Algoritmer och Datastrukturer

Föreläsning 1

Tidskomplexitet

Mål för dagen

- Veta vad en algoritm är
- Förstå vad tidskomplexitet är för någonting
- Förstå varför den används för att bedöma algoritmisk effektivitet i stället för körtid
- Veta vad Big O-notation är för något
- Kunna beräkna tidskomplexitet för några olika algoritmer

Datastrukturer? Algoritmer??

- Datastrukturer är någonting vi använder för att lagra data medan algoritmer är kodblock som opererar på data
- Ett objekt är en datastruktur medan en metod (inte alla metoder!) är en algoritm
- Datastrukturer och algoritmer är intimt förknippade eftersom hur vi lagrar och representerar data kommer att påverka hur vi kan
- Det är omöjligt att optimera algoritmer utan att förstå datastrukturer: en dålig datastruktur kan ha en mer förödande effekt på ett program än dåligt skriven kod

“Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”

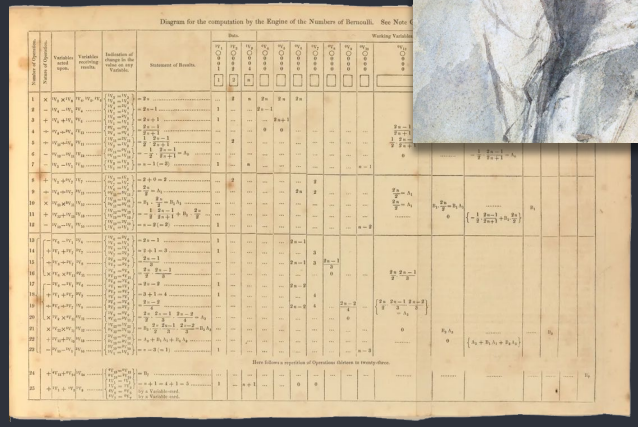
Linus Thorvalds

Vad är en algoritm?

- I grund och botten är algoritmer kod som **försöker lösa ett specifikt problem**
- En algoritm beskriver en högre nivå av abstraktion än rena maskin-instruktioner av typen “lyft en spak”, “sänk en nål”, osv
- En algoritm **beräknar någonting** på en ändlig tid: de körs inte för evigt
- Lite mer avancerade algoritmer **kan producera nya resultat** beroende på sin indata: de kan ha olika beteende beroende på datamängden, osv
- Att säga att **“en algoritm är som ett recept”** är därför egt. en dålig analogi: ett recept förändras ju inte oavsett hur mycket ingredienser vi slänger i
- När vi pratar om algoritmer på den här kursen pratar vi om beräknings-algoritmer: det vill säga sådana som tar emot data och gör någonting med den.

Historiens första algoritm

- Ada Lovelace: inte historiens första programmerare, men författare av historiens första datoralgoritm (mkt viktigare bidrag!)
- Dotter till Lord Byron
- Arbetade med Charles Babbage som byggde Analytical Engine, en tidig mekanisk datorprototyp som aldrig blev färdig
- Skrev en algoritm för att beräkna Bernoullinummer redan 1842
- Hon insåg också att man skulle kunna använda algoritmer för att inkoda (encode på engelska) saker i program, som t.ex. musik, och manipulera dem med en dator
- Dog 36 år gammal, innan hon hann ha mer inflytande på beräkningsteori



Algoritmer: två snabba exempel

Tekniskt sett en algoritm, men väldigt ointressant:

```
public void add(int a, int b)
{
   return a + b;
}
```

* Tydligt definierad
* Består av ändligt antal steg
* Tydlig input
* Körs alltid färdigt
* Alltid samma mängd indata

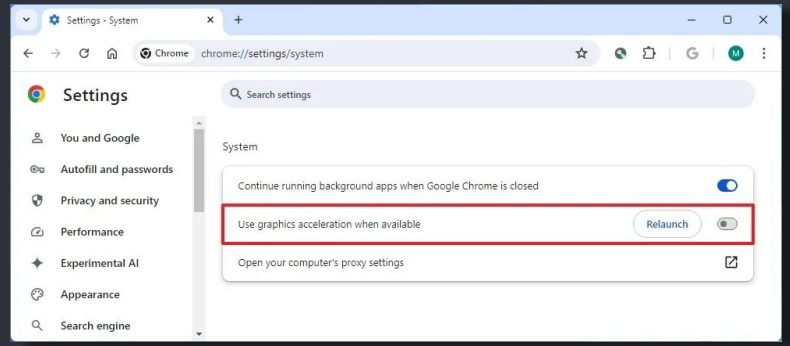
En mer intressant algoritm:

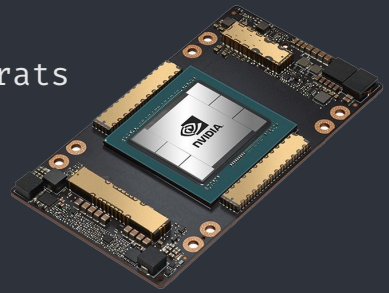
```
public int summarize(int[] n)
{
   int sum = 0;

   for (int value : n)
   {
      sum += value;
   }

   return sum;
}
```

* Tydligt definierad
* Består av ändligt antal steg
* Tydlig input
* Körs alltid färdigt
* Växer med indatan

Algoritmer är mjukvara (eller?)

- Hårdvaruutvecklare **bygger systemarkitektur**, programmerare **bygger program och algoritmer** brukar man säga. Stämmer dock inte alltid:
- **Core Rope Memory:** Under Apollouppdragen sydde man in program i minnet på månlandaren och skapade inbyggda algoritmer för att beräkna landningskoordinater
- **GPU:er** implementerar hårdvarualgoritmer för bland annat parallellprocessering, raytracing, videoencoding och liknande
- **AES-kryptering:** Mycket av den kryptering vi använder, som HTTPS (webbläsare), BitLocker (diskkryptering), VPN (nätverk) och-så-vidare utförs av hårdvarualgoritmer som är inbyggda i datorn - så kallad **hårdvaruaccelerering**
- Hårdvarualgoritmer är **permanenta** när de väl har implementerats och går inte att ändra på efter tillverkning
- Vi bryr oss bara om mjukvarualgoritmer eftersom vi är mjukisar (mjukvaruutecklare!)





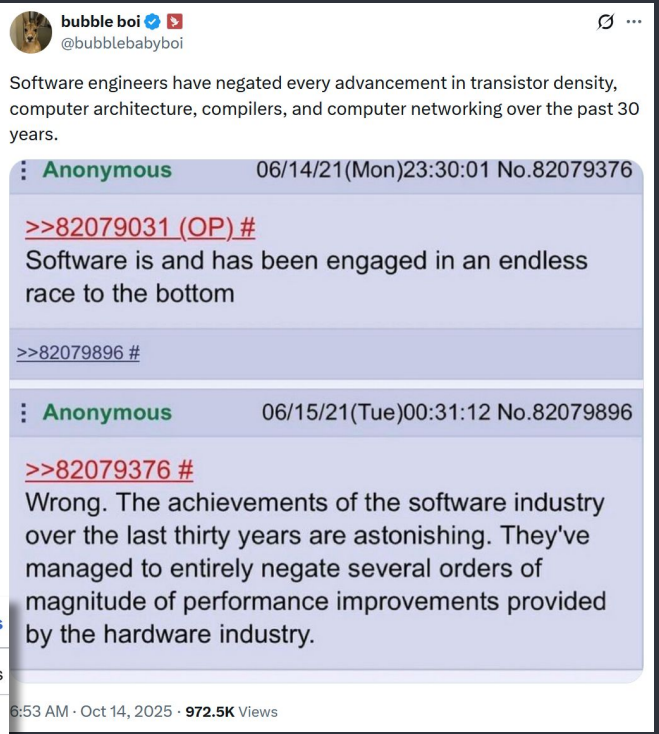
Varför lär vi oss algoritmer och datastrukturer i Java?

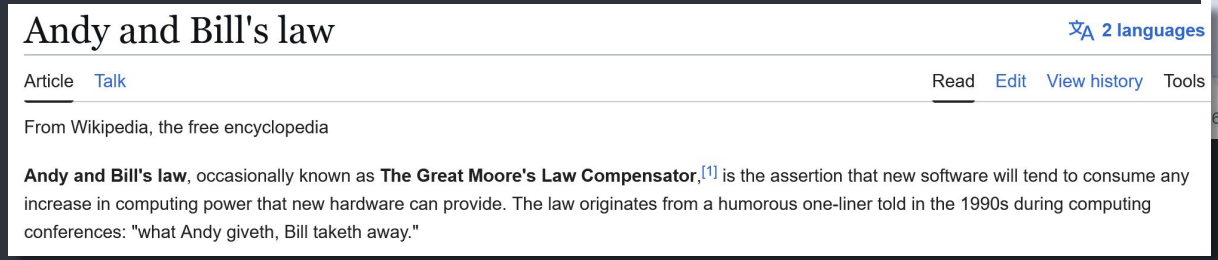
- Om man vill blir bra på att programmera behöver man **en djupare förståelse** för vad som händer under alla dessa lager av abstraktion
- **Church-Turing-hypotesen** gör gällande att alla datorer i grund och botten är likadana: en funktion som kan beräknas av en dator kan i teorin beräknas av alla datorer så länge de är Turingkompletta
- Det här innebär att **algoritmer är universella:** syntax och implementation kan skilja sig, men alla programmeringsspråk kan utföra samma underliggande beräkningar och **logiken för algoritmen är densamma**
- Även datastrukturer är universella: Arrayer, listor, osv finns i de flesta språk. Det du kan göra i ett språk kan du i princip göra i ett annat
- Att förstå hur algoritmer och datastrukturer fungerar är därför att förstå hur alla datorer fungerar

