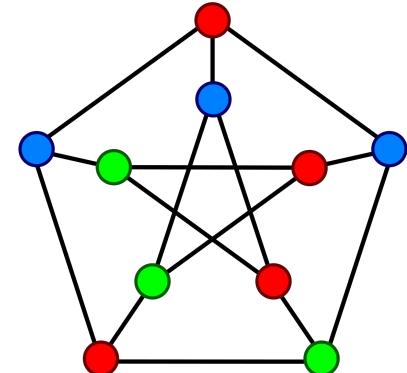
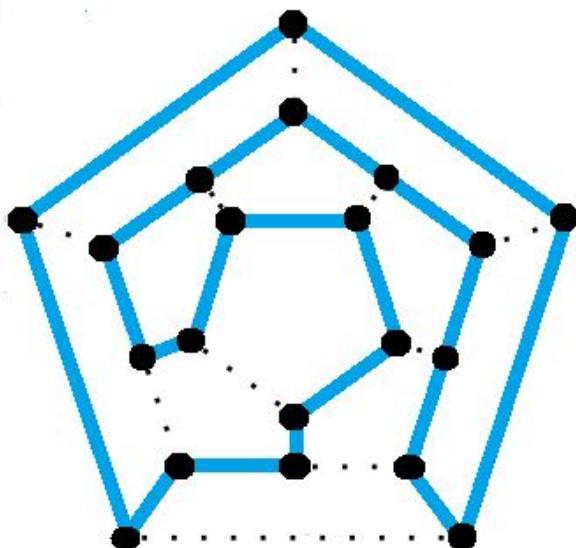
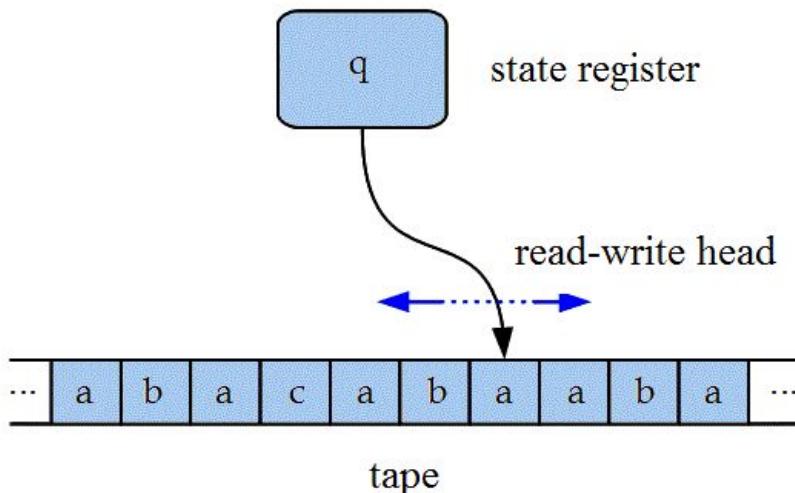

Timp polinomial: Determinism vs nedeterminism

5	9		4	
7	8	3	4	9
6	1			7 3
4	6	2	5	
3	8	5	7 2	6 4 9
1	7	4	8	2
2		1		4
	3	4		8 7
7		5 3		6



Scurtă prezentare: "Turing Machine"



O mașină Turing $M=(Q, \Gamma, b, \Sigma, \rho, q_0, F)$ unde:

- Q - multimea stărilor
- Γ - alfabetul de lucru al mașinii
- $b \in \Gamma$ - un simbol special, numit "blank"
- $\Sigma \subset \Gamma \setminus \{b\}$ - alfabetul de intrare (alfabetul pt input)
- q_0, F - starea inițială, respectiv multimea stărilor finale

ρ - funcția de tranziție:

cazul determinist: $\rho: \Gamma^* \times Q \rightarrow \Gamma^* \times Q \times \{\text{left}, \text{right}\}$

cazul nedeterminist: $\rho: \Gamma^* \times Q \rightarrow 2^{\Gamma^* \times Q \times \{\text{left}, \text{right}\}}$

Clasele de Complexitate P și NP

Formal spus, în clasa problemelor din P sunt acele probleme care pot fi rezolvate în timp polinomial, $O(n^c)$, de către un sistem determinist. (P=polynomial)

Iar cele din clasa NP sunt problemele care pot rezolvațe tot în timp polinomial (!) dar de către o mașină Turing nedeterminista. (NP=nondeterministic Polynomial)

Evident ca $P \subset NP$.

Se presupune ca $P \neq NP$, totuși încă nu există o demonstrație a acestui rezultat.

Clasele de Complexitate P și NP

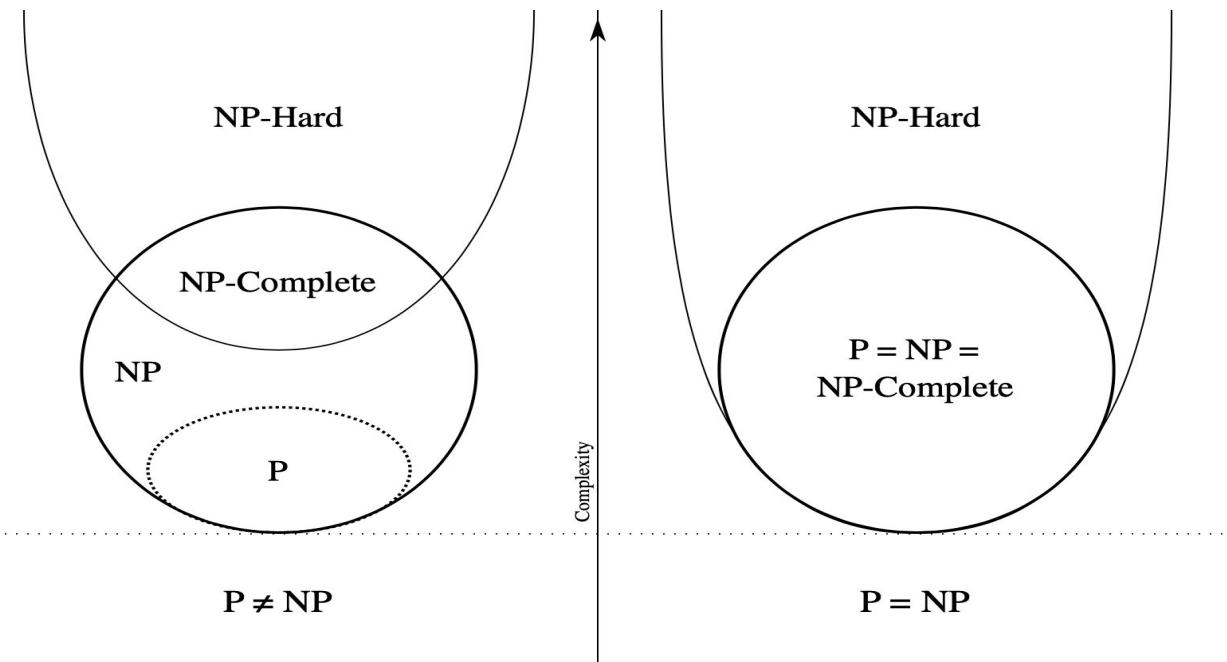
Mai ușor de înțeles:

P - clasa de probleme pentru care le putem afla soluția în timp polinomial

NP - clasa de probleme pentru care **putem verifica** în timp polinomial dacă un rezultat este soluție corectă pentru problema noastră.

5	9			4			
7	8	3	4	9			
6	1				7	3	
4	6	2	5				
3	8	5	7	2		6	4
1	7	4		8	2		
2			1				4
		3		4		8	7
7			5	3			6

Clasele de Complexitate P, NP, NP-C



[Source for further reading](#)

Ce ne facem cu problemele din NP

Avem la dispoziție mașini (calculatoare)
nedeterministe?

În cazul problemelor de optim există două soluții:
Plătim costul în timp (de multe ori nu se poate) sau
ne mulțumim cu o soluție apropiată de optim, dacă
nu optimă, dar ce se poate obține în timp fezabil.

Probleme de optim

O problemă de optim este de forma următoare:
Fie o mulțime de restricții. Să se construiască o
soluție care nu doar îndeplinește toate restricțiile, ci
minimizează/maximizeze o funcție de cost/profit.

Ex: Problema rucsacului (varianta discretă) sau
probleme de acoperire minimală pentru grafuri.

Probleme de optim

Fie OPT soluția optimă a problemei. Ea poate fi obținută foarte greu (practic imposibil) Două dintre căile de atac pentru astfel de probleme ar fi:

- avem un algoritm care construiește pe rand soluții la problemă, din ce în ce "mai optime", care converg către OPT. Lăsăm acest algoritm să ruleze un timp rezonabil, sau până când rezultatul nu se mai poate îmbunătăți și ne multumim cu ce avem.

(algoritmi evoluționiști)

Probleme de optim

Fie OPT soluția optimă a problemei. Ea poate fi obținută foarte greu (practic imposibil) Două dintre căile de atac pentru astfel de probleme ar fi:

- fie cazul în care OPT trebuie să minimizeze un cost.

Să reușim să contruim o soluție ALG, cu

$$\text{OPT} \leq \text{ALG} \leq \rho \times \text{OPT}$$

(algoritmi ρ -aproximativi)

Cursul prezent

- Terminologie de baza
- Un prim exemplu de algoritm aproximativ
- Un exemplu mai detaliat
- Un început pt Tema 1



Motivătie

Q: Daca avem nevoie să aflăm răspunsul la o problemă NP-hard?

A: Nu prea sunt șanse să găsim un algoritm care să ruleze în timp polinomial

Așa că....

