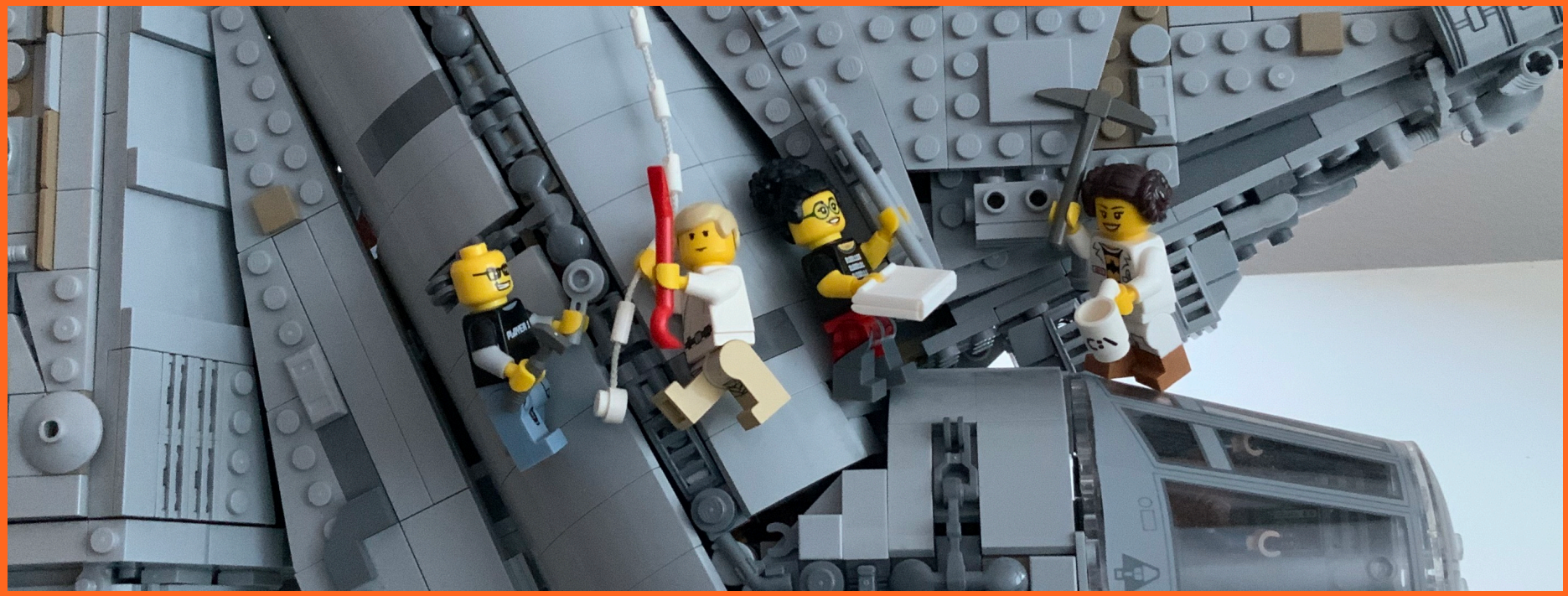
CS-A1120 Ohjelmointi 2



Aalto-yliopisto
Perustieteiden
korkeakoulu

Luento 7
kevät 2026

Sanna Suoranta (suomeksi, T1 klo 14) ja
Lukas Ahrenberg (englanniksi, T1 klo 10)

20.4.2026

https://presemo.aalto.fi/o2fi2026

Tänään O2:ssa

https://presemo.aalto.fi/o2fi2026

Tehokkuus (efficiency)

Aalto-yliopisto
Perustieteiden korkeakoulu

Tämän kierroksen jälkeen

- **Pystyt kertomaan esimerkkejä laskennallisista resursseista (computational resources)**
- **Pystyt mittaamaan ohjelman suoritusaikaa (run-time) Scalassa**
- **Tiedät matemaattiset määritelmät $\mathcal{O}$, Ω ja Θ**
- **Osaat määritellä ohjelmien ajoajalle Big-O:n ja**
  - Analysoida perusohjelmia tässä suhteessa
- **Sinulla on kokemusta järjestämisestä (indeksoinnista, indexing) ja hakemisesta (searching)**
- **Osaat toteuttaa ja käyttää binäärihakua (binary search)**

Tehokkuus (efficiency)?

- **Miksi ohjelmien tehokkuus on tärkeää?**
- **Miten tehokkuus voidaan määritellä?**
- **Kuinka tehokkuutta voi mitata?**

- **Keskustelkaa ja jakakaa:**

https://presemo.aalto.fi/o2fi2026

Tehokkuustavoite

- **Haluamme tehtävään tarvittavien resurssien määrän skaalautuvan hyvin, kun syöteinstanssin koko kasvaa**
  - syöteinstanssi = kaikki ongelman ratkaisemiseen tarvittavan laskennan vaatima syöte ≈ syötedata
  - Esim. Jos kuluu 10 s suorittaa funktio 10-elementtiselle taulukolle, kuinka kauan kuluu kun syöte-elementtejä on 100*n? Onko se 100 s, 1000 s? Miljoona vuotta?
- **Esimerkkejä:**
  - Tenttipaperit: arvostelu, kun opiskelijoiden määrä kasvaa
  - Kuva: prosessointi, kun resoluutio kasvaa
  - Tekoälyhahmot pelissä: pelimekaniikka, kun esim. pelin vaikeustaso kasvaa?
  - Matriisi: laskenta, kun elementtien määrä kasvaa
  - …

Laskenta vaatii resursseja

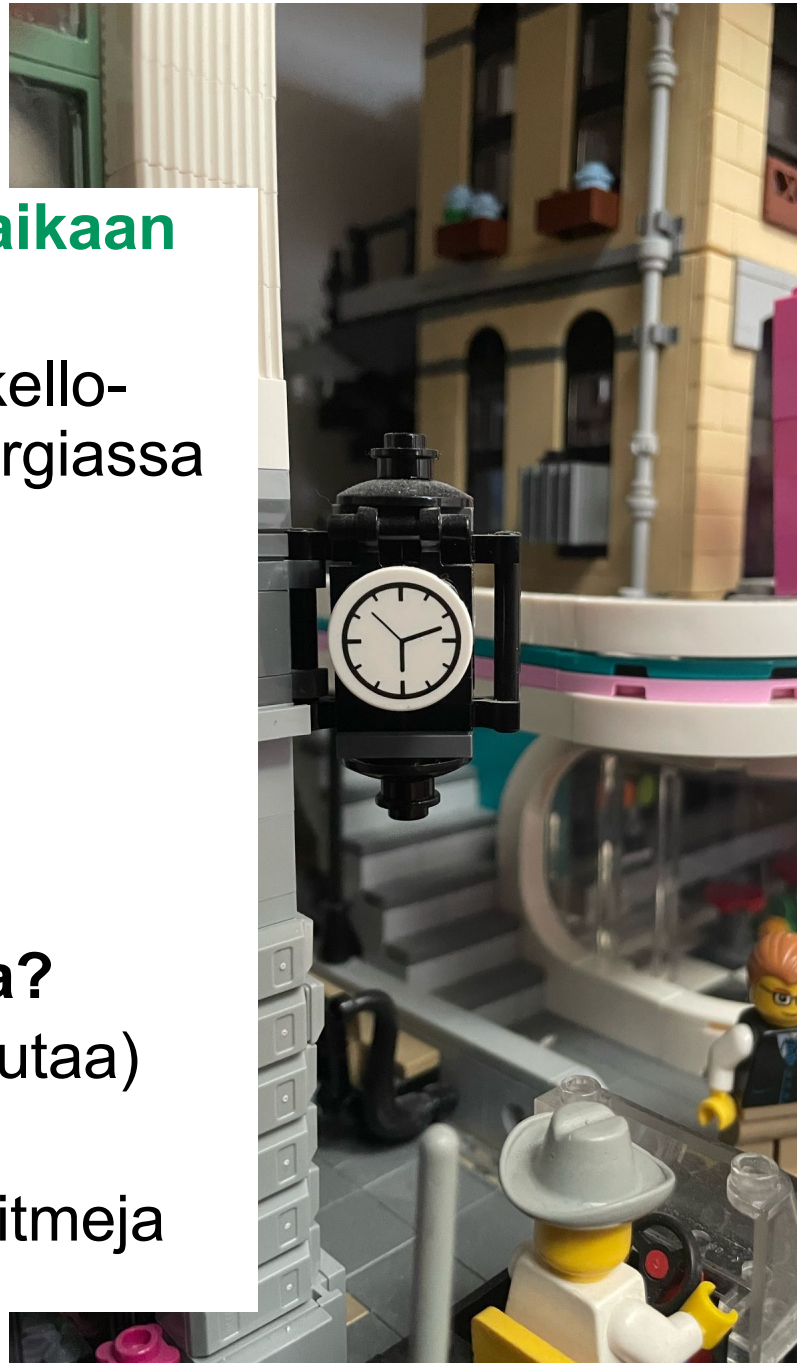
- **Aikaa**
- **Tilaa**
  - Muistia (memory)
  - Tallennustilaa (storage)
- **Energiaa**
- **Kaistaa (bandwidth)**
- **Prosessoreita**

