DD1331

Grundläggande programmering för F1

Några bilder till föreläsning 6

Innehåll

Innehåll

- Sorteringsalgoritmer

Innehåll

- Sorteringsalgoritmer
  - Enkla metoder

Innehåll

- Sorteringsalgoritmer
  - Enkla metoder $\mathcal{O}(n^2)$

Innehåll

- Sorteringsalgoritmer
  - Enkla metoder $\mathcal{O}(n^2)$
    - Bubbelsortering Insättningssortering Urvalssortering

Innehåll

- Sorteringsalgoritmer
  - Enkla metoder $\mathcal{O}(n^2)$
    - Bubbelsortering Insättningssortering Urvalssortering
  - Effektivare metoder

Innehåll

- Sorteringsalgoritmer
  - Enkla metoder $\mathcal{O}(n^2)$
    - Bubbelsortering Insättningssortering Urvalssortering
  - Effektivare metoder $\mathcal{O}(n\ log(n))$

Innehåll

- Sorteringsalgoritmer
  - Enkla metoder $\mathcal{O}(n^2)$
    - Bubbelsortering Insättningssortering Urvalssortering
  - Effektivare metoder $\mathcal{O}(n\ log(n))$
    - Mergesort Quicksort

Innehåll

- Sorteringsalgoritmer
  - Enkla metoder $\mathcal{O}(n^2)$
    - Bubbelsortering Insättningssortering Urvalssortering
  - Effektivare metoder $\mathcal{O}(n\ log(n))$
    - Mergesort Quicksort
  - Metoder utan jämförelser mellan element

Innehåll

- Sorteringsalgoritmer
  - Enkla metoder $\mathcal{O}(n^2)$
    - Bubbelsortering Insättningssortering Urvalssortering
  - Effektivare metoder $\mathcal{O}(n\ log(n))$
    - Mergesort Quicksort
  - Metoder utan jämförelser mellan element
    - Räknesortering Radixsortering

Innehåll

- Sorteringsalgoritmer
  - Enkla metoder $\mathcal{O}(n^2)$
    - Bubbelsortering Insättningssortering Urvalssortering
  - Effektivare metoder $\mathcal{O}(n\ log(n))$
    - Mergesort Quicksort
  - Metoder utan jämförelser mellan element
    - Räknesortering Radixsortering
- Algoritmexempel: Anagram

Innehåll

- Sorteringsalgoritmer
  - Enkla metoder $\mathcal{O}(n^2)$
    - Bubbelsortering Insättningssortering Urvalssortering
  - Effektivare metoder $\mathcal{O}(n\ log(n))$
    - Mergesort Quicksort
  - Metoder utan jämförelser mellan element
    - Räknesortering Radixsortering
- Algoritmexempel: Anagram
  - Jämför fyra algoritmers tidsprestanda

Bubbelsortering

Bubbelsortering

- Byt plats på element som ligger i fel ordning

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

```
6 3 7 8 5 1     n st element, sortera stigande
```

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

```
6 3 7 8 5 1     n st element, sortera stigande
6 3 7 8 5 1
```

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

```
6 3 7 8 5 1     n st element, sortera stigande
6 3 7 8 5 1     6>3  fel ordning, byt plats
```

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

```
6 3 7 8 5 1     n st element, sortera stigande
6 3 7 8 5 1     6>3  fel ordning, byt plats
3 6 7 8 5 1
```

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

```
6 3 7 8 5 1     n st element, sortera stigande
6 3 7 8 5 1     6>3  fel ordning, byt plats
3 6 7 8 5 1
3 6 7 8 5 1
```

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

```
6 3 7 8 5 1     n st element, sortera stigande
6 3 7 8 5 1     6>3  fel ordning, byt plats
3 6 7 8 5 1
3 6 7 8 5 1     6<7  rätt ordning, byt ej
```

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

```
6 3 7 8 5 1     n st element, sortera stigande
6 3 7 8 5 1     6>3  fel ordning, byt plats
3 6 7 8 5 1
3 6 7 8 5 1     6<7  rätt ordning, byt ej
3 6 7 8 5 1
```

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

```
6 3 7 8 5 1     n st element, sortera stigande
6 3 7 8 5 1     6>3  fel ordning, byt plats
3 6 7 8 5 1
3 6 7 8 5 1     6<7  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1
```

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

```
6 3 7 8 5 1     n st element, sortera stigande
6 3 7 8 5 1     6>3  fel ordning, byt plats
3 6 7 8 5 1
3 6 7 8 5 1     6<7  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1     7<8  rätt ordning, byt ej
```

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

```
6 3 7 8 5 1     n st element, sortera stigande
6 3 7 8 5 1     6>3  fel ordning, byt plats
3 6 7 8 5 1
3 6 7 8 5 1     6<7  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1     7<8  rätt ordning, byt ej
3 6 7 8 5 1
```

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

```
6 3 7 8 5 1     n st element, sortera stigande
6 3 7 8 5 1     6>3  fel ordning, byt plats
3 6 7 8 5 1
3 6 7 8 5 1     6<7  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1     7<8  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1
```

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

```
6 3 7 8 5 1     n st element, sortera stigande
6 3 7 8 5 1     6>3  fel ordning, byt plats
3 6 7 8 5 1
3 6 7 8 5 1     6<7  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1     7<8  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1     8>5  fel ordning, byt plats
```

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

```
6 3 7 8 5 1     n st element, sortera stigande
6 3 7 8 5 1     6>3  fel ordning, byt plats
3 6 7 8 5 1
3 6 7 8 5 1     6<7  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1     7<8  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1     8>5  fel ordning, byt plats
3 6 7 5 8 1
```

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

```
6 3 7 8 5 1     n st element, sortera stigande
6 3 7 8 5 1     6>3  fel ordning, byt plats
3 6 7 8 5 1
3 6 7 8 5 1     6<7  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1     7<8  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1     8>5  fel ordning, byt plats
3 6 7 5 8 1
3 6 7 5 8 1
```

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

```
6 3 7 8 5 1     n st element, sortera stigande
6 3 7 8 5 1     6>3  fel ordning, byt plats
3 6 7 8 5 1
3 6 7 8 5 1     6<7  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1     7<8  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1     8>5  fel ordning, byt plats
3 6 7 5 8 1
3 6 7 5 8 1     8>1  fel ordning, byt plats
```

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

```
6 3 7 8 5 1     n st element, sortera stigande
6 3 7 8 5 1     6>3  fel ordning, byt plats
3 6 7 8 5 1
3 6 7 8 5 1     6<7  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1     7<8  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1     8>5  fel ordning, byt plats
3 6 7 5 8 1
3 6 7 5 8 1     8>1  fel ordning, byt plats
3 6 7 5 1 8
```

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

```
6 3 7 8 5 1     n st element, sortera stigande
6 3 7 8 5 1     6>3  fel ordning, byt plats
3 6 7 8 5 1
3 6 7 8 5 1     6<7  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1     7<8  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1     8>5  fel ordning, byt plats
3 6 7 5 8 1
3 6 7 5 8 1     8>1  fel ordning, byt plats
3 6 7 5 1 8
```

Visar ett **svep** genom listan.

Bubbelsortering

- Byt plats på element som ligger i fel ordning
- Upprepa tills elementen är sorterade

```
6 3 7 8 5 1     n st element, sortera stigande
6 3 7 8 5 1     6>3  fel ordning, byt plats
3 6 7 8 5 1
3 6 7 8 5 1     6<7  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1     7<8  rätt ordning, byt ej
3 6 7 8 5 1
3 6 7 8 5 1     8>5  fel ordning, byt plats
3 6 7 5 8 1
3 6 7 5 8 1     8>1  fel ordning, byt plats
3 6 7 5 1 8
```

Visar ett **svep** genom listan. Efter n-1 **svep** är listan sorterad.

Bubbelsortering

```
def bubblesort(v):
    changed = True
    while changed:
        changed = False
        for i in range(1, len(v)):
            if v[i-1] > v[i]:
                v[i-1],v[i] = v[i],v[i-1]
                changed = True    # platsbyte
```

Bubbelsortering

```
def bubblesort(v):
    changed = True
    while changed:
        changed = False
        for i in range(1, len(v)):
            if v[i-1] > v[i]:
                v[i-1],v[i] = v[i],v[i-1]
                changed = True    # platsbyte
```

Variabelvärden byter plats: a,b = b,a

Bubbelsortering

```
def bubblesort(v):
    changed = True
    while changed:
        changed = False
        for i in range(1, len(v)):
            if v[i-1] > v[i]:
                v[i-1],v[i] = v[i],v[i-1]
                changed = True    # platsbyte
```

Variabelvärden byter plats: a,b = b,a

For-satsen är ett svep.

Bubbelsortering

```
def bubblesort(v):
    changed = True
    while changed:
        changed = False
        for i in range(1, len(v)):
            if v[i-1] > v[i]:
                v[i-1],v[i] = v[i],v[i-1]
                changed = True    # platsbyte
```

Variabelvärden byter plats: a,b = b,a

For-satsen är ett svep.

Upprepa svepen så länge de medför byten.

Bubbelsortering

```
def bubblesort(v):
    changed = True
    while changed:
        changed = False
        for i in range(1, len(v)):
            if v[i-1] > v[i]:
                v[i-1],v[i] = v[i],v[i-1]
                changed = True    # platsbyte
```

Variabelvärden byter plats: a,b = b,a

For-satsen är ett svep.

Upprepa svepen så länge de medför byten.

Avbryt när ett svep genomförts utan byte.

Om variabeln changed är False efter for-satsen
så gjordes inget byte.

Bubbelsortering, lite smartare

Bubbelsortering, lite smartare

Utnyttja att

- största elementet hamnar på rätt plats i svep 1

Bubbelsortering, lite smartare

Utnyttja att

- största elementet hamnar på rätt plats i svep 1
- näst största elementet hamnar rätt i svep 2 o.s.v.

