Problemlösning och Algoritmer

Dr. Johan Hagelbäck

johan.hagelback@lnu.se

http://aiguy.org

Problemlösning

- En datavetare kan enkelt beskrivas som en problemlösare
- För förstå datavetenskap och datalogiskt tänkande behöver vi veta vilka färdigheter och tekniker en datavetare kan använda för att lösa olika typer av problem
- Det finns många sätt för en datavetare att lösa problem, men här ska vi fokusera på fyra viktiga strategier:
  - Problembeskrivning
  - Logiskt resonemang
  - Dekomposition
  - Abstraktion

Problembeskrivning

Problembeskrivning

- Innan vi börjar utveckla ett program (mjukvara) behöver vi definiera och beskriva problemet
- Problembeskrivningen innehåller vilka funktioner mjukvaran ska innehålla
- Vi använder denna för att se när mjukvaran är färdigutvecklad
- Den kan även användas till andra saker som att mäta progress under utvecklingsfasen

Mjukvaruutveckling

- Området *programvaruteknik* (en del av datavetenskap) handlar om hur vi ska utveckla mjukvara
- Det finns många olika sätt att genomföra ett utvecklingsprojekt
- En viktig del som återfinns i samtliga modeller för mjukvaruutveckling är *analys*
- Syftet med analysen är att definiera och beskriva problemet

Analys

- Analys är, om inte den viktigaste, en väldigt viktig del av mjukvaruutveckling
- Vi måste veta vad vi ska utveckla för att slutresultatet ska bli bra
- Förutom utvecklare deltar ofta även beställare (kunder) och användare i analysfasen
- Kunden betalar utvecklaren för arbetet och är intresserad av att få valuta för pengarna
- Användare är de som ska använda produkten, vilket inte behöver vara samma som kunden

Aktörer

- De personer som är involverade i analysfasen kallar vi för *aktörer*

Utvecklare
(Bästa Konsulterna AB)

Kund
(Landstinget)

Användare
(sjukvårdspersonal)

Problembeskrivning

- Problembeskrivningen kan ses som ett kontrakt mellan utvecklare och kund
- Båda parterna kommer överens om vad som ska göras
- Problembeskrivningen består huvudsakligen av en *kravspecifikation*

Kravspecifikation

- Kravspecifikationen är en lista med *krav*
- Ett krav beskriver en liten del av programvaran
- Det finns två typer av krav:
  - Funktionella krav
  - Icke-funktionella krav
- Funktionella krav beskriver vad programvaran ska göra
- Icke-funktionella krav beskriver andra aspekter av programvaran som prestanda och säkerhet

Funktionella krav

- Ett funktionellt krav ska vara:
  - Tydligt – det ska vara enkelt att förstå kravet när vi läser beskrivningen
  - Konsekvent – det får inte finnas några tvetydigheter i beskrivningen
  - Komplett – allt som är relevant för kravet måste beskrivas så att vi inte måste fylla i med egna antaganden
- Alla krav ges ett unikt id/nummer så att vi kan referera till dem
- Som exempel ska vi definiera och beskriva några krav för en enkel videospelare

Video Spelare

ID:	R1
Namn:	Play
Beskrivning:	När användaren klickar på Play knappen (se ikon nedan) börjar videon spelas upp. Om videon är pausad, fortsätter uppspelningen där videon pausades. Om videon inte har startat, öppnas ett nytt fönster där videon börjar spelas upp från början. Om videon redan är under uppspelning har knappen ingen effekt.

Video Spelare

ID:	R2
Namn:	Paus
Beskrivning:	När användaren klickar på Paus knappen (se ikon nedan) pausas videon på det ställe där den spelas upp. Om videon inte har startats eller om videon redan är pausad har knappen ingen effekt.

Video Spelare

ID:	R3
Namn:	Öka volym
Beskrivning:	När användaren klickar på Öka volym knappen (se ikon nedan) ökas volymen med 1 steg. Om volymen redan är på max, 10 steg, har knappen ingen effekt.

Video Spelare

ID:	R4
Namn:	Minska volym
Beskrivning:	När användaren klickar på Minska volym knappen (se ikon nedan) minskas volymen med 1 steg. Om volymen redan är på 0 har knappen ingen effekt.

Kravbeskrivning

- Kraven ska vara tydliga och gå att förstå både av utvecklare, kund och användare
- Därför ska vi undvika ett alltför tekniskt språk
- Att kravet är komplett beskrivet är ofta det svåraste att uppfylla, och svårast att upptäcka om det inte är uppfyllt
- Vad händer till exempel om vi har pausat videon och stängt ner fönstret där uppspelningen visas, och sedan trycker på Play?
- Detta beskrivs inte direkt i R1 eller R2

Komplett kravlista?

- Det är inte bara beskrivningen av ett krav som måste vara komplett
- Även kravlistan behöver vara komplett så att vi vet vad som ska hända i alla situationer

- Är vår kravlista för Video Spelaren komplett?
- Nej, två grundläggande händelser saknas till exempel:
  - att starta och avsluta programmet

Video Spelare

ID:	R5
Namn:	Starta applikationen
Beskrivning:	När användaren dubbelklickar på programmets ikon (ser likadan ut som Play knappen, se bild nedan) startas programmet. När programmet har startats visas ett fönster med titeln ”Video Spelare” och tre ikoner: Play, Öka volym och Minska volym.

Video Spelare

ID:	R6
Namn:	Avsluta applikationen
Beskrivning:	När användaren klickar på Avsluta symbolen i något fönster (beror på operativsystemet, se exempel för Mac OS nedan) stannas videouppspelningen och programmet avslutas. Om en video är under uppspelning eller är pausad, ska programmet komma ihåg vilken video som spelades upp och var i uppspelningen programmet avslutades till nästa gång programmet startas.

Tillstånd-händelse tabell

- Ett sätt att kontrollera om kravlistan är komplett är att göra en tabell över alla kombinationer av tillstånd och händelser i programmet.
- Varje cell i tabellen kan antingen:
  - Ha ett krav ID, vilket anger det krav som beskriver situationen
  - Vara grå-markerad, vilket anger en situation som aldrig kan uppstå
  - Vara vit-markerad utan krav ID, vilket anger en situation som inte beskrivs i något krav
- Vi får en komplett kravlista om vi skriver krav så att inga vita celler utan krav-ID finns i tabellen

Video Spelare

	Program ej startat	Video spelas upp	Video pausad	Video ej startad	Video spelat klart
Tryck på Play		R1	R1	R1	
Tryck på Paus		R2	R2	R2	
Tryck på Öka		R3	R3	R3	R3
Tryck på Minska		R4	R4	R4	R4
Dubbel-klicka programikon	R5				
Klicka på Avsluta symbol		R6	R6	R6	R6

Tillstånd-händelse tabell

- Vi har två situationer som inte beskrivs i något krav: om videon har spelat färdigt och användaren klickar på Play eller Paus
- Vi kan antingen:
  - Skriva nytt/nya krav som täcker dessa situationer
  - Uppdatera texten i befintliga krav till att täcka dessa situationer

- Tillstånd-händelse tabeller fungerar endast för enklare program, annars blir de snabbt för svåröverskådliga

Logiskt resonemang

Logiskt resonemang

- Logik har många användningsområden, inte enbart i digitala kretsar
- Logik är en viktig del i datavetares verktyg för problemlösning
- Välskrivna krav kan till exempel översättas till logiska uttryck
- Dessa använder operatorn *medför*, vilket motsvarar om … då konstruktioner (if … then)