Aika
ja sen mittaaminen

Aalto-yliopisto
Perustieteiden
korkeakoulu

Aika

- **Tällä kurssilla keskitymme (pääosin) ajoaikaan (run time)**
  - Ei vaan odotusaika vaan se, että CPU:n kello-pulssi (clock tick) tyypillisesti maksaa energiassa
- **Kuinka voimme päätellä tietyn ohjelman ajamiseen kuluvan ajan?**
  - Mittaamalla sen
  - Analysoimalla koodia
  - Analysoimalla algoritmi
- **Kuinka voimme parantaa koodin ajoaikaa?**
  - Lisäämällä resursseja (lisää/parempaa rautaa)
  - Optimoimalla koodia
  - Toteuttamalla paremmin suoriutuvia algoritmeja



Ajoajan (run time, running time) mittaus

Erilaisia tapoja ohjelman ajoaikaa:

- **Seinäkelloaika, kulunut aika (wall clock time, elapsed time) mitataan ulkoisella kellolla**
  - Huom. Ohjelma saattaa joutua odottamaan muita resursseja vaativia ohjelmia tietokoneessa
- **CPU-aika: CPU:n ohjelman ajoon käyttämä aika, jaettuna:**
  - **Käyttäjän aika** (user time): ohjelmakoodin suorittamiseen kuluva aika, sisältäen esim. käyttäjän toimintojen odottamisen
  - **Järjestelmäaika** (system time): ohjelman tekemien järjestelmäkutsujen (system call) käyttämä aika (esim. I/O)
- **Jatkossa keskitymme mittaamaan CPU-aikaa**

CPU-ajan mittaaminen Scalassa

- **Perusidea: seuraavat vaiheet suoritetaan useamman kerran tarkkuuden parantamiseksi:**
  1. Tarkista CPU-aika (`getCPUTime`)
  2. Suorita funktio
  3. Tarkista CPU-aika (`getCPUTime`)
  4. Laske erotus
- **Scala itse tarjoaa vain seinäkelloajan (system.nanoTime)**
- **Käytämme CPU-ajan mittaamiseen java.lang.management-kirjastosta löytyvää ThreadMXBean-rajapintaa**
  - Nanosekunnin tarkkuus (sisäinen tarkkuus, *precision*), mutta ei tietoa siitä, mittaako ihan oikeaa asiaa (ulkoinen tarkkuus, *accuracy*)

https://docs.oracle.com/en/java/javase/11/docs/api/java.management/java/lang/management/ThreadMXBean.html

CPU-ajan mittaaminen Scalassa (jatkuu)

- **Määritellään funktiot**
  - `getCPUTime` ajan mittaamiseksi
  - `measureCpuTime` yhden suorituskerran mittaamiseksi
  - `measureCpuTimeRepeated` keskiarvolle tarkkuuden parantamiseksi

```
// Import from Java import
java.lang.management.{
  ManagementFactory,
  ThreadMXBean }
// Set-up magic
val bean: ThreadMXBean = ManagementFactory
  .getThreadMXBean()
// Get time, or 0 if functionality
// not supported
def getCpuTime: Long =
  if bean.isCurrentThreadCpuTimeSupported() ther
    bean.getCurrentThreadCpuTime()
  else
    0L
```

- **Mittausfunktiot seuraavilla kalvoilla**

CPU-ajan mittaaminen – yhden kerran

- **Kertamittaus riittää, jos operaatio vie yli 0,1 s aikaa**

```
/** Define minimum positive time greater than 0.0*/
val minTime = 1e-9 // Needed as Windows gives 0.0 on small durations
/**
  * Runs the argument function f and measures the user+system time spent in it in seconds.
  * Accuracy depends on the system, preferably not used for runs taking less than 0.1 s.
  * Returns a pair consisting of
  * - the return value of the function call and
  * - the time spent in executing the function.
  */
def measureCpuTime[T](f: => T): (T, Double) =
  val start: Long = getCpuTime
  val r = f
  val end: Long = getCpuTime
  val t: Double = minTime max (end - start) / 1e9 // because ns
  (r, t)
```

