Rapportera gärna fel och oklarheter i detta dokument till progII-lärare.

Inledning

Detta dokument är material till det första modulen (M1) på kursen Programmeringsteknik II. Det handlar om algoritmkonstruktion med rekursion samt analys av algoritmer.

En del övningar är märkta med texten "Obligatorisk!". Dessa utgör den första obligatoriska uppgiften.

Till modulen hör ett antal inspelade lektioner som ger ytterligare förklaringar och exempel.

Instruktioner

1. Den zip-fil som hör till modulen innehåller följande filer:
   (a) `MA1.py` : Uppgifterna ska skrivas i detta dokument.
   (b) `MA1_test.py` : *Unittester* för de koduppgifter du ska skriva.
   (c) `MA1_examples.py` : Kodexempel som finns i texten.
2. Uppgifterna ska lösas i den ordning de kommer och med de verktyg (klasser, funktioner, metoder, ...) som är genomgångna före övningen.
3. Om inget annat sägs får du *inte* använda andra paket än de som redan finns inkluderade i den nedladdade koden.
4. Lösningarna måste heta och returnera exakt det som står i uppgiften.
5. Skriv egen testkod i `main`-funktionen. När du tycker att dina lösningar fungerar kör du den nedladdade `MA1_test.py`. Observera unittester av en funktion kan bygga på att andra funktioner fungerar. Det är alltså ingen mening att prova unittester innan man löst och själv testat alla de obligatoriska uppgifterna i denna modul.
6. Avsnitt och uppgifter märkta med asterisk (*) är överkurs och behöver inte läsas eller göras.
7. Uppgifter som ska besvaras med text skrivs som kommentarer eller textsträngar sist i koddokumentet. Textsvaren kan hållas korta.
8. Uppgifterna ska först redovisas muntligt för assistent eller lärare och sedan laddas upp i Studium.
9. Vid den munliga redovisningen ska du börja med att legitimera dig. När redovisningen är klar så fyller du i de första raderna i dokumentet (namn, mail, granskare, datum) och laddar upp filen i Studium under MA1.

Observera att du bara ska ladda upp **en** fil och det ska vara en `.py`-fil som är direkt körbar i Python! Vi accepterar inga andra format (word, pdf-filer, Jupyter notebook, ...) och inte heller några inlämningar via mail.

Observera:
Du får samarbeta med andra studenter, men du måste skriva och kunna förklara din egen kod. Du får inte kopiera eller skriva av kod vara sig från andra studenter eller från nätet förutom från de ställen som explicit pekas ut i denna lektion. Att byta variabelnamn och liknande modifikationer räknas inte som att skriva sin egen kod.
Du får inte sprida dina lösningar varken direkt till andra studenter eller allmänt på nätet — även medhjälp till fusk räknas som fusk.
Eftersom uppgifterna ingår som moment i examinationen är vi skyldiga att anmäla brott mot dessa regler.
Allt material som vi tillhandahåller är upphovsrättsligt skyddat och får inte spridas.

Algoritmkonstruktion med rekursion

Vi skall demonstrera hur rekursion går till genom ett antal exempel. I många av exemplen finns det enkla iterativa algoritmer eller inbyggda funktioner/metoder som kan lösa problemen men vårt fokus är här att *öva rekursivt tänkande*!

Exempel 1: Fakultetsberäkning

Antag att vi vill skriva en funktion som beräknar $n! = n(n-1)(n-2)\cdot\ldots\cdot 2\cdot 1$ för ett naturligt tal $n \geq 1$. Man definierar också $0! = 1$.

En Python-funktion för att utföra beräkningen:

```
1 def fac(n):
2     p=1
3     for x in range(1,n+1):
4         p=p*x
5     return p
```

Denna algoritm som ansluter väl till den givna definitionen, sägs vara *iterativ* eftersom den använder en *iteration* (en `for`-sats) för att utföra beräkningen.

Det går också att definiera fakultetsuttrycket *rekursivt*:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n > 0 \end{cases}$$

Definitionen av $n!$ är rekursiv eftersom den använder sig av $(n-1)!$ dvs. av samma sak som ska definieras.

Detta kan utnyttjas i programmeringen genom att skriva en *rekursiv* funktion:

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

Den är rekursiv eftersom den innehåller ett anrop till sig själv, dock med ett annat argument

(`n - 1` i stället för `n`) förutom om funktionen anropas med argumentet 0 — ett så kallat *basfall*.

För att, till exempel, beräkna `fac(3)` måste först `fac(2)` beräknas. För att beräkna `fac(2)` måste `fac(1)` beräknas vilket i sin tur kräver att `fac(0)` beräknas. Värdet på `fac(0)` kräver inte något nytt anrop utan värdet kan returneras direkt.
Detta kan illustreras i vidstående figur.

```
1  fac(3):
2  |  3*fac(2)
3  |     |  2*fac(1)
4  |     |     | 1*fac(0)
5  |     |     |   1
6  |     |     1
7  |     2
8  6
```

När programmet kommer in i `fac(0)` finns det alltså ytterligare tre oavslutade anrop till `fac` var och en med sitt argument. Eftersom `fac(0)` inte anropar vidare kan den returnera sitt värde till den väntande `fac(1)`.

Den första multiplikationen som utförs är den på rad 4 och den sista är den på rad 2.

Observera att det alltid måste finnas minst ett *basfall* som bryter anropskedjan (rekursionen).

Exempel 2: Beräkning av x^n - första version

Vi skall beräkna x^n, n heltal ≥ 0, med upprepade multiplikationer.

