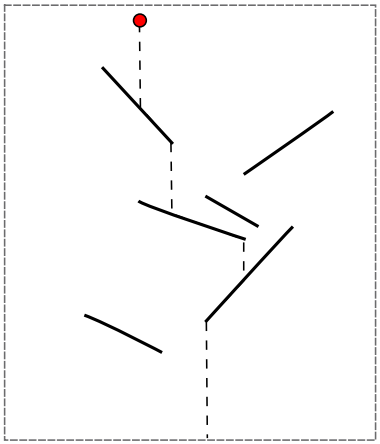
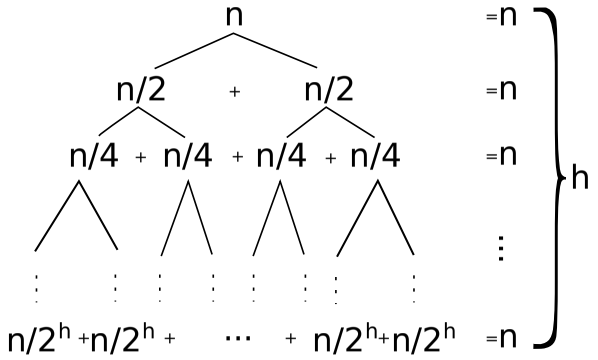
Introduktion Til Konkurrenceprogrammering



Søren Dahlgaard og Mathias Bæk Tejs Knudsen
{soerend,knudsen}@di.ku.dk

Version 0.1



Figur 2.2: Illustration af antal sammenligninger for merge short.

2.4 Køretid for merge sort

Vi vil nu prøve at diskutere hvor hurtig merge sort er på en liste af n tal. Det vil vi gøre ved at tælle hvor mange gange funktionen *sammenligner* værdien af to tal. Vores påstand er, at dette antal er en god estimering for hvor effektiv funktionen er.

Lad os forestille os, at $n = 16$. Vores funktion deler nu op i to lister af størrelse 8, sorterer dem, og sætter dem sammen ved at lave højest 16 sammenligninger. Når vi sorterer de to lister med 8 elementer, deler vi dem hver op i to lister med 4 elementer, og bruger højest 8 sammenligninger til at sætte dem sammen. Sådan fortsætter vi indtil vi har 16 lister med 1 element. Vi har således brugt højest $16 + 2 \cdot 8 + 4 \cdot 4 + 8 \cdot 2 = 4 \cdot 16$ sammenligninger til at sortere de 16 tal.

Hvis vi prøver at bruge den samme argumentation for et generelt n får vi situationen illustreret i fig. 2.2. På første niveau bruger vi n sammenligninger, på næste bruger vi $2 \cdot n/2 = n$ sammenligninger, osv. Antallet af niveauer er netop antallet af gange vi kan dele n med 2 før vi når ned til 1, dvs. $n/(2 \cdots 2)$, hvor antallet af to-taller er antallet af niveauer. Dette tal er kendt som $\log_2(n)$.

Vi kan nu konkludere, at merge sort bruger højest $n \cdot \log_2(n)$ sammenligninger til at sortere en liste.

Indhold

```
1  def mergesort(L):
2    if len(L) == 1:
3      return L
```

Hvis listen har flere end et element, så deler vi den i to på midten, sorterer de to dele rekursivt og sammenfletter de to sorterede lister med vores `merge` funktion:

```
1    else:
2      mid = len(L)/2
3      L1 = mergesort(L[:mid])
4      L2 = mergesort(L[mid:])
5      return merge(L1,L2)
```

Det kan være en fordel at forholde koden til fig. 2.1 for at forstå hvordan og hvorfor den fungerer.

Øvelser

Opgave 2.5. Lav en funktion der sorterer en liste `L` faldende (dvs. det største tal først).

Opgave 2.6. Lav en funktion, der får en liste `L` og returnerer en sorteret version af alle de lige tal i `L`. *Hint: Se på fig. 2.1 og overvej hvor i processen der skal ændres.*

Opgave 2.7*. Lav en version af merge sort, der deler listen i tre dele i stedet for to. *Du kan bruge, at* `merge(L1,L2,L3) = merge(L1,merge(L2,L3))`.

nu følgende: Det mindste element i de to lister er enten `L1[0]` eller `L2[0]`. Vi finder det mindste element og sætter det forrest. Hvis det mindste element var `L1[0]` har vi nu to lister tilbage: `L2` og `L1[1:]` (som er hele `L1` uden det første element). Vi kan nu bruge vores funktion rekursivt på disse to lister og sætte resultatet bag på.