CPU-ajan mittaaminen – toistuvasti

- **Nopeille operaatioille: toistuva mittaus parantaa tarkkuutta**

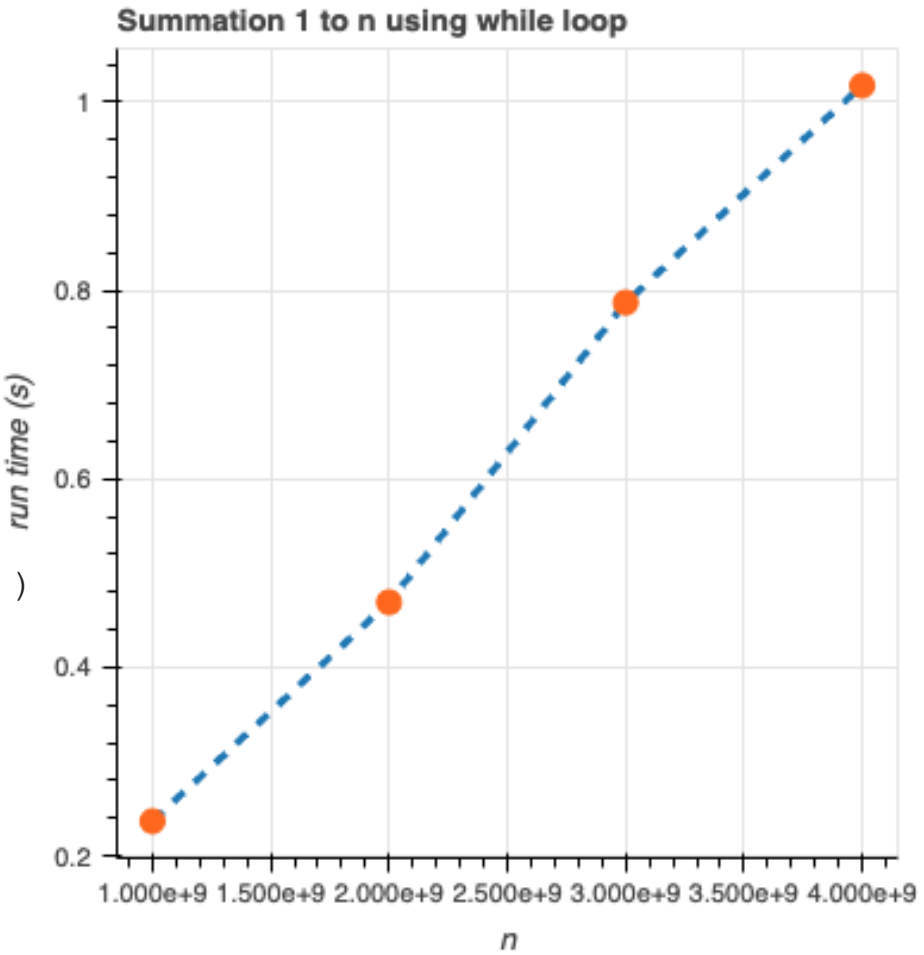
```
/**
 * The same as measureCpuTime but the function f is applied repeatedly
 * until a cumulative threshold time use is reached (currently 0.1 seconds).
 * The time returned is the cumulative time divided by the number of repetitions.
 * Therefore, better accuracy is obtained for very small run-times.
 * The function f should be side-effect free!
 */
def measureCpuTimeRepeated[T](f: => T): (T, Double) =
  val start: Long = getCpuTime
  var end = start
  var runs = 0
  var r: Option[T] = None
  while end - start < 100000000L do
    runs += 1
    r = Some(f)
    end = getCpuTime
  val t = minTime max (end - start) / (runs * 1e9)
  (r.get, t)
```


https://a1120.cs.aalto.fi/2026/notes/round-efficiency--measuring.html#measuring-run-time

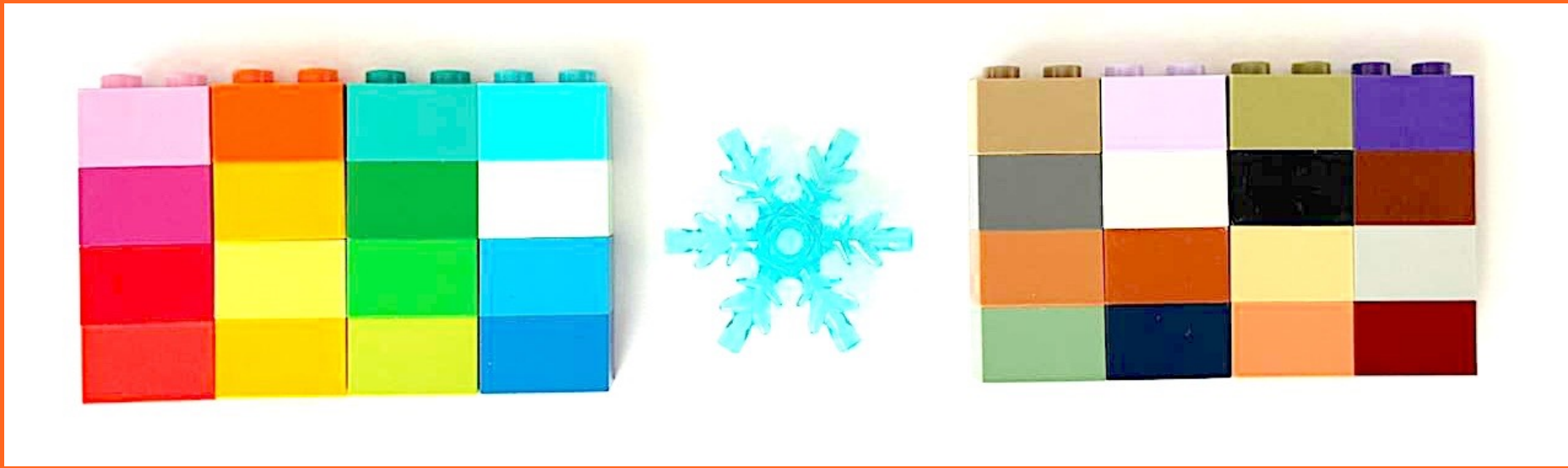
Kauanko kestää n:n luvun yhteenlasku?

```
// Our summation of values from 1 to n
def f(m: Long): Long =
  var i = 1L
  var s = 0L
  while i <= m do
    s = s + i
    i = i + 1
  s
// Time for all n in this sequence
val ns = Seq(1000000000L, 2000000000L,
  3000000000L, 4000000000L)
// Perform measurement for each n in ns
val fData = ns.map(n => measureCpuTime( f(n) ) )

scala> fData
val res: Seq[(Long, Double)] = List(
  (5000000005000000000,0.23691685),
  (2000000001000000000,0.469175766),
  (4500000001500000000,0.787165003),
  (8000000002000000000,1.017464763))
```



Monimutkaisempi esimerkki: matriisioperaatioiden tehokkuus



Aalto-yliopisto
Perustieteiden
korkeakoulu

Monimutkaisempi esimerkki: matriisioperaatiot

- **Käytämme *n* x *n* matriiseja eli neliömatriiseja**
- **Rivien ja sarakkeiden indeksit alkavat nollasta, jotta noudatamme Scala Array:n indeksointia**
  - Matematiikassa yleensä aloitetaan ykkösestä

$$A = \begin{pmatrix} a_{(0,0)} & a_{(0,1)} & \cdots & a_{(0,n-1)} \\ a_{(1,0)} & a_{(1,1)} & \cdots & a_{(1,n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(n-1,0)} & a_{(n-1,1)} & \cdots & a_{(n-1,n-1)} \end{pmatrix}$$

Matriisien summa ja kertolasku

- **Yhteenlasku**

$$C = A + B$$

$$c_{(i,j)} = a_{(i,j)} + b_{(i,j)}$$

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} =$$

$$\begin{pmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{pmatrix} =$$

$$\begin{pmatrix} 6 & 8 \\ 10 & 12 \end{pmatrix}$$

- **Kertolasku**

$$C = AB$$

$$c_{(i,j)} = \sum_{k=0}^{n-1} a_{(i,k)} \times b_{(k,j)}$$

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} =$$

$$\begin{pmatrix} 1\times5+2\times7 & 1\times6+2\times8 \\ 3\times5+4\times7 & 3\times6+4\times8 \end{pmatrix} =$$

$$\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

Matriisien summa ja kertolasku

- **Yhteenlasku**

$$C = A + B$$

$$c_{(i,j)} = a_{(i,j)} + b_{(i,j)}$$

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} =$$

$$\begin{pmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{pmatrix} =$$

$$\begin{pmatrix} 6 & 8 \\ 10 & 12 \end{pmatrix}$$

- **Kertolasku**

$$C = AB$$

$$c_{(i,j)} = \sum_{k=0}^{n-1} a_{(i,k)} \times b_{(k,j)}$$

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} =$$

$$\begin{pmatrix} 1\times5+2\times7 & 1\times6+2\times8 \\ 3\times5+4\times7 & 3\times6+4\times8 \end{pmatrix} =$$

$$\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

- **Montako peruslasku-toimitusta (+ tai *) tarvitaan?**
  - 2x2-matriisille?
  - 3x3-matriisille?