Iterativ definition: $x^n = x \cdot x \cdot x \cdot x \cdot \ldots \cdot x$

Iterativ funktion:

```
1  def power(x,n):
2      p=1
3      for i in range(1,n+1):
4          p=p*x
5      return p
```

Rekursiv definition:

$$x^n = \begin{cases} 1 & \text{om } n = 0 \\ x \cdot x^{n-1} & \text{om } n > 0 \end{cases}$$

Övning 1: Funktionen `power`.

Skriv funktionen `power(x, n)` som beräknar och returnerar x^n med ovanstående *rekursiva* algoritm.

Utvidga också programmet så att det kan hantera negativa exponenter. Tänk även här rekursivt!

Anmärkning: I kapitlet om algoritmanalys kommer vi presentera ett väsentligt effektivare sätt att göra beräkningen.

Övning 2: Funktionen `multiply`. Obligatorisk!

Skriv en rekursiv funktion (dvs. utan iteration) `multiply(m, n)` där `m` och `n` är två icke-negativa heltal. Funktionen ska returnera produkten `m*n` och beräkningen ska göras *utan* att använda multiplikation. Produkten skall alltså beräknas med *additioner*.

Om du skulle vilja minimera antalet rekursiva anrop hur skulle du då modifiera koden?

Övning 3: Funktionen `divide`.

Skriv en rekursiv funktion (dvs. utan iteration) `divide(t, n)` som beräknar hur många gånger `n` går i `t` (dvs. en *heltalsdivision*) *utan* att använda division eller multiplikation. Du kan förutsätta att `t` och `n` är större än 0.

Övning 4: Harmoniska summan. Obligatorisk!

Skriv en rekursiv funktion `harmonic(n)` som beräknar den harmoniska summan $h(n)$ definierad som

$$h(n) = 1 + 1/2 + 1/3 + \cdots + 1/n$$

där `n` är ett positivt heltal.

Övning 5: Siffersumman

Skriv funktionen `digit_sum(x)` som returnerar summan av siffrorna i den digitala representationen av heltalet `x`.

Exempel:
`digit_sum(2301)` ska returnera 6
`digit_sum(101207)` ska returnera 11.

Övning 6: Binär representation av heltal.

Skriv en rekursiv funktion `get_binary(x)` som returnerar den binära representationen av heltalet `x` som en sträng.

Exempel:
`get_binary(5)` ska returnera '101'
`get_binary(15)` ska returnera '1111'
`get_binary(-18)` ska returnera '-10010'

Tips: Det blir lite lättare om man låter `get_binary(0)` returnera en tom sträng, inte `'0'`.

I ovanstående exempel blir de rekursiva metoderna varken enklare eller effektivare än den iterativa men rekursiva resonemang är ett ändå kraftfullt sätt att lösa problem och hitta effektiva algoritmer.

Att uttrycka sig rekursivt är ofta naturligt i matematiken. Deriveringsreglerna (”derivatan av en summa är summan av derivatorna”) är ett exempel på detta. Inte desto mindre brukar rekursion betraktas som svårt när man börjar med det i programmering. Orsaken till detta är säkert att man i de flesta fall lärt sig programmera med iterationer.

Frågor att besvara vid konstruktion av en rekursiv algoritm:

1. Hur kan jag dela upp ursprungsproblemet i mindre problem av samma slag?
2. Hur kombinerar jag lösningarna till delproblemen till en lösning på ursprungsproblemet?
3. Vilka rekursionsterminerande fall är lämpliga? Kommer man alltid till ett sådant oberoende av indata?

Detta är mycket besläktat med induktionsbevis: Vi antar att vi kan lösa problemet för ett eller flera mindre problem. Sedan visar man hur man med hjälp av dessa lösningar kan lösa ursprungsproblemet. Precis som i induktionsbevis så måste man ha minst ett basfall.

Exempel 3: Skapa en omvänd sträng

Antag att vi vill skriva en funktion som skriver ut en sträng där tecknen kommer i omvänd ordning.

Hur definiera problemet i termer av sig självt?

Induktionsantagande: Antag att vi kan lösa problemet för de $n-1$ första tecknen i strängen.

Basfall: Att skriva ut en sträng med *ett* tecken i omvänd ordning. Trivialt.

Eftersom det sista tecknet skall vara först måste vi börja med att skriva det.

```
1  def reverse(s):
2    if len(s)<=1:
3        print(s, end='')
4    else:
5        print(s[-1])      # Print the last character
6        reverse(s[:-1])   # All but the last character in reverse order
```

Övning 7: Alternativ `reverse`.

Skriv en rekursiv funktion `reverse(s)` som *returnerar en sträng* (alltså inte skriver ut den) där tecknen kommer i omvänd ordning mot tecknen i `s`. Använd antagandet att vi kan lösa problemet för ut de $n-1$ *sista* tecknen i strängen.

Man behöver inte dela vid första eller sista tecknet — det går bra att dela strängen var som helst, t.ex. i mitten:

```
1 def reverse(x):
2     if len(x) <= 1:
3         return x
4     else:
5         mid = len(x)//2
6         return reverse(x[mid:]) + reverse(x[:mid])
```

På rad 6 ser vi att funktionen anropar sig själv två gånger. Man säger då att programmet är *trädrekursivt*. I detta fall spelar det ingen större roll vilket uppdelning vi väljer men vi i diskussionen om algoritmanalys kommer vi se att valet av delproblem kan vara avgörande för algoritmens effektivitet.

Övning 8: Funktionen `largest`. Obligatorisk!

Skriv en rekursiv funktion `largest(a)` som returnerar det största värdet av elementen i listan `a`. Du kan anta att listan inte är tom och att den bara innehåller jämförbara element (t.ex. alla tal eller alla strängar). I denna uppgift får du inte använda metoden `max`!

Övning 9: Funktionen `count`. Obligatorisk!

Skriv en rekursiv funktion `count(x, s)` som räknar och returnerar hur många gånger `x` finns på högsta (yttersta) nivån i listan `s`.