Bubbelsortering, lite smartare

Utnyttja att

- största elementet hamnar på rätt plats i svep 1
- näst största elementet hamnar rätt i svep 2 o.s.v.

```
def bubblesort2(v):
    changed = True
    grans = len(v)
    while changed:
        changed = False
        for i in range(1, grans):
            if v[i-1] > v[i]:
                v[i-1],v[i] = v[i],v[i-1]
                changed = True
        grans = grans-1
```

Bubbelsortering, lite smartare

Utnyttja att

- största elementet hamnar på rätt plats i svep 1
- näst största elementet hamnar rätt i svep 2 o.s.v.

```
def bubblesort2(v):
    changed = True
    grans = len(v)
    while changed:
        changed = False
        for i in range(1, grans):
            if v[i-1] > v[i]:
                v[i-1],v[i] = v[i],v[i-1]
                changed = True
        grans = grans-1
```

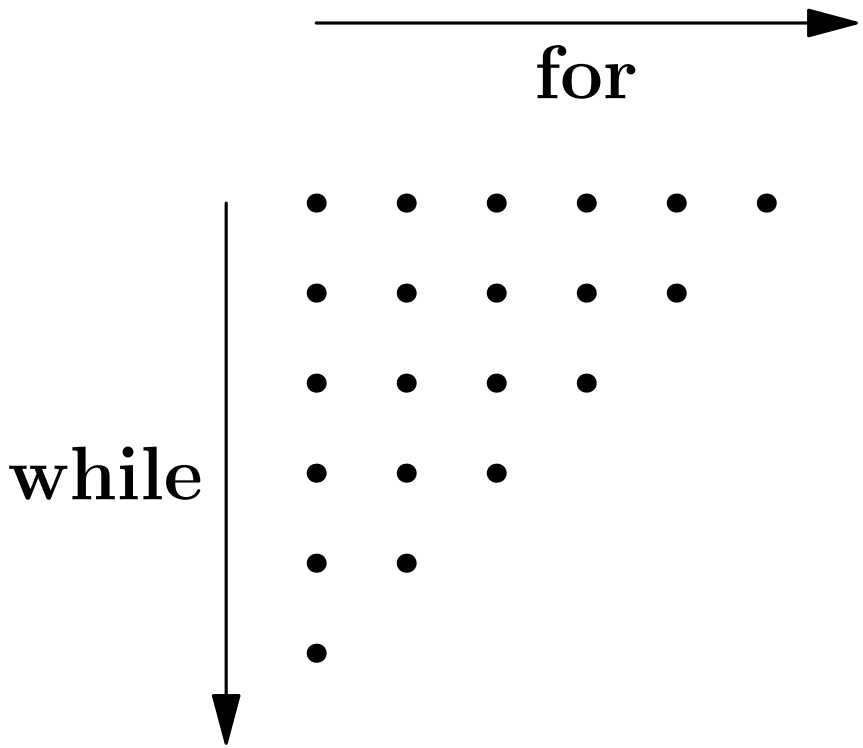
grans = svepets övre gräns, minskas med 1 för varje svep.

Bubbelsortering, tidsåtgång i antal jämförelser

smartare versionen med grans, bubbelsort2, $n = 7$

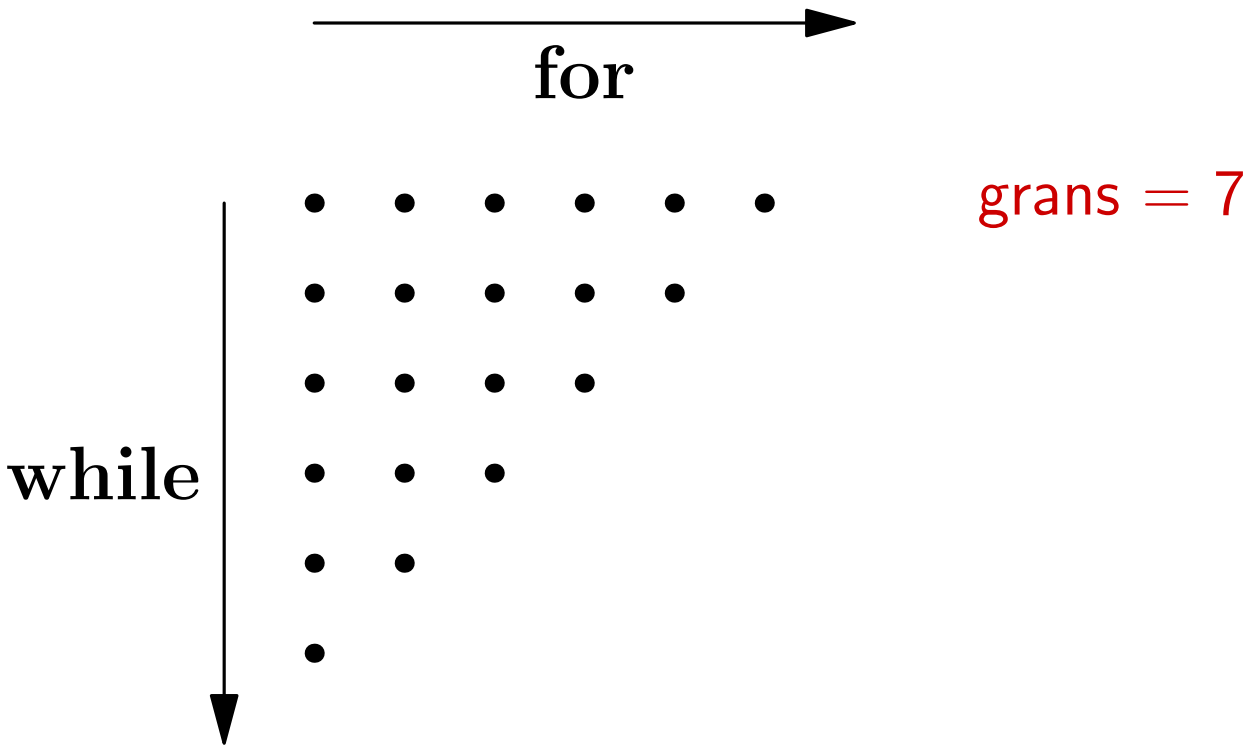
Bubbelsortering, tidsåtgång i antal jämförelser

smartare versionen med grans, bubbelsort2, n = 7



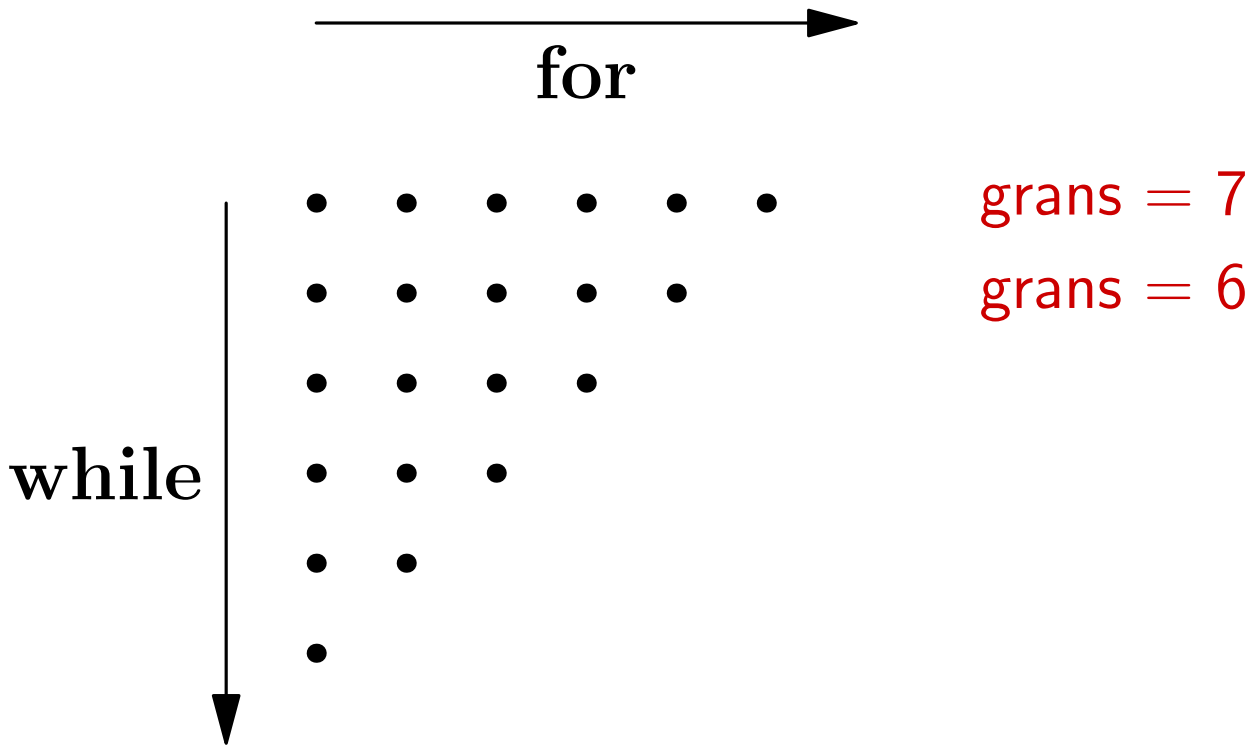
Bubbelsortering, tidsåtgång i antal jämförelser

smartare versionen med grans, bubbelsort2, n = 7



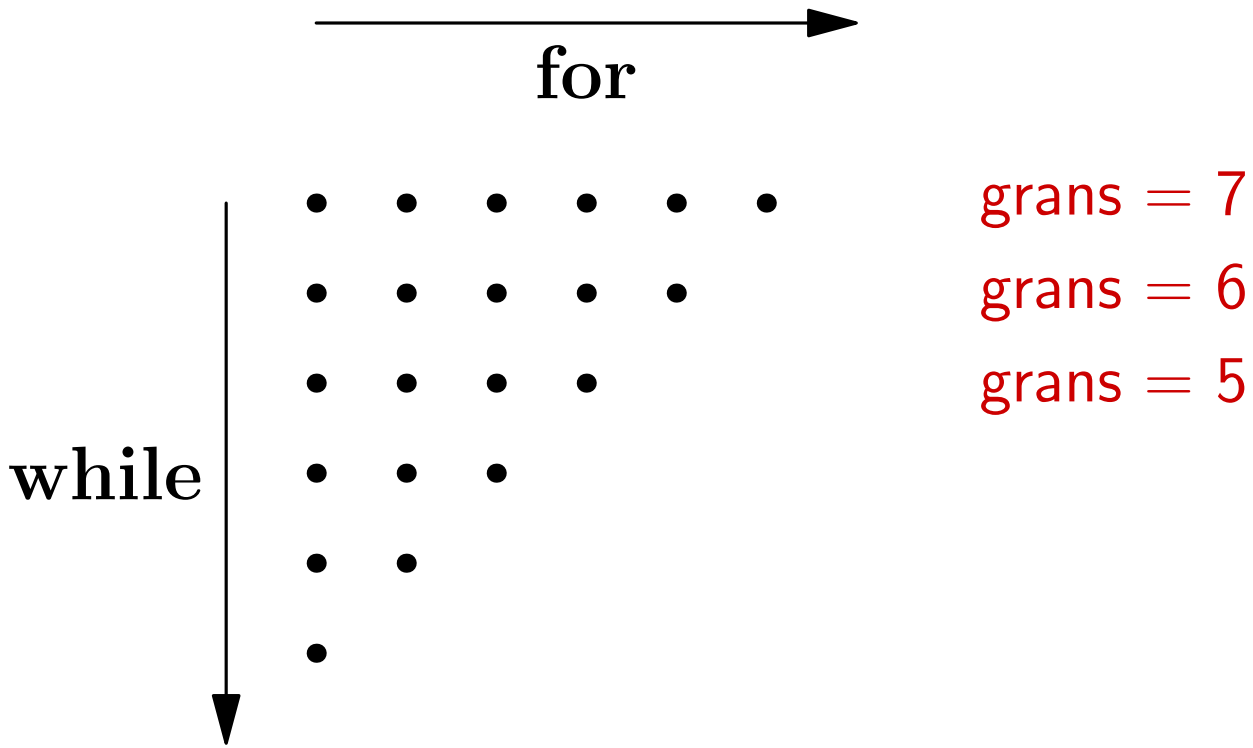
Bubbelsortering, tidsåtgång i antal jämförelser

