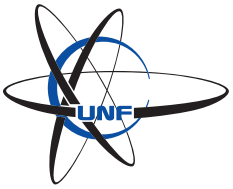
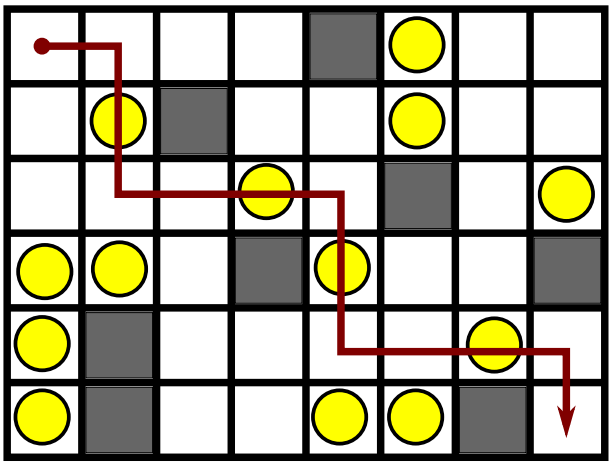


Computer Science Camp

Uge 29, 2015



Ungdommens Naturvidenskabelige Forening

Det faglige team

Søren Dahlgaard (koordinator)	sda@unf.dk
Mathias Bæk Tejs Knudsen (koordinator)	mbk@unf.dk
Kasper Fabæch Brandt	kbr@unf.dk
Davy Leth Eskildsen	dle@unf.dk
Sofie Aleksandra Borup Harning	sabh@unf.dk
Mads Rogild	mro@unf.dk
Simon Binderup Støvring	sbs@unf.dk

Kompendium til UNF Computer Science Camp 2015

Kompendiet er udfærdiget af Søren Dahlgaard, Mathias Bæk Tejs Knudsen, Simon Binderup Støvring og Sofie Aleksandra Borup Harning.

Indhold

Introduktion

Dette kompendium blev brugt ved UNF Computer Science Camp 2015. Kompendiet indeholder en introduktion til programmering samt introduktion til flere datalogiske emner. Disse emner er *rekursion*, *dynamisk programmering* og *grafteori*.

Gennemgangen af hvert emne tager udgangspunkt i relevante problemer, og viser hvordan vi kan bruge de forklarede teknikker til at løse disse problemer i Python programmeringssproget. Til hvert kapitel følger også en mængde øvelser med udgangspunkt i de problemer kapitlet har arbejdet med, samt en mængde sværere problemer, der kan løses med brug af lignende teknikker. Der er mange øvelser og problemer i kompendiet, så det forventes ikke at du, som læser, løser dem alle sammen. I stedet opfordre vi dig til at udvælge de opgaver som du finder mest interessant.

Til de fleste af opgaverne i kompendiet hører test data, som kan testes på vores online server i løbet af campen. Efter campen vil disse test data blive offentliggjort online på campens hjemmeside sammen med dette kompendium.

God fornøjelse!

Søren Dahlgaard & Mathias Bæk Tejs Knudsen
Faglige koordinatorer

Kapitel 1

Introduktion til Programmering

Dette kompendium guider dig igennem materialet, der undervises i på campen. I første kapitel introduceres du til nogle generelle koncepter indenfor programmering. Derudover præsenteres du for nogle udtryk, som er specifikke for Python, programmeringssproget som bliver udgangspunktet for opgaverne som du stilles i løbet af ugen.

1.1 Terminal

I løbet af campen kommer du til at stifte bekendtskab med din computers terminal. Vi interagerer med terminalen ved at give den kommandoer. Terminal udfører en handling og giver et svar tilbage, hvis nødvendigt.

I løbet af campen vil vi primært bruge terminalen til at compile Python-programmer. Et Python-program har filendelse `.py`. Hvis du på et tidspunkt ser en fil med filendelsen `.pyc`, er det en fil som indholder byte code. Python oversætteren, opretter byte code filerne og tolker dem.

Tabel 1.1 viser et udpluk af kommandoer som terminalen forstår. Disse kommandoer kan være til gavn under campen.

Tabel 1.1: Udvalgte kommandoer, som kan være til gavn under campen.

Handling	**Unix**	**Windows**
Skift mappe	`cd sti/til/mappen`	`cd sti\til\mappen`
Gå en mappe tilbage	`cd ..`	`cd ..`
Vis mapper og filer i nuværende mappen	`ls`	`dir`
Kør et Python program	`python file_name.py`	`python file_name.py`

1.2 Variabler

Variabler bruges til at holde en reference til data, som vi senere ønsker at anvende på et senere tidspunkt.

```
1 my_var = "Hello"
```

Ovenstående kode viser, hvordan vi kan gemme tekststrengen "Hello" i variablen `my_var`. Variabelnavne kan indeholde alle bogstaverne A-Å, tallene 0-9 samt underscore. Variabelnavne er case-sensitive, dvs. $aA \neq AA$.

Vi kan senere oprette en ny variabel, `my_new_var`, og lægge den sammen med `my_var`. Resultatet af additionen varierer mellem datatyperne.

```
1 my_var = "Hello"
2 my_new_var = " world!"
3 my_result = my_var + my_new_var # Result is a string with the
      contents "Hello world!"
```

I afsnit 1.3 ser vi nærmere på nogle af de typer data der kan gemmes i variabler.

1.3 Data Typer

Vi bruger data typer til at repræsentere dataen programmet arbejder med. Python stiller følgende typer til rådighed samt nogle mere komplekse typer som vi ikke stifter bekendtskab med i dette kompendium.

Boolske Typer

De boolske typer er en sandhedsværdi. En variabel med en boolsk type kan enten være `True` eller `False`.

Numeriske Typer

Vi bruger numeriske typer til at repræsentere tal. `int` bruges til at repræsentere heltal og `float` til at repræsentere decimaltal.

Tekststreng

En streng er en sekvens af tegn og repræsenteres af `str`-typen. Strenge indkapsles i gåseøjn, eksempelvis er "`Min streng`" en streng.

Sekvenser

Sekvenser kan repræsenteres med lister og tupler. Vi kan oprette en liste af betydningsfulde personer og gemme dem i variablen `imp_pers` med nedenstående udtryk. Bemærk at vi gemmer repræsenterer navnene i en streng, dvs. navnene er af `str`-typen.

```
1 imp_pers = ["Larry Page", "Sergey Bin", "Bill Gates", "Steve Jobs"]
```

Når vi arbejder med lister er vi ofte interesserede i at få fat i et bestemt element i listen. Det kan vi gøre ved at refererer til elementet med dets indeks hvor det første element i listen har indeks 0. Dvs. `imp_pers[0] = "Larry Page"`, `imp_pers[1] = "Sergey Bin"`.

Tabel 1.2 viser nogle af operationer man kan lave på lister.

Tabel 1.2: Udpluk af de operationer man kan lave på lister.

Udtryk	Resultat
`"Steve Jobs" in imp_pers`	Tjek om et element eksisterer i listen.
`"Bill Gates" not in imp_pers`	Tjek om et element ikke eksisterer i listen.
`imp_pers[i]`	Hent elementet med indeks `i`.
`imp_pers[i:j]`	Hent elementerne i intervallet `i` til `j`.
`len(imp_pers)`	Antallet af elementer i listen.
`min(imp_pers)`	Mindste element i listen.
`max(imp_pers)`	Højeste element i listen.
`imp_pers.index("Steve Jobs")`	Indeks af elementet.
`imp_pers.count("Larry Page")`	Antallet af specifikke elementer i listen.
`imp_pers.append("Mark Zuckerberg")`	Tilføjer et element til enden af listen.
`imp_pers.remove("Bill Gates")`	Fjerner et element fra listen.
`imp_pers.pop(i)`	Fjerner og returnerer elementet på indeks `i` fra listen.

Tupler opfører sig meget som lister, men modsat lister kan vi ikke ændre dem efter de er oprettet. Det vil sige, at vi hverken kan tilføje eller fjerne elementer fra en tupel.

En tupel oprettes som vist nedenfor. Bemærk at vi bruger paranteser når vi opretter tuplen og ikke firkantede paranteser, som vi gjorde da listen blev oprettet.

```
1  imp_pers = ("Larry Page", "Sergey Bin", "Bill Gates", "Steve Jobs")
```

Konvertering

Det kan sommetider være nødvendigt at konvertere fra en type til en anden. Hvis programmet får et input som en streng, eksempelvis `"2"`, og er interesseret i at lave aritmetiske operationer på inputtet, konverterer vi det til et heltal, en `int`.

Det gør vi med `int` funktionen som vist nedenfor.

```
1  my_str = "2"
2  my_int = int(my_str)
```

Der findes ligeledes funktionerne `str` og `float` til at konvertere til henholdsvis en streng og et decimaltal.

Konverting er vigtigt, når vi arbejder med brugerinput. `raw_input` funktionen, som benyttes til at få input fra brugeren, returnerer altid en streng. For at lave aritmetiske operationer på værdien returnet fra `raw_input`, skal strengen konverteres til et tal.

Kodeudsnit 1.1 viser et eksempel på, hvordan `raw_input` benyttes til at tage to input fra brugeren og gemme dem i variablerne `a` og `b`. Indeholdet af de to variabler konverteres til heltal med `int`-funktionen hvorefter de adderes og resultatet printes med `print`-funktionen.

Listing 1.1: Eksempel på at addere to tal givet af brugeren.

```
1 a = raw_input("Enter a number: ")
2 b = raw_input("Enter one more number: ")
3 print int(a) + int(b)
```

`print`-funktionen er nærmere beskrevet i afsnit 1.5 og `raw_input` er beskrevet i større detalje i afsnit 1.7.

Øvelser

Opgave 1.3.1. `[intro_hello_world]` Lav et program, som udskriver strengen "`Hello world!`".

Opgave 1.3.2. `[intro_repeat]` Lav et program, der indlæser en streng s og udskriver denne streng to gange.

1.4 Aritmetik

I programmering har vi ofte brug for at lave udregninger på tal. Derfor har Python implementeret en række forskellige aritmetiske operatorer. Tabel 1.3 viser hvilke operatorer der er til rådighed og hvordan de bruges.

Tabel 1.3: De implementerede aritmatiske operatorer der er til rådighed i Python

Operator	Navn	Eksempel	Resultat
$+$	Addition	$3+6$	9
$-$	Subtraktion	$9-12$	-3
$*$	Multiplikation	$2*3$	6
/	Division	$3/2$ $3/2.0$ $3.0/2.0$	1 1.5 1.5
%	Modolus (rest ved division)	$8\%5$	3
$**$	Eksponent	$2**3$	8
//	Nedrundet division	$3/2$ $3/2.0$ $3.0/2.0$	1 1.0 1.0
$+=$	Læg til x	$x+=3$	
$-=$	Træk fra x	$x-=3$	

Det er ikke alle regneoperationer der er implementeret på denne måde, men modulet `math` indeholder en stor mængde matematiske funktioner. Der står mere om hvordan dette virker i 1.5.

Øvelser

Opgave 1.4.1. `[intro_sum]` Lav et program, som indlæser to heltal, lægger dem sammen og printer resultatet.

Opgave 1.4.2. `[intro_divide]` Lav et program, som indlæser to positive heltal, del det første tal med det andet og print resultatet. Brug nedrundet division.

Opgave 1.4.3. `[intro_mod]` Lav et program, som indlæser to positive heltal og beregner resten ved division. Udskriv resultatet.

Opgave 1.4.4. `[intro_difference]` Lav et program, som indlæser to positive heltal a og b. Lad $c = a - b$. Udskriv tekststrengen $a - b = c$.

Opgave 1.4.5. `[intro_power]` Lav et program som indlæser to positive heltal a og b. Udskriv tallet a^b.

Opgave 1.4.6. `[intro_prodpower]` Lav et program, som indlæser tre positive heltal a, b og c. Beregn $a^b \cdot c$. Udskriv resultatet.

Opgave 1.4.7. `[intro_sumconcat]` Lav et program som indlæser to positive heltal a og b. Lad $c = ab$ være tallet man får når man konkatenerer a og b som tekststrenge. Udregn og udskrev derefter tallet $a + b + c$.

1.5 Funktioner

Funktioner anvendes, når vi ønsker at nedbryde logikken i et program i mindre bestanddele. Vi kan betragte funktioner i programmeringssprog som matematiske funktioner der tager et input og returnerer et resultat.

$$f(x) = x^3 + 2x - 4 \tag{1.1}$$

Vi kan formulere ovenstående funktion f som vist i kodeudsnit 1.2. `def` er et nøgleord i Python, som indikerer at vi er interesseret i at definere en ny funktion. `f` er navnet på funktionen. Ligesom variabler kan funktionsnavne bestå af bogstaverne A-Å, tallene 0-9 samt underscore.

`f` tager én parameter, `x`. Vi kalder funktionen med et argument, som giver parameteren en værdi, som vi kan anvende når funktionen eksekveres. Vi kan kalde `f` med udtrykket `f(4)`, hvor `x` tildeles værdien 4. Resultatet af `f(4)` bliver 68.

Listing 1.2: Definition af funktionen `f`.

```
1  import math
2
3  def f(x):
4      return math.pow(x, 3) + 2 * x - 4
```

`f` bruger `pow` funktionen fra `math`-modulet, et modul af standard funktioner som alle relaterer sig til matematiske operationer. Disse operationer er først tilgængelige, når vi importerer modulet med `import math`.

Samme resultat som `math.pow(x, 3)` kan fås enten med udtrykket `x**3` eller `x * x * x`.

`pow` tager to argumenter, `a` og `b` og returnerer resultatet af a^b. I `f`'s tilfælde er det resultatet af x^3 og for det specifikke kald `f(4)` er det $4^3 = 64$.

`return`-udtrykket får funktionen til at returnere. I `f`'s tilfælde returneres et tal. Nogle funktioner har ikke en retur-værdi.

Bemærk, at `return`-udtrykket er rykket ind i forhold til definitionen af `f`. I Python er indrykningen vigtig og giver programmet betydning. Det er ikke tilfælde i andre programmeringssprog, hvor indrykning og andre blanktegn ikke giver programmet betydning.

Definitionen af en funktion er færdig, når indrykningen stopper.

Modsat matematiske funktioner, er det ikke en forudsætning at funktioner i et programmeringssprog returnerer en værdi.

Ligesom matematiske funktioner, kan funktioner i programmeringssprog også tage flere parametre.

$$g(x, y) = \frac{x}{2y} \tag{1.2}$$

Den matematiske funktion g tager parametrene x og y og udregner hvor mange gange det dobbelte af y går op i x. Funktionen kan formuleres i Python som vist i kodeudsnkit 1.3.

Listing 1.3: Den matematiske funktion g udtrykt i Python.

```
1  def g(x, y):
2      return x / (2 * y)
```

I Python benytter vi `print`-funktionen til vise tekst på skærmen. Et kald til funktioner omringes af pareneteser. Hvis funktionen tager nogle argumenter, skrives de indeni paranteserne, eks. `print("Min besked")`.

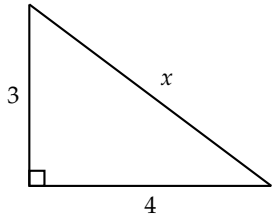
I ovenstående kald til `print`, er "Min besked" et argument til funktionen.

Øvelser

Opgave 1.5.1. Lav et program med funktionen `double_sum` som returnerer den dobbelte sum af sine to parametre `x` og `y`. Kald funktionen med argumenterne $x = 13$ og $y = 8$ og udskriv resultatet.

Opgave 1.5.2. `[intro_magic_num]` Tilføj funktionen `magic_num`, som også tager to parametre `x` og `y`. `magic_num` returnerer resultatet af at kalde `double_sum` med argumenterne `x` og `y`, hvor begge argumenter er sat i anden potens. Udskriv resultatet af at kalde `magic_num`.

Opgave 1.5.3. `[intro_pythagoras]` Lav et program og implementer funktionen `pythagoras`. Funktionen skal implementere Pythagoras' sætning. Kald funktionen for at udregne længden af x og udskriv resultatet.



Opgave 1.5.4. Undersøg i Pythons dokumentation hvad funktionen `math.hypot` gør. Opret et program og implementer selv en tilsvarende funktion med navnet `my_hypot`. Udskriv resultatet af at kalde `my_hypot` med argumenterne $x = 2$ og $y = 7$.

1.6 Betingelsessætninger

Bettingelsessætninger bruges, når vi ønsker at udføre en del af programmet, hvis en eller flere betingelser er opfyldt. Betingelsessætninger evalueres altid til en boolskværdi, det vil sige, at de altid evalueres til enten `True` eller `False`.

Listing 1.4: Kroppen af betingelsessætningen udføres kun, hvis `x` er mindre end 42.

```
1 if x < 42:
2     print "x is less than 42"
```

Kodeudsnit 1.4 viser et eksempel på en betingelsessætning. `if`-nøgleordet indikerer starten på en betingelsessætning. Kroppen af sætningen, dvs. `print`-udtrykket, udføres kun hvis værdien af `x` er mindre end 42. Hvis `x` er mindre end 42 evalueres udtrykket `x < 42` til `True`, ellers evalueres det til `False`.

Vi kan også gemme værdien af det evaluerede udtryk i en variabel og evt. bruge denne i vores `if`-udtryk som vist i kodeudsnit 1.5.

Listing 1.5: Værdien af `x < 42` gemmes i en variabel.

```
1 should_execute = x < 42
2 if should_execute:
3     print "x is less than 42"
```

Vi kan bruge `and`- og `or`-nøgleordene til at kombinere udtryk som vist i kodeudsnit 1.6.

Listing 1.6: `and`- og `or`-nøgleordene kombinerer flere udtryk.

```
1 if x < 42 and y > 8:
2     print "x is less than 42 and y is greater than 8."
```

Et vilkårligt antal udtryk kan kombineres. Vær opmærksom på, at udtrykkene kan være tvetydige. Derfor bruger vi, ligesom i matematiske udtryk, parenteser til at gruppere udtrykkene. Tag et kig på kodeudsnit 1.7 og overvej hvordan betingelsessætningen skal læses.

Listing 1.7: Tvetydig betingelsessætning. Det er ikke tydeligt hvilket udtryk `and` og `or` knytter sig til.

```
1 if x < 42 and y > 8 or z > 19:
2     print "I'm alive!"
```

Udtrykket i kodeudsnit 1.7 er gyldig Python-kode, men betydningen af det er ikke klart. Udtrykket kan læses enten som vist i kodeudsnit 1.8 eller 1.9. De to betingelsessætninger har vidt forskellig betydning. Python tolker sætningen fra 1.7 som vist i kodeudsnit 1.8.

Listing 1.8: Først evalueres udtrykket `x < 42 and y > 8` og derefter `z > 19`.

```
1 if (x < 42 and y > 8) or z > 19:
2     print "I'm alive!"
```

Listing 1.9: Først evalueres udtrykket `z > 19` og derefter `x < 42 and y > 8`.

```
1 if x < 42 and (y > 8 or z > 19):
2     print "I'm alive!"
```

Sommetider er vi interesserede i at gøre én ting, hvis en betingelsessætning evaluerer til `True` og noget andet, hvis den ikke gør. I disse tilfælde bruger vi `else`-nøgleordet som vist i kodeudsnit 1.10.

Listing 1.10: Eksempel på brug af `else`-nøgleordet.

```
1 if x < 42 and (y > 8 or z > 19):
2     print "I'm alive..."
3     print "...and it feels great!"
4 else:
5     print "I'm not alive."
```

Der er også situationer, hvor vi er interesserede i at gøre ét hvis en sætning er opfyldt og noget andet, hvis en anden sætning er opfyldt. Til dette bruger vi `elif`-nøgleordet som vist i kodeudsnit 1.11.

Listing 1.11: Eksempel på brug af `elif`-nøgleordet.

```
1 if x < 42 and (y > 8 or z > 19):
2     print "I'm alive..."
3     print "...and it feels great!"
4 elif x == 42:
5     print "Awesome!"
6 else:
7     print "I'm not alive."
```

Hvis vi er interesserede i at udføre kroppen af en betingelsessætning, hvis udtrykket `x` evaluerer til `False`, kan vi bruge operatoren til negation. Der gælder, at hvis `x` evaluerer til `False`, så evaluerer udtrykket `!x` til `True` og vice versa.

Tabel 1.4 viser nogle af de operatorer og nøgleord vi kan bruge i betingelsessætninger.

Øvelser

Opgave 1.6.1. Kopier koden fra kodeudsnit 1.12 ind i dit redigeringsværktøj. Forsøg at bestemme hvad programmet printer for hver af nedenstående værdier før du kører programmet.

- 100
- 28
- 16
- 14

Tabel 1.4: Udpluk af de operatorer og nøgleord vi kan bruge i betingelsessætninger.