Motivătie

Trebuie să renunțăm măcar la unul dintre următoarele 3 elemente:

1. Găsirea unui algoritm polinomial pentru problemă
2. Găsirea unui algoritm general (pentru o instanță oarecare) a problemei
3. Găsirea soluției exacte (optime) pentru problema

Basic Terminology & Notations

Problema de Optim:

Informal spus este problema in care trebuie sa gasesti o “cea mai buna” solutie/constructie fezabila.

“Cea mai buna” - poate avea doua sensuri:

Fie avem o problema de **minimizare** precum Problema Comis-voiajorului.

Fie o problema de **maximizare** precum cea de a găsi o acoperire de cardinal maxim pentru multimea varfurilor unui graf

Basic Terminology & Notations

Problema de Optim:

Fie P - o problema de optim, și I o intrare pe aceasta problema. Vom nota cu $OPT(I)$ "valoarea" soluției optime.

În mod analog, atunci când propunem un algoritm care să ofere o soluție fezabilă pentru problema noastră, vom nota "valoarea" acelei soluții cu $ALG(I)$.

De cele mai multe ori, atunci când nu se crează confuzie, vom simplifica notațiile folosind termenii "OPT", respectiv "ALG".

Pe parcursul prezentării vom presupune că atât OPT , cât și ALG sunt ≥ 0 .

Basic Terminology & Notations

Problema de Optim:

Pentru a justifica ca un algoritm este *util*, acesta trebuie însotit de o justificare că soluția oferită este fezabilă pentru problema, precum și o relație între ALG și OPT . Aceast tip de relație este descrisă astfel:

Definiție 1

- Un algoritm ALG pentru o problema de **minimizare** se numește ρ -aproximativ, pentru o valoare $\rho > 1$, dacă $ALG(I) \leq \rho \cdot OPT(I)$ pt $\forall I$ – intrare
- Un algoritm ALG pentru o problema de **maximizare** se numește ρ -aproximativ, pentru o valoare $\rho < 1$, dacă $ALG \geq \rho \cdot OPT(I)$ pt $\forall I$ – intrare

Basic Terminology & Notations

OBSERVAȚIE

(pt probleme de minim) Orice algoritm ρ -aproximativ este la rândul lui ρ' -aproximativ pentru orice $\rho' > \rho$. De aceea, în cazul unui algoritm ALG pentru o problemă de minimizare, spre exemplu, trebuie ca justificarea ce însوtește pe ALG să ofere cea mai mică valoare ρ pentru care ALG este ρ -aproximativ.

Definiție 1

- Un algoritm ALG pentru o problema de **minimzare** se numește ρ -aproximativ, pentru o valoare $\rho > 1$, dacă $ALG(I) \leq \rho \cdot OPT(I)$ pt $\forall I$ – intrare
- Un algoritm ALG pentru o problema de **maximizare** se numește ρ -aproximativ, pentru o valoare $\rho < 1$, dacă $ALG \geq \rho \cdot OPT(I)$ pt $\forall I$ – intrare

Basic Terminology & Notations

Definiție 2

Fie ALG un algoritm ρ -aproximativ pentru o problema de minimizare. Spunem că factorul de aproximare este "tight bounded" atunci când avem $\rho = \sup_{\mathcal{I}} \frac{ALG(\mathcal{I})}{OPT(\mathcal{I})}$

Ca să arătăm că un algoritm este ρ -aproximativ "tight bounded", trebuie deci să justificăm următoarele 2 lucruri:

1. Trebuie să arătmăm că este ρ -aproximativ, adică $ALG(\mathcal{I}) \leq \rho \times OPT(\mathcal{I})$ pentru orice intrare \mathcal{I}
2. Pentru orice $\rho' < \rho$ există un \mathcal{I} pentru care $ALG(\mathcal{I}) > \rho' \times OPT(\mathcal{I})$. Adesea totuși ne este mai la îndemână să arătăm ca există un \mathcal{I} pentru care $ALG(\mathcal{I}) = \rho \times OPT(\mathcal{I})$

O primă provocare: 1/o Knapsack problem

Enunț pe scurt: Trebuie să găsim o submulțime de obiecte de valoare totală maximă, fără ca greutatea lor totală să depășească o capacitate dată a rucsacului. Obiectele sunt puse integral în rucsac sau sunt date deoparte. Nu pot fi fracționate!

Rezolvare propusă:

Fie L – lista obiectelor sortate după raportul valoare/greutate

Fie O_p – obiectul cu profitul cel mai mare din lista de obiecte.

$S=0$, $G=\text{capacitatea rucsacului}$;

Pentru fiecare $O:L$

Dacă $\text{greutate}(O) \leq G$, atunci $S += \text{val}(O)$, $G -= \text{greutate}(O)$

$$\text{ALG}(I) = \max(S, O_p)$$

O primă provocare: 1/0 Knapsack problem

Demonstrați că algoritmul de mai jos este un algoritm 1/2-aproximativ pentru problema 1/0 a Rucsacului!

Rezolvare propusă:

Fie L – lista obiectelor sortate după raportul valoare/greutate

Fie O_p – obiectul cu profitul cel mai mare din lista de obiecte.

$S=0$, $G=\text{capacitatea rucsacului}$;

Pentru fiecare $O:L$

Dacă $\text{greutate}(O) \leq G$, atunci $S += \text{val}(O)$, $G -= \text{greutate}(O)$

$$\text{ALG}(I) = \max(S, O_p)$$

O primă provocare: 1/0 Knapsack problem

Demonstrați că algoritmul de mai jos este un algoritm 1/2-aproximativ pentru problema 1/0 a Rucsacului! [Justificare](#)

Rezolvare propusă:

Fie L – lista obiectelor sortate după raportul valoare/greutate

Fie O_p – obiectul cu profitul cel mai mare din lista de obiecte.

$S=0$, $G=\text{capacitatea rucsacului}$;

Pentru fiecare $O:L$

Dacă $\text{greutate}(O) \leq G$, atunci $S += \text{val}(O)$, $G -= \text{greutate}(O)$

$$\text{ALG}(I) = \max(S, O_p)$$

Load Balancing Problem

Input:

- m calculatoare identice; n activitați ce trebuie procesate. Fiecare activitate j având nevoie de t_j unități de timp pentru execuție.
- Odată inițiată, fiecare dintre activitați trebuie derulată în mod continuu pe același calculator
- Un calculator poate executa cel mult o activitate în același timp.

Scop:

Să asignăm fiecare activitate unui calculator astfel încât să minimizăm timpul până când toate activitățile sunt terminate.

Load Balancing Problem

Notății:

- $J(i)$ - submulțimea tuturor activităților (job-urilor) care au fost programate să se desfășoare pe mașina i .
- L_i va reprezenta "load-ul" (timpul de lucru) al mașinii i .
- $L_i = \sum_{j \in J(i)} t_j$

Scop: O asignare a activităților astfel încât L_k este minimizat, unde $k = \max_i(L_i)$, adică mașina cu cel mai mare load.

Load Balancing Problem

Pseudocodul:

$Load - Balance(m, t_1, t_2, \dots, t_n)$

for $i = 1$ to m :

$L_i = 0; J(i) = \emptyset$ # initializare: Fiecare Load este 0 iar multimea joburilor este nula pt fiecare masina

for $j = 1$ to n :

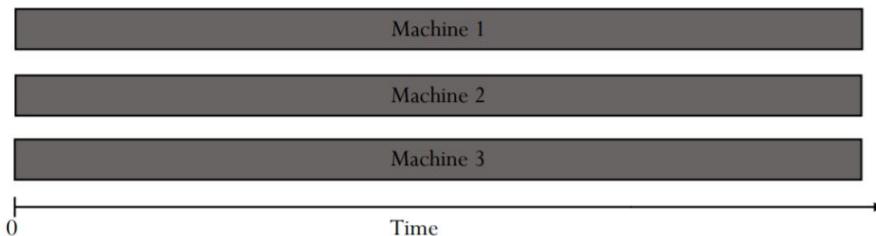
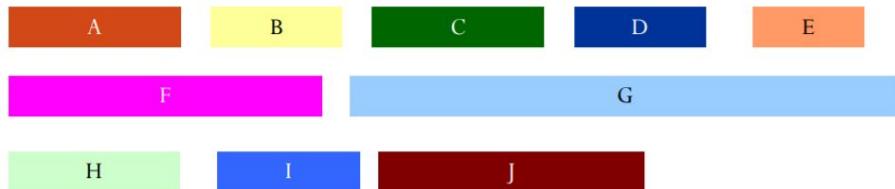
$i = \arg\left(\min\{L_k \mid k \in \{1, \dots, m\}\}\right)$ # i – masina cu incarcatura cea mai mica in acest moment

$J(i) = J(i) \cup \{j\}$

$L_i += t_j$

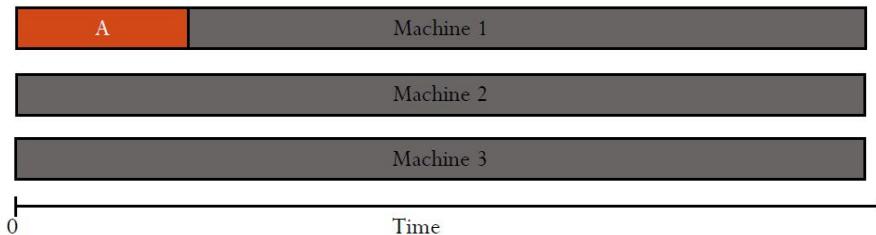
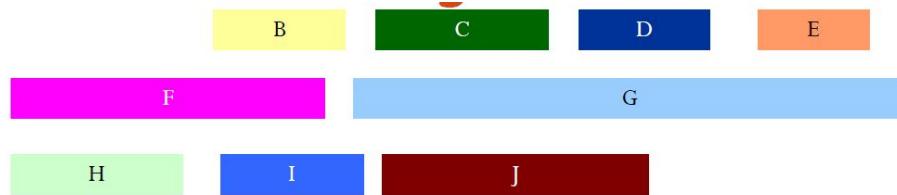