Exempel: Anropet `count(4, [1, [4, 4], 3, 1, 4, 2, ['a', [[4, 4], 4, 4]]])` ska returnera 1.

Modifiera sedan koden så att den räknar förekomster på *alla* nivåer. Anropet ovan ska då returnera 7.

Tips: Uttrycket `type(x) == list` returnerar `True` om x är en lista, annars `False`.

Övning 10: Rekursiv zip. Obligatorisk!

Skriv en rekursiv funktion `zippa(l1, l2)` som returnerar en lista med vartannat element från listan `l1` och vartannat från listan `l2`.

Exempel: Anropet
`zippa([1, 3, 5], [2, 4, 6, 8, 10])`
ska returnera listan
`[1, 2, 3, 4, 5, 6, 8, 10]`

Obs: Du ska *inte* använda den inbyggda metoden `zip`!

Exempel 4: Trava brickor

En mängd med n brickor av olika storlek ligger travade på en plats (f) i storleksordning med den största underst. Problemet går ut på att flytta hela traven till en annan plats (t) under iakttagande av följande regler:

1. Endast en bricka får flyttas per gång.
2. En större bricka får aldrig läggas på en mindre.

Till hjälp har man ytterligare en plats (h) som får användas för mellanlagring.

Induktionsantagande: Vi kan lösa problemet med $n-1$ brickor.

Basfall: Flytta 0 brickor. Trivialt – gör ingenting!

Induktionssteg:
1. Flytta de $n-1$ översta brickorna till h.
2. Flytta den kvarvarande från f till t.
3. Flytta de $n-1$ brickorna på h till t.

Problemet löses således rekursivt genom att lösa två problem av storlek $n-1$.

Övning 11: Brickleken. Obligatorisk!

Skriv en funktion `def bricklek (f, t, h, n)` som returnerar en lista med instruktioner om hur brickorna ska flyttas. Parametrarna `f`, `t` och `h` är strängar med som identifierar de olika platserna och `n` är antalet brickor.

Anropet `bricklek('f', 't', 'h', 2)` ska returnera listan `['f->h', 'f->t', 'h->t']` och ska tolkas som "*flytta från `f` till `h`, flytta från `f` till `t`, flytta från `h` till `t`*".

Varje element i listan är alltså en sträng som anger varifrån och vart. Eftersom det alltid är den översta brickan som ska flyttas behövs ingen annan information.

Det är viktigt att strängarna ser ut exakt som i exemplet och inte innehåller några blanktecken!

Tips: Funktionen behöver inte mer än 5 rader, inklusive `def`-raden!

Exempel 5: Växlingsproblemet

På hur många sätt kan man växla a kronor i 1, 5, 10, 50 och 100-mynt (sedlar)? (t.ex. 90 kronor i $50 + 4 \times 10$, 9×10, $8 \times 10 + 10 \times 1$ etc.)

Formulera en lösning av problemet i termer av sig självt. Viktigt att rekursionsfallet/fallen löser ett i någon mening mindre problem: Ordna myntsorterna i någon ordning.

Dela in växlingsförsöken i två grupper:

- de som *inte använder* ett mynt av någon sort, t.ex. den första
- de som *använder* första sortens mynt

Problemets lösning kan nu formuleras:

Antalet sätt att växla a kronor vid användande av n olika sorters mynt är

1. antalet sätt att växla a kronor vid användande av alla utom den första sortens mynt *plus*
2. antalet sätt att växla $a - d$ kronor användande alla sorters mynt där d är första myntsortens valör.

Delproblem 1 är mindre än ursprungsproblemet eftersom det använder färre myntslag och delproblem 2 är mindre eftersom det växlar en mindre summa.

Antag att myntsorterna representeras i en lista `coins`.

Ger grundprogram:

```
1 def exchange(a, coins):
2     return exchange(a, coins[1:]) + \
3             exchange(a-coins[0], coins)
```

Vilka specialfall behövs för att undvika oändlig rekursion?

- $a = 0$ vilket innebär ett lyckat försök (räkna det)
- $a < 0$ vilket innebär ett misslyckat försök (räkna ej)
- listan `coins` kan bli tom vilket innebär ett misslyckat försök (räkna ej)

Slutlig version:

```
1 def exchange(a, coins):
2     if a == 0:
3         return 1
4     elif (a < 0) or len(coins) == 0:
5         return 0
6     else:
7         return exchange(a, coins[1:]) + \
8                exchange(a-coins[0], coins)
```

Övning 12:

Testkör programmet ovan. På hur många sätt kan man växla 100 respektive 1000 kronor givet mynten 1, 5, 10, 50, 100?

Rekursionsdjup

I vissa av ovanstående och kommande uppgifter och exempel kan man få problem med det så kallade *rekursiondjupet*. Exempel på funktioner där detta kan uppstå är `exchange`, `zippa` och `largest`. Det betyder att det minne som är reserverat för att hantera funktionsanrop tar slut. Generellt sett kan man säg att rekursera över mycket långa listor kan vara problematiskt.

Vi poängterar igen att många av uppgifterna ovan är till för att öva rekursivt tänkande.

Övning 13:

Undersök hur stort värde på n den rekursiva funktionen för fakultetsberäkning (`fac`) din dator kan hantera!

Algoritmanalys

I algoritmanalysen studerar man hur tiden för en algoritm beror av problemstorleken. Av tradition kallar man detta för algoritmens *komplexitet* vilket alltså är ett uttryck för algoritmens effektivitet som inte har något att göra med hur svår algoritmen är att förstå eller implementera.

Resultaten är av typen ”tiden för denna algoritm växer proportionellt mot kvadraten på problemstorleken”.

I stället för att uttryck sig i tid kan man tala om hur många gånger någon viss operation utförs.