Operator/nøgleord	Resultat
`a == b`	Evaluerer til `True` hvis `a` er lig med `b`.
`a < b`	Evaluerer til `True` hvis `a` er mindre end `b`.
`a > b`	Evaluerer til `True` hvis `a` er større end `b`.
`a <= b`	Evaluerer til `True` hvis `a` er mindre end eller lig med `b`.
`a >= b`	Evaluerer til `True` hvis `a` er større end eller lig med `b`.
`x and y`	Evaluerer til `True` hvis udtrykket `x` og udtrykket `y` evaluerer til `True`.
`x or y`	Evaluerer til `True` hvis udtrykket `x` eller udtrykket `y` evaluerer til `True`.
`not x`	Evaluerer til `True` hvis negeringen af udtrykket `x` evaluerer til `True`, dvs. hvis `x` evaluerer til `False`.

- -5

Listing 1.12: Kopier koden ind i dit redigeringsværktøj. Prøv at bestemme outputtet for hver værdi 28, 100, 16 og -5 af `num`.

```
1  num = 0 # Change the value to 28, 100, 16 and -5
2  if num > 27 and False:
3      print "Hey!"
4  elif num < 31 and num > 27:
5      print "Yo!"
6  elif num > 27:
7      print "Hephey!"
8  elif num < 26 and num < 15:
9      print "Howdy!"
10 elif num < 19:
11     print "Hi!"
```

Opgave 1.6.2. `[intro_age]` Lav et program, som klassificerer personer baseret på deres alder. Givet en alder, skal programmet printe aldersgruppen ud. Aldre og aldersgrupper findes i tabel 1.5.

Tabel 1.5: Klassificering af personer baseret på deres alder.

Alder	Aldersgruppe
0 - 12 år	Barn
13 - 19 år	Teenager
20 - 49 år	Ung
50 år og ældre	Gammel

Opgave 1.6.3. `[intro_oddoreven]` Lav et program, som bestemmer om et tal er lige eller ulige. Print "Lige", hvis tallet er lige og "Ulige", hvis tallet er ulige.

Opgave 1.6.4. `[intro_max]` Lav et program, som tager to tal som input og bestemmer hvilket af de to tal der er størst. Udskriv det største af de to.

Opgave 1.6.5. `[intro_greetings]` Lav et program, som tager et navn som input og skriver "`Hejsa `" efterfulgt af navnet, hvis navnet er seks tegn eller færre og "`Yo `" hvis navnet er mere end seks tegn.

Opgave 1.6.6. Lav et nyt program og kopier nedenstående kode ind. Det nuværende timetal gemmes i variablen `hour` og minuttallet gemmes i variable `minute`. Sørg for at programmet altid printer en timetal og minuttal tocifret med formatet `hh:mm`. Dvs. hvis klokken er 9:13 skal programmet udskrive "09:13" og hvis klokken er 20:45 skal det bare udskrie "20:45".

```
1 from datetime import datetime
2 now = datetime.now()
3 hour = now.hour
4 minute = now.minute
```

Ekstra Opgave 1.6.7. `[intro_countdown]` Brug de redskaber du har lært indtil nu til at lave et program, som tager et tal som input og tæller ned fra tallet til nul. Print alle tallene ud.

1.7 Input

Data som kommer ind i programmet kaldes for "input". I Python er der to måder at få input direkte fra brugeren. Det sker enten med `input`- eller `raw_input`-funktionen.

`raw_input`

`raw_input`-funktionen beder brugeren indtaste en streng og returnerer den indtastede streng. Funktionen bruges som vist i kodeudsnit 1.13. `raw_input` kaldes med en streng, som vises til brugeren. Strengen fortæller brugeren, hvad programmet forventer der indtastes. Den indtastede streng gemmes i variablen `name` hvorefter vi hilser på brugeren.

Listing 1.13: Eksempel på brugen af `raw_input`-funktionen.

```
1 name = raw_input("Enter your name: ")
2 print "Hey", name
```

Resultatet af at køre programmet i kodeudsnit 1.13 er som vist nedenfor, såfremt brugerens navn er Simon.

```
>> Enter your name: Simon
>> Hey Simon!
```

Bemærk, at `raw_input` altid returnerer en streng. Hvis brugeren indtaster værdien "2" som navn, vil resultatet ikke være tallet to, men en streng indeholdende tegnet "2".

Dette har en betydning, hvis vi skriver et program som vist i kodeudsnit 1.14.

Listing 1.14: Bemærk at `raw_input` altid returnerer en streng. Følgende kode er ugyldig.

```
1 fav_num = raw_input("Enter your favorite number and I will add 10 to
      it: ")
2 print fav_num + 10
```

Koden vil give en fejl, når brugeren indtaster sit yndlings nummer. Indtaster brugeren eksempelvis "2", giver programmet nedenstående fejl.

```
>> Traceback (most recent call last):
>>   File "<stdin>", line 1, in <module>
>>   File "/path/to/dir/fav_num.py", line 2, in <module>
>>     print fav_num + 10
>> TypeError: cannot concatenate 'str' and 'int' objects
```

Fejlen fortæller, at der opstod en fejl på linje 2, hvor vi forsøger at addere tallet 10 og strengen "2". Python kan lægge tal sammen, men det har ingen betydning at lægge et tal og en streng sammen og derfor giver programmet en fejl.

For at løse problemet, skal `fav_num` konverteres til en integer. Løsningen ses i kodeudsnit 1.15.

Listing 1.15: `fav_num` konverteres til et heltal inden vi adderer med 10.

```
1 fav_num = raw_input("Enter your favorite number and I will add 10 to
      it: ")
2 print int(fav_num) + 10
```

> Vær opmærksom på, at selvom Python ikke understøtter addition for strenge og tal, så understøttes nogle operatorer på tværs af typerne. Eksempelvis vil nedenstående program printe strengen "Hey!" fire gange. Resultatet bliver altså "Hey!Hey!Hey!Hey!".
>
> ```
> 1 print "Hey!" * 4
> ```

input

`input`-funktionen virker på samme måde som `raw_input`, men den retrunerer ikke en streng. Funktionen tager i stedet et input fra brugeren og evaluerer det som Pythonkode.

Kodeudsnit 1.16 bruger `input` i stedet for `raw_input`.

Listing 1.16: Eksempel på brugen af `input`-funktionen.

```
1 name = input("Enter your name: ")
2 print "Hey", name
```

Når brugeren indtaster sit navn, vil programmet give en fejl som vist nedenfor. I nedenstående eksempel er brugerens navn "Simon". Programmet evaluerer inputtet "Simon" som Pythonkode. Her leder programmet efter variablen `Simon`, som ikke er defineret.

```
>> Enter your name: Simon
>> Traceback (most recent call last):
>>   File "`<stdin>"', line 1, in <module>
>>   File "`/Users/simonbs/Desktop/demo.py"', line 1, in <module>
>>     name = input("`Enter your name: "`)
>>   File "`<string>"', line 1, in <module>
>> NameError: name 'Simon' is not defined
```

Hvis vi definerer variablen `Simon` og i dette tilfælde sætter værdien til "Peter", fejler programmet ikke.

Listing 1.17: Eksempel på brugen af `input`-funktionen hvor variablen `Simon` er defineret.

```
1 Simon = "Peter"
2 name = input("Enter your name: ")
3 print "Hey", name
```

Når variablen `Simon` er defineret som `Peter` er resultatet som vist nedenfor. Derfor bør vi bruge `raw_input` i dette tilfælde.

```
>> Enter your name: Simon
>> Hey Peter!
```

Da `input`-funktionen let kan misbruges af ondsindede eller uomhyggelige brugere, er vi ofte interessrede i at bruge `raw_input` og konvertere værdierne til den forventede type, hvis nødvendigt.

Nedenstående kørsel af kodeudsnit 1.16 viser hvordan brugere kan misbruge hullet i koden til at slette en fil på computeren.

```
Enter your name: (os.remove("some_file.txt"), "Simon")[-1]
Hey Simon!
```

I ovenstående tilfælde opretter vi en tupel med følgende to elementer.

1. Resultatet af at funktionskaldet `os.remove("some_file.txt")`, som sletter `some_file.txt` fra computeren.
2. Tekststrengen "Simon", som udgør brugerens navn.

Pythons lister og tupler understøtter negative indeks. Et negativt indeks starter fra slutningen af listen eller tuplen i stedet for starten. Således er indeks -1 det sidste element i listen eller tuplen.

Vi tager det sidste element for at sørge for, at det bliver gemt i variablen `name` så resultatet af at køre programmet umiddelbart ser rigtigt ud, selvom det har den sideeffekt at `some_file.txt` slettes fra computeren.

Det er ikke så vigtigt, at du forstår præcist hvad der skete under kørslen eller hvorfor det virker. Det er derimod vigtigt, at du husker på, at være påpasselig når du bruger `input`-funktionen.

split

Til tider kan det give mening at brugeren skal indtaste flere oplysninger samtidig, og man så selv deler det op i mindre bider bagefter. Til dette kan man bruge funktionen `split`.

Listing 1.18: Udskriv en række ord på hver sin linje.

```
1 string = raw_input("Enter words seperated with space: ")
2 words = string.split()
3 for word in words:
4     print word
```

Med strengen "Hello, World!"som input, vil resultatet af 1.18 blive:

```
>> Hello,
>> World!
```

`split` kan enten bruges som i 1.18, hvor den splitter strengen ved hvert mellemrum og returnere alle sub-strengene i en liste, eller man kan specificere hvilket tegn der skal splittes på som i 1.19.

Listing 1.19: Programmet modtager en dato, delt af "-" og udskriver hvad der er dag, måned og år på hver sin linje. "\n" betyder `newline`.

```
1 date = raw_input("Write date as dd-mm-yyyy: ").split("-")
2 print "Day:", date[0], "\nMonth:", date[1], "\nYear:", date[2]
```

Øvelser

Opgave 1.7.1. `[intro_product]` Lav et program, der læser to heltal på samme linje og udskriver produktet.

Opgave 1.7.2. `[intro_summany]` Lav et program der tager imod flere tal, opdelt med mellemrum, og udskriver summen.

Ekstra Opgave 1.7.3. `[intro_calculator]` Lav en lille lommeregner der kan addere og subtrahere. Den skal tage imod en streng med tal, + og -, f.eks. "`2 + 3 - 1 + 8`" og så udregne og udskrive resultatet.

1.8 Løkker

Når man programmere kan det ofte være praktisk at kunne gentage noget kode flere gange. Dette gøres med løkker. Forskellige programmerings sprog har forskellige løkker, Python har `while`- og `for`-løkker.

while-løkke

`while`-løkker bliver kørt så længe en givet betingelse er opfyldt.

Listing 1.20: Udskriv alle tal fra x til og med 1.

```
1 while x > 0:
2     print x
3     x -= 1
```

Koden i 1.20 vil køre så længe x er større end 0. For hver gennemkørsel vil x blive en lavere, indtil x tilsidst bliver 0, hvorefter `x > 0` vil evaluere til `False`, og linjen under while-løkken vil blive eksekveret.

Listing 1.21: Uendelig løkker.

```
1  while True:
2      print "Running..."
3
4  x = 1
5  while (x > 0):
6      x += 1
```

Når man bruger `while`-løkker, er det vigtigt at være opmærksom på at man ikke kommer til at lave en uendelig løkke, ved at have et statement der aldrig bliver evalueret til `False`. Det er nemt at se at den første løkke i 1.21 aldrig vil blive falsk. Den anden løkke er et mere skjult tilfælde, da den afhænger af værdien af `x`. Den vil aldrig blive falsk, fordi `x` starter på 1 og bliver talt op ved hver iteration, og derfor aldrig vil komme ned på 0.

for-løkke

`for`-løkker kører for hvert element i en sekvens, f.eks. en liste eller en streng.

Listing 1.22: Udskriv alle elementerne i en liste.

```
1  imp_pers = ["Larry Page", "Sergey Bin", "Bill Gates", "Steve Jobs"]
2  for person in imp_pers:
3    print(person)
```

Kodeeksemplet 1.22 viser hvordan man kan bruge en `for`-løkke til at interagere med hvert element i en liste, i dette tilfælde bliver hvert element blot udskrevet.

Listing 1.23: Udskriver alle bogstaver i ordet "Python" og fortæller hvor mange bogstaver det var.

```
1  counter = 0
2  for letter in "Python":
3      print letter
4      counter += 1
5  print "Python is", counter, "letters long"
```

Kodeeksemplet 1.23 viser hvordan man kan bruge en `for`-løkke til at manipulere en streng.

range

Det kan være praktisk at kunne køre en løkke et bestemt antal gange, eller at kunne iterere over en række tal. Til dette kan man bruge funktionen `range`. `range` generere en list ud fra nogle givne parametre.

range(x)

`range(x)` laver en liste der går fra `0` til og med `x-1`.

range(x,y)

`range(x,y)` laver en liste fra `x` til og med `y-1`.

range(x,y,z)

`range(x,y,z)` laver en liste der starte ved `x`, og derefter har hvert `z`'ne element op til det højeste tal i sekvensen inden `y`.

Listing 1.24: Kode der summere alle tallene fra 0-4 og returnere resultatet, altså 10.

```
1 y = 0
2 for x in range(5):
3   y += x
4 print y
```

Listing 1.25: Kode der udskriver hvert andet tal fra 1 til 10, altså alle de ulige tal i intervallet.

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

break og continue

Der kan opstå situationer hvor man har lyst til at afslutte en løkke tidligere end normalt, til dette kan vi bruge `break` og `continue`.

`break` afslutter en løkke og programmet fortsætter med den næste linje i koden. `continue` derimod afslutter den nuværende gennemkørsel af løkken, og hopper tilbage til starten af en `while`-løkke eller til næste element i `for`-løkken.

Listing 1.26: Udskriver alle bogstaver af 'python' indtil den støder på et 'h'.

```
1 for letter in 'Python':
2     if (letter == 'h'):
3         break
4     print letter
```

Kodeeksemplet 1.26 vil udskrive følgende:

```
>> p
>> y
>> t
```

Listing 1.27: Udskriver alle bogstaver af "python" pånær "h".

```
1 for letter in 'Python':
2     if (letter == 'h'):
3         continue
4     print letter
```

Kodeeksemplet 1.27 vil udskrive følgende:

```
>> p
>> y
>> t
>> o
>> n
```

Nestede løkker

Der er muligt at lave nestede løkker, altså løkker inde i løkker. Hvis man bruger et `break` eller `continue` i en nested løkke, vil det kun påvirke den løkke statementet er skrevet i.

Listing 1.28: Eksempel på nestede `for`løkker.

```
1  for x in range(5):
2      for y in range(5):
3          z = y + x
4          if (z > 3):
5              print y, '+', x,'is greater than 3'
6              break
7          print y, '+', x, '=', z
```

Koden i 1.28 udskriver summen for alle talkombinationer medmindre den er større end 3, i hvilket tilfælde den breaker, og forsætter med det næste x.

List Comprehension

Du kender formentlig nedenstående syntax fra matematikens verden. S sættet af alle talene fra 0 til 100 opløftet i anden potens. P er sættet af alle tal i S som er lige.

$$S = \{x^2 | x \in \{0 \dots 100\}\} \tag{1.3}$$

$$P = \{x | x \in S \wedge x \text{ is even}\} \tag{1.4}$$

Ovenstående kan også let formuleres i Python med *list comprehension*. Det er en teknik, hvor vi laver loops, som returnerer lister. S og P kan formuleres i Python som vist i kodeudsnit 1.29.

Listing 1.29: Generering af sættene S og P i Python.

```
1  S = [ x**2 for x in range(0, 101) ]
2  P = [ x for x in S if x % 2 == 0 ]
```

Det første udtryk i listen returnerer værdien, som indsættes i listen. I tilfældet for `S` er det `x **2`, dvs. x^2. Dernæst er der en løkke som løber over alle talene fra 0 til 100. I starten af løkken deklarerer vi, at variablen hedder `x`. Vi bruger altså variablen `x` i udtrykket `x**2` før den er navngivet i løkken. Python oversætteren finder selv ud af dette.

Under genereringen af `P`, udføres også et `if`-udtryk, som bestemmer om værdien af `x` skal gemmes i listen. I dette tilfælde gemmer vi kun værdien, såfremt der gælder, at `x % 2 == 0`, altså at `x` er lige.

Øvelser

Opgave 1.8.1. Hvad sker der, hvis du kalder `range(x)` hvis x er negativ? Hvad sker der, hvis du kalder `range(x,y)` hvis x er større end y? Print `range(5,0,-1)` og `range(0,5,-1)` og se hvad resultatet er.

Opgave 1.8.2. `[intro_countdown]` Lav et program, som tager et tal som input og tæller ned fra tallet til nul. Print alle tallene ud.

Opgave 1.8.3. `[intro_listcomparison]` Lav et program, som indlæser et tal n. Indlæs tallene $x_0, x_1, x_2, \cdot, x_n$ og gem dem i listen X. Dernæst skal du indlæse tallene $y_0, y_1, y_2, \cdot, y_n$ og gemme dem i listen Y. Tjek om de to lister er ens og udskriv `True`, hvis listerne er ens, ellers `False`.

Opgave 1.8.4. `[intro_squareroot]` Lav et program, som udregner kvadratroden af et positivt heltal n. Resultatet skal rundes ned til nærmeste heltal.

Opgave 1.8.5. `[intro_oddnumbers]` Lav et progam, der indlæser et positivt heltal, n, og derefter udskriver alle ulige tal mindre end n.

Opgave 1.8.6. `[intro_factorial]` Lav et program, som indlæser et positivt heltal og udregner og udskriver dets fakultet. Dvs. $n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \ldots \cdot 1$.

Opgave 1.8.7. `[intro_palindrome]` Et palindrom er en streng, der er den samme, hvis den læses bagfra. Fx er `abba` et palindrom, men `abe` er ikke. Lav et program, som indlæser en streng og bestemmer om det er et palindrom. Print `True`, hvis det er et et palindom, ellers `False`.

Kapitel 2

Rekursion

I dette kapitel skal vi fokusere på en af de mest grundlæggende teknikker i datalogi og programmering, nemlig *rekursion*. Rekursion går ud på at tage et stort problem og dele det op i mindre bidder indtil bidderne er så små at de kan løses nemt. Hver af de bidder vi har delt problemet op i løses så på samme måde som det oprindelige problem.

2.1 Introduktion til rekursion

Et af de mest klasiske eksempler på rekursion er *fakultetsfunktionen*. Fakultetsfunktionen af n også skrevet som $n!$ er produktet af de første n heltal – dvs. $n! = 1 \cdot 2 \cdots n$. Derudover definerer vi $0! = 1$ for bekvemmelighed.

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

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 \\ 2! &= 2 \cdot 1 = 2 \\ 3! &= 3 \cdot 2 \cdot 1 = 6 \\ 4! &= 4 \cdot 3 \cdot 2 \cdot 1 = 24 \end{aligned}$$

Det smarte ved fakultetsfunktionen fra et rekursivt perspektiv er, at vi også kan definere den ud fra fakultetsfunktionen for mindre værdier. F.eks. kan vi se, at

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 4 \cdot 3!$$

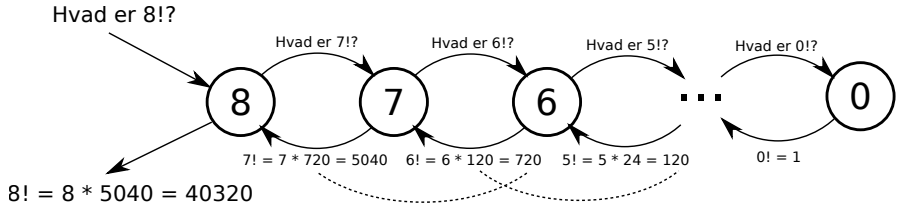
og generelt gælder det, at

$$\begin{aligned} n! &= n \cdot (n-1) \cdots 1 \\ &= n \cdot \Big((n-1) \cdot (n-2) \cdots 1 \Big) \\ &= n \cdot (n-1)! \,. \end{aligned}$$

Det betyder, at vi kan beregne $n!$ ved at gøre følgende to ting:

1. Beregne $(n-1)!$ *rekursivt*.
2. Gange resultatet af ovenstående med n.

Når vi skriver, at vi vil beregne $(n-1)!$ rekursivt betyder det, at fremgangsmåden vi bruger til at beregne $(n-1)!$ er den samme som den vi bruger til $n!$. Altså beregner vi første $(n-2)!$ rekursivt og så ganger vi med $(n-1)$. Dette fortsætter vi med indtil $n = 0$, der er en "lille nok bid" til at vi bare kan returnere at $0! = 1$. Dette er illustreret for $n = 8$ i Figur 2.1



Figur 2.1: Illustration af fakultetsfunktionen der bliver beregnet rekursivt for $n = 8$.