Step-by-step example



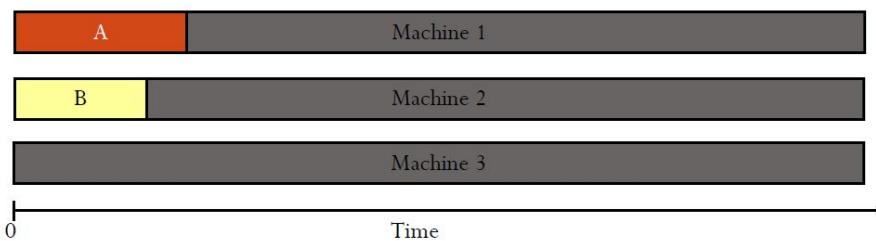
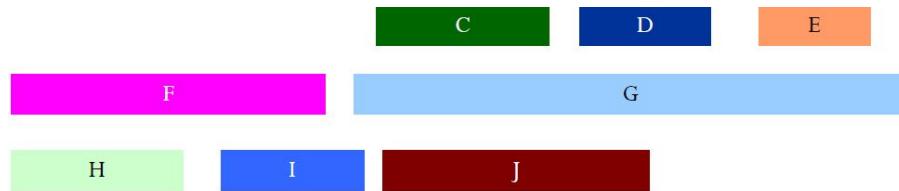


Step-by-step example



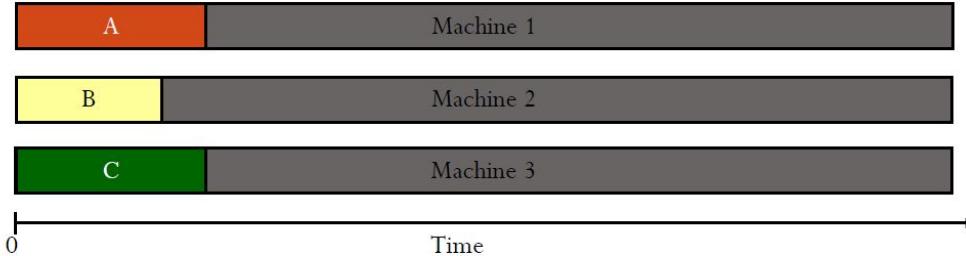
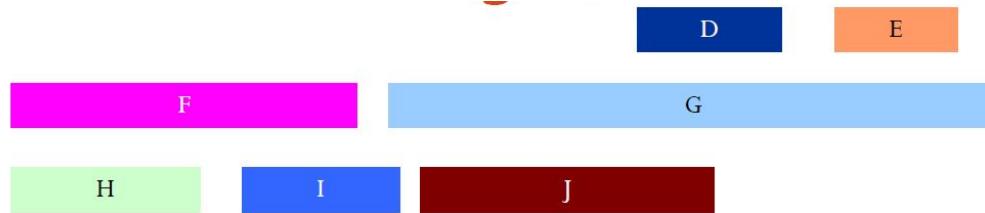


Step-by-step example



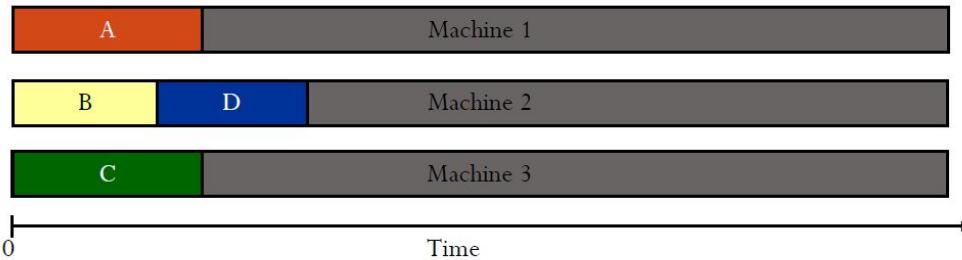
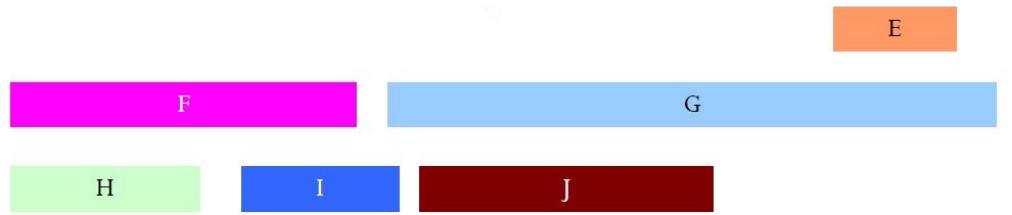


Step-by-step example



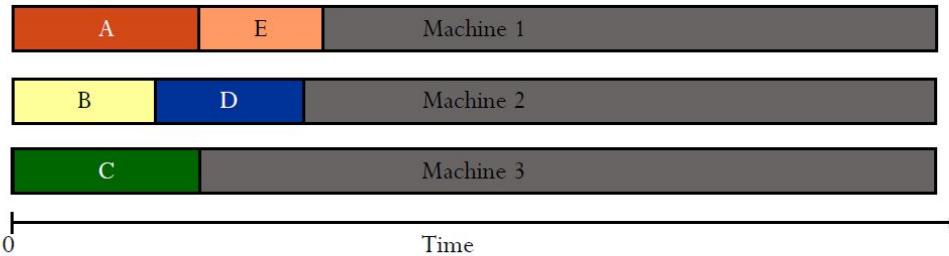


Step-by-step example



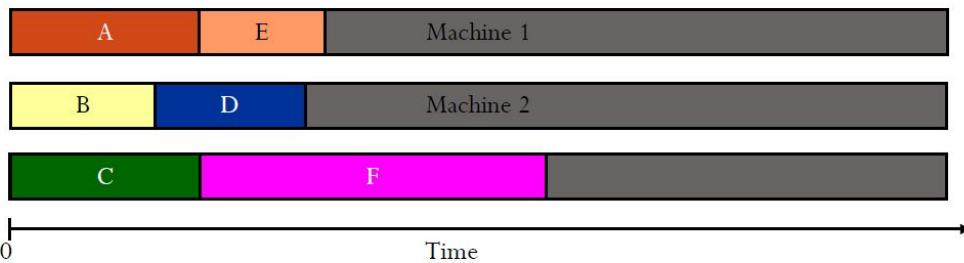


Step-by-step example





Step-by-step example



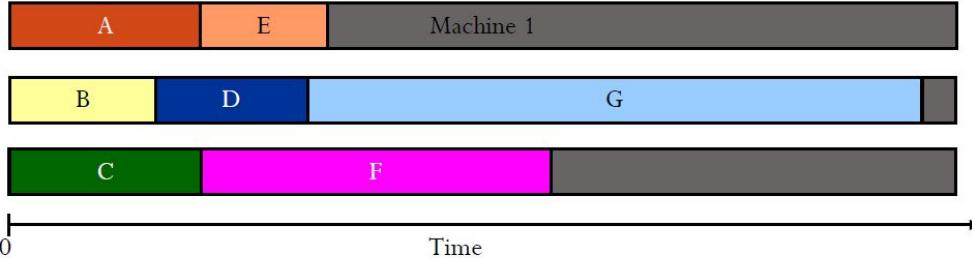


Step-by-step example

H

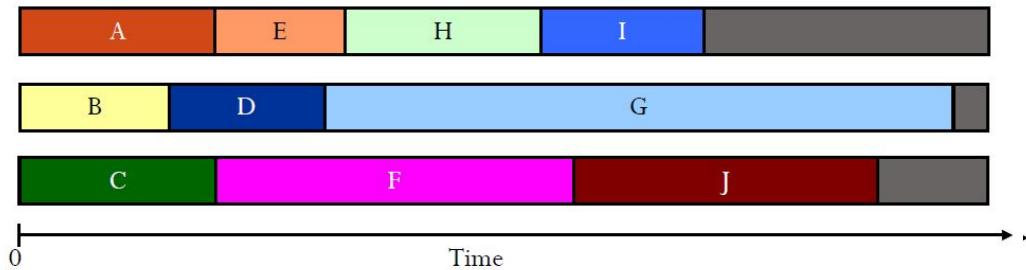
I

J





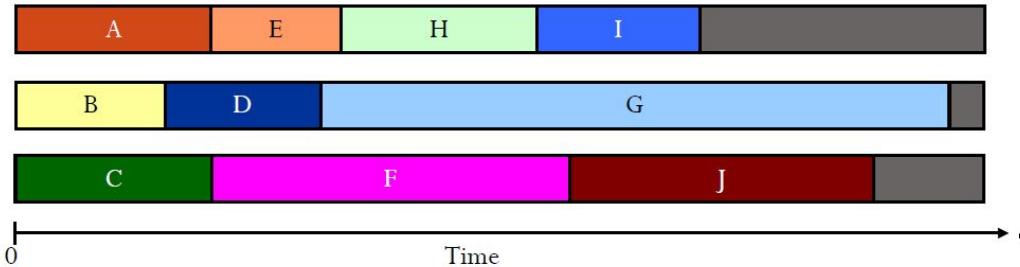
Step-by-step example (3 steps)