Exempel 6: Beräkning av x^n igen

I kapitlet om rekursion presenterade vi två olika sätt att beräkna x^n.

Det första gjordes iterativt med en for-sats:

```
1 def power(x,n):
2     p=1
3     for i in range(n):
4         p=p*x
5     return p
```

Denna funktion utför uppenbarligen n multiplikationer. Om x är ett vanligt tal så tar alla multiplikationer lika lång tid oavsett värdet på x. Det betyder att vi kan räkna med att tiden för denna funktion växer linjärt med n. Man brukar då säga att det är en *komplexiteten* är $\mathcal{O}(n)$ (uttalas ”Ordo” eller, på engelska ”Big Oh”)) eller $\Theta(n)$ (uttalas ”Theta”). Vi

återkommer längre fram till de exakta definitionerna av dessa termer.

I det andra sättet skrev vi en rekursiv funktion:

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

Den här funktionen kommer att göra n funktionsanrop och n multiplikationer så även denna har komplexiteten $\Theta(n)$.

En annan möjlig definition är:

$$x^n = \begin{cases} 1 & \text{om } n = 0 \\ (x^{n/2})^2 & \text{om } n > 0 \text{ och } n \text{ jämn} \\ x \cdot (x^{n/2})^2 & \text{om } n > 0 \text{ och } n \text{ udda} \end{cases} \quad (1)$$

Även denna definition är rekursiv, men i stället för att återföra lösningen av ett problem av storleken n på ett problem av storleken $n-1$, så använder vi oss av problem av storleken $n/2$, dvs. ett betydligt mindre problem.

```
def power(x, n):
    if (n == 0):
        return 1.0
    else:
        p = power(x, n//2)  # Integer division
        if (n % 2 == 0):    # n even
            return  p*p
        else:               # n odd
            return  x*p*p
```

Anropet `power(x,1000)` kommer att generera följande sekvens av anrop:

```
power(x,1000) -> power(x,500) -> power(x,250) -> power(x,125) ->
power(x,62)   -> power(x,31)  -> power(x,15)  -> power(x,7)   ->
power(x,3)    -> power(x,1)   -> power(x,0)
```

Här kommer alltså resultatet beräknas med 10 funktionsanrop och och 15 multiplikationer (10 kvadreringar och 5 extra multiplikationer när argumentet är udda). Detta är ju en betydande förbättring över det två första implementationerna.

Eftersom problemet halveras varje rekursionssteg så blir det generellt ungefär $\log_2(n)$ anrop (i själva verket $\lfloor \log_2(n) \rfloor + 1$ men vi gör det lite enklare för oss) och lika många kvadreringar plus högst lika många extra multiplikationer för att hantera udda argument. Tiden bör alltså växa proportionellt mot $\log(n)$. Man säger då att algoritmen har komplexiteten $\Theta(\log(n))$

Övning 14:* Effektiv `power` utan rekursion.

Hur kan man implementera denna effektiva algoritm *utan* rekursion?

Asymptotisk notation

Man använder Ordo- och Theta-uttrycken för att enkelt beskriva den *väsentliga* tillväxten utan att behöva räkna ut exakt hur uttrycken ser ut.

Formellt

En (matematisk) funktion $t(n)$ är $\mathcal{O}(f(n))$ om det existerar två konstanter c och n_0 sådana att:

$$|t(n)| \leq c \cdot |f(n)| \qquad \forall n > n_0 \tag{2}$$

Eftersom $\mathcal{O}$ betecknar en övre gräns, är en algoritm som är $\mathcal{O}(\log(n))$ också $\mathcal{O}(n)$, $\mathcal{O}(n^2)$, $\mathcal{O}(2^n)$ och så vidare.

Tecknet Ω används för att ange en undre gräns: En funktion $t(n)$ är $\Omega(f(n))$ om det existerar två konstanter c och n_0 sådana att:

$$|t(n)| \geq c \cdot |f(n)| \qquad \forall n > n_0 \tag{3}$$

Om funktionen $t(n)$ är både $\mathcal{O}(f(n))$ och $\Omega(f(n))$ sägs den vara $\Theta(f(n))$. (Naturligtvis måste man använda olika konstanter c för den övre och den undre gränsen.)

Ofta säger man $\mathcal{O}$ när man egentligen menar Θ.

Notera att det inte spelar någon roll vilken logaritm man använder i $\mathcal{O}$-, Θ- och Ω-uttrycken, eftersom

$$\log_a(x) = \frac{1}{\log_b(a)} \cdot \log_b(x)$$

och $\frac{1}{\log_b(a)}$ kan bakas in i c. Därför behöver man heller inte ange vilken logaritm som avses.

Fler exempel

I detta avsnitt ska vi analysera några algoritmer som du antagligen redan känner till.

Exempel 7: Indexering i Python-listor

Antag att du har en lista som du ska hämta ett element från en given position (första, nittonde, sista, ...). Detta är ett exempel på när tidsåtgången inte växer med problemstorleken. Oavsett hur lång listan är kommer det att krävas lika många operationer för att hämta elementet. Indexering i listor är alltså $\Theta(1)$ eller $\mathcal{O}(1)$ vilket är samma sak.

Observera att andra programmeringsspråk kan ha andra sätt att implementera listor som då kan ge att indexering kräver $\Theta(n)$ operationer.

Exempel 8: Sökning i osorterad lista

När det gäller sökning brukar man vid algoritmanalys skilja på *lyckad* och *misslyckad* sökning.

Om värdet vi letar efter inte finns i listan, säger vi att sökningen misslyckas. En misslyckad sökning i en osorterad lista kräver att vi kontrollerar varje element i listan. (Hur ska vi annars kunna vara säkra på att värdet inte finns?) Antalet operationer växer alltså linjärt med problemstorleken (dvs. listans längd) och vi säger att en misslyckad sökning i en osorterad lista är $\Theta(n)$.