Hvis vi vil implementere fakultetsfunktionen kan vi gøre det med et *rekursivt kald*. Det vil sige, at vi skriver en funktion der *kalder sig selv*. Dette kan virke en smule mærkeligt i starten, men så længe vi sørger for at denne kæde af kald stopper på et tidspunkt (eksempelvis ved 0 i Figur 2.1), er det ikke noget problem.

```
1  def fak(n):
2      if n == 0:
3          return 1
4      return n * fak(n-1)
```

Denne funktion gør præcist det vi har snakket om i denne sektion. Den sørger for at huske, at $0! = 1$ og ellers bruger den de to trin fra tidligere, hvor det ene er det rekursive kald til `fak(n-1)` i linje 4. Vi kan teste at funktionen giver det rigtige svar for små værdier:

```
>>> fak(0)
1
>>> fak(5)
120
>>> fak(8)
40320
>>> fak(20)
2432902008176640000
```

Fakultetsfunktionen er et godt eksempel på hvordan rekursion virker, men den er også meget simpel. I resten af dette kapitel skal vi se på flere eksempler på rekursive funktioner.

Øvelser

Opgave 2.1.1. Hvis vi kalder fakultetsfunktionen fra denne sektion med eksempelvis værdierne -1 og 1.5 giver Python os en fejl. Hvordan kan det være?

Opgave 2.1.2. `[rec_sum]` Lav et program der tager et ikke-negativt heltal n som input. Ved hjælp af en rekursiv funktion skal programmet udskrive summen af de første n heltal.

Opgave 2.1.3. `[rec_sumlist]` Lav et program, der som input tager en liste af heltal og ved hjælp af en rekursiv funktion udskriver summen af disse tal. Listen af tal vil være separeret af mellemrum og fx skal inputtet `1 3 6` give output `10`.

Opgave 2.1.4. `[rec_pairproducts]` Lav et program der tager en liste af heltal som input i samme format som opgaven ovenfår. Ved hjælp af en rekursiv funktion skal programmet udskrive summen af produktet af alle par af tal, der står ved siden af hinanden, i listen. F.eks. skal din funktion give følgende resultat:

$$\texttt{SumProd}([1,2,3,4,5]) = 1\cdot 2 + 2\cdot 3 + 3\cdot 4 + 4\cdot 5 = 2 + 6 + 12 + 20 = 40$$

Hvis funktionen er givet en liste med ét eller færre elementer skal den returnere `0`.

2.2 Flere eksempler

Fakulktetsfunktionen er muligvis det mest brugte eksempel på at illustrere rekursion, men der er mange andre gode eksempler. I denne sektion vil vi kigge på nogle flere eksempler, for at blive mere vant til at bruge rekursion.

Fibonacci tal

En funktion, som vi vil bruge flere gange til at illustrere forskellige teknikker, nemlig *Fibonacci tallene*. Disse tal er opkaldt efter den italienske matematiker af samme navn, der brugte dem til at beskrive kaniners ynglen. Fibonacci tallene er defineret rekursivt således:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-2} + F_{n-1}\ .$$

Eksempel: Fibonacci tallene $F_0, F_1, \ldots, F_{10}$ er:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55\ .$$

For at implementere denne funktion kan vi bruge definitionen direkte. Vi skal dog huske på, at der nu er to *basistilfælde* nemlig $n = 0$ og $n = 1$, og vi har også to rekursive kald. I Python kan vi gøre det på følgende måde:

```
1  def fibonacci(n):
2      if n == 0:
3          return 0
4      if n == 1:
5          return 1
6      return fibonacci(n-2) + fibonacci(n-1)
```

Vi kan prøve at køre dette program for nogle små værdier af n og se om det regner rigtigt:

```
>>> fibonacci(5)
5
>>> fibonacci(10)
55
>>> fibonacci(20)
6765
>>> fibonacci(30)
832040
```

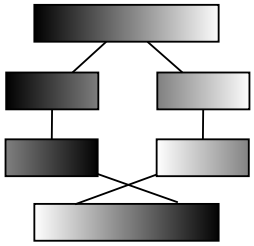
Bemærk, at dette program er meget langsomt, og ikke kan regne særligt store værdier af fibonacci tallene ud. Vi vil se i Section 3.1 hvordan vi kan lave vores funktion bedre, så vi kan beregne langt større værdier ud.

Rekursion med lister og strenge

Et problem man ofte kommer ud for når man arbejder med tekst, er at man skal vende en streng om. I python er dette rigtigt nemt at gøre, da vi kan bruge Pythons liste syntax:

```
>>> s = "asdf"
>>> s[::-1]
'fdsa'
```

Vi skal nu kigge på hvordan vi kan gøre dette med en rekursiv funktion. Idéen er illustreret i Figur 2.2 med en gradient. Idéen er som følger:



Figur 2.2: Idéen ved at vende en streng eller liste om rekursivt illustreret med en gradient.

1. Del strengen op i to halvdele.
2. Vend disse to halvdele om rekursivt.

3. Sæt de to halvdele sammen i omvendt rækkefølge (sidste halvdel først).

Rekursionen stopper ved en streng af længde 1, da denne er ens både forfra og bagfra.

Vi kan implementere denne funktion i Python på følgende måde. Først skal vi håndtere basistilfældet:

```
1  def reverse(s):
2      if len(s) <= 1:
3              return s
```

Dernæst skal vi dele strengen op i to, vende dem om rekursivt og til sidst sætte dem sammen igen. Vi sørger for at dele dem på midten ved at kigge på længden delt med to:

```
4      mid = len(s)/2
5      s1 = reverse(s[:mid])
6      s2 = reverse(s[mid:])
7      return s2+s1
```

Vi kan prøve funktionen efter og se, at den virker:

```
>>> reverse("en af dem der red med fane")
'enaf dem der red med fa ne'
```

Bemærk, at denne funktion dog ikke er at foretrække over Pythons indbyggede funktionalitet, men blot er en god illustration af rekursion.

Rekursion med struktur

Indtil videre har vi lavet rekursion på tal og lister. Dette er fordi de er de konceptuelt nemmeste tilfælder at forstå. Lad os nu kigge på et mere avanceret problem:

Undersåtproblemet

I et land langt væk bor der n mennesker. Landet er opdelt i hierarkier således at hver person er direkte undersåt af netop én anden person. Dette gælder dog ikke for kongen, der ikke er undersåt af nogen. Givet navnet på en person i landet skal du finde ud af hvor mange undersåtter denne person har (ikke kun direkte).

Dette problem er måske ikke umiddelbart til at forstå, så lad os kigge på et eksempel.

Eksempel: Lad $n = 6$ og lad personerne hedde $p_1, p_2, \ldots, p_6$. Det gælder at følgende direkte undersåt relationer gælder (undersåt $\rightarrow$

chef):

$$p_1 \to p_3, \quad p_2 \to p_3$$
$$p_4 \to p_2, \quad p_5 \to p_4$$
$$p_6 \to p_1 \, .$$

I dette land er p_3 altså kongen. Han har 2 direkte undersåtter og 5 undersåtter i alt. p_2 har 1 direkte undersåt (p_4), men 2 undersåtter i alt (p_4 og p_5).

For at finde antallet af undersåtter for en person skal vi altså tælle antallet af personens direkte undersåtter, antallet af deres direkte undersåtter, osv. Dette er præcist den type opgave, som vi kan løse med rekursion!

Vi vil antage, at vi er givet undersåt relationerne som en liste af par med `(navn, navn på chef)`. Altså ville relationerne fra eksemplet se således ud:

```
rel = [("p1","p3"),("p2","p3"),("p4","p2"),("p5","p4"),("p6","p1")]
```

Lad os først lave et program, der tæller antallet af direkte undersåtter under en person:

```
def direkte_undersaatter(L,p):
    antal = 0
    for (pers,chef) in L:
        if chef == p:
            antal = antal + 1
    return antal
```

Dette program læser listen `L` igennem og finder antallet af par, hvor chefen har samme navn som den person, `p`, vi tæller for. Det vil sige, at hver gang vi findet et par `(pers,chef)` med `p==chef` skal vi også finde antallet af undersåtter for `pers`. Da hver person kun har én chef vil hver person højest være i venstre side at et par én gang, så vi skal ikke være bange for at tælle den samme undersåt mere end én gang. Vi kan nu udvide vores funktion fra før med et simpelt rekursiv kald:

```
def undersaatter(L,p):
    antal = 0
    for (pers,chef) in L:
        if chef == p:
            antal = antal + 1 + undersaatter(L,pers)
    return antal
```

Vi kan køre dette program på eksemplet fra før og se at det virker:

```
>>> undersaatter(rel,"p1")
1
>>> undersaatter(rel,"p2")
2
>>> undersaatter(rel,"p3")
5
```

Vi har nu set hvordan vi kan bruge rekursion på en mere avanceret struktur end lister og tal. Dette er blot en forsmag på den type af problemer vi kan bruge rekursion til at løse.

Øvelser

Opgave 2.2.1. `[rec_tribonacci]` Tribonacci tallene er defineret som

$$T_0 = 0, \quad T_1 = 0, \quad T_2 = 1, \quad T_n = T_{n-3} + T_{n-2} + T_{n-1} \, .$$

Dvs. summen af de tre forrige i stedet for de to forrige som i Fibonacci tallene. Skriv et rekursivt program i stil med det for Fibonacci tallene, der udregner det n'te Tribonacci tal.

Opgave 2.2.2. `[rec_binomial]` Binomial koefficienten $\binom{n}{k}$ beregner hvor mange måder man kan tage k bolde af en spand med n bolde i. Vi kan beregne denne funktion rekursivt som:

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

Vi kan bruge basistilfældene:

$$\binom{n}{0} = \binom{n}{n} = 1 \, .$$

Lav et program der indlæser 2 heltal og ved hjælp af en rekursiv funktion udskriver binomial koefficienten $\binom{n}{k}$. Tallene vil stå på hver sin linje.

Opgave 2.2.3. I undersåtproblemet kan vi forestille os, at nogle af beboerne i landet gør oprør. Lad `oproer` være en liste af navne på alle de personer der gør oprør.

Omskriv funktionen `undersaatter` så den ikke tæller folk der gør oprør eller undersåtter af folk der gør oprør med. Hvis f.eks. `oproer = ["p4"]` vil antallet af undersåtter for p_3 kun være 3, da p_4 gør oprør og p_5 er en undersåt af p_3.

Lav et program der først indlæser en undersåt relation i formatet `p1,p2;p3,p4`. Dernæst skal det indlæse liste af oprører i formatet `p1,p2,p3,p4` og tilsidst indlæser en person som vi vil udregne antalet af undersåtter for. Programmet skal så udskrive resultatet af den omskrevet `undersaatter` funktion.

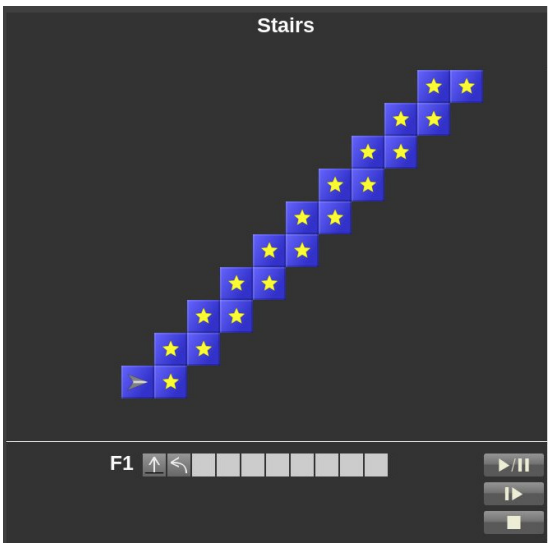
2.3 Robozzle

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 time (eller gerne mere) på at løse nogle af opgaverne på siden. Spillet kan ses på Figur 2.3.

Øvelser

Opgave 2.3.1. Følgende opgaver i Robozzle sorteret efter sværhedsgrad: ID 27, 4993, 3961, 46, 140, 23, 644, 425, 59, 109.

Bemærk at den sidste opgave er meget tricky og benytter sig af det vi kalder for *rekursions-stacken*.



Figur 2.3: Eksempel på et spil Robozzle. Skærmbillede taget fra `www.robozzle.com`.

2.4 Merge sort

Den måske mest arketypiske rekursive algoritme bruges til at sortere lister og hedder *merge sort*. Sortering er et af de mest basale og velstuderede problemer i datalogi. Merge sort minder meget om den algoritme vi lavede til at vende en streng om tidligere. Den deler netop listen op i to dele og sorterer dem rekursivt. Dette gør algoritmen ved at benytte to simple observationer:

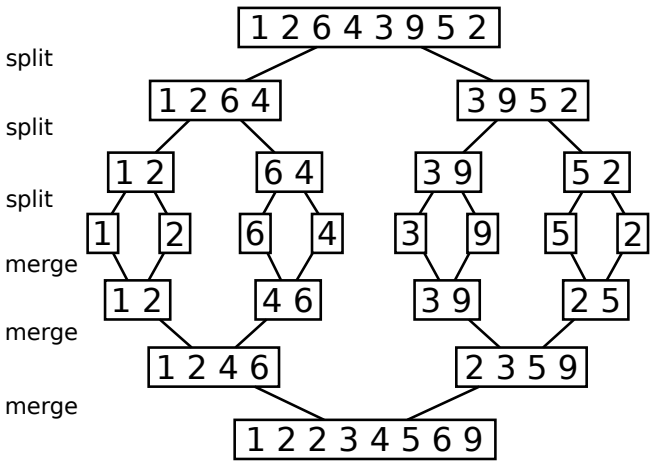
1. En liste med ét tal er trivielt sorteret.
2. Givet to sorterede lister kan vi sammensætte dem til én sorteret liste "nemt".

Merge sort fungerer således ved at dele listen op i to halvdele, sortere dem rekursivt, og "flette" listerne sammen igen. Denne proces er illustreret i Figur 2.4.

Lad os prøve at implementere denne algoritme i Python. Først skal vi bruge en funktion, der tager to sorterede lister og sætter dem sammen til én sorteret liste. Lad os kalde listerne `L1` og `L2`. Idéen er 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 f.eks. det mindste element var `L1[0]` har vi nu to lister tilbage: `L2` og `L1[1:]`. 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 og håndterer basistilfældet, hvor den ene af de to lister er tom (da er svaret blot den anden liste):

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



Figur 2.4: Illustration of mergesort.

```
4     elif L2 == []:
5       return L1
```

Vi har nu udgangspunktet for rekursionen. Tilbage er blot at finde det mindste element og bruge funktionen rekursivt, som beskrevet herover:

```
6     elif L1[0] < L2[0]:
7       return [L1[0]] + merge(L1[1:], L2)
8     else:
9       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. Ligesom i vores `reverse` funktion vil vi bruge en liste med ét element, som basistilfældet, da sådan en liste er trivielt sorteret.

```
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:

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

Det kan være en fordel at forholde koden til figur 2.4 for at forstå hvordan og hvorfor den fungerer.

Øvelser

Opgave 2.4.1. Lav et program indlæser en liste af tal i formatet `1,2,3,4,5` og sorterer listen faldende (dvs. det største tal først) og udskriver det.

Opgave 2.4.2. Lav en ny version der kun soterer alle de lige.
Hint: Se på figur 2.4 og overvej hvor i processen der skal ændres.

Opgave 2.4.3. Lav en ny version af merge sort, der deler listen i tre dele i stedet for to.
Hint: Du kan bruge, at `merge(L1,L2,L3) = merge(L1,merge(L2,L3))`.

2.5 Review

I dette kapitel har vi lært om en af de mest benyttede teknikker inden for datalogi og programmering: *Rekursion*. Vi har brugt denne teknik til at udregne rekursive funktioner som Fibonacci tallene, tæller antallet af undersåtter i et hierarki og sortere en liste af tal. Vi har set at man kan bruge rekursion på en række forskellige strukturer, og hvordan man kan løse relativt avancerede problemer med meget få linjer kode ved at fokusere på at løse små bidder af problemerne for sig.

2.6 Problemer

Problem 2.1. *Farvede stænger*

Problem 2.2. *Fotobog*

Problem 2.3. *Nulsum*

Problem 2.4. *Gammeldags tastatur*

Farvede stænger[1]

Lars har fundet en stak med forskellige farvede stænger. Disse stænger har hver længden 1, og kan sættes sammen til en længere stang. Hver stang er enten rød, blå eller grøn.

Lars har én regel: Der må ikke sidde to blå stænger efter hinanden. Han er nu interesseret i hvor mange forskellige stænger af forskellige længder.

Opgave

Skriv et program, der beregner hvor mange forskellige stænger man kan lave af længde n.

Input

Input består af netop én linje, der indeholder et positivt heltal: n.

Du kan antage, at der altid er mindst n stænger af hver farve.

Output

Et heltal: Antallet af stænger det er muligt at lave af længde n.

Eksempler

Input	Output
3	22

Input	Output
4	60

Pointgivning

Delopgave 1 (100 point): $1 \leq n \leq 15$

Begrænsninger

Tidsbegrænsning: 1 s.

Hukommelsesbegrænsning: 256 MB.

[1] Denne opgave er taget fra DDD 2015.

Fotobog[1]

Simon har en masse billeder fra sin sidste ferie, og han ønsker at placere dem i en fotobog. Han har dog en mærkelig måde at arrangere sin fotobog: På hver side skal der være enten lige så mange billeder, som på den forrige side, eller der skal være dobbelt så mange. På første side skal der altid være netop et billede.

Simon vil nu vide hvor få sider han kan nøjes med at bruge.

Opgave

Skriv et program, der beregner det mindste antal sider Simon skal bruge til at placere n billeder.

Input

Input består af netop én linje, der indeholder et positivt heltal: n.

Output

Et heltal: Det mindste antal sider der skal bruges til at placere n billeder.

Eksempler

Input	Output
3	2

Input	Output
70	9

Pointgivning

Delopgave 1 (100 point): $1 \leq n \leq 100$

Begrænsninger

Tidsbegrænsning: 1 s.

Hukommelsesbegrænsning: 256 MB.

[1] Denne opgave er taget fra DDD 2015.

Nulsum[1]

Simon er begyndt i 2. klasse og er vild med matematik. Han har lige lært at lægge til og trække fra, og har et interessant spørgsmål:

Forestil dig, at du har tallene fra 1 til n skrevet ned efter hinanden. Du må nu indsætte plusser og minusser mellem tallene. Du må også indsætte et mellemrum imellem to tal. Du kan f.eks. indsætte et mellemrum mellem 2 og 3, og så bliver de til tallet 23. Spørgsmålet er nu: Hvad er alle måderne at indsætte `+`, `-` og `' '` for at regnestykket giver 0?

Hvis vi f.eks. har $n = 7$, så kan vi se, at

```
1-2 3+4+5+6+7 = 0
```

Opgave

Skriv et program, der læser et tal, n, og finder alle måder man kan indsætte +, - og ' ' mellem tallene fra 1 til n således at regnestykket giver 0.

Input

Input består af netop én linje, der indeholder et positivt heltal: n.

Output

En linje med hvert regnestykke, der giver 0. Hvis der ikke er nogen sådanne regnestykker skal du skrive teksten `umuligt`.

Du må udskrive regnestykkerne i en vilkårlig rækkefølge.

Eksempler

Input	Output
7	1+2-3+4-5-6+7 1+2-3-4+5+6-7 1-2 3+4+5+6+7 1-2 3-4 5+6 7 1-2+3+4-5+6-7 1-2-3-4-5+6+7

Input	Output
1	umuligt

[1] Denne opgave er taget fra DDD 2015.

Pointgivning

Delopgave 1 (100 point): $1 \leq n \leq 9$

Begrænsninger

Tidsbegrænsning: 1 s.

Hukommelsesbegrænsning: 256 MB.

Gammeldags tastatur[1]

Jens har fundet en gammeldags telefon, der har et sjovt tastatur. Tastaturet har 12 knapper, og på de 8 af dem er der bogstaver. Hver knap har 3 bogstaver, som denne knap kan svare til. Disse bogstaver er som følger:

$$\begin{array}{ll} 2 \to (A,B,C) & 3 \to (D,E,F) \\ 4 \to (G,H,I) & 5 \to (J,K,L) \\ 6 \to (M,N,O) & 7 \to (P,R,S) \\ 8 \to (T,U,V) & 9 \to (W,X,Y) \end{array}$$

(Bemærk, at Q og Z ikke er på nogen af knapperne i denne forsimplede udgave).

