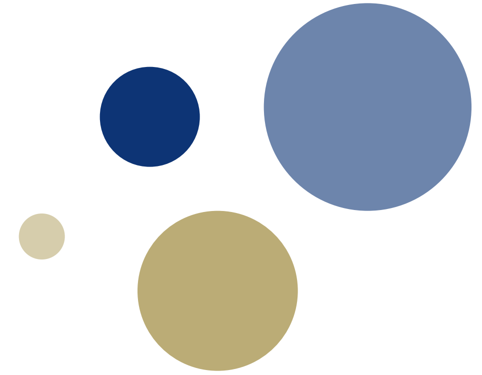
NTNU
Kunnskap for en bedre verden



Python: Rekursjon
(og programmering av algoritmer)

Python-bok: Kapittel 12 + teoribok om Algoritmer

TDT4110 IT Grunnkurs

Professor Guttorm Sindre

Læringsmål og pensum

- Mål
  - Forstå, og kunne bruke, algoritmer for søk og sortering
  - Forstå, og kunne bruke, rekursjon
- Pensum
  - Theory Book IT Grunnkurs: Algorithms
  - Starting out with Python:
    - 3rd edition: Chapter 12 Recursion

Algoritmeeffektivitet

- Måles som antall utførte instruksjoner, f.eks.
  - Antall sammenlikninger (if-setninger)
  - Antall basis aritmetiske operasjoner (+ , *, …)
  - Antall ombyttinger mellom plasser i ei liste etc.
- Viktigst for store datamengder
- ”BigTheta”-notasjon klassifiserer algoritmer
  - Hvordan utvikler kjøretid seg i forhold til datamengden?
    - Sekvensielt søk tilhører $\Theta(n)$, dvs. mengden av algoritmer hvor tidsforbruk utvikler seg lineært med økende datamengde
      - For sorterte data, bedre med binærsøk som er $\Theta(\log n)$
      - Med hashing kan søk være $\Theta(1)$
        - » men da med økt bruk av lagerplass, jfr. Mengder og Dictionary
    - Innstikksortering (insertion sort) tilhører $\Theta(n^2)$
      - Bedre: quicksort, mergesort: $\Theta(n \log n)$

Eksempel

- Hvis **n** er antall oppslag og **m** antall element i data
  - Dvs. m er hvor mange tall det er i lista / tuplet / mengda
- Hva er average case tidskompleksitet til n_oppslag( ) ?
  - (a) Θ(n)   (b) Θ(n·m)   c) Θ(m)   d) avh. av type data

```
def n_oppslag(n, data):
    # Gjør n oppslag i data'ene, som kan være liste, tuppel, mengde...
    # Returnerer tiden som det tar fra funksjonens start til slutt
    start = time() # klokka nå, dvs. starttidspunkt for funksjonen
    big_value = len(data)*2
    for i in range(n):
        if randint(0, big_value) in data:
        # for å oppnå at ca. halvparten av de tilfeldige tallene er der
            a = 1
        else:
            a = 0
    return time() - start # diff mellom klokka nå og starttidspunkt
```

Kode: tidstest.py

Funksjoners vekst

n	n^2	n^3	2^n	log(n)	nlog(n)
1	1	1	2	0	0
2	4	8	4	1	2
3	9	27	8	2	5
4	16	64	16	2	8
5	25	125	32	2	12
6	36	216	64	3	16
7	49	343	128	3	20
8	64	512	256	3	24
9	81	729	512	3	29
10	100	1000	1024	3	33
100	10000	1000000	1,2677E+30	7	664
1000	1000000	1000000000	1,072E+301	10	9966
10000	100000000	1000000000000	#NUM!	13	132877
100000	10000000000	1000000000000000	#NUM!	17	1660964
1000000	1000000000000	1000000000000000000	#NUM!	20	19931569
10000000	100000000000000	1000000000000000000000	#NUM!	23	232534967
100000000	10000000000000000	1E+24	#NUM!	27	2657542476
1000000000	1000000000000000000	1E+27	#NUM!	30	29897352854
10000000000	100000000000000000000	1E+30	#NUM!	33	332192809489
100000000000	1E+22	1E+33	#NUM!	37	3654120904376
1000000000000	1E+24	1E+36	#NUM!	40	39863137138648
10000000000000	1E+26	1E+39	#NUM!	43	431850652335357
100000000000000	1E+28	1E+42	#NUM!	47	4650699332842310
1000000000000000	1E+30	1E+45	#NUM!	50	49828921423310400
10000000000000000	1E+32	1E+48	#NUM!	53	531508495181978000
100000000000000000	1E+34	1E+51	#NUM!	56	5647277761308520000
1000000000000000000	1E+36	1E+54	#NUM!	60	59794705707972500000
10000000000000000000	1E+38	1E+57	#NUM!	63	631166338028599000000
100000000000000000000	1E+40	1E+60	#NUM!	66	6643856189774730000000
1000000000000000000000	1E+42	1E+63	#NUM!	70	69760489992634600000000
10000000000000000000000	1E+44	1E+66	#NUM!	73	730824180875220000000000
100000000000000000000000	1E+46	1E+69	#NUM!	76	7640434618240930000000000
1000000000000000000000000	1E+48	1E+72	#NUM!	80	79726274277296700000000000
10000000000000000000000000	1E+50	1E+75	#NUM!	83	830482023721841000000000000
100000000000000000000000000	1E+52	1E+78	#NUM!	86	8637013046707140000000000000
1000000000000000000000000000	1E+54	1E+81	#NUM!	90	89692058561958800000000000000
10000000000000000000000000000	1E+56	1E+84	#NUM!	93	930139866568462000000000000000
100000000000000000000000000000	1E+58	1E+87	#NUM!	96	9633591475173350000000000000000