Om det eftersökta värdet finns i listan, säger vi att sökningen lyckas. Här är det relevant att skilja på bästa fall, värsta fall och genomsnitt. I *värsta fall* behöver vi leta igenom hela listan innan vi hittar det eftersökta värdet. Under förutsättning att alla värden är lika sannolika att bli sökta behöver vi i *genomsnitt* leta igenom halva listor för att hitta det eftersökta värdet. I båda dessa fall växer antalet operationer linjärt med problemstorleken. I *bästa fall* hittar vi det eftersökta värdet på det första stället vi letar på. Då behövs ett konstant antal operationer — det spelar ingen roll hur lång listan är. En lyckad sökning i en osorterad lista är alltså:

- $\Theta(1)$ i bästa fall
- $\Theta(n)$ i värsta fall
- $\Theta(n)$ i genomsnitt (men med en mindre konstant än i värsta fallet)

Generellt är man oftast intresserad av en algoritms komplexitet i värsta fall och/eller i genomsnitt.

Observera att koden på rad 7 respektive 10 i vidstående exempel båda har komplexiteten $\Theta(n)$ (där n är listans längd) men den på rad 10 är både snabbare och elegantare!

Att en operation är inbyggd i språket ändrar inte på komplexiteten även om den ofta väsentligt påverkar tiden för operationen.

```
1  def search(x, lst)
2      for i = range(len(lst))
3          if x == lst[i]:
4              return True
5      return False
6
7  if search(x, mylist):
8      do_something()
9
10 if x in mylist:
11     do_something()
```

Exempel 9: Sökning i sorterad lista

Om listan som vi ska söka i är sorterad i storleksordning, minskar mängden arbete avsevärt. I detta exempel kallar vi listan för `a` och det eftersökta värdet för `v`. Vi antar att elementen i `a` är sorterade i stigande ordning, dvs. så att `a[i+1]` $\geq$ `a[i]` för alla index `i`.

En sökning går då till så här:

1. Om listan är tom har sökningen misslyckats. `v` kan ju omöjligt finnas i en tom lista.
2. Om `a[n//2] == v` är vi klara.
3. Annars, upprepa proceduren, den här gången med en hälften så lång lista:
   - Om `a[n//2] < v`, ge `a[n//2+1:n]` (dvs. den delen av `a` som ligger till höger om det $n/2$:te elementet) som input.
   - Om `a[n//2] > v`, ge `a[0:n//2-1]` (dvs. den delen av `a` som ligger till vänster om det $n/2$:te elementet) som input.

På det här sättet halverar vi problemstorleken i varje steg. Jämför med hur du själv skulle leta i en mängd sorterade element, exempelvis en telefonkatalog, en klasslista eller registret i en bok!

Vid en misslyckad sökning behöver vi fortsätta tills listan är tom. Om vi skulle dubblera a:s längd, skulle antalet nödvändiga operationer öka med 1. Med andra ord växer mängden arbete logaritmiskt; algoritmen är $\Theta(\log(n))$.

Vid en lyckad sökning behöver vi utföra samma procedur, men kommer i genomsnitt att hitta `v` efter halva tiden. Mängden arbete växer alltså som $0,5 \cdot \log(n)$, men $0,5$ är bara en konstant och kan bakas in i c. Algoritmen är alltså $\Theta(\log(n))$ i genomsnitt.

Övning 15:

> Hur många operationer kräver en lyckad sökning i en sorterad lista i bästa respektive värsta fall?

Exempel 10: Hashtabeller, lexikon

Genom att sortera posterna i storleksordning går att det alltså att att minska komplexiteten för sökning från $\Theta(n)$ till $\Theta(\log n)$ vilket är en dramatisk förbättring för stora mängder. Det går dock att göra det ännu bättre genom så kallade *hashtabeller*. Python använder denna teknik för att implementera lexikon (”dictionaries”). I sådana är tiden för sökoperationen oberoende av storleken på mängden som man skriver $\Theta(1)$ (eller $\mathcal{O}(1)$ vilket är samma sak).

Vi återkommer till hashtabeller i ett senare avsnitt i kursen som behandlar datastrukturer.

Exempel 11: Urvalssortering

Bredvidstående kod implementerar en enkel sorteringsalgoritm som kallas *urvalssortering*.
(Observera den "pythonska" konstruktionen på rad 7 för att byta innehåll på två variabler!)

```
1 def selection_sort(a):
2     for i in range(len(a)-1):
3         k = i
4         for j in range(i+1, len(a)):
5             if a[j] < a[k]:
6                 k = j
7         a[i], a[k] = a[k], a[i]
```

Om vi betecknar längden av `a` med n så ser vi att den första `for`-satsen exekverar $n-1$ gånger med i-värdena $0, 1, 2, \ldots, n-2$. Den inre loopen exekverar första gången $n-1$ gånger, andra gången $n-2$ gånger etc. Jämförelsen på på rad 5 kommer således utföras

$$(n-1) + (n-2) + (n-3) + \cdots + 2 + 1 = \frac{n(n-1)}{2}$$

som är $\Theta(n^2)$ (övning: bevisa det!).

Tiden för koden på raderna 5 - 6 betraktar vi som konstant vilket gör detta till en typisk $\Theta(n^2)$ –metod.

