Algoritmide keerukus

Konstantin Tretjakov (kt@ut.ee)

06. november 2004

Kava

- Programmi pikkus vs kiirus vs keerukus
- O–notatsioon
- Keerukuse hindamine
- Mida on praktikas vaja osata
- Boonus: $P \neq NP$ (ehk kuidas saada 1000000$)

Pikkus/Kiirus/Keerukus

```
for i = 1 .. 10
  samm1();
  samm2();
  samm3();
end;

for i = 1 .. 10
  samm4();
  samm5();
end;
```

50 sammu

```
for i = 1..10
  for j = 1..10
    samm1();
  end;
end;
```

100 sammu

Pikkus/Kiirus/Keerukus

- Sõltumatud mõisted:
  - Algoritmi pikkus — tema koodiridade arv
  - Algoritmi kiirus — kui palju aega kulub täitmiseks
  - Algoritmi keerukus — kuidas täitmise aeg sõltub sisendi suurusest
- Praktikas meid huvitab just **kiirus**. Selle hindamiseks kasutatakse **keerukuse** mõistet.

Kiirus = Täitmiseks kuluv aeg

- Algoritm täidetakse samm-sammult, ning kokku tehakse mingi arv samme $f(n)$, kus n on sisendi suurus või maht.
- Samm = "elementaarne operatsioon" mida arvuti täidab fikseeritud [konstantse] aja jooksul.
- Kui sammu täitmiseks kulub c sekundit, siis kogu programmi täitmiseks kulub $cf(n)$ sekundit, kusjuures c sõltub masina kiirusest.

$n \approx 3n + 4$

- Oletame, et algoritm A kulutab oma töö tegemiseks n sammu, ning algoritm B — $3n + 4$ sammu.
- Kui me võrdleme algoritmide kiirust ühel ja samal masinal, siis ilmselt on A alati kiirem kui B.
- Kui me aga paneme A jooksma arvutil mis teeb sammu sekundiga, aga B — arvutil, mis teeb neli sammu sekundiga, siis kui $n > 0$ töötab B kiiremini kui A.
- Seega mingis mõttes on algoritmid A ja B ekvivalentsed.
- *Keerukus* = "kiirus konstandi täpsuseni".

Kas kõik ei ole siis sarnased?

- Kui me vaatleme kiirusi konstandi täpsuseni, kas siis ei ole võimalik suvaline algoritm teha sama kiire kui suvaline teine?

Kas kõik ei ole siis sarnased?

- Kui me vaatleme kiirusi konstandi täpsuseni, kas siis ei ole võimalik suvaline algoritm teha sama kiire kui suvaline teine?
- Tehku algoritm A kokku n sammu, ning B — n^2 sammu. Siis sõltumata sellest, kui kiire arvuti peal me paneme B jooksma, mingist n-ist alates töötab A kiiremini.
- Täpsemalt: iga c jaoks leidub selline n_0 et iga $n > n_0$ jaoks $n^2 > cn$.

O-notatsioon

- Olgu antud funktsioonid f ja g. Ütleme et f kasv ei ületa g kasvu, kui leiduvad positiivsed arvud c ja n_0, mille korral:

$$\forall n > n_0 : \quad f(n) \leq cg(n)$$

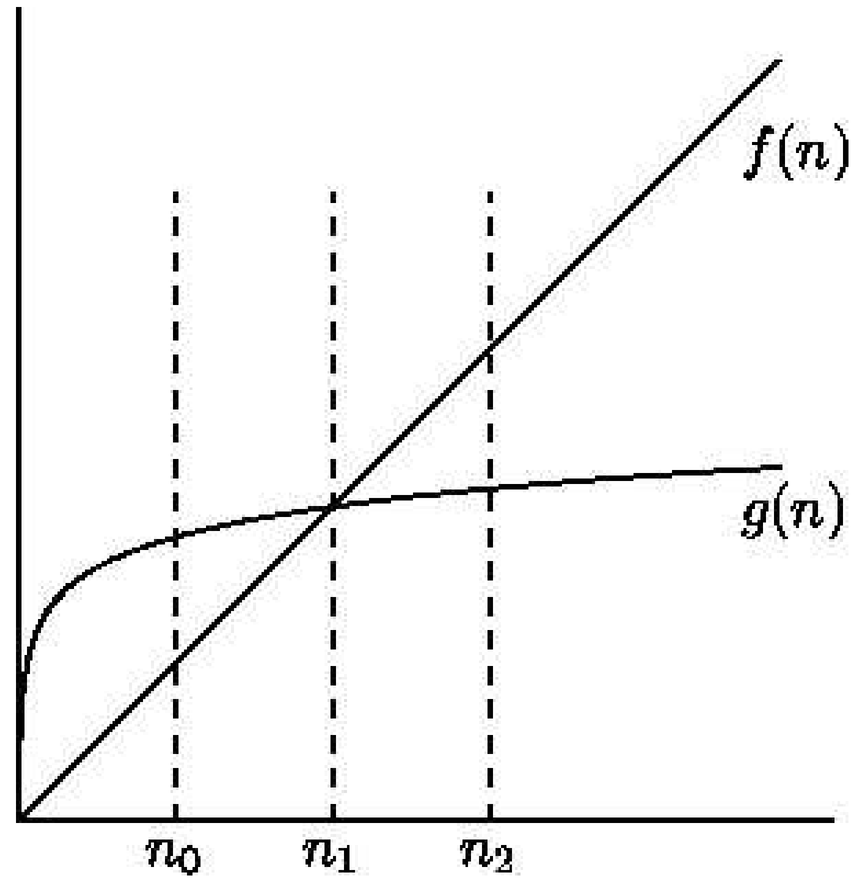
  Tähistame seda: $f = O(g)$ [eff on suur-oo geest].
- Vastupidist seost tähistame tähega Ω, s.t.

$$f = O(g) \quad \Leftrightarrow \quad g = \Omega(f)$$

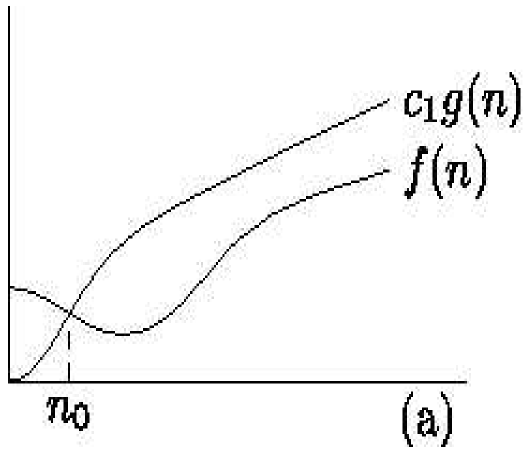
- Ekvivalentsust O-suhtes tähistame Θ-ga, s.t.

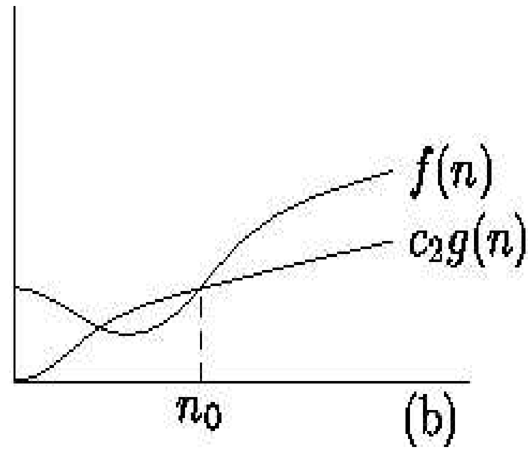
$$f = \Theta(g) \quad \Leftrightarrow \quad (f = O(g) \text{ ja } g = O(f))$$

