Problemlösning och Algoritmer

Algoritmer

En *algoritm* är en “steg för steg” beskrivning av en strategi för att lösa ett problem. En algoritm *terminerar* alltid, dvs efter en tids arbete stannar de och då har de kommit fram till lösningen. Algoritmer behöver och bör inte presenteras som program, men de kan kodas som program (*implementeras*).

När vi i datalogiska sammanhang pratar om att lösa problem menar vi att vi vill hitta en algoritm som utför en viss uppgift. Det finns oftast många olika lösningar på samma problem och många olika sätt att hitta dem. Man kan dela upp sätten att hitta lösningar i två grupper beroende på hur man först närmar sig sitt problem.

Bottom-up innebär att man börjar med grundoperationer och enkla algoritmer och sedan arbetar sig uppåt i abstraktionsnivå.

Top-down innebär att man börjar från slutmålet, hela det stora problemet. Man delar upp problemet i mindre och mindre problem ända tills delproblemen är tillräckligt små för att man ska kunna lösa dem direkt.

I renodlad form är inget av dessa två sätt helt bra för lösning av stora problem. Bottom-up metoden är inte tillräckligt målinriktad och använder man top-down metoden är risken att man löser samma (del)problem flera gånger stor. I synnerhet om delproblem löses av olika personer i ett arbetslag. Ett vettigare arbetssätt är att börja med att fråga sig

- Vilka dataabstraktioner som behövs
- Vilka funktioner som behövs

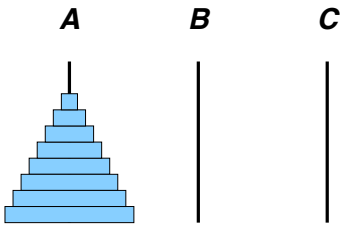
och att därefter arbeta t ex top-down.

Divide and Conquer

En metod för problemlösning är “divide and conquer”. Enligt den löses problem på följande sätt:

1. Dela upp problemet i ett antal mindre problem.
2. Lös delproblemen.
3. Kombinera delproblemens lösningar till en lösning av ursprungsproblemet.

Exempel: Hanois torn



Hanois torn är ett matematiskt pussel som introducerades av den franske matematikern Edouard Lucas redan 1883. Vi börjar med ett torn med 8 skivor av olika storlek som staplats i storleksordning med den största längst ner på en av tre pinnar.

Målet är att flytta hela tornet till en annan pinne utan att flytta mer än en skiva i taget och utan att lägga en större skiva ovanpå en mindre.

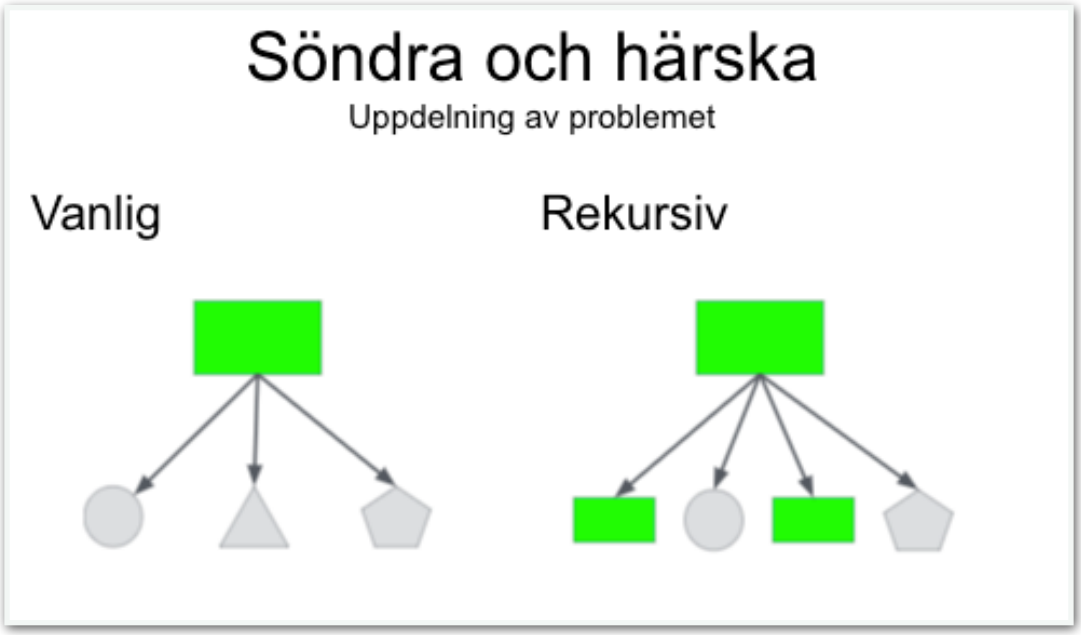
Ett sätt att lösa problemet är att arbeta enligt följande strategi för att flytta n skivor från en pinne **A** till en pinne **C** med hjälp av ``mellanlandningspinnen'' **B**.

