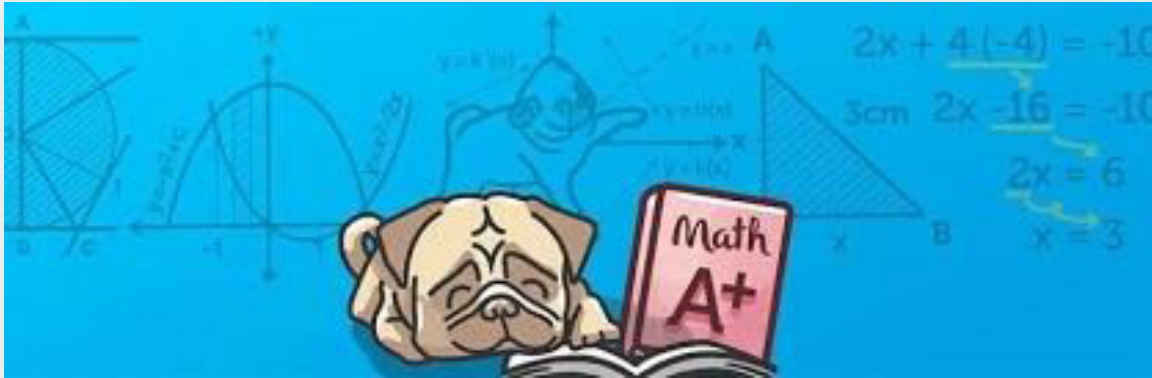
Algoritmi Avansați 2021
c-1

Lect. Dr. Ștefan Popescu

Email: stefan.popescu@fmi.unibuc.ro

Grup Teams:



Index

- Desfășurare examen & predare
- Ce este un algoritm
- Complexitatea timp a unui algoritm
- P, NP, NPC
- Ce este o problema de optim?
- Idei alternative de rezolvare (prelude)

Desfășurare examen & predare

- Curs Modular;
- Laborator 50% + examen final 50%
- Limbaj de programare: La alegere Python sau C++
- Prima jumătate a cursului [7 saptamani] va fi o continuare a cursului de AF
- Prezenta nu este obligatorie dar probabil este necesara
- Va voi fi profesor la primele 4 laboratoae si primele 4 seminarii.
- Promovarea unui mediu interactiv
- Feedback-ul este mereu de apreciat

Ce este un algoritm?

- informal: o succesiuni de pasi elementari/simpli dupa a căror execuție pe un input dat, obținem un output care este soluție pentru problema noastră

- Formal: Echivalent cu **Mașina Turing** (*to be continued*)

Complexitatea timp

Informal spus, complexitatea timp a unui algoritm este dat de *numărul de operații* efectuate până ce se ajunge la rezultat. Evident numărul de operații va depinde și de input, mai exact lungimea inputului.

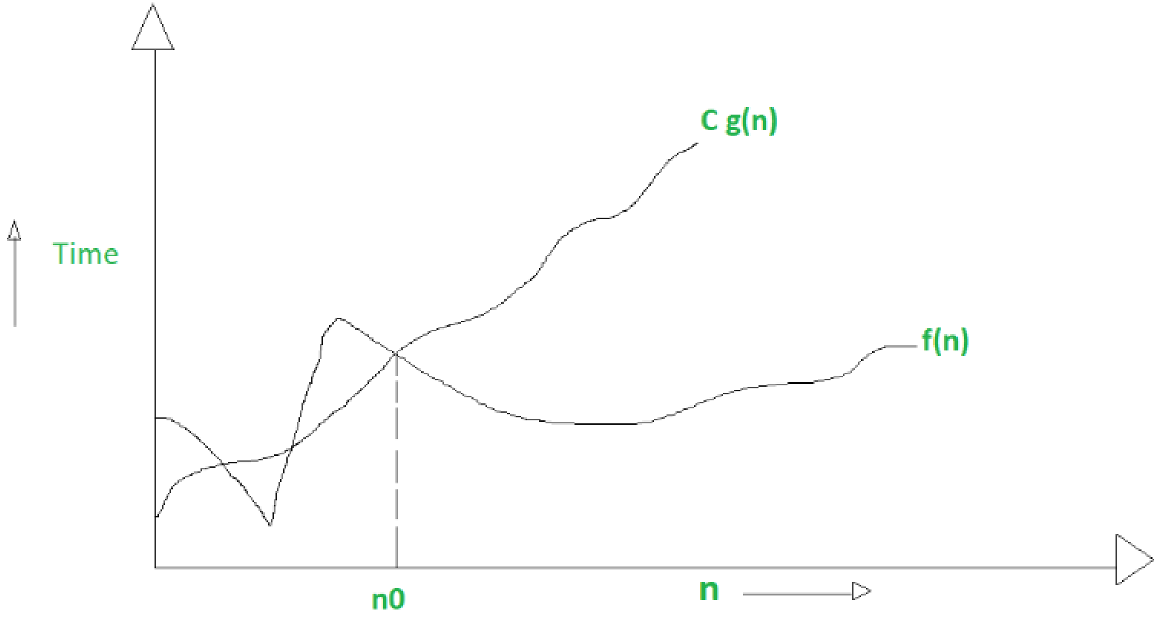
Fie un algoritm care pentru o intrare (de lungime) *n* efectuează *f(n)* operații.

Definim clasele de complexitate **O, Ω, Θ** după cum urmează

Clasa O (Big Oh)

- Descrie o **limită superioară** pentru numărul de operații efectuate de algoritm pentru orice intrare de la o lungime n_0 încolo.
- Spunem ca un algoritm rulează în timp O(g(n)) dacă există o funcție ***g*** și o valoare $\boldsymbol{n_0}$, astfel încât să avem relația: C x **g(n)≥f(n)≥0 | ∀n>**$\boldsymbol{n_0}$ (unde C este o constantă pozitivă).
- *f* este asimptotic mărginită superior de către *g* (multiplicată cu un factor constant C)
- Observăm, spre exemplu, că **O(n)** este inclusă în **O(n^2)**

Clasa O (Big Oh)



Clasa O (Big Oh)

Definiția riguroasă este ”un pic mai complicată”:

Fie un algoritm Alg și o funcție f:N→N, astfel încât Alg se termină exact în f(n) pași pentru o intrare de lungime ”n”. Spunem că Alg rulează în O(g(n)) dacă avem relația:

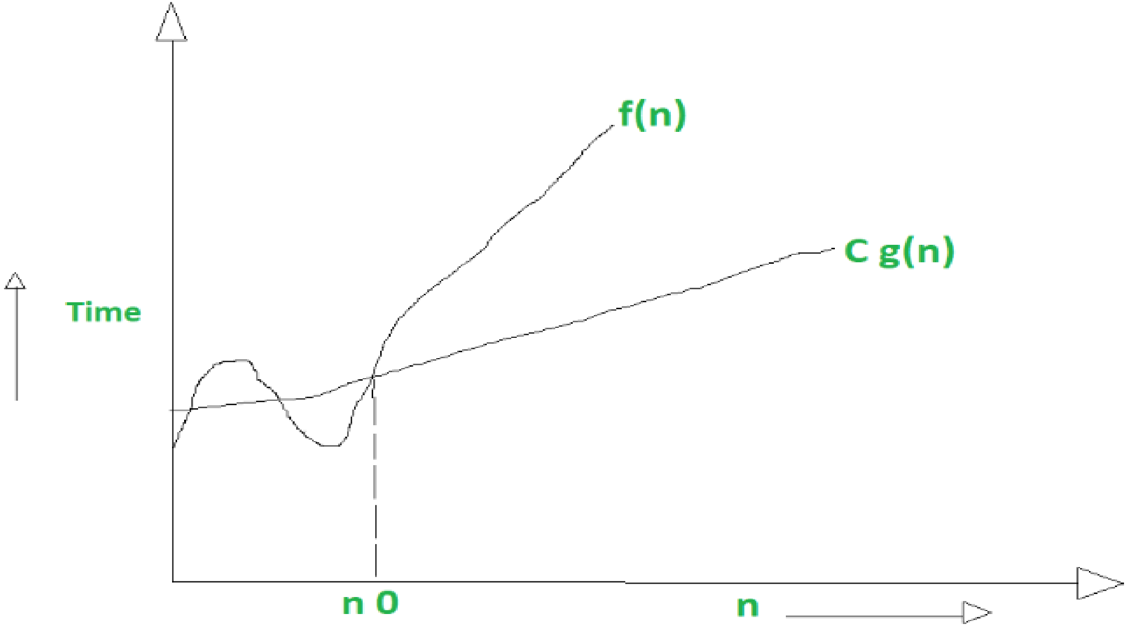
$$\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

Această definiție ne arată ca în clasa de complexitate O, factorul dominant este cel care ne dă complexitatea. Ex: $O(n^2+2n) \equiv O(n^2)$

Clasa Ω (Big Omega)

- Descrie o **limită inferioară** pentru numărul de operații efectuate de algoritm pentru orice intrare de la o lungime n_0 încolo.
- Spunem ca un algoritm rulează în timp Ω(g(n)) dacă există o funcție ***g*** și o valoare $\boldsymbol{n_0}$, astfel încât să avem relația: $\mathbf{f(n)} \geq C \times \mathbf{g(n)} \geq \mathbf{0} \mid \forall n > n_0$ (unde C este o constantă pozitivă).
- *f* este asimptotic mărginită inferior de către *g* (multiplicată cu un factor constant C)
- Observăm, spre exemplu, că Ω**(n)** este inclusă în Ω**(n^2)**

Clasa Ω (Big Omega)



Clasa Ω (Big Omega)

Definiția riguroasă:

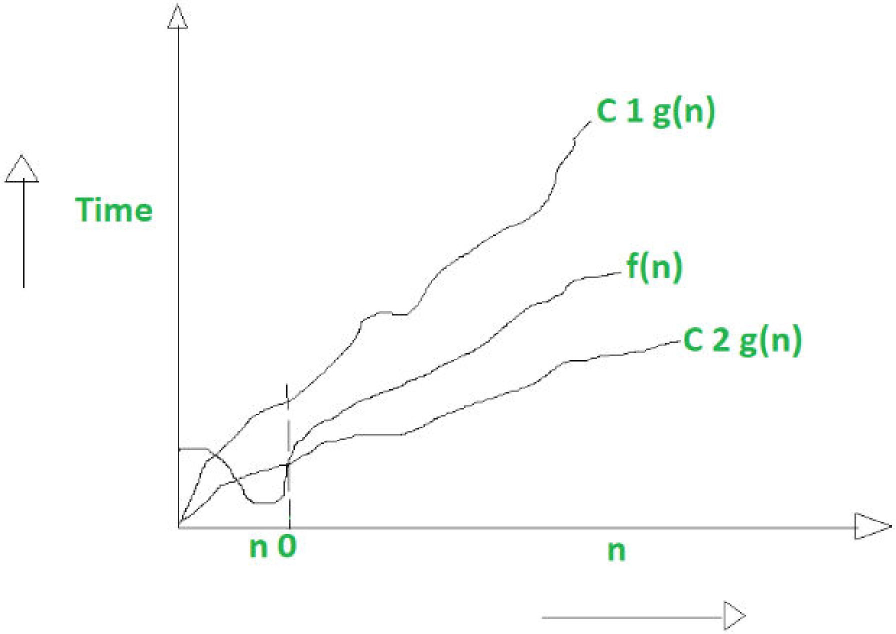
Fie un algoritm Alg și o funcție f:N→N, astfel încât Alg se termină exact în f(n) pași pentru o intrare de lungime ”n”. Spunem că Alg rulează în Ω(g(n)) dacă avem relația:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} > 0$$

Clasa Θ (Big Theta)

- O combinație între cele două clase anterioare. Presupune o mărginire atât superioară cât și inferioară.
- Spunem ca un algoritm rulează în timp Θ(g(n)) dacă există o funcție ***g*** și o valoare $\boldsymbol{n_0}$, astfel încât să avem relația: $C_1 \times \mathbf{g(n)} \geq \mathbf{f(n)} \geq C_2 \times \mathbf{g(n)} \geq \mathbf{0} \mid \forall n > n_0$ (unde C_1 și C_2 sunt două constante pozitive).
- *f* este asimptotic mărginită inferior de către *g* (multiplicată cu un factor constant C_2) respectiv superior tot de *g* (multiplicată cu un factor constant C_1)
- **Nu mai este valabilă observația** că Θ**(n)** ar fi inclusă în Θ**(n²)**
- Nu toți algoritmii au o complexitate Theta

Clasa Θ (Big Theta)



Clasa Θ (Big Theta)

Cum ar arăta definiția folosind limite?

Clasa Θ (Big Theta)

Fie un algoritm Alg și o funcție f:N→N, astfel încât Alg se termină exact în f(n) pași pentru o intrare de lungime "n". Spunem că Alg rulează în Θ(g(n)) dacă avem relația:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \in \mathbb{R}_+$$

Source

Discuții libere

Cel mai adesea folosim clasa O.

Evident ne interesează numitele ”tight bounds”.

Ce se întâmplă când pe o intrare de aceeași lungime ai număr de pași semnificativ diferiți?