smartare versionen med grans, bubbelsort2, n = 7



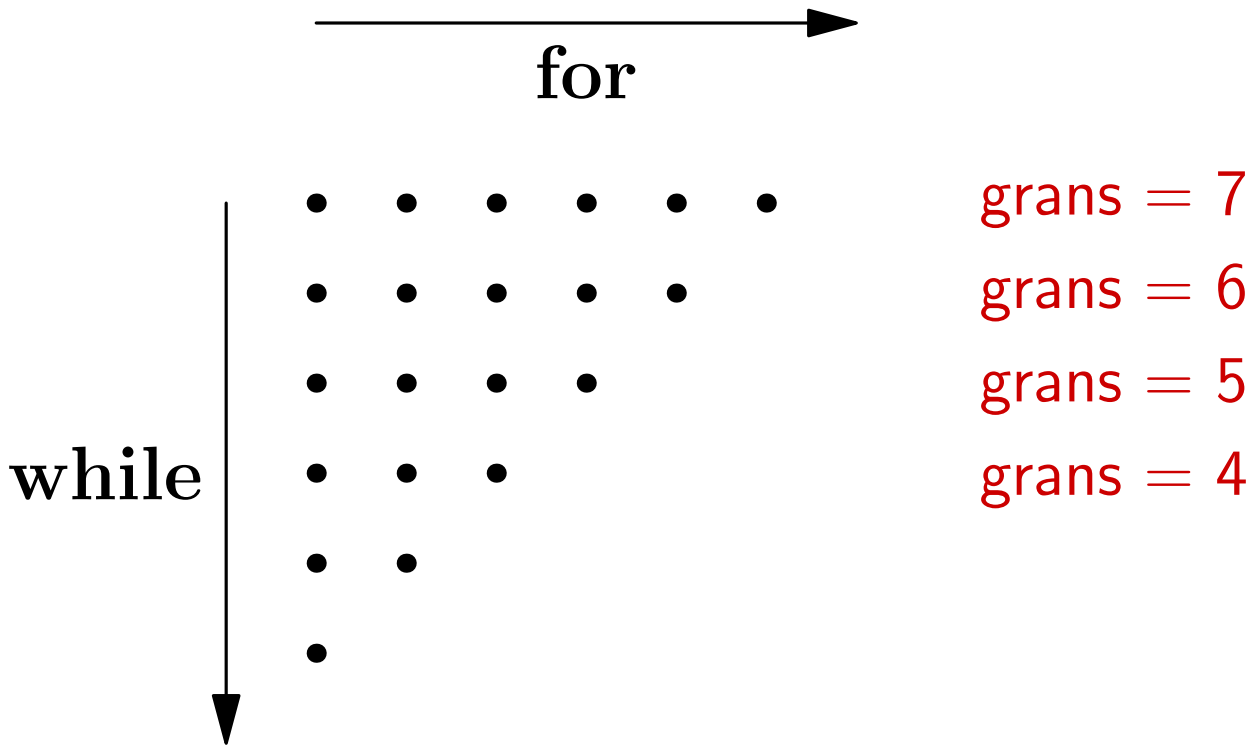
Bubbelsortering, tidsåtgång i antal jämförelser

smartare versionen med grans, bubbelsort2, n = 7



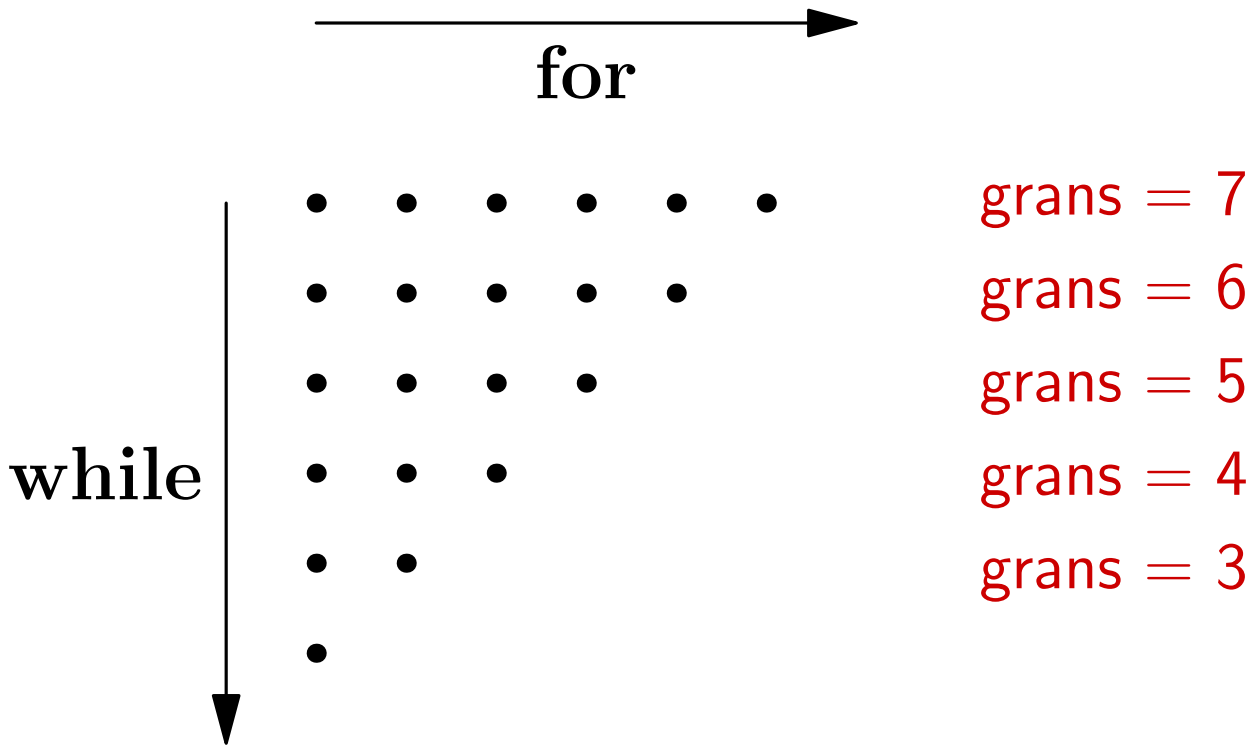
Bubbelsortering, tidsåtgång i antal jämförelser

smartare versionen med grans, bubbelsort2, n = 7



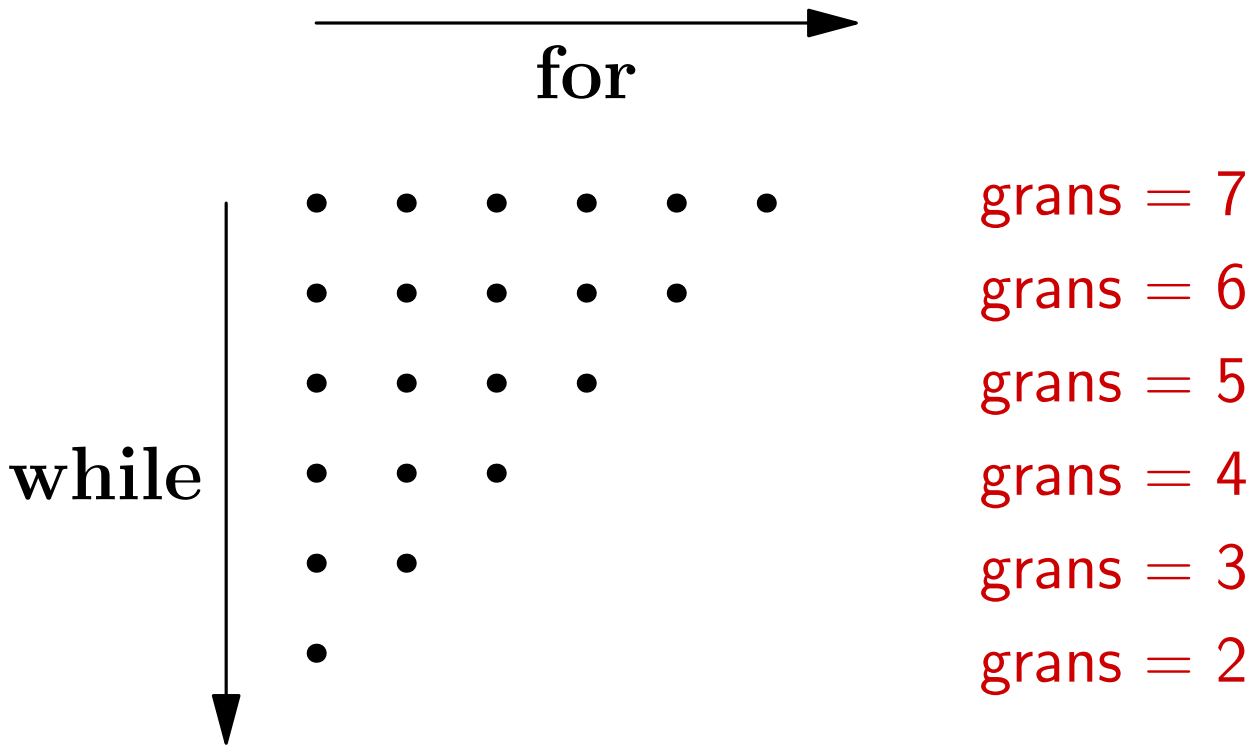
Bubbelsortering, tidsåtgång i antal jämförelser