Jens har en lang liste over navne i det engelske sprog, og han er interesseret i hvor mange af disse navne der kan fremkomme ved at trykke på knapperne, der passer til hans yndlingstal. Hvis hans yndlingstal f.eks. er 262 kan følgende tekststrenge komme frem:

```
AMA, AMB, AMC, ANA, ANB, ANC, AOA, AOB, AOC, BMA, BMB, BMC, BNA, BNB, BNC,
BOA, BOB, BOC, CMA, CMB, CMC, CNA, CNB, CNC, COA, COB, COC
```

Her kan vi tydeligt se, at både `ANA` og `BOB` svarer til navne.

Opgave

Skriv et program, der læser Jens's yndlingstal n, og udskriver alle de navne der kan være fremkommet ved at taste denne talkombination ind på den telefon han har fundet.

Input

Første linje af input indeholder et positivt heltal: n.

Anden linje af input indeholder et positivt heltal $1 \leq m \leq 5000$.

hver af de efterfølgende m linjer indeholder et navn. Navnene er givet i alfabetisk rækkefølge.

Output

En linje for hvert navn (defineret af listen) der kunne fremkomme. Navnene skal udskrives i sorteret rækkefølge.

Hvis der ikke findes noget navn der passer på tallet skal du udskrive teksten `INGEN`

Eksempler

[1] Opgave kraftigt inspireret af USACO.

Input	Output	Kommentarer
262 2 ANA BOB	ANA BOB	Bemærk, at navnene er i sorteret rækkefølge.

Input	Output
262 1 COC	COC

Input	Output
6235 1 BOB	INGEN

Pointgivning

Delopgave 1 (100 point): n har højest 7 cifre.

Begrænsninger

Tidsbegrænsning: 1 s.

Hukommelsesbegrænsning: 256 MB.

Kapitel 3

Dynamisk programmering I

I Kapitel 2 lærte vi om en utroligt nyttig teknik: *Rekursion*. I dette kapitel skal vi se på hvordan nogle langsomme rekursive programmer kan gøres mere effektive ved brug af en anden generel teknik kaldet: *dynamisk programmering*. Vi vil først demonstrere denne teknik ved at kigge på Fibonacci tallene, og så prøve at bruge den på nogle sværere problemer.

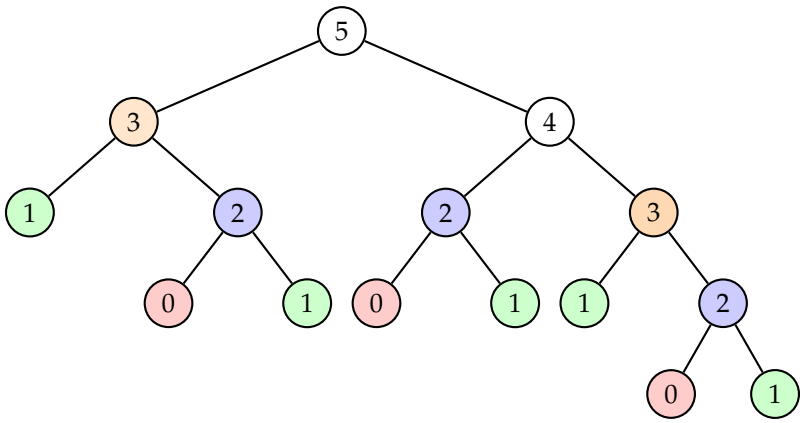
3.1 Fibonacci tal

I Section 2.2 definerede vi Fibonacci tallene og lavede et rekursivt program, der regnede værdien af F_n ud. Hvis vi bruger dette rekursive program til at beregne det 40. Fibonacci tal kommer vi dog til at vente i lang tid. Dette skyldes, at vores rekursive program er meget *ineffektivt* og beregner de samme ting igen og igen. Dette er illustreret i Figur 3.1. Denne figur kaldes et *rekursionstræ* for udførelsen af kaldet `fibonacci(5)`. I hver *knude* i træet er der et tal, der illustrerer hvilket funktionskald denne knude svarer til. Således svarer den aller øverste til kaldet `fibonacci(5)` og de to knuder på det næste niveau til kald med hhv. argument 3 og 4. Knuder der svarer til funktionskald, som beregner det samme er markeret med samme farve. F.eks. kan vi se, at vi beregner Fibonacci tal nummer 1 hele fem gange! Vi kalder dette for *overlappende delproblemer* fordi der er et overlap mellem de delproblemer programmet udregner (at den regner det samme flere gange).

En løsning med dynamisk programmering

Vi så i Figur 3.1 hvordan vores rekursive program fra sidste sektion var alt for langsomt fordi det lavede mange beregninger flere gange. Vi skal nu kigge på hvordan vi kan bruge hukommelse til at undgå dette. Denne teknik kaldes for *dynamisk programmering*.

Idéen er, at vi laver et global tabel, hvor vi kan gemme svarene, så vi blot kan slå op om vi har beregnet værdien før i stedet for at udføre hele funktionskaldet. Vi starter med at fylde denne tabel med en værdi der indikerer at svaret er "ikke beregnet endnu". For Fibonacci tallene kan vi bruge -1 til at



Figur 3.1: Rekursionstræ for at beregne det femte Fibonacci tal. Knuder der er ens har samme farve.

indikere dette, da vi ved at alle Fibonacci tal er ikke-negative. Således starter vi med følgende:

```
1  svar = [-1] * 100
2  svar[0] = 0
3  svar[1] = 1
```

Bemærk, at vi også har gemt værdierne for F_0 og F_1, så vi ikke skal tage højde for dem inde i funktionen. Vi gør nu præcis som før, men slår først op i tabellen om vi allerede kender svaret:

```
4  def fibonacci_dp(n):
5      if svar[n] != -1:
6          return svar[n]
```

Hvis vi ikke kender svaret skal vi beregne det og gemme det i tabellen:

```
7      svar[n] = fibonacci_dp(n-2) + fibonacci_dp(n-1)
8      return svar[n]
```

Og så er vi faktisk færdige! Vi kan prøve at bruge vores nye Fibonacci funktion til at beregne de tal, som vi ikke kunne før og se, at den bliver færdig på ingen tid.

```
>>> fibonacci_dp(30)
832040
>>> fibonacci_dp(40)
102334155
>>> fibonacci_dp(50)
12586269025
>>> fibonacci_dp(80)
23416728348467685
```

Øvelser

Opgave 3.1.1. `[dp_tribo]` Skriv et dynamisk program, der udregner det n'te Tribonacci tal, som defineret i øvelserne til forrige sektion. Sammenlign

køretiderne af de to programmer.

Bottom-up dynamisk programmering

Vores løsning fra sidste sektion tog udgangspunkt i det rekursive program og modificerede det lidt. Denne type dynamisk programmering kaldes for *memoization*. I denne type laver vi vores løsning rekursivt oppe-fra og ned, men husker de forskellige delløsninger, så vi kan genbruge dem. En ulempe ved denne metode er, at vi ofte skal bruge mere hukommelse end vi har behov for. I tilfældet med Fibonacci tallene kan vi ret hurtigt se, at vi jo kun skal bruge de to forrige værdier for at kunne beregne den næste. Vi kan derfor bygge vores svar nede fra og op i stedet:

```
1  def fibonacci_dp2(n):
2      fib2prev = 0
3      fibprev = 1
4      for i in range(1,n):
5          fib = fib2prev + fibprev
6          fib2prev = fibprev
7          fibprev = fib
8      return fibprev
```

I denne løsning har vi ikke nogen global tabel, men i stedet blot to variable der holder styr på de to forrige værdier. På denne måde behøver vi ikke at vide hvor store Fibonacci tal vi har lyst til at udregne på forhånd når vi laver vores `svar` tabel.

Øvelser

Opgave 3.1.2. `[dp_tribo]` Skriv et program, der bruger bottom-up dynamisk programmering til at beregne det n'te Tribonacci tal. Sammenlign det med dine implementationer fra de forrige sektioners opgaver. Hvilket program synes du er bedst og hvorfor?

Opgave 3.1.3. `[dp_evenfibo]` Skriv et program der beregner summen af de n første *lige* Fibonacci tal.

Eksempel:

Input:	Output
1	0
2	2
3	10

3.2 Møntveksling

Med vores nylærte viden om beregning af Fibonacci tal skal vi nu prøve at kaste os ud i et lidt sværere problem. Forestil dig, at du sidder i kassen i netto og skal give byttepenge tilbage og ønsker at bruge så få mønter og sedler som muligt. Vi kender alle algoritmen til at gøre dette optimalt: Først ser vi hvor mange 1000kr sedler vi kan give, dernæst hvor mange 500kr sedler, osv.

Eksempel: Hvis man skal veksle 37kr til mønter vil det optimale være at bruge: En 20'er, en 10'er, en 5'er og en 2'er.

Denne algoritme virker imidlertid kun fordi de danske mønter har de værdier de har. Hvis de forskellige møntværdier f.eks. ikke gik op i hinanden på samme måde ville det se helt anderledes ud.

Eksempel: Hvis man skal veksle 37kr til mønter med så få mønter som muligt ved kun at bruge 1-kroner, 10-kroner, 18-kroner og 20-kroner vil det optimale være at bruge to 18-kroner og en 1-krone. Hvis man til gengæld brugte den *grådige* algoritme fra før ville man ende med en 20-krone, en 10-krone og syv 1-kroner. Altså seks mønter flere!

Vi er nu klar til at definere det problem vi vil kigge på:

Møntvekslingsproblemet

Du har k forskellige typer mønter af værdierne $v_1, v_2, \ldots, v_k$ med $v_1 = 1$. Du skal veksle et beløb på n kroner til mønter ved at bruge så få mønter som muligt.

I eksemplet herover havde vi således $k = 4$, $v_1 = 1$, $v_2 = 10$, $v_3 = 18$, $v_4 = 20$ og $n = 37$.

En rekursiv løsning

Som vi så med Fibonacci tallene er det tit smart at finde en simpel rekursiv løsning på et problem før man prøver at forbedre den, så lad os først prøve at gøre det. Som regel når vi laver rekursion og dynamisk programmering kan vi dele løsningen op i to dele:

- Vi skal træffe et valg – hvilken mønt er den første vi vil bruge i vores veksling.
- Vi skal løse det problem der er tilbage efter valget rekursivt.

For at gøre dette vil vi først definere en funktion som vi kan bruge til at beregne svaret ligesom vi definerede F_i, så vil vi definere M_i til at være *antallet af mønter der minimalt skal bruges til at veksle et beløb på i kroner*. Vi kan tydeligt se, at f.eks. $M_{v_1} = 1$, $M_{v_2} = 1$, osv. da disse beløb kan dækkes af netop én mønt. Ligeledes kan vi se, at $M_0 = 0$. Hvis den optimale måde at veksle i kroner på indeholder en mønt af værdi v_1 så kan vi også se, at $M_i = 1 + M_{v_1}$. På denne måde kan vi faktisk beskrive hele den rekursive M_i på en måde der minder lidt om den vi bruge til Fibonacci tallene:

$$M_0 = 0$$
$$M_i = \min_{\substack{j=1,\ldots,k \\ v_j \leq i}} \left(M_{i-v_j} + 1 \right)$$

Vi kan omsætte denne rekursive definition til et Python program. Først skal vi finde ud af hvilke argumenter vores Python-funktion skal tage. Ligesom M_i er den nødt til at vide, hvad i er, og så lader vi den også tage en liste af de forskellige møntværdier. Dette gør vi således:

```
1  def M(i, v):
2      if i == 0:
3          return 0
```

Vi skal nu finde den bedste af de k muligheder. Dette vil vi gøre ved at prøve hver mulighed rekursivt og sammenligne med den hidtil bedste hver gang:

```
4      best = float('inf')
5      for x in v:
6          if x <= i:
7              best = min(best, M(i - x, v) + 1)
8      return best
```

Denne funktion ligner de funktioner vi har set indtil videre med en undtagelse:

> **Syntax:** I Python repræsenterer tallet `float('inf')` uendeligt som et decimaltal. Python tillader direkte sammenligning mellem decimaltal og heltal. Dette bruger vi når vi kalder funktionen `min` i ovenstående funktion.

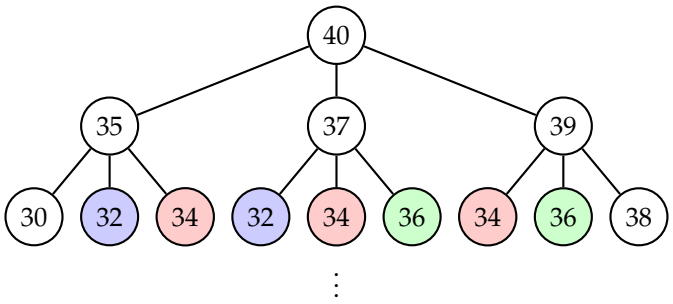
Lad os først prøve at se om der overhovedet er nogen grund til at optimere vores program, eller om det faktisk allerede er hurtigt nok. Herunder ses udprintet fra en kørsel af programmet på nogle små problemer.

```
>>> v = [1,3,5]
>>> M(8, v)
2
>>> M(11,v)
3
>>> M(16,v)
4
>>> M(20,v)
4
```

Her prøver vi at veksle hhv. 8, 11, 16 og 20 kroner med mønter af værdierne 1, 3 eller 5. Vi kan også prøve at veksle 40 kroner med de samme mønter, men så bruger Python pludseligt meget længere tid. Noget tyder på, at der er behov for at tænke sig lidt om.

Ineffektivitet af det rekursive program

Lad os kigge lidt nærmere på eksekveringen af vores program fra forrige sektion. I eksempeleksekveringen havde vi mønter af værdierne $1, 3, 5$ og vi prøvede at veksle 40 kroner. For at veksle 40 kroner skal vi således veksle 39, 37 og 35 kroner rekursivt. Vi kan illustrere det på samme måde som vi gjorde for Fibonacci tallene i Figur 3.1. Dette er gjort i Figur 3.2 herunder. I figuren er funktionskald der bliver beregnet flere gange markeret med samme farve. Således kan vi se, at vi hele tre gange prøver at beregne den bedste måde at veksle 34 kroner alene inden for de første tre niveauer i rekursionstræet.



Figur 3.2: Rekursionstræ for programeksekveringen af `M(40, [1,3,5])`.

Som vi kan se på figuren bliver mange af funktionskaldene eksekveret flere gange, og dette vil være endnu mere tydeligt, hvis vi kigger på næste niveau af rekursionstræet. Det er altså tydeligt, at problemet har overlappende delproblemer. Bemærk at der faktisk kun er 40 egentlige delproblemer i træet i Figur 3.2, så noget kunne tyde på, at vi beregner virkeligt mange af de samme delproblemer mange gange!

En løsning med dynamisk programmering

Vi skal nu se hvordan vi kan bruge de samme idéer fra vores udregning af Fibonacci tal til at udregne den optimale møntveksling. Ligesom i Section 3.1 skal vi bruge en tabel, som vi kan gemme vores svar i. Vi vil igen bruge værdien `-1` til at indikere, at en værdi ikke er udregnet endnu.

```
1  svar = [-1] * 10000
2  svar[0] = 0
```

Nu er der blot tilbage at oversætte det program til at bruge tabellen:

```
3  def M(i, v):
4      if svar[i] == -1:
5          svar[i] = float('inf')
6          for x in v:
7              if x <= i:
8                  svar[i] = min(svar[i], M(i - x, v) + 1)
9      return svar[i]
```

Som det kan ses herover kan vi faktisk genbruge næsten al koden fra vores rekursive løsning. Den eneste forskel er, at vi nu gemmer resultatet i `svar`

tabellen. Med dette program kan vi nu veksle langt større beløb end før. Dette kan ses herunder:

```
>>> M(40,[1,3,5])
8
>>> M(232,[1,3,5])
48
>>> M(1324,[1,3,5])
266
>>> M(5786,[1,3,5])
Traceback (most recent call last):
...
RuntimeError: maximum recursion depth exceeded
    while calling a Python object
```

Her kan vi se, at 1324 kroner kan veksles med 266 mønter (af slagsen 1-kroner, 3-kroner og 5-kroner). På sidste linje kan vi dog se, at Python kommer i problemer. Dette skyldes, at Python ikke kan håndtere så mange rekursive funktionskald med standardindstillingerne. Dette svarer til, at vi er kommet for mange niveauer ned i træet fra Figur 3.2 til at Python kan holde styr på det længere.

Vi kan komme dette problem til livs på to måder: Den ene er at fortælle Python, at den skal øge begrænsningen på rekursiondybden[1]. Den anden er at bruge bottom-up dynamisk programmering ligesom vi gjorde det i Fibonacci eksemplet. For at regne Fibonacci tal F_i var vi kun nødsaget til at gemme de to forrige, F_{i-1} og F_{i-2}. Da formlen for at beregne M_i er mere kompleks kan vi risikere, at det er nødvendigt at gemme alle $M_0, M_1, \ldots, M_i$, så vi laver en tabel til det hele. Når vi laver bottom-up dynamisk programmering behøver vi ikke at initialisere vores `svar` tabel til at have en særlig værdi, da vi ved, at vi altid kun kigger på tabelindgange, som allerede er regnet ud. En bottom-up version af møntvekslingsalgoritmen kan ses i koden herunder:

```
1  def M(n,v):
2      svar = [0]* (n+1)
3      for i in range(1,n+1):
4          svar[i] = float('inf')
5          for x in v:
6              if x <= i:
7                  svar[i] = min(svar[i], svar[i-x] + 1)
8      return svar[n]
```

Med dette program kan vi veksle flere millioner kroner, hvilket er en væsentlig forbedring over vores første program, der ikke var i stand til at veksle 40:

```
>>> M(5786,[1,3,5])
1158
>>> M(57486,[1,3,5])
11498
>>> M(574386,[1,3,5])
```

[1]Dette gøres med `sys` modulet.

```
114878
>>> M(1574386,[1,3,5])
314878
```

Dette illustrerer perfekt hvor kraftfuldt dynamisk programmering kan være til at løse visse problemer, hvor en simpel rekursiv løsning beregner de samme ting alt for mange gange..

Øvelser

Opgave 3.2.1. `[dp_coins2]` Lav et modificeret dynamisk program til møntveksling hvor alle ulige mønter tæller for 2 – altså det "koster" 2 mønter at bruge en mønt af ulige værdi. Så hvis man vil veksle 5 kroner med mønterne $1, 2, 3$ kan man gøre det med tre mønter (en 2'er og en 3'er, hvor den sidste tæller for to) eller fire mønter (to 2'ere og en 1'er, hvor den sidste tæller for to).

Første linje af input består af møntværdierne adskilt af mellemrum. Anden linje består af beløbet der skal veksles:

Input	Output
1 2 3 6	3

Forklaring: Selvom to 3-mønter giver 6 vil det "koste" 4 mønter at gøre det, så det er smartere af tage tre 2-mønter.

At genskabe en løsning

Indtil videre har vi kun kigget på hvordan man beregner hvor mange mønter der skal bruges, men i praksis vil vi også gerne vide hvilke mønter det er – der er trods alt ikke noget ved at vide, hvor godt noget kan gøres uden at kunne gøre det efter.

Tidligere kiggede vi på hvordan man kunne finde en optimal løsning. Denne fandt vi ved, at prøve alle muligheder for hvilken mønt vi skulle starte med og vælge den der gav det færreste antal i alt ved at kigge på $M(i - v_1), \ldots, M(i - v_k)$. Hvis vi ønsker at kunne genskabe løsningen kan vi altså gemme dette *optimale valg* for enhver værdi i og bruge det til at genskabe løsningen!

For at kunne gøre dette skal vi bruge endnu en tabel til at gemme det optimale valg for hver i-værdi. Lad os kalde denne for `valg`.

```
1  def M(n,v):
2      svar = [0] * (n+1)
3      valg = [0] * (n+1)
4      for i in range(1,n+1):
5          svar[i] = float('inf')
6          for x in v:
```

Som det kan ses gør vi helt ligesom før bortset fra, at vi laver en ekstra tabel. Når vi finder en mønt, der giver en bedre løsning skal vi huske at opdatere `valg` tabellen:

```
7               if x <= i and svar[i-x] + 1 < svar[i]:
8                   svar[i] = svar[i-x] + 1
9                   valg[i] = x
```

Nu mangler vi blot at genskabe den faktiske løsning. Vi vil lave en liste med alle de mønter der skal bruges. Vi kan lave denne ved først at finde det bedste valg for at veksle n kroner. Hvis dette valg var at bruge en mønt til værdi x finder vi nu det bedste valg for at veksle $n - x$ kroner, osv.

```
10      loesning = []
11      while n > 0:
12          loesning.append(valg[n])
13          n = n - valg[n]
14      return loesning
```

Dette program kan nu give os en liste af de mønter vi skal bruge:

```
>>> M(7,[1,3,5])
[1, 1, 5]
>>> M(8,[1,3,5])
[3, 5]
>>> M(9,[1,3,5])
[1, 3, 5]
>>> M(10,[1,3,5])
[5, 5]
>>> M(11,[1,3,5])
[1, 5, 5]
>>> M(12,[1,3,5])
[1, 1, 5, 5]
```

Den teknik vi har brugt her er en meget generel en der kan bruges i næsten alle dynamiske programmer til at genskabe en optimal løsning.