Lad os prøve at implementere denne idé i Python. Først definerer vi funktionen:

```
1  def merge(L1, L2):
2    if L1 == []:
3      return L2
4    elif L2 == []:
5      return L1
```

Vi har nu lavet funktionen, så den kan håndtere at sætte to lister sammen, hvis den ene liste er tom. Dette er vores udgangspunkt for rekursionen. Vi skal nu finde det mindste element og bruge funktionen rekursivt:

```
1    elif L1[0] < L2[0]:
2      return [L1[0]] + merge(L1[1:], L2)
3    else:
4      return [L2[0]] + merge(L1, L2[1:])
```

Bemærk, at hvis det samme tal optræder mere end en gang, er det ligemeget hvilken rækkefølge vi putter dem i.

Vi har nu en funktion, der kan sammenflette to sorterede lister til én, og er klar til at implementere selve merge sort i Python. Lad os først definere funktionen. Vi vil bruge en liste med ét element, som udgangspunktet for vores rekursion. Husk, at sådan en liste er trivielt sorteret:

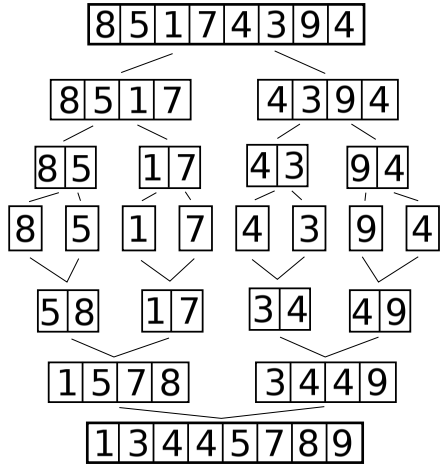
Introduktion

Denne folder er lavet som undervisningsmateriale på Datalogisk Institut, Københavns Universitet (DIKU). Formålet med folderen er at introducere elever på gymnasieniveau til algoritmisk tænkning. Deriblandt metoder og tankegange som bruges til programmeringskonkurrencer. Dette inkluderer også en kort introduktion til programmering i Python.

Folderen er delt op i kapitler, der handler om specifikke emner. Hvert kapitel fokuserer på et hovedproblem og indeholder diverse metoder og mindre opgaver, som bliver brugt til at løse hovedproblemet. Mange af opgaverne og problemerne er taget fra programmeringskonkurrencer såsom IOI og ICPC. Efter hver sektion er der en mængde opgaver, som gennemgår sektionens stof. En opgave markeret med en stjerne ($\star$) er særligt udfordrende.

- Givet to sorterede lister kan vi sammensætte dem til én sorteret liste "nemt".

Merge sort fungerer således ved at dele listen op i to halvdele, sortere dem rekursivt, og så sætte listerne sammen. Denne proces er illustreret i fig. 2.1.



Figur 2.1: Illustration of mergesort.

Vi vil nu gå igennem hvordan vi kan implementere merge sort i Python på samme måde som vi gennemgik problem 1.

Lad os først lave en funktion, der tager to sorterede lister og laver én sorteret liste. Lad os kalde listerne `L1` og `L2`. Idéen er

Hvor vi kan bruge, at $\binom{n}{0} = \binom{n}{n} = 1$ som udgangspunkt.

Skriv en rekursiv funktion der beregner $\binom{n}{k}$.

2.2 Robozzle

Et spil der er rigtigt godt til at introducere rekursion er *Robozzle* som kan findes på `www.robozzle.com`. Vi anbefaler læseren at bruge en times tid på at løse nogle af opgaverne derinde.

Øvelser

Opgave 2.4. Løs følgende opgaver i Robozzle (sorteret efter stigende sværhedsgrad): ID 27, 4993, 3961, 46, 140, 23, 644, 425, 59, 109.