Yksinkertainen matriisi-luokka Scalassa

```
/** Basic square matrix class.*/
class Matrix(val n: Int):
  require(n > 0, "The dimension n must be positive")
  protected[Matrix] val entries = new Array[Double](n * n)
/** With this we can access elements by writing M(i,j) */
def apply(row: Int, column: Int) =
  require(0 <= row && row < n)
  require(0 <= column && column < n)
  entries(row * n + column)
end apply
/** With this we can set elements by writing M(i,j) = v */
def update(row: Int, column: Int, value: Double) : Unit =
  require(0 <= row && row < n)
  require(0 <= column && column < n)
  entries(row * n + column) = value
end update
// ... More methods to come on subsequent slides...
```

Yksinkertainen matriisi-luokka Scalassa

- **Esitetään matriisi Array:nä** `val entries = new Array[Double](n * n)`
- **Eli esim. 3x3-matriisi**

$$\begin{pmatrix} a_{(0,0)} & a_{(0,1)} & a_{(0,2)} \\ a_{(1,0)} & a_{(1,1)} & a_{(1,2)} \\ a_{(2,0)} & a_{(2,1)} & a_{(2,2)} \end{pmatrix}$$

voisi olla Array-tyyppinen muuttuja `entries`

$$[a_{(0,0)}, a_{(0,1)}, a_{(0,2)}, a_{(1,0)}, a_{(1,1)}, a_{(1,2)}, a_{(2,0)}, a_{(2,1)}, a_{(2,2)}]$$

- **Matriisin elementti *(i, j)* on taulukossa kohdassa *i * n + j***
  - Esim. matriisin elementti $a_{(1,2)}$ on `entries`-taulukon indeksissä 1 * 3 + 2 = 5 (i ja j elementin indeksi, n=3, 3x3-matriisille)

Matriisien yhteenlasku Scalassa

$$C = A + B$$

$$c_{(i,j)} = a_{(i,j)} + b_{(i,j)}$$

```
class Matrix //... as before

/** Returns a new matrix that is the sum of this and that */
def +(that: Matrix): Matrix =
  val result = new Matrix(n)
  // This is a double for loop:
  for i <- 0 until n; j <- 0 until n do
    result(i, j) = this(i, j) + that(i, j)
  result
end +
```

Matriisien yhteenlasku Scalassa

$$C = A + B$$

$$c_{(i,j)} = a_{(i,j)} + b_{(i,j)}$$

```
class Matrix //... as before

/** Returns a new matrix that is the sum of this and that */
def +(that: Matrix): Matrix =
  val result = new Matrix(n)
  // This is a double for loop:
  for i <- 0 until n; j <- 0 until n do
    result(i, j) = this(i, j) + that(i, j)
  result
end +
```

- **Lyhennysmerkintä `for i <- 0 until n; j <- 0 until n do` tarkoittaa:**

```
for i <- 0 until n do
  for j <- 0 until n do
```

Matriisien kertolasku Scalassa

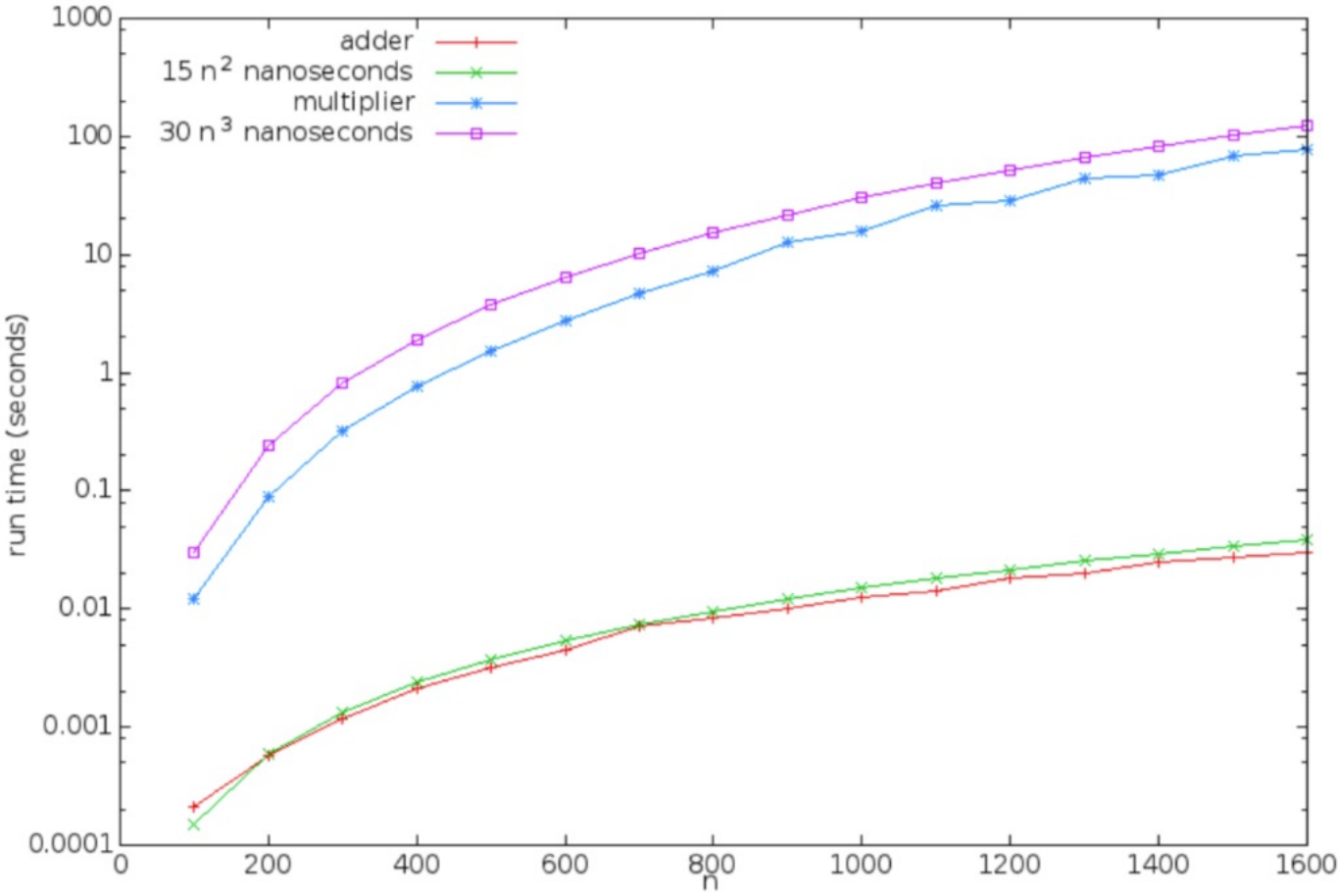
$$C = AB$$

$$c_{(i,j)} = \sum_{k=0}^{n-1} a_{(i,k)} \times b_{(k,j)}$$

```
class Matrix //... as before
/** Returns a new matrix that is the product of this and that */
def *(that: Matrix): Matrix =
  val result = new Matrix(n)
  for i <- 0 until n; i <- 0 until n do
    var v = 0.0
    for k <- 0 until n do
      v += this(i, k) * that(k, j)
    result(i, j) = v
  end for
  result
end *
```

Matriisioperaatioiden ajoaikaprofiilit

- **Käyttäen `measureCpuTime Repeated`, matriisien koko *n* = 100, 200, …, 1600**
- **Huom! Y-akseli on logaritminen**
- **(punavihersokeille**
  - summa on alin ja $15n^2$ toiseksi alin)



Käyrien sovitus

- **Edellisen kalvon ajoajat on tuotettu vuosia sitten koneella, jonka spkesit ovat 3.1 GHz i5-2400 CPU, 8 GB RAM, käyttäen Scala 2.10.3:ea**
  - Vanhentuneita numeerisesti
- **Todelliset ajoajat vaihtelevat tietokonejärjestelmien välillä, mutta**

Käyrän muoto pysyy samana!