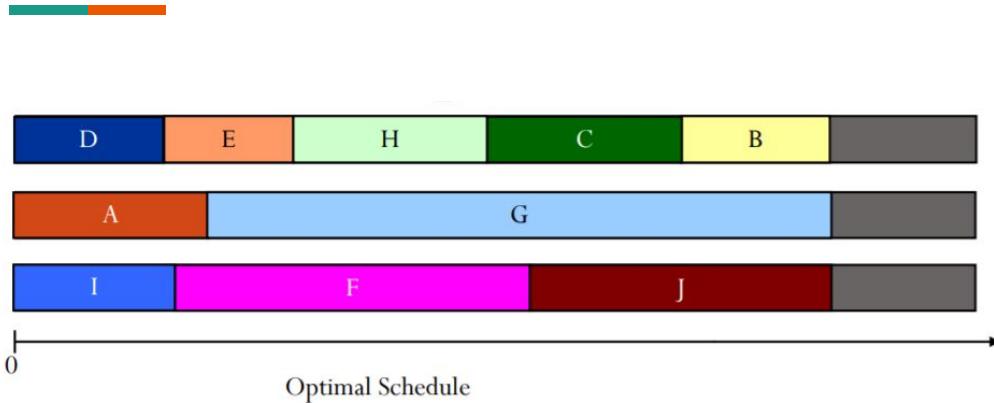


Step-by-step example (3 steps)

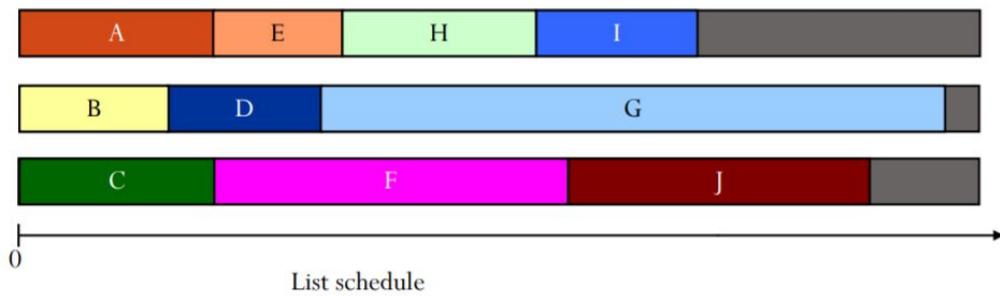


Este Optim?





NU



Care este Factorul de Aproximare?

Lema 1.

$$OPT \geq \max \left\{ \frac{1}{m} \sum_{1 \leq j \leq n} t_j, \max \left\{ t_j \mid 1 \leq j \leq n \right\} \right\}$$

Lema 2.

Algoritmul descris anterior este un algoritm 2-Aproximativ.

Altfel spus, fie $ALG = \max(L_i \mid i \in \{1, \dots, m\})$ masina "cea mai incarcata". Avem de arătat că $ALG \leq 2 \times OPT$

Justificari

Se poate imbunătății LB-ul?

3 abordari:

- a) Acelasi Algoritm, o analiză mai buna asupra lower-bound-ului folosit
Teorema: Algoritmul Greedy descris anterior este un algoritm $2 - 1/m$ aproximativ
- b) Acelasi Algoritm, gasirea unui alt lower bound folosind alte inegalități
Nu se poate! m mașini, $m(m-1)$ activități de cost 1 și o activitate de cost m
- c) Un cu totul alt Algoritm care poate da un total alt LB

Ordered-Scheduling Algorithm

Fie algoritmul precedent la care adaugam următoarea preprocesare: Înainte de a fi programate, activitățile sunt sortate descrescător după timpul de lucru.

Lema 3.

Fie o multime de n activitati cu timpul de procesare t_1, t_2, \dots, t_n astfel incat $t_1 \geq t_2 \geq \dots \geq t_n$

Daca $n > m$, atunci $OPT \geq t_m + t_{m+1}$

TEOREMA 2

Algoritmul descris anterior (Ordered-Scheduling Algorithm) este un algoritm 3/2-aproximativ

Justificări

Ciclu Hamiltonian (HC-Problem)

Fie $G=(V,E)$ un graf neorientat.

Numim *ciclu hamiltonian* un ciclu în G cu proprietatea că fiecare nod apare exact o singură dată.

HC-Problem este problema de decizie dacă într-un graf oarecare există sau nu un astfel de ciclu.

HC-Problem este NP-Completa

Traveling Salesman Problem (TSP)

TSP:

"Un vânzător ambulant vrea să își promoveze produsele în n locații. El dorește să treacă prin toate localitățile o singură dată, la final ajungând în localitatea de unde a plecat. Pentru a lucra cât mai eficient, vânzătorul dorește să minimizeze costul total al deplasării"

TSP este o problema NP-hard. Găsirea unui algoritm aproximativ este necesară!

După cum vom vedea, nu dispunem de un astfel de algoritm.

Teorema 1.

Nu există nicio valoare c pentru care să existe un algoritm în timp polinomial și care să ofere o soluție cu un factor de aproximare c pentru TSP, decât dacă $P=NP$.

Demo: Vom arată că există un asemenea algoritm aproximativ, dacă și numai dacă putem rezolva problema HC în timp polinomial.

Traveling Salesman Problem (TSP)

În ciuda pesimismului oferit de rezultatul anterior, putem fi optimiști. :-)

Pug-ul nostru comis-voiajor se deplasează într-un spațiu euclidian. Deci se respectă întotdeauna regula triunghiului!

Regula triunghiului (recap): Pentru orice triunghi cu lungimea laturilor $L_1 \geq L_2 \geq L_3$, avem $L_3 + L_2 \geq L_1$

Pentru un graf complet, ponderat, care respectă regula triunghiului, există algoritmi aproximativi pentru rezolvarea TSP!!!

Traveling Salesman Problem (TSP)

Regula triunghiului pe grafuri ne spune că pentru oricare 3 noduri interconectate u, v, w avem:

$$\text{len}((u,v)) \leq \text{len}((v,w)) + \text{len}((w,u))$$

Altfel spus, odată ce am traversat nodurile u, v, w - în această ordine, este mai eficient ca să ne întoarcem în u direct din w decât via v .

Observație 2:

Fie G un graf complet, ponderat, care respectă regula triunghiului. Și fie $v_1, v_2, v_3, \dots, v_k$ un lanț în graful G . Atunci avem $\text{len}((v_1, v_k)) \leq \text{len}(v_1, v_2, v_3, \dots, v_k)$

Justificare

Traveling Salesman Problem (TSP) algoritm 2-aproximativ:

Arbore parțial de cost minim - algoritmi și timpi de lucru

Asemănare dintre MST și TSP?

Ambele caută un traseu de cost total minim care să cuprindă toate nodurile

Diferențe dintre MST și TSP?

unul este un arbore, altul este un ciclu

una este P iar alta este NP hard!

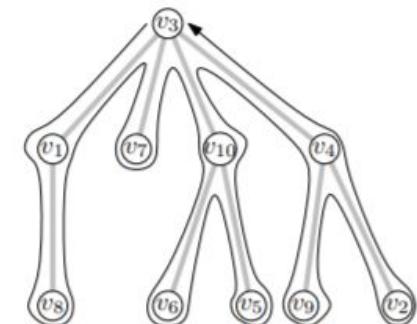
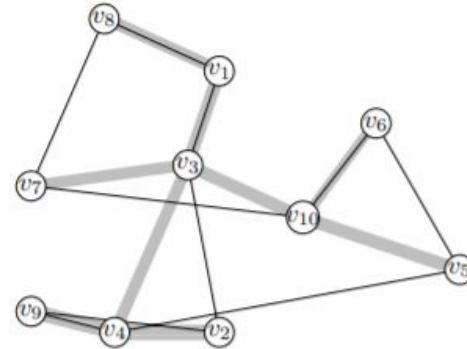
Traveling Salesman Problem (TSP) algoritm 2-aproximativ:

Lema 3:

Fie OPT costul soluției optime pentru TSP, iar MST - ponderea totală a unui Arbore parțial de cost minim pe baza aceluiași graf. Avem relația:

$$\text{OPT} \geq \text{MST}$$

Justificare



Traveling Salesman Problem (TSP) algoritm 2-aproximativ:

ApproxTSP(G)

1: Calculăm arborele parțial de cost minim T pentru graful G .

2: Alegem un nod $u \in T$ pe post de radacina.

3: $\Gamma = \emptyset$.

4: Parcurgere (u, Γ)

5: Concatenăm nodul u la finalul lui Γ pentru a închide un ciclu .

6: return Γ

Traveling Salesman Problem (TSP)

algoritm 2-aproximativ:

Parcurgere(u, Γ)

1: Concatenăm pe u la Γ .

2: pentru fiecare v , fiu al lui u :

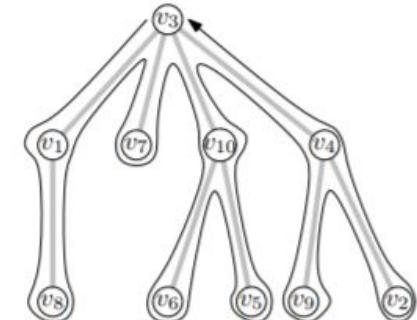
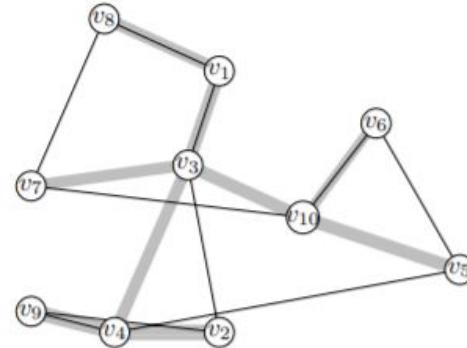
3: Parcurgere(v, Γ)

Traveling Salesman Problem (TSP) algoritm 2-aproximativ:

Teorema 4:

Algoritmul descris anterior este un algoritm 2-aproximativ pentru TSP

Justificare



Traveling Salesman Problem (TSP) BONUS!

Se poate oare mai bine?

Algoritmul lui Christofides!

