NTNU

Kunnskap for en bedre verden

TDT4110 Informasjonsteknologi grunnkurs:
Tema: Algoritmer i praksis

Terje Rydland - IDI/NTNU

Datastruktur: *Dictionaries* Kap 9.1

- Dictionary er et objekt som lagrer en samling av data.
- Minner litt om lister men har klare forskjeller:
  - Defineres ved å bruke krøllparanteser { } (ALT+SHIFT 8/9 på Mac)
  - Kan bruke **hva som helst som nøkkel** (indeks) (ikke bare tall som i lister):
    - Tekststrenger, Heltall (men trenger ikke å være i rekkefølge), Flyttall, Sannhetsverdier (True eller False), En kombinasjon av de ovenfor

```
A = {}     # Tom dictionary
A['Kari'] = 92925492 # Oppretter et element
tlf={'Jo':73540000,'Per':92542312,'Else':54239212}
print(tlf['Per']) # Skriver ut verdien 92542312
```

- {nøkkel$_1$: verdi, nøkkel$_2$: verdi,…,nøkkel$_n$: verdi}

Bruk av operatorene in og not in i Dictionaries

- Man kan bruke **`in`** og **`not in`** i dictionaries for å sjekke om elementer finnes (nøkler!):

```
tlf={'Jo':73540000,'Per':92542312,'Else':54239212}
if ('Per' in tlf):
    print(tlf['Per'])
if ('Lars' not in tlf):
    print('Lars er ikke i dictionarien')
```


Operasjoner for *dictionaries*

Operasjon	Forklaring
len(d)	Antall elementer i d
d[k]	Verdi til element i d med nøkkel k
d[k] = v	Sett element k til verdi v
del d[k]	Slett element k i d
d.clear()	Fjern alle elementer i d
d.copy()	Lag kopi av d
~~d.has_key(k)~~	Fjernet i Python 3. Må bruke in

Operasjon	Forklaring
d.items()	Returnerer liste av (nøkkel,verdi) par
d.keys()	Returnerer liste av nøkler i d
d.values()	Returnerer liste av verdier i d
d.get(k)	Samme som d[k]
d.get(k,v)	Returnerer d[k] hvis k er gyldig, ellers v

`dictionary_metoder.py`

for-løkke og *dictionary*

- Bruker man for-løkke på en ***dictionary***, løper man igjennom nøklene:

```
d = {'navn':'Per','alder':28,'IQ':29}
for x in d: # går igjennom nøkler i d
    print(x) #  skriver ut nøklene navn, alder, IQ
```

- For å sikre gjennomgang av nøkler alfabetisk: **for_dictionary.py**

```
for x in sorted(d.keys()): # sorterer nøkler
    print(x)
```

for_dictionary.py - /Users/terjery/Library/Mobile Documents/com~apple~CloudDocs/ITGK 2016/Python/Keynote/Uk...

```
d ={'navn':'Ole','alder':42,'weight':45} #Oppretter dict

print(d)

print('\nKeys non-sorted:\n')
for x in d:
    print(x, d[x])

print('\nKeys sorted:\n')
for x in sorted(d.keys()): # sorterer keys
    print(x, d[x])
```

Ln: 1 Col: 0

Serialisering av objekter Kap 9.3

- Serialisering av objekter er prosessen å **konvertere et objekt til en strøm av bytes** som kan lagres til fil, som senere kan lastes inn igjen.
  - F.eks. for lagring av **dictionary** eller **set**

- Python har biblioteket pickle som gjør det mulig å lagre og laste dictionary til/fra disk.