Exempel 12: Analys av brickleken

Här löser vi ett problem av storleken n med hjälp av två problem av storleken $n-1$ plus en brickförflyttning. Om vi låter $t(n)$ stå för antalet brickförflyttningar så gäller alltså

$$t(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + 2 \cdot t(n-1) & \text{if } n > 1 \end{cases}$$

This difference equation can be easily solved by telescoping", i.e. that we apply the equation over and over again:

$$\begin{aligned} t(n) &= 1 + 2 \cdot t(n-1) = 1 + 2 \cdot [1 + 2 \cdot t(n-2)] \\ &= 1 + 2 + 4 \cdot t(n-2) = 1 + 2 + 4 \cdot [1 + 2 \cdot t(n-3)] \ldots \\ &= 1 + 2 + 4 + \ldots + 2^k \cdot t(n-k) \\ &= \sum_{i=0}^{n-1} 2^i = 2^n - 1 \end{aligned}$$

Övning 16: Tid för brickleken. Obligatorisk!

Antag att du har en hög med 50 brickor och det tar 1 sek att flytta en bricka. Hur lång tid kommer det ta för dig att genomföra hela förflyttningen?

Exempel 13: Fibonaccitalen

Fibonacci-talen är en följd av tal som börjar med 0 och 1 och därefter är varje tal summan av de två närmast föregående talen: 0, 1, 1, 2, 3, 5, 8, 13, 21 ...

Så här kan en funktion för att beräkna det n:te talet (vi numrerar dem med början på 0) se ut:

```
1 def fib(n):
2     if n==0:
3         return 0
4     elif n==1:
5         return 1
6     else:
7         return fib(n-1) + fib(n-2)
```

Denna algoritm är lite svårare att analysera.

I stället för att titta på *tid* så undersöker vi hur *många additioner* som anropet `fib(n)` ger upphov till.

Om n är 0 eller 1 görs inga additioner. Om n är större än 1 så görs 1 addition *plus* antalet additioner som görs av anropen `fib(n-1)` och `fib(n-2)`.

Om vi låter $t(n)$ beteckna antalet additioner för som utförs av anropet `fib(n)` så gäller alltså:

$$t(n) = \begin{cases} 0 & \text{om } n \leq 1, \\ 1 + t(n-1) + t(n-2) & \text{om } n > 1. \end{cases} \tag{4}$$

Observera likheten med Fibonaccitalen!

Detta är en linjär differensekvation. Vi ger lösningen nedan men vi börjar med att göra en enklare uppskattning.

Eftersom $t(n)$ måste vara en växande funktion så kan vi uppskatta så här för $n > 1$:

$$t(n) = 1 + t(n-1) + t(n-2) < 1 + 2 \cdot t(n-1)$$

Vi löste detta som som en *ekvation* vid analys av brickleken ovan och konstaterade att lösningen till denna var $2^n - 1$. Eftersom detta är en *överskattning* kan vi konstatera att antalet additioner och därmed även tiden inte växer snabbare än $\mathcal{O}(2^n)$. Om du svarade på frågan om tiden för 50 brickor inser du att det man inte blir särskilt lycklig av att veta att det inte är värre än så.

Vi skulle kunna genomföra en uppskattning nedåt på ett liknande sätt:

$$t(n) = 1 + t(n-1) + t(n-2) > 1 + 2 \cdot t(n-2) \tag{5}$$

(lämnas som frivillig övning) och då komma fram till att $t(n)$ är $\Omega((\sqrt{2})^n)$ och därmed inse att antalet additioner växer snabbare än $(1.4142)^n$.

Differensekvationen (4) går att lösa exakt och ger $t(n) \approx 1.618^n$ Vi ger beviset här men

det ingår inte i kursen.

Bevis*

Den karaktäristiska ekvationen för den homogena ekvationen är

$$r^2 - r - 1 = 0,$$

som har lösningen

$$r_{1,2} = \frac{1 \pm \sqrt{5}}{2},$$

dvs. den homogena ekvationen har lösningen

$$F(n) = ar_1^n + br_2^n,$$

där a och b bestämmes ur begynnelsevillkoren.

Eftersom

$$t(n) = -1$$

är en partikulärlösning, kan den allmänna lösningen skrivas

$$t(n) = ar_1^n + br_2^n - 1.$$

Med hjälp av begynnelsevillkoren kan a och b bestämmas och ger lösningen

$$\begin{aligned} a &= (1 - r_2)/(r_1 - r_2) \\ b &= -(1 - r_1)/(r_1 - r_2) \end{aligned}$$

Eftersom $r_1 \approx 1.618$ och $r_2 \approx 0.618 < 1$ så ser man att, för stora n, gäller

$$t(n) \approx 1.618^n$$

Antalet additioner växer således exponentiellt och därmed växer också tiden $t(n)$ exponentiellt.

QED

Övning 17: Tid för Fibonacci. Obligatorisk!

a) Verifiera genom testkörning att tiden för den givna algoritmen växer som $\Theta(1.618^n)$

b) Tag reda på hur lång tid anropen `fib(50)` respektive `fib(100)` tar (skulle ta) på din dator. Svara med lämpliga enheter! (Sekunder är inte en lämplig enhet om det tar flera timmar. Timmar är inte en lämplig enhet om det tar flera dagar eller år.)

Obs: Du ska inte försöka optimera koden utan använda den givna funktionen!

Tips 1: För att mäta tid i ett Python-program kan man använda sig av funktionen `perf_counter` i modulen `time`. Exempel:

```
import time
tstart = time.perf_counter()
   # code to be timed
tstop = time.perf_counter()
print(f"Measured time: {tstop-tstart} seconds")
```

Tips 2: Du kommer inte kunna provköra `fib(100)` utan du måste uppskatta tiden genom att räkna!

Programmet är *trädrekursivt*, det vill säga varje anrop resulterar i två nya anrop. Detta kan potentiellt ge orimliga exekveringstider.