Logiska krav

- Kravet:
  - ”När användaren klickar på Play knappen (se ikon nedan) börjar videon spelas upp.”
- Kan översättas till följande uttryck:
  - Om play klickas på då startar videon
- Nästa mening i kravet:
  - ”Om videon är pausad, fortsätter uppspelningen där videon pausades.”
- Kan översättas till:
  - Om play klickas på och videon är pausad då fortsätter uppspelningen

Logiska instruktioner

- I programvara är det viktigt att programmet kan göra olika saker beroende på till exempel input från användaren
- Detta löser vi med if … then instruktioner:
  - if temperature > 25 then print(”It is hot!”)
  - if playClicked then playVideo
- Att använda if … then kräver att utvecklaren tänker logiskt
- Mjukvara kan ses som en samling orsak-verkan situationer

Orsak-verkan

Orsak	Verkan
Användaren har matat in användarnamn och lösenord	Verifiera att användarnamn och lösenord är giltiga
Användaren klickar på Stäng knappen på ett fönster	Fönstret stängs ner
Batteriet på telefonen är 20% eller lägre	Visa en varning att batteriet börjar bli lågt
Du blir taggad i ett inlägg på Facebook	Visa en notis att du har blivit taggad i ett inlägg

Orsak-verkan

- Orsaken beskrivs med ett logiskt villkor
- Verkan är vad som ska hända om villkoret uppfylls
- Vi ska skapa komplexa logiska villkor med logiska operatorer
  - Förutom medför, eftersom den avgränsar orsaken från verkan
- En stor del av mjukvaruutvecklarens arbete handlar om att identifiera orsak-verkan situationer och översätta dem till instruktioner

Deduktivt resonerande

- Att använda generella regler i specifika situationer kallas deduktivt resonerande
- Ett skönlitterärt exempel är när Sherlock Holmes drar slutsatsen att Watson varit på postkontoret eftersom han har röd lera på skorna och det enda stället där man måsta trampa i röd lera för att nå ingången till är just postkontoret

Deduktivt resonerande

- Deduktion används när vi använder en regel, axiom eller sats i matematiken
- Till exempel beskriver Pytagoras sats att $A^2 + B^2 = C^2$ där A och B är sidorna och C är hypotenusan i en triangel
- Uppgiften är att ta reda på hypotenusan i en triangel med sidorna 12 och 16 cm
- Vi stoppar då in våra specifika värden i den generella satsen och får att:

$$C = \sqrt{12^2 + 16^2} = \sqrt{400} = 20$$

Deduktivt resonerande

- Datavetare använder deduktion på flera sätt
- En vanlig regel i programutveckling är att ofta är det kritiskt att en sak görs före en annan
- Detta kallas en sekventiell form av exekvering
- En instruktion körs först efter att föregående instruktion är klar – instruktionerna körs i en sekvens

Sekvens

- Exempel: beräkna vad vi ska betala för en TV när vi får 20% rabatt
- Alternativ 1:

```
ordinariePris = 9900
rabatt = 20
pris = 9900 * (100 – rabatt) / 100
```


Sekvens

- Alternativ 2:

```
ordinariePris = 9900
pris = 9900 * (100 – rabatt) / 100
rabatt = 20
```

- Här blir resultatet fel, eftersom vi anger vilken rabatt som ska ges efter priser har beräknats

Algoritm

- Prisberäkningen är ett exempel på en enkel algoritm
- En algoritm är en *sekvens av otvetydiga instruktioner som löser ett problem*
- En algoritm kan ha indata (ordinariepris och rabatt i vårt exempel) och utdata (resultatet av prisberäkningen)
- Den tidigast kända, icke-triviala algoritmen anses vara Euklides algoritm
- Den hittar den största gemensamma delaren för två positiva heltal

Euklides algoritm

1. Två heltal *a* och *b*, där *a* > *b*, är givna
2. Om *b* = *0* är algoritmen klar och svaret är *a*
3. Om inte, beräknas *c* som resten när man dividerat *a* med *b*
4. Sätt *a* = *b* och *b* = *c* och börja om från steg två igen

Euklides algoritm

```
10    a = 15
20    b = 4
30    if (b != 0)
40      c = a modulus b
50      a = b
60      b = c
70      goto 30
80    print ”Största gemensamma delaren är ” + a
```


Iteration

- Det går inte att göra en generell Euklides algoritm med enbart sekvens
  - Eftersom vi inte vet hur många beräkningar vi måste göra innan b är 0 – det beror på indatan
- Vi måste även använda iteration (upprepning)
- Flera tidiga programspråk som *Basic* använde instruktionen goto [rad] för val och upprepning
- De flesta moderna språk har två konstruktioner för iteration: while och for

Iteration - while

- Euklides algoritm med while-konstruktionen för iteration:

```
10    a = 15
20    b = 4
30    while(b != 0)
40      c = a modulus b
50      a = b
60      b = c
70    print ”Största gemensamma delaren är ” + a
```

- While-blocket (rad 30 till 60) körs så länge villkoret är uppfyllt

Iteration - for

- For-konstruktionen används i stället om vi vet exakt hur många upprepningar som behöver göras
- Ett enkelt exempel är om vi ska beräkna summan av alla tal från 1 till 10:

```
10    summa = 0
20    for (tal = 1 to 10)
30      summa = summa + tal
40    print ”Summan är ” + summa
```


Val av konstruktion

- Vilka konstruktioner som ska användas (if … then, while, for) beror på problemet
- Olika problem kräver att vi kombinerar olika konstruktioner och instruktioner

Mönster

- Ett mönster (pattern) är en lösning på en återkommande mindre uppgift
- Ett exempel är att byta värden på två variabler *a* och *b*
- För att inte skriva över något av värdena måste vi göra en mellanlagring:

```
10    temp = a
20    a = b
30    b = temp
```


Standardiserad algoritm

- Många problem är återkommande och det finns standardiserade algoritmer för att lösa dessa
- Ett exempel är sökning i en lista med tal:

```
10    //Definiera lista och tal att söka efter
20    lista = {5, 12, 6, 9, 13, 7, 8}
30    tal = 6
40
50    //Algoritm för sökning
60    i = 0
70    found = false
80    while (i < lista.length)
90      if (lista[i] == tal)
100       found = true
110     i = i + 1
120   if (found == true)
130     print ”Talet ” + tal + ” finns i listan”
140   else
150     print ”Talet ” + tal + ” finns inte i listan”
```

Algoritmer och deduktion

- Algoritmer för att lösa olika vanliga problem är välanvända och vi vet att de fungerar
- Därför kan vi använda dessa i deduktivt resonerande
- Vi kombinerar och anpassar mönster och algoritmer för att lösa vårt specifika problem

Andra typer av mönster

- Mönster är inte enbart användbara för programinstruktioner
- Det finns till exempel mönster för:
  - Testning av mjukvara
  - Säkerhet för mjukvara
  - Lösningar för bättre prestanda

Specialisering

- Deduktion kan ses som en form av specialisering
- I deduktion applicerar vi vår kunskap om mer generella regler, algoritmer och mönster för att lösa ett specifikt problem
- Deduktion går alltså från det generella till det specifika

Generella  **Specifika**

Induktivt resonerande

- Det är också viktigt att datavetare hanterar induktivt resonerande
- Induktion går åt andra hållet – från det specifika till det generella
- Vi utgår ifrån en lösning på ett specifikt problem
- … och försöker se vad som kan generaliseras

Exempel

- Många program separerar data och logik från presentation och interaktion
- De kan delas upp i följande delar:
  - Presentation och kommunikation med användaren
  - Lagring och hantering av data
  - Bearbetning och hantering av händelser
- Det här är en så vanlig lösning att det finns ett mönster för hur vi bygger upp ett system utifrån dessa principer

