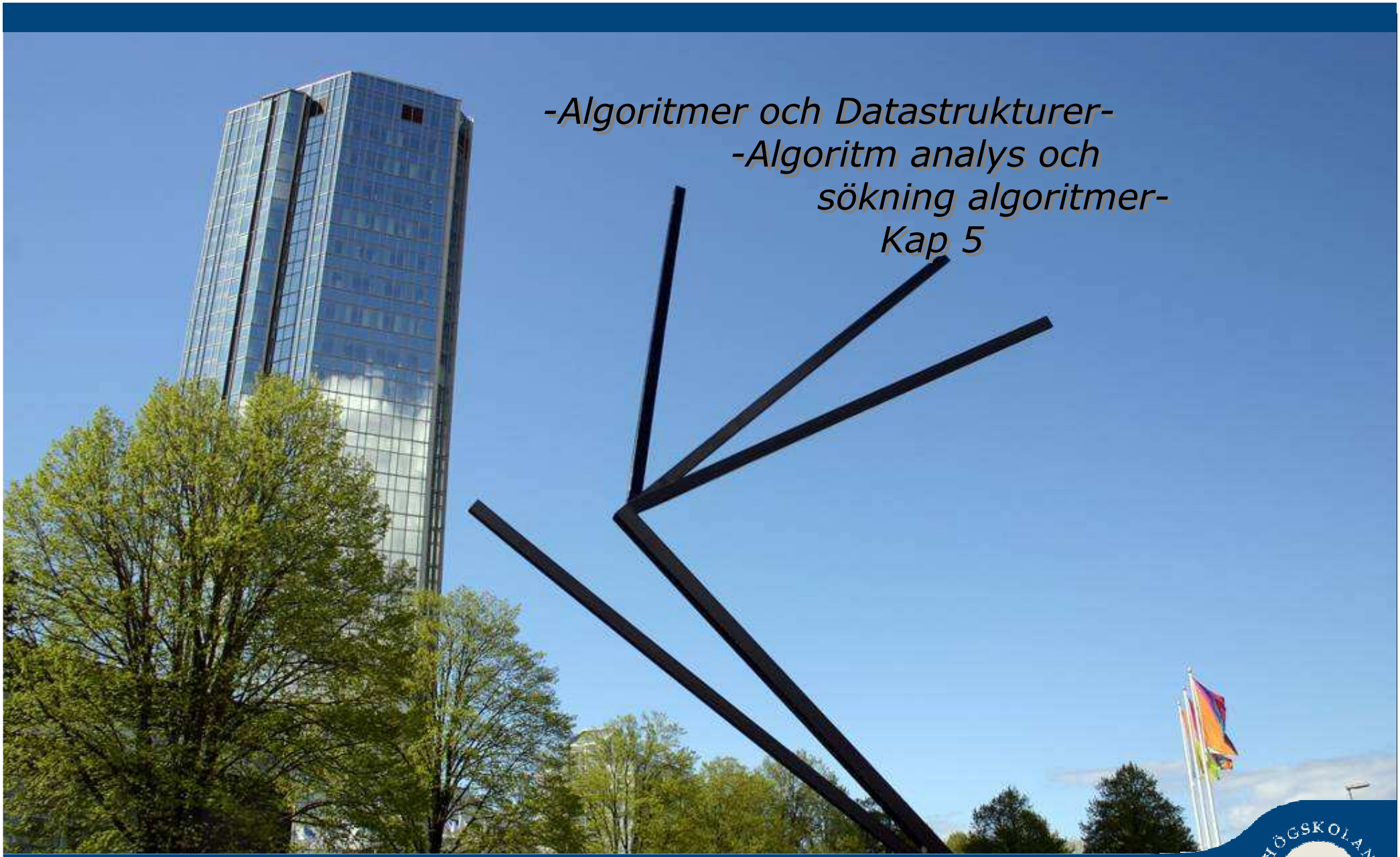


-Algoritmer och Datastrukturer-
-Algoritm analys och sökning algoritmer-
Kap 5

För utveckling av verksamhet, produkter och livskvalitet.