Øvelser

Opgave 3.2.2. `[dp_coins3]` Omskriv programmet fra denne sektion, så det returnerer hvor mange af hver slags mønt der skal bruges i stedet for en liste af alle mønterne.

Første linje af input består af møntværdierne adskilt af mellemrum. Anden linje består af beløbet der skal veksles. Output skal bestå af en linje, som er antallet af hver mønt i en optimal løsning:

Input	Output
1 2 3 7	1 0 2

Opgave 3.2.3. Omskriv programmet fra denne sektion, så det returnerer to løsninger hvis der er mere end én optimal måde at veksle de n kroner.

3.3 Review

I dette kapitel er vi blevet introduceret til dynamisk programmering, der er en måde at tage en ellers langsom rekursiv løsning og gøre den effektiv ved at

gemme mellemregninger af delproblemer. Vi har set hvordan denne metode kan bruges til at beregne Fibonacci tal og møntveksling hurtigt. Vi har også set hvordan man kan gemme de valg ens algoritme har taget for at kunne genskabe en optimal løsning.

Dynamisk programmering kan bruges, hvis et problem har *optimal delstruktur* og *overlappende delproblemer*, og det er oftest smartest at beregne løsningen fra bunden og starte med de mindste delproblemer som vi gjorde i slutningen af sidste sektion.

3.4 Problemer

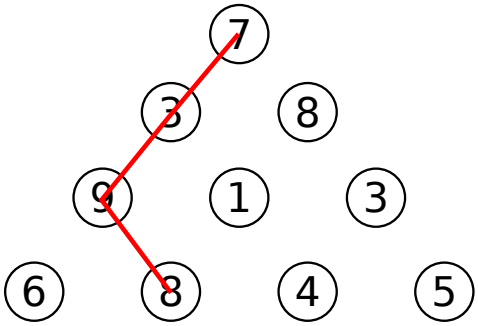
Disse problemer kan findes i folderen med problemer. Deres navn er det samme som i listen af problemer herunder.

Problem 3.1. *Triangle*

Problem 3.2. *Rodcut*

Triangle

Du er givet en trekant af tal, og starter helt i toppen. I hvert skridt kan du enten gå ned til venstre eller ned til højre. Hvad er den maksimale sum der kan opnås, af de tal du besøger? Problemet er illustreret i figuren herunder.



Opgave

Skriv et program, der læser trekanten af tal ind og og udskriver den største mulige sum af tal man kan opnå på turen ned igennem trekanten.

Input

Den første linje af input består af et positivt heltal, n, der angiver højden af trekanten.

De næste n linjer beskriver hver en linje af trekanten. Den første linje indeholder ét tal: toppen af trekanten. Den næste linje indeholder to tal: Den anden linje af trekanten i rækkefølge fra venstre til højre, osv. Alle tal vil være mellem 0 og 1000 inklusive.

Output

Output skal bestå at et enkelt heltal, den størst mulige sum der kan opnås på en tur ned igennem trekanten, hvor man kun må gå ned til højre eller ned til venstre.

Eksempler

Input	Output	Kommentarer
4 7 3 8 9 1 3 6 8 4 5	27	Dette er eksemplet fra figuren.

Pointgivning

Delopgave 1 (30 point): $1 \leq n \leq 20$

Delopgave 2 (70 point): $1 \leq n \leq 1000$

Begrænsninger

Tidsbegrænsning: 1 s.

Hukommelsesbegrænsning: 256 MB.

Rodcut

Sigurd er gået ind i træhandel. Han har et stykke træ af længde n, og vil skære det i stykker så han kan sælge det for mest muligt.

Sigurd har fået en liste med priser for stænger af længde $1, 2, \ldots, n$. Hvis han f.eks. har en stang af længde 5 og følgende priser:

Længde	1	2	3	4	5
Pris	2	5	6	8	11

Så er det optimale, at skære stangen ud i to af længde 2 og en af længde 1. Det vil give en samlet pris på $5 + 5 + 2 = 12$.

Opgave

Du er givet stangens længde, n, samt alle priserne. Beregn hvad det maksimale antal penge Sigurd kan få for sin stang er.

Input

Første linje af input består af et positivt heltal n.

Den næste linje består af n positive heltal, som er prisen for en stang af længde $1, 2, \ldots, n$ i den rækkefølge.

Output

Det største antal penge Sigurd kan få for sin stang hvis han skærer den over smartest.

Eksempler

Input	**Output**	**Kommentarer**
`5` `2 5 6 8 11`	`12`	Dette er eksemplet fra opgaveteksten.

Pointgivning

Delopgave 1 (30 point): $1 \leq n \leq 20$

Delopgave 2 (70 point): $1 \leq n \leq 1000$

Begrænsninger

Tidsbegrænsning: 1 s.

Hukommelsesbegrænsning: 256 MB.

Kapitel 4

Dynamisk programmering II

I sidste kapitel så vi hvordan man kunne bruge dynamisk programmering til at løse forholdsvist simpmle dynamiske programmer. I dette kapitel skal vi se hvordan man kan bruge de samme idéer til at tackle en række af mere komplicerede problemer.

4.1 Dynamisk programmering med begrænsninger

Vi har indtil nu primært kigget på spørgsmål med ét argument: "Hvad er det n'te Fibonacci tal" og "Hvordan veksler jeg optimalt n kroner"[1], men hvad nu hvis der er flere parametre der spiller ind? Det vil vi kigge på med følgende problem:

Kassebåndsproblemet

Du har n forskellige varer på et kassebånd, der koster hhv. $p_1, p_2, \ldots, p_n$, som er heltal i øre. Når du køber varerne skal du betale den nærmeste hundrede i øre (50 rundes op). Du har k skillere du må sætte på båndet, så du kan foretage op til $k+1$ forskellige køb. Hvordan skal du sætte skillerne for at betale så lidt som muligt?

Dette problem er illustreret i følgende eksempel:

Eksempel: Hvis $n = 5$, $k = 1$ og vi har

$$p_1 = 130, p_2 = 215, p_3 = 470, p_4 = 200, p_5 = 340$$

så gælder det, at $p_1 + p_2 + p_3 + p_4 + p_5 = 1355$, hvilket bliver rundet op til 1400 øre i alt, hvis vi ikke indsætter en skiller. Indsætter vi derimod en skiller mellem vare nummer to og tre får vi to indkøb af

[1] Hvis vi ser bort fra at vi havde en liste med mønter.

hhv. $p_1 + p_2 = 345$ og $p_3 + p_4 + p_5 = 1010$. Begge disse indkøb bliver rundet ned, og vi kan nøjes med at betale 1300 øre.

At finde den bedste pris rekursivt

Ligesom i forrige kapitel vil vi starte med at finde en løsning rekursivt, og vi vil først beskæftige os med at finde prisen for den optimale løsning.

Lad os kigge på hvad den optimale pris er for at købe de første i varer med j skillere. Dette kan vi gøre ved at placere den j'te skiller et sted, og løse problemet for alle de forrige varer med $j-1$ skillere. Opgaven handler nu om to ting:

1. At finde det bedste sted at sætte den j'te skiller.
2. At finde løsningen på det mindre problem med $j-1$ skillere.

Hvis vi kalder den optimale pris for at købe de i første varer med j skillere for $P(i,j)$ ser vi altså nu, at vi kan opskrive følgende rekursive formel:

$$P(i,0) = \texttt{Round}(p_1 + \ldots + p_i) \tag{4.1}$$
$$P(i,j) = \min_{\ell<i} \left(P(\ell, j-1) + \texttt{Round}(p_{\ell+1} + \ldots + p_i)\right). \tag{4.2}$$

Hvor `Round` funktionen afrunder et tal til nærmeste hundrede. Vi vil starte med at implementere denne funktion i Python:

```
1  def round100(n):
2      if n % 100 < 50:
3          return (n/100) * 100
4      else:
5          return (n/100 + 1) * 100
```

Denne funktion gør brug af nogle tricks ved heltal, som er indbygget i Pythons standardopførsel.

Syntax (Genopfriskning): I Python betyder `%` operatoren *rest ved division*. Således beregner `n % 100` hvad de to sidste cifre af `n` er.

Når man arbejder med division og heltal i Python afrunder Python automatisk nedad. Således giver `599 / 100 = 5`.

Vi er nu klar til at implementere vores rekursive funktion i Python. Først sørger vi for, at den giver det rigtige svar, hvis der er 0 skillere tilbage:

```
1  def P(i,j, p):
2      if j == 0:
3          return round100(sum(p[:i]))
```

Bemærk, at denne kode bruger Pythons indbyggede `sum` funktion, der beregner summen af en liste. Vi mangler nu blot de sidste

```
4       best = float('inf')
5       for l in range(1,i):
6           best = min(best, P(l, j-1, p) + round100(sum(p[l:i])))
7       return best
```

Bemærk, at dette lille stykke kode er meget omhyggelig med at undgå såkaldte *off-by-one* fejl. Vi kan prøve at køre vores funktion på eksemplet fra tidligere:

```
>>> P(5,0,[130,215,470,200,340])
1400
>>> P(5,1,[130,215,470,200,340])
1300
```

Men hvis vi prøver at give dette program en liste med blot 30 varer går det galt. Vores program er alt for langsomt, og det er helt tydeligt, at vi er nødt til at optimere på programmet!

Øvelser

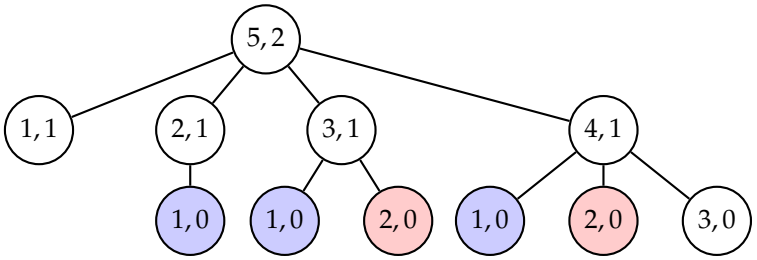
Opgave 4.1.1. Omskriv det rekursive program herover, så det afrunder til nærmeste 25 i stedet for nærmeste 100.

Optimeringer og dynamisk programmering

Der er to store problemer med vores løsning rent tidsmæssigt:

1. Der er mange overlappende delproblemer.
2. Vi bruger meget lang tid på at beregne summer.

Det kan være lidt sværere at se i dette problem, at der skulle være mange overlappende delproblemer, men hvis vi kigger nærmere er det ganske tydeligt. I Figur 4.1 er tegnet et rekursionstræ over et problem med $n = 5$ varer og $k = 2$ skillere. Selv i dette lille eksempel ser vi, at vi prøver at beregne $P(1,0)$ og $P(2,0)$ mange gange, og jo større k bliver jo flere overlappende delproblemer vil vi se. Vi ved dog allerede hvordan vi kan løse dette problem.



Figur 4.1: Rekursionstræ for kasseproblemet med $n = 5$ og $k = 2$.

Nemlig med dynamisk programmering! Vi skal blot have en tabel vi kan gemme svaret i.

Lad os dog først kigge på vores andet problem: Hver eneste gang, der står `sum` i vores kode kigger Python hele den givne liste igennem og lægger

priserne sammen. Vi bliver nødt til at udnytte, at hvis vi allerede har udregnet summen af de første i priser, så er det nemt at udregne summen af de første $i+1$ priser, ved blot at lægge p_{i+1} til. Vi gør derfor klogt i først at beregne en tabel med alle summerne af de i første priser:

```
1  def P(n, k, p):
2      sums = [0] * (n+1)
3      for i in range(1,n+1):
4          sums[i] = sums[i-1] + p[i-1]
```

Bemærk her, at `p[0]` er prisen på den første vare, men vi sætter `sums[i]` til at være prisen på de første i varer (dvs. `sums[1] = p[0]` og `sums[2] = p[0] + p[1]`). Vi er nu klar til at lave vores bottom-up dynamisk programmering. Lad os først lave en tabel til at gemme alle vores resultater, og gemme resultaterne med 0 skillere, da vi blot kan aflæse disse direkte fra `sums` tabellen.

```
5      svar = [[0 for x in range(k+1)] for x in range(n+1)]
6      for i in range(n+1):
7          svar[i][0] = round100(sums[i])
```

Bemærk at vi nu gemmer vores svar i en to-dimensionel tabel, og at syntaxen for at lave sådan en er lidt besværlig. Lad os nu fylde resten af tabellen ud ved at bruge vores rekursive formel. Vi beregner først alle priserne med 1 skiller, så 2 skillere, osv. op til k skillere:

```
8      for j in range(1,k+1):
9          for i in range(1,n+1):
10             svar[i][j] = float('inf')
11             for l in range(1,i):
12                 svar[i][j] = min(svar[i][j], svar[l][j-1] + round100
                       (sums[i] -
13                 sums[l]))
14     return svar[n][k]
```

Som det kan ses har vi ændret meget lidt i forhold til vores rekursive program. En vigtig ting vi har ændret er, at vi bruger vores `sums` array i vores kald til `Round100` funktionen. Her beregner vi summe $p_{\ell+1} + \ldots + p_i$ ved at tage

$$p_1 + \ldots + p_i - (p_1 + \ldots + p_\ell) .$$

Dette er en generel teknik, der er kendt under navnet *inklusion-eksklusion*, da vi kan se det som først at *inkludere* alle priserne fra p_1 til p_i, og dernæst ekskludere dem fra p_1 til p_ℓ. Med denne nye kode kan vi nemt beregne større instanser. Herunder kan ses en udskrift af at køre koden på en instans med $n = 540$ varer og $k = 100$ skillere:

```
>>> P(540, 5, range(50,20000,37))
5411400
>>> P(540, 10, range(50,20000,37))
5411100
>>> P(540, 100, range(50,20000,37))
5408100
```

Øvelser

Opgave 4.1.2. Omskriv koden fra denne sektion, så alle køb man laver under 4000 ører er gratis.

Opgave 4.1.3. Omskriv programmet så det ikke bruger inklusion-eksklusion til at beregne afrundingen i det sidste for-loop. Hvor meget langsommere bliver programmet af dette? Er det i stand til at beregne eksemplet med de 540 varer og 100 skillere?

Opgave 4.1.4. Omskriv programmet, så det udskriver hvor man skal sætte skillerne i stedet for hvad den optimale pris er. Du kan bruge de samme teknikker som vi brugte til møntvekslingsproblemet.

4.2 Tælleopgaver

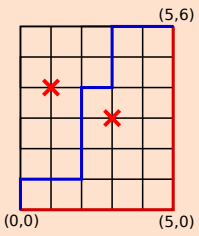
Vi har indtil videre primært kigget på problemer, hvor man skulle beregne en bestemt værdi – den mindste pris, det færreste antal, eller lignende. Alle problemer havde det til fælles, at der var mange overlappende delproblemer. Det er ikke altid helt nemt at indse, at et problem har mange overlappende delproblemer. En egenskab der tit fortæller os det er dog, hvis problemet handler om at tælle til enormt store tal. Dette er tit opgaver der starter med teksten "På hvor mange måder kan man ...". En sådan opgave skal vi kigge på i denne sektion:

Trafikvejsproblemet

Du befinder dig i en by, der er udformet som et $n \times m$ gitter, hvor hver kant i gitteret er en vej, og hvert kryds i gitteret er et lyskryds. Nogle lyskryds i stykker, og kan ikke krydses. Der er i alt k af disse, og de har koordinaterne $(x_1, y_1), \ldots, (x_k, y_k)$. På hvor mange måder kan du komme fra krydset i $(0,0)$ til krydset i (n,m) ved at bruge højest $n+m$ veje.

I dette problem kan vi risikere at svaret bliver enormt stort, og det er derfor helt umuligt at prøve at finde samtlige veje.

Eksempel: På figuren til højre ses en instans af trafikvejsproblemet med $n = 5$, $m = 6$ og $k = 2$, hvor $(x_1, y_1) = (1,4)$ og $(x_2, y_2) = (3,3)$. På figuren er også markeret to eksempler på veje fra $(0,0)$ til $(5,6)$ af længde 11. Der er i eksemplet mange flere veje af samme længde.



Lige som tidligere vil vi prøve at beskrive en rekursiv løsning først og observere at den er alt for langsom. Lad os kigge på følgende funktion:

$N(x, y, \ell) =$ Antal forskellige veje fra (i, j) til (n, m) af længde højest ℓ.

Vi kan hurtigt se, at følgende tilfælde må gælde:

$$N(x,y,\ell) = \begin{cases} 0 & \text{hvis } (x,y) = (x_1,y_1) \\ \vdots & \\ 0 & \text{hvis } (x,y) = (x_k,y_k) \\ 0 & \text{hvis } (x,y) \neq (n,m) \text{ og } \ell = 0 \\ 1 & \text{hvis } (x,y) = (n,m) \end{cases} \tag{4.3}$$

Altså, hvis vi står i et kryds der er blokeret kan vi ikke komme videre, og der er altså 0 muligheder. Hvis $\ell = 0$ og vi ikke er i mål kan vi heller ikke komme videre, og hvis $(x,y) = (n,m)$ er vi allerede i mål, og der er altså 1 måde at komme i mål på[2]. Vi mangler derfor blot at håndtere tilfældet hvor $\ell > 0$ og $(x,y) \neq (n,m)$. I dette tilfælde kan vi enten gå op, ned, til højre eller til venstre og reducere ℓ med 1. Altså kan vi tilføje følgende case:

$$\begin{aligned} N(x,y,\ell) = {} & N(x,y-1,\ell-1) + N(x,y+1,\ell-1) + N(x-1,y,\ell-1) \\ & + N(x+1,y,\ell-1) \quad \text{hvis de andre tilfælde ikke gælder.} \end{aligned} \tag{4.4}$$

Bemærk at vi har overset noget i denne formel, som du bliver bedt om at finde i øvelserne.

Vi kan implementere denne rekursive formel i Python således. Bemærk, at vi giver parametrene n, m samt listerne x og y med som argument til funktionen.

```
1  def N(i,j,l,n,m,x,y):
2      k = len(x)
3      for t in range(k):
4          if (i,j) == (x[t],y[t]):
5              return 0
6      if (i,j) == (n,m):
7          return 1
8      if l == 0:
9          return 0
10     return N(i-1,j,l-1,n,m,x,y) + N(i+1,j,l-1,n,m,x,y) + N(i,j-1,l
           -1,n,m,x,y) + N(i,j+1,l-1,n,m,x,y)
```

Med dette program kan vi teste hvor mange mulige veje der er i vores eksempel, men selv det lille eksempel tager forholdsvist lang tid at udregne:

```
>>> N(0,0,11,5,6,[1,3],[4,3])
187
```

Øvelser

Opgave 4.2.1. Hvad er det der er blevet overset i den rekursive formel der er udgjort af ligningerne (4.3) og (4.4)? Er det i virkeligheden en fejl eller betyder det ikke noget for resultatet?

[2]Her antager vi, at man ikke må gå ud og ind igen, hvis $\ell \geq 2$.

En smartere rekursiv formel og repræsentation

Den kvikke læser har nok allerede bemærket, at kravet om at vejen har længde højest $n+m$ er en stor begrænsning, som kan hjælpe os meget med at forsimple vores formel. Vi kan nemlig se, at det kun kan betale sig at enten køre op eller til højre, for lige så snart man er kørt enten ned eller til venstre er det umuligt at nå i mål uden at have kørt for langt. Vi kan derfor erstatte ligning (4.4) med følgende:

$$N(x,y,\ell) = N(x+1,y,\ell-1) + N(x,y+1,\ell-1) \tag{4.5}$$

Hvis vi ændrer den sidste linje af vores program herover til at reflektere denne ændring kan vi allerede se at vores program kører *meget* hurtigere. Vi kan også tilføje et ekstra tilfælde, så vores kode aldrig kører uden for gitteret:

$$N(x,y,\ell) = 0 \quad \text{hvis } x<0,\, y<0,\, x>n \text{ eller } y>m\,. \tag{4.6}$$

Dette vil have stor betydning når vi skal bruge dynamisk programmering, så vi ikke kommer til at læse et sted i vores tabel, som ikke eksisterer.