MVC

- Mönstret kallas Model-View-Controller
- Det består av tre delar, ofta kallade *lager*
- De tre delarna är:
  - Model: lagring och hantering av data
  - View: användargränssnitt och interaktion med användaren
  - Controller: bearbetar och svara på händelser, och kan framkalla förändringar i *model* eller *view*
- MVC är ett exempel på hur en generell lösning tagits fram utifrån flera specifika lösningar

Mönster

- Erfarna utvecklare har tagit fram en mängd generella lösningar på givna problem
- Dessa kallas design- eller arkitekturmönster
- Arkitekturmönster löser problem på hög nivå – hur ett större system byggs upp av olika delar
- Designmönster löser problem på lägre nivå – hur ett problem kan lösas i programkod

Dekomposition

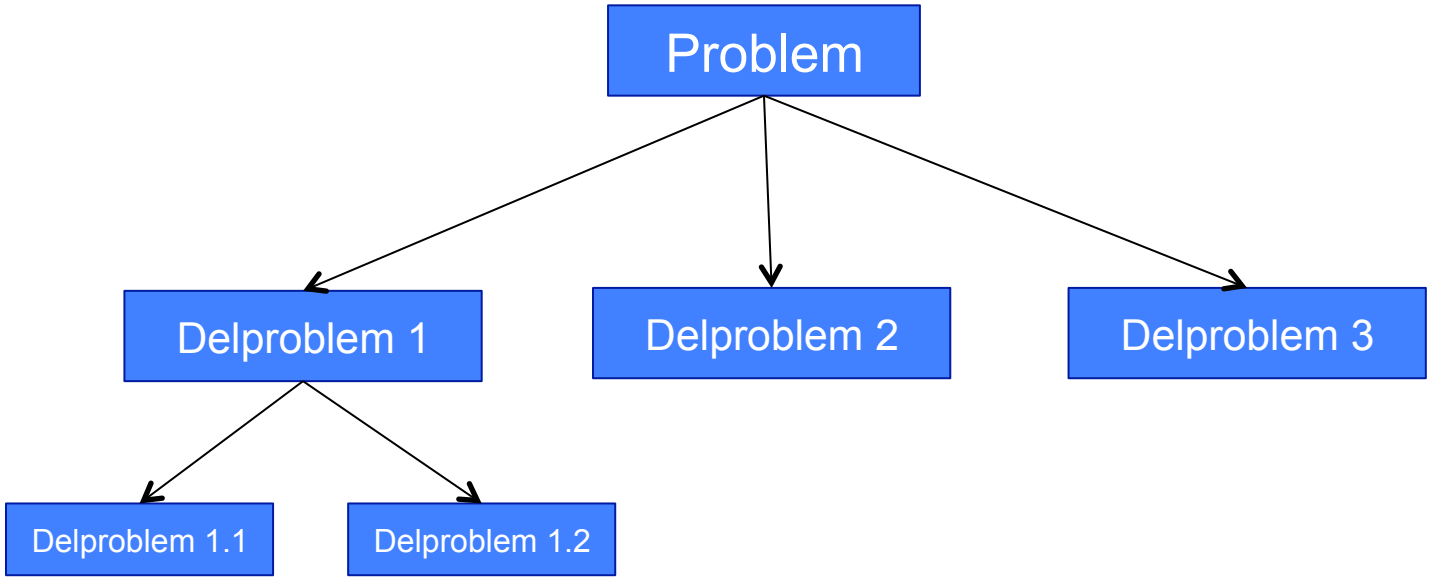
Dekomposition

- För enkla program kan vi ofta direkt få en bild av hur vi ska lösa problemet
  - Till exempel att söka igenom en lista av tal efter ett specifikt tal
- Oftast är dock problemet betydligt större och mer komplext
- Det kan då vara svårt att direkt se en komplett lösning på problemet
- En användbara strategi är dekomposition – att dela upp problemet i mindre delar

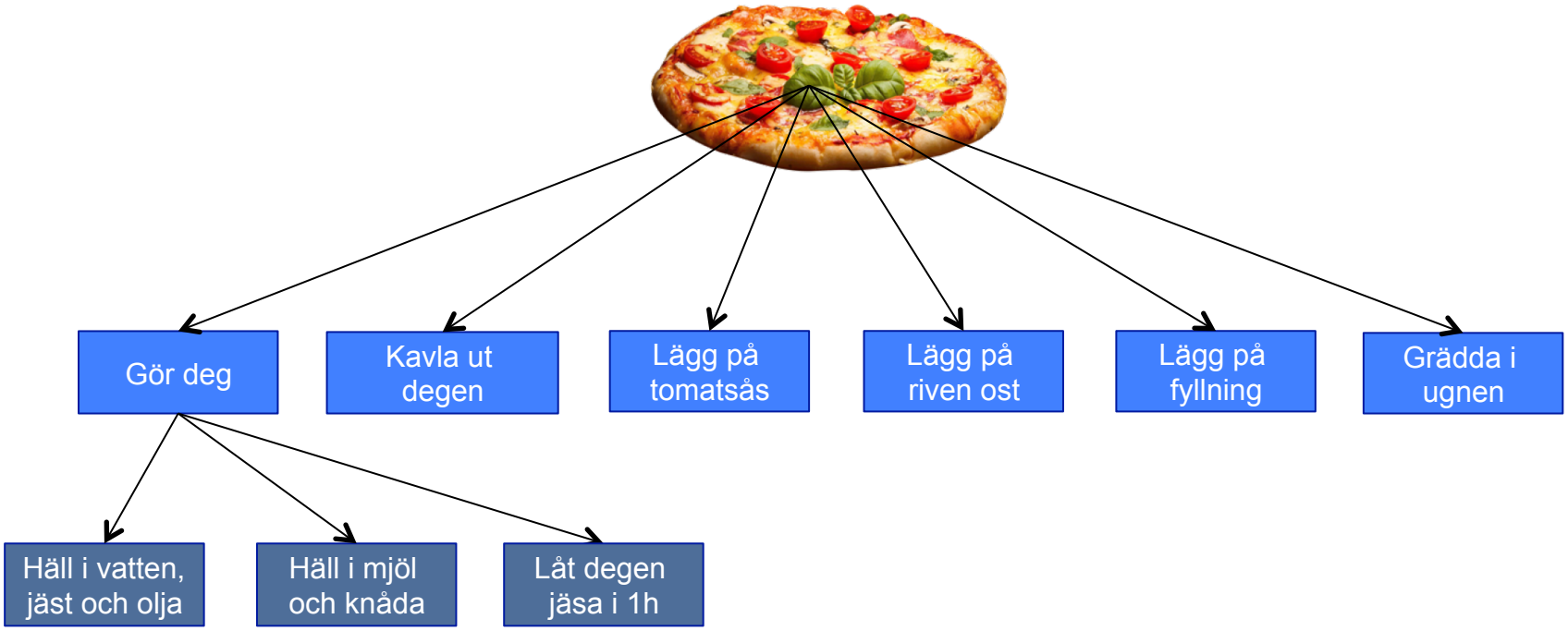
Dekomposition

- Strategier för dekomposition kallas ofta divide-and-conquer
  - söndra och härska?
- Vi tar ett större problem och delar upp det i mindre delproblem (divide), som vi sedan löser ett i taget (conquer)
- Eventuellt kan ett delproblem vara för komplext för att vi direkt ska se en lösning, och vi kan då dela upp det i mindre delproblem med samma strategi