Problemet med koden är att vi kommer lösa samma problem många gånger (av utrymmesskäl skriver vi f i stället för `fib`) :

$$\begin{aligned} f(5) &= f(4) + f(3) \\ &= (f(3) + f(2)) + (f(2) + f(1)) \\ &= ((f(2) + f(1)) + (f(1) + f(0)) + ((f(1) + f(0)) + 1 \\ &= f(1) + f(0) + 1 + 1 + 0 + 1 + 0 + 1 \\ &= 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 = 5 \end{aligned}$$

Vi ser alltså att f(4) anropas 1 gång, f(3) anropas 2 gånger, f(2) anropas 3 gånger, f(1) anropas 5 gånger och f(0) 3 gånger.

Det vanliga sättet att beräkna Fibonacci-tal är att i stället börja nerifrån och successivt beräkna f(0), f(1), ... f(n) vilket då görs med $\Theta(n)$ additioner.

Vi ska emellertid här passa på att introducera en annan teknik som kallas *memoirering* (eng. *memoization* — ett konstigt ord men det är så det heter och det oklart vad det ska översättas med) eller, ibland, *dynamisk programmering*. Problemet med den rekursiva lösningen är ju att man löser identiska problem många gånger. Memoirering innebär att man, när man löst ett visst delproblem, sparar denna lösning och, innan man ger sig på ett nytt delproblem först kontrollerar om det redan är löst. Om det är löst så använder man sig av lösningen annars löser man det enligt algoritmen.

För Fibonaccitalen karaktäriseras problemet entydigt av talets ”nummer”, alltså parametern n i koden. Vi kan då använde ett lexikon för att spara lösningarna:

```
memory = {0:0, 1:1}

def fib_mem(n):
    if n not in memory:
        memory[n] = fib_mem(n-1) + fib_mem(n-2)
    return memory[n]
```

Övning 18: `fib_mem(100)`

Vad är Fibonaccitalet nr 100 och hur lång tid tog det att beräkna det med `fib_mem`-metoden?

Ett snyggare sätt

Funktionen `fib_mem` ovan använder en *global* variabel `memory` för att hålla reda på beräknade värden. Det vore snyggare om detta lexikon kunde döljas inuti funktionen. Det fungerar dock inte att bara flytta in satsen `memory = {0:0, 1:1}` för då kommer varje instans av funktionen skapa sin egen `memory`. Redan beräknade värden kommer då att ligga i *olika* lexikon (dock alla med namnet `memory`).

Följande kod löser dock problemet:

```
def fib(n):
    memory = {0:0, 1:1}

    def fib_mem(n):
        if n not in memory:
            memory[n] = fib_mem(n-1) + fib_mem(n-2)
        return memory[n]

    return fib_mem(n)
```

Vi har återtagit namnet `fib` för den som ska användas. Inuti den skapar vi dels lexikonet `memory` och dels hjälpfunktionen `fib_mem`. Båda dessa är lokala i `fib` men alla instanser av `fib_mem` använder samma `memory`.

Denna väsentliga förbättring av den ursprungliga `fib` kan alltså göras helt ”osynlig” för den anropande koden.

Mer om storheters *räckvidd* (eng. *scope*) finns i Corey Schafers YouTube-lektion.

Tekniken att beräkna Fibonaccital med memoirering finns till exempel beskriven i Socraticas YouTube-lektion.

Tekniken att skriva funktioner lokalt i funktioner kommer vi använda fler gånger i kursen.

Ett alternativ sätt till att använda en omslagsfunktion är att skicka `memory` som parameter:

```
def fib(n, memory = None):
    if memory is None:
        memory = {0: 0, 1: 1}
    if n not in memory:
        memory[n] = fib(n-1, memory) + fib(n-2, memory)
    return memory[n]
```

Om man anropar utan att ange `memory` kommer alltså alltså lexikonet skapas och sedan skickas med i de rekursiva anropen.

Övning 19:* Exchange med memoirering.

Om du testkörde växlingsprogrammet med stora värden på `a` och `n`, märkte du förmodligen att exekveringstiden växte snabbt. Prova t.ex. att växla beloppen 1000 och 2000 Den algoritmen är trädrekursiv men den är betydligt svårare att analysera eftersom tiden beror inte bara på *summan* som ska växlas utan både på *antalet* mynt och, i någon mån, på myntens *valör*. Även här kan man använda memoirering för att undvika att samma problem löses upprepade gånger. Man behöver dock en matris för med mynten på ena axeln och beloppen på den andra.

Skriv om `exchange`-funktionen så att den utnyttjar memoirering. Vilken tid tog det att räkna på stora summor som 1000 och 2000? Vad händer om du räknar på t.ex. 10000?

Som vi ser i avsnittet om sortering nedan, behöver trädrekursiva algoritmer inte nödvändigtvis ge orimliga exekveringstider. I själva verket är många klassiska effektiva algoritmer (t.ex. sorteringsalgoritmer och snabb Fouriertransform) trädrekursiva.

Sortering

Här presenterar och analyserar vi två rekursiva sorteringsalgoritmer.

Insticksortering

Den vanliga enkla insticksortering kan uttryckas rekursivt enligt följande:

Induktionsantagande: Vi kan sortera $n-1$ element.

Basfall: Vi kan sortera 1 element.

Induktionssteg: Stoppa in det n:te elementet bland de $n-1$ redan sorterade elementen så att sorteringen bibehålls.

```
def insertion_sort(l, n):
    if n > 1:
        insertion_sort( l, n-1 )     # sort the n-1 first
        x = l[n-1]
        i = n-2                      # move elements larger than x to the right
        while i>=0 and l[i]>x :
            l[i+1] = l[i]
            i=i-1
        l[i+1] = x                   # put in x
```

Se en visualisering på YouTube!

Analys

Tiden att sortera n tal med denna algoritm beror på hur talen är permuterade.