Lagre dictionary til disk

- For å lagre en dictionary eller en mengde til disk kan biblioteket pickle brukes sammen med metoden dump.
- Må åpne en fil som binærfil og ikke som tekst, ettersom det er mer enn tekst som lagres (struktur):

```
db={'Jo':[10,'Skogata 3'],'Per':[20,'Height 2']}
import pickle                    # Importerer modulen
f = open('database.dat','wb') # b=binary
pickle.dump(db,f)                # Dumper db til disk
f.close()                        # Stenger fila
```

lagre_dictionary.py

lagre_dictionary.py

pickle.py - /Users/terjery/Library/Mobile Documents/com~apple~CloudDocs/ITGK 2016/Python/Eksempler på forel...

```
import pickle   # Importerer pickle biblioteket

print('Skriv inn navn, telefon og adresse. \
Avslutt med ENTER når programmet ber om navn.')

navn = input('Oppgi navn: ')

tlf = {}    # Oppretter en tom dictionary

while navn != '':
    telefon = input('Telefonnr: ')
    adresse = input('Adresse:   ')
    tlf[navn] = [telefon,adresse]
    navn = input('Oppgi navn: ')

filnavn = input('Oppgi navn på fila du vil lagre data til: ')

f = open(filnavn,'wb')  # Åpner fila for binær skriving
pickle.dump(tlf,f)      # Konverterer til binært og
                        # dumper dictionary tlf til disk
f.close()               # Stenger fila
```

Ln: 1 Col: 23

lagre_dictionary.py

pickle 2.py - /Users/terjery/Library/Mobile Documents/com~apple~CloudDocs/ITGK 2016/Python/Eksempler på forelesning/pic...

```
import pickle   # Importerer pickle biblioteket

print('Skriv inn navn, telefon og adresse. \
Avslutt med ENTER når programmet ber om navn.')

tlf = {}    # Oppretter en tom dictionary

while True:
    navn = input('Oppgi navn: ')
    if navn == '':
        break           # Avslutt while-løkka og hopp ut
    telefon = input('Telefonnr: ')
    adresse = input('Adresse:   ')
    tlf[navn] = [telefon,adresse]

filnavn = input('Oppgi navn på fila du vil lagre data til: ')

f = open(filnavn,'wb')  # Åpner fila for binær skriving
pickle.dump(tlf,f)      # Konverterer til binært og
                        # dumper dictionary tlf til disk
f.close()               # Stenger fila
```

Ln: 3 Col: 0

Laste inn dictionary fra disk

- Laste inn dictionary fra disk gjøres ved hjelp av load metoden i pickle-biblioteket.
- Husk at åpne fila som binærfil og ikke tekst.

```
import pickle
f = open('datafil.dat','rb')        # r=read, b=binary
data = pickle.load(f)               # Laster inn dict fra disk
f.close()
```

*laste_dictionary.py - /Users/terjery/Library/Mobile Documents/com~apple~CloudDocs/ITGK 2016/Python/Keynote/Uke 44...

```
import pickle # Importerer pickle bibliotek

filnavn = input("Skriv inn navn på datafil: ")

f = open(filnavn,"rb") # Åpner for lesing og binærfil
database = pickle.load(f) # Laste inn dictionary fra disk
f.close() # Stenger fila

for item in database:
    print(item,":",database[item])
```

laste_dictionary.py

Ln: 12 Col: 0

Oppsummering

- Sets er nyttige for å gjøre operasjoner på mengder:
  - intersection og union
  - Sets defineres ved å bruke funksjonen `set(liste)`
- Dictionary ligner på lister men man kan bruke hva som helst som nøkkel (indeks).
  - Opprettes ved å bruke `A = {}` eller `A = {'navn':'Petter Ole'}`
  - En fordel med dictionary er at man ikke trenger en indeks som er i rekkefølge.
  - Det finnes flere kommandoer som man kan bruke på dictionary.
  - Dictionary oppfører seg som en database.