smartare versionen med grans, bubbelsort2, n = 7



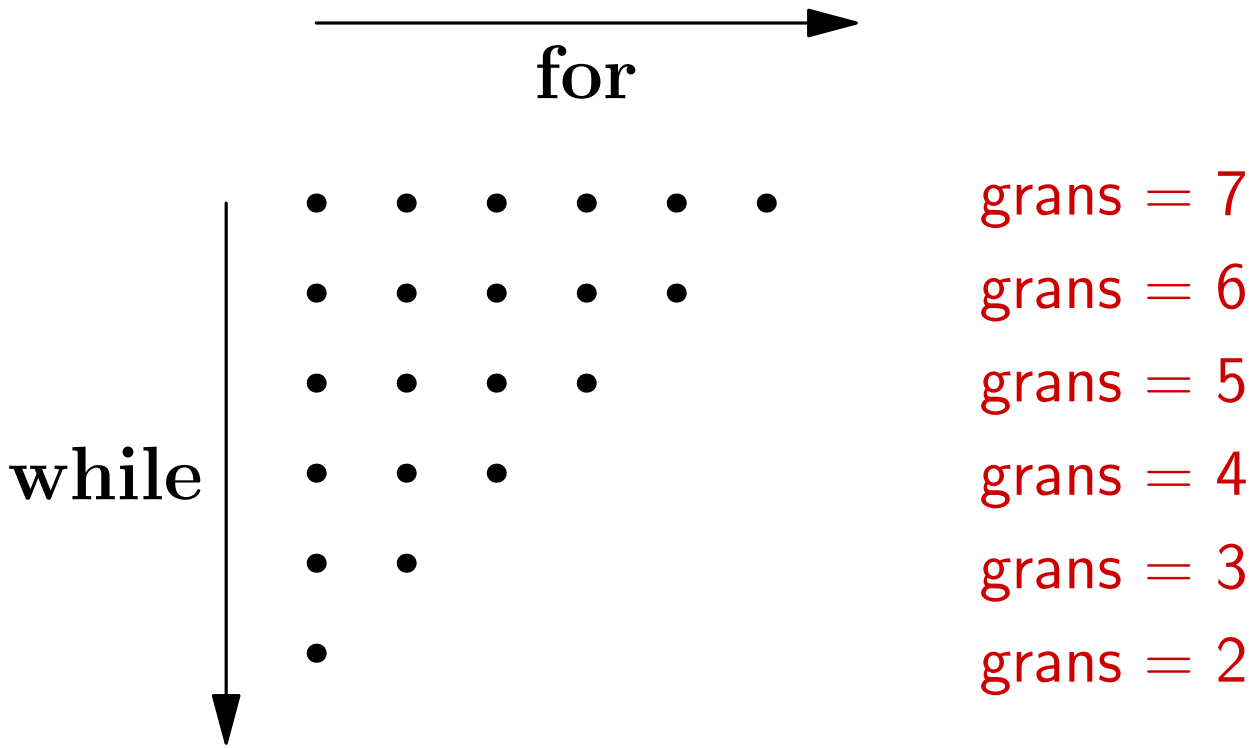
Bubbelsortering, tidsåtgång i antal jämförelser

smartare versionen med grans, bubbelsort2, n = 7



Bubbelsortering, tidsåtgång i antal jämförelser

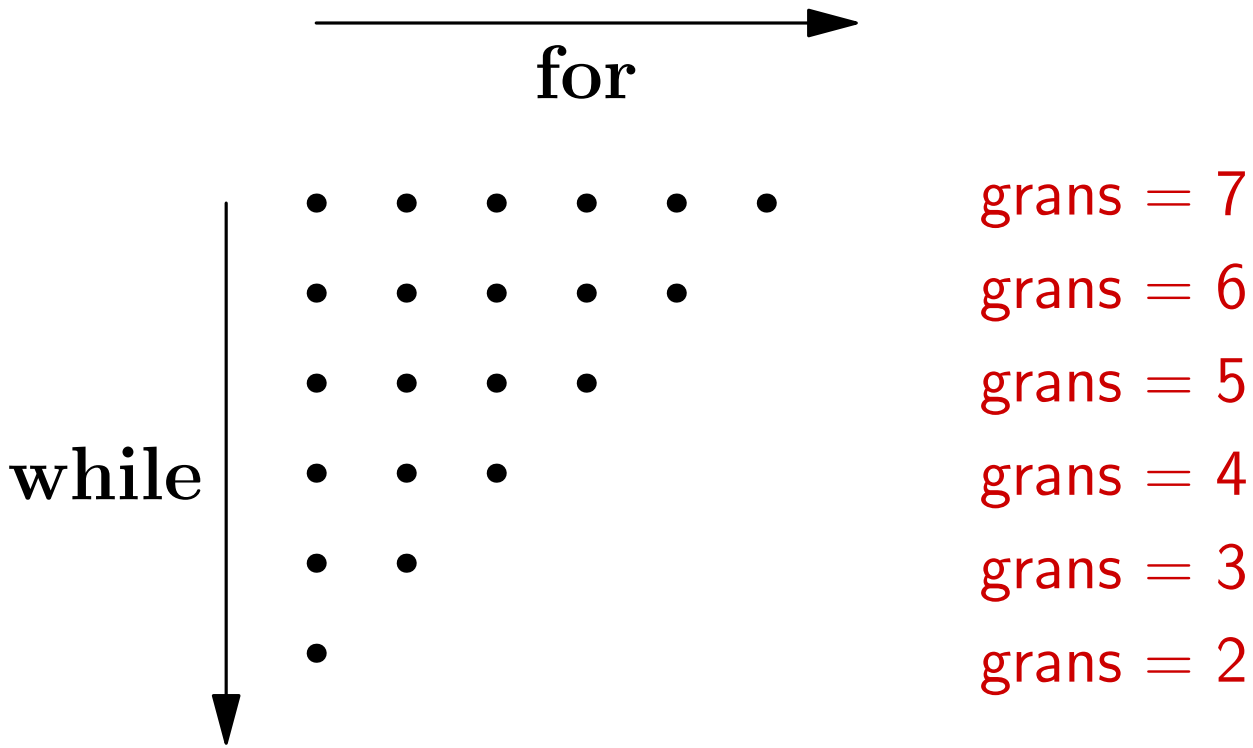
smartare versionen med grans, bubbelsort2, n = 7



- • = en jämförelse

Bubbelsortering, tidsåtgång i antal jämförelser

smartare versionen med grans, bubbelsort2, n = 7



- • = en jämförelse
- Antal •: $n(n-1)/2 \rightarrow \mathcal{O}(n^2)$

Insättningssortering

Insättningssortering

Icke-rekursiv formulering

Insättningssortering

Icke-rekursiv formulering

Sortera **lista**

Insättningssortering

Icke-rekursiv formulering

Sortera **lista**

- Starta med en tom resultatlista, **res = []**.

Insättningssortering

Icke-rekursiv formulering

Sortera **lista**

- Starta med en tom resultatlista, **res = []**.
- För varje element **e** i **lista**
  Sätt in **e** på rätt plats i **res** (*)

Insättningssortering

Icke-rekursiv formulering

Sortera **lista**

- Starta med en tom resultatlista, **res = []**.
- För varje element **e** i **lista**
  Sätt in **e** på rätt plats i **res** (*)

Tidsåtgång

- Insättning enligt (*): $\mathcal{O}(n)$

Insättningssortering

Icke-rekursiv formulering

Sortera **lista**

- Starta med en tom resultatlista, **res = []**.
- För varje element **e** i **lista**
  Sätt in **e** på rätt plats i **res** (*)

Tidsåtgång

- Insättning enligt (*): $\mathcal{O}(n)$
- n st steg $\rightarrow \mathcal{O}(n^2)$

Insättningssortering

Icke-rekursiv formulering

Sortera **lista**

- Starta med en tom resultatlista, **res = []**.
- För varje element **e** i **lista**
  Sätt in **e** på rätt plats i **res** (*)

Tidsåtgång

- Insättning enligt (*): $\mathcal{O}(n)$
- n st steg $\rightarrow \mathcal{O}(n^2)$

Skriv Pythonkoden som övning!

Insättningssortering

Insättningssortering

Rekursiv formulering

Insättningssortering

Rekursiv formulering

Sortera **lista**

Insättningssortering

Rekursiv formulering

Sortera **lista**

- **Basfall:**
  Endast ett (eller noll) element i **lista**? $\rightarrow$ Sortering klar.

Insättningssortering

Rekursiv formulering

Sortera **lista**

- **Basfall:**
  Endast ett (eller noll) element i **lista**? $\rightarrow$ Sortering klar.
- **Rekursiv tanke:**

Insättningssortering

Rekursiv formulering

Sortera **lista**

- **Basfall:**
  Endast ett (eller noll) element i **lista**? $\rightarrow$ Sortering klar.
- **Rekursiv tanke:**
  - Tag ut första elementet ur **lista**, **el**

Insättningssortering

Rekursiv formulering

Sortera **lista**

- **Basfall:**
  Endast ett (eller noll) element i **lista**? $\rightarrow$ Sortering klar.
- **Rekursiv tanke:**
  - Tag ut första elementet ur **lista**, **el**
  - Sortera resten av **lista**, alltså **lista[1:]**

Insättningssortering

Rekursiv formulering

Sortera **lista**

- **Basfall:**
  Endast ett (eller noll) element i **lista**? $\rightarrow$ Sortering klar.
- **Rekursiv tanke:**
  - Tag ut första elementet ur **lista**, **el**
  - Sortera resten av **lista**, alltså **lista[1:]**
  - Sätt in **el** på rätt plats (*)

Insättningssortering

Rekursiv formulering

Sortera **lista**

- **Basfall:**
  Endast ett (eller noll) element i **lista**? $\rightarrow$ Sortering klar.
- **Rekursiv tanke:**
  - Tag ut första elementet ur **lista**, **el**
  - Sortera resten av **lista**, alltså **lista[1:]**
  - Sätt in **el** på rätt plats (*)

Tidsåtgång

- Insättning enligt (*): $\mathcal{O}(n)$

Insättningssortering

Rekursiv formulering

Sortera **lista**

- **Basfall:**
  Endast ett (eller noll) element i **lista**? $\rightarrow$ Sortering klar.
- **Rekursiv tanke:**
  - Tag ut första elementet ur **lista**, **el**
  - Sortera resten av **lista**, alltså **lista[1:]**
  - Sätt in **el** på rätt plats (*)

Tidsåtgång

- Insättning enligt (*): $\mathcal{O}(n)$
- n st steg $\rightarrow \mathcal{O}(n^2)$

Insättningssortering, visar idén, slösar med minne

Insättningssortering, visar idén, slösar med minne

```
def insertsort(v):
    if len(v) <=1:          # basfall
        return v

    sorted = insertsort(v[1:])   # sortera
                                 # "svansen"
    ind = 0
    while ind < len(sorted) and \ # hitta
          v[0] >= sorted[ind]:    # plats
        ind += 1                  # till v[0]

    sorted.insert(ind,v[0])       # satt in v[0]

    return sorted
```

Urvalssortering

Urvalssortering

- Hitta minsta elementet i $v_0, \ldots, v_{n-1}$
- Byt plats på 0:e element och minsta element

Urvalssortering

- Hitta minsta elementet i $v_0, \ldots, v_{n-1}$
- Byt plats på 0:e element och minsta element

- Hitta minsta element i $v_1, \ldots, v_{n-1}$
- Byt plats på 1:a element och minsta element

Urvalssortering

- Hitta minsta elementet i $v_0, \ldots, v_{n-1}$
- Byt plats på 0:e element och minsta element

- Hitta minsta element i $v_1, \ldots, v_{n-1}$
- Byt plats på 1:a element och minsta element

och så vidare ...

