Gagnaskipan glósur

**3. kafli**

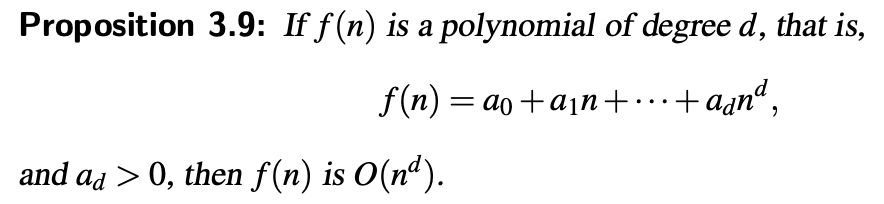
* Data structure er leið til að skipuleggja og sækja gögn
* Algóriðmi er skref fyrir skref uppskrift af því hvernig eitthvað er gert á ákveðnum tíma
* Skoðum running time á forritum ásamt minnismagni
* Taka tíma á forriti:
  + from time import time  
    start time = time( )  
    run algorithm  
    end time = time( )  
    elapsed = end time − start time
* Það getur verið tímamunur á milli keyrlsna þó að forritinu hafi ekkert verið breytt, timeit moduleinn gerir þetta aðeins betra.
* Til að meta niðurstöður er gott að vera með á x-ás input stærð og y-ás tímann.
* Gallar við þessa aðferð (nota tilraunir):
  + Erfitt að bera saman tvo algóriðma án þessa ð vera í sömu tölvu.
  + Bara hægt að gera tilraunir á takmörkuðum inputs, running tími fyrir input sem eru ekki í tilrauninni er ekki tekinn með, gætu verið mikilvæg input!
  + Það þarf að vera 100% búið að gera kóðann til að tékka á tímanum.
* Okkar markmið er að:
  + Bera saman efficiency tveggja algóriðma, skiptir ekki máli í hvaða tölvum
  + Það er gert með því að lýsa mjög vel algóriðmanum án þess að implementa
  + Tekur tillit til allra inputa
* Svo við notum ekki tilraunir
* Það sem við gerum:
  + Framkvæmum analysis á high level description á algoriðma
  + Skilgreinum grunnaðgerðir
    - Grunnaðgerð er með fastan execution tíma.
    - Í stað þess að meta tímann á öllum grunnaðgerðum teljum við hve margar eru framkvæmdar og notum þá tölu, t, sem mælikvarða á keyrslutímann
  + Til að meta stækkun á keyrslutíma er fall tengt við hvern algóriðma sem líkir eftir fjölda aðgerða sem eru framkvæmdar sem fall af input stærð, n. Fallið er f(n).
* Það fer mismikill tími í hvert input, erfitt að nota meðaltal af því gætum fengið input sem er bara með hæstu gildi.
* Í bókinni verður alltaf tekið worst case tímana (hæstu). Leiðir yfirleitt til betri algóriðma.
* Sjö föll sem eru notuð í bókinni til að greina algóriðma:
  + Constant function
    - F(n) = c
    - Öll input fá gildið c.
    - Notum oftast g(n) = 1 sem er sama og c\*g(n)
    - Þegar einhver aðgerð tekur alltaf sama tíma, sama hvert inntakið er
  + Logarithm function
    - F(n) = logb(n), b>1.
    - x = logb(n) þá og því aðeins af b^x = n
    - log(1) = 0
    - í bókinni er b-ið yfirleitt 2 þannig ef það stendur ekki þá er það 2
    - Hægt að finna minnsta int stærri en eða jafn logb(n) með því að deila n með b. Minnsta intið er þá hversu oft við náum því.
      * log3 27à is 3, because ((27/3)/3)/3 = 1. Likewise, àlog4 64à is 3, because ((64/4)/4)/4 = 1, and àlog2 12à is 4, because (((12/2)/2)/2)/2 = 0.75 ≤ 1.
    - T.d. ef ég er með lista og framkvæmi eina aðgerð og þá þarf ég bara helminginn af listanum, framkvæmi svo aðra og þá helminginn af því osfrv.
  + Linear function
    - F(n) = n
    - Gerist þegar við þurfum að gera eina basic aðgerð fyrir hvert af n elementum.
  + N-log-N function
    - F(n) = n\*log(n)
    - Stækkar hraðar en linear en hægar en quadratic
  + Quadratic function
    - f(n) = n^2
    - T.d. þegar nested loop og innri loopan framkvæmir línulegt magn af aðgerðum, ytri loopan er svo líka framkvæmt línulega oft.
    - Getur líka verið þegar nested loops og fyrsta gerir eina aðgerð, önnur tvær, þriðja þrjár osfrv
      * 1+2+3+···+(*n*−2)+(*n*−1)+*n* = *n*(*n*+1)/ 2
  + Cubic function and other polynomials
    - F(n) = n^3
    - Kemur sjaldnar til sögunnar en það sem fyrir hefur komið
    - Polynomial (margliða) fall er:
      * *f*(*n*) = *a*0 +*a*1*n*+*a*2*n*2 +*a*3*n*3 +···+*adnd*
    - Keyrslutími margliðna af lægra stigi er styttri en af hærra stigi
    - Summation:A close up of a logo

