o nouă submulţime), în total algoritmul nu va folosi mai mult de *d* etichete.

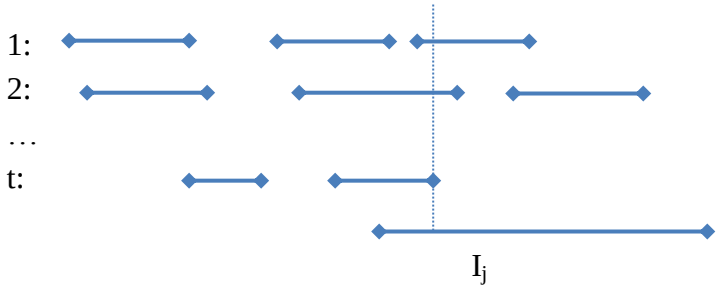
Din 1), 2) şi 3) rezultă că algoritmul găsește o etichetare optimă.

Punctul c) Se sortează intervalele crescător după extremitatea finală Pentru fiecare interval I în această ordine execută: se adaugă I la o mulţime deja construită, dacă se poate, altfel se creează o nouă mulţime cu intervalul I. Dacă există mai multe mulţimi la care se poate adăuga I, se alege acea submulţime care conţine intervalul cu extremitatea finală cea mai mare (care se termină cât mai aproape de începutul intervalului I)

Corectitudine – Indicaţie Ca şi la a), considerăm t numărul de etichete folosite de algoritm la pasul curent (numărul de submulţimi construite până la acel pas). Trebuie să demonstrăm că la fiecare pas $t \leq d$ (în final va avea loc chiar egalitatea). Fie I_j intervalul care este etichetat la pasul curent.

Este suficient să demonstrăm cazul în care lui I_j nu i se poate asocia o etichetă deja existentă şi i se asociază eticheta t+1 (este creată o nouă submulţime cu intervalul Ij).

Ca şi la a), dacă lui I_j nu i se poate asocia o etichetă deja existentă (nu se poate adăuga la o submulţime existentă), atunci există *t* intervale cu etichete distincte două câte două (câte unul din fiecare submulţime deja construită) ce se intersectează cu intervalul curent I_j, şi care au extremitatea finală mai mică sau egală cu cea a lui I_j . Trebuie să evidenţiem însă t intervale care au un acelaşi punct comun cu Ij. Considerăm din fiecare submulţime intervalul cu extremitatea finală maximă (ultimul adăugat). Considerăm m cea mai mică extremitate finală a unui astfel de interval. Se demonstrează că există câte un interval în fiecare sală care conţine m şi în plus m aparţine şi lui Ij, deci există t+1 intervale care se suprapun în m. Rezultă că $t+1 \leq d$.



5. (tema varianta 1.3.a) Se dă un şir *v* de numere naturale. Să se descompună şirul de numere într-un număr minim de subşiruri descrescătoare (nestrict) şi să se afişeze aceste subşiruri.

Algoritm Greedy – v. şi slideurile din arhivă Se parcurge şirul element cu element. Elementul curent v[i] se adaugă la subşirul care se termină cu cel mai mare element mai mic decât v[i], dacă un astfel de subşir există; altfel se creează un subşir nou cu elementul v[i]

Observaţii legate de implementare şi notaţii. Notăm *ns* numărul de subşiruri construite de algoritm. Vom numi subşirurile construite şi stive (prin analogie cu Solitaire)

Iniţial *ns* = 0.

La **pasul i** al algoritmului sunt construite deja ns = k subșiruri descrescătoare cu elementele șirului $v[1 \dots i-1]$.
Notăm cu $u[i]$, $1 \leq i \leq k$ ultimul element adăugat în al i-lea șir descrescător (vârful stivei i).
Avem că $u[1] < \cdots < u[k]$
La **pasul i** se caută (binar) $1 \leq j \leq k+1$ astfel încât să avem: $u[j-1] < v[i] < u[j]$ (prin convenție $u[0] = -\infty$ și $u[k+1] = \infty$). Dacă $j = k+1$ atunci se începe un nou subșir care va conține $v[i]$ și $ns = k+1$. Altfel, se adaugă $v[i]$ la șirul j, deci u[j] devine $u[j] = v[i]$.

Demonstrarea corectitudinii.

Propoziție: Fie $s = (v_{i_1}, v_{i_2}, \dots, v_{i_{ls}})$ $i_1 \leq i_2 \dots \leq i_{ls}$ și $v_{i_1} < \cdots < v_{i_{ls}}$ un subșir crescător al lui s. Orice descompunere a lui v în subșiruri descrescătoare conține cel puțin ls subșiruri.

Demonstrație:

Dacă ar exista o descompunere cu $ls-1$ subșiruri, două numere din s ar fi incluse în același subșir. Atunci acest subșir nu ar fi ordonat descrescător.

Consecinţă: O descompunere în subşiruri descrescătoare conține cel puțin $lmax$ subșiruri, unde $lmax$ este lungimea maximă a unui subșir crescător al lui s.

Vom demonstra că algoritmul Greedy construieşte cel mult $lmax$ subşiruri (stive), de unde va rezulta folosind consecinţa propoziției că numărul de subşiruri construite de algoritm este minim.
Pentru aceasta este suficient să demonstrăm că:
(*) *Ultimul termen al subșirului ns din descompunere (vârful ultimei stive), aparține unui șir crescător al lui v care are lungimea ns.*

Demonstrăm (*) prin inducție după ns.
Dacă $ns = 1$ afirmația este adevărată.
Presupunem că dacă algoritmul împarte un șir w în $ns-1$ subșiruri, ultimul termen al stivei $ns-1$, aparține unui șir crescător al lui w care are lungimea $ns-1$.
Considerăm subșirurile construite de algoritm **până la momentul la care urmează să fie adăugat elementul x = u[ns] la ultimul subşir (până la pasul la care x = u[ns] devine elementul din v curent care trebuie adăugat unui subşir).**
Considerăm subşirul w al lui v format cu elementele subșirurilor (stivelor) $1..ns-1$ (în ordinea în care apăreau în v). Conform ipotezei de inducţie w are un subşir crescător care se termină cu $u[ns-1]$:

$$s = (v_{i_1}, v_{i_2}, \dots, v_{i_{ns-2}}, u_{ns-1}).$$

Deoarece elementul curent x= $u[ns]$ se adaugă în stiva ns, înseamnă că aveam $u[ns-1] < u[ns]$, altfel $u[ns]$ ar fi fost adăugat la stiva $ns-1$. Atunci șirul $s' = (v_{i_1}, v_{i_2}, \dots, v_{i_{ns-2}}, u_{ns-1}, u_{ns})$ este subșir crescător al lui v care are lungimea ns.