Urvalssortering

- Hitta minsta elementet i $v_0, \ldots, v_{n-1}$
- Byt plats på 0:e element och minsta element

- Hitta minsta element i $v_1, \ldots, v_{n-1}$
- Byt plats på 1:a element och minsta element

och så vidare ...

- Till sist: Hitta minsta av v_{n-2}, v_{n-1}
- Byt plats på (n-2):a element och minsta

Urvalssortering

- Hitta minsta elementet i $v_0, \ldots, v_{n-1}$
- Byt plats på 0:e element och minsta element

- Hitta minsta element i $v_1, \ldots, v_{n-1}$
- Byt plats på 1:a element och minsta element

och så vidare ...

- Till sist: Hitta minsta av v_{n-2}, v_{n-1}
- Byt plats på (n-2):a element och minsta

Tidsåtgång

- $\mathcal{O}(n^2)$

Urvalsssortering

```
def selectionsort(v):

    n = len(v)
    for k in range(n-1):
        m = v[k]; im = k
        for i in range(k,n):
            if v[i] < m:
                m, im = v[i], i

        v[k], v[im] = v[im], v[k]
```

Urvalssortering

Rekursiv formulering

Urvalssortering

Rekursiv formulering

- Hitta största elementet och tag ut det

Urvalssortering

Rekursiv formulering

- Hitta största elementet och tag ut det
- Sortera det som återstår

Urvalssortering

Rekursiv formulering

- Hitta största elementet och tag ut det
- Sortera det som återstår
- Sätt in det största elementet sist

Urvalssortering

Rekursiv formulering

- Hitta största elementet och tag ut det
- Sortera det som återstår
- Sätt in det största elementet sist
- Basfallet är att listan har längden 1

Urvalssortering

Rekursiv formulering

- Hitta största elementet och tag ut det
- Sortera det som återstår
- Sätt in det största elementet sist
- Basfallet är att listan har längden 1

Tidsåtgång

- $\mathcal{O}(n^2)$

Urvalsssortering, rekursivt

```
def selectsortR(v):

    if len(v) <= 1:              # basfall
        return v

    m, mi = v[0], 0                      # hitta
    for i in range(1, len(v)):           # index
        if v[i] > m:                     # f.
            m, mi = v[i], i              # max-element

    del v[mi]                            # tag bort max

    return selectsortR(v) + [m]          # rekursion
```

Enkla sorteringsmetoder

Bubbelsortering	$\mathcal{O}(n^2)$
Insättningssortering	$\mathcal{O}(n^2)$
Urvalssortering	$\mathcal{O}(n^2)$

Samsortering – Mergesort

Samsortering – Mergesort

- **Rekursiv tanke:**

Samsortering – Mergesort

- **Rekursiv tanke:**
  - Dela på mitten

Samsortering – Mergesort

- **Rekursiv tanke:**
  - Dela på mitten
  - Sortera varje del med Mergesort

Samsortering – Mergesort

- **Rekursiv tanke:**
  - Dela på mitten
  - Sortera varje del med Mergesort
  - Sätt ihop delarna, kallas **merge**

Samsortering – Mergesort

- **Rekursiv tanke:**
  - Dela på mitten
  - Sortera varje del med Mergesort
  - Sätt ihop delarna, kallas **merge**
- **Basfall:**
  - Lista med 0 eller 1 element, gör ingenting

Mergesort

Mergesort

```
def mergesort(v):
    n = len(v)

    if n <= 1:
        return v

    mid = n//2
    lefty = mergesort(v[:mid])      # kopia
    righty = mergesort(v[mid:])     # kopia

    return merge(lefty, righty)
```

Mergesort

```
def mergesort(v):
    n = len(v)

    if n <= 1:
        return v

    mid = n//2
    lefty = mergesort(v[:mid])      # kopia
    righty = mergesort(v[mid:])     # kopia

    return merge(lefty, righty)
```

Algoritmen visas tydligt men funktionen slösar med minne!

Mergesort

```
def mergesort(v):
    n = len(v)

    if n <= 1:
        return v

    mid = n//2
    lefty = mergesort(v[:mid])      # kopia
    righty = mergesort(v[mid:])     # kopia

    return merge(lefty, righty)
```

Algoritmen visas tydligt men funktionen slösar med minne!
Varje `v[x:y]` skapar en kopia av (en del av) listan

Mergesort

```
def mergesort(v):
    n = len(v)

    if n <= 1:
        return v

    mid = n//2
    lefty = mergesort(v[:mid])      # kopia
    righty = mergesort(v[mid:])     # kopia

    return merge(lefty, righty)
```

Algoritmen visas tydligt men funktionen slösar med minne!
Varje `v[x:y]` skapar en kopia av (en del av) listan

Hjälpmetoden merge finns på nästa bild

Hur går merge till?

Antag två listor sorterade i stigande ordning

a = `[3, 8, 17, 25, 33, 68]`
b = `[1, 10, 12, 23]`

Hur går merge till?

Antag två listor sorterade i stigande ordning

a = `[3, 8, 17, 25, 33, 68]`
b = `[1, 10, 12, 23]`

Upprepa så länge ingen lista är slut:

Hur går merge till?

Antag två listor sorterade i stigande ordning

```
a = [3, 8, 17, 25, 33, 68]
b = [1, 10, 12, 23]
```

Upprepa så länge ingen lista är slut:

- Jämför listornas första värden `a[0]` och `b[0]`, tag **ut** det minsta värdet och lägg till resultatlistan.

Hur går merge till?

Antag två listor sorterade i stigande ordning

```
a = [3, 8, 17, 25, 33, 68]
b = [1, 10, 12, 23]
```

Upprepa så länge ingen lista är slut:

- Jämför listornas första värden `a[0]` och `b[0]`, tag **ut** det minsta värdet och lägg till resultatlistan.

Lägg den icke-tomma listan till resultatlistan

Hur går merge till?

Antag två listor sorterade i stigande ordning

a = `[3, 8, 17, 25, 33, 68]`
b = `[1, 10, 12, 23]`

Upprepa så länge ingen lista är slut:

- Jämför listornas första värden `a[0]` och `b[0]`, tag **ut** det minsta värdet och lägg till resultatlistan.

Lägg den icke-tomma listan till resultatlistan

`[]` `[3, 8, 17, 25, 33, 68]` `[1, 10, 12, 23]`

Hur går merge till?

Antag två listor sorterade i stigande ordning

```
a = [3, 8, 17, 25, 33, 68]
b = [1, 10, 12, 23]
```

Upprepa så länge ingen lista är slut:

- Jämför listornas första värden `a[0]` och `b[0]`, tag **ut** det minsta värdet och lägg till resultatlistan.

Lägg den icke-tomma listan till resultatlistan

```
[]  [3, 8, 17, 25, 33, 68]  [1, 10, 12, 23]
[1]  [3, 8, 17, 25, 33, 68]  [10, 12, 23]
```

Hur går merge till?

Antag två listor sorterade i stigande ordning

```
a = [3, 8, 17, 25, 33, 68]
b = [1, 10, 12, 23]
```

Upprepa så länge ingen lista är slut:

- Jämför listornas första värden `a[0]` och `b[0]`, tag **ut** det minsta värdet och lägg till resultatlistan.

Lägg den icke-tomma listan till resultatlistan

```
[]  [3, 8, 17, 25, 33, 68]  [1, 10, 12, 23]
[1]  [3, 8, 17, 25, 33, 68]  [10, 12, 23]
[1, 3]  [8, 17, 25, 33, 68]  [10, 12, 23]
```

Hur går merge till?

Antag två listor sorterade i stigande ordning

a = `[3, 8, 17, 25, 33, 68]`
b = `[1, 10, 12, 23]`

Upprepa så länge ingen lista är slut:

- Jämför listornas första värden `a[0]` och `b[0]`, tag **ut** det minsta värdet och lägg till resultatlistan.

Lägg den icke-tomma listan till resultatlistan

```
[]  [3, 8, 17, 25, 33, 68]  [1, 10, 12, 23]
[1]  [3, 8, 17, 25, 33, 68]  [10, 12, 23]
[1, 3]  [8, 17, 25, 33, 68]  [10, 12, 23]
[1, 3, 8]  [17, 25, 33, 68]  [10, 12, 23]
```

...

Hur går merge till?

Antag två listor sorterade i stigande ordning

a = `[3, 8, 17, 25, 33, 68]`
b = `[1, 10, 12, 23]`

Upprepa så länge ingen lista är slut:

- Jämför listornas första värden `a[0]` och `b[0]`, tag **ut** det minsta värdet och lägg till resultatlistan.

Lägg den icke-tomma listan till resultatlistan

```
[]  [3, 8, 17, 25, 33, 68]  [1, 10, 12, 23]
[1]  [3, 8, 17, 25, 33, 68]  [10, 12, 23]
[1, 3]  [8, 17, 25, 33, 68]  [10, 12, 23]
[1, 3, 8]  [17, 25, 33, 68]  [10, 12, 23]
 ...
[1, 3, 8, 10, 12, 17, 23]  [25, 33, 68] []
```

Hur går merge till?

Antag två listor sorterade i stigande ordning

a = `[3, 8, 17, 25, 33, 68]`
b = `[1, 10, 12, 23]`

Upprepa så länge ingen lista är slut:

- Jämför listornas första värden `a[0]` och `b[0]`, tag **ut** det minsta värdet och lägg till resultatlistan.

Lägg den icke-tomma listan till resultatlistan

```
[]  [3, 8, 17, 25, 33, 68]  [1, 10, 12, 23]
[1]  [3, 8, 17, 25, 33, 68]  [10, 12, 23]
[1, 3]  [8, 17, 25, 33, 68]  [10, 12, 23]
[1, 3, 8]  [17, 25, 33, 68]  [10, 12, 23]
 ...
[1, 3, 8, 10, 12, 17, 23]  [25, 33, 68] []
[1, 3, 8, 10, 12, 17, 23, 25, 33, 68]  [] []
```

Hjälpmetoden merge

Indata **a** och **b** måste vara sorterade listor

Hjälpmetoden merge

Indata **a** och **b** måste vara sorterade listor

Resultat: lista där **a** och **b** samsorterats

Hjälpmetoden merge

Indata **a** och **b** måste vara sorterade listor

Resultat: lista där **a** och **b** samsorterats

```
def merge(a,b):
    v = []
    while a!=[] and b!=[]:
        if a[0] < b[0]:
            v.append(a.pop(0))
        else:
            v.append(b.pop(0))

    return v + a + b    # a or b is empty
```

Mergesort – Samsortering

Mergesort – Samsortering

Hur lång tid tar det?

Mergesort – Samsortering

Hur lång tid tar det?