Programmerare: små sabotörer!

- Datorer i dag är flera miljoner gånger mer kraftfulla än för 30 år sedan men vi skriver ofta program som konsumerar enormt mycket mer resurser

- Att förstå hur vi kan optimera kod och data är livsviktigt om vi vill undvika att sabotera de framsteg som görs rent hårdvarumässigt





Hur vi bedömer algoritmisk effektivitet

Hur bedömer man egentligen hur effektiv en algoritm är? Någonstans handlar det ju om att försöka uppskatta tiden, men hur?

* Vi skulle kunna mäta hur fort den går att köra, men det är ingen bra metod: en snabb dator kommer att exekvera koden fortare än en långsam dator
* Tiden kommer också att skilja sig mellan två snabba datorer, eller till med samma dator som kör den vid två olika tillfällen, eftersom det finns så många andra program som körs samtidigt i bakgrunden och påverkar
* Kompilering, språk, hårdvara, etc påverkar körtiden, och därför kan vi inte använda körtid som ett mått på hur effektiv en algoritm är

I stället använder vi något vi kallar för **tidskomplexitet**

Tidskomplexitet

- Tidskomplexitet beskriver hur exekveringstiden för att köra en algoritm ökar baserat på storleken på dess indata
- Det vi vill veta är: när vi stoppar in större och större datamängder, vad händer med tidsutvecklingen?
- Dvs inte “Exakt hur många sekunder/minuter/timmar kommer algoritmen ta att köra?”, utan **“Hur snabb är tillväxthastigheten?”**
- Ett annat ord för tillväxt som vi kommer att använda från och med nu är **komplexitet**. Och vi kommer inte bry oss om hur en algoritm presterar i det allra bästa fallet, under optimala förhållanden, eftersom sådana mätningar helt enkelt inte är värdefulla
- Det vi är intresserade av är: **vad är det absolut värsta utfallet? Vad är gränsen för hur dåligt en viss algoritm kan prestera givet en viss indata?**
- Men hur mäter vi komplexitet då?? Hur definierar vi den?