- **Millaisia käyriä meillä voi olla?**

https://presemo.aalto.fi/o2fi2026

Big-O eli $\mathcal{O}$ notaatio

Aalto-yliopisto
Perustieteiden
korkeakoulu

Big-O-notaatio (Big-O notation, merkitään $\mathcal{O}$)

- **Todelliset ajoajat ovat paljon monimutkaisempia kuin edellä esitetyt peruskäyrät, esim.**
  - $f(n)=150n^3+2n^2+32$ tai
  - $f(n)=44n\log n+15n$
- **Mutta vakiot johtuvat yleensä järjestelmästä tai ovat toteutuksesta riippuvia**
- **Kun syöte on riittävän suuri, algoritmin yksityiskohdat tapaavat dominoida**
- **Big-O-notaatio abstrahoi pois vakiotermit ja sellaiset termit, joiden kasvua jokin toinen termi dominoi**

Big-O-notaatio (Big-O notation, merkitään $\mathcal{O}$)

- **Määritelmä (Big-O) ”kasvaa korkeintaan yhtä nopeasti kuin”:**

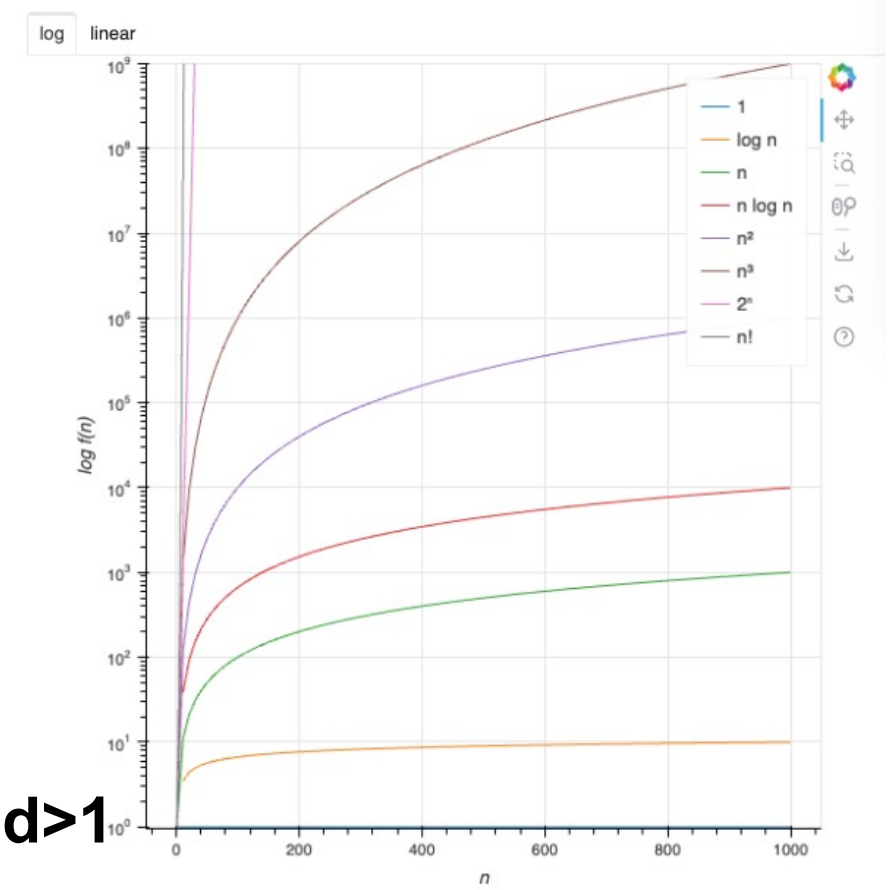
  Kahdelle (positiiviselle, reaaliarvoiselle) funktiolle, *f* ja *g*,
  jotka on määritelty ei-negatiivisille kokonaisluvuille *n*,
  kirjoitamme $f(n) = \mathcal{O}(g(n))$
  jos on olemassa vakio *c*, $n_0 > 0$ siten,
  että $f(n) \leq cg(n)$ kaikilla $n \geq n_0$.

- **Eli *f* kasvaa korkeintaan yhtä nopeasti kuin *g* kun *n* on kyllin suuri (vakiokertoimeen asti)**
  - ***g* on asymptoottinen yläraja *f*:lle**
- **Tarkalleenottaen yhtäsuuruusmerkin käyttö $f(n) = \mathcal{O}(g(n))$ on väärin**
  - **Oikeampi olisi sanoa $f \in \mathcal{O}(g)$, jos ajattelemme $\mathcal{O}(g)$:ta joukkona**

‘Yleisiä’ muotoja käyrille

- **Vakioaikainen (constant, c)**
  - käyttää tietyn vakioajan
- **Logaritminen (log n)**
  - Yleensä oletetaan 2-kantainen log
- **Lineaarinen (n)**
- **Linearitminen (linearithmic, n log n)**
- **Neliöllinen (quadratic, n^2)**
- **Kuutiollinen (cubic, n^3)**
- **Polynominen (polynomial, n^k), k>0**
- **Eksponentiaalinen (exponential, d^n), d>1**
- **Faktoriaalinen (factorial, n!)**

Logaritminen y-akseli, miksi?