2.3 Sortering

Sortering er et af de mest basale og velstuderede problemer i datalogi. I denne sektion skal vi se på en algoritme kendt som *merge sort*, der kan sortere en liste af tal med det optimale[1] antal sammenligninger. Algoritmen bygger på nogle få observationer, som vi forklarer herunder:

- En liste med ét tal er trivielt sorteret.

[1] Vi vil ikke gå i detaljer med hvad der menes med optimal i denne sammenhæng.

1 Palindromer

Dette kapitel vil fungere som en introduktion til programmering i Python samtidig med, at vi vil arbejde frem mod at løse et klassisk problem. Vi vil beskæftige os med palindromer. Uformelt siger vi, at en streng der er ens forfra og bagfra er et palindrom. Mere formelt:

Definition 1.1. Lad S være en tekststreng og lad $\bar{S} = \texttt{reverse}(S)$ være S baglæns. Da kaldes S et *palindrom* hvis $S = \bar{S}$.

Problemet vi skal løse handler om at finde det længste palindrom i en streng:

Problem 1. *Givet en tekststreng S, hvad er den længste delstreng af S, der også er et palindrom?*

1.1 Introduktion til Python

Lad os starte ud med en kort introduktion til programmering. For at kunne løse problem 1 skal vi være i stand til at læse input og printe løsningen. I python kan vi bruge funktionerne `raw_input`, og `print`. Følgende program læser en tekststreng ind og skriver den igen.

```
1  S = raw_input('Skriv en streng: ')
2  print S
```

> **Syntax:** Python er *case-sensitive*, hvilket betyder at vi ikke må blande store og små bogstaver som vi vil. Hvis vi i stedet havde skrevet `print s` i koden herover, ville den ikke fungere.

Hvis vi vil gøre noget med specifikke tegn i strengen er det lige til. Følgende program læser en streng og skriver det tredje bogstav ud til skærmen. Bemærk at i Python er det første bogstav placeret på plads 0 – Vi kalder dette for 0-indeksering.

```
1  S = raw_input('Skriv en streng: ')
2  print S[2]
```

Det er også muligt at tage et stykke af strengen ud – kaldet en delstreng. Følgende program skriver delstrengen der starter med det andet tegn og slutter med det sjette tegn ud.

```
1  S = raw_input('Skriv en streng: ')
2  print S[1:7]
```



```
1  def fak(n):
2    if n == 0:
3      return 1
4    else:
5      return fak(n-1) * n # Beregn (n-1)! rekursivt.
```

Bemærk, at vi har puttet en kommentar i koden herover.

> **Syntax:** I Python betyder symbolet `#`, at resten af det der står på linjen er en kommentar, og ikke skal medtages i koden. Det kan være smart at putte kommentarer i sin kode, så andre kan forstå hvad man laver.

Øvelser

Opgave 2.1. Hvis vi kalder den ovenstående funktion `fak` med et negativt tal, giver Python os en fejl. Hvordan kan det være?

Opgave 2.2. *Fibonacci tallene* er defineret som $F_0 = 0$, $F_1 = 1$, $F_i = F_{i-1} + F_{i-2}$. De første syv Fibonacci tal er således $0, 1, 1, 2, 3, 5, 8$.

Skriv en rekursiv funktion, der beregner det i'te Fibonacci tal. Din funktion må gerne give en fejl for $i < 0$.

Opgave 2.3. Binomial koefficienten $\binom{n}{k}$ beregner antallet af forskellige måder man kan vælge k bolde ud af en mængde af n bolde. Den kan beregnes rekursivt som:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

For at forklare idéen bag rekursion kan vi betragte fakultetsfunktionen $n!$. Fakultetsfunktionen er defineret som

$$n! = n \cdot (n-1) \cdots 1 \, .$$

For belejlighed definerer vi, at $0! = 1$.