Un algoritm $3/2$ aproximativ

ChristofidesTSP(G)

1: Calculăm T , un APCM în G

2: Fie $V^* \subset V$ mulțimea de vârfuri de grad impar din T . (va exista mereu un număr par de vârfuri de grad impar.)

3: Fie graful $G^* = (V^*, E^*)$ - graful complet induș de V^* .

4: Calculăm M - cuplajul perfect de pondere totală minimă pentru G^*

5: reunim mulțimile M și T ,

6: deoarece toate nodurile au grad par, putem evidenția un ciclu Eulerian Γ în multigraful induș de $M \cup T$

7: Pentru fiecare vârf din Γ , eliminăm toate "dublurile" sale, reducând costul total.

8: return Γ

Vertex cover problem

Problema:

Fie o rețea de calculatoare în care trebuie să testăm toate conexiunile.

Pentru a testa conexiunile, trebuie să instalăm un program software pe mai multe calculatoare. Acest program poate testa toate conexiunile directe care pleacă din respectivul calculator.

Evident, putem instala acest program pentru a monitoriza întreaga rețea, dar dorim să minimizam intervenția. Deci se pune problema găsirii unei submulțimi de calculatoare de cardinal minim care să poată monitoriza întreaga rețea.

Vertex cover problem

Problema formală:

Fie un graf neorientat $G=(V,E)$.

Numim "acoperire" o submulțime $S \subset V$ cu proprietatea ca pentru orice $(x,y) \in E$ avem

$x \in S$ sau $y \in S$ (sau $x,y \in S$)

Se pune problema găsirii unei acoperiri S de cardinal minim!

Această problemă este NP-hard.

Vertex cover problem

Fie următorul algoritm:

INPUT: $G=(V,E)$

$E'=E; S=\emptyset;$

cât timp $E' \neq \emptyset$:

aleg $(x,y) \in E'$;

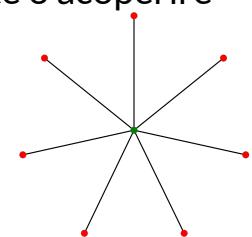
$S=S \cup \{x\}$

ștergem din E' toate muchiile incidente lui x

return S

Q1. Multimea de noduri S este o acoperire pentru graful G ?

DA!



Q2. Algoritmul de alături:

poate furniza și un răspuns de 100 de ori mai slab decât soluția optimă

Q3. Cum putem modifica algoritmul alăturat astfel încât să îmbunătățim rezultatul?

Deși pare o abordare cel puțin ciudată, algoritmul alăturat este un algoritm 2-aproximativ pentru vertex cover problem!

Vertex cover problem

Fie următorul algoritm:

ApproxVertexCover (V, E)

$E' = E; S = \emptyset;$

cât timp $E' \neq \emptyset$:

aleg $(x, y) \in E'$;

$S = S \cup \{x, y\}$



stergem din E' toate muchiile incidente lui x și lui y

return S

Deși pare o abordare cel puțin ciudată, algoritmul alăturat

- 1) generează o acoperire validă
- 2) este un algoritm 2-aproximativ

Lema 1. Fie $G = (V, E)$ un graf neorientat și OPT cardinalul unei acoperiri de grad minim a lui G . Fie $E^* \subset E$ o mulțime de muchii nod disjuncte.

Atunci avem că $OPT \geq |E^*|$

Demonstratie

Teorema 2. Algoritmul alăturat este un algoritm 2 aproxiimat pentru VCP.

Demonstratie

Complicam Problema! Weighted Vertex Problem.

Fie un graf $G=(V,E)$ - un graf simplu, si $f:V \rightarrow R_+$ care asociază fiecărui vârf, un cost

Trebuie să găsim o acoperire de varfuri S astfel încât să minimizăm: $\sum_{v \in S} f(v)$

Este dificil să găsim un algoritm aproximativ pt aceasta problemă prin metodele "tradiționale"

Tb sa gasim o abordare noua!

Programare Liniara

O problemă de programare liniară arată în felul următor:

- o funcție de "cost" cu d variabile x_1, x_2, \dots, x_d
- un set de n constrângerile liniare peste variabilele x_1, x_2, \dots, x_d

Scopul este asignarea de valori pentru variabilele de tip x_i , astfel încât să minimizăm (sau, după caz, să maximizăm) funcția de cost, respectând totodată toate cele n constrângeri

Programare Liniara

O problemă de programare liniară arată în felul următor:

Ex:

$$\text{Tb minimizat } c_1x_1 + \dots + c_dx_d$$

astfel încât

$$a_{1,1}x_1 + \dots + a_{1,d}x_d \leq b_1$$

$$a_{2,1}x_1 + \dots + a_{2,d}x_d \leq b_2$$

...

$$a_{n,1}x_1 + \dots + a_{n,d}x_d \leq b_n$$

Programare Liniara

O constrângere poate conține adunări de variabile,
poate folosi inegalități de orice tip ($<$, $>$, \geq , \leq , $=$)

O constrângere nu poate fi optională! Toate constrângerile sunt
"binding"

În constrangeri nu pot apărea elemente de forma " $x_i * x_j$ " sau " x^2 " -
trebuie să fie liniare!

Programare Liniara

O problemă de programare liniară arată în felul următor:

Ex:

Tb minimizat $c_1x_1 + \dots + c_dx_d$
astfel încât

$$a_{1,1}x_1 + \dots + a_{1,d}x_d \leq b_1$$

$$a_{2,1}x_1 + \dots + a_{2,d}x_d \leq b_2$$

...

$$a_{n,1}x_1 + \dots + a_{n,d}x_d \leq b_n$$

Astfel de sisteme pot fi rezolvate în timp polinomial prin algoritmi *simplex* (vezi cursul de Tehnici de Optimizare).

OBSERVAȚIE:

Algoritmii simplex rezolvă inegalitatea pentru x_i - numere reale!

Putem formula această problemă ca o problemă de programare liniară - suport de curs 4

Genetic Algorithms / Algoritmi Genetici

What are they used for?

Sunt utilizăți în probleme de optim, pentru care

- spațiul de căutare a soluțiilor posibile este mare
- nu se cunosc algoritmi exacti mai rapizi Furnizează o soluție care nu este neapărat optimă.
- Căutarea în spațiul soluțiilor candidat – euristică, bazată pe principii ale evoluției în genetică

Denumirea lor se datorează preluării unor mecanisme din biologie: moștenirea genetică și evoluția naturală pentru populații de indivizi

Aplicații

- Robotică, bioinformatică, inginerie
- Probleme de trafic, rutare, proiectare
- Criptare, code-breaking
- Teoria jocurilor
- Clustering etc

Algoritmi Genetici: Noțiuni

Cromozom = mulțime ordonată de elemente (gene) ale căror valoare (alele) determină caracteristicile unui individ

1	0	1	1	1	0	0	1	0	0
---	---	---	---	---	---	---	---	---	---

Populație = mulțime de indivizi care trăiesc într-un mediu la care trebuie să se adapteze

Fitness (adecvare) = măsură a gradului de adaptare la mediu pentru fiecare individ (funcție de fitness)

Algoritmi Genetici: Noțiuni

Generație = etapă în evoluția populației

Selectie = proces prin care sunt promovați indivizii cu grad ridicat de adaptare la mediu

Operatori genetici:

- **încrucișare** (combinare, crossover) - indivizi din noua generație moștenesc caracteristicile părinților
- **mutație** - indivizi din noua generație pot dobândi și caracteristici noi

Algoritm

- $t=0$
- Consideră, o populație inițială $P(0)$: alegem aleator indivizi din intervalul D
- Cât timp nu există condiția de terminare:
 - construim o populație nouă $P(t+1)$ pe baza indivizilor din $P(t)$ astfel:
 - selecție: generează o populație intermediară $P^1(t)$ selectând indivizi din $P(t)$ după un anumit criteriu de selecție
 - aplicăm operatorul de încrucișare pentru (unii) indivizi din $P^1(t)$ obținând populația intermediară $P^2(t)$
 - aplicăm operatorul de mutație peste (unii) indivizi din $P^2(t)$ obținând populația $P(t+1)$
 - optional: la $P(t+1)$ se adaugă elementul/elementele elitiste din $P(t)$
- $t=t+1$

Exemplu: maximizarea unei funcții pozitive

Date de intrare + parametri de control

- intervalul $[a, b]$
- precizia p (numărul de zecimale)
- dimensiunea populației n
- numărul de generații
- probabilitatea de încrucișare pc
- probabilitatea de mutație pm

Populația

Dimensiune (număr de cromozomi) :

n - fixă, dată

constantă pe parcursul algoritmului

Codificare = cum asociem unei configurații din spațiul de căutare un cromozom

În general: codificare binară, lungime fixă

Pentru D = [a,b] și o precizie p dată (ca număr de zecimale):

- discretizarea intervalului => $(b-a)x10^p$ subintervale (elemente)
- lungimea cromozomului este:

$$2^{l-1} < (b-a)10^p \leq 2^l \Rightarrow l = \lceil \log_2((b-a)10^p) \rceil$$

- valoarea codificată din D=[a,b] – translație liniară

$$X_{(2)} \rightarrow X_{(10)} \rightarrow \frac{b-a}{2^l - 1} X_{(10)} + a$$

Populația inițială se generează aleator

Funcția de fitness

- se pot folosi distanțe cunoscute (euclidiană, Hamming)
- pentru problema de maxim funcția este chiar f

Selectia

determinarea unei populații intermediare, ce conține indivizi care vor fi supuși operatorilor genetici

- Selectie proporțională
- Selectie elitistă
- Selectie turneu
- Selectie bazată pe ordonare