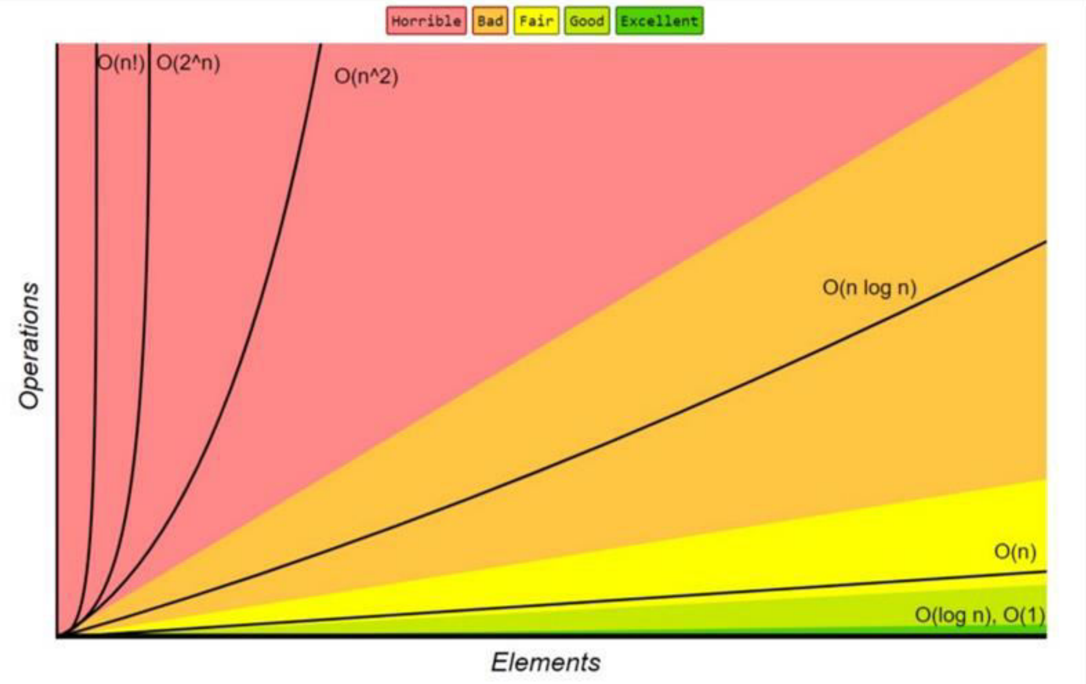
- Complexitate ”*worst case*” vs complexitate medie

Care este complexitatea următorilor algoritmi?

- Căutare binară; Bubble sort; quicksort, merge-sort;

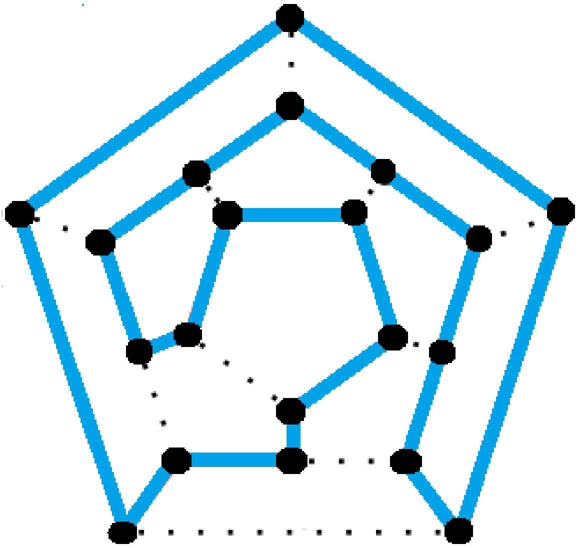
Paradoxul în care Big-O nu redă realitatea...

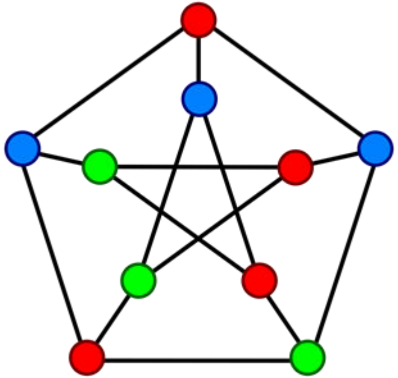
Ce inseamnă ”algoritm eficient”?



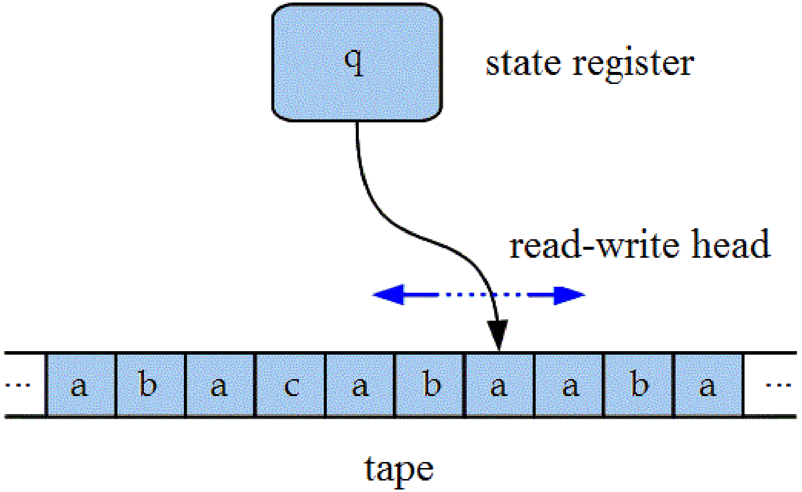
Timp polinomial:
Determinism vs nedeterminims

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





Scurt prezentare: ”Turing Machine”



O mașină Turing M=(Q, **Γ**, b, **Σ**, $\boldsymbol{\delta}$, q_0, F) unde:

- Q - mulțimea stărilor
- **Γ** - alfabetul de lucru al mașinii
- b ∊ **Γ** - un simbol special, numit ”blank”
- **Σ**⊂**Γ**\{b} - alfabetul de intrare (alfabetul pt input)
- q_0, F - starea inițială, respectiv mulțimea stărilor finale

$\boldsymbol{\delta}$ - funcția de tranziție:
cazul determinist: $\boldsymbol{\delta}: \Gamma x Q \to \Gamma x Q x \{left, right\}$

cazul nedeterminist: $\boldsymbol{\delta}: \Gamma x Q \to 2^{\Gamma x Q x \{left, right\}}$

Clasele de Complexitate P și NP

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

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

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

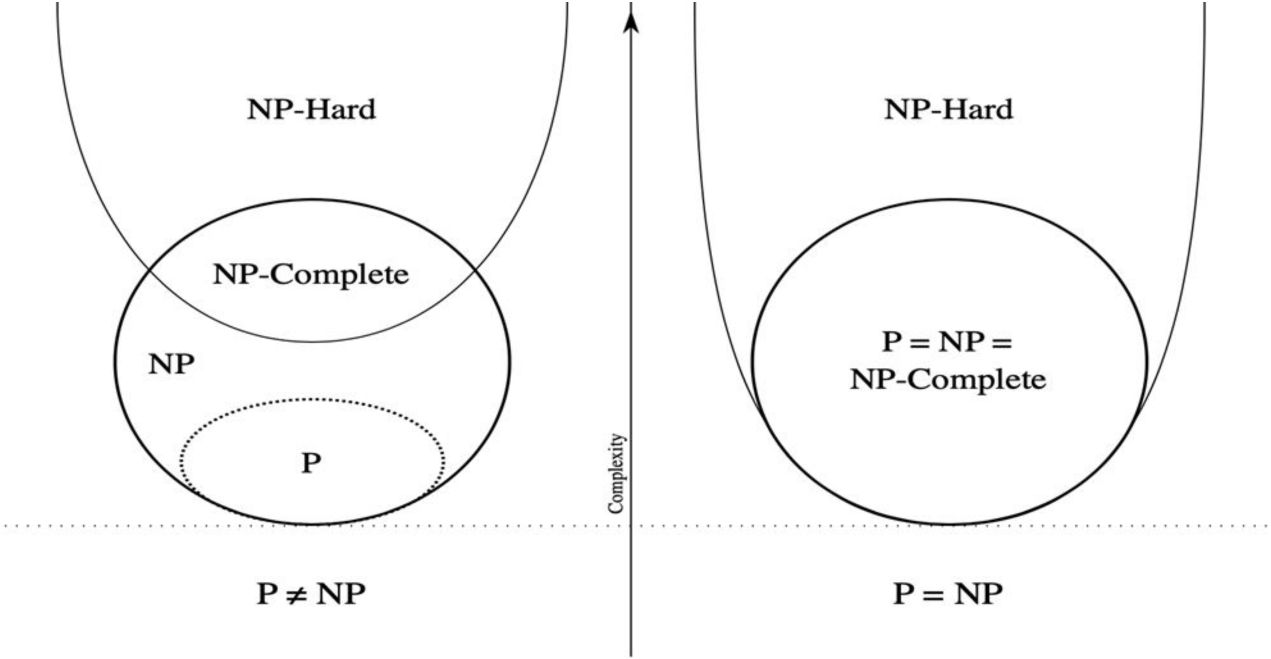
Clasele de Complexitate P și NP

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	9
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 optim 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 si ne multumim cu ce avem.

(algoritmi evoluționiști)

Probleme de optim

Fie OPT soluția optim a problemei. Ea poate fi obținută foarte greu (practic imposibil) Două dintre căile de atac pentru astfel de probleme ar fi:
- fie cazul in care OPT trebuie să minimizeze un cost. Să reușim să contruim o soluție ALG, cu
OPT<=ALG<=ρxOPT

(algoritmi ρ-aproximativi)

Aplicatie:

Algoritm aproximativ pentru 1/0 Kanspack Problem

[Whiteboard]

Next time:

Seminar & Lab - Recapitulare Fundamentele Algoritmilor

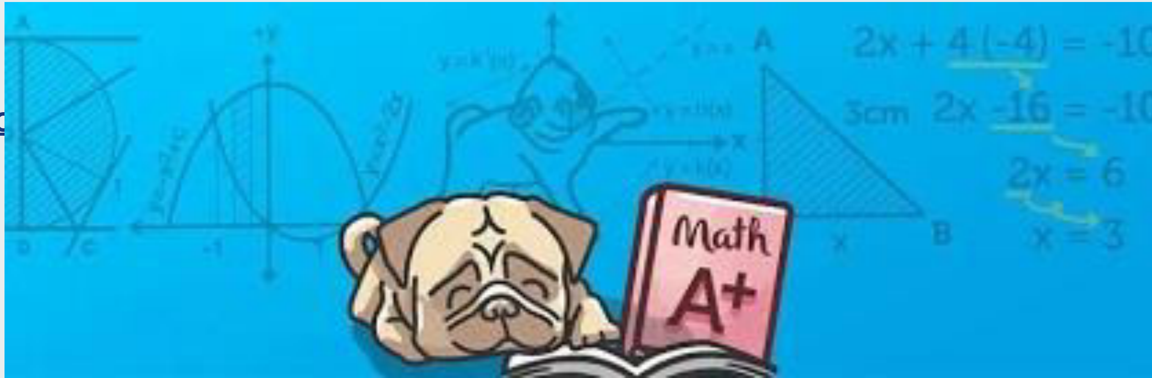
Curs 2: introducere în algoritmi aproximativi

Algoritmi Avansați 2021
c-2
Algoritmi ρ-aproximativi

Lect. Dr. Ștefan Popescu

Email: stefan.popescu@fmi.unibuc.ro

Grup Teams:



Din cursul anterior

Recapitulare:

reamintit ce estee acela un algoritm

complexitatea unui algoritm

timp determinist vs nedeterminist

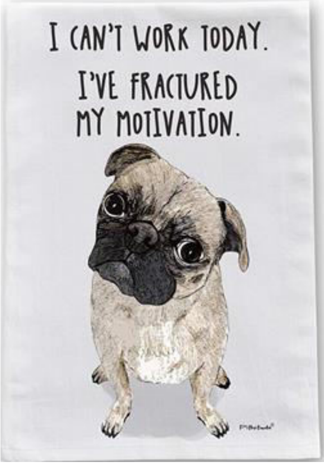
crash-course in ce inseamna P, NP, NPC

Cursul prezent

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

Motivație

Q: Daca avem nevoie să aflăm raspunsul la o problemă NP-hard?
A: Nu prea sunt șanse să găsim un algoritm care să ruleze în timp polinomial

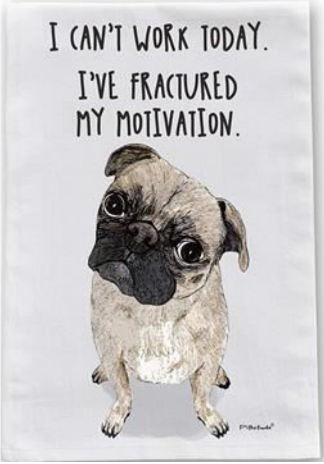


Așa că....

Motivație

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 un algoritm este *util*, acesta trebuie însoțit 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 **minimzare** se numește ρ-aproximativ, pentru o valoare $\rho > 1$, dacă $ALG(I) \le \rho \cdot OPT(I)\ pt\ \forall I$ – intrare

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

Basic Terminology & Notations

OBSERVAȚIE

(pt probleme de minim) Orice algoritm ρ-aproximativ este la rândul lui ρ'-aproximativ pentru orice ρ'>ρ. De aceea, în cazul unui algoritm ALG pentru o problemă de minimizare, spre exemplu, trebuie ca justificarea ce însoțește 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 $ALG(I) \le \rho \cdot OPT(I)\ pt\ \forall I$ – intrare

- Un algoritm ALG pentru o problema de **maximizare** se numește ρ-aproximativ, pentru o valoare $\rho < 1$, dacă $ALG \ge \rho \cdot OPT(I)\ pt\ \forall I$ – 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 $\rho = supremum_I \frac{ALG(I)}{OPT(I)}$

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(I)≤ρxOPT(I)* pentru orice intrare *I*
2. Pentru orice ρ’<ρ există un *I* pentru care *ALG(I)>ρ’xOPT(I)*. Adesea totuși ne este mai la îndemână să arătăm ca există un *I* pentru care *ALG(I)=ρxOPT(I)*

O primă provocare:
1/0 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!

O primă provocare:
1/0 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!

Presupunere: Fiecare obiect are o greutate mai mică sau egală cu capacitatea rucsacului!