Eksempel: De første fire værdier af $n!$ er:

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 \\ 2! &= 2 = 1 \cdot 2 \\ 3! &= 6 = 1 \cdot 2 \cdot 3 \end{aligned}$$

Vi kan også udtrykke $n!$ rekursivt, som

$$\begin{aligned} n! &= n \cdot (n-1) \cdots 1 \\ &= n \cdot [(n-1) \cdot (n-2) \cdots 1] \\ &= n \cdot (n-1)! \end{aligned}$$

Dette betyder, at vi kan dele problemet "at beregne $n!$" op i et delproblem, som er at beregne $(n-1)!$ og derefter gange dette resultat med n. Vi løser da det første problem rekursivt. Det er vigtigt, at det problem vi løser rekursivt er *mindre* end det oprindelige problem, da vi ellers aldrig ville blive færdige. Fakultetsfunktionen kan beskrives således i Python:

Bemærk, at selvom vi skriver `S[1:7]` er det kun `S[1]`, ..., `S[6]`, der bliver skrevet ud. Vi kan se det som, at vi skriver delstrengen der starter på plads 1 og har længde $7-1$ ud. Bemærk også, at hvis `S` ikke har 7 tegn vil programmet skrive alt andet end det første tegn ud. Hvis vi vil vide hvad længden af en streng er kan vi bruge `len` funktionen. Følgende program læser en streng og udskriver længden.

```
1  S = raw_input('Skriv en streng: ')
2  print len(S)
```

I Python er det også muligt at lægge strenge sammen. Følgende program læser en streng og skriver den ud to gange.

```
1  S = raw_input('Skriv en streng: ')
2  print S+S
```

Vi kan også arbejde med talrækker. Følgende program skriver alle tallene fra 0 til 9 ud.

```
1  for i in range(0,10):
2      print i
```

Funktionen `range` kan bruges til at generere en liste af tal. I ovenstående program bad vi om alle tal mellem 0 og 10 (ikke inklusiv 10). Nøgleordet `for` er en såkaldt "løkke"-konstruktion, som går igennem hvert element i listen, som `range` har genereret og udfører noget kode for hvert element. Alternativt kunne vi blot have skrevet `print range(0,10)`.

> **Syntax:** I Python er det vigtigt at afslutte løkkekonstruktioner med et kolon samt at indentere de linjer, der er omfattet af løkken. Koden herover skal læses som *For hvert element i listen med tal fra* 0 *til* 10*: Print elementet til skærmen.* Uden kolonet ville Python give en fejl, da den ikke ved hvornår selve "indholdet" af løkken starter.

Udover et start- og sluttal kan vi bede `range` om at tage større skridt. Følgende program udskriver alle lige tal mellem 0 og 9

```
1 for i in range(0,10,2):
2     print i
```

Vi kan også gå den anden vej. Følgende program skriver alle lige tal mellem 1 og 10 i omvendt rækkefølge.

```
1 for i in range(10,0,-2):
2     print i
```

> **Bemærk:** Hvis vi i koden i stedet havde skrevet `range(0,10,-2)` ville programmet ikke printe noget. Det er fordi `range` ser 0 som "starten" og 10 som "slutningen", og det er ikke muligt at lægge -2 til 0 nok gange til at nå 10.

2 Rekursion

Dette kapitel vil fungere som en introduktion til en utroligt nyttig teknik, som vi vil gøre meget brug af: *Rekursion*. Vi vil fokusere på at løse et af de mest klassiske problemer i datalogi: At sortere en liste af tal.

Problem 2. *Givet en liste L af tal, lav en liste L' med de samme tal som L, som er sorteret fra mindste til største.*

Vi vil starte med en introduktion til rekursion med nogle simple eksempler, der illustrerer idéen.

2.1 Introduktion til rekursion

Rekursion er en teknik til at løse et problem ved at dele problemet op i mindre instanser. Disse problemer løses så *rekursivt* og sættes sammen til en løsning af det oprindelige problem.

Opgave 1.11. Skriv et "effektivt" program, der finder det længste palindrom af lige længde. *(Hint: Bemærk, at et palindrom af lige længde vil have to ens tegn i midten.)*

Opgave 1.12. Brug din løsning til opgave 1.11 til at lave et "effektivt" program, der finder det længste palindrom i en streng.

1.5 Review

Vi har i dette kapitel kigget på at løse problem 1 på en effektiv måde. Vi har lært hvordan man håndterer strenge og ranges i python, og hvordan vi laver løkker og funktioner. Vi har lavet et program der kan løse problem 1 med cirka n^3 instruktioner og vi har i opgave 1.12 lavet et program, der løser det med kun n^2 instruktioner.

