Gagnaskipan glósur

**3. kafli**

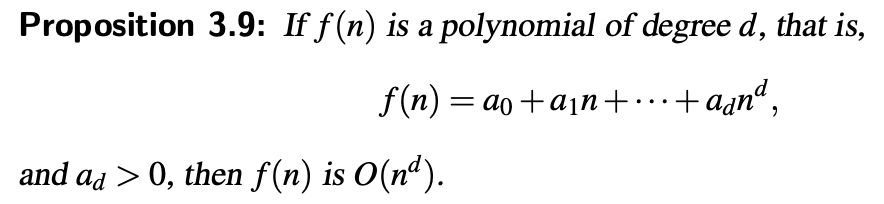
* Data structure er leið til að skipuleggja og sækja gögn
* Algóriðmi er skref fyrir skref uppskrift af því hvernig eitthvað er gert á ákveðnum tíma
* Skoðum running time á forritum ásamt minnismagni
* Taka tíma á forriti:
  + from time import time  
    start time = time( )  
    run algorithm  
    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

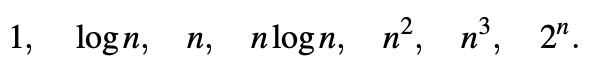
  Description automatically generated
* Algoriðmar með quadric eða cubic keyrslutíma eru óhentug og með exponential eru nánast ómöguleg nema fyrir mjög lítil input.
* A close up of a piece of paper

  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:

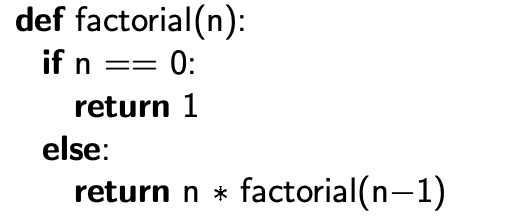
return 1

if n == 2:

return

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

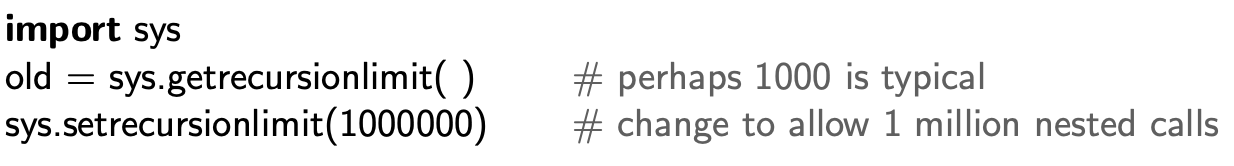
**4. kafli – recursion**

* Recursion = Fall kallar í sjálft sig einu sinni eða oftar eða a data structure relies upon smaller instances of the very same type of structure in its representation.
* Fjögur dæmi um recursion:
  + Factirial function (n!): Fall kallar í sjálft sig n sinnum
    - Dæmi:
    - 
  + English ruler: Reglustika með major tick length og svo nokkrar minor tick lengths. Þegar lengd bilsins minnkar um helming minnkar lengd ticks um 1.
    - In general, an interval with a central tick length *L* ≥ 1 is composed of:
    - • An interval with a central tick length *L* − 1
    - • A single tick of length *L*• An interval with a central tick length *L* − 1
  + Binary search: Notað til að finna target value inní runu af n elements. Ef runan er óröðuð er hægt að fara í gegnum öll element þar til targetið er fundið (eða finnst ekki ef það er farið í gegnum alla rununa) – þetta kallast sequential search, keyrir á O(n). Ef runan er sorted og „indexable“ er hægt að skipta henni upp í helming, athuga í hvor helmingnum hún er og skipta þeim helming aftur í helming osfrv – kallast binary search, keyrir á O(log(n)).
  + File system: Stýrikerfi kalla í möppur á recursive hátt. Efsta mappan er með fleiri möppur sem eru svo með fleiri osfrv. Stýrikerfin leyfa að vera með nested möppur svo lengi sem það er nóg minni en á endanum þarf bara að vera files, engar submöppur.
    - Immediate disk space er minni notað í hverri entry
    - Cumulative disk space r minni notað í þessu entry og öllum nested features.
    - Þetta cumulative disk space er hægt að reikna með recursive algorithm. Það er samasem immediate disk space plús summan af cumulative disk space af nested entries.
    - Os module hjá python:
      * os.path.getsize(path) – skilar immediate disk usage (í bytes) fyrir fileinn eða möppuna sem er identified með path strengnum
      * os.path.isdir(path) – returns True ef strengurinn path er mappa, annars False
      * os.listdir(path) – skilar lista af strengjum með nöfnum á öllu sem er í möppunni skilgreindri af path.
      * os.path.join(path, filename) – setur saman path streng við filename, notar / fyrir mac, skilar strengnum sem er full path to the file.