O primă provocare:
1/0 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:

$Fie\ L$ – $lista\ obiectelor\ sortate\ după\ raportul\ {}^{valoare}/_{greutate}$

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

$S = 0,\ G = capacitatea\ rucsacului;$

$Pentru\ fiecare\ O{:}L$

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

$$ALG(I) = max\left(S, O_p\right)$$

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:

$Fie\ L\ – lista\ obiectelor\ sortate\ după\ raportul\ {}^{valoare}/_{greutate}$

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

$S=0,\ G=capacitatea\ rucsacului;$

$Pentru\ fiecare\ O:L$

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

$ALG(I) = max(S, O_p)$

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 S23 / Justificare S24

Rezolvare propusă:

$Fie\ L\ -\ lista\ obiectelor\ sortate\ după\ raportul\ {}^{valoare}/_{greutate}$

$Fie\ O_p\ -\ obiectul\ cu\ profitul\ cel\ mai\ mare\ din\ lista\ de\ obiecte.$

$S=0,\ G=capacitatea\ rucsacului;$

$Pentru\ fiecare\ O:L$

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

$ALG(I) = max\left(S, O_p\right)$

Load Balancing Problem

Input:

- *m* calculatoare identice; *n* activitați ce trebuiesc procesate. Fiecare activitate *j* având nevoie de t_j unități de timp pentru execuție.
- Odată inițiată, fiecare dintre activități 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

Notații:

- 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.
- $L_i=\Sigma_{j\in J(i)}t_j$

Scop: ???

Load Balancing Problem

Notații:

- 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.
- $L_i = \Sigma_{j \in J(i)} t_j$

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:

$Load - Balance\left(m, t_1, t_2, \ldots, t_n\right)$

$for\ i=1\ to\ m:$

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

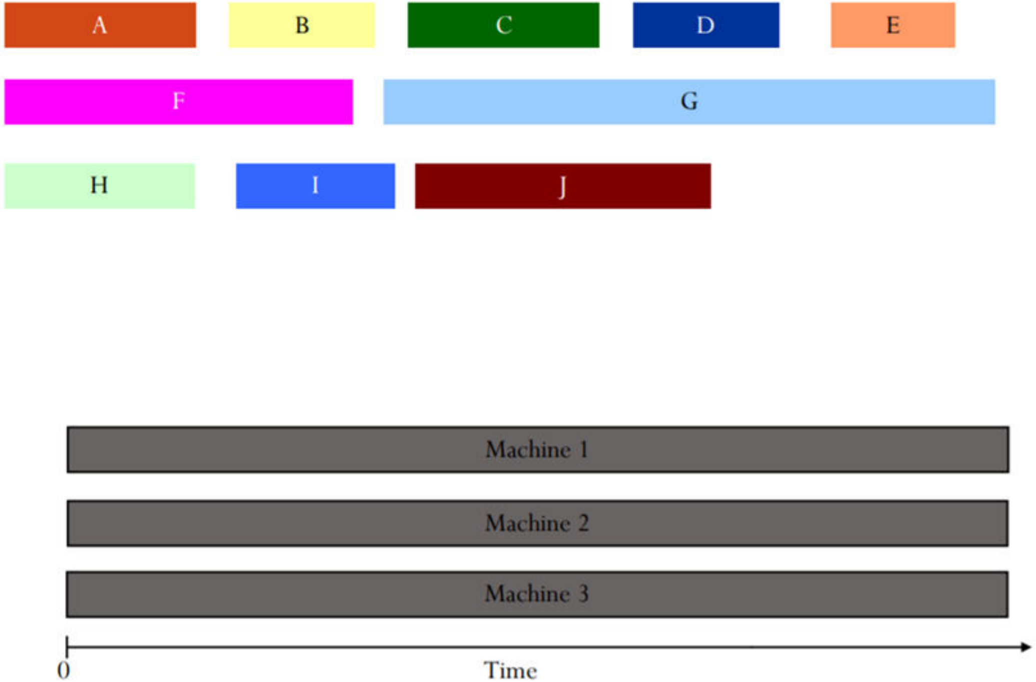
$for\ j=1\ to\ n:$

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

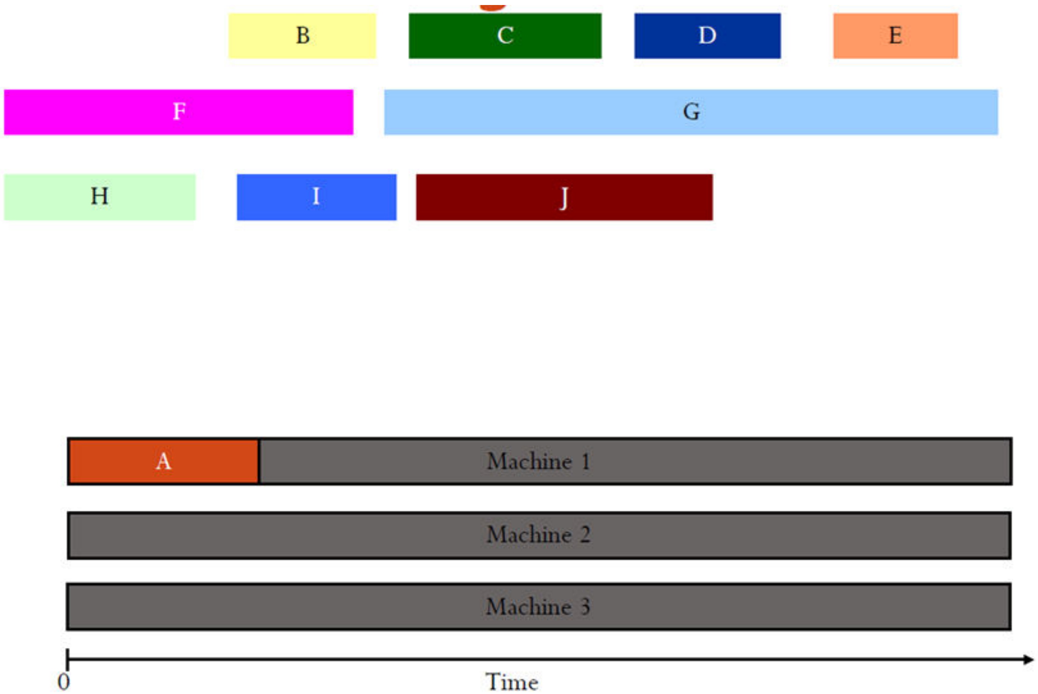
$J(i)=J(i) \cup \{j\}$

$L_i += t_j$

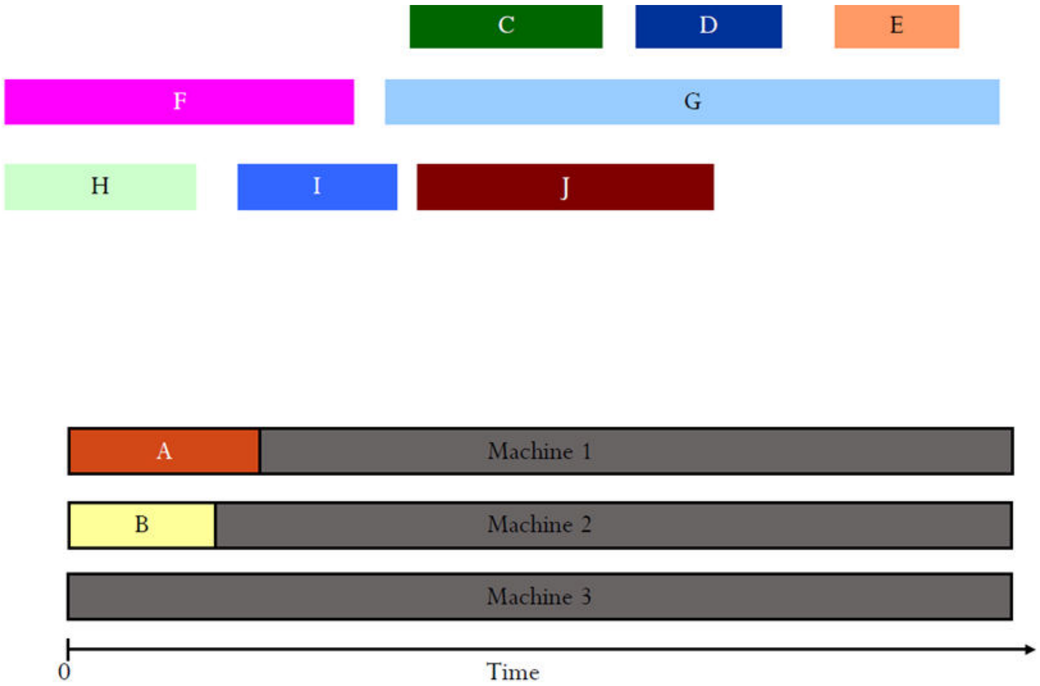
Step-by-step example



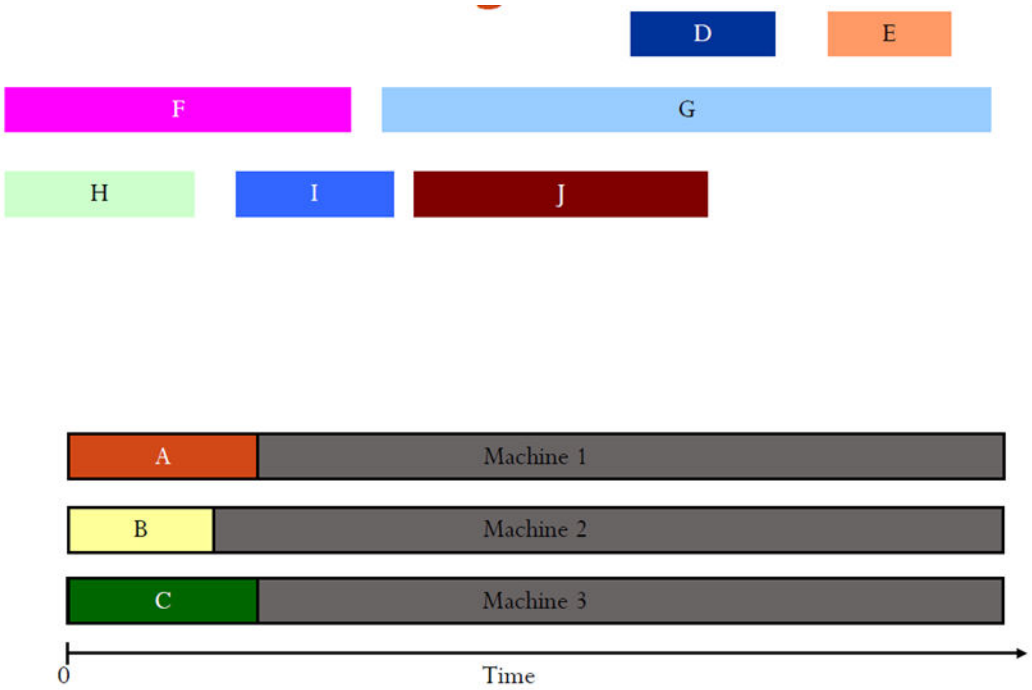
Step-by-step example



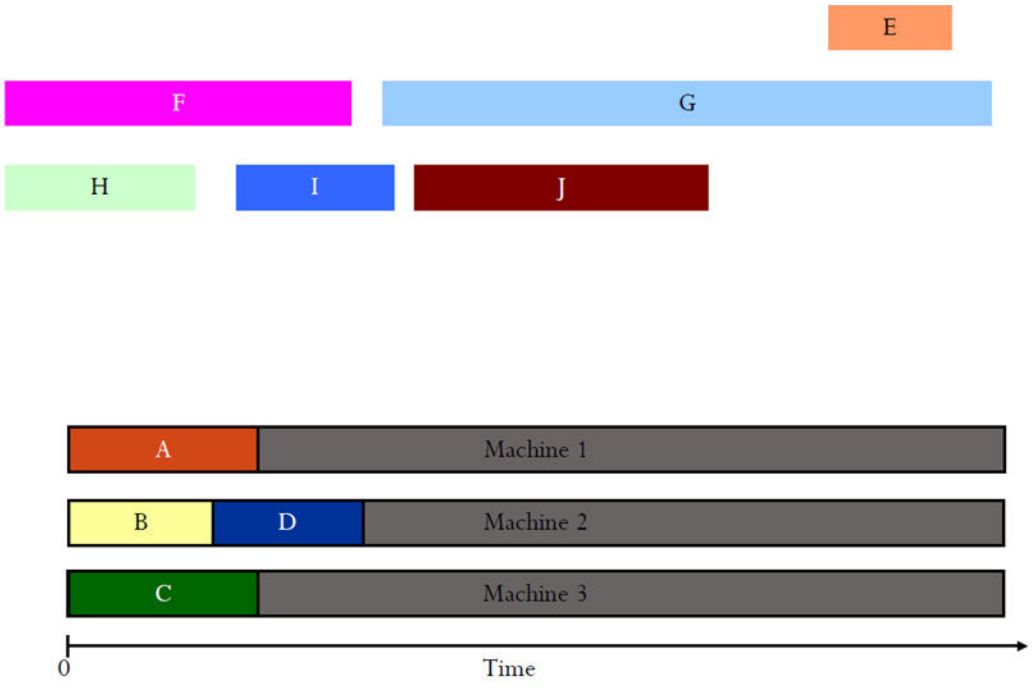
Step-by-step example



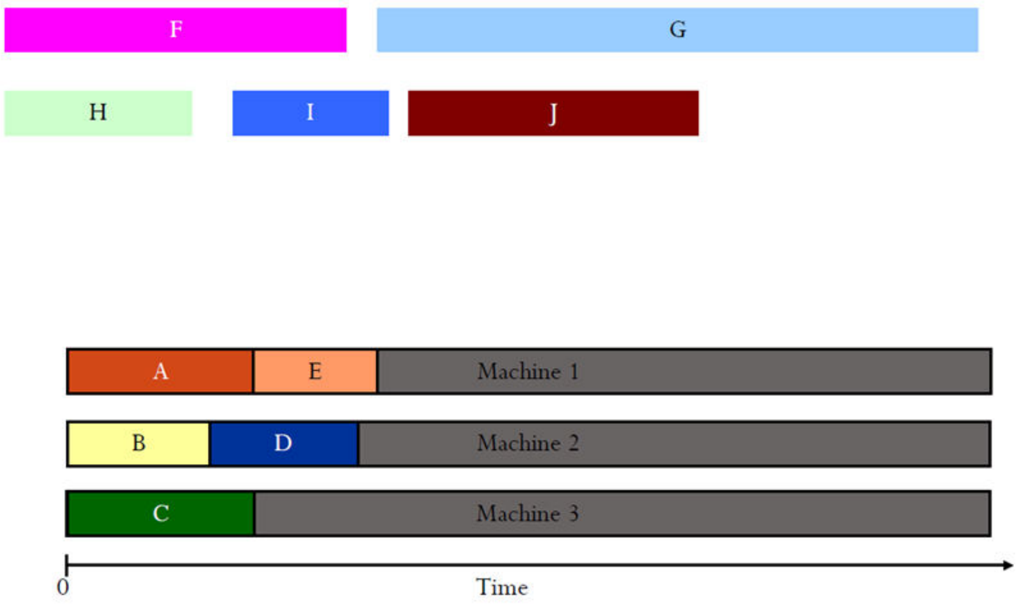
Step-by-step example



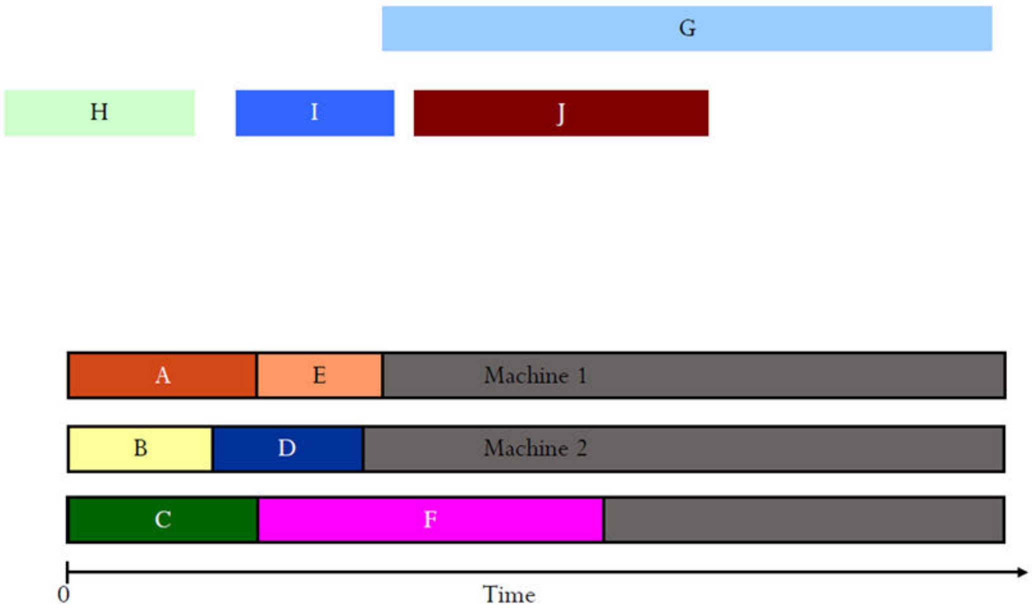
Step-by-step example



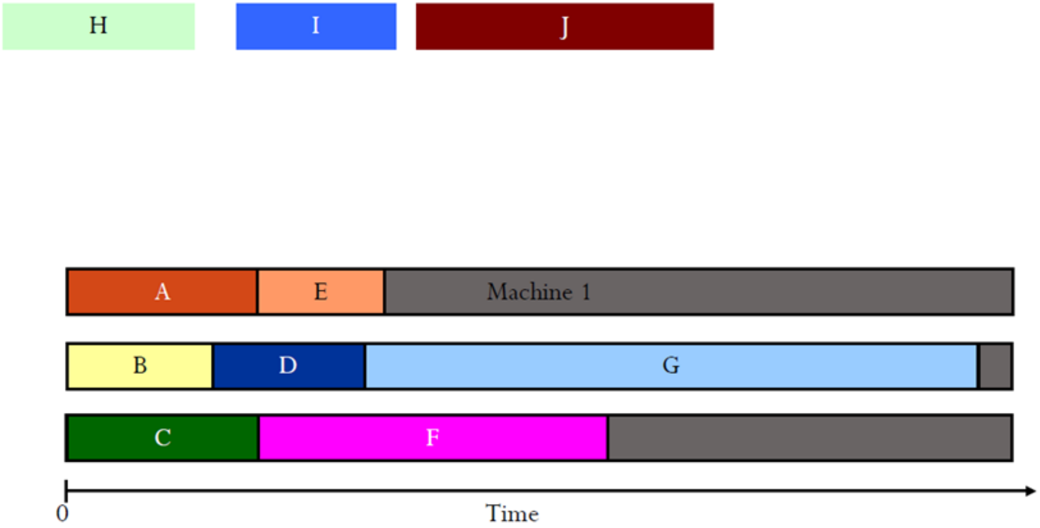
Step-by-step example



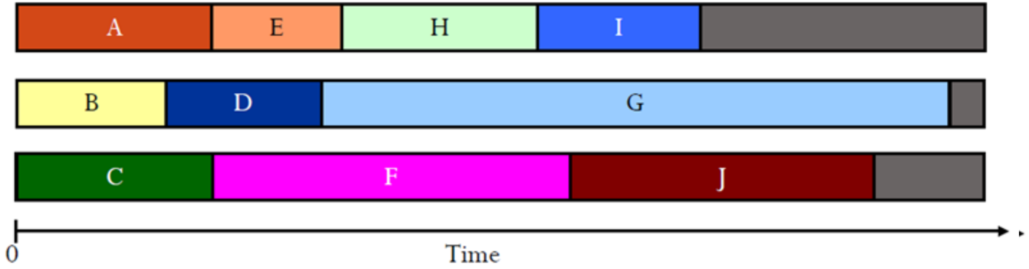
Step-by-step example



Step-by-step example

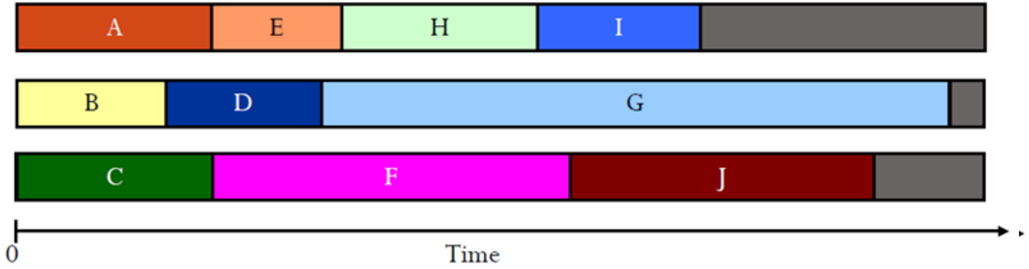


Step-by-step example (3 steps)



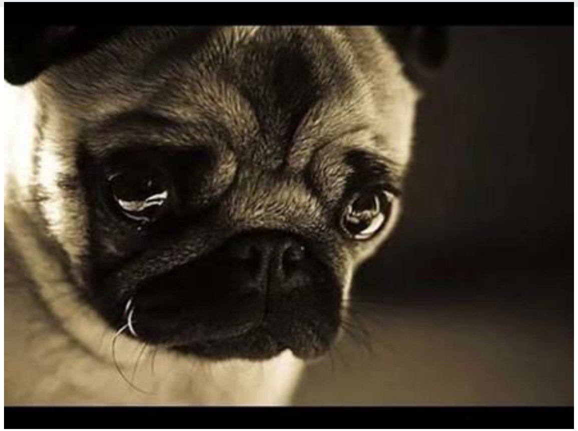
Step-by-step example (3 steps)

Este Optim?



Optimal Schedule

List schedule



NU

Care este Factorul de Aproximare?

Care este Factorul de Aproximare?

Lema 1.

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

Lema 2.
Algoritmul descris anterior este un algoritm 2-Aproximativ.
Altfel spus, fie T - max(L_i| *i* ∈{1,...,m}) masina “cea mai incarcata”. Avem de arătat că T≤2xOPT

Care este Factorul de Aproximare?

Lema 1.

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

Lema 2.
Algoritmul descris anterior este un algoritm 2-Aproximativ.
Altfel spus, fie T - max(L_i| *i* ∊{1,...,m}) masina “cea mai incarcata”. Avem de arătat că T≤2xOPT

Justificari pt Seria 23 & Seria 24

Care este Factorul de Aproximare?

Lema 2.
Algoritmul descris anterior este un algoritm 2-Aproximativ.
Altfel spus, fie T - max(L_i| *i* ∊{1,...,m}) masina “cea mai incarcata”. Avem de arătat că T≤2xOPT

Justificari pt [Seria 23] & [Seria 24]

Este ”tight bound”? Ce ar mai putea fi de facut?

Se poate imbunatății LB-ul?

3 abordari:

a) Acelasi Algoritm, o analiză mai buna asupra lower-bound-ului folosit
b) Acelasi Algoritm, gasirea unui alt lower bound folosind alte inegalități
c) Un cu totul alt Algoritm care poate da un total alt LB.

Se poate imbunatății LB-ul?

3 abordari:

a) Acelasi Algoritm, o analiză mai buna asupra lower-bound-ului folosit
b) Acelasi Algoritm, gasirea unui alt lower bound folosind alte inegalități
c) Un cu totul alt Algoritm care poate da un total alt LB.

Se poate imbunatății 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

Justificari (continuare) [Seria 23](#) & [Seria 24](#)

Se poate imbunatății LB-ul?

3 abordari:

a) Acelasi Algoritm, o analiză mai buna asupra lower-bound-ului folosit
b) Acelasi Algoritm, gasirea unui alt lower bound folosind alte inegalități
c) Un cu totul alt Algoritm care poate da un total alt LB.

Se poate imbunatății LB-ul?

3 abordari:

Acelasi Algoritm, gasirea unui alt lower bound folosind alte inegalități

Nu se poate!
m mașini, m(m-1) activitati de cost 1 si o activitate de cost *m*

Se poate imbunatății LB-ul?

3 abordari:

a) Acelasi Algoritm, o analiză mai buna asupra lower-bound-ului folosit
b) Acelasi Algoritm, gasirea unui alt lower bound folosind alte inegalități
c) Un cu totul alt Algoritm care poate da un total alt LB.

Se poate imbunatății LB-ul?

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.

Tema (preludiu)

Lema 3.

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

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

TEOREMA 2

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

Next time:

Saptamana 3:

Veti primi prima parte din tema 1.

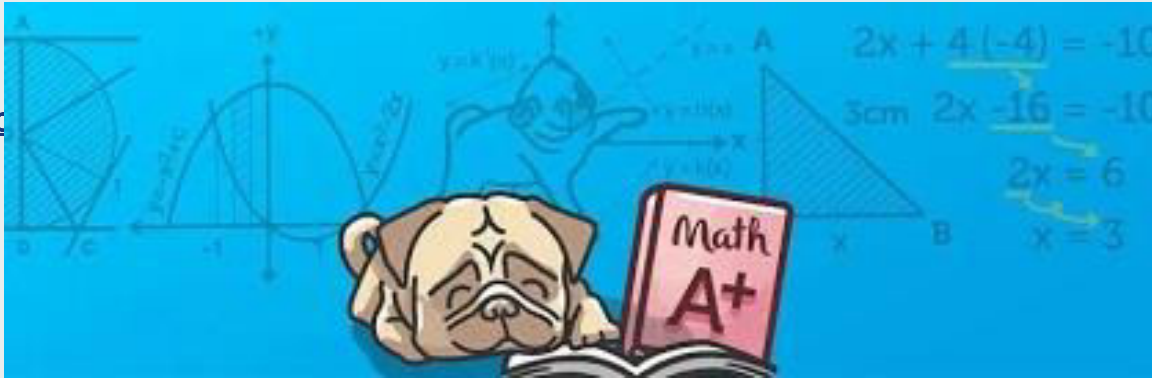
Curs 3: TSP & Christofides

Algoritmi Avansați 2021
c-3
Hamiltonian Cycle Problem, TSP, bonus: Christofides' algorithm

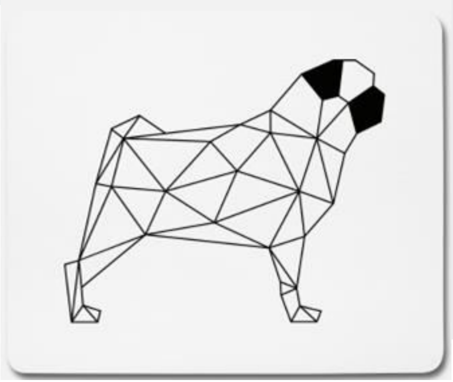
Lect. Dr. Ștefan Popescu

Email: stefan.popescu@fmi.unibuc.r

Grup Teams:



Ciclu Hamiltomian (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.

<u>HC-Problem este NP-Completa</u>

Traveling Salesman Problem (TSP)

Fie G un graf complet cu ponderi>0 pe muchii.

Evident G este graf hamiltonian, dar se pune problema găsirii ciclului hamiltonian de cost total minim.

Costul unui ciclu este suma costurilor muchiilor din componența sa.

Traveling Salesman Problem (TSP)

Fie G un graf complet cu ponderi>0 pe muchii.

Evident G este graf hamiltonian, dar se pune problema găsirii ciclului hamiltonian de cost total minim.

Costul unui ciclu este suma costurilor muchiilor din componența sa.

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 sa minimizeze costul total al deplasării”

Traveling Salesman Problem (TSP)

Fie G un graf complet cu ponderi>0 pe muchii.