Læringsmål og pensum

- Mål
  - Lære å forstå og kunne programmere algoritmer for søk og sortering.
  - Lære å forstå og kunne bruke rekursjon.
- Pensum
  - Theory Book IT Grunnkurs: Algorithms, s.125-178
  - Starting out with Pyton:
    - 3rd edition: Chapter 12 Recursion

Algoritmeeffektivitet

- Måles som antall utførte instruksjoner
  - For eksempel antall sammenlikninger (if-setninger), antall ombytninger mellom plasser i ei liste etc.
- Ser på hva som skjer for svært mange inputs

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	10000000000000000000	#NUM!	23	232534967
100000000	10000000000000000	1E+24	#NUM!	27	2657542476
1000000000	100000000000000000	1E+27	#NUM!	30	29897352854
10000000000	1000000000000000000	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	664385618977473000000
1000000000000000000000	1E+42	1E+63	#NUM!	70	697604899926346000000
10000000000000000000000	1E+44	1E+66	#NUM!	73	730824180875220000000
100000000000000000000000	1E+46	1E+69	#NUM!	76	764043461824093000000000
1000000000000000000000000	1E+48	1E+72	#NUM!	80	797262742772967000000000
10000000000000000000000000	1E+50	1E+75	#NUM!	83	830482023721841000000000
100000000000000000000000000	1E+52	1E+78	#NUM!	86	863701304670714000000000
1000000000000000000000000000	1E+54	1E+81	#NUM!	90	896920585619588000000000000
10000000000000000000000000000	1E+56	1E+84	#NUM!	93	930139866568462000000000000
100000000000000000000000000000	1E+58	1E+87	#NUM!	96	963359147517335000000000000

Sekvensielle søk

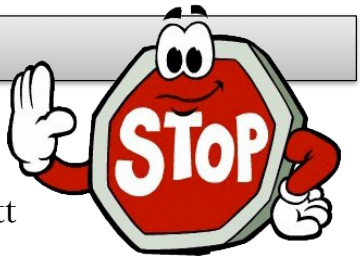
- Sekvensielt søk
  - algoritmer som søker igjennom en liste for en bestemt verdi ved å sjekke hvert eneste element i rekkefølge inntil verdien er funnet.
- Sekvensielt søk
  - den enkleste søkealgoritmen som bruker en **brute force** tilnærming.
- Fordel med sekvensiell søk er at den fungerer på en usortert liste.

Oppgave: Sekvensielt søk

- Skriv Python-koden for å gjøre et sekvensielt søk:
  - Lag en funksjon som tar inn ei liste og en variabel **item** som spesifiserer hva det søkes etter
  - Gå igjennom lista og undersøk om hvert element i lista er lik variabelen **item**.
  - Hvis man finner et element i lista som har samme verdi som **item**, så skal funksjonen returnere True
  - Hvis ikke skal den returnere False

`seq_search.py`

Oppgave: Sekvensielt søk



2 funksjoner som gjør det samme, men den ene er litt mer effektiv enn den andre…

seq_search.py - /Users/terjery/Library/Mobile Documents/com~apple~CloudDocs/ITGK 2016/Python/PP/uke45/Python/seq_search.py (3.5.1)

```
def seq_search(liste,item):
    funnet = False # Variabel som sier om verdien er funnet eller ikke
    for x in liste:
        if (x==item):
            funnet=True
    return funnet

def seq_search2(liste,item):
    for x in liste:
        if(x==item):
            return True
    return False


A=[1,9,3,4,5,6,7,8,9,10]
print(seq_search2(A,3))
```

Ln: 1 Col: 0

Om sekvensielt søk

- Hvordan kan vi gjøre algoritmen mer effektiv?
  - Stoppe når vi finner det vi leter etter.
- Hva er best-case scenario for et sekvensielt søk?
  - Finner første element: 1 sammenligning
- Hva er worst-case scenario for et sekvensielt søk?
  - Finner siste element: n sammenligninger
- Hva er tidsbruken worst-case for sekvensielt søk?
  - Tidsbruk proporsjonal med antall elementer i lista: O(n)

Innstikksortering (insertion sort)

- Theory Book IT Grunnkurs
- Algorithms
- Kapittel 5.4

Innstikksortering

- Enkel sorteringsalgoritme som sorterer ei liste, element for element
- Ganske enkel å programmere
- Effektiv for korte lister
- Kan sortere alle lister som de er
- Algoritmen minner mye om hvordan mennesker tenker, som f.eks. sortering av en kortstokk

