TÖL403G GREINING REIKNIRITA

17. Jafnaðargreining 1

Hjálmtýr Hafsteinsson
Vor 2022

Í þessum fyrirlestri

- Bestun á fail-færslu í KMP
  - Sem hluti af reikniritinu sjálfu

DC 7.7

- Jafnaðargreining (*amortized analysis*)
  - Upphækkun á bitateljara
  - Ólíkar framsetninga á jafnaðargreiningu
    - Summa
    - Skattlagning
    - Rukkun
    - Stöðuorka
  - Hækkun og lækkun á teljara

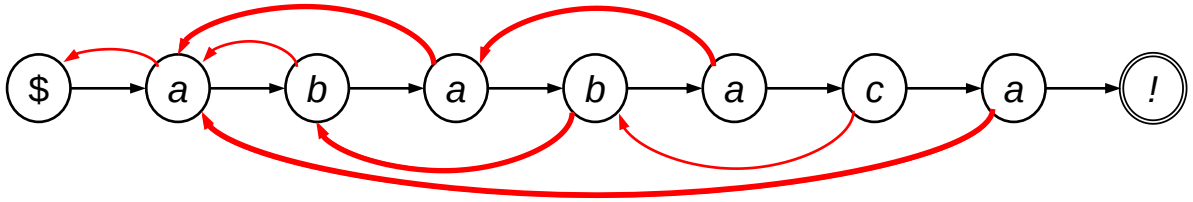
DC 9.1 − 9.3

Knuth-Morris-Pratt upprifjun

- Búa til stöðuvél úr mynstrinu á $O(m)$ tíma
- Keyra textann í gegnum hana á $O(n)$ tíma

- Upphaflega reikniritið:
  - Ef $T[i] \neq P[j]$ þá athugar reikniritið hvort $T[i]$ sé = $P[fail[j]]$, jafnvel þó $P[j] = P[fail[j]]$

Dæmi:



Prófa *a* aftur þó *a* passi ekki
Prófa *b* aftur þó *b* passi ekki
Prófa *a* aftur þó *a* passi ekki
Prófa *a* aftur þó *a* passi ekki

Sáum síðast að það er hægt að laga *fail*-fylkið eftirá

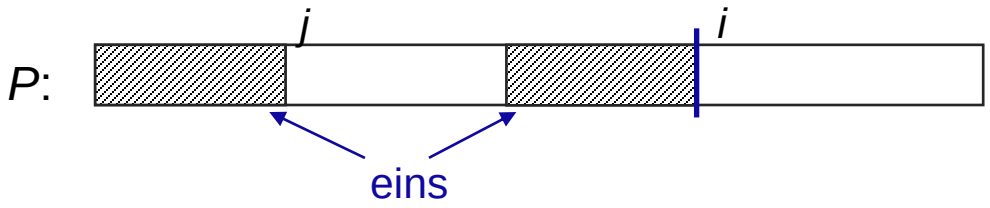
```
OptimizeFailure(P[1..m], fail[1..m]):
  for i ← 2 to m
      if P[i] = P[fail[i]]
          fail[i] ← fail[fail[i]]
```

Betra KMP reiknirit

- Getum líka byggt bestunina inn í upphaflega reikniritið

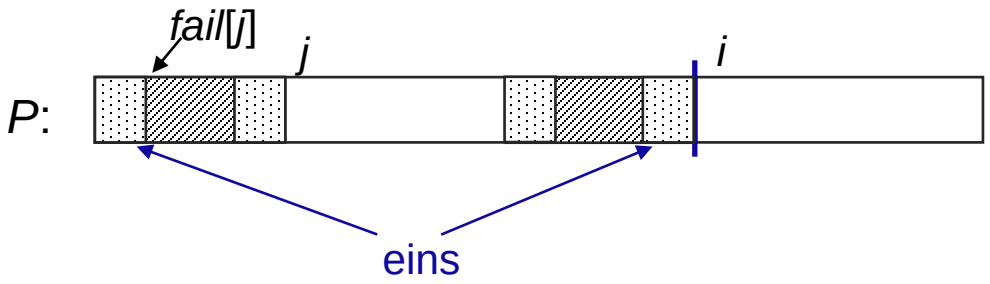
```
ComputeOptFailure(P[1..m]):
    j ← 0
    for i ← 1 to m
        if P[i] = P[j]
            fail[i] ← fail[j]
        else
            fail[i] ← j
        while j > 0 and P[i] ≠ P[j]
            j ← fail[j]
        j ← j + 1
```

Viðbót



Upphaflega var *fail*[*i*] sett sem *j* án þess að athuga *P*[*i*] og *P*[*j*]