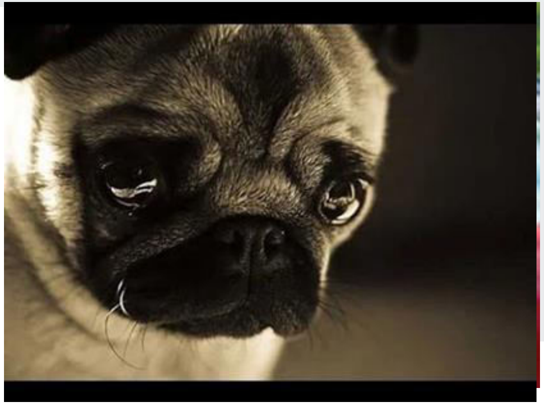
Evident G este graf hamiltonian, dar se pune problema găsirii ciclului hamiltonian de cost total minim.

Costul unui ciclu este suma costurilor muchiilor din componența sa.

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 sa minimizeze costul total al deplasării”

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

Traveling Salesman Problem (TSP)

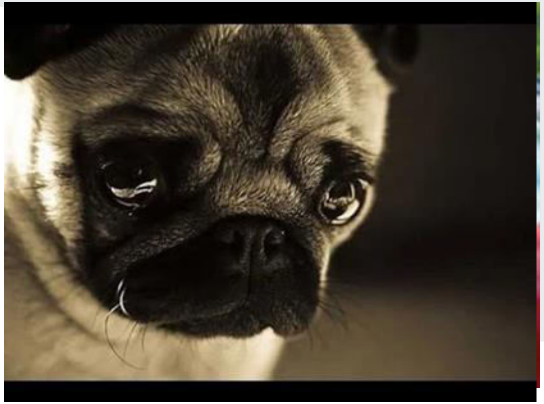


TSP:
”Un vânzător ambulant vrea să își promoveze produsele în *n* locații. El dorește să treacă prin toate localitățile o singură dată, la final ajungând în localitatea de unde a plecat. Pentru a lucra cât mai eficient, vânzătorul dorește sa minimizeze costul total al deplasării”

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

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

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 sa minimizeze costul total al deplasării”

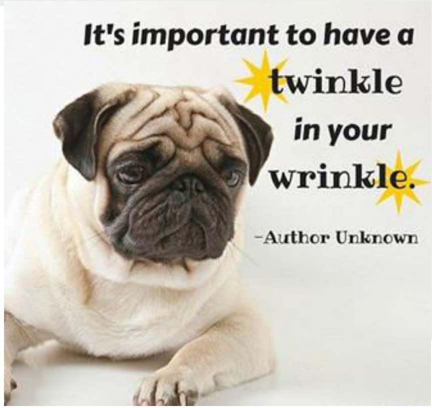
Teorema 1.

Nu există nicio valoare *c* pentru care sa 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.

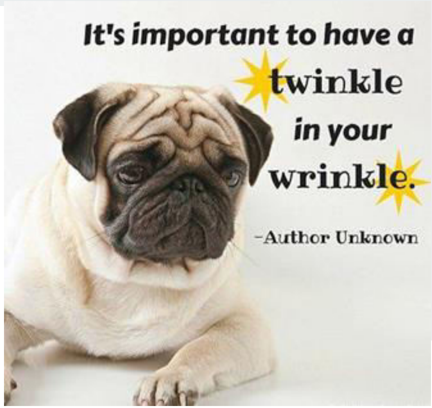
[Seria 23](#) & [Seria 24](#)

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!



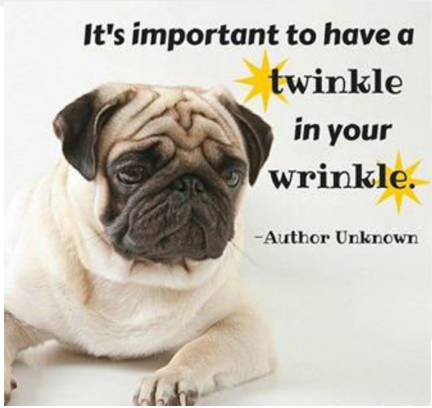
Traveling Salesman Problem (TSP)

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 $L_3 + L_2 \geq L_1$

Traveling Salesman Problem (TSP)

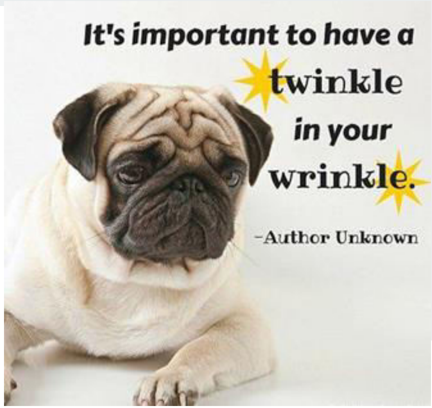


În ciuda pesimismului oferit de rezultatul anterior, putem fi optimiști. :-)

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)

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:

len((u,v))≤len((v,w))+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 $v_1, v_2, v_3, \ldots, v_k$ un lanț în graful G. Atunci avem $len((v_1,v_k)) \le len(v_1, v_2, v_3, \ldots, v_k)$

Demo: Seria 23 & Seria 24 Hint: Inducție

Traveling Salesman Problem (TSP) algoritm 2-aproximativ:

Arbore parțial de cost minim - algoritmi și timpi de lucru

Asemănare dintre MST și TSP?

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 sa cuprindă toate nodurile

Traveling Salesman Problem (TSP) algoritm 2-aproximativ:

Arbore parțial de cost minim - algoritmi și timpi de lucru

Diferențe dintre MST și TSP?

Traveling Salesman Problem (TSP) algoritm 2-aproximativ:

Arbore parțial de cost minim - algoritmi și timpi de lucru

Diferențe dintre MST și TSP?

- unul este un arbore, altul este un ciclu
- una este P iar alta este NP hard!

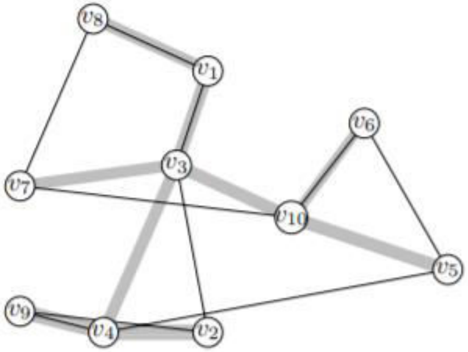
Traveling Salesman Problem (TSP) algoritm 2-aproximativ:

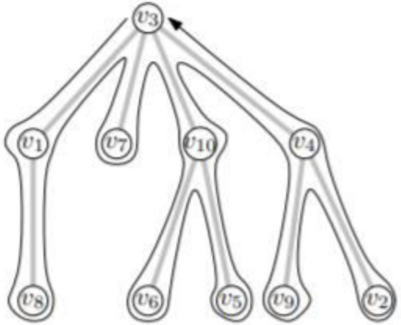
Lema 3:

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

OPT≥MST

Demo: Seria 23 & Seria 24





Traveling Salesman Problem (TSP) algoritm 2-aproximativ:

ApproxTSP(G)

1: Calculam arborele partial de cost minim T pentru graful G.

2: Alegem un nod u ∈ T pe post de radacina.

3: Γ=Ø.

4: Parcurgere (u, Γ)

5:concatenam nodul u la finalul lui Γ pentru a inchide un ciclu .

6: return Γ

Traveling Salesman Problem (TSP) algoritm 2-aproximativ:

Parcurgere(u, Γ)

1: Concatenam pe u la Γ.

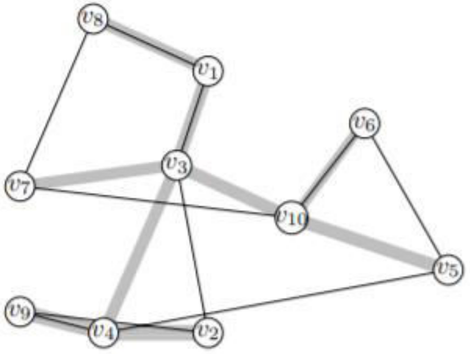
2: pentru fiecare v, fiu al lui u:

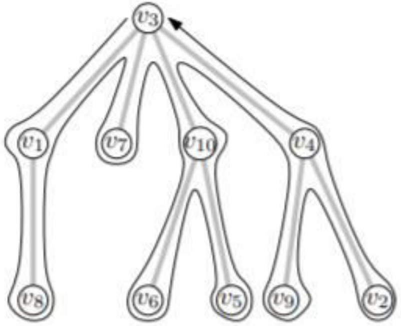
3: Parcurgere(v, Γ)

Traveling Salesman Problem (TSP) algoritm 2-aproximativ:

Teorema 4:

Algoritmul descris anterior este un algoritm 2-aroximativ pentru TSP





Demo: [Seria 23](#) & [Seria 24](#)

Traveling Salesman Problem (TSP)

Se poate oare mai bine?

Traveling Salesman Problem (TSP)

Se poate oare mai bine?

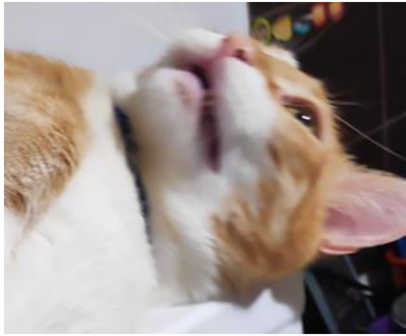
DA!

Traveling Salesman Problem (TSP) BONUS!

Se poate oare mai bine?

Algoritmul lui Christofides!

Traveling Salesman Problem (TSP) BONUS!



ChristofidesTSP(G)

1: Calculam T, un APCM in G

2: Fie $V* \subset V$ multimea de varfuri de grad impar din T . (va exista mereu un numar par de varfuri de grad impar)

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

4: Calculam M - cuplajul perfect de pondere totala minima pentru $G*$

5: reunim multimile M si T ,

6: deoarece toate nodurile au grad par, putem evidentia un ciclu Eulerian Γ in multigraful indus de M U T

7: Pentru fiecare varf din Γ, eliminam toate “dublurile” sale, reducand costul total.

8: return Γ

Next time:

Algoritmi Avansați 2021

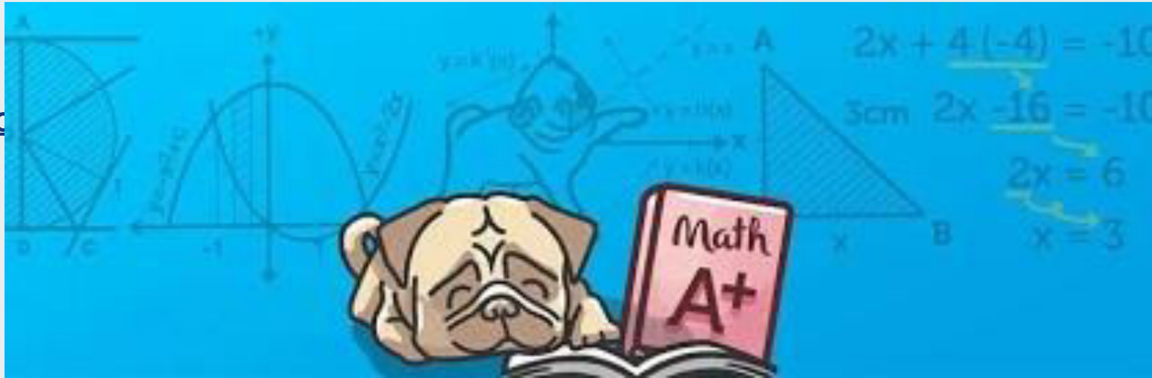
c-4

Vertex Cover Problem, Linear Programming

Lect. Dr. Ștefan Popescu

Email: stefan.popescu@fmi.unibuc.r

Grup Teams:



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.

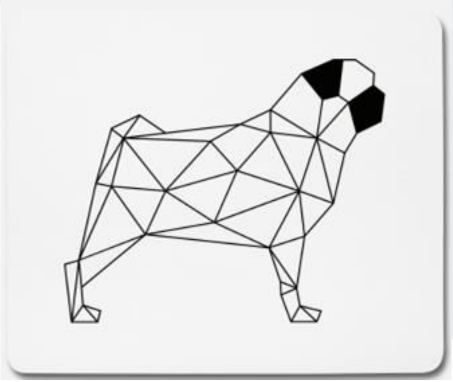
Vertex cover problem

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



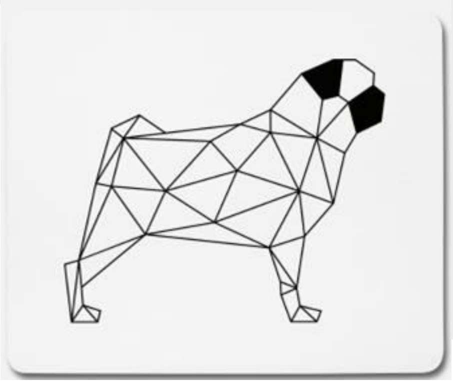
Problema formală:
Fie un graf neorientat G=(V,E).

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!