Illustrasjon av innstikksortering

3	0	1	8	7	2	5	4	9	6

folkedansdemo

Oppgave: Insertion sort

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

```
La i gå fra 1 til lengden av lista – 1:
   La element få verdien liste[i]
   La hull få verdien i
   Så lenge hull>0 og liste[hull-1] > element
        La liste[hull] få verdien liste[hull-1]
        La hull få verdien hull – 1
   liste[hull] = element
```

insertion_sort.py

Oppgave: Insertion sort

insertion_sort.py - /Users/terjery/Library/Mobile Documents/com~apple~CloudDocs/ITGK 2016/Python/Keynote/Uke 45/Pyt...

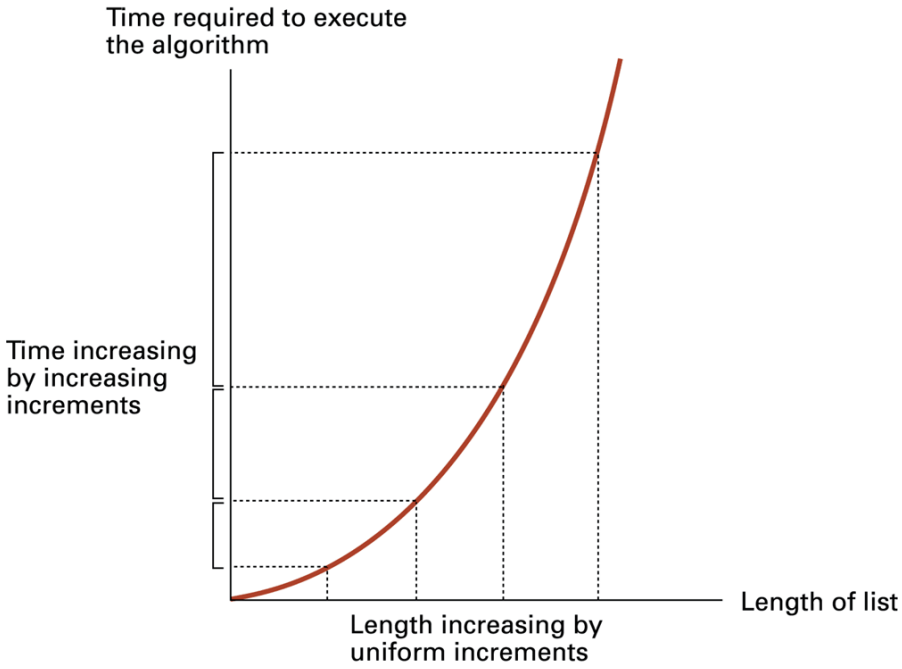
```
# Name: insertion_sort
# Input: a list
# Output: a sorted liste
# Description: sort a list using the insertion sort algorithm

def insertion_sort(liste):
    L=len(liste) # L er lengden av lista
    for i in range(1,L): # range gaar til L-1
        element = liste[i]
        hull = i
        while(hull>0 and liste[hull-1]>element):
            liste[hull] = liste[hull-1]
            hull = hull - 1
        liste[hull] = element
    return liste

A=['Petter','Alex','Diana','Bodil','Anne']
B=insertion_sort(A)
print(B)
```

Ln: 1 Col: 0

Karakteristikker av innstikkssorteringsalgoritmen

- Hva ytelsen på denne algoritmen i best-case?
  - Best-case er en nesten sortert liste der bare ett element må flyttes en plass.
  - Fører til $\Omega(n)$ sammenlikninger og $\Omega(1)$ ombyttinger
- Hva er ytelsen på denne algoritmen i worst-case?
  - Worst-case er at alle elementene må byttes om.
  - Fører til $O(n^2)$ sammenlikninger og ombytninger.
- 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 ut.