Listans längd delas med två i varje steg ned till längd 1

Mergesort – Samsortering

Hur lång tid tar det?

Listans längd delas med två i varje steg ned till längd 1

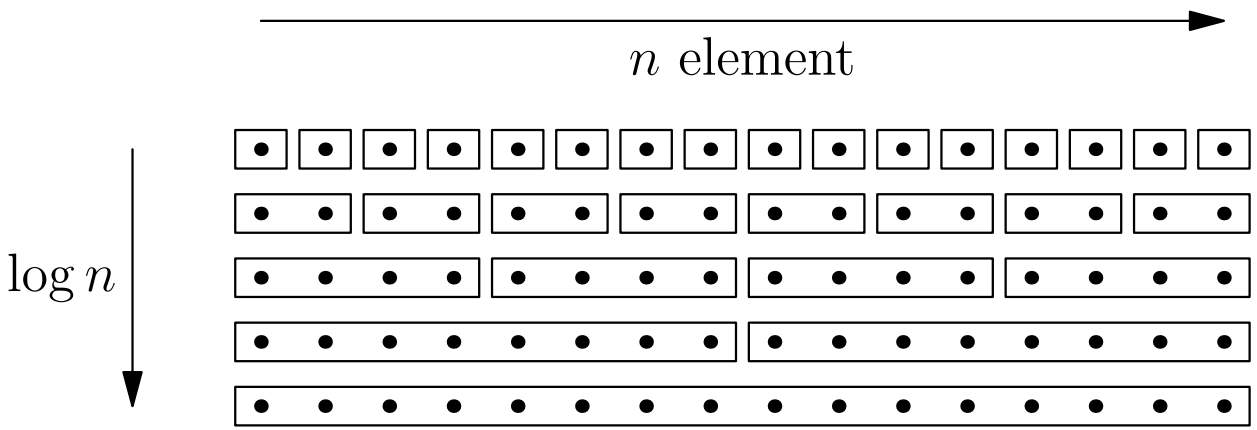
Jämför med binärsökning

Mergesort – Samsortering

Hur lång tid tar det?

Listans längd delas med två i varje steg ned till längd 1

Jämför med binärsökning

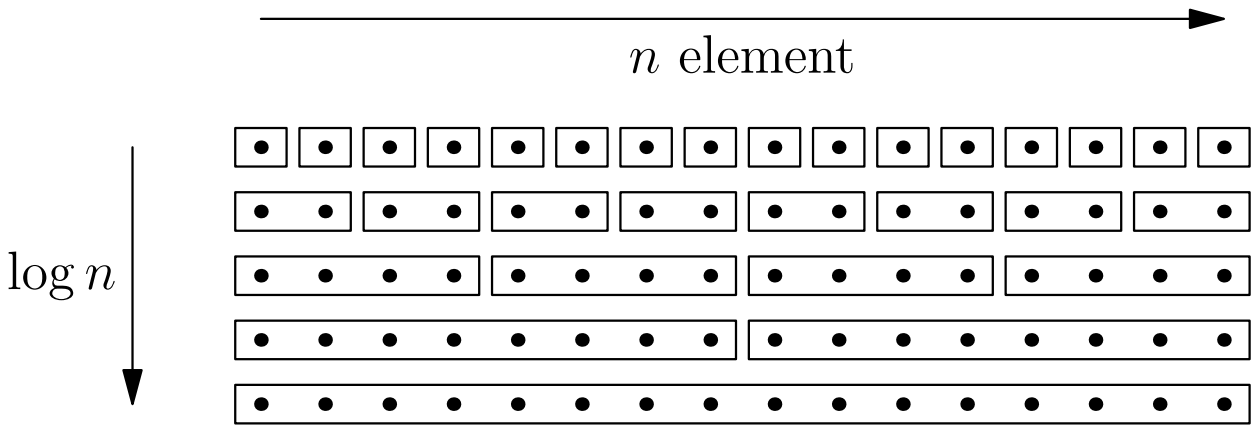


Mergesort – Samsortering

Hur lång tid tar det?

Listans längd delas med två i varje steg ned till längd 1

Jämför med binärsökning



- $\mathcal{O}(n \log n)$

Quicksort

Quicksort

- Välj ett element **el**, kallas pivotelement

Quicksort

- Välj ett element **el**, kallas pivotelement
- Lägg element som är mindre än **el** till vänster och element som är större eller lika till höger

Quicksort

- Välj ett element **el**, kallas pivotelement
- Lägg element som är mindre än **el** till vänster och element som är större eller lika till höger
- Sortera de två delarna med quicksort, sätt ihop med **el** mellan

Quicksort

- Välj ett element **el**, kallas pivotelement
- Lägg element som är mindre än **el** till vänster och element som är större eller lika till höger
- Sortera de två delarna med quicksort, sätt ihop med **el** mellan
- Basfall: 0 eller 1 element

Quicksort

- Välj ett element **el**, kallas pivotelement
- Lägg element som är mindre än **el** till vänster och element som är större eller lika till höger
- Sortera de två delarna med quicksort, sätt ihop med **el** mellan
- Basfall: 0 eller 1 element

Exempel:

Quicksort

- Välj ett element **el**, kallas pivotelement
- Lägg element som är mindre än **el** till vänster och element som är större eller lika till höger
- Sortera de två delarna med quicksort, sätt ihop med **el** mellan
- Basfall: 0 eller 1 element

Exempel:

`5 2 0 8 21 13 1 3 1`

Quicksort

- Välj ett element **el**, kallas pivotelement
- Lägg element som är mindre än **el** till vänster och element som är större eller lika till höger
- Sortera de två delarna med quicksort, sätt ihop med **el** mellan
- Basfall: 0 eller 1 element

Exempel:

5 2 0 8 21 13 1 3 1

2 0 1 3 1 5 8 21 13

Quicksort

- Välj ett element **el**, kallas pivotelement
- Lägg element som är mindre än **el** till vänster och element som är större eller lika till höger
- Sortera de två delarna med quicksort, sätt ihop med **el** mellan
- Basfall: 0 eller 1 element

Exempel:

```
5 2 0 8 21 13 1 3 1
2 0 1 3 1 5 8 21 13     partitionering
```

Quicksort

- Välj ett element **el**, kallas pivotelement
- Lägg element som är mindre än **el** till vänster och element som är större eller lika till höger
- Sortera de två delarna med quicksort, sätt ihop med **el** mellan
- Basfall: 0 eller 1 element

Exempel:

```
5 2 0 8 21 13 1 3 1

2 0 1 3 1 5 8 21 13     partitionering
  < 5          >= 5
```

Quicksort

- Välj ett element **el**, kallas pivotelement
- Lägg element som är mindre än **el** till vänster och element som är större eller lika till höger
- Sortera de två delarna med quicksort, sätt ihop med **el** mellan
- Basfall: 0 eller 1 element

Exempel:

```
5 2 0 8 21 13 1 3 1

2 0 1 3 1 5 8 21 13     partitionering
  < 5           >= 5
```

Gör sen likadant med varje del!

Quicksort

```
def quicksort(v):
    if len(v) <= 1:          # Basfall
        return v

    lefty, righty = [],[]
    piv = v[0]
    for el in v[1:]:
        if el < piv:
            lefty.append(el)
        else:
            righty.append(el)

    return quicksort(lefty) +
           [piv] +
           quicksort(righty)
```

Funktionen visar algoritmen men slösar med minne

Quicksort

Hur lång tid tar det?

Quicksort

Hur lång tid tar det?

- Medelfallet: $\mathcal{O}(n \ log \ n)$

Quicksort

Hur lång tid tar det?

- Medelfallet: $\mathcal{O}(n \ log \ n)$
- Sämsta fallet: $\mathcal{O}(n^2)$

Quicksort

Hur lång tid tar det?

- Medelfallet: $\mathcal{O}(n \ log \ n)$
- Sämsta fallet: $\mathcal{O}(n^2)$

  T.ex. om data är sorterade och
  första elementet väljs som jämförelseelement.

Quicksort

Hur lång tid tar det?

- Medelfallet: $\mathcal{O}(n \log n)$
- Sämsta fallet: $\mathcal{O}(n^2)$

  T.ex. om data är sorterade och
  första elementet väljs som jämförelseelement.
- Välj gärna medianen av minst tre element som jämförelseelement.

Quicksort

Hur lång tid tar det?

- Medelfallet: $\mathcal{O}(n \log n)$
- Sämsta fallet: $\mathcal{O}(n^2)$

  T.ex. om data är sorterade och
  första elementet väljs som jämförelseelement.
- Välj gärna medianen av minst tre element som jämförelseelement.
- Använd enklare metod på korta (ca ≤ 50) delföljder

Tydlighet kontra effektivitet

Tydlighet kontra effektivitet

- Pythonkoden för **mergesort** och **quicksort** är kort.

Tydlighet kontra effektivitet

- Pythonkoden för **mergesort** och **quicksort** är kort.
- Algoritmerna visas mycket tydligt.

Tydlighet kontra effektivitet

- Pythonkoden för **mergesort** och **quicksort** är kort.
- Algoritmerna visas mycket tydligt.
- Delar av listor kopieras ofta, t.ex.
  `v[:mid]` tar en kopia av halva listan `v`.

Tydlighet kontra effektivitet

- Pythonkoden för **mergesort** och **quicksort** är kort.
- Algoritmerna visas mycket tydligt.
- Delar av listor kopieras ofta, t.ex.
  `v[:mid]` tar en kopia av halva listan `v`.
- Kopieringen slösar med minne.

Tydlighet kontra effektivitet

- Pythonkoden för **mergesort** och **quicksort** är kort.
- Algoritmerna visas mycket tydligt.
- Delar av listor kopieras ofta, t.ex.
  `v[:mid]` tar en kopia av halva listan `v`.
- Kopieringen slösar med minne.
- En minnessnål implementation
  gör alla operationer inom **samma lista**.

Tydlighet kontra effektivitet

- Pythonkoden för **mergesort** och **quicksort** är kort.
- Algoritmerna visas mycket tydligt.
- Delar av listor kopieras ofta, t.ex.
  `v[:mid]` tar en kopia av halva listan `v`.
- Kopieringen slösar med minne.
- En minnessnål implementation
  gör alla operationer inom **samma lista**.
- En sådan **in place** - implementation visar inte
  algoritmidén lika tydligt.

Tydlighet kontra effektivitet

- Pythonkoden för **mergesort** och **quicksort** är kort.
- Algoritmerna visas mycket tydligt.
- Delar av listor kopieras ofta, t.ex.
  `v[:mid]` tar en kopia av halva listan `v`.
- Kopieringen slösar med minne.
- En minnessnål implementation
  gör alla operationer inom **samma lista**.
- En sådan **in place** - implementation visar inte algoritmidén lika tydligt.
- Något **in place** – exempel visas på övning.

