Föreläsning 3: Sortering

Sortering är en av dom vanligaste uppgifterna för ett program. Här följer en beskrivning av några sorteringsalgoritmer.

- Urvalssortering (Selection sort)
- Insättningssortering (Insertion sort)
- Damerna först
- Räknesortering (Distribution count)
- Quicksort
- Samsortering (Mergesort)
- Sortering i Javas API

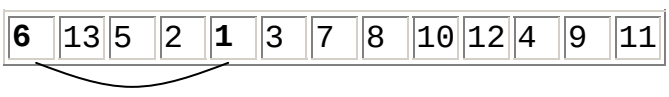
Urvalssortering (Selection sort)

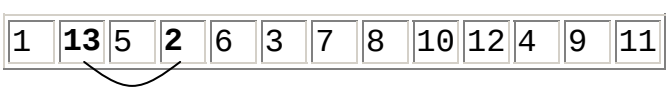
Vi tänker oss att vi ska sortera en vektor med N tal.

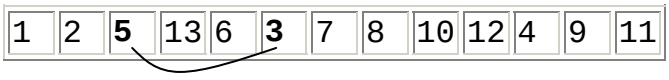
- Sök igenom vektorn efter minsta talet. (N-1 jämförelser)
- Flytta det till första positionen. (Ett byte)
- Sök efter näst minsta talet. (N-2 jämförelser)
- Flytta det till andra positionen. (Ett byte) - - -

Totalt krävs N(N-1)/2 jämförelser och N-1 byten, helt oberoende av hur pass osorterad vektorn är från början. Tiden är alltså i stort sett proportionell mot kvadraten på N, alltså komplexiteten är $O(N^2)$.

```
def selection_sort(vek):
    i = 0
    while i < len(vek):
        minst = i
        j = i
        while j < len(vek):
            if (vek[j] < vek[minst]):
                minst = j
            j += 1
        print vek[minst], "bör byta plats med ", vek[i]
        (vek[minst], vek[i]) = (vek[i], vek[minst])
        i += 1
```







Insättningssortering (Insertion sort)

Denna metod känns särskilt naturlig om man hämtar talen ett efter ett från en fil och sorterar in dom i en vektor. Jämför med att sortera en korthand.

- Jämför nya talet med sista talet i vektor.
- Om nya är större, lägg det sist i vektorn.
- Annars, putta ner sista talet ett pinnhål och jämför igen.
- Putta ner så många tal som behövs för att sätta in nya talet på rätt plats.
- Upprepa för varje nytt tal.

Insättning fungerar också om talen finns i vektorn från början. Efter k insättningar är vektorsegmentet `data[1]...data[k]` ordnat och `data[k+1]` är det nya talet som ska sättas in.

Komplexiteten är i allmänhet $O(N^2)$ men för att sortera in ett nytt värde i en redan sorterad vektor är insättning den snabbaste algoritmen.

Damerna först

Den enklaste sorteringsuppgiften är att sortera om en personarray med damerna först och den smarta algoritmen kallas **damerna-först-sortering**.

1. Sätt ett pekfinger i var ände av arrayen!
2. Rör fingrarna mot varandra tills vänstra fingret fastnat på en herre och högra fingret på en dam!
3. Låt damen och herren byta plats!
4. Upprepa från 2 tills fingrarna korsats!

Idén kan utvecklas till Quicksort, som är den snabbaste av alla sorteringsalgoritmer.

Räknesortering (Distribution count)

Om man vet att det bara finns ett litet antal nyckelvärden, till exempel 100 olika, är distributionsräkning oslagbart snabbt. Det kräver att talen som sorteras in i vektorn hämtas från en annan vektor eller fil.

- Läs igenom filen och räkna hur många det finns av varje nyckelvärde.
- Dela in vektorn i lagom stora segment för denna distribution.
- Läs filen igen och lägg in varje värde i sitt segment.

Komplexiteten blir $O(N)$.

Quicksort

Damerna-först-algoritmen med två pekfingrar används i Quicksort. Först bestämmer man vilka (små) nyckelvärden som ska kallas damer. Resten kallas herrar och så utför man algoritmen. Vektorn delas alltså i två segment, det första med små värden, det andra med stora värden. Nu behöver man bara sortera segmenten var för sej. Det här är en rekursiv tanke!

- Bestäm vilka värden som ska kallas damer.
- Partitionera vektorn så att damerna kommer först.
- Sortera varje segment för sej.

Rekursion

Rekursion kommer vi att gå igenom grundligt nästa vecka. Tills vidare kan vi säga att det är när en funktion anropar sig själv. För att inte ha en oaändlig rekuriosn måste det finnas ett avbrottsvillkor.