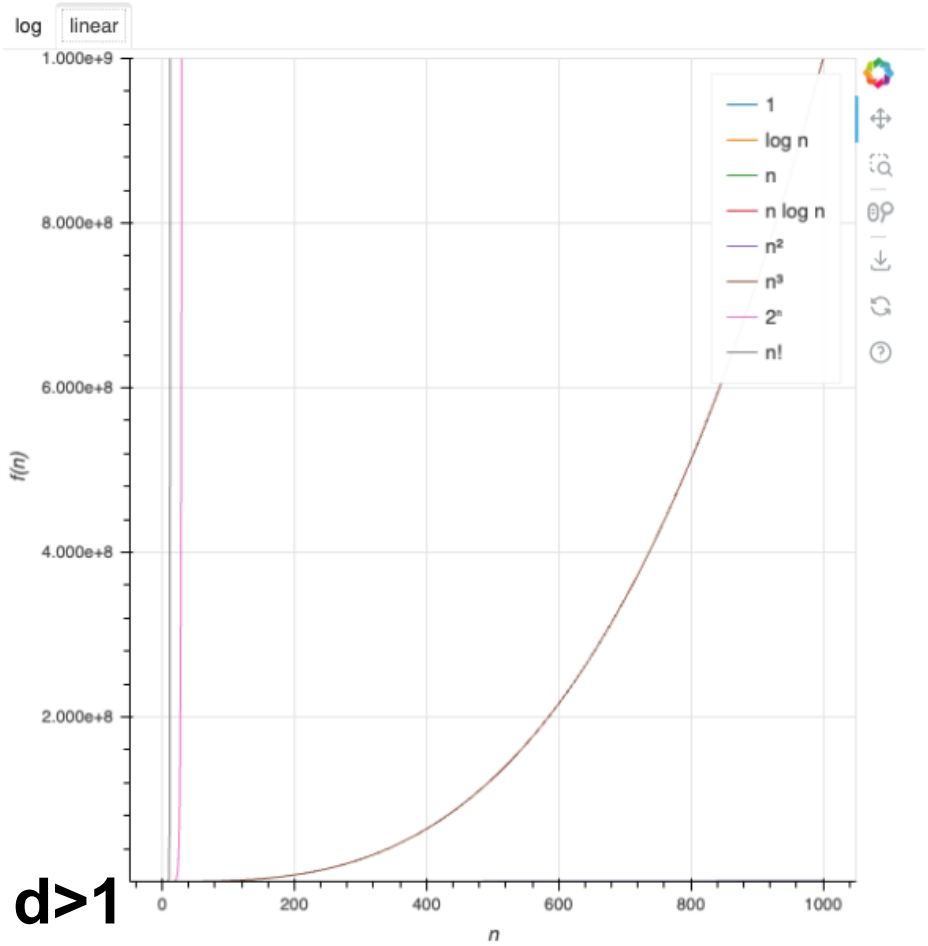


Logaritmisella asteikolla väli on kerrottu jollain kertoimella edelliseen nähden

Yleisiä muotoja käyrille

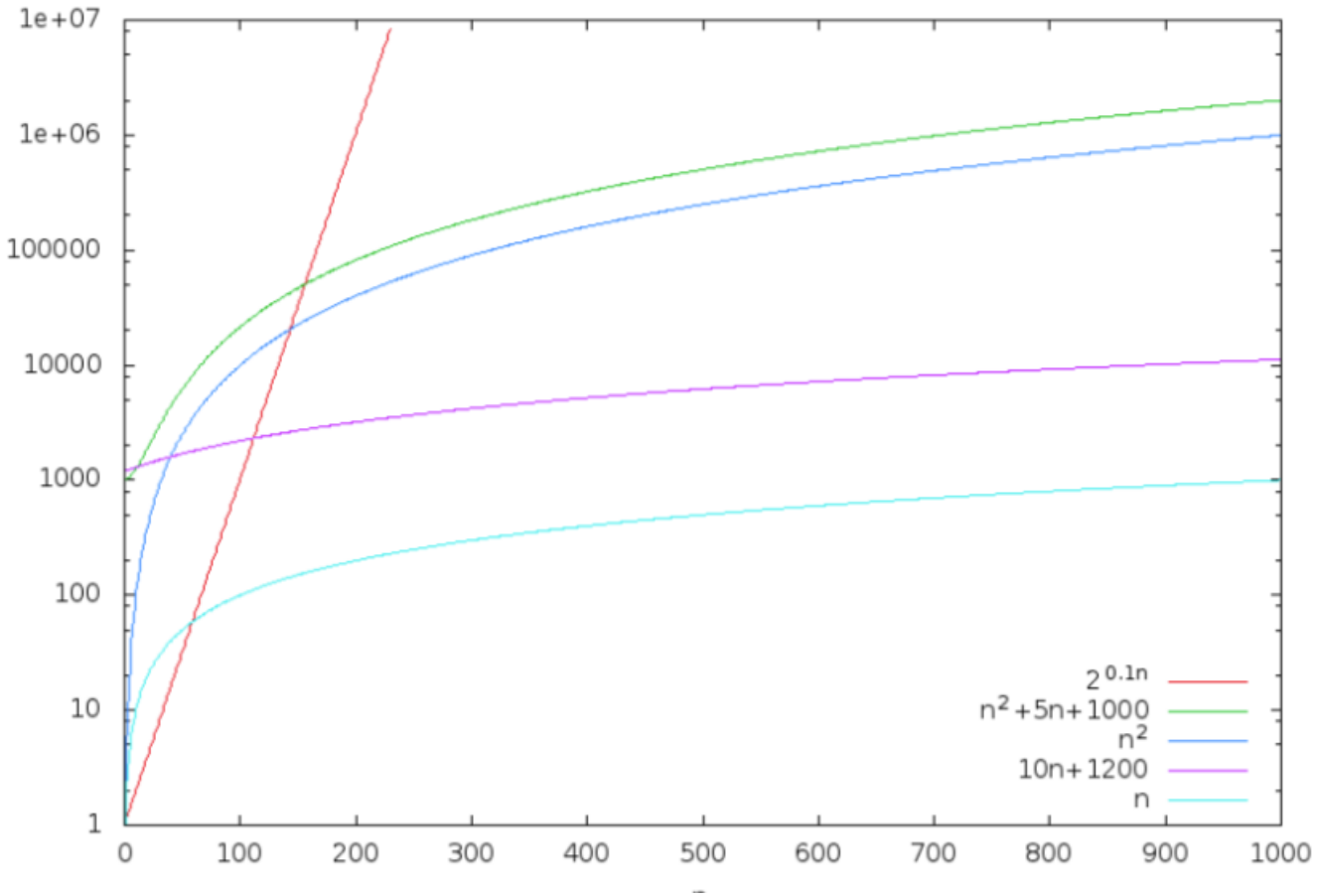
- **Vakioaikainen (constant, c)**
  - käyttää tietyn vakioajan
- **Logaritminen (log n)**
  - Yleensä oletetaan 2-kantainen
- **Lineaarinen (n)**
- **Linearitminen (linearitmic, n log n)**
- **Neliöllinen (quadratic, n^2)**
- **Kuutiollinen (cubic, n^3)**
- **Polynominen (polynomial, n^k), k>0**
- **Eksponentiaalinen (exponential, d^n), d>1**
- **Faktoriaalinen (factorial, n!)**

Koska lineaarinen ei erottele:



Skaalauksen vertailua käyttäen $\mathcal{O}$

- **$n = \mathcal{O}(10n+1200)$, määritelmän mukaan**
- **$10n + 1200 = \mathcal{O}(n)$, koska $10n + 1200 < 12n$, jossa $n > 600$**
  - (eli valitaan sopiva vakio c=12 ja sopiva kohta n>600)
- **Eli $10n + 1200$ ja n ovat toisiaan vastaavia big-O-notaatiossa**