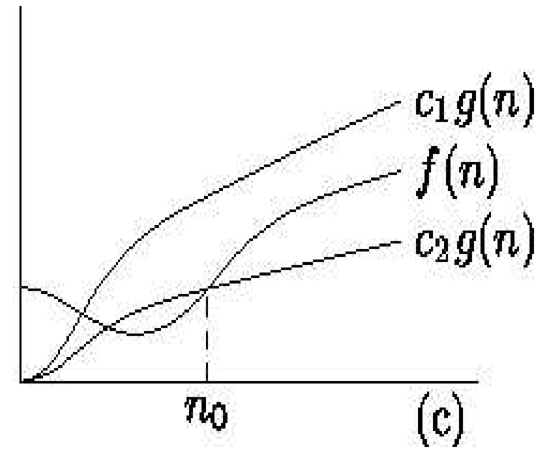
$g = O(f)$



$g = \Theta(f)$







Omadused

- $f = O(g) \quad \Leftrightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} > 0$
- $f = \Omega(g) \quad \Leftrightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $f = \Theta(g) \quad \Leftrightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} = c,\ 0 < c < \infty$
- $\forall c, d > 0 : \quad cf(n) + d = \Theta(f(n))$
- $f = O(g),\ g = O(h) \Rightarrow f = O(h)$
- $f = \Theta(h),\ g = O(h) \Rightarrow f + g = \Theta(h)$

Omadused

- $a_0 + a_1 n + a_2 n^2 + \cdots + a_k n^k = \Theta(n^k)$
- $n^k = O(2^n)$
- $\log_2^k(n) = O(n)$
- $\log_2(n) = \Theta(\log_3(n))$
- $2^n = O(3^n)$
- $1^k + 2^k + \cdots + n^k = \Theta(n^{k+1})$

Kokkuvõteks

- O-notatsioon lubab võrrelda algoritmide kiirust “konstandi täpsuseni”.
- Ta lubab lihtsalt kirjeldada algoritmide kiirusejärku: selle asemel et öelda “see algoritm töötab minu masina peal $3.05n^2 + n + 2\log_2(n)$ sekundit” saab öelda “see algoritm on $O(n^2)$”
- Kasutatakse mõisteid “keerukus parimal juhul”, “keerukus halvimal juhul”, “keskmine keerukus”.
- Peale ajalise keerukuse (*time complexity*) vaadeldakse ka “mahulist keerukust” (*space complexity*). Nii et omab mõtet lause “see algoritm on $O(n)$ aja suhtes ja $O(n^2)$ mälu suhtes”.

Keerukuse hindamine

Olgu meil ülesanne: leida etteantud massiivis `a` kaks võrdset elementi. Algoritmi sisendi mahuks nimetame antud massiivi suurust n.

Vaatame erinevaid lahendusi:

```
for i = 1 .. n
  for j = 1 .. n
    if (i != j) and (a[i] == a[j])
      then return a[i];
  end;
end;
return null;
```

Keerukuse hindamine

```
sort(a);
for i = 1..(n-1)
  if (a[i] == a[i+1])
    then return a[i];
end;
return null;
```

Keerukuse hindamine

```
H = new Hashtable;
for i = 1..n
  if (H.contains(a[i])) then return a[i];
  H.put(a[i]);
end;
return null;
```

Mis juhtub kui võtame paisktabeli asemel puu? ...ahela?
...massiivi?

Keerukuse hindamine

```
while (true)
  i = random(1 .. (n-1));
  j = random((i+1) .. n);
  if (a[i] == a[j]) then return a[i];
end;
```

Rekursiivsed algoritmid

Nende keerukuse hindamine nõuab mõnikord natuke mõtlemist. Näiteks:

```
function factorial(n)
  if (n == 0) then return 1;
  else return n * f(n - 1);
end
```

Rekursiivsed algoritmid

```
function fibonacci(n)
  if (n == 0) or (n == 1) then return 1;
  else return f(n-1) + f(n-2);
end
```

Rekursiivsed algoritmid

```
function mergesort(a)
  m = length(a) / 2;
  if (m == 0) return;
  else
    mergesort(a[1..m]);
    mergesort(a[(m+1)..n]);
    a = merge(a[1..m], a[(m+1)..n]);
  end
end
```

Mida tegelikult on vaja osata

- Oma algoritmi kirjutades saada aru, mis järku tema keerukus on.
- Keerukuse järgi hinnata, kui palju kulub aega kõige hullema testi jooksutamiseks.
- Ülesande spetsifikatsiooni järgi otsustada, mis keerukusega algoritmi otsitakse.