Komplexiteten blir i allmänhet *O(N log N)*. Det beror på att man kan dela vektorn på mitten log N gånger. Exakt hur snabb den är beror på hur man avgör vilka värden som ska vara damer. Ofta tar man det första talet i vektorn och utnämner det och alla mindre tal till damer. Då blir Quicksort kvadratisk, $O(n^2)$, för redan nästan sorterade vektorer! Det värde man väljer att sortera efter kallas *pivottal*. Ett sätt att välja detta tal är att slumpa fram en position i vektorn. Bättre är att ta ut tre tal - det första, det sista och något i mitten - och låta det mellersta värdet bestämma vad som är damer. Det kallas *median-of-three*. Man kan visa att antal jämförelser då blir *1.4 N log N* i genomsnitt. Koden kan i princip beskrivas med följande funktioner:

```
def quicksort(vek, first, last):
    pivotindex = (first + last) / 2
    correctpos = partition(vek, first, last, pivotindex)
    if (correctpos >= first and correctpos <= last):
        quicksort(vek, first, correctpos -1)
        quicksort(vek, correctpos + 1, last)

def partition(vek, first, last, pivot):
    (vek[pivot], vek[last]) = (vek[last], vek[pivot]) # pivot ytterst
    low = first
    high = last
    while (low <= high):
        while (low <= high and vek[low]  <= vek[last]) : low += 1
        while (high >= low and vek[high] >= vek[last]) : high -= 1
        if (low < high):
            (vek[low], vek[high]) = (vek[high], vek[low])
            low += 1
            high -= 1
    (vek[pivot], vek[last]) = (vek[last], vek[pivot])  # byt tillbaka
    return pivot
```

```
<Ivar, Filip, Cesar, David, Gustav, Johan, Erik, Bertil, Helge, Adam>
  Byter plats på pivot <Filip> och <Adam>
    Byter plats på <Ivar> och <Bertil>
    Byter plats på <Gustav> och <Erik>
  Byter plats på pivot <Filip> och <Johan>
 <Bertil, Adam, Cesar, David, Erik, Filip, Gustav, Ivar, Helge, Johan>
  DELAR UPP <Bertil, Adam, Cesar, David, Erik>, PIVOT: 4 (<David>)
    Byter plats på pivot <David> och <Erik>
```

```
    Byter plats på pivot <David> och <Erik>
  EFTER DELNING: <Bertil, Adam, Cesar, David, Erik>
    DELAR UPP <Bertil, Adam, Cesar>, PIVOT: 3 (<Cesar>)
      Byter plats på pivot <Cesar> och <Cesar>
      Byter plats på pivot <Cesar> och <Cesar>
    EFTER DELNING: <Bertil, Adam, Cesar>
      DELNING BASFALL: Byter plats på <Bertil> och <Adam>
    EFTER SORTERING: <Adam, Bertil, Cesar>
    DELNING BASFALL: Ett element <Erik>
  EFTER SORTERING: <Adam, Bertil, Cesar, David, Erik>
  DELAR UPP <Gustav, Ivar, Helge, Johan>, PIVOT: 1 (<Gustav>)
    Byter plats på pivot <Gustav> och <Johan>
    Byter plats på pivot <Gustav> och <Johan>
  EFTER DELNING: <Gustav, Ivar, Helge, Johan>
    DELAR UPP <Ivar, Helge, Johan>, PIVOT: 3 (<Johan>)
      Byter plats på pivot <Johan> och <Johan>
      Byter plats på pivot <Johan> och <Johan>
    EFTER DELNING: <Ivar, Helge, Johan>
      DELNING BASFALL: Byter plats på <Ivar> och <Helge>
    EFTER SORTERING: <Helge, Ivar, Johan>
  EFTER SORTERING: <Gustav, Helge, Ivar, Johan>
<Adam, Bertil, Cesar, David, Erik, Filip, Gustav, Helge, Ivar, Johan>
```

Samsortering (Mergesort)

Om man har flera sorterade småfiler är det lätt att *samsortera* dom till en fil. Det här kan man också göra med en vektor om man har extrautrymme för att kopiera den till två andra hälften så långa vektorer. Det här ger en rekursiv tanke!

- Dela vektorn i två hälften så långa vektorer.
- Sortera varje halva för sej.
- Samsortera till ursprungliga vektorn.

Komplexiteten blir *O(N log N)*, lika snabb som quicksort men kräver extra minnesutrymme.

Sortering i python API

Det finns färdiga sorteringsrutiner i python.

```
vektor = [1, 2, 3, 4, 5, 7]
list.sort(vektor)
```

Men vad gör man om man vill sortera på bokens titel ena stunden och författaren nästa? Man kan skicka en funktionspekar `cmp` som löser det problemet.

Sortering av större mängder data

Alla metoderna ovan förutsätter att de data som ska sorteras kan lagras i primärminnet. Om så inte är fallet får man ta till *extern sortering* men det ingår inte i den här kursen.