Ordo

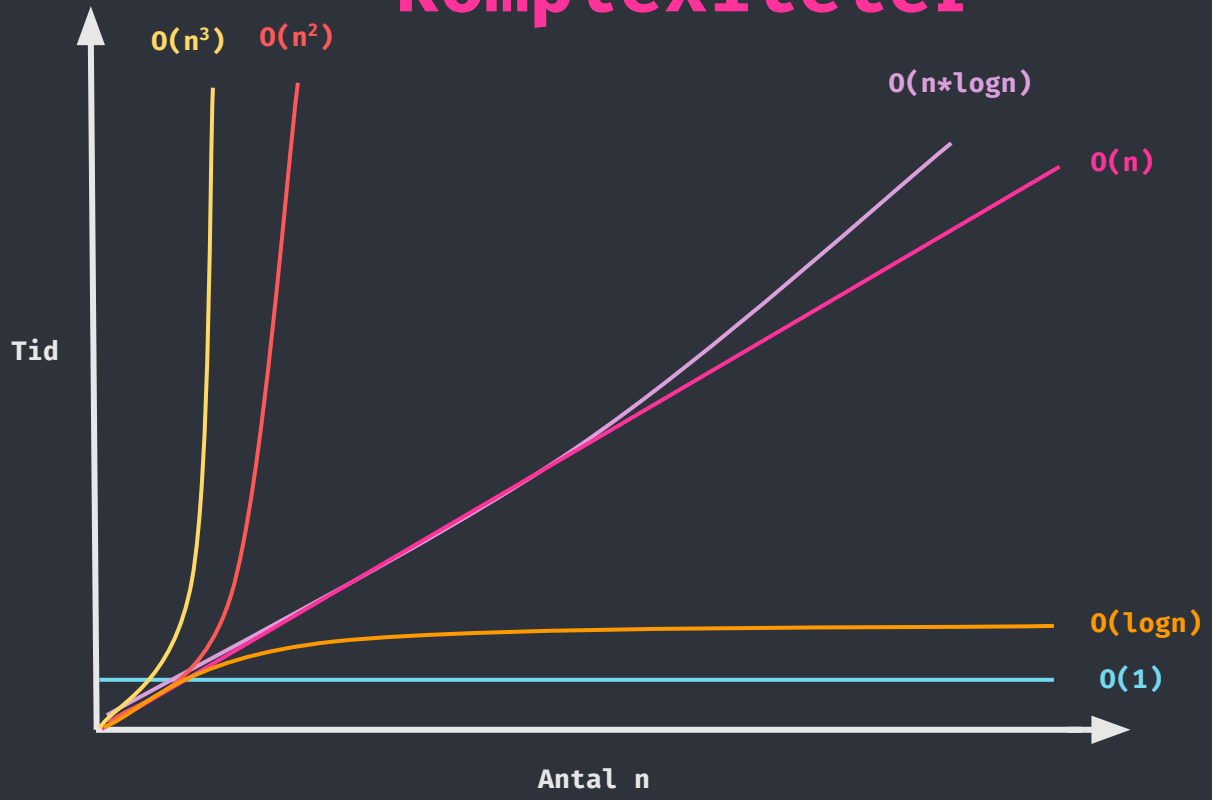
- Tidskomplexitet definieras i “Big O”-notation
- Vi använder den här notationen, även kallad **“ordo”**, för att klassificera algoritmer utifrån hur **exekveringstiden ökar i förhållande till input**
- Ordo kommer från latinets *ōrdō* (men brukar skrivas utan makronerna) som betyder ordning. Ordningen som åsyftas är alltså inte sorteringsordning eller liknande, utan **storleksordning:** *hur stort är någonting*?