Divide-and-conquer



Exempel: Baka pizza



Divide-and-conquer

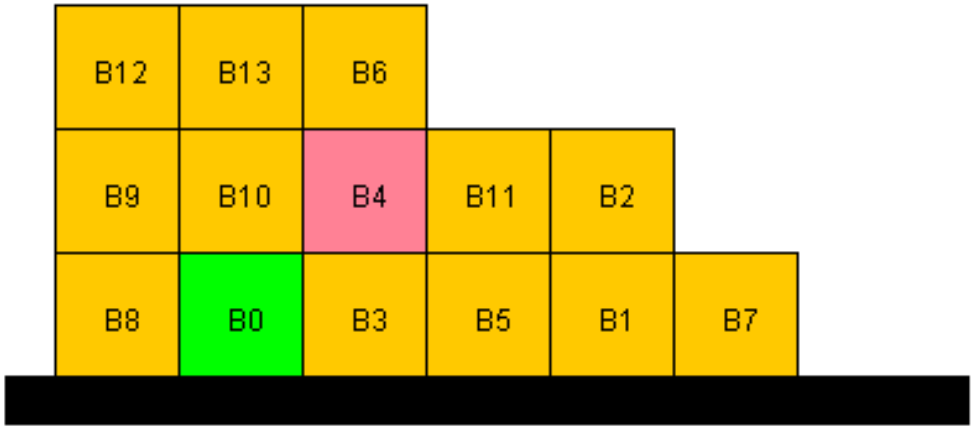
- Det tidigast kända exemplet på divide-and-conquer är från en fabel av greken Aesop runt 600 f.Kr.
- I fabeln ber fadern sin söner att bryta en bunt av pinnar ihopsatta med ett snöre
- Bunten är för tjock för att sönerna ska kunna bryta den
- Fadern berättar då för sönerna att de ska knyta upp snöret och bryta en pinne i taget, vilket dom klarar av

Divide-and-conquer

- Psykologen George Miller skrev i en artikel 1956 om en anledning till att divide-and-conquer är en bra strategi
- Miller visade att den mänskliga hjärnan är begränsad i sin kapacitet att hantera information
- Vi kan ofta inte hålla mer än 7±2 informationsenheter i huvudet i taget
  - Vad en enhet är beror lite på typen av information…
- Därför är många problem för komplexa för att vi ska kunna hantera en lösning i huvudet

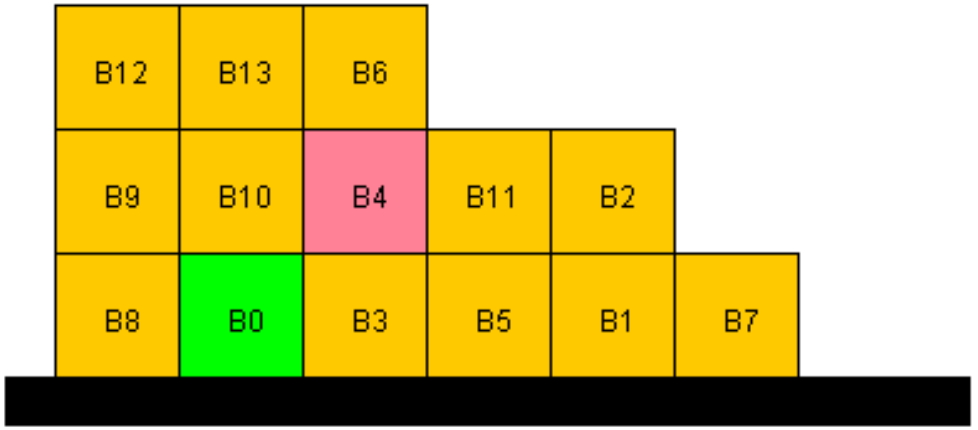
Exempel: Stapla lådor

- Vi har 14 lådor, numrerade från B0 till B13, staplade i 6 staplar
- Vi sak flytta runt så att den gröna lådan B0 står ovanpå den röda lådan B4
- Hur kan vi angripa problemet?

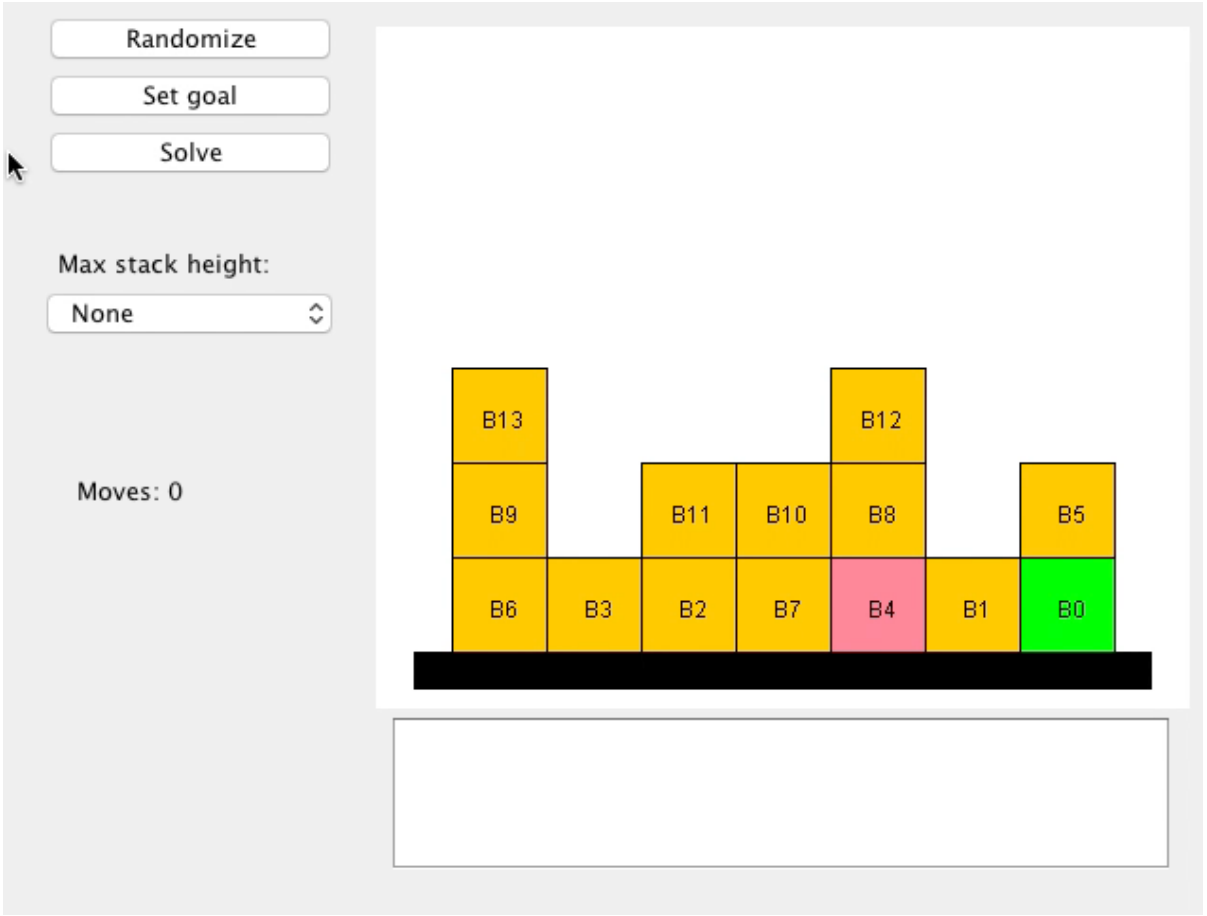


Lösning

- Flytta alla lådor som står ovanpå B0 till staplar som inte innehåller B0 eller B4
- Flytta alla lådor som står ovanpå B4 till staplar som inte innehåller B0 eller B4
- Ställ B0 på B4