Kan man sortera snabbare än $\mathcal{O}(n \log n)$?

Kan man sortera snabbare än $\mathcal{O}(n \log n)$?

- Ja, om nyckelvärdena endast antar ett litet antal värden.

Kan man sortera snabbare än $\mathcal{O}(n \log n)$?

- Ja, om nyckelvärdena endast antar ett litet antal värden.
  Distribution Sort – Räknesortering

Kan man sortera snabbare än $\mathcal{O}(n \log n)$?

- Ja, om nyckelvärdena endast antar ett litet antal värden.

  Distribution Sort – Räknesortering

  Exempel: Sortera 100000 personer

Kan man sortera snabbare än $\mathcal{O}(n \log n)$?

- Ja, om nyckelvärdena endast antar ett litet antal värden.

  Distribution Sort – Räknesortering

  Exempel: Sortera 100000 personer

  - efter födelsedag (366 alternativ)

Kan man sortera snabbare än $\mathcal{O}(n \log n)$?

- Ja, om nyckelvärdena endast antar ett litet antal värden.

  Distribution Sort – Räknesortering

  Exempel: Sortera 100000 personer

  - efter födelsedag (366 alternativ)
  - efter födelsemånad (12 alternativ)

Kan man sortera snabbare än $\mathcal{O}(n \log n)$?

- Ja, om nyckelvärdena endast antar ett litet antal värden.

  Distribution Sort – Räknesortering

  Exempel: Sortera 100000 personer

  - efter födelsedag (366 alternativ)
  - efter födelsemånad (12 alternativ)

  Tidsåtgång: $\mathcal{O}(n)$

Kan man sortera snabbare än $\mathcal{O}(n\ log\ n)$?

- Ja, om nyckelvärdena endast antar ett litet antal värden.

  Distribution Sort – Räknesortering

  Exempel: Sortera 100000 personer

  - efter födelsedag (366 alternativ)
  - efter födelsemånad (12 alternativ)

  Tidsåtgång: $\mathcal{O}(n)$

- Ja, om alla nycklar är lika långa.

  Radix sort – Hålkortssortering

  Tidsåtgång: $\mathcal{O}(kn), k =$ nyckelns längd

Radix Sort – exempel

Radix Sort – exempel

Sortera de tvåsiffriga talen

79 13 27 48 86 10 85 93 65 94 36 50 19 43 86 57 47 15 19 4

Radix Sort – exempel

Sortera de tvåsiffriga talen

79 13 27 48 86 10 85 93 65 94 36 50 19 43 86 57 47 15 19 4

Ordna i grupper efter entalssiffran:

Radix Sort – exempel

Sortera de tvåsiffriga talen

79 13 27 48 86 10 85 93 65 94 36 50 19 43 86 57 47 15 19 4

Ordna i grupper efter entalssiffran:

0: 10 50 1: 2: 3: 13 93 43 4: 94 04 5: 85 65 15

6: 86 36 86 7: 27 57 47 8: 48 9: 79 19 19

Radix Sort – exempel

Sortera de tvåsiffriga talen

79 13 27 48 86 10 85 93 65 94 36 50 19 43 86 57 47 15 19 4

Ordna i grupper efter entalssiffran:

0: 10 50 1: 2: 3: 13 93 43 4: 94 04 5: 85 65 15

6: 86 36 86 7: 27 57 47 8: 48 9: 79 19 19

Behåll ordningen inom grupperna ovan och gruppera efter tiotalssiffran:

Radix Sort – exempel

Sortera de tvåsiffriga talen

79 13 27 48 86 10 85 93 65 94 36 50 19 43 86 57 47 15 19 4

Ordna i grupper efter entalssiffran:

0: 10 50 1: 2: 3: 13 93 43 4: 94 04 5: 85 65 15

6: 86 36 86 7: 27 57 47 8: 48 9: 79 19 19

Behåll ordningen inom grupperna ovan och gruppera efter tiotalssiffran:

0: 04 1: 10 13 15 19 19 2: 27 3: 36 4: 43 47 48

5: 50 57 6: 65 7: 79 8: 85 86 86 9: 93 94

Radix Sort – exempel

Sortera de tvåsiffriga talen

79 13 27 48 86 10 85 93 65 94 36 50 19 43 86 57 47 15 19 4

Ordna i grupper efter entalssiffran:

0: 10 50 1: 2: 3: 13 93 43 4: 94 04 5: 85 65 15

6: 86 36 86 7: 27 57 47 8: 48 9: 79 19 19

Behåll ordningen inom grupperna ovan och gruppera efter tiotalssiffran:

0: 04 1: 10 13 15 19 19 2: 27 3: 36 4: 43 47 48

5: 50 57 6: 65 7: 79 8: 85 86 86 9: 93 94

SORTERAT!

Radix Sort – exempel

Sortera de tvåsiffriga talen

79 13 27 48 86 10 85 93 65 94 36 50 19 43 86 57 47 15 19 4

Ordna i grupper efter entalssiffran:

0: 10 50 1: 2: 3: 13 93 43 4: 94 04 5: 85 65 15

6: 86 36 86 7: 27 57 47 8: 48 9: 79 19 19

Behåll ordningen inom grupperna ovan och gruppera efter tiotalssiffran:

0: 04 1: 10 13 15 19 19 2: 27 3: 36 4: 43 47 48

5: 50 57 6: 65 7: 79 8: 85 86 86 9: 93 94

SORTERAT!

För k-siffriga tal behövs k genomgångar, $\mathcal{O}(kn)$.

Problem för algoritmanalys – ANAGRAM

- Avgör om en sträng s_1 är en permutation av en annan sträng s_2.

Problem för algoritmanalys – ANAGRAM

- Avgör om en sträng s_1 är en permutation av en annan sträng s_2.
- Visa fyra olika algoritmer och jämför deras prestanda

Problem för algoritmanalys – ANAGRAM

- Avgör om en sträng s_1 är en permutation av en annan sträng s_2.
- Visa fyra olika algoritmer och jämför deras prestanda

Anagramexempel:

Problem för algoritmanalys – ANAGRAM

- Avgör om en sträng s_1 är en permutation av en annan sträng s_2.
- Visa fyra olika algoritmer och jämför deras prestanda

Anagramexempel:

HOSTANFALL –

Problem för algoritmanalys – ANAGRAM

- Avgör om en sträng s_1 är en permutation av en annan sträng s_2.
- Visa fyra olika algoritmer och jämför deras prestanda

Anagramexempel:

HOSTANFALL – HALLONSAFT

Problem för algoritmanalys – ANAGRAM

- Avgör om en sträng s_1 är en permutation av en annan sträng s_2.
- Visa fyra olika algoritmer och jämför deras prestanda

Anagramexempel:

HOSTANFALL – HALLONSAFT
MIKAEL –

Problem för algoritmanalys – ANAGRAM

- Avgör om en sträng s_1 är en permutation av en annan sträng s_2.
- Visa fyra olika algoritmer och jämför deras prestanda

Anagramexempel:

HOSTANFALL – HALLONSAFT
MIKAEL – ALKEMI

Problem för algoritmanalys – ANAGRAM

- Avgör om en sträng s_1 är en permutation av en annan sträng s_2.
- Visa fyra olika algoritmer och jämför deras prestanda

Anagramexempel:

HOSTANFALL – HALLONSAFT
MIKAEL – ALKEMI
RÅDHUS –

Problem för algoritmanalys – ANAGRAM

- Avgör om en sträng s_1 är en permutation av en annan sträng s_2.
- Visa fyra olika algoritmer och jämför deras prestanda

Anagramexempel:

HOSTANFALL – HALLONSAFT
MIKAEL – ALKEMI
RÅDHUS – DÅRHUS

Problem för algoritmanalys – ANAGRAM

- Avgör om en sträng s_1 är en permutation av en annan sträng s_2.
- Visa fyra olika algoritmer och jämför deras prestanda

Anagramexempel:

HOSTANFALL – HALLONSAFT
MIKAEL – ALKEMI
RÅDHUS – DÅRHUS
MADAM CURIE –

Problem för algoritmanalys – ANAGRAM

- Avgör om en sträng s_1 är en permutation av en annan sträng s_2.
- Visa fyra olika algoritmer och jämför deras prestanda

Anagramexempel:

HOSTANFALL – HALLONSAFT
MIKAEL – ALKEMI
RÅDHUS – DÅRHUS
MADAM CURIE – RADIUM CAME

Problem för algoritmanalys – ANAGRAM

- Avgör om en sträng s_1 är en permutation av en annan sträng s_2.
- Visa fyra olika algoritmer och jämför deras prestanda

Anagramexempel:

HOSTANFALL – HALLONSAFT
MIKAEL – ALKEMI
RÅDHUS – DÅRHUS
MADAM CURIE – RADIUM CAME
LISTEN –

Problem för algoritmanalys – ANAGRAM

- Avgör om en sträng s_1 är en permutation av en annan sträng s_2.
- Visa fyra olika algoritmer och jämför deras prestanda

Anagramexempel:

HOSTANFALL – HALLONSAFT
MIKAEL – ALKEMI
RÅDHUS – DÅRHUS
MADAM CURIE – RADIUM CAME
LISTEN – SILENT

Problem för algoritmanalys – ANAGRAM

- Avgör om en sträng s_1 är en permutation av en annan sträng s_2.
- Visa fyra olika algoritmer och jämför deras prestanda

Anagramexempel:

HOSTANFALL – HALLONSAFT
MIKAEL – ALKEMI
RÅDHUS – DÅRHUS
MADAM CURIE – RADIUM CAME
LISTEN – SILENT
TEACHING –

Problem för algoritmanalys – ANAGRAM

- Avgör om en sträng s_1 är en permutation av en annan sträng s_2.
- Visa fyra olika algoritmer och jämför deras prestanda

Anagramexempel:

HOSTANFALL – HALLONSAFT
MIKAEL – ALKEMI
RÅDHUS – DÅRHUS
MADAM CURIE – RADIUM CAME
LISTEN – SILENT
TEACHING – CHEATING

Anagram – Idé 1: Avprickning

Anagram – Idé 1: Avprickning

- För varje tecken i s_1
  - Hitta samma tecken i s_2
  - Stryk tecknet ur s_2

Anagram – Idé 1: Avprickning

- För varje tecken i s_1
  - Hitta samma tecken i s_2
  - Stryk tecknet ur s_2
- Om alla tecken i s_2 strukits så är s_1 och s_2 anagram.