Storleks-ordningar

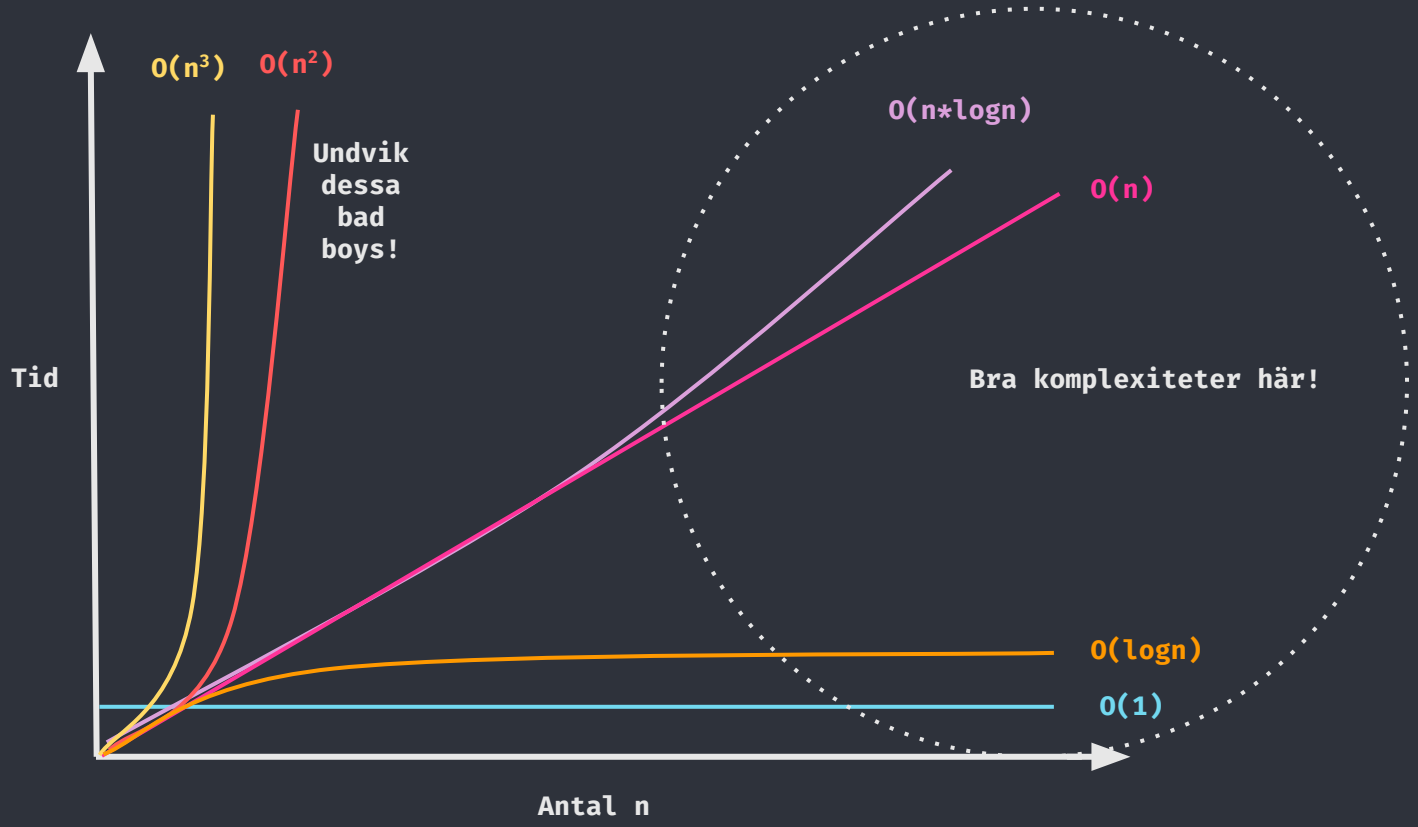
Några vanliga O-notationer

Om tiden:	**... är det:**	**... som betecknas:**
~ **Dubblas** när datamängden dubbleras	Linjär tidskomplexitet	$O(n)$
~ **Fyrdubblas** när datamängden dubbleras	Kvadratisk tidskomplexitet	$O(n^2)$
~ **Åttadubblas** när datamängden dubbleras	Kubisk tidskomplexitet	$O(n^3)$

Tillväxthastigheten för olika komplexiteter



Bra och dåliga komplexiteter



Hur ordo beräknas

- Ordo är språk- och hårdvaruoberoende: om en algoritm har tidskomplexiteten O(n) så har den det på både Mac och Windows, i både C++ och Java

- När vi försöker analysera ordo är vi **intresserade av hur algoritmen beter sig vid stora mängder input.** En konvention är att kalla denna input för “n”, men den måste inte vara just nummer: den kan även vara objekt eller text eller annat som ska processeras

- Vi genomför två steg:

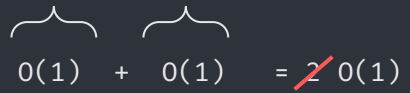
  * Vi identifierar den snabbast växande termen
  * Vi tar bort de andra koefficienterna

- Varför? Eftersom när inputen blir väldigt stor kommer de övriga termerna att ha en marginell betydelse

Paus
10 minuter

Vad blir tidskomplexiteten?

```
public void add(int a, int b)
{
    return a + b;
}
```



Konstant tidskomplexitet, O(1). Oavsett hur stora talen är gör vi samma mängd operationer:

* Lägger ihop två tal, a och b O(1)
* Returnerar summan O(1)

Vad blir tidskomplexiteten?

```
public int summarize(int[] n)
{
   int sum = 0;              // O(1) +

   for (int value : n)       // O(n) *
   {
      sum += value;          // O(1) +
   }

   return sum;               // O(1)
}
```