Lösning


Randomize
Set goal
Solve
Max stack height:
None
Moves: 0
B13
B12
B9
B11
B10
B8
B5
B6
B3
B2
B7
B4
B1
B0

Design

Från analys till implementation

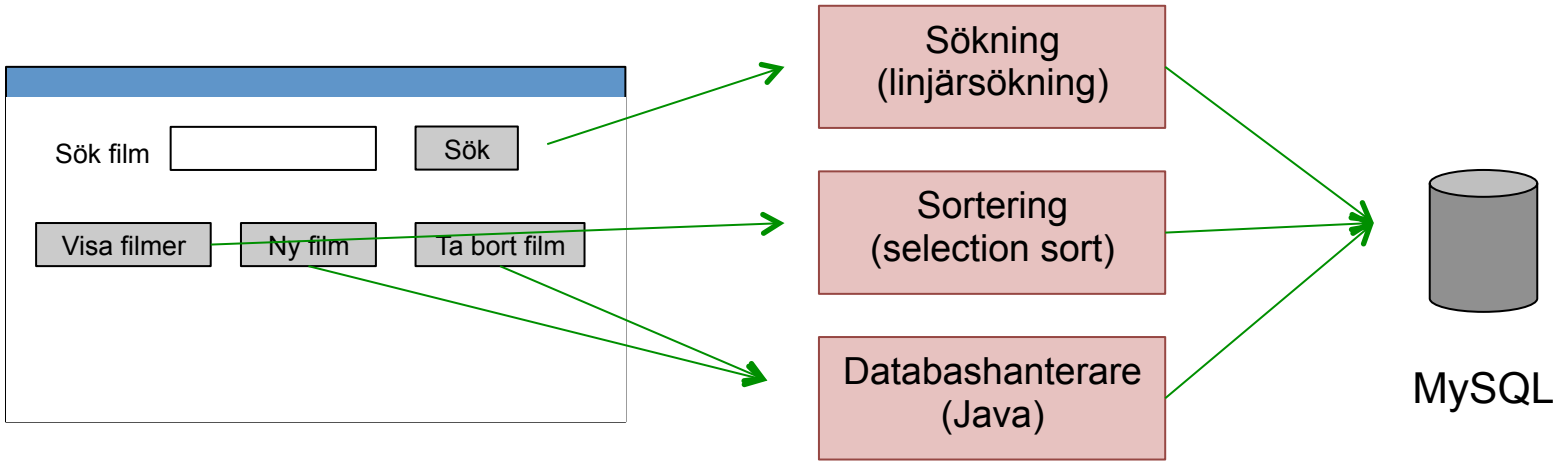
- I analysen definierar vi och beskriver problemet
- I implementationen skriver vi programkod för själva mjukvaran
- Precis som ett problem kan vara för komplext för att vi ska kunna hitta en lösning direkt, är ofta steget för stort att gå direkt från analys till implementation
- Vi mellanlandar därför i ett steg som kallas design
- I designen beskriver vi hur vi ska implementera en lösning på problemet

Design

- I design fasen definierar vi vilka moduler, algoritmer, verktyg och tekniker vi behöver för att utveckla programvaran
- Exempel: hur kan vi designa ett användargränssnitt till vår filmdatabas?

Titel	År	Regissör	IMDB
Nycken till frihet	1994	Frank Darabont	9,3
Inglorious Basterds	2009	Quentin Tarantino	8,3
Pulp Fiction	1994	Quentin Tarantino	8,9
Star Wars: Episod I – Det mörka hotet	1999	George Lucas	6,5
The Dark Knight Rises	2012	Christopher Nolan	8,4

Filmdatabas



HTML/CSS/JavaScript
Spring ramverk

Design

- I design fasen är det många beslut som behöver tas:
  - Databas: utveckla egen eller använda t.ex. MySQL eller Oracle DB?
  - Gränssnitt: webbaserat, mobilapp eller applikation?
  - Algoritm: ska sortering göras med insertion sort eller selection sort?
  - Programspråk: använda Java eller php?
- En erfaren utvecklare har en verktygslåda med algoritmer, tekniker och verktyg för att lösa problem

Exempel: Gissa talet

- Ett heltal mellan 1 och 100 väljs slumpmässigt ut
- Du ska gissa vilket talet är på så få gissningar som möjligt
- Vid varje gissning får du vet om:
  - Din gissning är korrekt
  - Om det korrekta talet är större än din gissning
  - Om det korrekt talet är mindre än din gissning
- Hur kan vi skapa en algoritm för detta?

Beskriva en algoritm

- En algoritm kan beskrivas på olika sätt
- Vi kan beskriva den i text, eller med programkods-liknande språk kallat pseudokod:

```
10    summa = 0
20    for (tal = 1 to 10)
30      summa = summa + tal
40    print ”Summan är ” + summa
```

Algoritm: Gissa talet

```
10  tal = RandomNumber(1,100)
20  gissning = 0
30  while (tal != gissning)
40    tal = GissaTal()
50    if (tal > gissning)
60      print(”Talet är större än din gissning”)
70    if (tal < gissning)
80      print(”Talet är mindre än din gissning”)
90  print(”Talet är ” + tal)
```


Beskriva en algoritm

- Pseudokod är att föredra framför textbeskrivning då det finns mindre utrymme för feltolkningar
- Pseudokod liknar programspråk som Java och C#
- Det finns dock inget standardiserat sätt att skriva pseudokod på, och kan se väldigt olika ut
- Beskrivning i text lämpar sig endast för enklare algoritmer
  - hur beskriver vi t.ex. iteration i text?

Top-down

- När vi tog fram algoritmen för gissa talet var det ett exempel på top-down design
- Vi börjar med en problembeskrivning, och bryter ner det i mindre delar som så småningom blir programkod
- Top-down är ett exempel på dekomposition av problemet till instruktioner/programkod

Prototyping

- Ett annat sätt att använda dekomposition när vi utvecklar mjukvara är prototyping
- En prototyp är en tidig version av en produkt
- Det kan till exempel vara ett användargränssnitt för vår filmdatabas, men inga knappar fungerar ännu
- Efter hand skapar vi nya prototyper där vi lägger till mer och mer funktionalitet tills vi har en färdig produkt

Prototyping

- Vår filmdatabas kan till exempel konstrueras i följande steg:

Prototyp	Funktionalitet
1	Webbgränssnitt utan funktionalitet
2	MySQL databas med filmer Visa samtliga filmer i databasen (ej krav på att de ska vara sorterade)
3	Gränssnitt för och hantering av lägga till eller ta bort filmer
4	Sök på film och filmlistan ska vara sorterad när den visas

Prototyping

- I prototyp 4 har vi lagt till all funktionalitet
- När vi har testat att programvaran fungerar som den ska och att vi uppfyllt alla krav i kravspecifikationen blir prototyp 4 i stället den slutliga produkten
- Olika varianter av iterativ utveckling, som till exempel prototyping, är det (troligen) vanligaste sättet att utveckla mjukvara idag
- Vi kan löpande visa programmet för kunden och få tidig feedback
- Får kunden inte se något förrän hela programvaran är färdigutvecklat är det risk att vi inte gjort vad kunden önskar…

Algoritmer för Sökning

Problemet

- Vi har en lista med heltal:
  - int[ ] = {9, 3, 12, 1, 7, 19, 13, 6}
- Vi ska nu se om ett specifikt tal finns i listan, till exempel 7
- Detta kräver att vi söker igenom listan efter talet 7

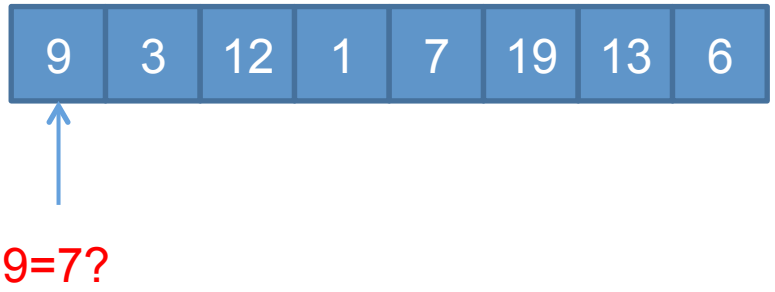
- Hur kan vi enklast göra detta?