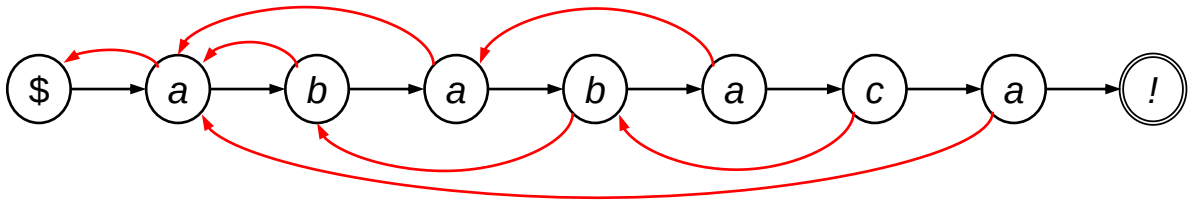
En nú athugum við hvort *P*[*i*] sé = *P*[*j*] og ef svo er þá förum við einni *fail*-ör lengra aftur



Áfram $O(m)$ tími, með aðeins hærri fasta

Sýnidæmi

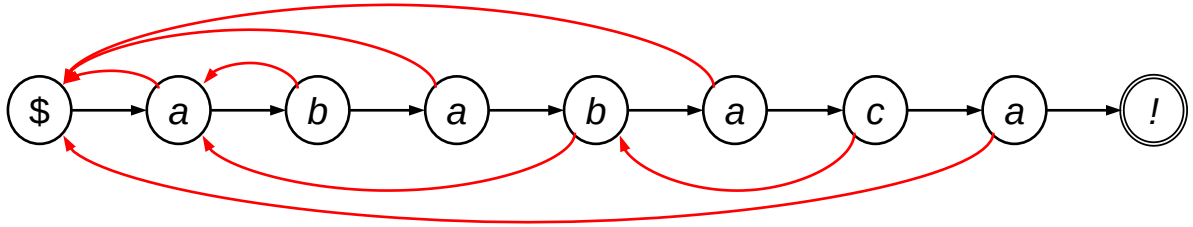
- Upphafleg stöðuvél:



Upphaflegt *fail*:

0	1	1	2	3	4	1
1	2	3	4	5	6	7

- Bestuð stöðuvél:



Bestað *fail*:

0	1	0	1	0	4	0
1	2	3	4	5	6	7

Gefur hraðvirkari strengleit − þurfum sjaldnar að bíða á sama textastaf

Jafnaðargreining (*amortized analysis*)

- Upphaflega sett fram af Robert Tarjan [1985]
- Önnur leið til að meta kostnað við reiknirit
- Gagnast þegar einstakar aðgerðir hafa áhrif á hver aðra
  - Ef við gerum þessa aðgerð þá munu aðrar aðgerðir dýrari
  - eða, ef ein dýr aðgerð er framkvæmd þá verða aðrar aðgerðir ódýrari
- Söfnum þá upp kostnaðinum og dreifum honum á allar aðgerðirnar
  - Þetta er versta tilfellis greining

Einfalt verkefni

- Hækka tvíundarteljara (*binary counter*) um 1

  Höfum tvíundarteljara með gildið $n-1$, hækkum hann í n

  Geymdur í ótakmarkaða fylkinu B $\quad$ $B[i] = 1$ ef og aðeins ef 2^i er í summunni $n-1$

Reiknirit:

```
INCREMENT(B[0..∞]):
  i ← 0
  while B[i] = 1
      B[i] ← 0
      i ← i + 1
  B[i] ← 1
```

Til dæmis: $1000 \rightarrow 1001$

$1001 \rightarrow 1010$

$1010 \rightarrow 1011$

Tími:

Ef fyrstu (neðstu) k bitarnir eru 1 þá tekur reikniritið $O(k)$ tíma

Til dæmis:

$11111 \rightarrow 100000$

Tímaflækja tvíundarteljara

Ef teljarinn *B* táknar tölu á milli 0 og *n*, þá er versta tilfellis tími reikniritsins *Increment* $O(\log(n))$

En ef við köllum *n* sinnum á *Increment* með upphafsgildinu 0?

Tekur það $O(n \cdot \log(n))$ tíma?

En það eru bara örfáar aðgerðir sem eru dýrar og þær eru dýrar vegna ódýru aðgerðanna sem komu á undan

Til dæmis:

11100	→	11101	1 br.
11101	→	11110	2 br.
11110	→	11111	1 br.
11111	→	100000	6 br.