I *bästa fall* är talen elementen redan sorterade. Det innebär att `while`-loopen aldrig exekverar. Det blir alltså sammanlagt $n-1$ anrop som tar konstant tid dvs. $\Theta(n)$.

I *värsta fall* är talen sorterade i omvänd ordning. Varje anrop kommer då också innehålla en iteration som går $n-1$ gånger dvs. totalt

$$(n-1)+(n-2)+\ldots+2+1=\frac{(n-1)n}{2}$$

gånger dvs. $\Theta(n^2)$

I *genomsnitt* kan man tänka sig att varje element ”reser” halva i stället för hela som i värsta fallet. Summan blir då $(n-1)n/4$ vilket också är $\Theta(n^2)$.

Samsortering (”Merge sort”)

Denna algoritm är, som analysen och övningen nedan visar, väsentligt effektivare än insticksortering.

Induktionsantagande: Vi kan sortera $n/2$ element.

Induktionssteg: Dela mängden i två delar med vardera $n/2$ element, sortera dessa var för sig och sammanfoga sedan de två sorterade delarna.

Basfall: En mängd med 0 eller 1 element är sorterad.

Bredvidstående kod är avsedda att illustrera *idéen* med samsorteringen.

Det är *inte* en bra implementation för stora mängder data.

Ett problem är att hjälpmetoden `merge` rekurserar över listans längd vilket gör att ett arbetsutrymme (”stacken”) tar slut vid stora mängder data.

```
1  def merge_sort(l):
2      if len(l) <= 1:
3          return l.copy()
4      else:
5          n = len(l)//2
6          l1 = merge_sort(l[:n])
7          l2 = merge_sort(l[n:])
8          return merge(l1, l2)
9      return res
10
11 def merge(l1, l2):
12     if len(l1) == 0:
13         return l2
14     elif len(l2) == 0:
15         return l1
16     elif l1[0] <= l2[0]:
17         return [l1[0]] + merge(l1[1:], l2)
18     else:
19         return [l2[0]] + merge(l1, l2[1:])
```

Se en visualisering på YouTube!

Arbetet att sammanfoga de två sorterade delarna (hjälpfunktionen `merge` ovan) är proportionellt mot antalet element. (Uttrycket ”tiden är $\Theta(f(n))$” kan, löst uttryckt, tolkas om att tiden är proportionell mot $f(n)$ dvs. $t(n)=c\cdot f(n)$.)

Låt $t(n)$ beteckna tiden att sortera n element. Då gäller

$$t(n)=\begin{cases}c & \text{om } n=0,\\ 2\cdot t(n/2)+dn & \text{om } n>0.\end{cases}$$

Om n är en jämn 2-potens, $n = 2^k$ så gäller

$$\begin{aligned} t(n) =& 2t(n/2) + dn = 2(2t(n/4) + dn/2) + dn \\ =& 4t(n/4) + dn + dn \ldots \\ =& 2^k t(n/2^k) + dnk \\ =& nt(1) + dn \log n \end{aligned}$$

dvs. tiden är $\Theta(n \log n)$.

Övning 20: Jämförelse av sorteringsmetoder. Obligatorisk!

> Antag att instickssortering och mergesort tar lika lång tid för 1000 slumptal – säg 1 sekund. Hur lång tid tar det för respektive algoritm att sortera 10^6 tal respektive 10^9 slumptal? Svara med lämpliga enheter!

Som vi har sett i ett par olika exempel ovan, kan tidsåtgången för algoritmer snabbt bli orimligt stor om man väljer fel algoritm (eller om det inte finns någon effektiv algoritm tillgänglig). Vi vill ha algoritmer för vilka tidsåtgången växer långsamt. Tiden för att lösa ett litet problem är ofta försumbar; det är när problemstorleken växer som det börjar bli viktigt att ha en effektiv algoritm!

Som följande övning visar, ska man dock inte stirra sig blind på komplexiteten. I praktiken kan till exempel en algoritm som är $\Theta(n \cdot \log(n))$ vara minst lika effektiv som en algoritm som är $\Theta(n)$, om konstanten (c) är mycket större för den senare.

Övning 21: Teoretisk jämförelse av $\Theta(n)$ och $\Theta(n \cdot \log n)$. Obligatorisk!

> Antag att du kan välja mellan två algoritmer, A och B, för att lösa ett problem. Vi låter n beteckna antalet element i den datastruktur som algoritmerna opererar på. Du vet att algoritm A löser ett problem av storlek n på n sekunder. Tidsåtgången för algoritm B är $c \cdot n \cdot \log(n)$ sekunder, där c är en konstant. Du testkör algoritm B på din dator och finner att den tar 1 sekund på sig för att lösa ett problem när $n = 10$. Hur stort måste n vara för att algoritm A ska ta kortare tid på sig än algoritm B?

Lärdomar

Exponentiell tillväxt: Hopplöst att lösa stora problem!

Hur ska man veta om en rekursiv lösning är bra eller ej?

Om lösningen av ett problem är baserad på lösningar av två eller flera problem som nästan lika stora som ursprungsproblemet så kommer man får en exponentiell tillväxt vilket gör det omöjligt att klara stora problem. Den för koden för Fibonacci-talen liksom brickleken är exempel på sådana algoritmer.

Om delproblemen är väsentligt mindre än ursprungsproblemet så (t.ex. hälften så stora) kan det vara en effektiv algoritm. Kvadreringsalgoritmen för att beräkna x^n liksom samsorteringsalgoritmen.

I sorteringsexemplet visade det sig att basera lösningen på *två* delproblem av storlek $n/2$ var väsentligt bättre än att basera den på *ett* delproblem av storlek $n - 1$.

En annan observation är att ett $\log n$-beroende är *mycket* bättre än ett n-beroende men att $\log n$ inte är mycket sämre än konstant.