Skaalauksen vertailua käyttäen $\mathcal{O}$

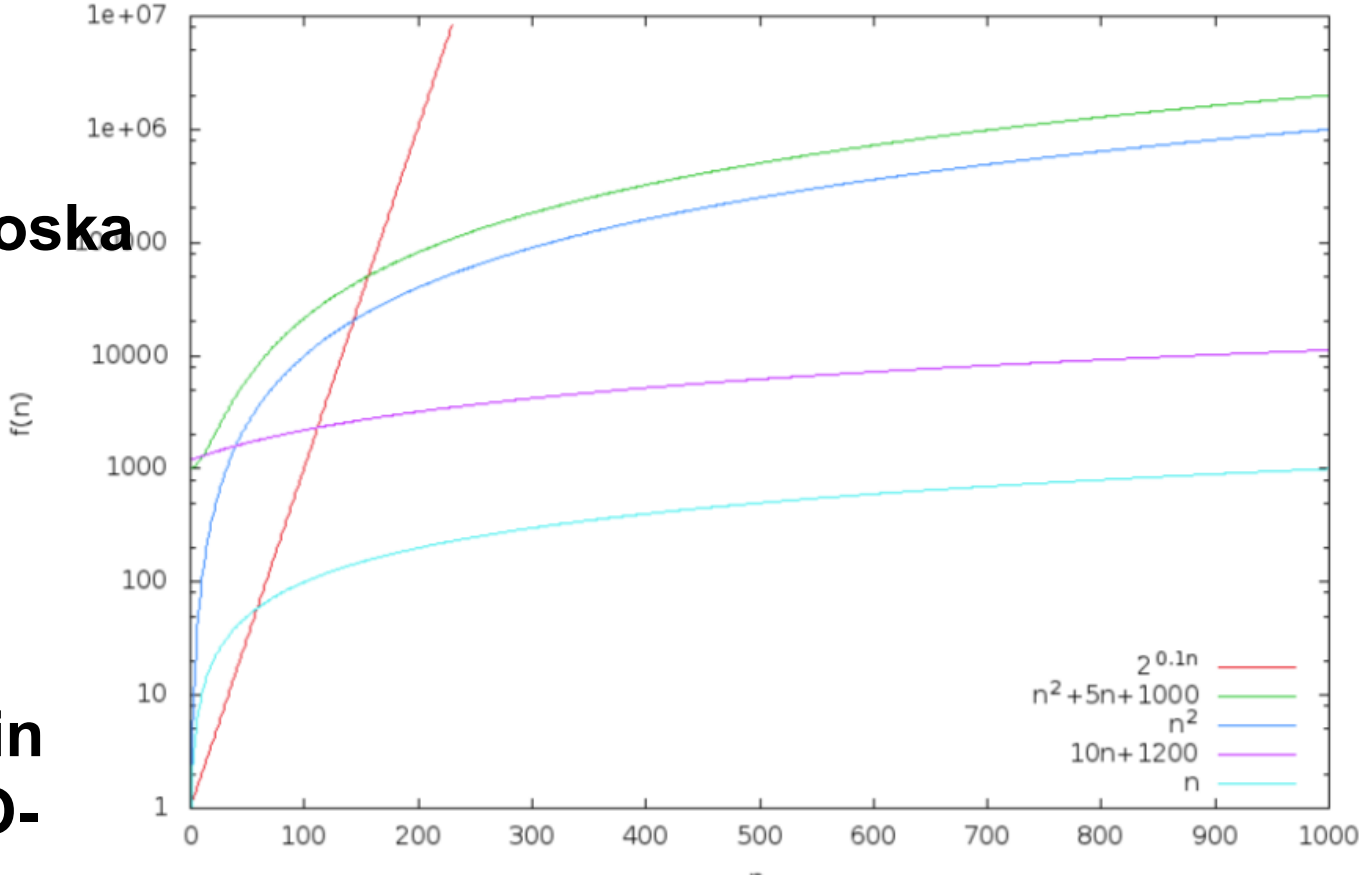
- **$10n+1200 = \mathcal{O}(n^2)$, koska $10n+1200 < 10n^2$ kun $n>12$**
- **$n^2 \neq \mathcal{O}(10n + 1200)$, koska**

$$\lim \frac{n^2}{10n+1200} \longrightarrow \infty$$

kun $n \longrightarrow \infty$

  - Eli ei löydy vakiota/ leikkauspistettä

- **n^2 kasvaa nopeammin kuin $10n$ +1200 big-O-notaatiossa**



Skaalauksen vertailua käyttäen $\mathcal{O}$

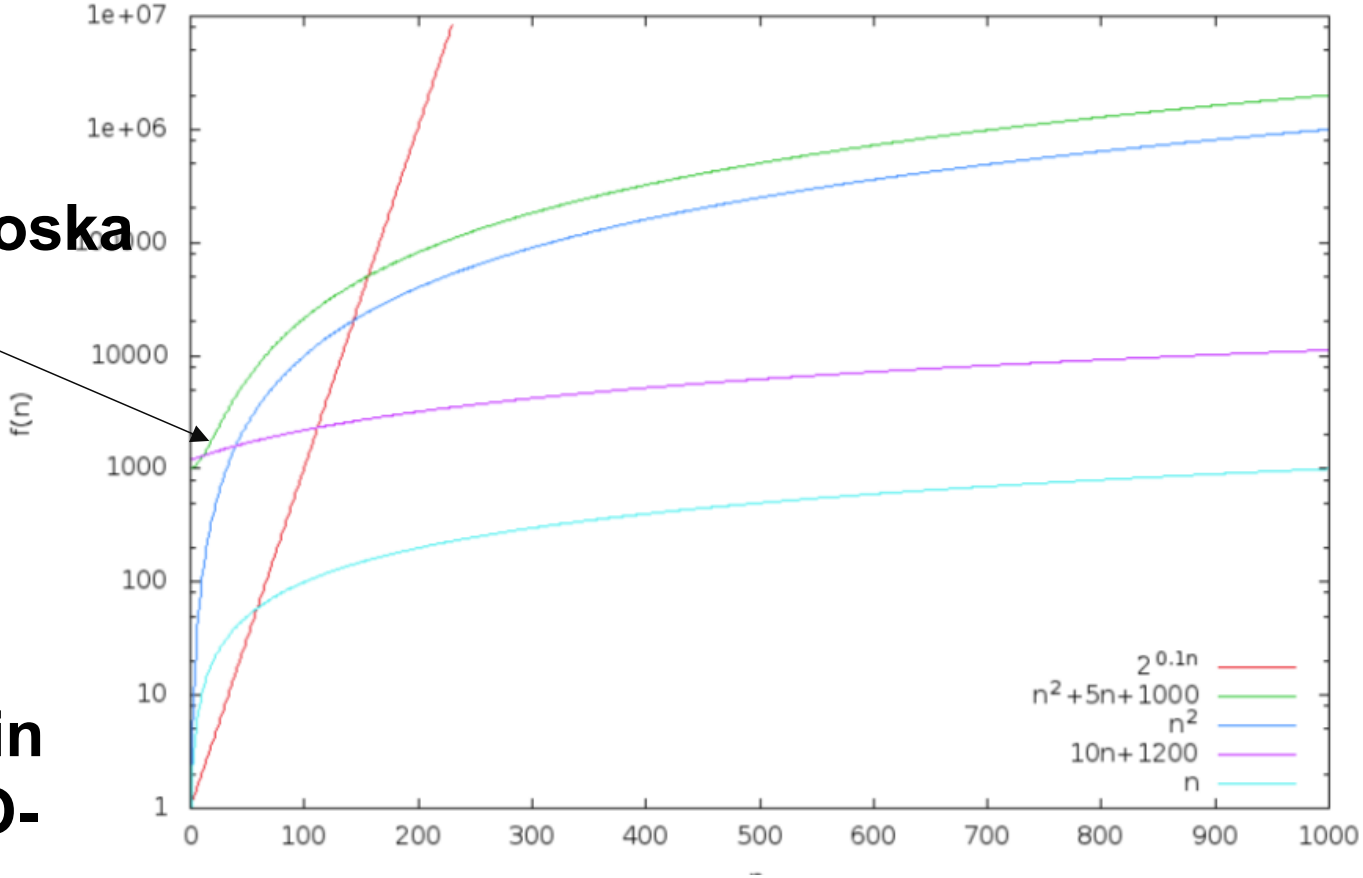
- **$10n+1200 = \mathcal{O}(n^2)$, koska $10n+1200 < 10n^2$ kun $n>12$**
- **$n^2 \neq \mathcal{O}(10n + 1200)$, koska**

$$\lim \frac{n^2}{10n+1200} \longrightarrow \infty$$

kun $n \longrightarrow \infty$

  - Eli ei löydy leikkauspistettä

- **n^2 kasvaa nopeammin kuin $10n$ +1200 big-O-notaatiossa**



Seuraavien määritelmien kanssa, onko $f_i = \mathcal{O}(f_j)$?