Hægt að sýna að heildartími á *n* köllum á *Increment* er $O(n)$

Með jafnaðargreiningu (*amortized analysis*)

Getum þá sagt að jafnaðartími hverrar aðgerðar sé $O(1)$

Jafnaðargreining

- Munum skoða 4 aðferðir til að gera jafnaðargreiningu
  - Allar aðferðirnar eru jafngildar
  - en henta misvel fyrir einstök verkefni
- Summa (*summation*, *aggregation*)
  - Finna versta tilfellis tíma fyrir runu *n* aðgerða, deila svo í hann með *n*
- Skattlagning (*taxation*, *accounting*)
  - Láta hverja aðgerð borga gjald, sem nægir til að borga fyrir allar runur *n* aðgerða
- Rukkun (*charging*)
  - Dýrar aðgerðir geta rukkað fyrri aðgerðir um kostnað

mjög svipuð skattlagningu

- Stöðuorka (*potential*)
  - Sumar aðgerðir byggja upp stöðuorku í gagnagrindinni sem hægt er að nota síðar

Summuaðferð

- Þegar við hækkum teljarann frá 0 til *n* þá breytast ekki allir bitarnir

Auðvelt að sjá:

Biti *B*[0] breytist í hvert sinn

Biti *B*[1] breytist í annað hvert sinn

Biti *B*[2] breytist í fjórða hvert sinn

⋮

Sjáum þá: Biti *B*[*i*] breytist $\lfloor n/2^i \rfloor$ sinnum þegar talið er frá 0 til *n*

Heildarfjöldi bitabreytinga er

$$\sum_{i=0}^{\lfloor \log_2 n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n$$

Hægt að sýna að fjöldi bitabreytinga er nákvæmlega $2n$ − (fjöldi 1-bita í n)

Jafnaðartími *Increment*

- Heildarfjöldi bitabreytingar við að telja frá 0 til n er $2n$

  Að meðaltali tekur því hver hækkun 2 bitabreytingar

  Athugið að hér er meðaltalið yfir allar aðgerðirnar, ekki yfir mögulega keyrslutíma hverrar aðgerðir

  Dreift yfir allar aðgerðirnar, svipað og jafngreiðslulán!

  Segjum þá að hver hækkun taki 2 bitabreytingar að jafnaði (*amortized*)

Höfum þá:

versta tilfellistími *Increment* er $O(\log(n))$

en jafnaðartími *Increment* er $O(1)$

Ef $T(n)$ er versta-tilfellis tími fyrir runu n aðgerða þá er jafnaðartími hverrar aðgerðar $T(n)/n$

Skattlagningaraðferð

- Leggjum skatt á hverja hækkun og notum hann til að borga fyrir framtíðaraðgerðir
- Við hverja hækkun (*Increment*) er lagður $2 skattur

  $1 fer í að borga fyrir að breyta 0 í 1, en hinn er geymdur til að borga fyrir breytinguna til baka, úr 1 í 0

Dæmi:

Hækkun	Skattur	Geymt
0000 → 0001	2	1
0001 → 0010	2	1
0010 → 0011	2	2

Notum geymda $1 til að borga *B*[0] breytingu. Annar $1 af skattinum borgar fyrir *B*[1] breytingu. Hinn geymdur

Eigum nú fyrir báðum breytingunum til baka

Þurfum að finna út réttu upphæðina til að skattleggja og sýna að hún virki (þ.e. eigum alltaf fyrir aðgerðunum)

Rukkunaraðferð

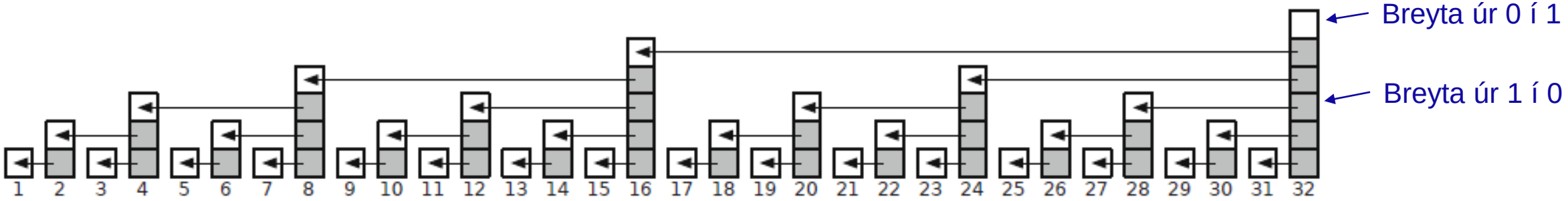
- Þessi aðferð rukkar fyrri skref um kostnað við seinni skref
  - Í skattlagningu leggja fyrri skref fyrir upphæð vegna seinni skrefa