Linjärsökning

- Den enklaste, och oftast mest användbara, algoritmen är linjärsökning
- Den söker igenom tal för tal i listan och jämför med talet vi söker efter

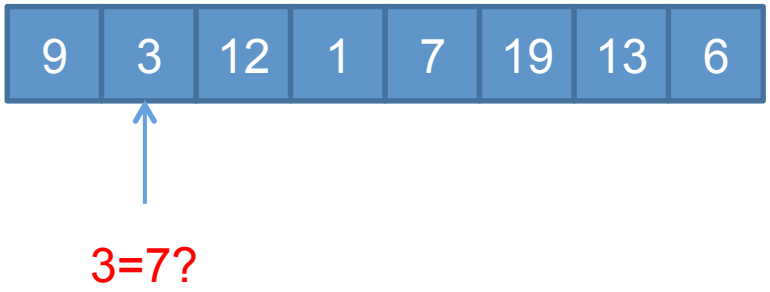
Linjärsökning

Sök efter 7



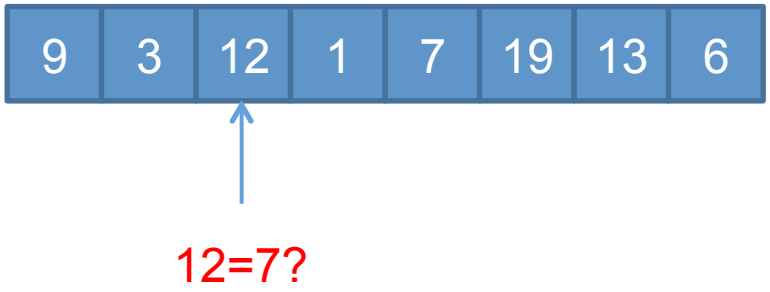
Linjärsökning

Sök efter 7



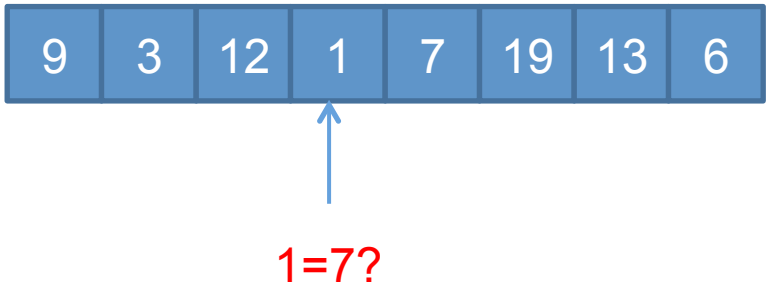
Linjärsökning

Sök efter 7



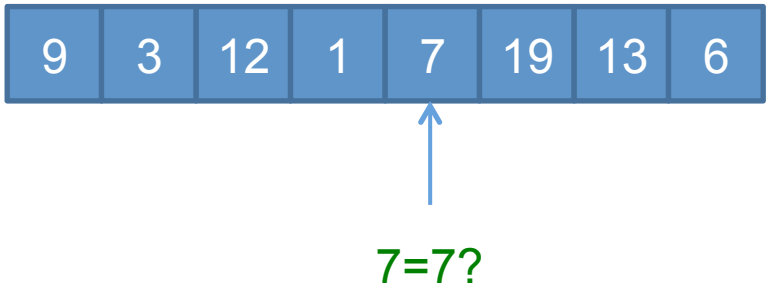
Linjärsökning

Sök efter 7



Linjärsökning

Sök efter 7



Vi har hittat talet 7!

Fråga: Hur konstaterar vi att ett tal inte finns, t.ex. om vi söker efter 2?

Linjärsökning

```
10    //Definiera lista och tal att söka efter
20    lista = {9, 3, 12, 1, 7, 19, 13, 6}
30    tal = 7
40
50    //Linjärsökning
60    i = 0
70    while (i < lista.length)
80      if (lista[i] == tal)
90        return true
100     i = i + 1
110   return false
```


Binärsökning

- En annan algoritm är binärsökning
- Den fungerar dock endast om listan är sorterad
- Detta begränsar algoritmen ganska mycket, men den har sina fördelar som vi ska återkomma till

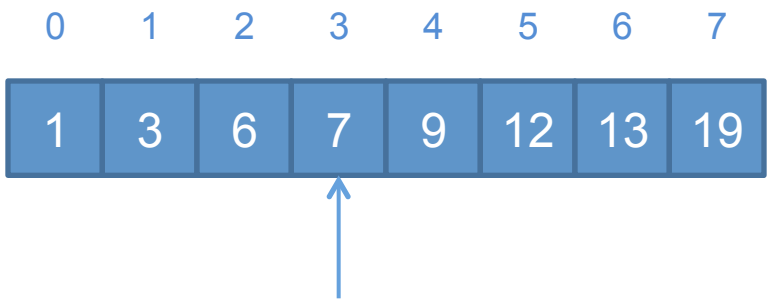
Binärsökning

Sök efter 9

Vi har en lista med tal sorterade i nummerordning.

Binärsökning

Sök efter 9

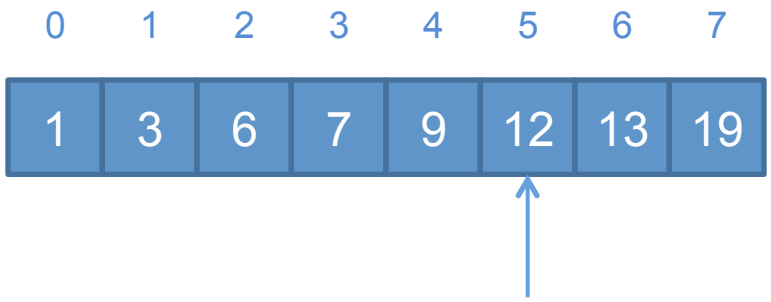


Börja med mittentalet
7 / 2 = 3 (avrundas neråt)

- Är mittentalet 9?
- Är 9 mindre än mittentalet?
- Är 9 större än mittentalet?

Binärsökning

Sök efter 9

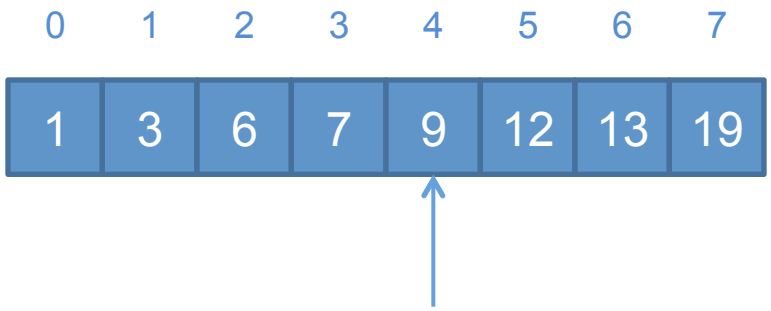


Beräkna nytt mittental
(7 + 3) / 2 = 5

- Är mittentalet 9?
- Är 9 mindre än mittentalet?
- Är 9 större än mittentalet?

