Algoritm analys

- ***Effektivitet.*** (Tidskomplexiteten)
  I termer av problemets storlek
- ***Minneskrav***. (Rumskomplexiteten)
  I termer av problemets storlek
- ***Svårighetsgrad.***
  Ju mer komplicerad desto svårare att översätta till program och underhålla.

(Analys av tidskomplexitet tilldrar sig mest intresse. Därför används ibland den förkortade termen **"Komplexitet"** när man egentligen avser tidskomplexitet)

Algoritm analys

- Tiden det tar att lösa ett problem är oftast en strängt växande funktion av problemets storlek.

Exempel 1:

Det tar längre tid att sortera 100 000 tal än att sortera 100 tal.

Exempel 2:

- Vissa enkla problem kan dock lösas på konstant tid oavsett storleken .*Om vi t ex har en array med **n** tal kan vi alltid ta reda på det i:e talet i arrayen på konstant tid oavsett hur stort **n** är.*

Fråga : Att summera talen i en array, kostar det mer tid jo större arrayen är ?

Exekveringstid, yttre faktorer?

Problemets storlek (n)
Jo mer data algoritmen behandlar, jo mer tid den kräver

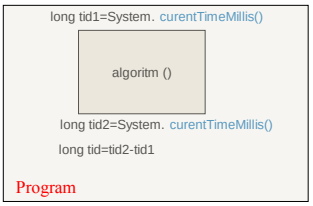
Dator där programmet exekveras
De grundläggande operationerna som =, +, register allokering, tar olika mycket tid på olika datortyper

Programmeringsspråk
Om vi implementerar samma algoritm i annat språk så kommer vi efter kompilering inte att ha exakt samma maskinkod.

Kompilator
Även om vi använder samma språk så kan två olika kompilatorer översätta programmet på något olika sätt, bra kompilatorer *optimerar kod…*

Hur kan man ta reda på exekverings tiden?

1. Genom experiment : att exekvera programmet med olika input och räkna tidsskillnaden



n	tid
1000	0.008
10000	0.01
100000	0.05

Nackdelar med experimental analys

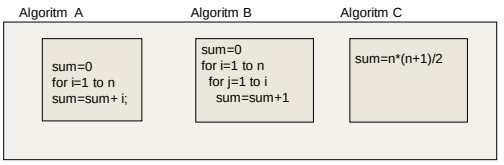
- Algoritmen måste implementeras, kompileras och exekveras på samma dator
- Experiment kan göras på en begränsad antal input, och kan inte vara relevant för andra inputstorlekar
- Om du ska jämföra två algoritmer måste dessa testas med samma hårdvaru och mjukvaru förutsättningar
- Den exakta ex.tiden kräver mycket arbete och är inte relevant i valet av algoritmen

I stället:

Använd en high-level beskrivning av algoritmen och uppskatta ex.tid med hjälp av ”analys av tidskomplexitet”

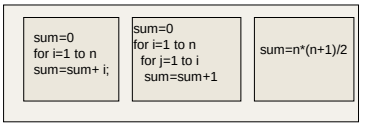
Algoritm analys

Gör en algoritm som beräknar summan av 1+2+3+ .... + n, alla positiva tal.



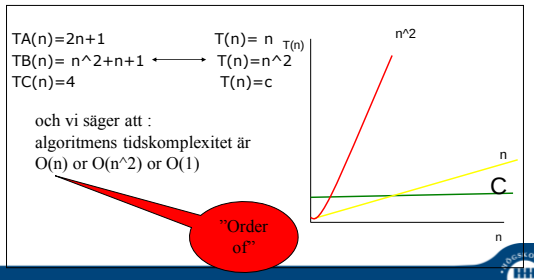
Vilken algoritm har den bästa ex.tiden?

Algoritm analys



tilldelning	**n+1**	**1+n(n+1)/2**	**1**
addition	**n**	**n(n+1)/2**	**1**
multiplikation			**1**
div			**1**
Total operationer	**2n+1**	**n^2+n+1**	**4**

Tidskomplexitet/ Tillväxthastigheten T(n)



Tidskomplexitet/ Tillväxthastigheten T(n)

Fördelar med uppskattning av tidskomplexiteten

1. Vi kan lätt se hur exekveringstiden växer som en funktion av problemets storlek. T ex att den växer linjärt, kvadratiskt, ...
2. Ger en uppfattning om hur stora probleminstanser man kan klara av på rimlig tid
3. Man kan *jämföra* olika algoritmer för samma problem med varandra.
4. Vi har ett mått som är oberoende av datortyp, språk och kompilator.Ett sådant mått kallas *algoritmens tidskomplexitet*.

I praktiken

```
public void myAlgorithm() {
        int s = 0;
        for (int i = 0; i < N; i++) {
                s += i;
        }
        for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                        s *= (j - i);
                }
        }
        System.out.println("s=" + s);
}
```

Linear

Quadratic

Konstant

O(n + n^2+1)
= O(n^2)

Generellt:

1. Ex.tiden för en loop är högst exekverings tiderna för satserna i loopen * antal iterationer
2. Ex.tiden för en grundoperation är konstant O(1)
3. Ex.tiden för en följd av satser är exekveringstiden för den dominanta satsen
4. För n>10 tillväxthastigheten för algoritmer växer enligt

O(1) < O(logn) < O(N) < O(nlogn) < O(n^2) < O(n^3) < O(2^n)

Tidskomplexiten?

```
public static int search ( int [] a, int x)
      {

        int NOT_FOUND=-1;

        for (int i=0; i<a.length;i++)
        {
          if(a[i]==x)
            return i;
        }

           return NOT_FOUND;

      }
```


Binärt sökning, förutser att data är sorterad

```
public static int binarySearch( int [ ] a, int x ){

        int NOT_FOUND=-1;
        int low = 0;
        int high = a.length - 1;
        int mid;

        while( low <= high )
        {
           mid = ( low + high ) / 2;

           if ( x> a[ mid ] )
              low = mid + 1; // leta i högra halvan
           else if(x< a[ mid ] )
              high = mid - 1; // leta i vänstra halvan
           else
              return mid;
        }

        return NOT_FOUND;     // NOT_FOUND = -1
    }
```

För utveckling av verksamhet, produkter och livskvalitet.