Dæmi:

Þegar bita er breytt frá 1 til 0 þá rukkum við það skref sem setti bitann sem 1

Þegar við þurfum að breyta *k* bitum í keyrslu á *Increment*, þá borgum við sjálf fyrir eina aðgerð og rukkum fyrri bita um hinar $k-1$ aðgerðirnar

Hver aðgerð kostar því 2:
1 fyrir að breyta einum bita og
1 sem einhver framtíðaraðgerð mun rukka okkur



Stöðuorkuaðferðin

- Þetta er almennasta aðferðin (en líka oft flóknust)

  Segjum að sumar aðgerðir byggi upp stöðuorku (*potential energy*) í gagnagrindinni sem hægt er að nota síðar til að borga fyrir aðrar aðgerðir

  Lát D_i vera gagnagrindina eftir i aðgerðir á hana og Φ_i vera stöðuorku hennar

  c_i er raunkostnaður i-tu aðgerðarinnar, sem breytir D_{i-1} í D_i

  Þá er jafnaðarkostnaður i-tu aðgerðarinnar: $a_i = c_i + \Phi_i - \Phi_{i-1}$

  Raunkostn. — Mismunur í stöðuorku

Heildar jafnaðarkostnaður n aðgerða er þá:

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = (\sum_{i=1}^{n} c_i) + \Phi_n - \Phi_0$$

Munur í stöðuorku frá upphafs gagnagrind

Viljum alltaf hafa að

$$\Phi_i - \Phi_0 \geq 0$$

Sýnidæmi um stöðuorku

- Í tvíundarteljara verkefninu getum við skilgreint stöðuorkuna Φ_i sem fjölda bita sem eru 1
  - Upphaflega eru allir bitarnir 0, svo $\Phi_0 = 0$

Kostnaður:

Raunkostn. → $c_i =$ (fjöldi bita breytt úr 0 í 1) + (fjöldi bita breytt úr 1 í 0)

Breyting í stöðuorku → $\Phi_i - \Phi_{i-1} =$ (fjöldi bita breytt úr 0 í 1) − (fjöldi bita breytt úr 1 í 0)

Φ_i er fjölda bita sem eru 1

Styttist út!

Höfum þá að

$a_i = c_i + \Phi_i - \Phi_{i-1} =$ 2*(fjöldi bita breytt úr 0 í 1)

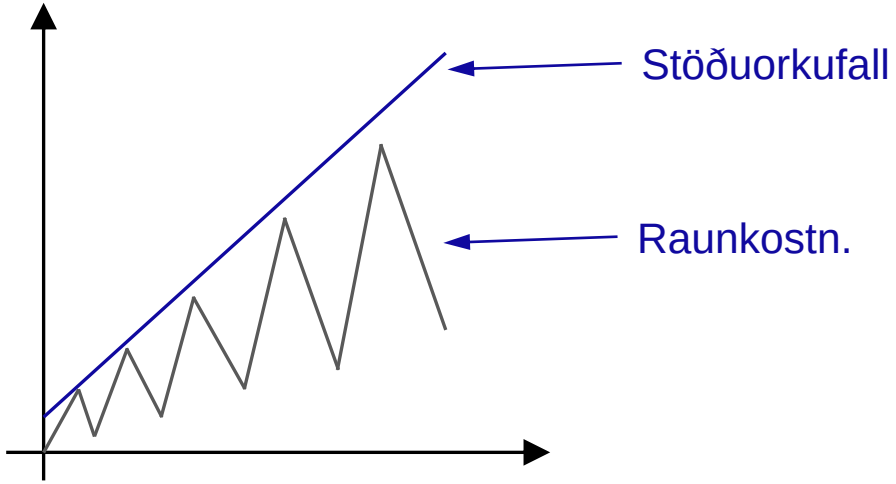
Reikniritið *Increment* breytir aðeins einum bita úr 0 í 1, svo jafnaðarkostnaður *Increment* er 2

Stöðuorkufall

- Hér er stöðuorkan nákvæmlega uppsafnaðir skattar í skattaaðferðinni
  - Fáum því nákvæmlega sömu útkomu
- Í sumum verkefnum er engin eðlileg leið til að túlka breytingu í stöðuorku sem "skatt" eða "rukkun"
- Stöðuorkuaðferðin gefur okkur almennari leið
  - Þegar ekki er hægt að dreifa kostnaðinum frá einni aðgerð til annarar

