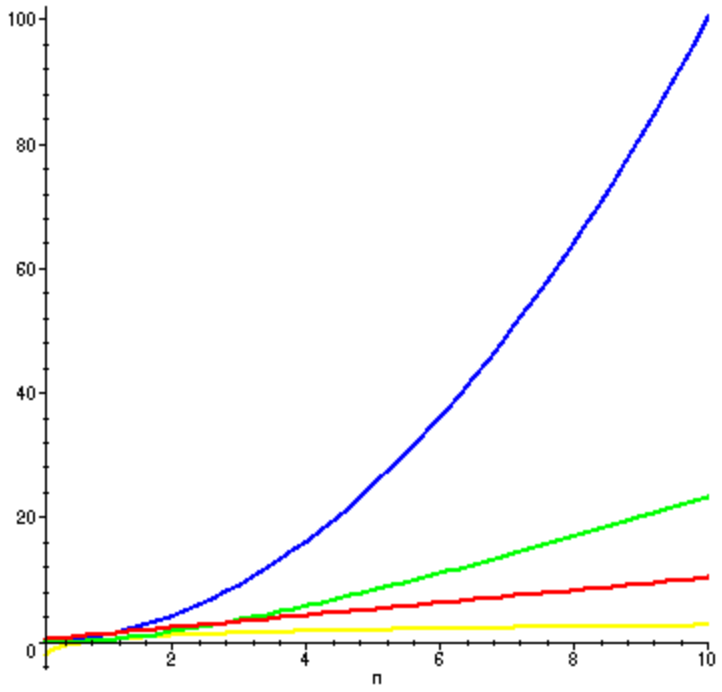
Föreläsning 2: Komplexitet, sökning

- Komplexitetsanalys
- Linjärsökning
- Binärsökning
- Enkel sortering

Komplexitetsanalys
Det är intressant att se *tidsåtgången* för en algoritm. Denna anges ofta som funktion av indatas storlek, som av tradition kallas `n`. Exempel: För sortering är `n` antalet tal som ska sorteras. Hur växer tidsåtgången `T(n)` för växande `n`?



Vi analyserar oftast *värsta fallet* (det är i regel enklare att räkna på) men man kan också räkna på *medelfallet*.

Istället för att ange den exakta tidsfunktionen `T(n)` nöjer vi oss med att ange ordoklassen `O(n)`.

- Anger en övre gräns för `T(n)` då `n` är stort.
- Strunta i multiplikation med konstant.
- Vilken logaritm menas med `log`? Spelar ingen roll!
- Ta bara med den term som växer snabbast.

Exempel: $T(n)=10n^2 + 100n + \log_{10}n + 1000$ säger vi är $O(n^2)$.

- För små indata är analys onödig - använd den enklaste algoritmen!
- Inläsning från fil tar längre tid än åtkomst av redan inlästa värden.
- Minnesåtgång kan också vara intressant.

Sökning

Förutsättningar:

- Vi söker efter något element i en array (vektor) med n element.
- Det element vi söker efter karakteriseras av en söknyckel (eng *key*) men kan också ha andra data.

Frågor:

- Vad ska hända om det sökta elementet inte finns med?
- Kan det finnas flera element med samma nyckel? Vill man i så fall hitta alla?

Linjärsökning (Sequential search)

Algoritm

- Sök igenom elementen i tur och ordning.
- Bryt när det sökta elementet påträffas eller när det uppenbaras att det sökta elementet inte finns med.

Linjärsökning är `O(n)` (i värsta fallet måste vi titta på alla n elementen).

Här följer en metod för linjärsökning i en heltalsarray.

```
def search(vek, key):
    i = 0
    for elem in vek:
        if (elem == key):
            return i
        i += 1
    return -1
```

Metoden ovan kräver att objekten klarar operator= ett alternativ är att skicka med en compare-metod.

```
def search(vek, key, cmp):
    i = 0
    for elem in vek:
        if (cmp(elem, key) == 0):
            return i
        i += 1
    return –1
```

Binärsökning

Om vektorn är sorterad är den snabbaste sökningsalgoritmen binärsökning.
Algoritm

- Beräkna intervallets mittpunkt.
- Avgör om det sökta elementet finns i första eller andra halvan och fortsätt söka där.

Binärsökning är `O(log n)` (antal element att söka bland halveras i varje varv).

Här följer kod för binärsökning i en array med Comparable-element.

```
def binsearch(vek, key):
    low = 0
    mid = 0
    high = len(vek) - 1
    while (low <= high) :
        mid = (low + high) / 2
        if (key < vek[mid]):
            high = mid - 1
        elif (key > vek[mid]):
            low = mid +1
        else:
            return mid
    return -1
```

11	13	13	20	22	25	28	30	31	32	32	48	62
11	13	13	20	22	25							
			20	22	25							
				22								

Interpolationssökning är en variant av binärsökning, där man, istället för att välja mittpunkten som nästa sökpunkt, beräknar en bättre gissning.

```
    gissning = low + (high-low) * (key-a[low]) / (a[high]-a[low])
```

Om man t.ex. vill slå upp Andersson i en telefonkatalog så börjar man kanske inte mitt i (som man skulle gjort med binärsökning) utan bara en liten bit in i katalogen.

Interpolationssökning fungerar dåligt om söknycklarna är ojämnt fördelade över intervallet.

Antalet jämförelser i medeltal är `O(loglog(n))`, men i värsta fallet blir sökningen linjär, d.v.s. `O(n)`.

Simpel och ganska korkad sortering

Vi tänker oss att vi ska sortera en vektor med tal. Vi går igenom vektorn och jämför intilliggande element och byter plats på dessa om det behövs.

```
def bubblesort(array, cmp=lambda x, y: x > y):
  "The simplest working sort routine, for testing purposes.”

  indices = range(len(array))
  indices.reverse()
  for j in indices:
      for i in range(j):
          if cmp(array[i], array[i+1]):
              (array[i], array[i+1]) = (array[i+1], array[i])
```

Algoritmens komplexitet är **$O(n^2)$**.

Vi kommer senare i kursen gå igenom smartare sorteringsalgoritmer där komplexiteten är **O(n log(n))**. Skillnaden är väldigt stor. För en miljon poster blir det 10^{12} jämfört med $10^{7.}$

Komplexa datorproblem önskas

Ibland eftersträvar man komplexitet t.ex. vill man göra det svårt för datorer att automatregistrera epostadresser som hotmail. Då efterfrågas tecknen i bilden nedan.