Selectia proporcională

Presupunem $P(t) = \{X_1, \dots, X_n\}$

asociem fiecarui individ X_i o probabilitate p_i de a fi selectat, în funcție de performanță acestuia (dată de funcția de fitness f)

$$p_i = \frac{f(X_i)}{F}$$

$$F = \sum_{j=1}^n f(X_j) = \text{performanța totală a populației}$$

folosind **metoda ruletei** selectăm n indivizi (!copii), cu distribuția de probabilitate (p_1, p_2, \dots, p_n)

Selectia proportională - metoda ruletei

Folosind metoda ruletei selectăm n indivizi (!copii), cu distribuția de probabilitate (p_1, p_2, \dots, p_n)

Etapa de Selectie:

- $P^1(t) = \emptyset$
- repetă de n ori:
 - generează j cu probabilitatea (p_1, p_2, \dots, p_n) folosind **metoda ruletei**
 - genereaza u variabila uniformă pe $[0,1]$
 - determină indicele j astfel încât u este între $q_{j-1} = p_1 + \dots + p_{j-1}$ și $q_j = p_1 + \dots + p_j$ (cu convenția $q_0 = 0$)
 - adaugă la populația selectată $P^1(t)$ o copie a lui X_j

Selectia

Selecție elitistă = trecerea explicită a celui mai bun individ în generația următoare

Selecție turneu = se aleg aleatoriu k indivizi din populație și se selectează cel mai performant dintre ei

Selecție bazată pe ordonare = se ordonează indivizii după performanță și li se asociază câte o probabilitate de selecție în funcție de locul lor după ordonare

Încrucișarea

Permite combinarea informațiilor de la părinți

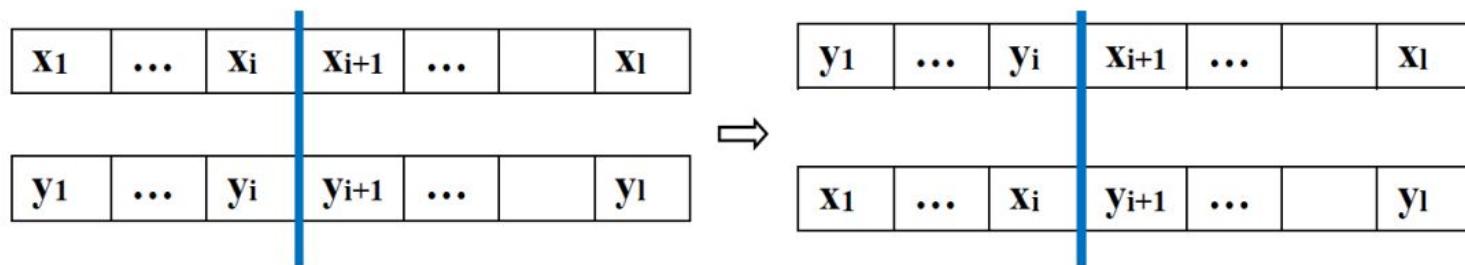
Doi părinți dau naștere la doi descendenți

- cu un punct de tăietură (de rupere)
- cu mai multe puncte de rupere
- uniformă
- etc

Încrucișarea

Cu un punct de tăietură (de rupere)

2 părinți => 2 indivizi noi care iau locul părinților în populație



i – punct de rupere generat aleator

Încrucișarea

Cu un punct de tăietură (de rupere)

Un cromozom participă la încrucișare cu o probabilitate fixată pc (probabilitate de încrucișare – dată de intrare)

Etapa de încrucișare:

- Notăm $P^1(t) = \{X_1, X_2, \dots, X_n\}$
- for $i = 1, n$
 - genereaza u variabila uniformă pe $[0,1]$
 - daca $u < pc$ atunci marcheaza (va participa la incruisare)
- formeaza perechi disjuncte de cromozomi marcați și
- aplică pentru fiecare pereche operatorul de încrucișare;
- **descendenții rezultați înlocuiesc părinții în populație**

Mutăția

schimbarea valorilor unor gene din cromozom
asigură diversitatea populației

probabilitatea de mutație pm – dată de intrare

Etapa de mutație - **Varianta 1** (mutație rară):

- Notăm $P^2(t) = \{X_1, \dots, X_n\}$ populația obținută după încrucișare
- for $i = 1, n$
 - genereaza u variabila uniformă pe $[0,1]$
 - daca $u < pm$ atunci generează o poziție aleatoare p și
 - trece gena p din cromozomul X_i la complement $0 \leftrightarrow 1$

Etapa de mutație - **Varianta 2** :

- Notăm $P^2(t) = \{X_1, \dots, X_n\}$ populația obținută după încrucișare
- for $i = 1, n$
 - for $j = 1, \text{len}(X_i)$
 - genereaza u variabila uniformă pe $[0,1]$
 - daca $u < pm$ atunci
 - trece gena j din cromozomul X_i la complement $0 \leftrightarrow 1$

Cuprins

Descriere

Probleme:

- Check Matrix multiplication;
- Primality Testing: The Solovay-Strassen Test
- Quicksort

Algoritmi probabilisti

- Ce sunt algoritmii probabilisti?

Orice algoritm care generează aleator un element $r \in \{1, 2, \dots, R\}$ și efectuează decizii în funcție de valoarea acestuia

Un astfel de algoritm poate rula un număr diferit de pași și poate oferi output-uri diferite pe aceeași intrare. Astfel devine relevant să avem mai multe iterații ale algoritmului pe un același input!

Algoritmi probabilisti

Algoritmii probabilisti pot fi impartiti in 2 (sau 3) clase:

- Algoritmi Monte Carlo:
 - rulează în timp polinomial (rapid) și oferă un răspuns "probabil" corect
- Algoritmi Las Vegas:
 - oferă mereu răspunsul corect în timp "probabil" rapid
- Algoritmi Atlantic City:
 - rulează în timp "probabil" rapid și oferă un rezultat "probabil" corect.

Algoritmi Monte Carlo

Matrix Multiplication:

Fie A, B - două matrici pătratice de dimensiune $n \times n$. Dorim să efectuăm calculul $A \times B$.

Alternative:

- Implementare naivă. Complexitate: $O(n^3)$
- Strassen (1969). $O(n^{\log 7}) = O(n^{2.81})$
- Coppersmith-Winograd (1990). $O(n^{2.376})$

Algoritmi Monte Carlo

Matrix Multiplication Check

Fie A, B, C - trei matrici pătratice de dimensiune $n \times n$. Dorim să verificăm dacă $A \cdot B = C$

Se poate mai bine decât "calea directă"?

DA!

Monte Carlo: Frievald's Algorithm

Algoritm probabilist cu următoarele proprietăți:

Fie A, B, C - matricile din problemă.

- Dacă $AxB=C$, atunci algoritmul va returna întotdeauna "DA"
- Dacă $AxB \neq C$, atunci algoritmul va returna "NU" cu o probabilitate $\geq 1/2$

Monte Carlo: Frievald's Algorithm

Problemă: A,B,C - 3 matrici pătrate de dimensiune $n \times n$; Trebuie să verificăm dacă $A \cdot B = C$.

Soluție:

Complexitate: $O(n^2)$

1. Generam un vector binar r de lungime n cu $\Pr[r_i=1]=\frac{1}{2}$.
2. Dacă $A \cdot B \cdot r = C \cdot r$, return "DA"
3. Altfel return "NU"

Observație: Dacă $A \cdot B \neq C$, atunci $\Pr[A \cdot B \neq C] \geq \frac{1}{2}$

Justificare Pt simplitate vom presupune ca matricile sunt binare (doar elemente de 0 și 1)

Primality Testing: The Solovay-Strassen Test

- Este utilizat pentru a determina dacă un număr este compus sau *probabil* prim
- se bazează pe proprietățile simbolului Jacobi și ale criteriului lui Euler
- Dat fiind un număr impar n pentru a testa primalitatea, testul alege un număr aleator a în intervalul $[2, n-1]$ și calculează simbolul Jacobi (a/n) . Dacă n este un număr prim, atunci simbolul Jacobi va fi egal cu simbolul Legendre și va satisface criteriul lui Euler

Primality Testing: The Solovay-Strassen Test

- Dat fiind un număr impar n pentru a testa primalitatea, testul alege un număr aleator a în intervalul $[2, n-1]$ și calculează simbolul Jacobi (a/n) . Dacă n este un număr prim, atunci simbolul Jacobi va fi egal cu simbolul Legendre și va satisface criteriul lui Euler
- Dacă simbolul Jacobi calculat nu satisface criteriul lui Euler, atunci n este compus. Testul este rulat pentru mai multe iterații pentru a-i crește acuratețea.

[Source](#)

Algoritmi Las Vegas

Quicksort. (C.A.R. Hoare, Moscova, 1959)

Algoritm Bazat pe strategia Divide-et-Impera

Primește ca input un șir A de elemente comparabile, returnează șirul A sortat.

Sortează oarecum asemănător ca sortarea prin inserție: la fiecare pas se fixează un element pe poziția sa.

Pași:

- Divide: se alege un element x din șirul A pe post de pivot. Se partașionează A în L (elementele $< x$), G (elementele $> x$) și E (elementele $= x$).
- Conquer: aplicăm recursiv sortarea pe șirurile L, respectiv G
- Combinare: ...

Algoritmi Las Vegas

Basic Quicksort.

1. Alegem pivotul x ca fiind fie $A[1]$, fie $A[n]$
2. În mod repetat eliminăm fiecare element y din A
 - a. inserăm y fie în L , G , sau E , în funcție de relația față de x

Fiecare inserție și ștergere durează $O(1)$

Partiționarea durează $O(n)$

detalii în [CLRS](#) pag 171; Analiza algoritmului:Justificare - $O(n^2)$

Algoritmi Las Vegas

Quicksort.

Q: Cum să asigurăm ca găsim un pivot bun?

A: Găsirea medianei!