4.2 - analyzing recursive algorithms

* Tökum tímann sem eitt fall tekur í einu og leggjum svo saman.
* Factorial functions – hvert skipti er O(1) og það eru n+1 skipti þannig að tímaflækjan er O(n).
* English ruler – í dæmi - For c ≥ 0, a call to draw interval(c) results in precisely 2^c − 1 lines of output.
* Binary search – O(log(n)) fyrir sorted sequence með n elementum.
* Disk space usage – n er fjöldi skjala og mappa, ferð bara einu sinni inn í hvert skjal/möppu. Tíminn er O(n) því hvert recursive call er O(1). Ef maður tekur worst case væri það O(n^2) en við notum cumulative effect í stað þess að skoða worst case, það er kallað amortization.

4.3 – Recursion run amok

* Slæma við bad fibonacci er að það kallar tvisvar sinnum í hvert skipti í sjálft sig, þess vegna 2^n
* Ef recursion kallar í recursion og nær aldrei base case þá verður þetta endalaust.
* Alltaf að passa að recursion call sé að fara í átt að base case, range að minnka eða eh svoleiðis.
* Hvert recursive action býr til activation record sem er stored í minni
* Python er með limit á recursive depth sem er 1000 nested functions, hægt að breyta svona: 

4.4 Further examples of recursion

* Linear recursion – ef recursive call starts at most one other, kallar s.s. bara einu sinni í sjálft sig í hvert skipti, s.s. n-1 sinni yfir allt
  + Þetta segir ekki til um tíma heldur frekar um recursion traceið
* Binary recursion – ef recursive call may start two others
  + Kallar í sjálfan sig 2\*n-1 sinni.
* Muliple recursion – ef recursive call may start three or more others
  + 3\*n-1 eða stærra en 3

4.5 – Designing Recursive Algorithms

* Skref til að hanna:
  + Testa base cases, passa að allar mögulegar chains of recursive calls munu enda á base case
  + Recur, test sem ákveður hvaða recursive call á að framkvæma, passa að hvert recursive call sé þannig að það leiði á endanum í base case

4.6 – Eliminating tail recursion

* Tail recursion er ef recursive call sem er kallað í í einu tilviki er síðasta aðgerðin í því tilviki og skilar valueinu úr þessu recursive calli beint í recursionið sem er að enda. Þarf að vera linear recursion, ekki hægt að gera annað recursive call ef það verður að skila strax niðurstöðu þess fyrsta.
  + = A recursion is a tail recursion if any recursive call that is made from one context is the very last operation in that context, with the return value of the recursive call (if any) immediately returned by the enclosing recursion.
* Oft hægt að breyta recursive calls í for eða while loopur.

Myndband

* 3 hlutir sem þarf að hafa í huga þegar búið til recursive föll
  + Kalla í sjálft sig
  + Þurfa að fara nær base case þegar þau kalla í sjálft sig
  + Vera með base case
* Í stack er hvert skref geymt, með öll n sem koma fyrir áður en það nær base case.
* Hlutir sem maður gerir fyrir ofan endurkvæmnina gerist þegar maður fer niður í endurkvæmnina en það sem er fyrir neðan gerist þegar maður ferð til baka upp aftur.