Et andet sted vores program er meget ineffektivt er i måden hvorpå den tjekker, om et kryds er i stykker: Hver gang vi når til et kryds tjekker vi altså hele listen af ødelagte kryds igennem for at se om dette var et af dem. Dette er en meget ineffektiv repræsentation af de ødelagt kryds, som vi kan optimere ved at bruge mere plads. I stedet kan vi godt tillade os at bruge en tabel, hvor vi kun behøver at kigge ét sted for at se om vores kryds er i stykker. Dette kan vi f.eks. gøre med en todimensionel tabel ligesom vi gjorde, da vi lavede dynamisk programmering med varerne på kassebåndet.

```
1  oedelagt_kryds = [[0 for x in range(m+1)] for x in range(n+1)]
2
3  def udfyld_kryds(x,y):
4      k = len(x)
5      for t in range(k):
6          oedelagt_kryds[x[t]][y[t]] = 1
```

Øvelser

Opgave 4.2.2. Modificér programmet fra forrige delsektion til at benytte vores formel fra ligning (4.5). Prøv at tage tid på hvor stor forskellen er på at køre de to programmer på nogle eksempler.

Opgave 4.2.3. Tilføj tilfældet fra ligning (4.6). Gjorde dette en stor forskel?

Opgave 4.2.4. Omskriv programmet, så det bruger en tabel til at tjekke om et kryds er ødelagt i stedet for at kigge hele listen igennem. Du kan evt. bruge funktionen `udfyld_kryds` herover.

Prøv at konstruere et eksempel hvor det gamle program var for langsomt, men det nye program er hurtigt nok. Du bliver nødt til at have mange ødelagte krydser i dit eksempel for at dette er tilfældet.

Dynamisk programmering og smartere formulering

Vi er nu klar til at bruge dynamisk programmering til at løse problemet. Spørgsmålet er bare hvor stor en tabel vi skal bruge til at gemme vores svar. Man kunne godt tro, at vi var nødt til at have en tre-dimensionel tabel, som kunne gemme svaret for $N(x, y, \ell)$, da denne funktion jo tager tre parametre. Hvis vi tænker os lidt om kan vi dog hurtigt indse, at vi ikke kan komme til krydset i (x, y) uden at bruge mindst $x + y$ træk. Samtidig kan vi ikke komme fra (x, y) til (n, m) inden for tiden, hvis vi ikke har brugt højest $x + y$ træk på at komme til (x, y). Altså er der kun en ℓ-værdi, der kan bruges for hvert kryds (x, y).

Lad os prøve at sætte alt dette sammen til et fuldt dynamisk program. Vi starter med at lave en tabel over oedelagt kryds:

```
1  def N(n,m,x,y):
2      k = len(x)
3      oedelagt_kryds = [[0 for _ in range(m+1)] for _ in range(n+1)]
4      for t in range(k):
5          oedelagt_kryds[x[t]][y[t]] = 1
```

Dernæst laver vi en tabel til at gemme værdierne i. Vi gemmer også, at der er 1 måde at nå i mål, hvis vi starter i målet.

```
6      svar = [[0 for _ in range(m+1)] for _ in range(n+1)]
7      svar[n][m] = 1
```

Da vi ikke vil læse udover "kanten" i vores tabel opdatere vi den øverste række og den yderst-til-højre kolonne for sig selv:

```
8      for i in range(n-1,-1,-1):
9          if oedelagt_kryds[i][m]:
10             break
11         else:
12             svar[i][m] = svar[i+1][m]
13     for j in range(m-1,-1,-1):
14         if oedelagt_kryds[n][j]:
15             break
16         else:
17             svar[n][j] = svar[n][j+1]
```

Altså, da der kun er én vej vi kan gå i denne række og kolonne kopierer vi blot svaret for denne vej over. Hvis vi møder et kryds på vejen stopper vi løkken, da alle resterende svar vil være 0 (fordi man er nødt til at gå igennem det ødelagte kryds). Bemærk, at vores løkke kører oppe fra og ned (med den lidt komplicerede syntax `range(n-1,-1,-1)`) da vi regner fra målet og baglæns.

Vi kan nu fylde resten af pladserne ud:

```
18     for i in range(n-1,-1,-1):
19         for j in range(m-1,-1,-1):
20             if oedelagt_kryds[i][j]:
21                 svar[i][j] = 0
22             else:
23                 svar[i][j] = svar[i+1][j] + svar[i][j+1]
24     return svar[0][0]
```

Vi har nu lavet et dynamisk program, der er meget hurtigere end det oprindelige rekursive program, og kan beregne meget store tal. Vi kan prøve at teste vores program på det oprindelige eksempel samt et meget større gitter af veje:

```
>>> N(5,6,[1,3],[4,3])
187
>>> N(100,100,[],[])
90548514656103281165404177077484163874504589675413336841320L
>>> N(500,500,[],[])
27028824094543656951561469362597527549615200844654828700739...
```

Øvelser

Opgave 4.2.5. `[dp_trafik1]` Omskriv programmet, så man i alle kryds (x, y), hvor $x + y$ er delelig med 5, *skal* gå opad.

Input har følgende format: Første linje indeholder n, anden linje indeholder m, tredje linje indeholder k. De næste k linjer indeholder de kryds der er i stykker. Hver linje indeholder x_i, y_i.

Input	Output
8 7 2 5 5 6 7	173

Opgave 4.2.6. `[dp_trafik2]` Omskriv programmet fra denne sektion, så der kan være genveje. En genvej er en ekstra vej fra (x_s, y_s) til (x_e, y_e), hvilket betyder, at man fra (x_s, y_s) udover at kunne gå op og til højre også kan gå til (x_e, y_e). Det gælder altid, at $x_e \geq x_s$ og $y_e \geq y_s$.

Input er det samme som før bortset fra, at der efter den sidste linje er en ekstra linje der indeholder tallet t, som er antallet af genveje. De næste t linjer indeholder fire tal x_s, y_s, x_e, y_e, der beskriver en genvej.

Input	Output
5 4 1 3 3 2 1 1 4 4 0 2 2 2	72

4.3 Review

I dette kapitel har vi set hvordan dynamisk programmering kan bruges til mere komplicerede problemer end at udregne fibonacci tal og tælle mønter. Vi har også set hvordan man med helt samme fremgangsmåde kan tackle disse mere komplicerede problemet: Formuler først problemet rekursivt, lav en naiv løsning, og udbyg denne med en tabel, der gemmer svarene.

4.4 Problemer

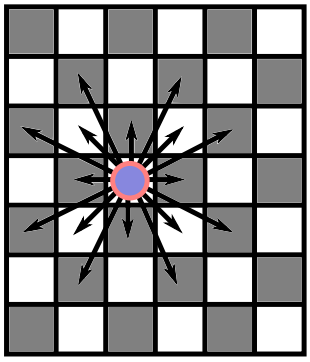
Disse problemer kan findes i folderen med problemer. Deres navn er det samme som i listen af problemer herunder.

Problem 4.1. *Robot*

Problem 4.2. *Super springer*

Problem 4.3. *Palindrom*

Super-springer



Sigurd og hans venner spiller en mærkelig version af skak. De har et stort bræt af størrelse $n \times m$, og en ekstra brik, som de kalder for en *super-springer*. Denne brik kan bevæge sig som en blanding af en springer og en konge (se billedet til højre).

Sigurd har helt styr på hvordan springeren bevæger sig i normal skak, men vil gerne have bedre styr på hvordan super-springeren kan bevæge sig. Din opgave er at hjælpe ham med dette. Mere præcist skal du beregne på hvor mange forskellige måder super-springeren kan komme fra en position (x_s, y_s) til en position (x_m, y_m) ved at bruge præcist k træk.

Opgave

Du er givet brættets størrelse $n \times m$, springerens position (x_s, y_s) og målets position (x_m, y_m). Givet et positivt heltal k skal du udregne hvor mange forskellige veje der er for super-springeren for at nå målet på præcist k træk.

Input

Input består af præcis en linje med 7 tal: n, m, x_s, y_s, x_m, y_m, og k i den rækkefølge. (position $(1, 1)$ er nederste venstre hjørne og (n, m) er øverste højre hjørne.

Det gælder altid, at $1 \leq x_s, x_m \leq n$ og $1 \leq y_s, y_m \leq m$.

Output

Antallet af forskellige veje der er for super-springeren fra (x_s, y_s) til (x_m, y_m) med præcist k træk.

Eksempler

Input	Output	Kommentarer
`5 6 3 4 5 4 1`	`0`	Det er ikke muligt for springeren at nå det givne felt på færre end 2 træk.

Input	Output	Kommentarer
`5 6 3 4 5 4 2`	`9`	Se figuren.

Pointgivning

Delopgave 1 (30 point): $1 \leq n, m \leq 16$ og $1 \leq k \leq 3$

Delopgave 2 (70 point): $1 \leq n, m \leq 16$ og $1 \leq k \leq 60$

Begrænsninger

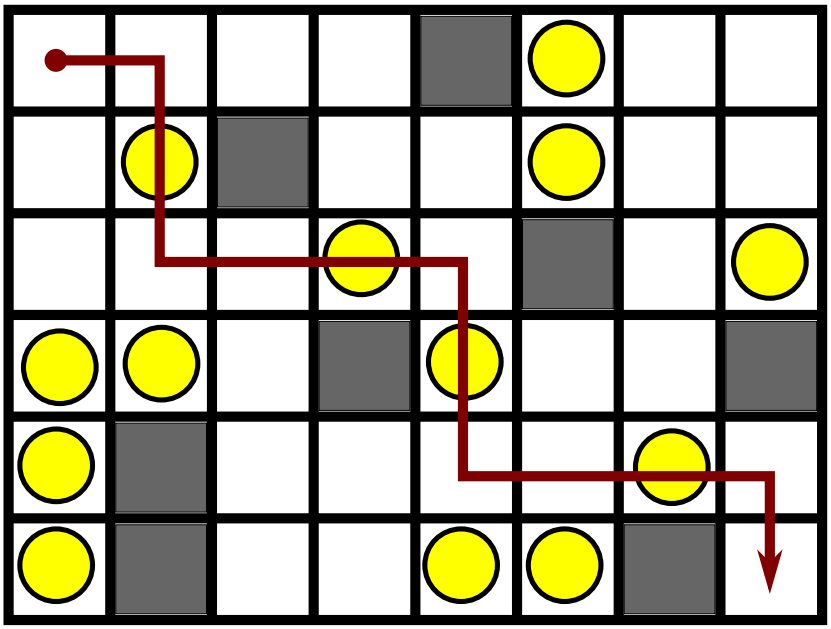
Tidsbegrænsning: 1 s.

Hukommelsesbegrænsning: 256 MB.

Robot

Sigurd har fundet en robot, og lavet en bane, som robotton skal igennem. Han har stillet robotten i øverste venstre hjørne, og målet er at robotten skal bevæge sig ned i nederste højre hjørne. Hvert felt på banen er enten tomt, en mur, eller indeholder en mønt. Målet er, at robotten skal samle så mange mønter som muligt. Robotten må kun gå til højre eller ned.

Et eksempel på en robot-tur kan ses i figuren herunder. Her kan robotten maksimalt samle 4 mønter op på vejen ned til målet.



Opgave

Givet banens layout og størrelse ($n \times m$). Beregn hvor mange mønter robotten maksimalt kan samle på en vej fra øverste venstre hjørne til nederste højre hjørne. Robotten kan ikke gå igennem mure, og skal nå ned til højre hjørne, hvis det er muligt.

Input

Første linje af input består af to tal n, m.

De næste n linjer består hver af m tegn: `.`, hvis der ikke er noget på feltet, `#`, hvis feltet er en mur, og `C` hvis der er en mønt på feltet.

Det øverste venstre hjørne og nederste højre hjørne vil altid være et `.`.

Output

Hvis robotten ikke kan nå det nederste højre hjørne skal du skrive `umuligt`. Ellers skal du skrive det maksimale antal mønter robotten kan samle på vejen derned.

Eksempler

Input	Output	Kommentarer
6 8 ....#C.. .C#..C.. ...C.#.C CC.#C..# C#....C. C#..CC#.	4	Dette er eksemplet fra figuren.

Input	Output	Kommentarer
3 3 ... ..# .#.	umuligt	Det er ikke muligt for robotten at nå sit mål.

Input	Output	Kommentarer
3 4 .... CC#.	0	Selvom der ligger to mønter i bunden kan robotten ikke få dem med pga. muren.

Pointgivning

Delopgave 1 (30 point): $1 \leq n, m \leq 8$

Delopgave 2 (70 point): $1 \leq n, m \leq 500$

Begrænsninger

Tidsbegrænsning: 1 s.

Hukommelsesbegrænsning: 256 MB.

Palindrom

Sigurd er blevet stor fan af palindromer, og prøver at finde palindromer i diverse ord og tekster. Et palindrom er en tekst, der er det samme forfra og bagfra. Sigurd finder palindromer ved at fjerne bogstaver indtil det, der er tilbage, er et palindrom. Dette kaldes for en *palindrom delsekvens*. F.eks. kan man i teksten `abekat` fjerne `b`, `e` og `t` og ende med teksten `aka`, hvilket er et palindrom.

Sigurd har fundet nogle store ord, og vil gerne have hjælp fra dig til at finde ud af hvor stort et palindrom der gemmer sig i ordet.

Opgave

Givet et ord S skal du finde ud af hvad den længste palindrom delsekvens af S er.

Input

Input består af en linje, der indeholder ordet S.

S består udelukkende af små bogstaver fra det engelske alfabet (`a`, `b`, ..., `z`).

Output

Længden af den længste palindrom delsekvens af S.

Eksempler

Input	Output	Kommentarer
`abekat`	3	`aka`, `aba` eller `aea` er alle palindrom delsekvenser af længde 3.

Input	Output	Kommentarer
`kebabmester`	5	For eksempel `ebabe`.

Input	Output	Kommentarer
`abaababbab`	7	For eksempel `abababa`.

Pointgivning

Delopgave 1 (30 point): $1 \leq |S| \leq 20$.

Delopgave 3 (70 point): $1 \leq |S| \leq 500$.

Begrænsninger

Tidsbegrænsning: 1 s.

Hukommelsesbegrænsning: 256 MB.

Kapitel 5

Introduktion til grafteori

En stor del af algoritmer drejer sig om grafteori. Vi vil derfor bruge lidt tid på at gøre os bekendt med begrebet *grafer*, inden vi i næste sektion skal se hvordan vi kan bruge disse til at løse mange problemer. Grafer er essentielle i rigtigt mange problemstillinger, så det kan godt betale sig at sætte sig ind i grafteori lidt mere generelt, før vi kigger på specifikke problemer.

5.1 Hvad er en graf?

Forestil dig følgende problem: Du er givet et over byerne og vejene i Danmark og skal udregne den korteste afstand mellem Aarhus og Varde.

For at kunne angribe problemet rent matematisk, er vi nødt til at kunne beskrive det med matematiske begreber.. Det gør vi ved at lave en *graf* over kortet, hvor vi kalder byerne for *knuder* og vejene for *kanter*. En graf over et forsimplet kort kan ses i figur 5.1

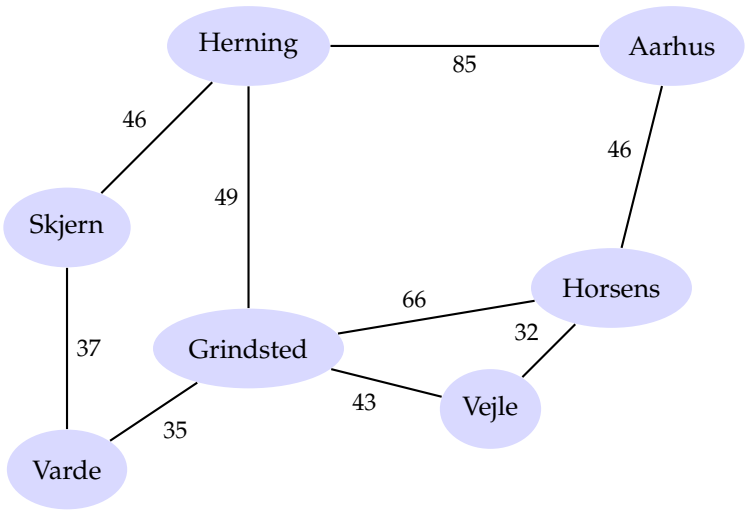
Vi siger at kanterne i grafen har en *vægt*, som i dette tilfælde svarer til afstandene mellem de to byer. F.eks. har kanten mellem Grindsted og Herning en vægt på 49. Givet sådan en graf er der mange spørgsmål vi kan stille:

- Hvad er den korteste vej fra Varde til Aarhus?
- Hvad er den korteste vej mellem alle byer?
- Hvor mange byer kan jeg nå fra Vejle uden at køre mere end 100km?
- Hvor mange veje kan højest gå i stykker før man ikke kan køre fra alle byer til hinanden?

I næste kapitel skal vi prøve at se hvordan vi svarer på nogle af disse spørgsmål.

5.2 At repræsentere en graf i kode

Vi vil ofte bruge en af to måder til at repræsentere en graf i koden: En *nabomatrix* eller *nabolister*.



Figur 5.1: En graf over et forsimplet kort med byer og veje mellem Varde og Aarhus.

I en nabomatrix har vi en tabel A af størrelse $n \times n$, hvor n er antallet af knuder, og indgang $A_{i,j}$ svarer til længden af kanten mellem knude nummer i og j. Hvis der ikke er nogen kant mellem i og j sætter vi $A_{i,j} = \infty$. I en naboliste gemmer vi for hver knude i grafen en liste over par med hvilke knuder der er en kant til fra denne samt hvad vægten er på kanterne. Dette er illustreret i eksemplet på næste side.

Eksempel: Nabomatrix og nabolister svarende til figur 5.1:

	Aarhus	Grindsted	Herning	Horsens	Skjern	Varde	Vejle
Aarhus	∞	∞	85	46	∞	∞	∞
Grindsted	∞	∞	49	66	∞	35	43
Herning	85	49	∞	∞	46	∞	∞
Horsens	46	66	∞	∞	∞	∞	32
Skjern	∞	∞	46	∞	∞	37	∞
Varde	∞	35	∞	∞	37	∞	∞
Vejle	∞	43	∞	32	∞	∞	∞

$$
\begin{aligned}
\text{Aarhus} &\to [(\text{Herning}, 85), (\text{Horsens}, 46)] \\
\text{Grindsted} &\to [(\text{Herning}, 49), (\text{Horsens}, 66), (\text{Varde}, 34), (\text{Vejle}, 43)] \\
\text{Herning} &\to [(\text{Aarhus}, 85), (\text{Grindsted}, 49), (\text{Skjern}, 46)] \\
\text{Horsens} &\to [(\text{Aarhus}, 46), (\text{Grindsted}, 66), (\text{Vejle}, 32)] \\
\text{Skjern} &\to [(\text{Text}, 46), (\text{Varde}, 37)] \\
\text{Varde} &\to [(\text{Grindsted}, 35), (\text{Skjern}, 37)] \\
\text{Vejle} &\to [(\text{Grindsted}, 43), (\text{Horsens}, 32)]
\end{aligned}
$$

Dette kan virke meget abstrakt, men er faktisk rigtigt nemt at implementere i kode. I stedet for tekstuelle navne vil vi dog oftest bruge et heltal som *id* til at referere til hver knude i grafen. Vi vil som regel repræsentere en nabomatrix med et to-dimensionelt array. I Python kan vi initialisere sådan et på følgende måde:

```
1  n = 5
2  A = [[float('inf') for x in range(n)] for x in range(n)]
```

Dette initialiserer et 5×5 array, hvor alle indgange er ∞. Vi kan sige at der er en kant af vægt 5 mellem knude 1 og 2 på følgende måde:

```
1  A[1][2] = 5
2  A[2][1] = 5
```

Syntax: Kommandoen `float('inf')` giver tallet ∞ (altså uendeligt). I Python kan du skrive `float('inf')-float('inf')`, som burde returnere tallet $\infty - \infty$. Da man ikke kan give mening til dette tal returnerer Python tallet `nan` (som står for Not A Number). Pas derfor på at du ikke trækker to tal, der begge er uendelig, fra hinanden.

> **Syntax:** Kommandoen `[float('inf') for x in range(n)]` brugt herover kaldes en *list comprehension* og er en relativt avanceret konstruktion. Hvis du vil vide mere om list comprehensions anbefaler vi, at du slår dem op på internettet.