Getur verið erfitt að finna gott stöðuorkufall, sem passar vel

Viljum hafa þannig að það sé þétt (*tight*)



Hækkun og lækkun teljara

- Ef við leyfum að tvíundarteljarinn sé lækkaður líka þá getur það tekið langan tíma

Dæmi: Höfum 2^k-1, hækkum í 2^k, lækkum í 2^k-1

Kostar $k+1$ bitaaðgerðir

Kostar $k+1$ bitaaðgerðir

Sniðug hugmynd:

Táknum teljarann með tveimur bitastrengjum (P, N)

Gildi teljarans er $P - N$

P og *N* túlkaðar sem tvíundartölur

Eina skilyrðið sem P og N verða að uppfylla er að $\underline{P \wedge N = 0}$

P og *N* OG-aðar bita fyrir bita eru 00...00

eða: Það má aldrei nema annar af bitunum $P[i]$ og $N[i]$ vera 1

Sýnidæmi um tvöfalda teljara

- Það er hægt að tákna sérhverja heiltölu á óendanlega marga vegu með þessari aðferð
  - Nema töluna 0, hún er alltaf (0, 0)

Dæmi: (01, 00) táknar 1, því 1−0 = 1

(010, 01) táknar 1, því 2−1 = 1

(100, 011) táknar 1, því 4−3 = 1 o.s.frv.

Hvað með 3−2?

Nei, (011, 010) uppfyllir ekki skilyrðið!

(1001, 0100) táknar 5, því 9−4 = 5

(0100, 1001) táknar −5, því 4−9 = −5

Getum nú líka táknað neikvæðar tölur

Hækkun og lækkun

- Höfum nú föllin *Increment* og *Decrement*:

```
INCREMENT(P,N):
  i ← 0
  while P[i] = 1
        P[i] ← 0
        i ← i + 1
  if N[i] = 1
        N[i] ← 0
  else
        P[i] ← 1
```

Alveg eins og í upphaflega *Increment*

Verðum að athuga hvort *N*-bitinn er 1 áður en við setjum *P*-bitann

```
INCREMENT(P,N):
  i ← 0
  while P[i] = 1
        P[i] ← 0
        i ← i + 1
  if N[i] = 1
        N[i] ← 0
  else
        P[i] ← 1
```

```
DECREMENT(P,N):
  i ← 0
  while N[i] = 1
        N[i] ← 0
        i ← i + 1
  if P[i] = 1
        P[i] ← 0
  else
        N[i] ← 1
```

Í stað þess að hækka *P* um 1, þá lækkum við það um 3 og lækkum *N* um 4

Dæmi:

P = 1001
N = 0100
P − *N* = 5

Inc →

P = 1010
N = 0100
P − *N* = 6

Hækkum bara *P*

Inc →

P = 1011
N = 0100
P − *N* = 7

Hækkum bara *P*

Inc →

P = 1000
N = 0000
P − *N* = 8

Breyting:

$(P - 3) - (N - 4)$

$= P - 3 - N + 4$

$= P - N + 1$

Lækkun

- Lækkun virkar svipað, nema víxlað á *P* og *N*:

```
DECREMENT(P,N):
  i ← 0
  while N[i] = 1
      N[i] ← 0
      i ← i + 1

  if P[i] = 1
      P[i] ← 0
  else
      N[i] ← 1
```

Getum ekki hækkað *N* um 1, svo við lækkum það um 1 og lækkum *P* um 2

Dæmi:

P = 0110	Dec →	P = 0100
N = 1001		N = 1000
$P - N = -3$		$P - N = -4$

Almennt:

Ef hækkun veldur því að bitar $P[k]$ og $N[k]$ verða báðir 1 þá:

Lækkum við P um $2^k - 1$ og lækkum N um 2^k

Svo $(P - (2^k - 1)) - (N - 2^k) = P - 2^k + 1 - N + 2^k$

$= P - N + 1$

Samskonar fyrir lækkun, nema víxlað á *P* og *N*

Fyrirlestraæfingar

1. Strengurinn "ababa" hefur *fail*-fylkið [0, 1, 1, 2, 3]. Reiknið bestaða útgáfu af fylkinu, þannig að textastafurinn sé ekki borinn aftur saman við sama tákn í mynstrinu.

2. Rökstyðjið að ef við bættum við einfalda tvíundarteljarann aðgerðinni *Lækka* (*Decrement*), þá gætu n aðgerðir á k-bita teljara tekið $O(nk)$ tíma.

3. Framkvæmið *Lækka* á samsetta teljaranum (1010, 0101) (sem er (10, 5), þ.e. $10-5 = 5$)