Binärsökning

Sök efter 9



Beräkna nytt mittental
(5 + 3) / 2 = 4

Vi har hittat talet vi söker!

- Är mittentalet 9?
- Är 9 mindre än mittentalet?
- Är 9 större än mittentalet?

Binärsökning

```
10    //Definiera lista och tal att söka efter
20    lista = {1, 3, 6, 7, 9, 12, 13, 19}
30    tal = 9
40
50    //Binärsökning
60    middle = 0
70    left = 0
80    right = lista.length - 1
90    while (left <= right)
100     middle = (left + right) / 2
110     if (lista[middle] == tal)
120       return true
130     else if (lista[middle] > tal)
140       right = middle – 1
150     else if (lista[middle] < tal)
160       left = middle + 1
170   return false
```


Varför binärsökning?

- Linjärsökning fungerar på alla listor, binärsökning kräver att listan är sorterad
- Fördelen med binärsökning är prestanda
- Antag att vi har en lista med 500 tal
- I värsta-fallet (talet finns inte i listan) kräver linjärsökning att vi går igenom och jämför med samtliga 500 tal
- I binärsökning krävs som mest 9 steg

Linjärsökning mot Binärsökning

Linjärsökning	
Steg	**Kvar att undersöka**
1	499
2	498
3	497
4	496
5	495
6	494
7	493
8	492
9	491
10	490
…	…
499	1
500	Klar!

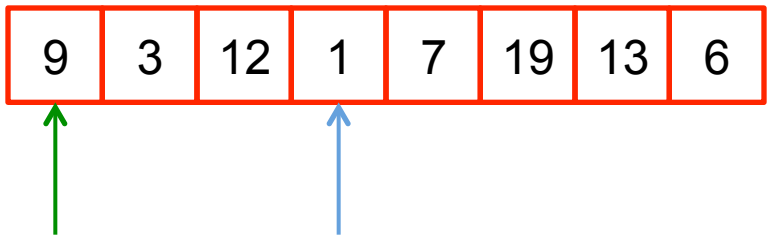
Binärsökning	
Steg	**Kvar att undersöka**
1	250
2	125
3	62
4	31
5	15
6	7
7	3
8	1
9	Klar!

Algoritmer för Sortering

Sortering

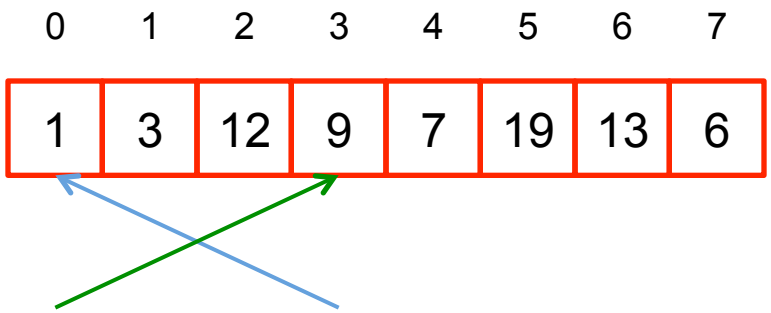
- Sortering är ett något mer komplext problem än sökning
- Det finns ett antal olika algoritmer för att sortera en lista:
  - Selection sort (urvalssortering)
  - Insertion sort (instickssortering)
  - Bubble sort (bubbelsortering)
  - Merge sort
  - Quick sort
  - …
- Vi ska se hur urvalssortering går till, och så får ni kolla upp de andra på t.ex. Wikipedia

Urvalssortering



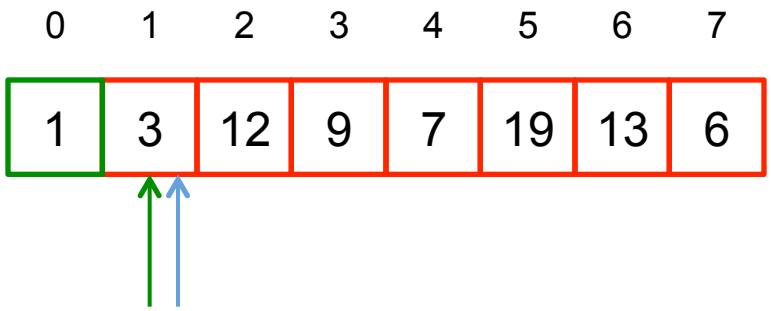
Steg 1: Börja på plats 0 och sök igenom resten av listan efter lägsta talet

Urvalssortering



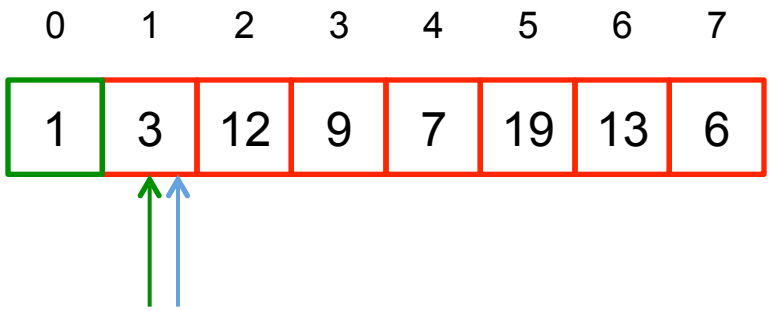
Steg 2: Byt plats på plats 0 och platsen med det minsta talet

Urvalssortering



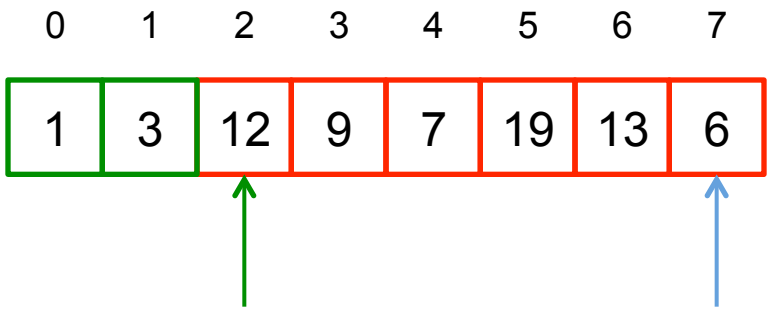
Steg 3: Börja på plats 1 och hitta minsta talet i resten av listan

Urvalssortering



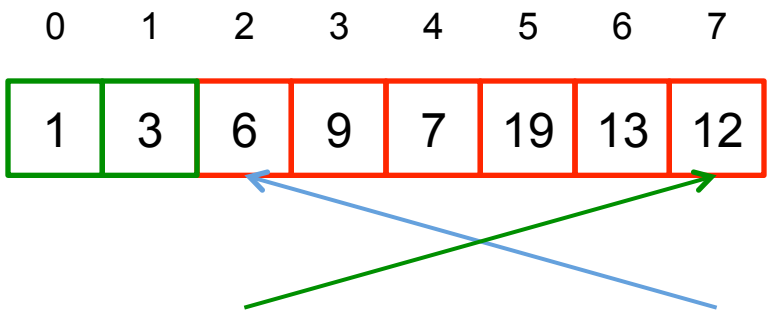
Steg 4: Byt plats mellan plats 1 och platsen med minsta talet. I detta fall är det samma plats så vi behöver inte göra någonting