Vertex cover problem



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

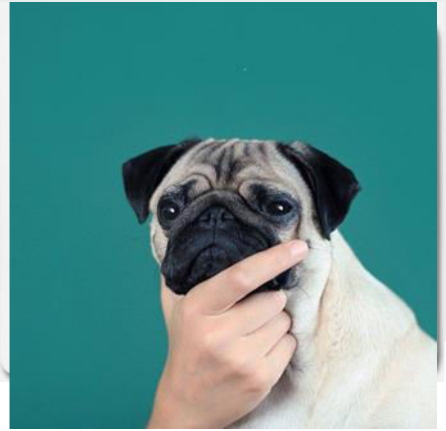
Numim ”acoperire” o submulțime $S \subset V$ cu oriorietatea ca pentru orice $(x,y) \in E$ avem

$x \in S$ 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=ø;

cât timp E'≠ø:

aleg (x,y)∊E';

S=S∪{x}

ștergem din E' toate muchiile incidente lui x

return S

Vertex cover problem

Fie următorul algoritm:

INPUT: G=(V,E)

E'=E; S=ø;

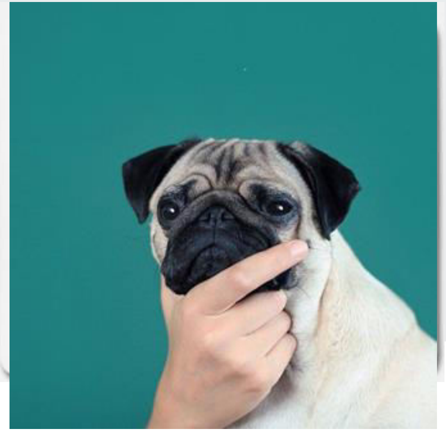
cât timp E'≠ø:

    aleg (x,y)∊E';

    S=S∪{x}

    ștergem din E' toate muchiile incidente lui x

return S



Q1. Mulțimea de noduri S este o acoperire pentru graful G?
DA/NU

Vertex cover problem

Fie următorul algoritm:

INPUT: G=(V,E)

E'=E; S=ø;

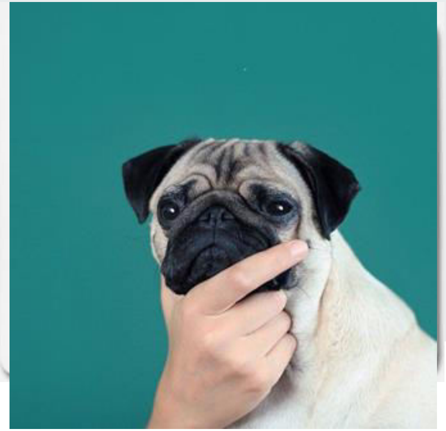
cât timp E'≠ø:

aleg (x,y)∊E';

S=S∪{x}

ștergem din E' toate muchiile incidente lui x

return S



Q1 Mulțimea de noduri S este o acoperire pentru graful G?
DA!

Vertex cover problem

Fie următorul algoritm:

INPUT: G=(V,E)

E'=E; S=ø;

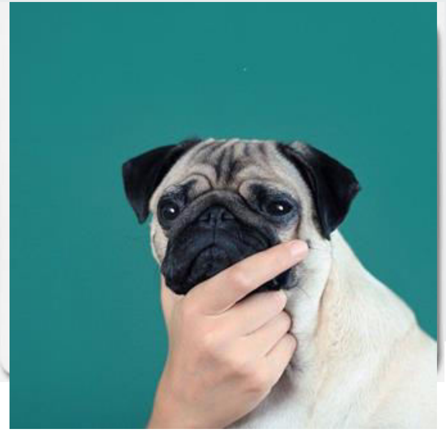
cât timp E'≠ø:

aleg (x,y)∈E';

S=S∪{x}

ștergem din E' toate muchiile incidente lui x

return S



Q2. Algoritmul de alături:

a) Este un algoritm care generează mereu soluția optimă
b) Este un algoritm 3-aproximativ pentru VCP
c) poate furniza si un răspuns de 100 de ori mai slab decât soluția optimă

Vertex cover problem

Fie următorul algoritm:

INPUT: G=(V,E)

E'=E; S=ø;

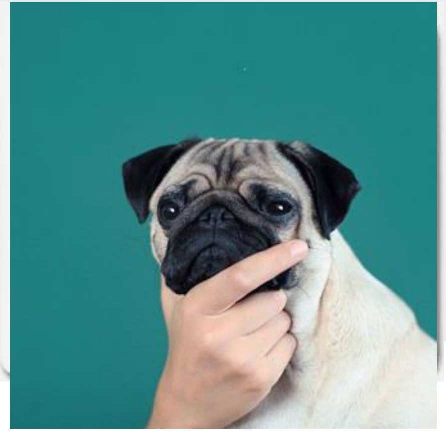
cât timp E'≠ø:

aleg (x,y)∊E';

S=S∪{x}

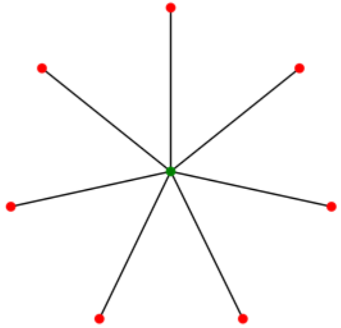
ștergem din E' toate muchiile incidente lui x

return S



Q2. Algoritmul de alături:

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



Vertex cover problem

Fie următorul algoritm:

INPUT: G=(V,E)

E'=E; S=ø;

cât timp E'≠ø:

aleg (x,y)∊E';

S=S∪{x}

ștergem din E' toate muchiile incidente lui x

return S

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

Vertex cover problem

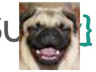
Fie următorul algoritm:

INPUT: G=(V,E)

E'=E; S=ø;

cât timp E'≠ø:

aleg (x,y)∊E';

S=SU}

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

return S

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

Vertex cover problem

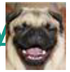
Fie următorul algoritm:

ApproxVertexCover (V,E)

E'=E; S=ø;

cât timp E'≠ø:

aleg (x,y)∊E';

S=S∪{x,y 

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

return S

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

Vertex cover problem

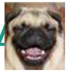
Fie următorul algoritm:

ApproxVertexCover (V,E)

E'=E; S=ø;

cât timp E'≠ø:

aleg (x,y)∊E'

S=S∪{x,y 

ștergem 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

Vertex cover problem

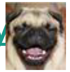
Fie următorul algoritm:

ApproxVertexCover (V,E)

E'=E; S=ø;

cât timp E'≠ø:

aleg (x,y)∊E';

S=S∪{x,y 

ştergem din E' toate muchiile incidente lui x şi lui y

return S

Lema 1. Fie G=(V,E) un graf neorientat și OPT cardinalul unei acoperiri de grad minim a lui G. Fie E'⊂E o mulțime de muchii nod disjuncte.
Atunci avem că OPT≥|E'|
Demonstratie:
Seria 23 & Seria 24

Vertex cover problem

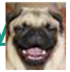
Fie următorul algoritm:

ApproxVertexCover (V,E)

E'=E; S=ø;

cât timp E'≠ø:

aleg (x,y)∊E';

S=S∪{x,y} 

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

return S

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

Demonstratie:
Seria 23 & Seria 24

Complicam Problema!
Weighted Vertex Problem.

Fie un graf G=(V,E) - un graf simplu, si $f:V \to R_+$ 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 $x_1, x_2, ..., x_d$
- un set de *n* constrângeri liniare peste variabilele $x_1, x_2, ..., x_d$

  Scopul este asignarea de valori pentru variabilele de tip x_i astfel încât să minimizăm (sau, după caz, să maximizăm) funcția de cost, respectand totodata toate cele *n* constrangeri

Programare Liniara

O problemă de programare liniară arată în felul următor:

Ex:

Tb minimizat $c_1x_1 + \cdots + c_dx_d$

astfel încât

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

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

...

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

Programare Liniara

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

O constrângere nu poate fi opțională! Toate constrângerile sunt ”binding”

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

Programare Liniara

O problemă de programare liniară arată în felul următor:

Ex:

Tb minimizat $c_1x_1 + \cdots + c_dx_d$

astfel încât

$a_{1,1}x_1 + \cdots + a_{1,d}x_d \leq b1$

$a_{2,1}x_1 + \cdots + a_{2,d}x_d \leq b2$

...

$a_{n,1}x_1 + \cdots + a_{n,d}x_d \leq b1$

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

Programare Liniara

O problemă de programare liniară arată în felul următor:

Ex:

Tb minimizat $c_1x_1 + \cdots + c_dx_d$

astfel încât

$a_{1,1}x_1 + \cdots + a_{1,d}x_d \leq b1$

$a_{2,1}x_1 + \cdots + a_{2,d}x_d \leq b2$

…

$a_{n,1}x_1 + \cdots + a_{n,d}x_d \leq b1$

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**!

Revenim la WVCP (slide 16)

Putem formula această problemă ca o problemă de programare liniară:
[Seria 23](#) & [Seria 24](#)

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**!

Further reading:

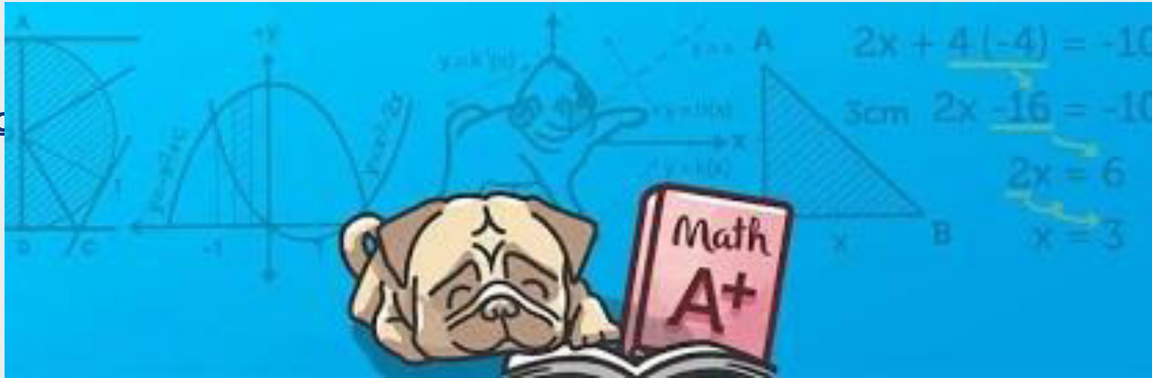
[Suport de curs saptamanile 4-5 (engl)]

Algoritmi Avansați 2021
c-6
Randomized Algorithms

Lect. Dr. Ștefan Popescu

Email: stefan.popescu@fmi.unibuc.ro

Grup Teams:



Cuprins

Descriere

Probleme:

- Check Matrix multiplication
- Quicksort

Algoritmi probabilisti

- Ce sunt?

Algoritmi probabilisti

- Ce sunt algoritmii probabilisti?

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

Algoritmi probabilisti

- Ce sunt algoritmii probabilisti?

  Orice algoritm care generează aleator un element $r \in \{1,2,...,R\}$ si 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 sa avem mai multe iterații ale algoritmului pe un același input!

Algoritmi probabilisti

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

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 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 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

Exemplu de problemă

Algoritmi Monte Carlo

Matrix Multiplication:

Algoritmi Monte Carlo

Matrix Multiplication:

Fie A,B - două matrici pătratice de dimensiune *nxn*. Dorim să efectuăm calculul AxB.

Algoritmi Monte Carlo

Matrix Multiplication:

Fie A,B - două matrici pătratice de dimensiune *nxn*. Dorim să efectuăm calculul AxB.

Alternative:

Algoritmi Monte Carlo

Matrix Multiplication:

Fie A,B - două matrici pătratice de dimensiune *nxn*. Dorim să efectuăm calculul AxB.

Alternative:

- Implementare naivă. Complexitate ?

Algoritmi Monte Carlo

Matrix Multiplication:

Fie A,B - două matrici pătratice de dimensiune *nxn*. Dorim să efectuăm calculul AxB.

Alternative:

- Implementare naivă. Complexitate: $O(n^3)$

Algoritmi Monte Carlo

Matrix Multiplication:

Fie A,B - două matrici pătratice de dimensiune *nxn*. Dorim să efectuăm calculul AxB.

Alternative:

- Implementare naivă. Complexitate: $O(n^3)$
- Strassen (1969). $O(n^{\log 7})=O(n^{2,81})$

Algoritmi Monte Carlo

Matrix Multiplication:

Fie A,B - două matrici pătratice de dimensiune *nxn*. Dorim să efectuăm calculul AxB.

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 *nxn*. Dorim să verificăm dacă AxB=C

Algoritmi Monte Carlo

Matrix Multiplication Check

Fie A,B,C - trei matrici pătratice de dimensiune *nxn*. Dorim să verificăm dacă AxB=C

Se poate mai bine decât ”calea directă”?

Algoritmi Monte Carlo

Matrix Multiplication Check

Fie A,B,C - trei matrici pătratice de dimensiune *nxn*. Dorim să verificăm dacă AxB=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 <u>întotdeauna *"DA"*</u>
- Dacă AxB≠C, atunci algoritmul va returna *"NU"* cu o probabilitate >1/2

Monte Carlo: Frievald's Algorithm

Problemă: A,B,C - 3 matrici pătrate de dimensiune *nxn*; Trebuie să verificăm dacă AxB=C.

Monte Carlo: Frievald’s Algorithm

Problemă: A,B,C - 3 matrici pătrate de dimensiune *nxn*; Trebuie să verificăm dacă AxB=C.

Soluție:

1. Generam un vector binar *r* de lungime *n* cu $Pr[r_i=1]=½$.
2. Dacă Ax(Br)=Cr, return ”DA”
3. Altfel return ”NU”

Monte Carlo: Frievald's Algorithm

Soluție:

1. Generam un vector binar *r* de lungime *n* cu $Pr[r_i=1]=½$.
2. Dacă Ax(Br)=Cr, return ”DA”
3. Altfel return ”NU”

Complexitate?

Monte Carlo: Frievald's Algorithm

Soluție:

1. Generam un vector binar *r* de lungime *n* cu $Pr[r_i=1]=½$.
2. Dacă $Ax(Br)=C$, return "DA"
3. Altfel return "NU"

Complexitate: $O(n^2)$

Monte Carlo: Frievald’s Algorithm

Complexitate: $O(n^2)$

Soluție:

1. Generam un vector binar *r* de lungime *n* cu $Pr[r_i=1]=½$.
2. Dacă Ax(Br)=Cr, return ”DA”
3. Altfel return ”NU”

Observație: Dacă $AxB \neq C$, atunci $Pr[Ax(Br) \neq Cr] \geq 1/2$

Justificare: Seria 23 & Seria 24; Pt simplitate vom presupune ca matricile sunt binare (doar elemente de 0 si 1)

Algoritmi Las Vegas

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

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, Reurnează șirul A sortat.

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

Algoritmi Las Vegas

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

Pașii:

- Divide: se alege un element *x* din șirul A pe post de pivot. Se partiționează 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)

Algoritmi Las Vegas

<u>Basic Quicksort.</u>

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 algortimului: [Seria 23](#) & [Seria 24](#) -$O(n^2)$

Algoritmi Las Vegas

Quicksort.