Anagram – Idé 1: Avprickning

- För varje tecken i s_1
  - Hitta samma tecken i s_2
  - Stryk tecknet ur s_2
- Om alla tecken i s_2 strukits så är s_1 och s_2 anagram.
- Tidsåtgång:

Anagram – Idé 1: Avprickning

- För varje tecken i s_1
  - Hitta samma tecken i s_2
  - Stryk tecknet ur s_2
- Om alla tecken i s_2 strukits så är s_1 och s_2 anagram.
- Tidsåtgång:

  För varje s i s_1: (n st)

  Leta efter s i s_2 med linjärsökning, $\mathcal{O}(n)$

Anagram – Idé 1: Avprickning

- För varje tecken i s_1
  - Hitta samma tecken i s_2
  - Stryk tecknet ur s_2
- Om alla tecken i s_2 strukits så är s_1 och s_2 anagram.
- Tidsåtgång:

  För varje s i s_1: (n st)

  Leta efter s i s_2 med linjärsökning, $\mathcal{O}(n)$

  Totalt $n * \mathcal{O}(n) = \mathcal{O}(n^2)$

Anagram – Idé 1: Avprickning

```
def avprickning(s1,s2):
    if len(s1) != len(s2):
        return False
    l2 = list(s2)
    for s in s1:
        found = False
        for i in range(len(l2)):
            if l2[i] == s:
                l2[i] = None
                found = True
                break
        if not found:
            return False

    return True
```

Anagram – Idé 1: Avprickning

```
def avprickning(s1,s2):
    if len(s1) != len(s2):
        return False
    l2 = list(s2)   # skapa lista f. avprickning
    for s in s1:
        found = False
        for i in range(len(l2)):
            if l2[i] == s:      # hittat s
                l2[i] = None    # pricka av
                found = True
                break           # ut ur inre for
        if not found:           # om ngt tecken inte
            return False        # finns, ret. False

    return True         # hit kommer man bara om
                        # alla s i s1 prickats av
```

Anagram – Idé 2: Sortera och jämför

Anagram – Idé 2: Sortera och jämför

- Sortera bokstäverna i s_1
- Sortera bokstäverna i s_2

Anagram – Idé 2: Sortera och jämför

- Sortera bokstäverna i s_1
- Sortera bokstäverna i s_2
- Är de sorterade strängarna lika?

Anagram – Idé 2: Sortera och jämför

- Sortera bokstäverna i s_1
- Sortera bokstäverna i s_2
- Är de sorterade strängarna lika?

- Tidsåtgång:

Anagram – Idé 2: Sortera och jämför

- Sortera bokstäverna i s_1
- Sortera bokstäverna i s_2
- Är de sorterade strängarna lika?

- Tidsåtgång:

  Sortering: $\mathcal{O}(n\ log\ n) + \mathcal{O}(n\ log\ n) = \mathcal{O}(n\ log\ n)$

Anagram – Idé 2: Sortera och jämför

- Sortera bokstäverna i s_1
- Sortera bokstäverna i s_2
- Är de sorterade strängarna lika?

- Tidsåtgång:

  Sortering: $\mathcal{O}(n\ log\ n) + \mathcal{O}(n\ log\ n) = \mathcal{O}(n\ log\ n)$

  Jämföra strängarna: $\mathcal{O}(n)$

Anagram – Idé 2: Sortera och jämför

- Sortera bokstäverna i s_1
- Sortera bokstäverna i s_2
- Är de sorterade strängarna lika?

- Tidsåtgång:

  Sortering: $\mathcal{O}(n \ log \ n) + \mathcal{O}(n \ log \ n) = \mathcal{O}(n \ log \ n)$

  Jämföra strängarna: $\mathcal{O}(n)$

  Totalt: $\mathcal{O}(n \ log \ n)$

Anagram – Idé 2: Sortera och jämför

```
def sortjfr(s1,s2):
    return sorted(s1) == sorted(s2)
```

Anagram – Idé 2: Sortera och jämför

```
def sortjfr(s1,s2):
    return sorted(s1) == sorted(s2)
```

Den inbyggda funktionen **sorted(seq)** returnerar en lista.

Anagram – Idé 2: Sortera och jämför

```
def sortjfr(s1,s2):
    return sorted(s1) == sorted(s2)
```

Den inbyggda funktionen **sorted(seq)** returnerar en lista.

Listan innehåller elementen från **seq**, sorterade.

Anagram – Idé 2: Sortera och jämför

```
def sortjfr(s1,s2):
    return sorted(s1) == sorted(s2)
```

Den inbyggda funktionen **sorted(seq)** returnerar en lista.

Listan innehåller elementen från **seq**, sorterade.

seq påverkas inte.

Anagram – Idé 2: Sortera och jämför

```
def sortjfr(s1,s2):
    return sorted(s1) == sorted(s2)
```

Den inbyggda funktionen **sorted(seq)** returnerar en lista.

Listan innehåller elementen från **seq**, sorterade.

seq påverkas inte.

Prova i Pythontolken!

Anagram – Idé 3: Uttömmande sökning

Anagram – Idé 3: Uttömmande sökning

- Generera alla permutationer av s_1

Anagram – Idé 3: Uttömmande sökning

- Generera alla permutationer av s_1
- Jämför alla med s_2

Anagram – Idé 3: Uttömmande sökning

- Generera alla permutationer av s_1
- Jämför alla med s_2

Hur många permutationer finns det av en sträng med n tecken?

Anagram – Idé 3: Uttömmande sökning

- Generera alla permutationer av s_1
- Jämför alla med s_2

Hur många permutationer finns det av en sträng med n tecken? Svar: n!

Anagram – Idé 3: Uttömmande sökning

- Generera alla permutationer av s_1
- Jämför alla med s_2

Hur många permutationer finns det av en sträng med n tecken? Svar: n!

- Tidsåtgång:

  Bara jämförelse mellan de n! permutationerna av s_1 och strängen s_2 kräver $\mathcal{O}(n!)$

  Total tidsåtgång: minst $\mathcal{O}(n!)$

Anagram – Idé 4: Räkna och jämför

Anagram – Idé 4: Räkna och jämför

- Nollställ två räknare, `count1` och `count2`,

Anagram – Idé 4: Räkna och jämför

- Nollställ två räknare, `count1` och `count2`, varje räknare är en lista som kan räkna *alla möjliga bokstäver*

Anagram – Idé 4: Räkna och jämför

- Nollställ två räknare, `count1` och `count2`, varje räknare är en lista som kan räkna *alla möjliga bokstäver*
- Räkna bokstäverna i s_1 i `count1`
- Räkna bokstäverna i s_2 i `count2`

Anagram – Idé 4: Räkna och jämför

- Nollställ två räknare, `count1` och `count2`, varje räknare är en lista som kan räkna *alla möjliga bokstäver*
- Räkna bokstäverna i s_1 i `count1`
- Räkna bokstäverna i s_2 i `count2`
- Om räknarna är lika så är s_1 och s_2 anagram

Anagram – Idé 4: Räkna och jämför

- Nollställ två räknare, `count1` och `count2`, varje räknare är en lista som kan räkna *alla möjliga bokstäver*
- Räkna bokstäverna i s_1 i `count1`
- Räkna bokstäverna i s_2 i `count2`
- Om räknarna är lika så är s_1 och s_2 anagram

- Tidsåtgång:

Anagram – Idé 4: Räkna och jämför

- Nollställ två räknare, `count1` och `count2`, varje räknare är en lista som kan räkna *alla möjliga bokstäver*
- Räkna bokstäverna i s_1 i `count1`
- Räkna bokstäverna i s_2 i `count2`
- Om räknarna är lika så är s_1 och s_2 anagram

- Tidsåtgång:

  Räkna bokstäver i s_1: `n` tilldelningar

  Räkna bokstäver i s_2: `n` tilldelningar

Anagram – Idé 4: Räkna och jämför

- Nollställ två räknare, `count1` och `count2`, varje räknare är en lista som kan räkna *alla möjliga bokstäver*
- Räkna bokstäverna i s_1 i `count1`
- Räkna bokstäverna i s_2 i `count2`
- Om räknarna är lika så är s_1 och s_2 anagram

- Tidsåtgång:

  Räkna bokstäver i s_1: `n` tilldelningar

  Räkna bokstäver i s_2: `n` tilldelningar

  Jämföra `count1` och `count2`: `256` jämförelser:

Anagram – Idé 4: Räkna och jämför

- Nollställ två räknare, `count1` och `count2`, varje räknare är en lista som kan räkna *alla möjliga bokstäver*
- Räkna bokstäverna i s_1 i `count1`
- Räkna bokstäverna i s_2 i `count2`
- Om räknarna är lika så är s_1 och s_2 anagram

- Tidsåtgång:

  Räkna bokstäver i s_1: `n` tilldelningar

  Räkna bokstäver i s_2: `n` tilldelningar

  Jämföra `count1` och `count2`: `256` jämförelser: $\mathcal{O}(1)$

Anagram – Idé 4: Räkna och jämför

- Nollställ två räknare, `count1` och `count2`, varje räknare är en lista som kan räkna *alla möjliga bokstäver*
- Räkna bokstäverna i s_1 i `count1`
- Räkna bokstäverna i s_2 i `count2`
- Om räknarna är lika så är s_1 och s_2 anagram

- Tidsåtgång:

  Räkna bokstäver i s_1: `n` tilldelningar

  Räkna bokstäver i s_2: `n` tilldelningar

  Jämföra `count1` och `count2`: `256` jämförelser: $\mathcal{O}(1)$

  Totalt: $\mathcal{O}(n)$

Anagram – Idé 4: Räkna och jämför

```
def countCmp(s1,s2):
    count1 = [0]*256
    count2 = [0]*256

    for s in s1:
        count1[ord(s)] += 1

    for s in s2:
        count2[ord(s)] += 1

    return count1 == count2
```

Anagram – sammanfattning

Vilken algoritm är snabbast för långa strängar?

Anagram – sammanfattning

Vilken algoritm är snabbast för långa strängar?

- Avprickning: $\mathcal{O}(n^2)$

Anagram – sammanfattning

Vilken algoritm är snabbast för långa strängar?

- Avprickning: $\mathcal{O}(n^2)$
- Sortera och jämför: $\mathcal{O}(n \ log \ n)$

Anagram – sammanfattning

Vilken algoritm är snabbast för långa strängar?

- Avprickning: $\mathcal{O}(n^2)$
- Sortera och jämför: $\mathcal{O}(n \ log \ n)$
- Uttömmande sökning: $\mathcal{O}(n!)$

Anagram – sammanfattning

Vilken algoritm är snabbast för långa strängar?

- Avprickning: $\mathcal{O}(n^2)$
- Sortera och jämför: $\mathcal{O}(n \ log \ n)$
- Uttömmande sökning: $\mathcal{O}(n!)$
- Räkna och jämför: $\mathcal{O}(n)$

Anagram – sammanfattning

Vilken algoritm är snabbast för långa strängar?

- Avprickning: $\mathcal{O}(n^2)$
- Sortera och jämför: $\mathcal{O}(n \ log \ n)$
- Uttömmande sökning: $\mathcal{O}(n!)$
- Räkna och jämför: $\mathcal{O}(n)$ **BÄST!**