Q: Timp?

A: Găsirea medianei se face în timp asimptotic liniar!

Algoritmi Las Vegas

Quicksort: Median selected as Pivot

- Ne asigură faptul că L și G sunt mereu echilibrate ca mărime
- Analiză complexitate: -prima $\Theta(n)$ este din cauza selectiei medianei, iar a doua pentru pasul de partiție.
- Avem: $T(n) = 2T\left(\frac{n}{2}\right) + \theta(n) + \theta(n)$
$$T(n) = \theta(n \cdot \log_2 n)$$

In practică acest algoritm perfomează mai prost decât varianta Basic.

Algoritmi Las Vegas

Randomized Quicksort:

- la fiecare pas al recursiei, pivotul este ales aleator.
- Este echivalent cu varianta Basic.
- Detalii în [CLRS](#) pag 181-184

Algoritmi Las Vegas

Paranoid Quicksort:

1. Repetă:
 - a. Alegem un pivot x aleator din A
 - b. Partitionam A în L, G, E , în funcție de x
2. Până când partițiile rezultate sunt de forma:
 - a. $|L| \leq 3/4|A|$ și $|G| \leq 3/4|A|$
3. Apelăm recursiv algoritmul pe L și G

Analiza algoritmului: [Justificare](#)

Randomized Data Structures

- O privire pe scurt, “din avion” asupra Skip lists.

Introduse în 1989 de către W. Pugh, sunt structuri dinamice bazate pe factor aleator (randomized)

Skip lists are a probabilistic data structure that seem likely to supplant balanced trees as the implementation method of choice for many applications. Skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space.

— William Pugh, Concurrent Maintenance of Skip Lists (1989)

Listele “standard”



Q: Complexitatea cautării unui element într-o lista sortată?

A: $O(n)$

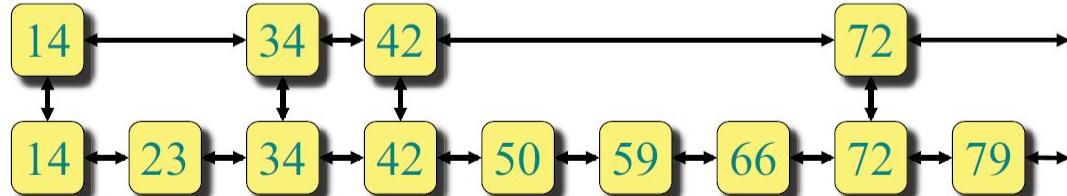
Q: Complexitate întă?

A: $O(\log n)$

Skip Lists



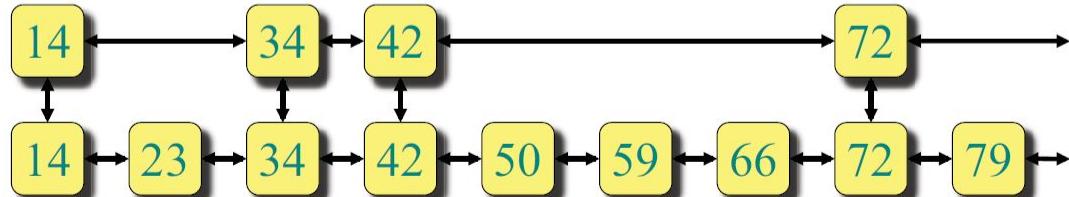
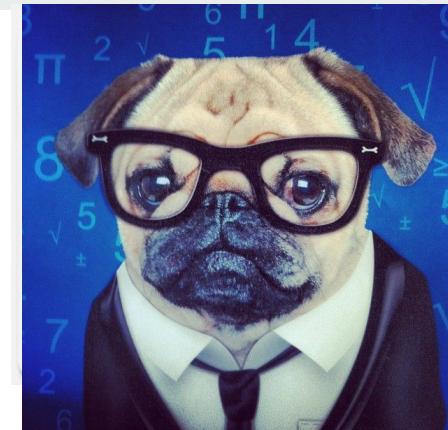
Pentru o cautare mai eficienta, vom retine 2 liste, dupa modelul urmator:



Skip Lists

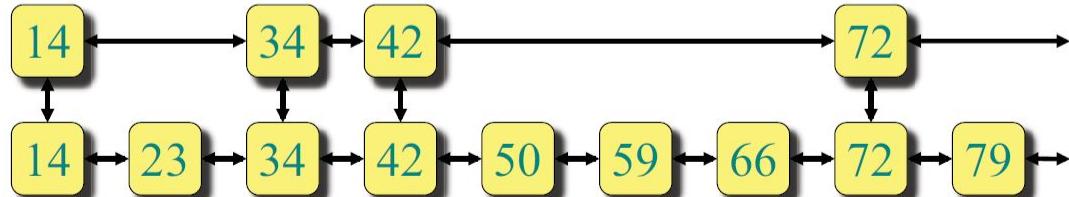
SEARCH(x):

- Walk right in top linked list (L1) until going right would go too far
- Walk down to bottom linked list (L2)
- Walk right in L2 until element found (or not)



Skip Lists

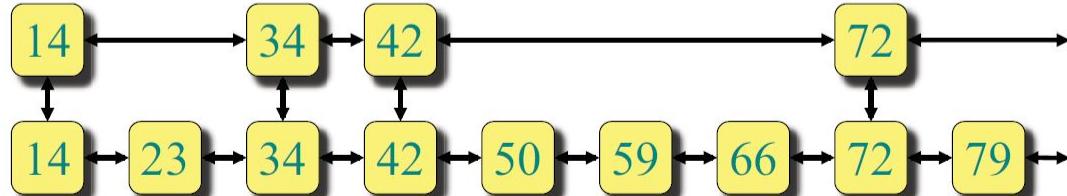
Q: Cum ar trebui distribuite nodurile din L1 (nivelul de mai sus) astfel incat cautarea sa fie cat mai eficienta?



Skip Lists

Q: Cum ar trebui distribuite nodurile din L1 (nivelul de mai sus) astfel incat cautarea sa fie cat mai eficienta?

A: Uniform distribuit!

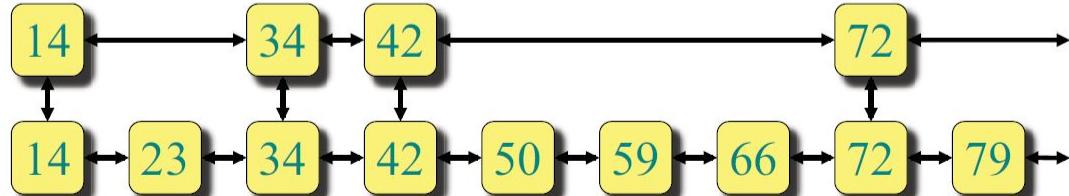


Skip Lists

Q: Cum ar trebui distribuite nodurile din L1 (nivelul de mai sus) astfel incat cautarea sa fie cat mai eficienta?

A: Uniform distribuit!

Q: cate noduri ar trebui sa existe in L1?



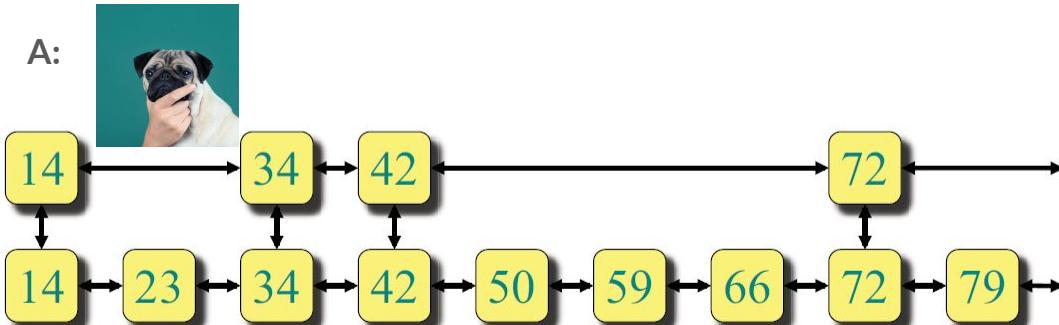
Skip Lists

Q: Cum ar trebui distribuite nodurile din L1 (nivelul de mai sus) astfel incat cautarea sa fie cat mai eficienta?

A: Uniform distribuit!

Q: cate noduri ar trebui sa existe in L1?

A:



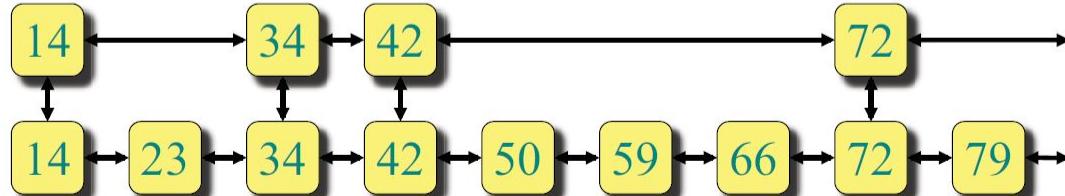
Skip Lists: Numarul de elemente per nivel

Costul unei căutări în listă este aprox $|L1| + \frac{|L2|}{|L1|}$



Relatia de mai sus este minimizata atunci cand $|L1|^2 = |L2| = n$; deci

$$|L1| = \sqrt{n}$$



Skip Lists: Numarul de elemente per nivel

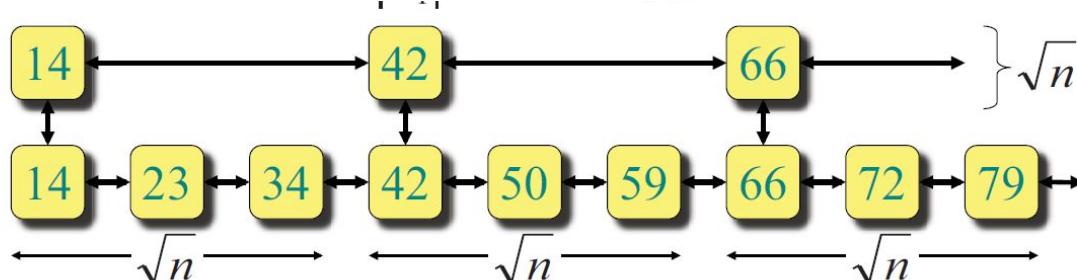
$|L1| = \sqrt{n}$; $|L2| = n$; Avem costul total de cautare pe 2 nivele:

$$|L1| + \frac{|L2|}{|L1|} = 2\sqrt{n}$$

Dar pentru 3 nivele?