Hvis vi har en uvægtet graf kan vi nøjes med at have en $n \times n$ matrix med boolske værdier, sådan at $A[i][j]$ er `True` hvis der er en kant fra i til j og `False` ellers. For en graf med 5 knuder kan vi initialiserer matricen således:

```
1  n = 5
2  A = [[False for x in range(n)] for x in range(n)]
```

Vi kan så indsætte en kant mellem knude 1 og knude 2 på følgende måde:

```
1  A[1][2] = True
2  A[2][1] = True
```

For at lave nabolister kan vi bruge en Python dictionary. Følgende kode laver nabolister for en graf med fem knuder der ikke er forbundet til noget.

```
1  n = 5
2  L = [[] for i in range(n)]
```

Vi kan tilfølge kanter til grafen ved at appende til de tilsvarende lister. Vi kan f.eks. igen tilføje en kant af vægt 5 mellem knude 1 og 2:

```
1  L[1].append((2,5))
2  L[2].append((1,5))
```

Hvis vi vil printe alle kanter ud fra knude 3 kan vi gøre det med et for loop som vi har gjort i de tidligere kapitler:

```
1  for e in L[3]:
2      print e
```

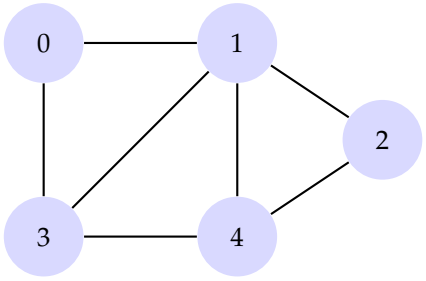
Dette vil skrive en liste af par ud, som repræsenterer alle de kanter der går ud af knude 3. Hvis vi vores graf er uvægtet kan vi i stedet for at indsætte et par `(knude,vægt)` nøjes med at indsætte knude-id'et i vores naboliste. I en uvægtet graf kan vi indsætte en kan mellem knude 1 og 2 på følgende måde:

```
1  L[1].append(2)
2  L[2].append(1)
```

> **Syntax:** Kommandoen `s.append(x)` sætter elementet x bagerst i listen s. Hvis s før var listen $[y, z]$ vil den efter kaldet indeholde elementerne $[y, z, x]$.

5.3 Eksempelkode

For at illustrere, hvordan en graf kan repræsenteres i hukommelsen vil vi kigge på noget kode, der repræsenterer grafen fra Figur 5.2 ved hjælp af en nabomatrix og nabolister. Da vi fortrinsvis vil beskæftige os med uvægtede og ikke-orienterede grafer er Figur 5.2 også uvægtet og ikke-orienteret.



Figur 5.2: En eksempelgraf, der kan implementeres ved hjælp af en nabomatrix eller nabolister.

Lad os først se, hvordan grafen fra Figur 5.2 kan implementeres ved hjælp af en nabomatrix:

```
1  def addEdge(A, i, j):
2      A[i][j] = True
3      A[j][i] = True
4
5  n = 5
6  A = [[False for x in range(n)] for x in range(n)]
7  addEdge(A, 0, 1)
8  addEdge(A, 0, 3)
9  addEdge(A, 1, 2)
10 addEdge(A, 1, 3)
11 addEdge(A, 1, 4)
12 addEdge(A, 2, 4)
13 addEdge(A, 3, 4)
```

Vi kan tælle antallet af kanter i denne graf ved for at tjekke alle værdier af $A[i][j]$ og se om de er `True` eller ej. Hver gang vi finder en værdi af $A[i][j]$ der er `True` har vi fundet en kant. Følgende kode går igennem alle værdier og gemmer antallet af kanter i en variabel m, som derefter udskrives.

```
1  m = 0
2  for i in range(n):
3      for j in range(n):
4          if A[i][j]:
5              m = m + 1
6  print m
```

Men hov! Når vi kører koden udskrives 14, men der er kun 7 kanter i grafen! Hvor er vores fejl? Når vi indsætter en kant fra i til j opdaterer vi både $A[i][j]$ og $A[j][i]$. Kanten mellem i og j tælles derfor to gange! Det kan vi enten gøre op for ved at dele m med to til sidst eller ved at insistere på kun at kigge på værdier $A[i][j]$ hvor $j < i$. Den sidste løsning ser sådan ud:

```
1  m = 0
2  for i in range(n):
3      for j in range(i):
4          if A[i][j]:
5              m = m + 1
6  print(m)
```

Følgende kode er et eksempel på hvordan grafen fra Figur 5.2 implementeres ved hjælp af nabolister.

```
1  def addEdge(L, i, j):
2      L[i].append(j)
3      L[j].append(i)
4
5  n = 5
6  L = [[] for i in range(n)]
7
8  addEdge(L, 0, 1)
9  addEdge(L, 0, 3)
10 addEdge(L, 1, 2)
11 addEdge(L, 1, 3)
12 addEdge(L, 1, 4)
13 addEdge(L, 2, 4)
14 addEdge(L, 3, 4)
```

Lad os nu sige at vi gerne vil finde antallet af kanter i grafen. Det kan vi gøre ved at gennemgå hver naboliste og tælle hvor mange kanter der er i listen. Vi skal huske til sidst at dele resultatet med to, så vi ikke tæller hver kant dobbelt. Følgende kode viser, hvad vi skal tilføje for at det sker.

```
1  m = 0
2  for i in range(n):
3      for j in L[i]:
4          m = m + 1
5  m = m / 2
6  print m
```

Måske kan du gøre dette smartere ved at bruge `len` funktionen?

5.4 Øvelser

Opgave 5.4.1. Lav et program, der læser en graf ind i følgende format: Først et heltal n, der angiver hvor mange knuder grafen har. Dernæst et heltal m, der angiver hvor mange kanter grafen har. Derefter m par af heltal i, j, der angiver, at der er en kant mellem knude i og j.

Dit program skal læse grafen ind og gemme den i en naboliste.

Et eksempel på input er følgende:

```
4
5
0 1
1 2
2 3
0 3
1 3
```

Opgave 5.4.2. `[gt_maxdegree]` Vi siger at *graden* af en knude er antallet af kanter, der rør knuden. Lav et program, der indlæser en graf som i opgave 5.4.1 og udskriver den maximale grad af alle knuder i grafen.

Opgave 5.4.3. `[gt_maxdegree2]` Lav et program, der indlæser en graf som i opgave 5.4.1. Find graden for alle knuder i grafen og udskriv summen af graderne af de to største knuder.

Opgave 5.4.4. `[gt_triangles]` Lav et program, der indlæser en graf som i opgave 5.4.1 og finder alle tripler (i, j, k) i grafen fra Figur 5.2 sådan at i, j, k er forskellige og sådan at der er en kant mellem i og j; en kant mellem j og k; og en kant mellem k og i. Sådan en tripel kaldes en *trekant*.

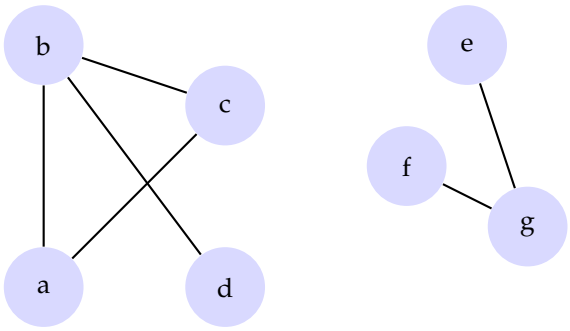
Kapitel 6

Grafsøgning

Vi har nu lært hvad en graf er og hvordan vi repræsenterer dem i programmering. I dette kapitel skal vi kigge på hvordan man søger i en graf. Vi skal kigge på to problemer. Det første er ligetil:

Problem 6.1. *Givet en graf G og to knuder $u, v \in G$. Er der en vej af kanter fra u til v?*

For at forklare det andet problem skal vi først definere et begreb for grafer: Vi siger at to knuder u, v er i det samme *forbundne komponent*, hvis der er en vej fra u til v. For eksempel har grafen i Figur 6.1 to forbundne komponenter.



Figur 6.1: Graf med to forbundne komponenter. Henholdsvis (a, b, c, d) og (e, f, g).

Det andet problem kan nu beskrives som følger:

Problem 6.2. *Givet en graf G. Hvor mange forbundne komponenter består G af, og hvor mange knuder er der i hver?*

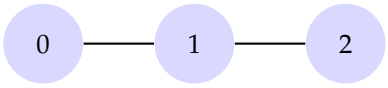
6.1 Dybde-først søgning

Den første type søgning vi skal kigge på er en dybde-først søgning (DFS). Vi er givet to knuder u og v i en graf, og vi ønsker at finde ud af om vi kan komme fra u til v. Den grundlæggende idé er følgende: Hvis u og v er den

samme knude, så kan vi svare "Ja, vi kan godt nå u fra v". Hvis u og v er forskellige kigger vi på alle u's naboer. Fra hver nabo, lad os kalde den w, spørger vi så om vi kan komme fra w til v. Hvis vi kan det for en af naboerne svarer vi "Ja", og ellers "Nej". Vi vil først se på en naiv implementation af denne idé, når grafen er repræsenteret ved nabolister.

```
1  def dfs(u, v):
2      if u == v:
3          return True
4      for w in L[u]:
5          if dfs(w, v):
6              return True
7      return False
```

I funktionen `dfs(u, v)` får vi to knuder u og v, hvor vi ønsker at besvare om der findes en sti fra u til v. Vi antager at der findes nabolister, der er gemt i variablen L. Denne implemenation er "naiv" fordi den aldrig vil stoppe. Betragt den meget simple graf i Figur 6.2, og lad os sige at vi kalder `dfs(0, 2)` da vi vil finde ud af om 0 og 2 er forbundne. Programmet vil herefter kigge på den første nabo og kalde `dfs(1, 2)`. Når denne funktion bliver kaldt vil programmet igen kalde `dfs(0, 2)` og vi er tilbage hvor vi startede! Programmet vil derfor gå i ring og aldrig finde ud af, at der er en sti fra 0 til 2! Konceptuelt kan vi løse problemet med at programmet går i ring på



Figur 6.2: En graf med tre knuder og to kanter.

følgende måde: Først gang vi søger ud fra en knude u (altså når vi kalder `dfs(u,v)`) markerer vi ved knuden at vi har besøgt den. De efterfølgende gange vi søger fra knuden u vil vi ikke søge videre. På denne måde sikrer vi os at programmet ikke går i ring. I praksis vil vi have et array `vis` (forkortelse for *visited*) af længde n, der til at starte med indeholder værdien `False` på alle pladser. Dette indikerer at ingen af knuderne er besøgt endnu. Vi kan lave sådan et array ved at skrive:

```
1  vis = [False for i in range(n)]
```

I toppen af vores funktion vil vi først tjekke om u er besøgt før ved at kigge på værdien af `vis[u]`. Hvis den ikke er besøgt før vil vi så sætte `vis[u] = True` of fortsætte som vanligt. Samlet vil vores funktion nu så således ud:

```
1  def dfs(u, v):
2      if vis[u]:
3          return False
4      vis[u] = True
5      if u == v:
6          return True
7      for w in L[u]:
8          if dfs(w, v):
9              return True
10     return False
```

Den opmærksomme læser vil nu bemærke at funktionen ovenfor er unødigt kompliceret. Svaret på om vi kan komme fra u til v kan vi nemlig læse ud fra værdien af `vis[v]`. Vi kan defor simplificere vores funktion sådan at den kun tager en parameter, u, i stedet. Vi kan slette sammenligningen hvor vi tjekker om u og v er ens, og vi behøver ikke returnere en værdi. Den endelige funktion ser således ud:

```
1 def dfs(u):
2     if vis[u]:
3         return
4     vis[u] = True
5     for w in L[u]:
6         dfs(w)
```

Øvelser

Opgave 6.1.1. `[gt_reachability]` Løs problem 6.4.

Opgave 6.1.2. `[gt_reachabilitydead]` Lav et program, der indlæser følgende: Først en graf i samme format som beskrevet i Opgave 5.4.1. Herefter en linje bestående af 2 tal u, v separeret af mellemrum , der er id's på knuder, som vi vil finde ud af om er forbundne. Til sidst en linje bestående af id's på knuder i grafen, der er døde. Disse id's er separeret af mellemrum. Programmet skal finde ud af om, der findes en vej fra u til v, der ikke bruger nogen døde knuder. Hvis der gør skal det udskrive `Ja` og ellers `Nej`.

Et eksempel på de to linjer ud over beskrivelsen af grafen kunne være:

```
0 1
2 3 4
```

Opgave 6.1.3. `[gt_doublereachability]` Lav et program, der indlæser følgende: Først en graf i samme format som beskrevet i Opgave 5.4.1. Herefter en linje bestående af 3 tal u, v, w separeret af mellemrum, der er id's på knuder. Hvis der findes en vej fra u til v eller fra u til w (evt. til begge) skal programmet udskrive `Ja` og ellers `Nej`.

Opgave 6.1.4. `[gt_triplereachability]` Lav et program, der indlæser følgende: Først en graf i samme format som beskrevet i Opgave 5.4.1. Herefter en linje bestående af 3 tal u, v, w separeret af mellemrum, der er id's på knuder. Programmet skal udskrive tre linjer:

På den første linje skal der stå `Ja`, hvis der er en sti mellem u og v og `Nej` ellers.

På den anden linje skal der stå `Ja`, hvis der er en sti mellem v og w og `Nej` ellers.

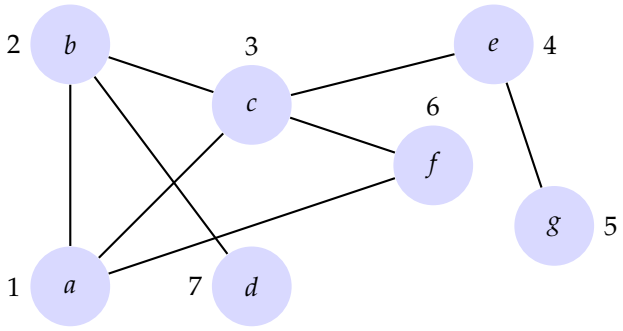
På den tredje linje skal der stå `Ja`, hvis der er en sti mellem w og u og `Nej` ellers.

6.2 Dybde-først søgning (i dybden)

I Section 6.1 så vi hvordan vi kunne afgøre om, der er en vej i mellem to knuder u og v. Vi fandt faktisk ud af, at det nemmeste var at starte i u og finde alle knuder, der kunne nås fra u. Vi har fundet en algoritme, der kan afgøre, om vi kan nå mellem to knuder - men vi har endnu ikke set om vi rent faktisk kan genskabe den vej, der er mellem knuderne. For at gøre dette, vil vi kigge lidt nærmere på, hvordan en DFS virker, og i hvilken rækkefølge knuderne bliver besøgt.

En eksekvering af en DFS er illustreret i Figur 6.3. I dette eksempel starter vi i knude a, og hver gang vi skal besøge den næste knude tager vi den nabo der har det første bogstav i alfabetisk orden. Dvs. at nabolisten er sorteret således at dette er tilfældet. Når vi er i en knude, hvor alle naboer er besøgt slutter funktionskaldet. Derfor tager vi et skridt tilbage til den knude vi kom fra. Hvis denne ikke har flere ubesøgte naboer gør vi det igen, osv. På denne måde bliver knuderne besøgt i rækkefølgen angivet i Figur 6.3. Herunder følger en mere detaljeret beskrivelse af hvordan:

1. a
2. b (kommer før c og f)
3. c (kommer før d og a er allerede besøgt)
4. e (kommer før f og a, b er allerede besøgt)
5. g (eneste ubesøgte nabo)
6. f (vi har backtracket til da vi besøgte c, og f er eneste ubesøgte nabo)
7. d (backtracket til da vi besøgte c, og d er eneste ubesøgte nabo)



Figur 6.3: Eksempel på dybde-først søgning. Vi starter i knude a og besøger altid først den nabo, der kommer først i alfabetisk orden.

Lad os først huske, hvordan vi implementerede en DFS i Python:

```
1  vis = [False for i in range(n)]
2  def dfs(u):
3      if vis[u]:
4          return
```

```
5       vis[u] = True
6       for w in L[u]:
7           dfs(w)
```

Vi ønsker nu at modificere vores implementation således at `vis[u]` indeholder -1, hvis en knude u aldrig bliver besøgt, og hvis u bliver besøgt som den i'te knude skal `vis[u]` indeholde tallet i. Først skal vi altså ændrer måden hvorpå `vis` initialiseres:

```
1  vis = [-1 for i in range(n)]
```

For at holde styr på, hvor mange knuder, der er besøgt indtil videre bruger vi en parameter, `num`, som vi initialiserer til at være 0:

```
1  num = 0
```

I stedet for at sætte `vis[u]` til `False` gør vi følgende: Først gør vi `num` 1 større, da vi nu har besøgt en knude mere. Herefter sætter vi `vis[u]` til at være lig værdien af `num`. Den endelige kode ser således ud:

```
1  vis = [-1 for i in range(n)]
2  num = 0
3  def dfs(u):
4      if vis[u] == -1:
5          return
6      num = num+1
7      vis[u] = num
8      for w in L[u]:
9          dfs(w)
```

6.3 Rekonstruktion af veje

Indtil videre har vi set, hvordan vi kan finde ud af om der er en vej fra u til v. Et helt naturligt spørgsmål er om vi også kan finde sådan en vej når den eksisterer. Vi vil i denne sektion se, hvordan vi kan bruge arrayet `vis` fra Section 6.2 til at gøre dette. Lad os til at starte med indse at følgende er sandt efter vi har kørt algoritmen fra Section 6.2.

> **Observation:** Lad u være en knude så `vis[u]` hverken er -1 eller 1. Da har u en nabo v sådan at `vis[v]` $<$ `vis[u]`.

Dette gælder af følgende årsag: Da `vis[u]` ikke er -1 har vi på et tidspunkt lavet funktionskaldet `dfs(u)`. Og da `vis[u]` ikke er 1 er det ikke det første funktionskald vi har lavet. Derfor må programmet på et tidspunkt have kaldt `dfs(v)` for en knude v, sådan at `dfs(u)` derefter er kaldt rekursivt. Dette betyder at u og v må være naboer. Da `dfs(v)` blev kaldt før `dfs(u)` må `vis[v]` være mindre end `vis[u]`.

Vi kan nu bruge denne observation til at finde en vej i mellem u og v på følgende måde. Først laver vi en DFS fra u ved at køre algoritmen fra Section 6.2. Herefter kan vi finde en vej fra v til u således:

- Hvis `vis[v]` $= -1$ findes ingen vej.

- Hvis `vis[v]` $= 1$ er $v = u$ og så behøver vi ikke gøre noget for at finde en vej.
- Ellers findes en nabo w til v sådan at `vis[w]` $<$ `vis[v]`.
- Find *rekursivt* en vej fra w til u og tilføj kanten mellem v og w i begyndelsen denne vej.

Ideen i algoritmen er altså at udnytte, at vi hele tiden kan finde en nabo med en mindre `vis`-værdi. Vi ønsker at repræsentere vejen ved hjælp af en liste på formen $[v, v_1, v_2, \cdots, v_k, u]$. Når der ikke er nogen vej mellem u og v vil vi udskrive den tomme liste, $[]$, og når $u = v$ vil vi udskrive listen $[u]$. Lad os se, hvordan vi kan implementere dette i Python:

```
1  def findPath(v):
2      if vis[v] == -1:
3          return []
4      if vis[v] == 1:
5          return [v]
6      for w in L[u]:
7          if vis[w] < vis[v]:
8              return [v] + findPath(w)
```

Nu kan vi bruge definitionerne af `dfs` og `findPath` til at finde en vej i mellem u og v på følgende måde:

```
1  vis = [-1 for i in range(n)]
2  num = 0
3  dfs(u)
4  print findPath(v)
```

> **Syntax:** Givet to lister `a` og `b` kan vi sætte dem efter hinanden ved at skrive `a+b`. Fx giver `[5,1]+[1,2] = [5,1,1,2]`

6.4 Forbundne komponenter

Endnu en anvendelse af DFS er at vi kan finde de forbundne komponenter i en graf. Mere specifikt vil vi i denne sektion løse følgende problem:

Problem 6.3. *Givet en graf G. Hvor mange forbundne komponenter består G af, og hvor mange knuder er der i hver?*

Vi vil gøre dette i to steps: Først vil vi give alle knuder et *mærkat*, sådan at alle knuder i samme komponent har samme mærkat. Dette vil vi gøre ved hjælp af en DFS. Dernæst vil vi tælle, hvor mange forskellige mærkater, der er givet, og hvilket mærkat, der optræder flest gange.

Mærkaterne gives på følgende måde: Til at starte med giver vi alle knuder mærkatet -1 - vi tænker på dette mærkat som et, der fortæller at vi ikke har besøgt knuden endnu. Derefter vælger vi en vilkårlig knude. Denne knude giver vi mærkatet, 0: Vi starter en DFS i knuden og alle knuder, som vi kan nå fra denne knude får mærkatet, 0. Derefter finder vi en ny knude, som har

mærkatet -1 og ved en DFS gives mærkatet 1 til denne knude samt alle andre knuder i dens forbundne komponent. Således bliver vi ved indtil ingen knude har mærkatet -1.

