Föreläsning 5: Rekursion

Rekursion

Rekursion är ett sätt att lösa problem som går ut på att man skriver en metod som anropar sig själv.

- *Rekursiv tanke*: reducerar problemet till ett enklare problem med samma struktur
- *Basfall*: det måste finnas ett fall som inte leder till rekursivt anrop

Sifferexempel

Fråga: Hur många siffror har heltalet n?
Rekursivt svar: En siffra mer än om man stryker sista siffran i n, ...men tal mindre än tio är ensiffriga.

```
def NoOfDigits(n):
    if n < 10:  return 1                # Basfall
    else: return 1 + NoOfDigits(n/10)   # Rekursivt anrop
```

Fråga: Vilken siffersumma har heltalet n?
Rekursivt svar: Sista siffran plus siffersumman om man stryker sista siffran i n, ...men noll har siffersumman noll.

```
def SumOfDigits(n):
    if (n==0): return 0                 # Basfall
    return (n%10) + SumOfDigits(n/10)   # Rekursivt anrop
```

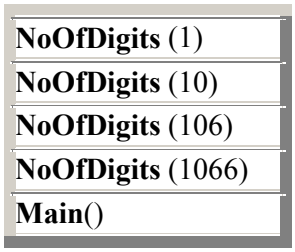
Fråga: Hur skriver man talet n binärt?
Rekursivt svar: Först skriver man n/2 (heltalsdivision) binärt och sen skriver man en nolla eller etta beroende på om n var jämnt eller udda. Avbryt då tal är 0 eller 1 som skrivs ut som de är.

```
def WriteBinary(n):
        if(n==0 or n==1): print n,      # Basfall
        else:
            WriteBinary(n/2)            # Rekursivt anrop
            print n % 2,
```

Hur fungerar det?

När man skriver egna rekursiva metoder bör man lita på att det rekursiva anropet fungerar - man behöver inte analysera anropsgången för varje fall. Men för att förstå varför rekursion kan vara extra minneskrävande är det vara bra att känna till hur Java (och andra programspråk) hanterar rekursiva anrop.

- För varje anrop skapas en *aktiveringspost* som innehåller data för anropet, t ex parametrar, lokala variabler och anropspunkt.
- Aktiveringsposten pushas på en stack.
- När det rekursiva anropet är klart poppas aktiveringsposten från stacken, varefter föregående anrop ligger överst på stacken.



Aktiveringsposterna tar extra minne i anspråk och stackhanteringen tar extra tid, så det kan löna sig att skriva om sina rekursiva metoder. En del kompilatorer kan optimera *svansrekursiva* metoder. En metod är svansrekursiv om det rekursiva anropet står sist i metoden. Då kan kompilatorn använda samma post i stacken i varje nytt anrop.

Listexempel

Vi tänker oss en länkad lista av objekt, där varje objekt innehåller ett tal och en next-pekare. Den representerar en kö och variabeln first pekar på den första posten.
Fråga: Hur lägger vi till en post i kön
Om listan inte är tom försöker vi stoppa in ett element i den lista som vår next-pekare pekar på. I basfallet är listan tom. Då returnerar vi en ny post. Observera att vi måste ta hand om returvärdet i det rekursiva anropet.

```
def insert(p, value) :
    if (p == None): return Node(value)
    else: p.next = insert(p.next, value)
    return p
```

När fungerar det dåligt?

Det finns fall där en rekursiv lösning inte lämpar sig. Ett exempel är Fibonaccitalen. Leonardo Fibonacci skrev år 1225 en bok där han beskrev denna intressanta talföljd: Sista december föds en kaninpojke och en kaninflicka. Vid två månaders ålder och varje månad därefter producerar varje kaninpar ett nytt kaninpar.

Månad	0 (nov)	1 (dec)	2 (jan)	3 (feb)	4 (mar)	5 (apr)	6 (maj)	7 (jun)	8 (jul)	9 (aug)
Kaninpar	0	1	1	2	3	5	8	13	21	34

```
def fib(n) :
    if (n <= 1): return n               # Basfall
    else: return fib(n-1) + fib(n-2)    # Rekursivt anrop
```

Om man ritar upp anropskedjan ser man att många delträd räknas ut flera gånger. För att räkna ut fib(5) så måste vi räkna ut fib(4) och fib(3). När vi räknar ut fib(4) kommer vi att ånyo räkna ut fib(3).

Antal anrop kan uppskattas med följande resonemang. Antal operationer för att räkna ut fib(n) = fib(n-1) + fib(n – 2) >= 2f(n – 2)
Alltså fib(n) >= 2f(n – 2) >= 4f(n – 4) >= 8f(n – 8) >= 16f(n – 16)
Att halvera så här kan man göra n/2 gånger och får $2^{\frac{n}{2}} = \sqrt{2}^{n}$

Om man använder två extra variabler för att spara redan beräknade värden kan man ta en for-slinga istället för rekursion vilket går i linjär tid, O(n).

```
def fib_iter(n) :
        f0=0
        f1=1
        f2=1
        for i in xrange(1, n):
            f2=f1+f0
            f0=f1
            f1=f2
        return f2
```

Några saker att minnas

- En rekursiv metod kan alltid omformuleras utan rekursion med hjälp av en stack.
- För många problem är en rekursiv metod mycket enklare att formulera och ger kortare kod än utan rekursion. Man kan gå via den rekursiva lösningen i tanken även om man gör en icke-rekursiv lösning.
- Den rekursiva metoden kräver i regel mer minne (och tar längre tid) än motsvarande icke-rekursiva metod.

Ett tentatal (27/10 2001)

Du och jag har var sin jättestack med tyska rapparskivor och vill veta vems stack som är störst. Ett sätt vore att räkna antalet skivor i varje stack, men nu gäller det att i stället uppfinna en rekursiv jämförelsemetod. Man vill att anropet compare(stack1,stack2) ska returnera -1 om stack1 är minst, 0 om stackarna är lika stora och 1 om stack2 är minst. Formulera en korrekt rekursiv tanke som omedelbart kan omsättas i fungerande programkod!
Efter anropet ska dom båda stackarna ha samma innehåll som före anropet.
Stackarna är abstrakta med metoderna push, pop och isEmpty och kan lagra vilka objekt som helst.

Rekursiv tanke: För att få reda på vilken stack som är störst poppar man ett element från vardera stacken och sparar i två lokala variabler. Nu är det bara att kolla vilken av dom återstående stackarna som är störst, men innan man returnerar svaret pushar man tillbaka dom båda poppade elementen.
. . .men om båda stackarna är tomma är svaret noll!
. . .men om ena stacken är tom är svaret 1 eller -1!

Kort om programmeringsparadigm

Det finns olika sätt att programmera. Man brukar tala om olika paradigm. Java är ett exempel på en objektorienterat programmeringsspråk. Det som skiljer dem åt är bl.a. olika mycket *tillstånd* i språken.

Funktionella språk

Rekursiva funktioner används väldigt mycket i funktionella språk. Det finns minimalt med tillstånd. Slingor skrivs ofta med hjälp av rekursion. Funktionella språk består mestadels av funktioner och anropsordningen är inte uppenbar.

Imperativa och procedurella språk

Dessa språk har funktioner men även slingor och tilldelning. Man skiljer på data och funktioner. De flesta datorers maskinkodsuppsättning är procedurell till sin natur.

Objektorienterade språk

Objektorienterade språk utökar procedurella språk genom att kombinera data och metoder. Olika tillstånd kan sparas i instansvariabler.

Multiparadigm

Python uppfyller flera paradigm. Man kan programmera imperativt, funktionellt eller objektorienterat.