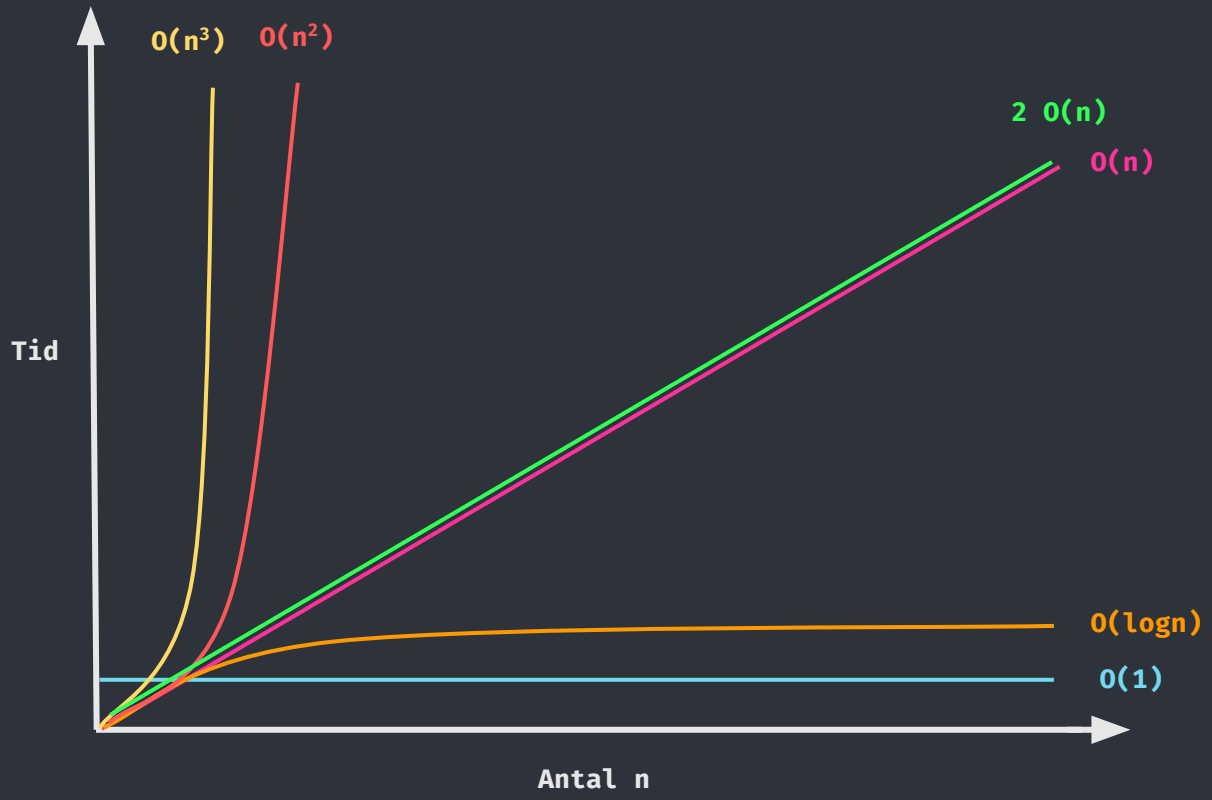
$$O(1) + O(n) * O(1) + O(1) = O(n)$$

Vad blir tidskomplexiteten med två loopar?

```
public int summarize(int[] n)
{
   int sum = 0;                          // O(1) +

   for (int value : n)                   // O(n) *
   {
      sum += value;                      // O(1) +
   }

   for (int value : n)                   // O(n) *
   {
      sum += value;                      // O(1) +
   }

   return sum;                           // O(1)
}
```

$$O(1) + O(n) * O(1) + O(n) * O(1) + O(1) = 2\,O(n)$$

2 O(n) och O(n) beter sig likadant:



Varför tar vi bort koefficienten?

- Big-Oh är definierad på ett sådant sätt att koefficienten ignoreras (en förenkling)
- Vi är bara intresserade av tillväxthastigheten, och den bestäms av den största faktorn
- Koefficienten är även svår att beräkna exakt eftersom t.ex. kompilatorer kan effektivisera kod (de slår ihop flera rader till en i bland genom att blicka framåt)
- **När n blir väldigt stort förlorar den betydelse.** Ta 2 O(n) om n är en miljon som exempel:
  * 2n blir då två miljoner
  * Men **$O(n^2)$, dvs O(n) * O(n), blir en miljon GÅNGER en miljon.** Dvs en biljon: 10^{12}!
  * Två miljoner n bleknar i jämförelse och kommer se ut precis som en miljon n i en utzoomad graf

Vad blir tidskomplexiteten med två nästlade loopar?

– Vid nästlade loopar **multipliceras** tidskomplexiteten för varje loop:

```
public int summarize(int[] n)
{
    int sum = 0;                        // O(1) +

    for (int value : n)                 // O(n) *
    {
        for (int value : n)             // O(n) *
        {
            sum += value;               // O(1) +
        }
    }

    return sum;                         // O(1)
}
```

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

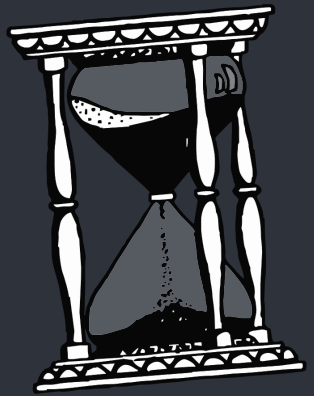
Vad blir tidskomplexiteten med tre nästlade loopar?

```
public int summarize(int[] n)
{
    int sum = 0;                        // O(1) +

    for (int value : n)                 // O(n) *
    {
        for (int value : n)             // O(n) *
        {
            for (int value : n)         // O(n) *
            {
                sum += value;           // O(1) +
            }
        }
    }

    return sum;                         // O(1)
}
```