Selvom vores løsning på problem 1 virker rimeligt godt skal det siges, at der findes en endnu mere effektiv algoritme til at løse problemet. Denne er kendt som Manacher's algoritme og bruger kun cirka n instruktioner på en streng af længde n.

For at løse problem 1 vil det være smart at kunne vende en streng om. Vi kan kombinere vores viden om tal og strenge til at udskrive en streng ud baglæns.

```
1 S = raw_input('Skriv en streng: ')
2 y = ''
3 for i in range(len(S) - 1, -1, -1):
4     y = y + S[i]
5 print y
```

For ikke at skulle skrive koden igen hver gang vi vil vende en streng om, kan vi lave det om til en funktion. Følgende program gør det samme som det forrige, men bruger en funktion i stedet.

```
1 def reverse(z):
2     y = ''
3     for i in range(len(z)-1, -1, -1):
4         y = y + z[i]
5     return y
6
7 S = raw_input('Skriv en streng: ')
8 print reverse(S)
```

Vi er nu klar til at lave et program, der kan finde ud af om en tekststreng er et palindrom eller ej

```
1 S = raw_input('Skriv en streng: ')
2 if S == reverse(S):
3     print 'Ja'
4 else:
5     print 'Nej'
```

Dette program bruger funktionen fra det forrige til at teste om den givne streng er det samme forfra og bagfra. Dette gøres

med en *if-sætning*, som kan bruges når vi vil teste om noget er sandt eller ej og træffe et valg afhængigt af dette.

Øvelser

Opgave 1.1. Skriv et program der læser en streng ind og skriver hvert andet bogstav ud.

Opgave 1.2. Skriv et program der læser en streng ind og skriver længden ud hvis det er et palindrom.

Opgave 1.3. Skriv et program der læser en streng ind, bytter det sidste bogstav ud med det første og skriver ud om den resulterende streng er et palindrom.

Opgave 1.4. Skriv et program der printer alle tal fra 1 til 1000 som er delelig med både 2 og 5.

Opgave 1.5. Skriv et program der læser en streng ind og finder ud af om strengen er et palindrom hvis man sletter hvert andet bogstav.

Opgave 1.6.* Skriv et program der læser en streng ind og finder ud af om strengen er et palindrom hvis man må fjerne et af bogstaverne *(Hint: Prøv alle muligheder for at fjerne et bogstav).*


```
2       best = z[i]
3       for j in range(1, min(i+1, len(z) - i)):
4           if z[i-j] == z[i+j]:
5               best = z[i-j:i+j+1]
6           else:
7               break
8       return best
9
10  S = raw_input('Skriv en streng: ')
11  i = int(raw_input('Skriv et tal: '))
12  print 'Bedste palindrom: ' + longestpal(S,i)
```

Med ovenstående funktion kan vi nu lave et program, der finder det længste palindrom af ulige længde ved at prøve at bruge alle positioner i strengen som midterpunkt.

```
1   S = raw_input('Skriv en streng: ')
2
3   best = ''
4   for i in range(0, len(S)):
5       long = longestpal(S, i)
6       if len(long) > len(best):
7           best = long
8   print best
```

1.4 Øvelser

Opgave 1.10. Brug ovenstående kode til at lave et "effektivt" program der finder det længste palindrom af ulige længde, som starter med et *a*.

5 indeholder således et palindrom af længde 3 samt to ens tegn på hver side (`kajak` indeholder f.eks. palindromet `aja` i midten).

I sidste sektion fandt vi det længste palindrom ved at prøve at teste om alle delstrenge var et palindrom. Nu vil vi i stedet lave en funktion, der kan svare på spørgsmål af formen "hvad er det længste palindrom der har position `i` som midterpunkt". Følgende program finder det længste palindrom, der har det femte tegn som midterpunkt.

```
1  S = raw_input('Skriv en streng: ')
2
3  best = S[4]
4  for i in range(1, min(5, len(S)-4)):
5      if S[4-i] == S[4+i]:
6          best = S[4-i:4+i+1]
7      else:
8          break
9  print best
```

Det smarte ved ovenstående kode er, at vi kun behøver at lave én ny sammenligning hver gang vi vil teste om der findes et større palindrom. Vi checker simpelthen først om der er et palindrom af størrelse 3, så 5, så 7, osv. og hvis der ikke er et palindrom af størrelse 5 kan der heller ikke være et af størrelse 7.