1. Flytta tornet bestående av de $n-1$ översta skivorna från **A** till **B,** (använd **C** som mellanlandningspinne).
2. Flytta den återstående skivan från **A** till **C**.
3. Flytta tornet bestående av $n-1$ skivor från **B** till **C** (med **A** som mellanlandningspinne).

Algoritmen ovan delar alltså upp problemet att flytta n skivor i tre mindre problem, att flytta $n-1$ skivor, att flytta *en* skiva och att flytta $n-1$ skivor. De tre delproblemen kombineras på ett sådant sätt att de tillsammans resulterar i att alla n skivorna flyttats.

Lösningsstrategin i exemplet med Hanois torn är en *rekursiv algoritm* för att lösa pusslet. En rekursiv algoritm är en algoritm som “använder sig själv”, dvs rekursion, för att lösa sina delproblem. Om du tycker att algoritmen inte verkar tillräckligt detaljerad för att lösa problemet bör du provköra den. (Grafisk provkörning kan göras på http://www.cut-the-knot.org/recurrence/hanoi.shtml).

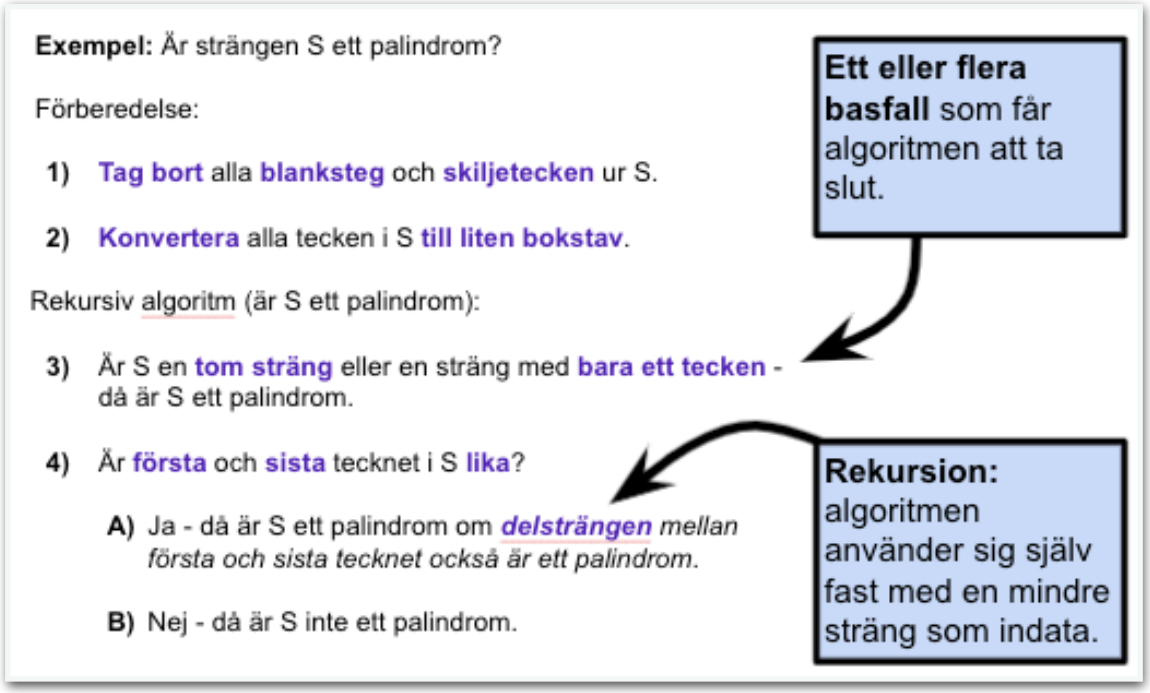
Rekursion



Rekursion är ett sätt att lösa problem som inbegriper lösning av delproblem som är av samma slag som det överordnade problemet. En rekursiv algoritm använder sig själv för att lösa dessa delproblem. För att man ska vara säker på att en rekursiv algoritm kommer att *terminera*, dvs att beräkningen blir klar någon gång, är det viktigt att två saker är uppfyllda.

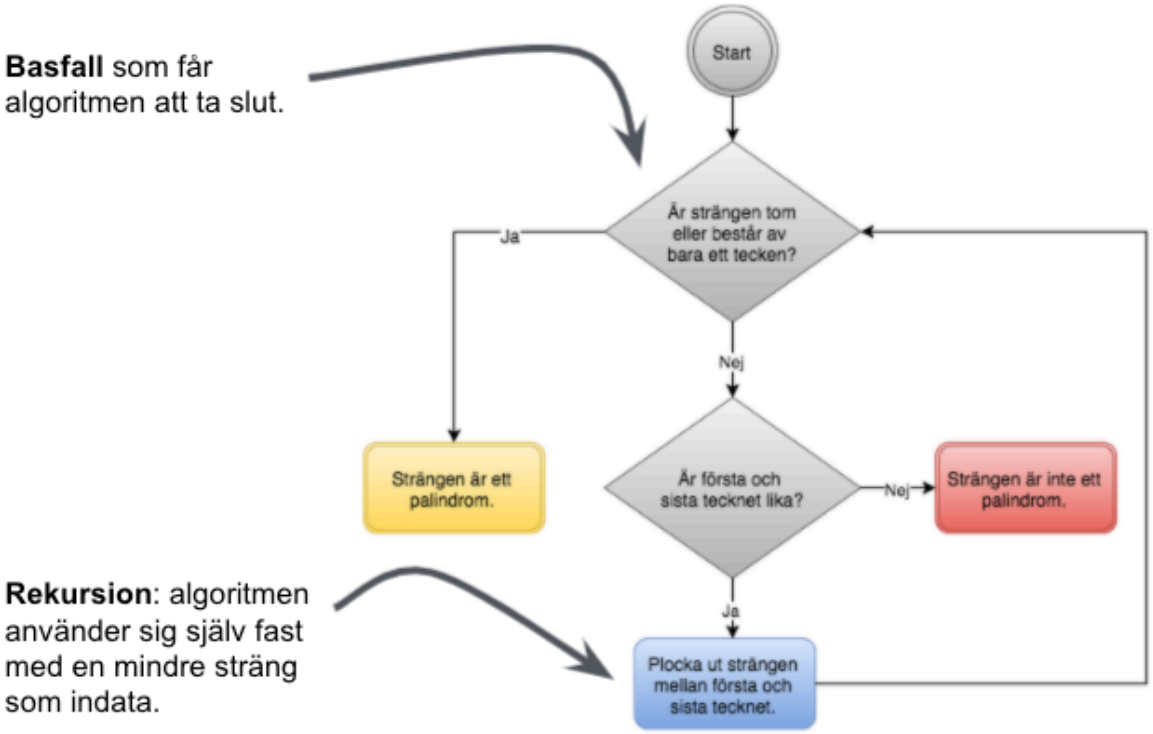
- Algoritmen måste ha ett slutvillkor, ett *basfall*
- Varje *rekursionssteg* leder mot slutvillkoret