Hva forårsaker kompleksitet?

- Store datamengder
- Operasjoner som må repeteres mange ganger
  - Løkker
  - Rekursjon
  - Standardfunksjoner / -metoder
    - …som ser ut som enkle setninger
    - Men ”under panseret” inneholder løkker
    - Se f.eks. https://wiki.python.org/moin/TimeComplexity
- Nødvendig kompleksitet
  - Problemet er så komplekst, fins ingen smartere løsning
- Unødig kompleksitet
  - Dårlig valg av algoritme eller datastruktur
  - Dårlig valg av standardfunksjon / - metode

Innstikksortering (insertion sort)

Theory Book IT Grunnkurs

Algorithms

Kapittel 5.4

Innstikksortering
(eng: insertion sort)

- Enkel og intuitiv sorteringsalgoritme
  – sorterer ei liste, element for element
- Fordeler:
  – Intuitiv: Minner om menneskers manuelle sortering
  – Ganske enkel å programmere
  – Effektiv for korte lister
  – Sorterer lister ”in place”
- Ulemper
  – Ikke den mest effektive

Oppgave: Insertion sort

- Lag funksjonen insertion_sort som tar inn ei liste og returnerer ei sortert liste ved hjelp av pseudokoden vist under.
- Pseudokoden antar at første element i lista har indeks 0.

```
def insertion_sort(liste)
    La i gå fra 0 til indeks for siste element i lista
        element = liste[i]
        hull = i
        Så lenge hull > 0 og liste[hull – 1] > element
            liste[hull] = liste[hull – 1]
            hull = hull – 1
        liste[hull] = element
    return liste
```

VANSKELIGERE:
(a) Legg til param. og kode så man kan velge mellom stigende og synkende sortering.
(b) Legg til param. og kode så man kan velge om opprinnelig liste skal beholdes som den er (returnere sortert kopi, men ikke endre lista i funk.-argumentet)