Vi kan bruge ovenstående teknik til at finde det længste palindrom med et vilkårligt tegn som midterpunkt. Dette bliver gjort af følgende funktion:

```
1  def longestpal(z, i):
```


1.2 Længste palindrom

Vi skal nu prøve at håndtere problem 1. Vi så i sidste sektion hvordan vi kan checke om en streng er et palindrom. Lad os starte med at lave en funktion der gør det for os:

```
1  def ispal(z):
2      y = ''
3      for i in range(len(z)-1, -1, -1):
4          y = y + z[i]
5      if z == y:
6          return True
7      else:
8          return False
```

Lad os først lave et program der læser en streng og finder ud af om den indeholder et palindrom af længde 5. Husk, at `S[i:i+5]` giver os delstrengen af `S`, der starter på position `i` og indeholder fem bogstaver. Det bruger vi i følgende program:

```
1  S = raw_input('Skriv en streng: ')
2  for i in range(0,len(S)-5+1):
3      if ispal(S[i:i+5]):
4          print 'Ja: ' + S[i:i+5]
5          break
```

I denne kode gør `break` nøgleordet, at programmet "bryder" ud af `for`-løkken, hvis der bliver fundet et palindrom på fem tegn. Hvis vi fjernede det ville programmet i stedet finde *alle* palindromer på fem tegn. Ligesom vores `ispal` funktion, kan vi generalisere ovenstående program til en funktion, der finder ud af om en streng indeholder et palindrom af længde `k` for et givent tal `k`:

```
1  def haspal(z, k):
2      for i in range(0, len(z)-k+1):
3          if ispal(z[i:i+k]):
4              return z[i:i+k]
5      return False
```

For at finde det længste palindrom kan vi nu tjekke om der er et palindrom af længde `len(S)`, et af længde `len(S) - 1`, osv. Følgende program finder det længste palindrom i en streng:

```
1  S = raw_input('Skriv en streng: ')
2
3  for i in range(len(S),0,-1):
4      z = haspal(S, i)
5      if z:
6          print z
7          break
```

Da programmet stopper så snart det finder et palindrom af en given længde og vi prøver de største længder først, er vi garanteret, at det er det længste palindrom der bliver fundet.

Øvelser

Opgave 1.7. Skriv et program der finder det længste palindrom, som starter med et *a*.

Opgave 1.8. Skriv et program der finder det længste palindrom af ulige længde. *(Hint: Du kan bruge `%`-operatoren til at beregne rest ved division med heltal – f.eks. `21 % 2 == 1`).*

Opgave 1.9* Skriv et program der finder det længste palindrom, som ikke indeholder det samme bogstav mere end tre gange.

1.3 Effektiv beregning

Vi har nu lavet et program, der finder det længste palindrom i en streng, men hvis vi prøver at finde et palindrom i en nogenlunde lang streng (1000 tegn), vil programmet tage meget lang tid om at køre. I denne sektion skal vi prøve at finde en måde at lave vores program mere effektivt.

Før vi går i gang er det vigtigt at gøre klart hvad der menes med ordet "effektivt" i denne sammenhæng. Vi vil prøve at definere et programs effektivitet ud fra hvor mange *instruktioner* det laver på et input af størrelse n (i dette tilfælde længden af input strengen). For vores program kan vi se, at der er n forskellige mulige palindromlængder. For hver af disse længder er der *cirka*[1] n pladser et palindrom kunne starte på, og vi laver cirka n sammenligninger for at se om den givne delstreng er et palindrom. I alt siger vi derfor, at vores program bruger cirka n^3 sammenligninger (instruktioner) på en streng af længde n. Bemærk, at vi sagtens kunne være så heldige, at programmet brugte færre, men vi er mest interesseret i det værste tilfælde.

Vi skal nu se på hvordan vi kan lave et program der bruger cirka n^2 instruktioner ved at udnytte at et stort palindrom indeholder mange små palindromer. Et palindrom af længde

[1] Vi vil ikke gå i detaljer med hvad der menes med cirka i denne sammenhæng.