Q: Cum să asigurăm ca găsim un pivot bun?

Algoritmi Las Vegas

<u>Quicksort.</u>

Q: Cum să asigurăm ca găsim un pivot bun?

A: Găsirea medianei!

Q: Timp?

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 Θ(n) este din cauza selectiei medianei, iar a doua pentru pasul de partiție.
- Avem: $T(n) = 2T\left(\frac{n}{2}\right) + \theta(n) + \theta(n)$

$$T(n) = \theta\left(n \cdot \log_2 n\right)$$

Algoritmi Las Vegas

<u>**Quicksort: Median selected as Pivot**</u>

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

$$T(n) = \theta\left(n \cdot \log_2 n\right)$$

In practică acest algoritm **<u>perfomează mai prost</u>** 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

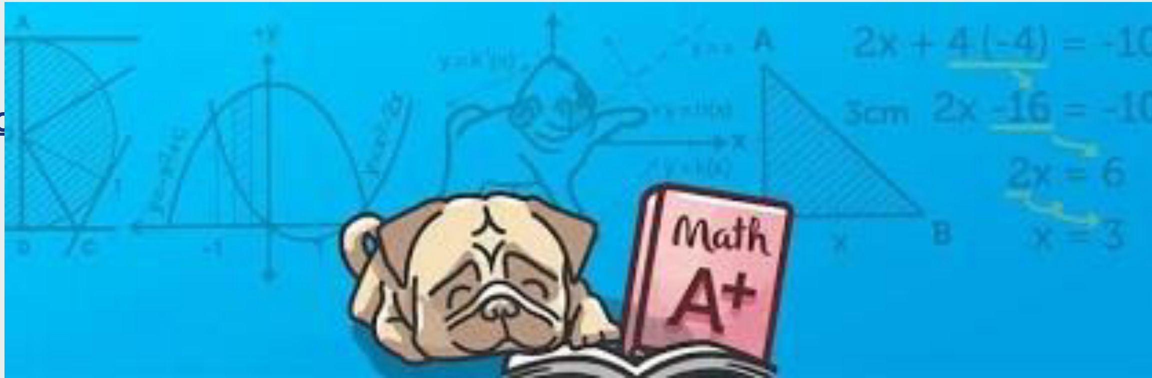
Analiza algortimului: [Seria 23] & [Seria 24]

Algoritmi Avansați 2021
c-7
Debriefing
Randomized Data Structures: Skip Lists

Lect. Dr. Ștefan Popescu

Email: stefan.popescu@fmi.unibuc.r

Grup Teams:



Debriefing

- Smalltalk about second half of the course
- presentation schedule for Tema 1 & Tema 2
- future lab work
- smalltalk about final exam

Randomized Data Structures

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

Randomized Data Structures

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

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

Randomized Data Structures

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

Introduse in 1989 de către W. Purgh, 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”

Listele “standard”

Q: Complexitatea cautarii unui element intr-o lista sortata?

Listele “standard”

Q: Complexitatea cautarii unui element intr-o lista sortata?

A: O(n)

Listele “standard”

Q: Complexitatea cautarii unui element intr-o lista sortata?

A: O(n)

Q: Suntem multumiti?

Listele “standard”

Q: Complexitatea cautarii unui element intr-o lista sortata?

A: O(n)

Q: Suntem multumiti?

A: NU!

Listele “standard”

Q: Complexitatea cautarii unui element intr-o lista sortata?

A: O(n)

Q: Complexitate țintă?

Listele “standard”

Q: Complexitatea cautarii unui element intr-o lista sortata?

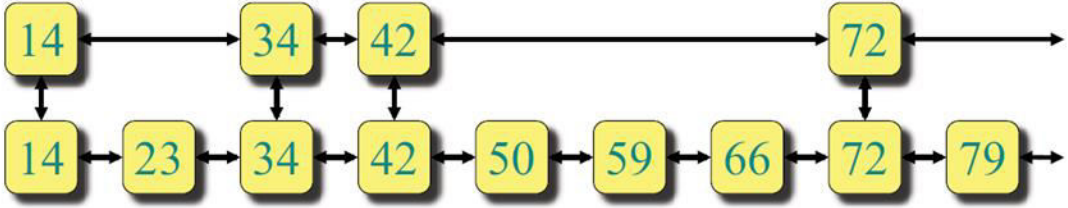
A: O(n)

Q: Complexitate țintă?

A: O(log n)

Skip Lists

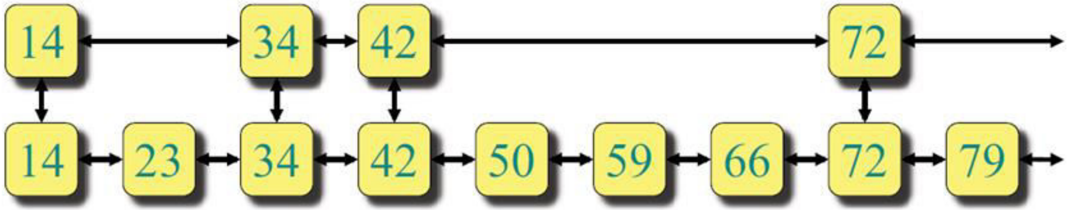
Pentru o cautare mai eficienta, vom retine 2 liste, dupa modelul urmator:



Skip Lists

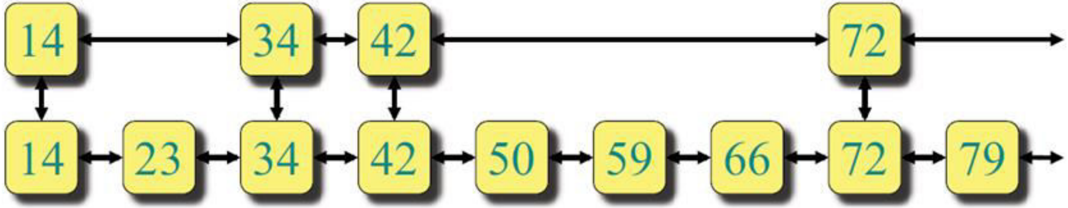
SEARCH(x):

- 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

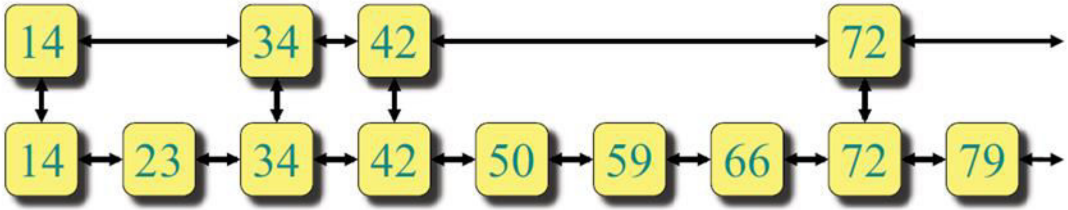
Q: Cum ar trebui distribuite nodurile din L1 (nivelul de mai sus) astfel incat cautarea sa fie cat mai eficienta?



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!

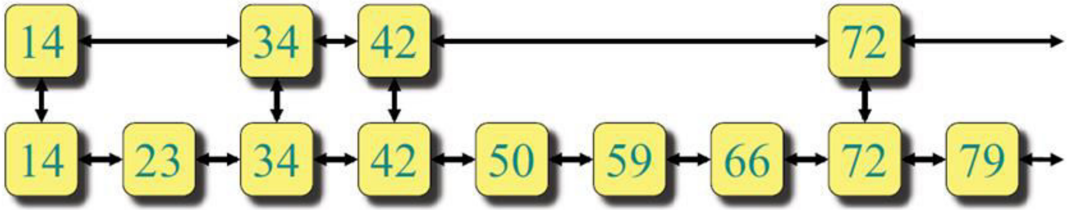


Skip Lists

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?



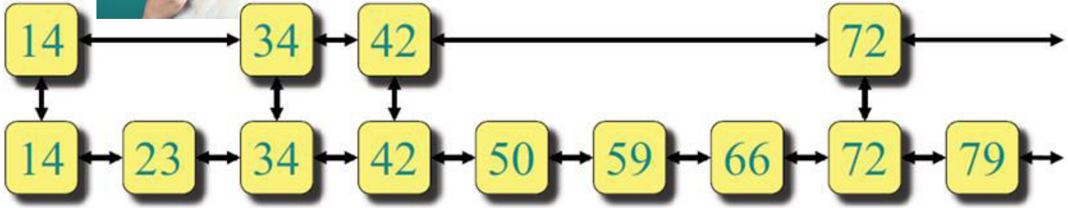
Skip Lists

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

A: Uniform distribuit!

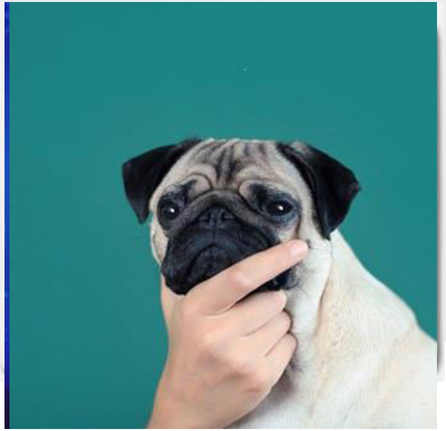
Q: cate noduri ar trebui sa existe in L1?

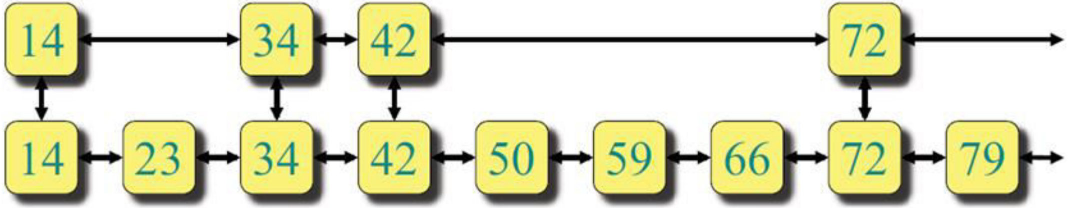
A: 



Skip Lists: Numarul de elemente per nivel

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



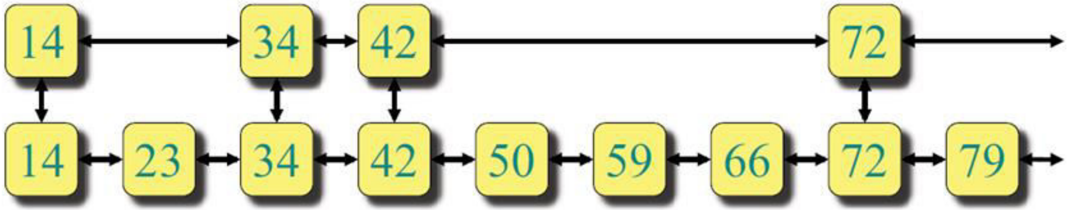


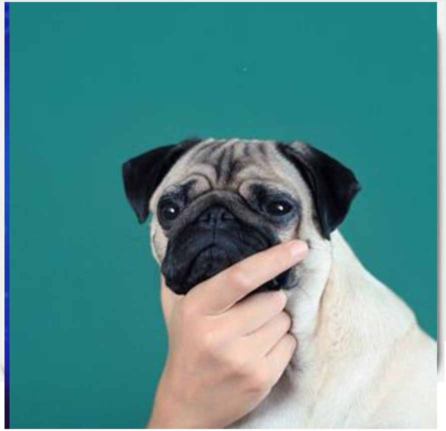
Skip Lists: Numarul de elemente per nivel

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

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

$|L1|=\mathrm{sqrt}(n)$

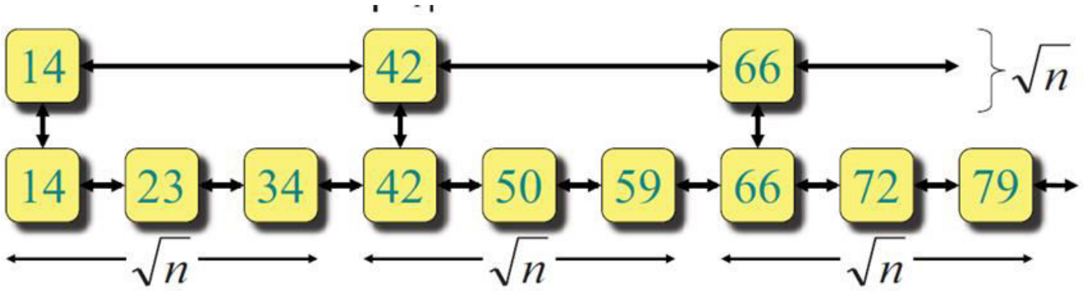


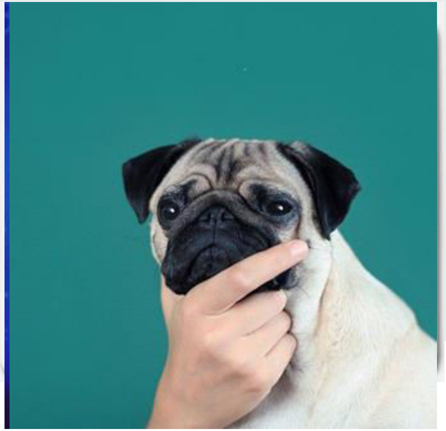


Skip Lists: Numarul de elemente per nivel

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

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



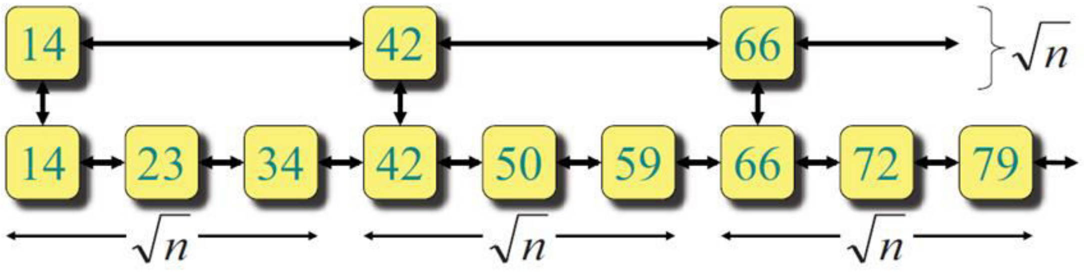


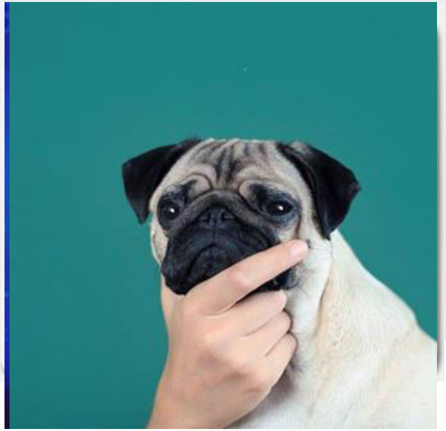
Skip Lists: Numarul de elemente per nivel

$|L1| = \sqrt{n}$; $|L2| = n$; *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?