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



Illustrasjon av binærsøk

Finnes 17 i listen?

indeks	0	1	2	3	4	5	6	7	8	9
verdi						13	17	25		

Sjekk det midterste elementet. (0 + 9)//2 -> 4, altså a(4)

17 > a(4)

Sjekk da det midterste elementet mellom a(5) og a(9) (5 + 9)//2 -> 7

17 < a(7)

Sjekk da det midterste elementet mellom a(5) og a(7) (5 + 7)//2 -> 6

17 == a(6)

17 finnes i listen

Algoritme for binærsøk

```
bin_search(liste, verdi, min, max)
   så lenge max > min:
      midtpunkt = midtpunkt(min,max)
      hvis verdi = liste[midtpunkt]
         returner True
      hvis verdi > liste[midtpunkt]
         sett min til midtpunkt+1
      hvis ikke
         sett max til midtpunkt-1
   returner False
```

binary_search.py

Binærsøk kode

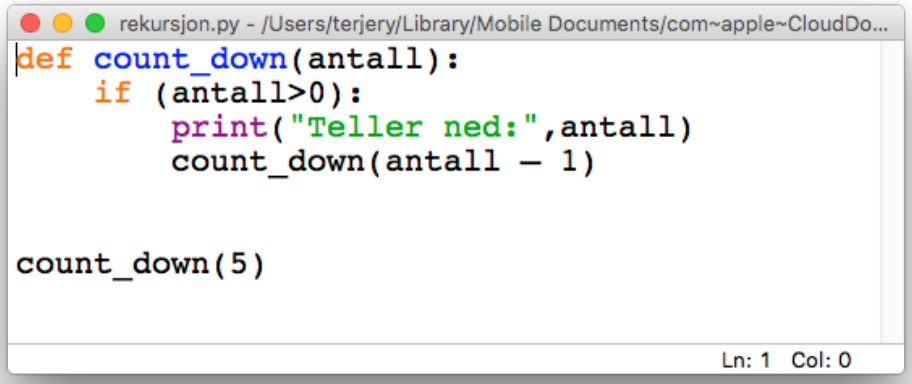
binary_search-it.py - /Users/terjery/Library/Mobile Documents/com~apple~CloudDocs/ITGK 2016/Python/1Forelesnin...

```
# Name: binary_search
# Input: list, search value, min index and max index
# Output: Index of found value or False
# Description: Search through sorted list using binary alogrithm

def binary_search(liste, verdi, imin, imax):
    while imin < imax:
        imid = (imin+imax)//2 # Heltallsdivisjon
        if verdi==liste[imid]:
            return True
        elif verdi > liste[imid]:
            imin = imid+1
        else:
            imax = imid-1
    return False

A=[1,2,3,9,11,13,17,25,57,90]
print(binary_search(A,57,0,len(A)-1))
```

Ln: 13 Col: 0

Rekursjon

- Rekursiv funksjon kalles det når en funksjon kaller seg selv.
- Rekursive funksjoner brukes ofte i algoritmer for å lage elegante løsninger på problemer.
- For at en rekursiv funksjon ikke skal fortsette i det uendelige, er det viktig at argumentet til funksjonen endres slik at den en gang får en stopp verdi.



rekursjon.py

Rekursjon til å kalkulere fakultet

- Fakultet til et tall beregnes på følgende måte:
  - Hvis n=0 så er n!=1
  - Hvis n>0 så er n!=1 x 2 x 3 x … x n
- For å lage en algoritme for fakultet må:
  - Først finne delen som ikke er rekursiv (som ikke skal kalles på nytt):
    - Hvis n = 0 så er n!=1 (denne er ikke avhengig av tidligere ledd)
  - Resten er den rekursive delen som stegvis kan beskrives:
    - Hvis n>0 så er fakultet(n) = n x fakultet(n-1)
  - Dette er alt vi trenger for å lage en rekursiv funksjon for å beregne fakultet