Kode: insertion_sort_V0.py

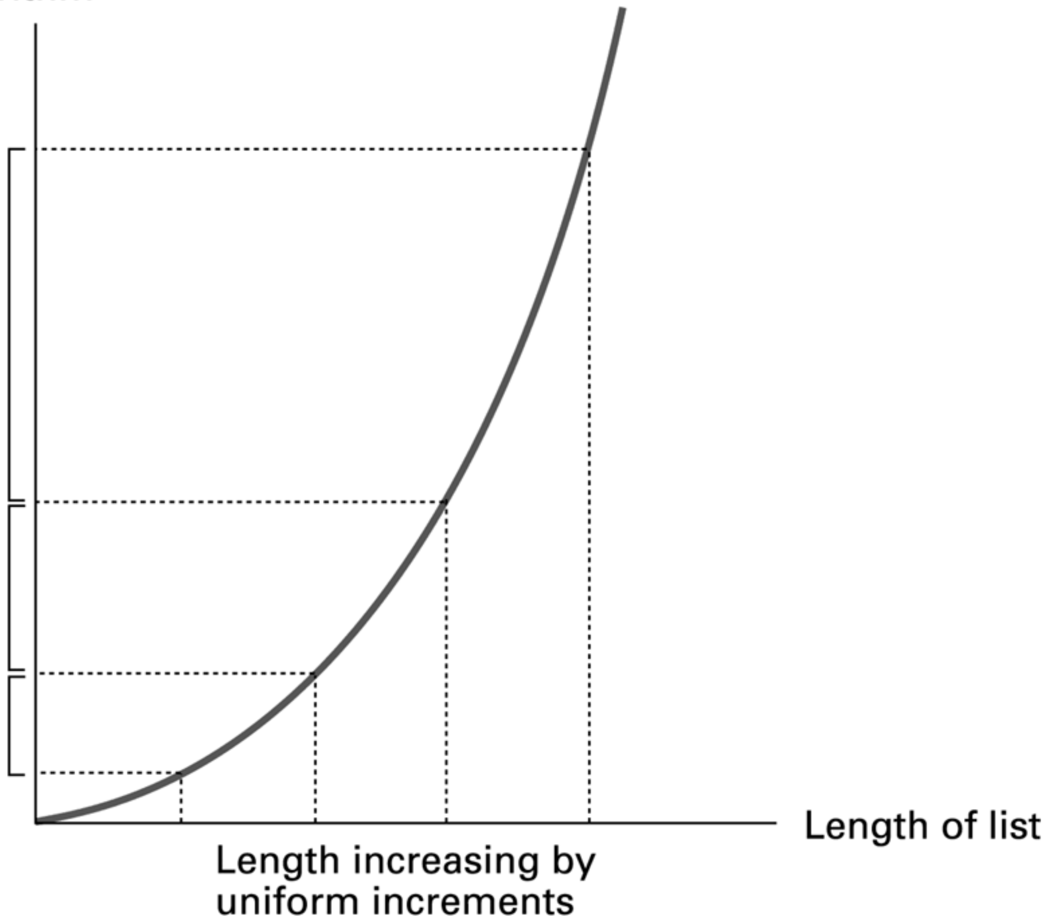
Løsning: insertion_sort_V1.py

Løsning: insertion_sort_V2.py

Karakteristikker av innstikkssorteringsalgoritmen

- Hva ytelsen på denne algoritmen i best-case?
  - Nesten sortert liste, kun ett element må flyttes
  - n sammenlikninger og 1 ombytting, dvs. $\Theta(n)$ som best case
- Hva er ytelsen på denne algoritmen i worst-case?
  - Worst-case er at alle elementene må byttes om.
  - (n-1)/2 sammenlikninger, n-1 ombyttinger, dvs. $\Theta(n^2)$
- Evt. Kopiering (jfr. ”vanskeligere” på forrige side)
  - Kopiere n elementer i tillegg
  - Gjøres før løkka, endrer ikke $\Theta(n)$ og $\Theta(n^2)$ som best og worst
- Hva er en god egenskap med algoritmen?
  - Krever lite ekstra plass i minnet:
    - Lista med elementer
    - En variabel for å ta vare på den som skal byttes.

Graf over worst-case analysen av *innstikksortering (insert sort)*



Rekursjon

Starting out with Python:

Chapter 12 Recursion

Rekursjon

- Noe inneholder eller defineres ved hjelp av seg selv
  - Rekursive funksjoner: kaller seg selv
  - Rekursive datastrukturer (del-elementer av samme type)
- Rekursive funksjoner brukes ofte i algoritmer
  - Kan gi korte og elegante løsninger på problemer
    - Basistilfelle: problemet har triviell løsning, ingen rekursjon
    - Generelt tilfelle: rekursivt kall
- Argumentet til funksjonen må gradvis endres
  - Så vi til slutt ender opp i basistilfellet
  - Ellers får vi evig løkke

Rekursjon, Eks.1: fakultet

NB: rekursjon gir ikke noen spesiell fordel her, minst like greit med løkke, men det er et enkelt eksempel for å introdusere konseptet rekursjon

- Fakultet til et tall beregnes på følgende måte:
  - Hvis n = 0 så er n! = 1
  - Hvis n > 0 så er n! = 1 · 2 · 3 · … · n
- Fremgangsmåte for problemløsning:
  1. Finne basistilfellet (hvor rekursivt kall unngås):
     - **Hvis** n == 0 **så** n! = 1 (ikke avhengig av tidligere ledd)
  2. Uttrykke det generelle tilfellet (med rekursjon):
     - **Hvis** n > 0 **så** fakultet(n) = n · fakultet(n-1)
     - NB: de rekursive kallene må bevege seg gradvis nærmere basistilfellet, her skjer dette ved at n blir 1 mindre per runde, vil dermed før eller senere bli 0

fakultet.py

Rekursjon, Eks.2: GCD

(Greatest Common Divisor, dvs. største felles faktor)