Exempel: Palindrom



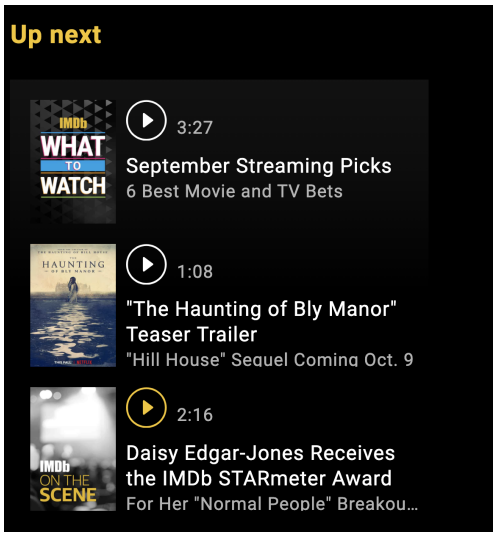
Flödesscheman

Ett vanligt sätt att beskriva algoritmer är att använda flödesscheman. I ett flödesschema beskrivs vad man gör i de olika stegen i noder och ordningen beskrivs med hjälp av pilar. Om noden innehåller en fråga kan det finnas olika vägar (pilar) ut beroende på frågans svar. Det markeras på pilarna. Bilden nedan visar algoritmen från palindromexemplet.

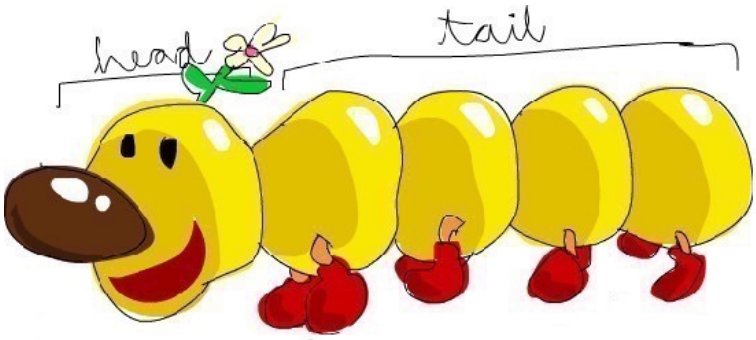


Listor

Ett sätt att samla saker i sekvens är i form av en lista. Listor används ofta i vardagslivet likaväl som i datavetenskapen.



Det första elementet i en lista kallar vi listans huvud, **head**. Det som blir kvar av listan om man hugger av huvudet kallar vi för svansen, **tail**.



Tabellen nedan visar vanlig notation som används när man arbetar med listor.

Notation	Förklaring
`[]`	En tom lista.
`Head(L)`	Det första elementet i listan **L**.
`Tail(L)`	Samma lista som **L** men där första elementet tagits bort.
`Cons(e, L)`	Den lista som uppstår då elementet **e** läggs till först i listan **L**.
`Empty(L)`	Sant om listan **L** är tom, annars falskt.

Seminarieuppgifter

Uppgifterna ska diskuteras i basgruppen.

Uppgift 1

Gruppen skall till seminariet försöka ta fram rekursiva algoritmer som löser ett antal problem där ni endast får använda de grundläggande operationerna Head, Tail, Cons och Empty på listor. Utöver

grundoperationerna för listor kan ni behöva grundoperationer för tal, t ex skapa och komma ihåg ett tal, öka värdet på ett tal, jämförelser, etc. Skapa algoritmer för följande:

- Length(L) - beräknar längden av listan L
- Count(x, L) - beräknar antalet förekomster av elementet x i listan L
- Member(x, L) - avgör om elementet x finns i listan L
- Last(L) - det sista elementet i listan L
- Append(A, B) - sätter ihop listorna A och B med A först.

Tänk särskilt på att beskriva basfall och det rekursiva steget.

Fungerar era algoritmer?
Testfölj era algoritmer steg för steg för att försäkra er om att de fungerar. Prova med några olika listor som indata.

Frivillig fördjupning 1

Beskriv algoritmer som använder grundoperationerna Head, Tail, Cons och Empty för listor för att avgöra om:
- en lista är ett palindrom
- två listor utgör ett anagram.

Frivillig fördjupning 2

Försök att algoritmiskt beskriva spelet hänga gubbe.