fakultet.py

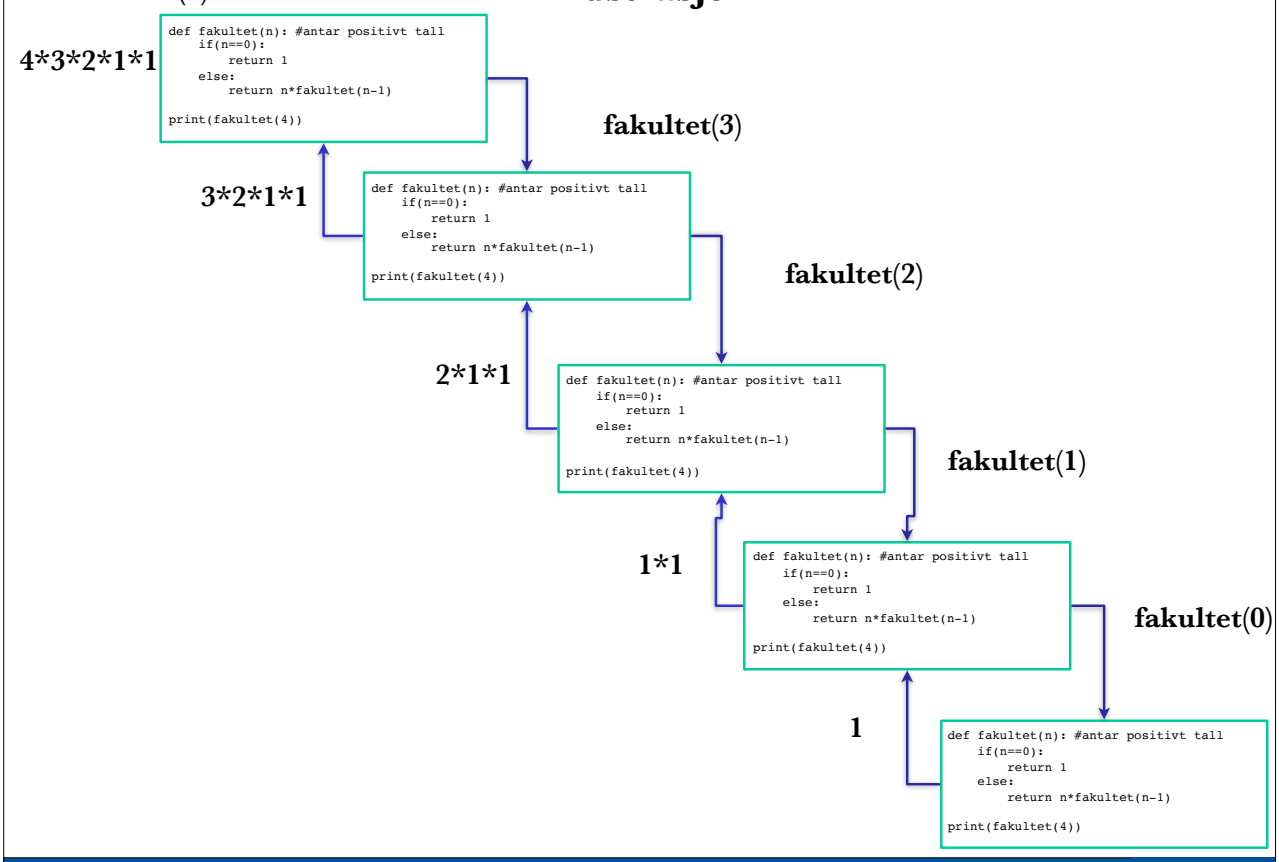
fakultet.py

fakultet.py - /Users/terjery/Library/Mobile Documents/com~apple~C...

```
def fakultet(n):
    if(n==0):
        return 1
    else:
        return n * fakultet(n-1)

print(fakultet(4))
```

Ln: 1 Col: 0

Illustrasjon



Binærsøk (Binary Search Algorithm)

- Theory Book IT Grunnkurs
- Kapittel: Algorithm, 5.5

Binærsøkealgoritmen

- Binærsøkealgoritmen fungere på samme måte som vi kan spille gjettespillet fra uke 39.
  - Gjetter på den midterste verdien, og fjerner dermed halvparten av mulighetene for hver gjetting.