Urvalssortering



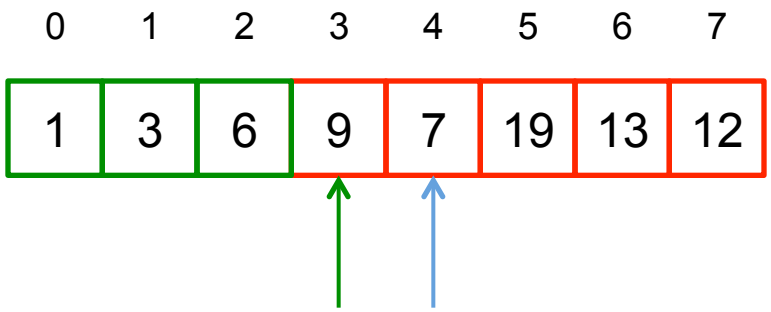
Steg 5: Börja på plats 2 och leta upp minsta talet i resten av listan

Urvalssortering



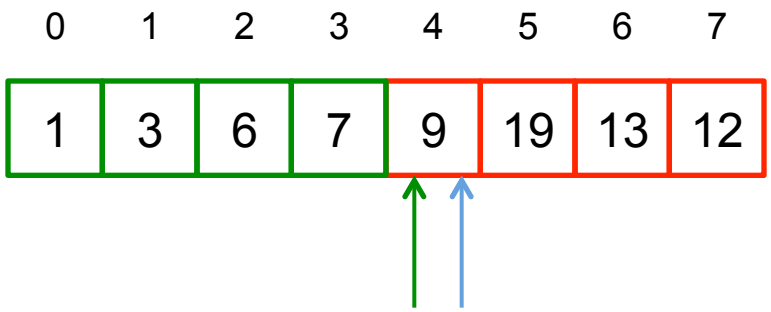
Steg 5: Byt plats på plats 2 och platsen med minsta värdet

Urvalssortering



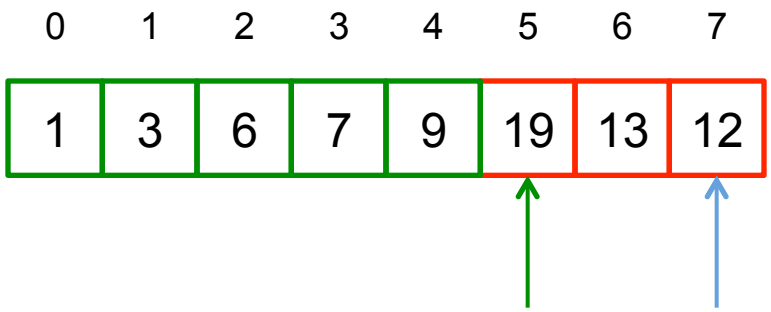
… fortsätt med plats 3

Urvalssortering



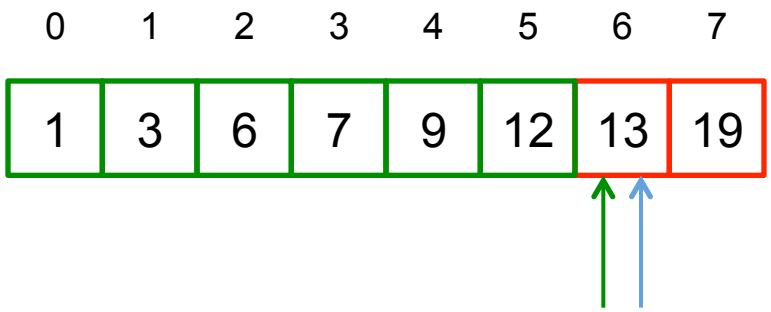
… fortsätt med plats 4

Urvalssortering



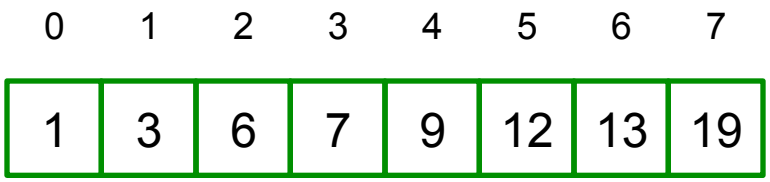
… fortsätt med plats 5

Urvalssortering



… fortsätt med plats 6

Urvalssortering



Klar! Listan är sorterad!

Urvalssortering

```
10    //Definiera lista att sortera
20    lista = {9, 3, 12, 1, 7, 19, 13, 6}
30
40    //Urvalssortering
50    for (i = 0 to i < lista.length - 1)
60      min = lista[i]
70      //Gå igenom resten av listan
80      for (j = i + 1 to j < lista.length)
90        //Se om vi har nytt lägsta-värde
100       if (lista[j] < min)
110         min = lista[j]
120         plats = j
130     //Byt plats på talen
140     lista[plats] = lista[i]
150     lista[i] = min
```


Funktioner och Klasser

Programkod

- Programkod består av flera delar:
  - Variabler
  - Instruktioner
  - Kontrollstrukturer
  - Funktion
- Vi har diskuterat variabler, instruktioner och kontrollstrukturer
- En funktion är en sammanhängande del programkod som utför en specifik uppgift

Funktion

- En funktion har ett namn som beskriver dess funktion
- Den kan också ha en eller flera parametrar som indata
- Och eventuellt ett returvärde
- Vi kan till exempel göra en funktion av vår algoritm för linjärsökning:

Linjärsökning som funktion

```
10    boolean linearSearch(int[] lista, int tal)
20      i = 0
30      while (i < lista.length)
40        if (lista[i] == tal)
50          return true
60        i = i + 1
70      return false
```


Funktioner

- En stor fördel med funktioner är att vi kan återanvända kod
- Varje gång vi behöver söka i en lista kan vi anropa en funktion i stället för att skriva om samma kod igen

Använda funktionen

```
10    //Definiera lista och tal att söka efter
20    int[] lista = {9, 3, 12, 1, 7, 19, 13, 6}
30    int[] tal = {7, 2, 9}
40    for (int i = 0 to tal.length)
50      //Anropa funktionen
60      boolean found = linearSearch(lista, tal[i])
70      if (found == true)
80        print(”Talet ” + tal[i] + ” finns i listan!”)
90      else
100       print(”Talet ” + tal[i] + ” finns inte i listan!”)
```


Klass

- En klass samlar relaterade variabler och funktioner i en enhet
- Klassen ska ha ett syfte, till exempel linjärsökning
- Klasser är grunden i objektorientering, där klasser modellerar enheter
  - Cirkel, Kvadrat, Triangel, …
  - Bil, Lastbil, Motorcykel, …
- En egenskap hos klasser är att funktionerna kan användas utifrån, men den inre koden är dold utanför klassen

Klass

- En klass modelleras med titel, ett block för variabler och ett block för funktioner:

LinearSearch
int[] lista int tal
boolean search(int[] lista, int tal)

Car
String brand int year String model
Car(String brand, int year, String model) String getBrand() int getYear() String getModel()

Klass och Objekt

- En klass är själva mallen för en enhet, t.ex. Car
- Vi kan använda mallen för att skapa flera Car objekt:

```
10  decentCar = new Car(”Ford”, 2015, ”Fiesta”)
20  betterCar = new Car(”Volvo”, 2015, ”XC60”)
30  evenBetterCar = new Car(”Toyota”, 2016, ”Avensis”)
40  bestCar = new Car(”Tesla”, 2017, ”Model S”)
```

- När vi skapar ett objekt anger vi värden på klassvariablerna