$$O(1) + O(n) * O(n) * O(n) * O(1) + O(1) = O(n) * O(n) * O(n) = O(n^3)$$

Recap: Hur vi bestämmer tidskomplexitet

- Vi letar efter **den snabbast växande faktorn:** de andra spelar ingen roll
- Multiplikation, subtraktion, jämförelseoperatorer och liknande behandlas som konstanter - **O(1)**
- En loop som körs **n** gånger får komplexiteten **O(n)**
- Nästlade loopar som körs **n** gånger får **$O(n^2)$**
- Tre nästlade loopar som körs **n** gånger får komplexiteten **$O(n^3)$**
- Komplexiteter vi bryr oss om: **O(1), O(logn), O(n), O(n*logn), $O(n^2)$, $O(n^3)$. Resten är för dåliga för att vara värdefulla.**



Uppskatta tidskomplexitet utifrån hur körtiden förändras

- Vi kan som sagt inte säga att en algoritm är effektiv om den tar ett visst antal sekunder att köra. Vi kan däremot enkelt **uppskatta tidskomplexiteten utifrån hur körtiden förändras när inputen förändras**

- Även om en algoritm körs på två helt olika system och i två helt olika språk, och tar olika lång tid att köra, bör själva förändringen (som vi kallar för algoritmens beteende) ändå vara likadan

- Det är ganska vanligt att vi inte känner till exakt hur implementationen för en algoritm ser ut, men genom att skicka in större och större indata kan vi ändå härleda dess tidskomplexitet

Exempel: O(1)

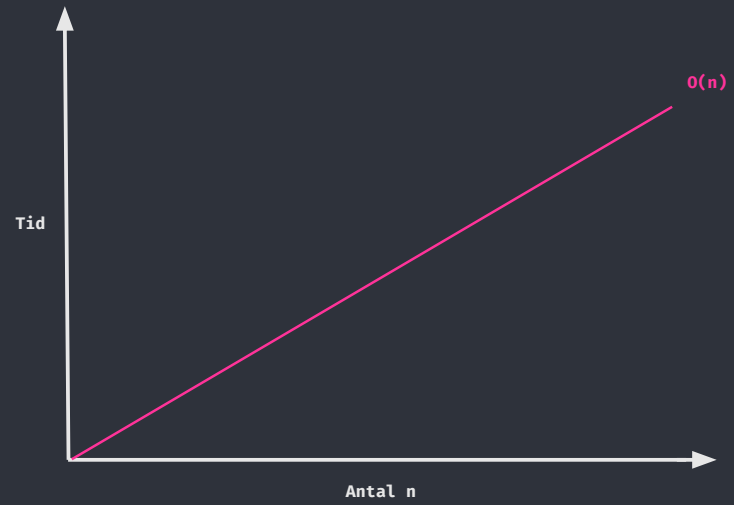
- Tiden påverkas inte av storleken (n). Vi ser ingen ökning av tiden när vi ökar Storleken på inputen.

n	tid
2	4
4	6
8	5
16	7
32	3
205	4

Exempel: O(n)

– Tiden växer linjärt i förhållande till storleken på n. När vi dubblar storleken dubblas även tiden (på ett ungefär!)

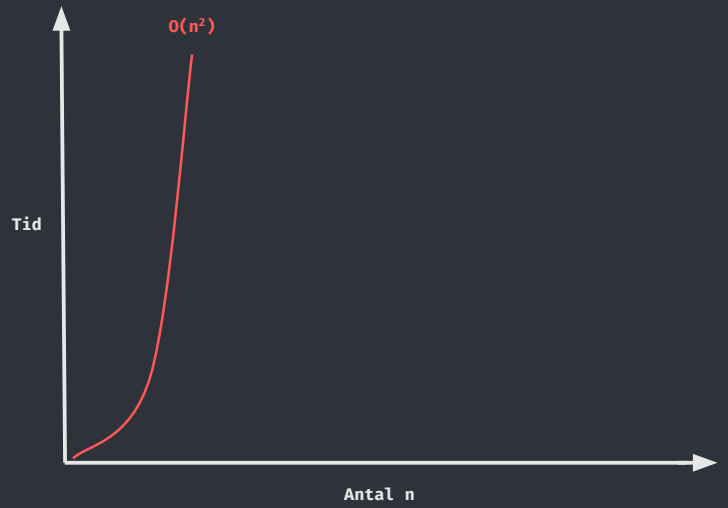
n	tid
2	4
4	9
8	21
16	37
32	83



Exempel: $O(n^2)$

– Tiden växer kvadratiskt i förhållande till storleken på n. När vi dubblar storleken fyrdubblas även tiden. När storleken ökar med *2 ökar tiden med $*2^2$.