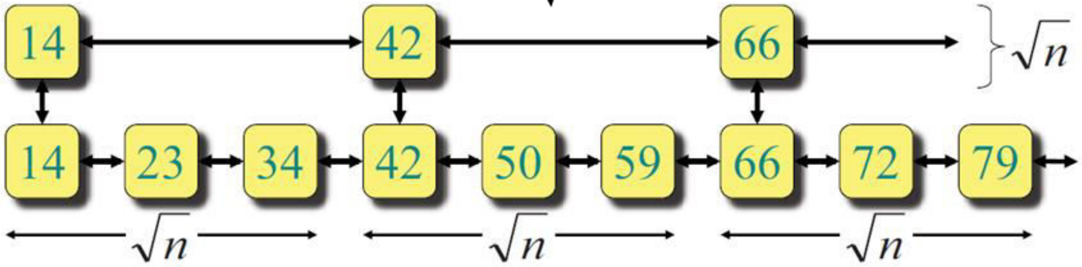
Skip Lists: Numarul de elemente per nivel

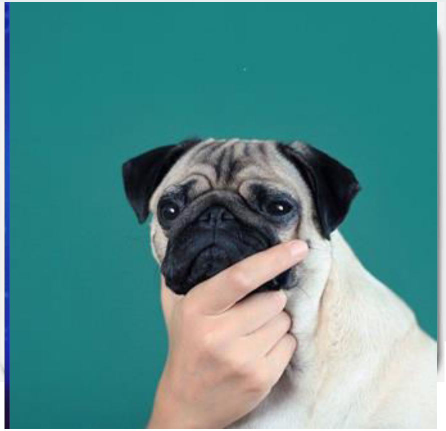
$$2\ nivele:\ 2\sqrt{n}$$

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

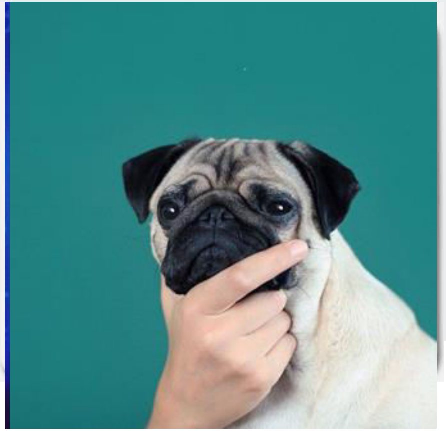
$$k\ nivele:\ k\sqrt[k]{n}$$

$$\lg(k)\ nivele:\ lg(k)\sqrt[lg(k)]{n}$$





Skip Lists: Numarul de elemente per nivel

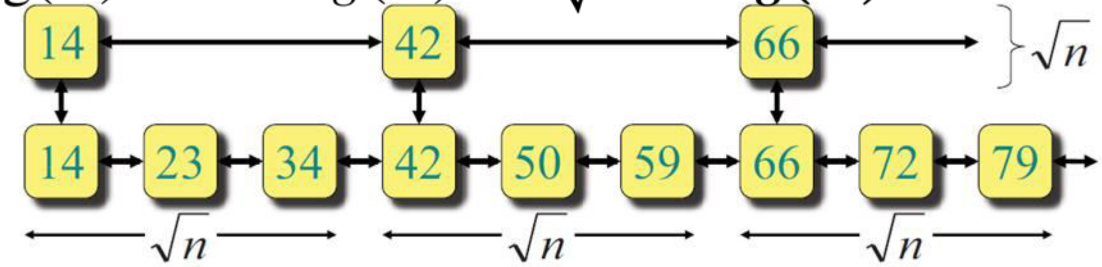


$$2\ nivele: 2\sqrt{n}$$

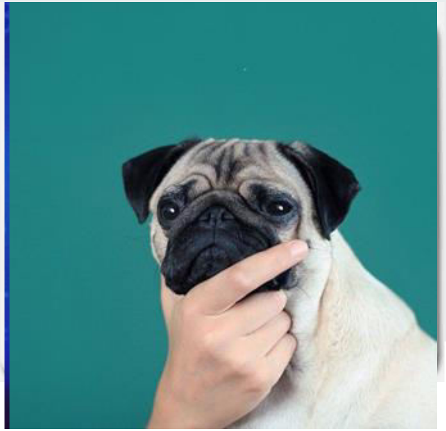
$$3\ nivele: 3\sqrt[3]{n}$$

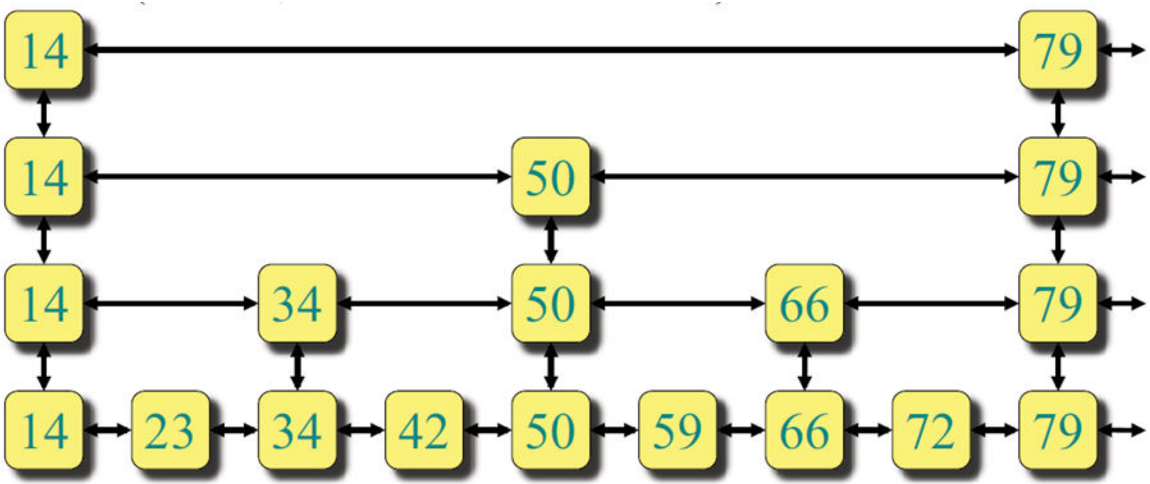
$$k\ nivele: k\sqrt[k]{n}$$

$$\lg(k)\ nivele: lg(k)\sqrt[lg(k)]{n} = \mathbf{2lg(n)}$$



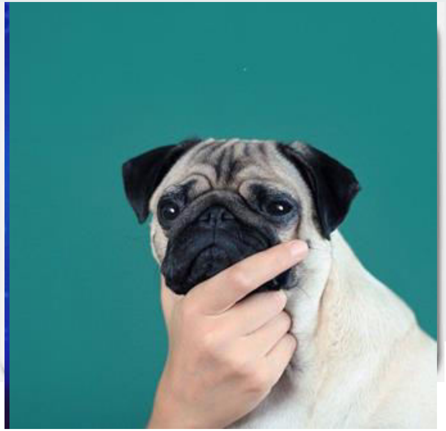
Skip Lists: Numarul de elemente per nivel

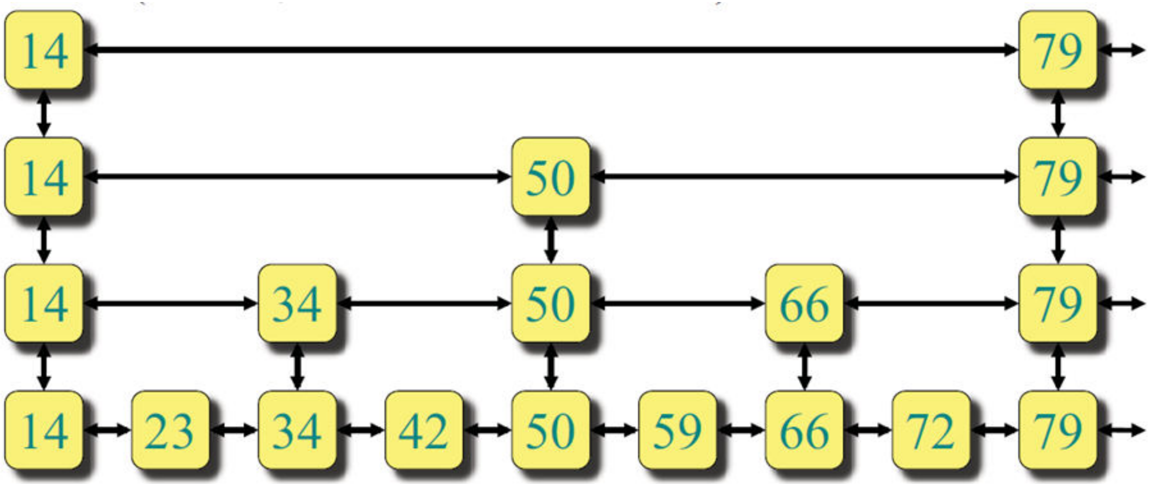




Cu ce seamana oare?

Skip Lists: Numarul de elemente per nivel

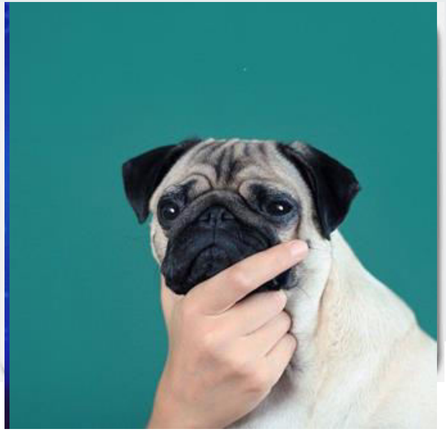


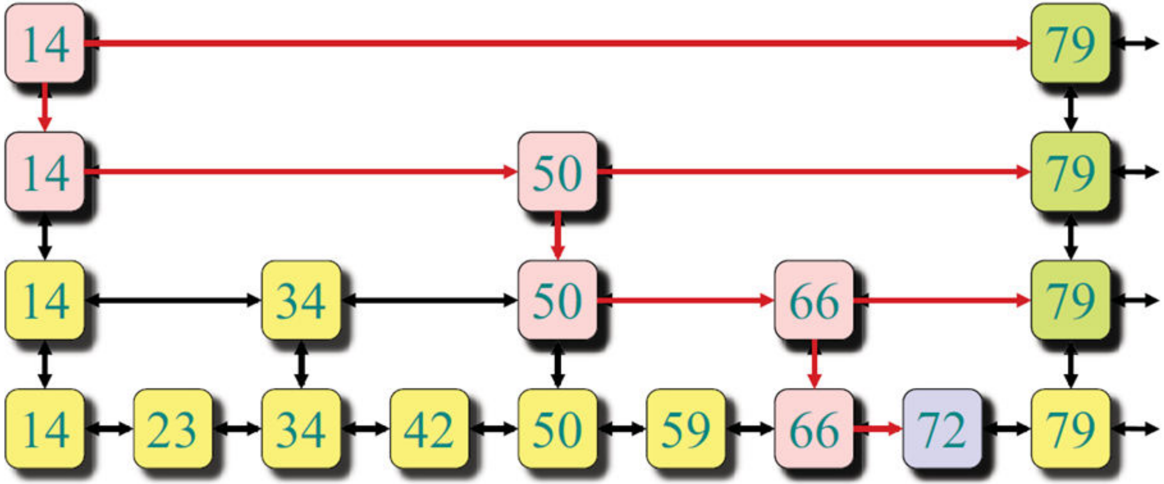


Cu ce seamana oare?

Un arbore!

Skip Lists: Search (X)





Search(72)

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

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 intoteauna toate elementele

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 intoteauna toate elementele

Q: Pe cate alte nivele inserez X?

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

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 intoteauna toate elementele

Q: Pe cate alte nivele inserez X?

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

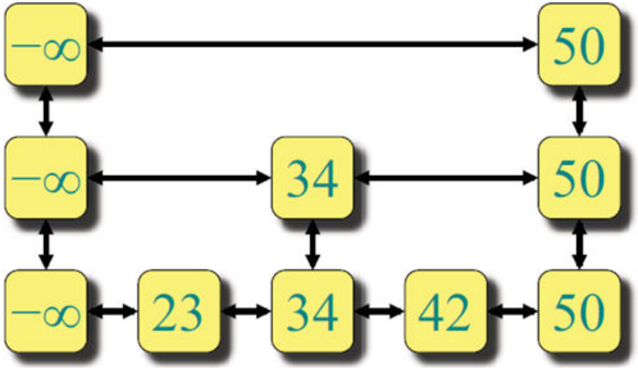
Consecinta: ½ dintre elemente vor fi doar pe nivelul 0. ¼ dintre elemente vor fi doar pe nivelele 0 si 1. ⅛ vor fi doar pe nivelele 0, 1 si 2, etc...

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 eelemntul de pe toate nivelele!

Skip Lists: How good are they?

[Whiteboard S23]

[Whiteboard S24]