- Finn største felles faktor for to heltall m >= n > 0
  - Største heltall **d** som både **m** og **n** er delelig på
  - Dvs., m % d == 0 og n % d == 0
- Naiv algoritme: løkke fra n og nedover (n–1, n–2,…)
- Kjente rekursive algoritmer
  - Euklids algoritme (200 f.Kr.)
    - Basistilfelle: m % n == 0 (tallene går opp direkte)
    - Generelt tilfelle: Utnytter at gcd(m, n) == gcd(n, m%n)
  - Dijkstras algoritme (1970)
    - Basistilfelle: m == n
    - Generelt tilfelle: Utnytter at gcd(m, n) == gcd(m–n, n)
      - Kan være raskere enn Euklid (subtraksjon kjappere enn modulo)
      - Men noen ganger tregere (trenger flere rekursive kall)

gcd.py

Binærsøk
(Binary Search Algorithm)

Theory Book IT Grunnkurs

Kapittel: Algorithm, 5.5

Pseudokode for binærsøk

```
bin_search(liste, verdi, min, max)
        if max < min
                Returner ikke funnet verdi
        else
                midtpunkt = midtpunkt(min, max)
                if verdi < liste[midtpunkt]
                         returner bin_search(liste,verdi,min,midtpunkt-1)
                elif verdi > liste[midtpunkt]
                        returner bin_search(liste,verdi,midtpunkt+1,max)
                else
                        returner midtpunkt
```

bin_soek.py

Karakteristikker av binærsøk

- Hva er best-case?
  - Treffer på første oppslag, $\Theta(1)$
- Hva er worst-case?
  - Halverer lista helt til det ikke er noe igjen, $\Theta(\log n)$
- Kan sekvensielt søk være raskere enn binærsøk?
  - Ja hvis sekvensielt søk treffer tidlig i lista
  - Men binærsøk er bedre både på average case og worst case
- Begrensninger på binærsøk:
  - Lista må være sortert
  - Sortering koster $\Theta(n \log n)$, altså mer enn sekvensielt søk
    - Men hvis vi trenger å søke ofte (f.eks flere mill. ganger per dag) men sortere sjelden (for eksempel bare en gang hver natt), kan det likevel lønne seg å sortere for å så kunne benytte binærsøk
    - NB: Alternative datastrukturer med hashing (mengde, dictionary) som gir direkte oppslag bør evt. også vurderes

Binærsøk, hva med å bruke slice?

- I eksempelkoden bin_soek.py fikk hadde søkefunksjonen tre parametre
  – liste, lav indeks og høy indeks
- Med slicing kan vi klare oss med kun én parameter
  – Gjenværende del av lista det skal søkes i
- Hva blir effekten av dette?
  – A) slicing vil være vesentlig raskere (mer enn 50% besparelse)
  – B) omtrent like raskt (mindre enn 10% forskjell)
  – C) slicing vil være vesentlig tregere (mer enn 50% lenger tid)

bin_soek_sammenlign.py

Graf over worst-case analyse av binært søk

Time required to execute the algorithm

Time increasing by decreasing increments

Length of list

Length increasing by uniform increments

Kodeforståelse m. rekursjon

Eksamen desember 2015…

Oppgave 2b (5%)

Hva blir skrevet ut til skjerm når du kjører koden som vist under? (3 %)

Forklar med en setning hva funksjonen **compute** gjør (2 %)

```
def compute(x):
  if x<10:
    return x*compute(x*2)
  else:
    return 1

print(compute(1))
```

Oppsummering

- Søk:
  - Sekvensielt søk, average og worst case O(n)
    - Enkel å implementere, fungerer på alle lister
  - Binærsøk: average og worst case O(log n)
    - Kun på sorterte lister
  - Søk i hash-liste (f.eks. mengde, dictionary)
    - Average case: $\Theta(1)$
      - Avhengig av robust hash-funksjon
      - Bruker ekstra minne
    - Worst case er $\Theta(n)$ – hvis det blir masse kollisjoner
- Sortering:
  - Innstikksortering: Enkel og intuitiv
    - Best case ved nesten sortert liste: $\Theta(n)$
    - Average og Worst case: $\Theta(n^2)$
  - Det fins raskere algoritmer som er $\Theta(n \log n)$