Tuntud keerukused

- Tsükkel: tüüpiliselt $O(n)$
- Kaks tsüklit üksteise sees: $O(n^2)$
- Sorteerimine: $n\log(n)$ või n^2
- Lühima tee leidmine graafis: $V\log(V) + E$ või V^2
- Graafi aluspuu leidmine: $E\log(E)$ või $E + V\log(V)$
- Alamhulkade läbivaatus: 2^n (või tihti isegi $2^n n$)
- Ümberjärjestuste läbivaatus: $n!$
  $(n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n = O(n^n))$
- jne...

Ajahinnangud

- Algoritmi umbkaudset keerukuse hinnangut on tegelikult vaja selleks, et hinnata, kui palju aega ta võtab maksimaalse suurusega testi peal.
- Hea on eeldada et arvuti teeb sekundis ≈ 1 kuni 10 miljonit sammu.
- Seega, kui ülesandes on öeldud et andmete maht on kuni 1000, ning teie algoritm on $O(n^2)$, siis läheb tal maksimaalse testi jaoks *ca* 1 sekund.
- Kui teie algoritm on aga $O(n^3)$, siis kukub ta maksimaalses testis läbi.

Ülesande keerulisuse hinnangud

- Kasulik on osata ülesande püstitusest arvata, mis keerukusega algoritmi nõutakse.
- Kui on öeldud et andmete maht/suurus on kuni:
  - 1 000 000 — peate leidma lineaarse algoritmi (s.t. $O(n)$), võib sobida ka $O(n \log n)$.
  - 1 000 — nõutakse ruutkeerukusega asja (s.t. $O(n^2)$).
  - 100 — kuupkeerukus.
  - 20 — ilmselt on lahendus seotud kõigi variantide läbivaatusega ning on eksponentsiaalse $O(2^n)$ või faktoriaalse keerukusega $O(n!)$.

Ülesande keerulisuse hinnangud

- Sageli on suhteliselt lihtne saada algoritm halva keerukusega, kuid keerukuse isegi väike parandamine on suhteliselt raske.
  - Kui näete et läheb läbi ka halb algoritm, ärge pingutage.
  - Tihti saab ajalist keerukust parandada mahulise keerukuse hinnaga (*time-space tradeoff*). Võtmesõnad: dünaamiline programmeerimine, eelarvutamine.
  - Mõnikord nõutakse mitte absoluutselt parimat lahendust, vaid parimat, mida te leida suudate. Ärge siis unustage taimerit kasutada!

Kuidas saada 1 000 000 $

- Ülesanded, mille lahenduse keerukus on üle polünomiaalse, ei ole praktikas lahendatavad. Seepärast vaadeldakse kõiki polünomiaalse lahendusega ülesandeid omaette klassina, mida nimetatakse P (*polynomial*).
- Mittepolünomiaalsete ülesannete seas on tähtsad sellised, mida me küll kiiresti lahendada ei oska, aga lahenduse õigsust saame kontrollida polünomiaalse ajaga (e.g. SAT, hamiltoni tsükkel, avaliku võtme krüpto, . . .). Nende ülesannete klassi nimetatakse NP (*nondeterministic polynomial*).

Kuidas saada 1 000 000 $

- Lahendamata ülesanne, mille lahendamise eest Clay matemaatika instituut pakub 1 000 000 $: tõestada formaalselt et $P \neq NP$ (või vastupidist, mis pigem ei kehti, aga keegi tõestanud ka ei ole). S.t. on vaja tõestada, et tõepoolest leiduvad ülesanded, mille lahenduse saab kiiresti kontrollida, kuid seda lahendust kiiresti leida ei saa.
- On olemas klass nn. $NP-complete$ ülesandeid. Need on sellised ülesanded, millele polünomiaalse ajaga lahenduse leidmisest järeldub, et kõikide NP-ülesannete jaoks leidub polünomiaalne lahendus (s.t. järeldub et $P = NP$).

Kuidas saada 1 000 000 $

- Seega, $P \neq NP$ tõestamiseks oleks vaja tõestada, et ühe sellise ülesande jaoks kiirest lahendust ei leidu.
- $P = NP$ tõestamiseks oleks vaja tõestada, et ühe sellise ülesande jaoks leidub kiire lahendus.
- Edasi mõelge ise.
  Infot saab siit:
  `http://www.claymath.org/millennium/`

Kõik

Küsimused?