I praksis implementerer vi det ved at have en liste, `labels`, der indeholder mærkaterne for alle knuderne. Vi initialiserer den ved:

```
1  labels = [-1 for i in range(n)]
```

I vores DFS er vi givet to parametre: Den knude vi er nået til, u, og det mærkat knuden skal have, l. Vores DFS funktion, der giver mærkatet, l, til alle knuderne i u's forbundne komponent ser således ud:

```
1  def dfs(u, l):
2      if labels[u] != -1:
3          return
4      labels[u] = l
5      for w in L[u]:
6          dfs(w, l)
```

Bemærk at vi ligesom i de tidligere afsnit antager at vi har en naboliste for hver knude.

Vi kan nu bruge funktionen `dfs` til at tildele mærkater til knuderne på følgende måde:

```
1  num = 0
2  for u in range(n):
3      if labels[u] == -1:
4          dfs(u, num)
5          num = num + 1
```

Variablen `num` angiver hvor mange forskellige mærkater vi har givet til knuderne. I løkken i linje 2 går vi i gennem alle knuderne i grafen. I linje 3 tjekkes om knuden er besøgt før. Hvis den ikke er skal vi bruge et nyt mærkat, som vi tildeler i linje 4 og 5.

Efter at have givet mærkater til knuderne skal vi nu løse det oprindelige problem. At finde antallet af forbundne komponenter viser sig at være let, da det er det samme, som antallet af forskellige mærkater - dvs. `num`. At finde det største komponent er en smule mere vanskeligt. Vi kan gøre dette ved at lave en liste over størrelserne af alle komponenterne og derefter finde det største tal i denne liste. Vi kan finde størrelserne således:

```
1  sizes = [0 for i in range(num)]
2  for u in range(n):
3      sizes[labels[u]] += 1
```

På linje 1 initialiserer vi en liste over størrelserne på alle komponenterne, der til at starte med indeholder 0 for alle komponenter. På linje $2-3$ findes størrelserne af komponenterne. At finde den største værdi overlades som en øvelse til læseren.

Øvelser

Opgave 6.4.1. `[gt_components]` Løs problem 6.5.

Opgave 6.4.2. `[gt_components2]` Indlæs en graf i samme format som beskrevet i Opgave 5.4.1. Find de to største komponenter i grafen og udskriv

summen af deres størrrelser. Hvis der kun er et komponent i grafen skal du udskrive antallet af knuder i det komponent.

6.5 Problemer

Problem 6.4. *Reachability*

Problem 6.5. *Components*

Problem 6.6. *SpringerDFS*

Problem 6.7. *Den (u)lige vej hjem.*

6.6 Bredde-først søgning

Indtil videre har vi kun undersøgt hvilke knuder, der er forbundne i en graf, men der er mange andre egenskaber, som er interessante. Afstanden mellem to knuder u og v defineres til at være det mindste antal kanter, der er på en vej fra u til v. Hvis u og v ikke er forbundne defineres afstanden til at være ∞. Vi vil ved hjælp af bredde-først søgning (BFS) løse følgende problem:

Problem 6.8. *Givet en graf G og to knuder $u, v \in G$. Hvad er afstanden mellem u og v?*

Da vi skulle finde ud af om u og v var forbundne fandt vi i Section 6.1 ud af at det var lettere at starte i u og finde alle de knuder u var forbundne til og derefter tjekke om v var en af disse knuder. På samme måde vil vi finde afstanden fra u til samtlige knuder i grafen og derefter blot aflæse afstanden til v.

Når vi laver en bredde-først søgning fra u vil vi først besøge alle knuder med afstand 1 til u, derefter alle knuder med afstand 2, osv. Med andre ord besøger vi først u's naboer, derefter u's naboers naboer, og sådan bliver vi ved indtil vi har besøgt alle knuder i grafen. Dette gør vi ved at have en *kø*, hvor vi indsætter alle de knuder, som vi vil besøge. Konceptuelt vil algoritmen virke således: Først oprettes køen, og der indsættes et enkelt element, u. Derefter gøres følgende indtil køen er tom: Lad v være den forreste knude i køen. Hvis v ikke er besøgt endnu vil vi besøge v. Når vi besøger v indsætter vi indsætter alle v's naboer bagerst i køen.

Lad os overveje i hvilken rækkefølge vi vil besøge knuderne med denne metode. Først vil vi naturligvis besøge u, og derfor indsætte alle u's naboer i i køen. Derefter vil vi besøge disse naboer en af gangen og indsætte alle deres naboer bagerst i køen. Bemærk at dette er i modsætning til, hvis vi havde lavet en DFS, hvor vi generelt ikke vil besøge alle u's naboer med det samme. Efter vi har besøgt alle u's naboer vil køen altså bestå af alle u's naboers naboer. Dette kan kun være knuder med afstand 2 samt knuder som vi allerede har besøgt. Når vi så har besøgt alle knuder med afstand 2 vil køen udelukkende bestå af knuder, der enten har afstand 3 eller allerede er besøgt, og sådan vil det altså blive ved.

Implementationen ligner implementationen af DFS, men der er den vigtige forskel at vi ikke laver rekursive kald.

```
dist = [float('inf') for _ in range(n)]
queue = [(u,0)]
while len(queue) > 0:
    (v,d) = queue[0]
    queue = queue[1:]
    if dist[v] != float('inf'):
        continue
  dist[v] = d
    for w in L[v]:
        queue = queue + [(w,d+1)]
```

Øvelser

Opgave 6.6.1. Professor Klogesen påstår, at hvis en knude puttes forrest i køen i stedet for bagerst i køen så laver vi en DFS i stedet for en BFS. Har han ret? Og hvordan vil du tjekke det?

6.7 Problemer

Problem 6.9. *Epidemi.*

Problem 6.10. *Djævlens divisorgraf.*

Reachability

Du har en graf med n knuder og m kanter samt to knuder u, v og skal finde ud af om der er en vej fra u til v.

Input

Første linje indeholder n, anden linje indeholder m.

De næste m linjer indeholder hver to tal a, b, som angiver at der er en kant mellem knuderne a og b.

Den sidste linje indeholder u efterfulgt af v.

Output

Ordet `Ja` hvis der er en vej fra u til v og `Nej` ellers.

Eksempler

Input	Output	Kommentarer
4 3 0 2 1 3 2 3 0 1	Ja	Vi kan følge vejen $0 \to 2 \to 3 \to 1$.

Input	Output	Kommentarer
5 4 0 1 1 2 2 0 3 4 0 4	Nej	Knuderne 0 og 4 er i to forskellige komponenter af grafen.

Pointgivning

Delopgave 1 (100 point): $1 \leq n, m \leq 10^6$.

Begrænsninger

Tidsbegrænsning: 1 s.

Hukommelsesbegrænsning: 256 MB.

Komponenter

Du har en graf med n knuder og m kanter. Du skal finde ud af hvor mange disjunkte komponenter grafen består af samt størrelsen på det største.

Input

Første linje indeholder n, og anden linje indeholder m.

De næste m linjer indeholder hver to tal a, b, som angiver at der er en kant mellem knuderne a og b.

Output

To mellemrumsseparerede heltal: Antallet af disjunkte komponenter efterfulgt af størrelsen af det største komponent.

Eksempler

Input	Output	Kommentarer
6 5 0 1 1 2 0 3 2 3 3 1	3 4	Knuderne $0, 1, 2, 3$ udgør et komponent. Knuderne 4 og 5 udgør hvert deres.

Input	Output	Kommentarer
7 4 0 4 4 5 1 2 2 6	3 3	Grafen indeholder to komponenter af størrelse 3 samt et komponent der bare er knude nummer 3.

Pointgivning

Delopgave 1 (100 point): $1 \leq n, m \leq 10^6$.

Begrænsninger

Tidsbegrænsning: 1 s.

Hukommelsesbegrænsning: 256 MB.

Springer DFS

Opgave

Du er givet et $n \times n$ skakbræt, hvor der står en springer på, og hvor nogle af felterne er *forgiftede*. Udregn hvilke felter springeren kan nå fra dens startposition, hvis den ikke må ramme et forgiftet felt. Springeren må flytte som normalt efter skakspillets regler.

Input

Første linje af input består af tre heltal n, i og j. n angiver skakbrættets størrelser og (i, j) springerens position således at springeren starter i den i'te række og j'te kolonne. Position er 0-indekseret (så $0 \leq i, j \leq n - 1$.)

Herefter følger n linjer. Den i'te af disse linjer beskriver den i'te række i skakspillet, således at det j'te tegn på den i'te linje er `X`, hvis feltet (i, j) er forgiftet og `-`, hvis det ikke er forgiftet.

Output

Dit ouput skal bestå af n linjer, der hver består af en tekstreng på n tegn. Det j'te tegn på den i'te linje skal være A hvis springeren kan nå feltet og B, hvis den ikke kan nå feltet.

Eksempler

Input	Output
4 0 0 -X-- ---- -X-- -XXX	ABBB BBAB ABBB BBBB

Pointgivning

Delopgave 1 (100 point): $1 \leq n \leq 100$

Begrænsninger

Tidsbegrænsning: 1 s.

Hukommelsesbegrænsning: 256 MB.

Den (u)lige vej hjem

Jan bor i Grafstrup, der kan modelleres som en graf: Alle veje i byen er kanter i en graf, der forbinder to knuder. En dag da Jan vågner og er på vej hjem finder han ud af at der er sket noget mærkeligt: Han har fået en pludselig trang til at finde en vej hjem, hvor han bruger et ulige antal kanter. Jan spørger dig om hjælp til at finde ud af om man kan finde en vej med et ulige antal kanter.

Opgave

Givet en beskrivelse af Grafstrup bestående af n knuder og m kanter, samt hvilke knuder Jan og Jans hjem er på skal du finde ud af om der findes en vej fra Jan til hans hjem, hvor der bruges et ulige antal kanter.

Input

Første linje indeholder fire tal n, m, u og v: antallet af knuder, antallet af kanter, Jans placering og Jans hjems placering. Knuderne kaldes $0, 1, 2, \ldots, n-1$

De næste m linjer indeholder hver to tal i, j, der angiver at der er en kant fra knude i til knude j.

Output

Hvis der findes en vej fra Jan til hans hjem med et ulige antal kanter skal du udskrive `Ja` og ellers `Nej`.

Eksempler

Input	Output
5 4 0 4 0 1 0 2 0 3 1 4	Nej

Input	Output	Kommentarer
5 5 0 4 0 1 0 2 0 3 1 4 2 3	Ja	Vejen $0 \to 2 \to 3 \to 0 \to 1 \to 4$ bruger et ulige antal kanter.

Pointgivning

Delopgave 1 (100 point): $1 \leq n, m \leq 10\,000$.

Begrænsninger

Tidsbegrænsning: 1 s.

Hukommelsesbegrænsning: 256 MB.

Epidemi[1]

I grafstrup er der en epidemi under opsejling. En i byen har fået den meget farlige grafsyge, der spreder sig ved at diskutere grafproblemer. I grafstrup er det kotyme at diskutere et grafproblem med alle sine venner hver dag, og derfor vil sygdommen hurtigt sprede sig. På første dag er der altså netop én person med sygdommen. Den næste dag har han og alle hans venner den, osv.

Da det er umuligt at få folk i byen til at ændre adfærd har du besluttet dig for at lave en undersøgelse. Du ved hvem der er venner med hvem i byen, og vil derfor finde ud af hvilken dag sygdommen har det største "boom". Altså, hvilken dag sygdommen spreder sig til flest nye mennesker.

Opgave

Givet n personer og m venskabsrelationer samt hvilken person der har sygdommen til at starte med skal du beregne hvad det største antal personer der får sygdommen på en given dag er samt den første dag dette sker.

Input

Første linje indeholder n, og anden linje indeholder m.

De næste m linjer indeholder hver to tal i, j, der angiver at person i er venner med person j.

Den sidste linje indeholder p – personen der er smittet til at starte med.

Output

To heltal adskilt af et mellemrum: (1) Det største antal nye mennesker der bliver smittet på en dag. (2) Den første dag hvor det antal mennesker bliver smittede.

Eksempler

Input	Output	Kommentarer
5 4 0 1 1 2 1 3 1 4 0	3 2	På den første dag smitter person 0 person 1. På den anden dag smitter person 1 personerne $2, 3, 4$.

[1] Kraftigt inspireret af http://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=865

Input	Output	Kommentarer
9 8 0 1 1 2 1 3 2 3 3 4 3 5 5 6 7 8 0	2 2	På dag 2 bliver personerne 2 og 3 smittet og det er første gang der er to der bliver smittet. Bemærk at hverken person 7 eller person 8 nogensinde bliver smittet.

Pointgivning

Delopgave 1 (100 point): $1 \leq n, m \leq 10^6$.

Begrænsninger

Tidsbegrænsning: 1 s.

Hukommelsesbegrænsning: 256 MB.

Djævlens divisorgraf

Bent har indgået en kontrakt med Djævlen og har desparat brug for din hjælp. Djævlen vil give Bent en opgave, og hvis Bent kan løse den får han hundredevis af sjætlandsponyer (og de er jo så søde!), men gør han ikke må han give afkald på sin sjæl.

Da Djævlen har hørt at Bent er meget dårlig til grafteori har han udtænkt en genial plan: Han vil give Bent en opgave i en meget mærkelig graf! Grafen, som vi kalder G, er defineret på følgende måde: Knuderne i grafen kaldes $1, 2, 3, \ldots, 100$. Der er en kant mellem to knuder n og m hvis og kun hvis forskellen, $m-n$, går op i n. Dvs. at hvis divisorene i n er $d_1, d_2, \ldots, d_k$ så har n kanter til $n + d_1, \ldots, n + d_k$ og $n - d_1, \ldots, n - d_k$ (dog kun såfremt at disse tal er mellem 1 og 100.)

Djævlens opgave til Bent er at han skal finde afstanden mellem to knuder i grafen. Hjælp Bent med at finde afstanden så han kan beholde sin sjæl!

Opgave

Du er givet to heltal a og b og skal udskrive afstanden mellem a og b i G.

Input

Input består af præcis en linje med 2 tal: a og b.

Output

Du skal udskrive et enkelt heltal, afstanden mellem a og b i djævlens divisorgraf.

Eksempler

Input	Output	Kommentarer
`6 21`	`3`	Fx ved vejen $6 \to 12 \to 18 \to 21$

Pointgivning

Delopgave 1 (100 point): $1 \leq a, b \leq 100$.

Begrænsninger

Tidsbegrænsning: 1 s.

Hukommelsesbegrænsning: 256 MB.

6.8 Teoretiske grafteori opgaver

Denne sektion indeholder grafteori opgaver, der er mere fokuserede på teorien.

Opgave 1 Find en graf G med $n = 5$ knuder således, at der ikke er 3 knuder uden en kant imellem eller 3 knuder med kanter mellem alle 3.

Opgave 2 Er det muligt at lave en vej i grafen på Figur 6.4, der bruger hver kant præcist én gang? Hvis ja, hvad er vejen?

Opgave 3 Hvad er det færreste antal kanter der skal tilføjes til grafen i Figur 6.5 for at det er muligt at lave en tur der starter i en knude, bruger alle kanter i grafen, og kommer tilbage til den samme knude?
Tilføj kanterne til figuren og illustrer denne tur.

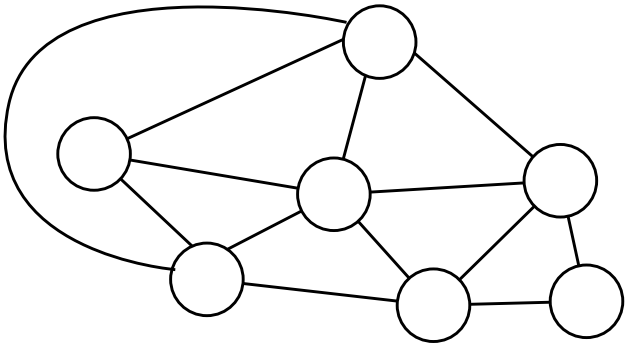
Opgave 4 Hvad er længden af den længste augmenting path i grafen i Figur 6.6?
Hvor mange augmenting paths er der i grafen i Figur 6.6?
Hvad er den største matching i grafen i Figur 6.6?

Opgave 5 Udfyld knuderne i grafen i Figur 6.7 med tallene $1, 2, 3$ således at hver trekant i grafen har en knude med hvert tal i.
Hvad er det mindste antal knuder i grafen der skal have et 1-tal for at alle trekanter har et 1-tal i sig?

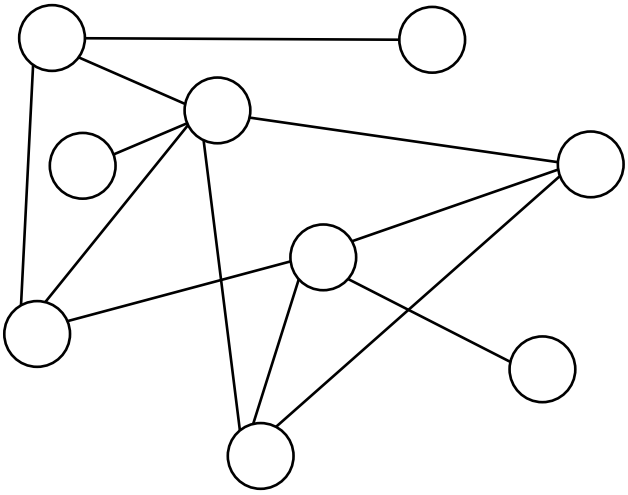
Opgave 6 Bevis at summen af grader i en graf G altid er et lige tal.

Opgave 7 Bevis at antallet af knuder i en graf G, der har ulige grad, er lige.

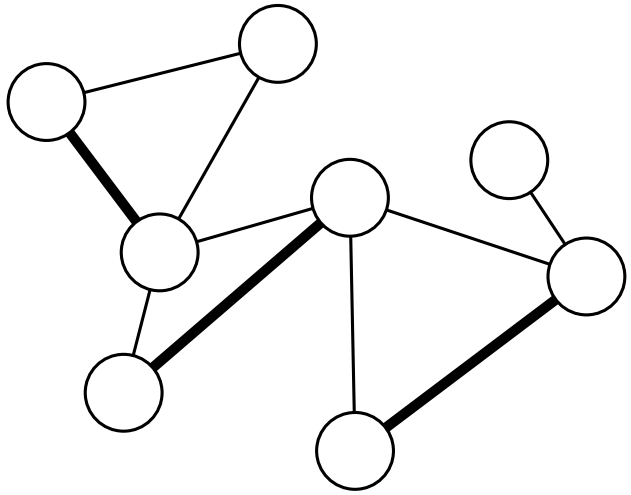
Opgave 8 Bevis, at hvis u er en knude med ulige grad, så er der en vej fra u til en anden knude v med ulige grad.



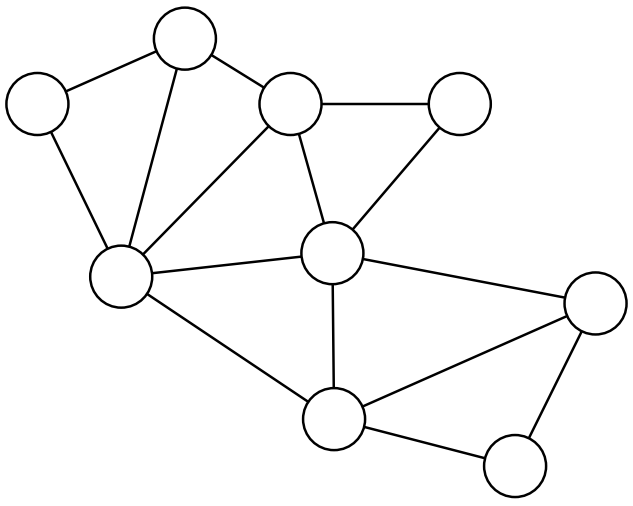
Figur 6.4: Graf til opgave 2



Figur 6.5: Graf til opgave 3



Figur 6.6: Graf til opgave 4. De fede kanter angiver knuder der er "matched"



Figur 6.7: Graf til opgave 5

Sponsorer:

Jobindex

COMPUTERWORLD

it-jobbank

ITMINDS
YOUNG MINDS - EXCELLENT SOLUTIONS

LUNDBECKFONDEN

OLE KIRK's Fond

oticon
PEOPLE FIRST

OTTO MØNSTEDS FOND

Knud Højgaards Fond

badgeland
Støtter videnskaben

be Copenhagen

Karl Pedersen og Hustrus Industrifond
Administreret af DI

KØBENHAVNS
UNIVERSITET

UNF