**5.kafli – las bara** 5.2.1 (and beginning of 5.2), 5.3 and 5.4.1

5-1

* List, tuple og str – hægt að nota index, nota array til að representa röðina

5-2

* Byte er 8 bits
* Tölvan notar memory address til að vita hvaða upplýsingar eru skráðar í hvaða byte. Hvert byte er tengt við unique númer sem er addressa þessa bytes.
* Any byte af main memory er hægt að sækja með memory addressinu hans.
* Main memoryið er random access memoru (RAM), þ.e. jafn auðvelt að sækja byte 242534 og 12. Hægt að senda eða sækja byte á O(1) tíma.
* Hver stafur (unicode) í python eru 2 bytes.
* Hver sella í array þarf að nota sama fjölda bytes.
* start + cellsize page208image6448index – hægt að nota ef maður veit hvar arrayið sem maður er að skoða byrjar í memory address, sjá dæmi á bls 208 í tölvubók

5-2-1

* Listar og tuplur sem geta innihaldið mislanga strengi t.d. nota innri minni mechanisma af gerð references. Hvert element í listanum/túplunni hefur eitt memory address og vísar (references) svo í strenginn.
* Ef listi/túpla geymir immutable objects hefur það engin áhrif ef tveir listar deila elementi.
* Reference ef það eru mutable objects í listum/túplum, hefur áhrif ef þeim er breytt.

Sleppi rest af 52 í bili

5-3

* Ekki hægt að stækka arrays með því að fara í næstu memory sellur, því þar gæti verið annað data. Ekki vandamál í túplum eða strengjum því þær eru immutable.
* Hægt að bæta við elementum í lista, notað dynamix array til þess. Kerfi bætir oft við auka plássi fyrir aftan array in case, t.d. ef maður býr til 5 staka lista þá býr kerfið til 8 staka í minninu. Ef auka plássið er búið færir (býr til nýtt og eyðir gamla) arrayið sig annað þar sem er meira pláss.
* Hvernig á að fatta hversu stórt array á að búa til? Algeng regla er að búa til array sem hefur 2x stærð þess arrays sem var verið að fylla
* getsizeof úr sys module segir manni hversu mikil bytes er verið að nota til að geyma object í python
* Í tilrauninni þeirra: array getur geymt 4 references
* Þar sem að listar referencea í elementin sín þá skoðar getsizeof fallið bara stærðina fyrir grunn byggingu, s.s. ekki minnið sem er notað af elementum listans.
* Vegna keyrslutíma er sniðug að núllstilla array í upphafi og bæta inní það
* Amortized analysis – lýst með pening sem er fyrir fastan keyrslutíma. Borgar með honum fyrir operation running time. Hægt að overchargea sumar aðgerðir til að spara pening annars staðar.
* Keyrslutími fyrir að keyra frá tómum array í array með n stökum er O(n) – proposition 5.1
* Verður að passa að taka ekki of mikið pláss frá fyrir tóm stök í array ef userinn skyldi vilja bæta við
* Performing a series of *n* append operations on an initially empty dynamic array using a fixed increment with each resize takes (*n*2) time.
* Þarf að passa þegar element eru removeuð að arrayið sé ekki alltof stórt miðað við fjölda elementa. T.d. ef fjöldi actual elementa eru undir ¼ þá er hægt að eyða hluta arraysins.

5-4

5-4-1

Immutating:

A screenshot of a cell phone

Description automatically generated

Mutating:

A screenshot of a cell phone

Description automatically generated

Myndband

* Ýmsar aðgerðir eru nú þegar til í python, ætlum að prófa að útfæra þær sjálf.
* Getum gert tvær aðgerðir á listum
* Arr = [0] \* size
* Arr[i] eitthvað
* Setja fram capacity breytu sem heldur utan um hversu mikið magn af plássum eru en size sem heldur utan um hversu mörg pláss er verið að nota
* Tvöföldum fylkið sem fylltist, þ.e. tvöföldum capacity
* Til að búa til nýtt fylki, síðasta skipunin er til að setja rétta vísun á nýja fylkið
* for i in range(size):
* new\_arr[i] = arr[i]
* arr = new\_arr