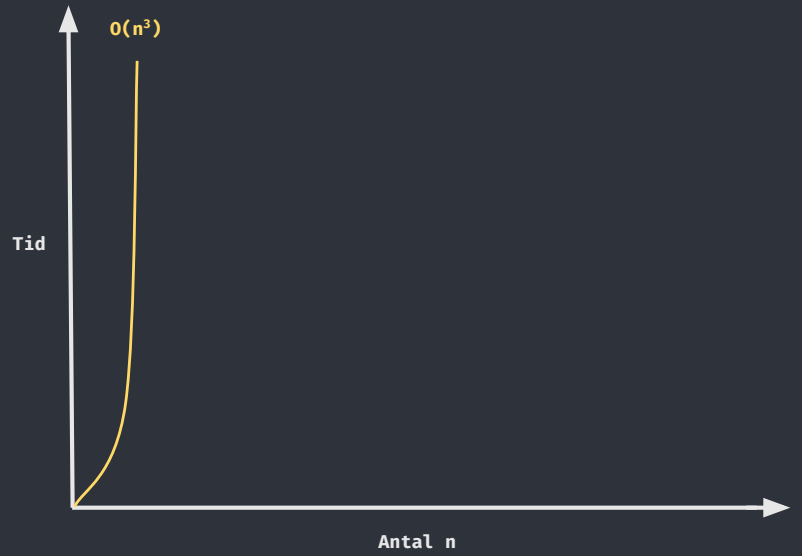
n	tid
2	2
4	9
8	30
16	130
32	508



Exempel: $O(n^3)$

- När vi dubblar mängden n åttadubblas tiden. När storleken (n) ökar med *2 så ökar tiden med $*2^3$.

n	tid
2	2
4	9
8	66
16	521
32	3960



Bästa och sämsta fallet

- Vi bryr oss om var golvet är någonstans, inte taket. Vi vill ha den undre gränsen för hur långsam en algoritm kan gå, för det är en garanti för att det aldrig kommer bli värre

- Samma algoritm kan teoretiskt ha vitt skiljda komplexiteter för bästa och sämsta fall. Om vi söker efter ett värde i en array med hjälp av följande alrogitm är bästa fallet att värdet ligger först, medan värsta fallet är att det ligger sist. Bästa fallet har då O(1) medan värsta fallet har O(n):

```
public int findValue(int[] n, int value)
{
   for (int integer : n)
   {
      if (integer == value) { System.out.println(“Found it!”); }
   }

   return sum;
}
```

Är bästa fallet betydelselöst?

- Det finns sammanhang då även ordo för bästa fallet kan vara viktigt, men de är ofta specifika. Några exempel:

  * Videoprocessering: Codecs som H264 återanvänder pixlar som inte förändras, så en film med många statiska scener behöver mindre processering

  * Kollisionsdetektering i spel (de flesta objekt kolliderar inte med varandra, så vi behöver nästan aldrig ta hänsyn till tillfällen då de teoretiskt skulle göra det när vi skriver vår kod)

  * Flera sorteringsalgoritmer är konstruerade för att vara väldigt snabba om en samling redan är delvis sorterad, men har ett uselt sämsta fall: det är vanligt att program gör en snabb check för att se om en array eller lista till största delen redan är i ordning, och i så fall väljer den sorteringsalgoritm med bäst tidskomplexitet för bästa fallet

I eftermiddag

- **Eget arbete:** en sal är bokad där ni kan sitta och jobba (men ni kan också sitta på systemet om ni hellre vill det)

- Jobba med uppgifterna på kurssidan under modulen för vecka 4

- När ni kikat på de uppgifterna kan ni börja ta er an Laboration 1, som inte kräver några särskilda kunskaper: den är lättare än de flesta av java-labbarna. Den kräver dock att ni övat lite på att beräkna tidskomplexitet

- Jag kommer gå mellan salen och systemet och se om någon behöver hjälp

- Innan torsdag bör ni även repetera föreläsningsmaterialet från i dag, samt läsa kapitel 3, 4 och 7 i kursboken: de är lättare att ta sig an nu i efter-hand när vi pratat lite om koncepten på sal

- **Slutligen en liten uppmaning:** Vi har folk som läser kursen fristående. Passa på att lära känna dem! De läser säkert ämnen som kan bli aktuella för er om ni tänker söka valbara kurser under termin 5. De kommer inte in på Systemet med sina kort, men var goda kurskamrater och bjud in dem om de vill hänga på