      Description automatically generated
    - Summurithátturinn gefur styttri leið í að útskýra föllin hér að ofan
  + Exponential Function
    - F(n) = b^n
    - Þar sem b er jákvæður fasti, kallast base
    - Oftast er b=2
    - Td ef það er loop sem framkvæmir fyrst eina aðgerð en tvöfaldar svo
* Geometric sums
  + Loopa þar sem í hver ítrun tekur inn margföldunarþátt lengri en fyrri ítrun
  + *an*+1 − 1/ *a*−1
  + For example, everyone working in computing should know that
  + 1+2+4+8+···+2*n*−1 = 2*n* −1,  
    for this is the largest integer that can be represented in binary notation using *n* bits.
* Samantekt af 7 föllunum!A screenshot of a cell phone

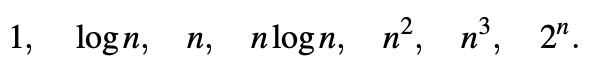
  Description automatically generated
* Algoriðmar með quadric eða cubic keyrslutíma eru óhentug og með exponential eru nánast ómöguleg nema fyrir mjög lítil input.
* A close up of a piece of paper

  Description automatically generated
* - floor function, stærsta int minna eða jafnt og x
* - ceiling function, minnsta int stærra eða jafnt og x

**Asymptotic analysis**

* Big oh notation - f(n) er O(g(n)) ef f(n)<=c\*g(n) fyrir n>= n0. Þar sem c er fasti stærri en 0 og n0 er jákvæð heil tala stærri en 1. Stærra en eða jafnt og annað fall.
* Big oh er notuð til að finna keyrslutíma
* Big oh leyfir okkur að hunsa fastaþætti og low order terms og fókusa á stærri hluta falla sem hafa áhrif á stækkun þess.
* 5*n*4 +3*n*3 +2*n*2 +4*n*+1 is *O*(*n*4) því 5*n*4+3*n*3+2*n*2+4*n*+1≤(5+3+2+4+1)*n*4 =*cn*4, for *c*=15,when *n*≥*n*0 =1.
* 
* Segja alltaf lægsta veldi sem við getum, ef það er O(n^2) þá er það líka O(n^3) en -segjum 2.
* Ef ég er með lúppu sem er línuleg og aðra nested sem er lógaríðmísk þá er hún O(n\*log(n))
* So, for example, we would say that an algorithm that runs in worst-case time 4*n*2 + *n* log *n* is a ***quadratic-time*** algorithm, since it runs in *O*(*n*2) time. Likewise, an algorithm running in time at most 5*n* + 20 log *n* + 4 would be called a ***linear-time*** algorithm.
* Big omega - minna en eða jafnt og annað fall
  + A screenshot of a cell phone

    Description automatically generated
  + Ath stærra og jafnt og merkið er öfugt við big oh
* Big Theta - stækka á sama hraðaA screenshot of a cell phone