Dar 4 nivele?

Dar k nivele?

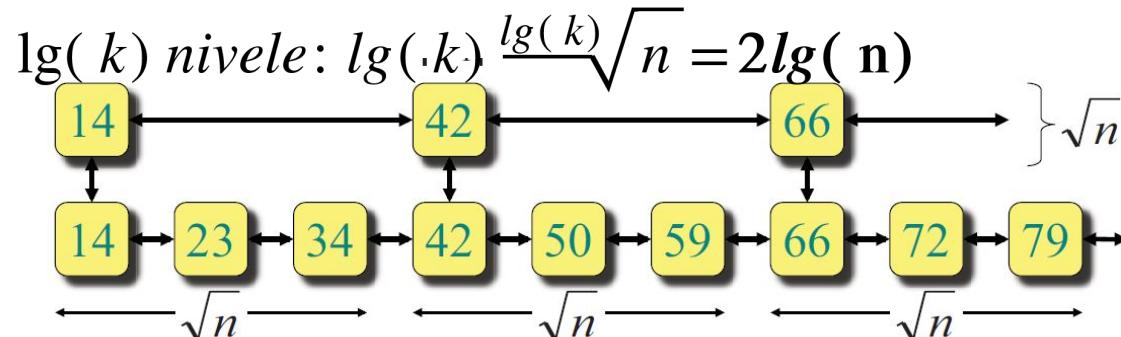


Skip Lists: Numarul de elemente per nivel

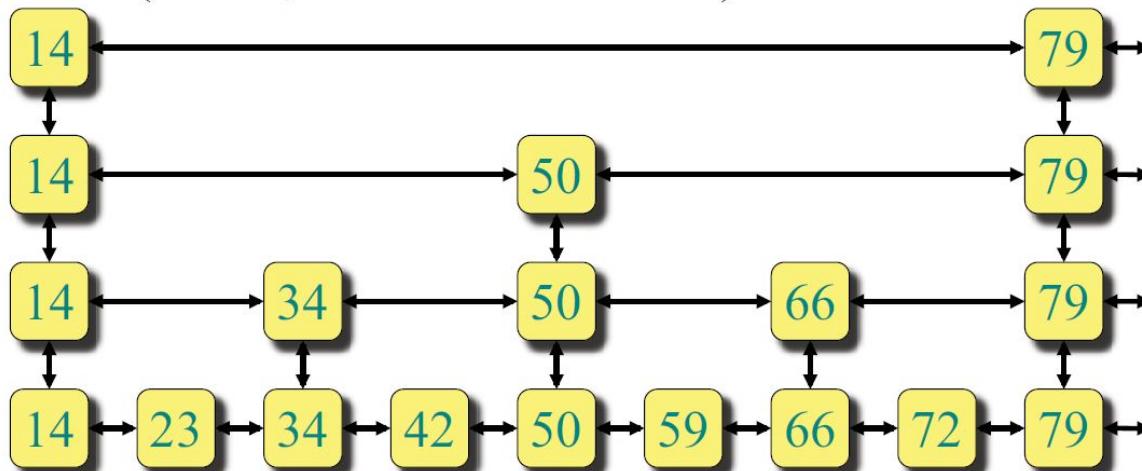
2 nivele: $2\sqrt{n}$

3 nivele: $3\sqrt[3]{n}$

k nivele: $k\sqrt[k]{n}$

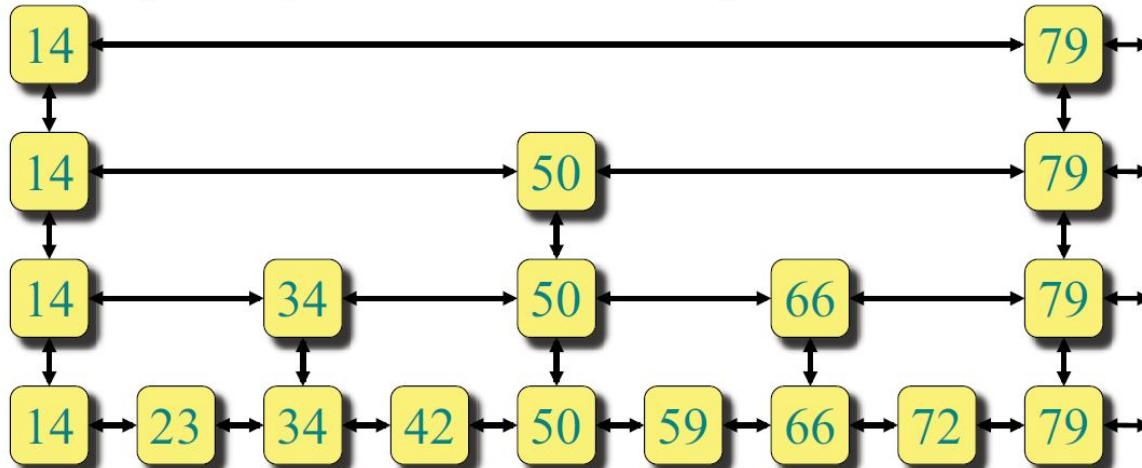


Skip Lists: Numarul de elemente per nivel



Cu ce seamănă oare?

Skip Lists: Numarul de elemente per nivel

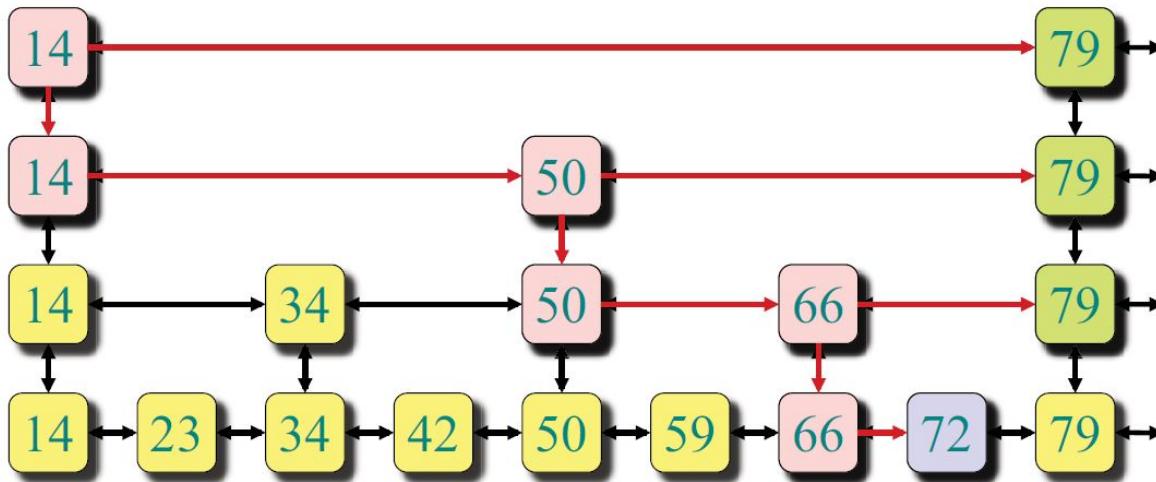


Cu ce seamana oare?

Un arbore!



Skip Lists: Search (X)



Skip Lists: Insert (X)

Pentru a insera un nou element X in lista:

- ii cautam pozitia in nivelul inferior (search(x))
- il inseram pe nivelul inferior
- il inseram si pe unele nivele superioare

OBSERVATIE: Nivelul inferior va contine intotdeauna toate elementele

Q: Pe cate alte nivale inserez X?

A: Dau cu banul! Daca pica pajura, inserez pe inca un nivel, altfel ma opresc!

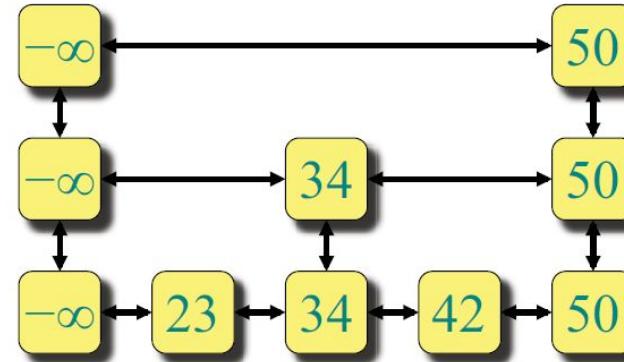
Consecinta: $\frac{1}{2}$ dintre elemente vor fi doar pe nivelul 0. $\frac{1}{4}$ dintre elemente vor fi doar pe nivelele 0 si 1. $\frac{1}{8}$ vor fi doar pe nivelele 0, 1 si 2, etc...

Skip Lists: Exercitiu

EXERCISE: Try building a skip list from scratch
by repeated insertion using a real coin

Small change:

- Add special $-\infty$ value to *every* list
 \Rightarrow can search with the same algorithm



Skip Lists: Delete (x)

Se cauta elementul x in lista (se gaseste pe cel mai de sus nivel).

Se sterge elemntul de pe toate nivelele!