gjettespill.py

Rekursiv algoritme for binærsøk

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

binary_search.py

binary_search.py

binary_search.py - /Users/terjery/Library/Mobile Documents/com~apple~CloudDocs/ITGK 2016/Python/Keynote/Uke 45/Python/binary_search.py (3.5.1)

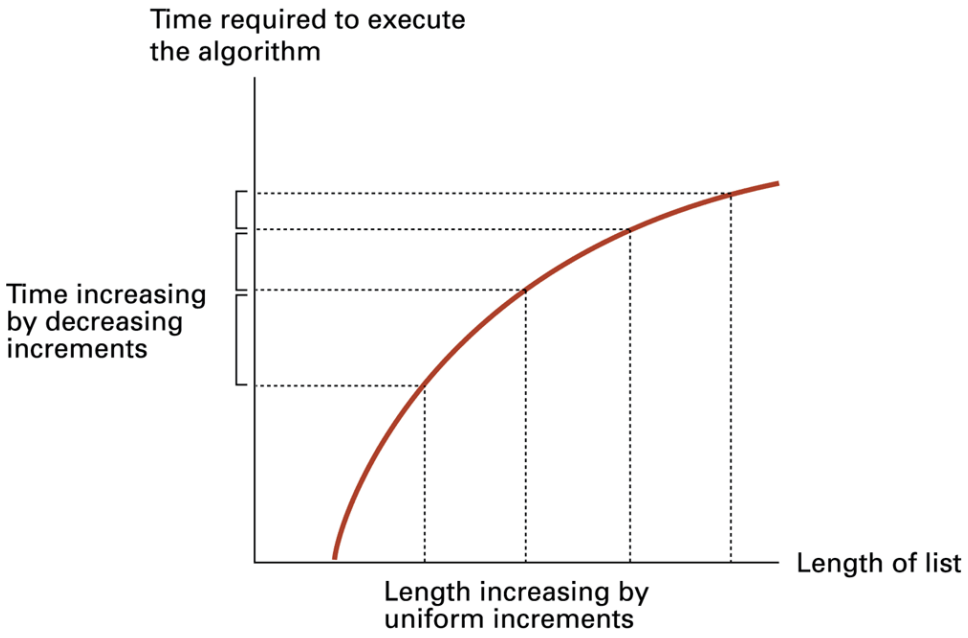
```python
# Name: binary_search
# Input: a list and a search value
# Ouput: True if searchValue is found, else False
# Description: Search through a sorted list using the binary search algorithm

def binary_search(liste, verdi, imin, imax):
    if(imax < imin):
        return False
    else:
        imid = (imin+imax)//2 # Midtpunktet
        if (verdi<liste[imid]):
            return binary_search(liste,verdi,imin,imid-1)
        elif (verdi>liste[imid]):
            return binary_search(liste,verdi,imid+1,imax)
        else:
            return imid

A = [1,2,3,9,11,13,17,25,57,90]
result = binary_search(A,57,0,len(A)-1)
print(result)
```

Ln: 1 Col: 0

Karakteristikker av binærsøk

- Hva er best-case?
  - Treffer på midten første gang
  - Tidsbruk: $\Omega(1)$
- Hva er worst-case?
  - Halverer lista helt til det er ett element igjen
  - Tidsbruk: $O(\log n)$
- Kan sekvensielt søk være raskere enn binærsøk?
  - Ja hvis sekvensielt søk treffer tidlig i lista
- Begrensninger på binærsøk:
  - Lista må være sortert (pre-prosessert)

Graf over worst-case analyse av binært søk



Oppsummering

- Sekvensielle søk:
  - Enkel å implementere og fungerer på usorterte lister
  - Brute-force
  - Worst-case: O(n)
- Innstikk sortering:
  - Enkel å implementere og fungerer som stokking av kortstokk
  - Worst-case: $O(n^2)$
- Binærsøk:
  - Kan implementeres rekursivt og fungerer kun på sorterte lister
  - Worst-case: O(log n)