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.

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

- 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.

Däremot kostar det mer och mer att summera talen i arrayen ju större **n** är.

Exekveringstid, yttre faktorer?

Problemets storlek (n)

Jo mer data algoritmen behandlar, jo mer mer tid krävs

Dator där programmet exekveras

De grundläggande operationerna som =, +, ... 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.

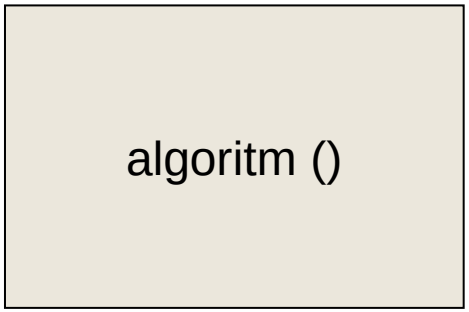
Kompilator

Även om vi använder samma språk så kan två olika kompilatorer översätta programmet på något olika sätt.

Experimental algoritm analys

- genom att exekvera programmmet med olika input och räkna tidenskillnaden

long tid1=System.curentTimeMillis()



long tid2=System.curentTimeMillis()

long tid=tid2-tid1

n	tid
1000	**0.008**
10000	**0.01**
100000	**0.05**

Nackdelar med experimental anlys

- 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 tidskoplexitet”

Algoritm analys

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

Algoritm A

```
sum=0
for i=1 to n
sum=sum+ i;
```

Algoritm B

```
sum=0
for i=1 to n
  for j=1 to i
    sum=sum+1
```

Algoritm C

```
sum=n*(n+1)/2
```

Vilken algoritm har den bästa ex.tiden?

Algoritm analys

```
sum=0
for i=1 to n
sum=sum+ i;
```

```
sum=0
for i=1 to n
  for j=1 to i
   sum=sum+1
```

```
sum=n*(n+1)/2
```

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 T(n)

TA(n)=2n+1 T(n)= n eller O(n)
TB(n)= n^2+n+1 T(n)=n^2 eller O(n^2)
TC(n)=4 T(n)=c eller O(1)

- Hur exekveringstiden växer som en funktion av problemets Storlek. T ex att den växer linjärt, kvadratiskt, ...
- Ger en uppfattning om hur stora probleminstanser man kan klara av på rimlig tid
- Man kan *jämföra* olika algoritmer för samma problem med varandra.
- Vi har ett mått som är oberoende av datortyp, språk och kompilator.
- Ett sådant mått kallas *algoritmens tidskomplexitet* och brukar betecknas med T(n).

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);
}
```

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

```
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
    }
```

ListIterator

ListIterator<E> är ett interface som ärver Iterator<E>
och där man lagt till metoder för att röra sig även bakåt i listor
samt för att sätta in och ta bort element:

```
public interface ListIterator<E> extends Iterator<E> {

boolean hasPrevious();
E previous();
void add(E x);
void remove();
...

}
```

List-klasser i java

- Det finns två konkreta generiska klasser i Javas API för listhantering. Båda implementerar alltså interfacet List

ArrayList som implementerats med vektor

LinkedList, som använder länkad struktur i implementationen

Man bör bara använda ArrayList om alla insättningar görs sist i listan. Insättningar i andra positioner orsakar flyttningar av element.

De indexerade operationerna **get(int idx)** och **set(int idx, E element)** är däremot effektivare i denna implementation än i LinkedList

Paketet weiss.nonstandard i kursboken

- Paketet weiss.nonstandard innehåller interface och klasser för samlingar, som *inte* direkt motsvarar klasser i java.util.

En del av klasserna utgör alternativa sätt att utforma vissa samlingar.

T ex finns det en **LinkedList** och en tillhörande iterator-klass som har ett annat gränssnitt än sin motsvarighet i java.util resp. weiss.util.
Det finns också klasser för hantering av prioritetsköer (**BinaryHeap** och PairingHeap) som inte har samma gränssnitt som javas prioritetsköklass.