  Description automatically generated
* Ef ég er með tvö forrit og annað er O(n) og hitt O(n^2) þá er keyrslutími O(n) minni fyrir öll n sem eru stærri en 1.
* Í stækkandi röð:
* 
* Það er hægt að leysa fleiri vandamál á stuttum tíma eftir því hve öflug tölvan er.
* Ef það er mjög stór fasti fyrir framan big oh þarf að skoða það sérstaklega. Ef t.d. það er verið að bera saman 10^100n og 10nlog(n) þá myndi maður nota big oh af nlog(n) í stað n!
* nlog(n) er metið efficient og n^2 fyrir lág n en 2^n verður nánast aldrei efficient.
* array er listi
* len(list) og list[i] er O(1)
* Prefix averages - Runa S með n tölur, við viljum fá runu A þannig að A[j] sé meðaltal elementa S[0},…,S[j] fyrir j=0,…,n-1. Þ.e.:
* A picture containing object

  Description automatically generated

**Simple justification techniques**

* Hægt að nota dæmi eða mótdæmi til að sanna staðhæfingu
* Contrapositive: To justify the statement “if p is true, then q is true,” we establish that “if q is not true, then p is not true” instead
* DeMorgan’s Law: This law helps us deal with negations, for it states that the negation of a statement of the form “p or q” is “not p and not q.” Likewise, it states that the negation of a statement of the form “p and q” is “not p or not q.”
* Contradiction: we establish that a statement q is true by first supposing that q is false and then showing that this assumption leads to a contradiction
* Líka hægt að nota þrepun!
* Loop invariants:
* To prove some statement L about a loop is correct, define L in terms of a series of smaller statements L0,L1,...,Lk, where:
* 1. The initial claim, L0, is true before the loop begins.  
  2. If Lj−1 is true before iteration j, then Lj will be true after iteration j.
* 3. The final statement, Lk, implies the desired statement L to be true.

**Glósur í tíma**

Hægt að vera með O(log(n)) tímaflækju til að finna stak í lista ef hann er sortaður, skippum helmingnum af listanum í hverju skrefi.

O(1) dæmi er t.d. að appenda lista því það tekur alltaf jafn langan tíma óháð fjölda staka

Ef tímaflækjan er margfeldi þá erum við að vinna með nested loopur, þ.e. n\*log(n) eða n^2 osfrv..

Ef það eru tveir forritsbútar þá er hægt að segja + á milli tímaflækjanna þeirra en þá er tímaflækjan í rauninni bara sú sem er lengri, t.d. ef annar búturinn er log(n) og hinn n þá er tímaflækjan n+log(n) en við segjum bara O(n) því það tekur lengri tíma. Tökum líka fasta út ef t.d. 2\*n þá er það O(n). Líka ef það eru tveir bútar af n þá er það samt bara O(n)

T.d. hér er heildartímaflækjan O(n)

n = 1000000

i = 1

# log(n)

counter = 0

while (i<n):

i \*= 2

counter += 1

print(counter)

# n

for j in range(n):

print(j)

Dæmi um log(n) - skippum helmingnum af stökunum í hverju skrefi:

n = 1000000

i = 1

counter = 0

while (i<n):

i \*= 2

counter += 1

print(counter)

Dæmi um n

for j in range(n):

print(j)

Dæmi um nlog(n):

# nlog(n)

# n þessi ytri

for j in range(n):

print(j)

i = 1

counter = 0

#log(n) er innri

while (i < n):

i \*= 2

counter += 1

Dæmi um n^2:

# n^2, ytri er n og innri er n

for bla in range(n):

print(bla)

for bla in range(n):

var = 1+1

Dæmi um n^3:

# n^3, ysta er n, miðjan er n og innsta er n

for bla in range(n):

print(bla)

for bla in range(n):

var = 1+1

for bla in range(n):

var2 = 2+2

Dæmi um 2^n - í hverju skrefi tvöfalda ég fjölda “skrefa”, kalla í fallið inní fallinu

# 2^n

def fib(n):

if n == 1:

return 1

if n == 2:

return

return fib(n-1) + fib(n-2)