- $\mathbf{f_1}(n) = n^2 + 5n + 1000$,
- $\mathbf{f_2}(n) = n^2$
- $\mathbf{f_3}(n) = 2^{0,5n}$
- $\mathbf{f_4}(n) = 2^{\log n}$
- $\mathbf{f_5}(n) = 18n$
- $\mathbf{f_6}(n) = \log n^2$

https://presemo.aalto.fi/o2fi2026

- **Vinkki: polynomifunktioilla vain kertaluokka merkitsee**

Ω (omega) ja Θ (theta)

- **Voimme määritellä skaalaukselle myös asymptoottisen alarajan (Ω) ja asymptoottisen yhtäsuuruuden (Θ)**
- **Määritelmä (Ω) – kasvaa vähintään yhtä nopeasti kuin**

  $f(n) = \Omega(g(n))$,

  jos ja vain jos

  $g(n) = \mathcal{O}(f(n))$

- **Määritelmä (Θ) – kasvaa yhtä nopeasti kuin**

  $f(n) = \Theta(g(n))$,

  jos ja vain jos

  $f(n) = \mathcal{O}(g(n))$ ja
  $f(n) = \Omega(g(n))$

- (g on sama funktio, vain vakio erottaa)

Big-O ajoajalle

Ohjelman/funktion/metodin ajoaika on $\mathcal{O}(f)$

jos ja vain jos

kaikilla syötteen koolla n,

ajoaika on $\mathcal{O}(f(n))$ aikayksikköä.

Big-O ajoajalle

Ohjelman/funktion/metodin ajoaika on $\mathcal{O}(f)$

jos ja vain jos

kaikilla syötteen koolla n,

ajoaika on $\mathcal{O}(f(n))$ aikayksikköä.

- **Big-O on tärkeä konsepti algoritmi- ja tietojenkäsittelyteoriassa**
- **Sen voi yleistää muillekin resursseille, esim muistin käyttöön**
- **Jos voit todistaa, että algoritmisi on $\mathcal{O}(f)$ jollekin funktiolle f jonkin resurssin suhteen, sinulla on asymptoottinen yläraja sille, kuinka tuo resurssitarve skaalautuu**
- **Kun etsimme $\mathcal{O}(f)$:ää, olemme kiinnostuneita f:stä, joka kasvaa mahdollisimman vähän, kuitenkin noudattaen big-O:n määritelmää (vastaavasti: $\Omega(f)$ mahd. nopeasti kasvava f)**

Koodiin perustuva analyysi

Koodiin perustuva analyysi – matriisin kertolasku

- **Katsotaan joka laseketta ja kysytään kuinka monta vakioaikaista käskyä sen suorittaminen vie**

```
/** Returns a new matrix that is the product of this and that */
def *(that: Matrix): Matrix =
  val result = new Matrix(n)
  for row <- 0 until n; column <- 0 until n do
    var v = 0.0
    for i <- 0 until n do
      v += this(row, i) * that(i, column)
    result(row, column) = v
  end for
  result
end *
```

Koodiin perustuva analyysi – matriisin kertolasku

- **Matriisissa on n^2 numeroa – jokainen pitää alustaa**

```
/** Returns a new matrix that is the product of this and that */
def *(that: Matrix): Matrix =
  val result = new Matrix(n)                          // O(n^2)
  for row <- 0 until n; column <- 0 until n do
    var v = 0.0
    for i <- 0 until n do
      v += this(row, i) * that(i, column)
    result(row, column) = v
  end for
  result
end *
```

Koodiin perustuva analyysi – matriisin kertolasku

- **Silmukka käy läpi kaikki n^2 elementtiä**

```
/** Returns a new matrix that is the product of this and that */
def *(that: Matrix): Matrix =
  val result = new Matrix(n)                          // O(n^2)
  for row <- 0 until n; column <- 0 until n do        // O(n^2)
    var v = 0.0
    for i <- 0 until n do
      v += this(row, i) * that(i, column)
    result(row, column) = v
  end for
  result
end *
```

Koodiin perustuva analyysi – matriisin kertolasku

- **Arvon asettaminen on $\mathcal{O}(1)$, mutta silmukan vuoksi sen tekeminen vie $\mathcal{O}$ (n^2) kertaa**

```
/** Returns a new matrix that is the product of this and that */
def *(that: Matrix): Matrix =
  val result = new Matrix(n)                          // O(n^2)
  for row <- 0 until n; column <- 0 until n do        // O(n^2)
    var v = 0.0                                       // O(n^2)
    for i <- 0 until n do
      v += this(row, i) * that(i, column)
    result(row, column) = v
  end for
  result
end *
```

Koodiin perustuva analyysi – matriisin kertolasku

- **Silmukka itse on $\mathcal{O}(n)$, mutta $\mathcal{O}(n^2)$-silmukan sisällä, tekee $\mathcal{O}(n^3)$**

```
/** Returns a new matrix that is the product of this and that */
def *(that: Matrix): Matrix =
  val result = new Matrix(n)                          // O(n^2)
  for row <- 0 until n; column <- 0 until n do        // O(n^2)
    var v = 0.0                                       // O(n^2)
    for i <- 0 until n do                             // O(n^3)
      v += this(row, i) * that(i, column)
    result(row, column) = v
  end for
  result
end *
```

Koodiin perustuva analyysi – matriisin kertolasku

- **Useampi vakioaikainen operaatio silmukan sisällä. Tärkeää: miksi arvoihin pääsy on vakio tässä tapauksessa?**

```
/** Returns a new matrix that is the product of this and that */
def *(that: Matrix): Matrix =
  val result = new Matrix(n)                          // O(n^2)
  for row <- 0 until n; column <- 0 until n do        // O(n^2)
    var v = 0.0                                       // O(n^2)
    for i <- 0 until n do                             // O(n^3)
      v += this(row, i) * that(i, column)             // O(n^3)
    result(row, column) = v
  end for
  result
end *
```

Koodiin perustuva analyysi – matriisin kertolasku

- **Useampi vakioaikainen operaatio silmukan sisällä. Tärkeää: miksi arvoihin pääsy on vakio tässä tapauksessa?**

```
/** Returns a new matrix that is the product of this and that */
def *(that: Matrix): Matrix =
  val result = new Matrix(n)                          // O(n^2)
  for row <- 0 until n; column <- 0 until n do        // O(n^2)
    var v = 0.0                                       // O(n^2)
    for i <- 0 until n do                             // O(n^3)
      v += this(row, i) * that(i, column)             // O(n^3)
    result(row, column) = v
  end for
  result
end *
```

Esim. Tämä on $\mathcal{O}(1)$, koska käytetty tieto-rakenne on nyt Array

Koodiin perustuva analyysi – matriisin kertolasku

- **Suoritetaan $\mathcal{O}(n^2)$ kertaa. Jälleeen oletetaan, että arvon asettaminen elementille on $\mathcal{O}(1)$**

```
/** Returns a new matrix that is the product of this and that */
def *(that: Matrix): Matrix =
  val result = new Matrix(n)                          // O(n^2)
  for row <- 0 until n; column <- 0 until n do        // O(n^2)
    var v = 0.0                                       // O(n^2)
    for i <- 0 until n do                             // O(n^3)
      v += this(row, i) * that(i, column)             // O(n^3)
    result(row, column) = v                           // O(n^2)
  end for
  result
end *
```

Koodiin perustuva analyysi – matriisin kertolasku

- **Lopputuloksen palautus $\mathcal{O}(1)$**

```
/** Returns a new matrix that is the product of this and that */
def *(that: Matrix): Matrix =
  val result = new Matrix(n)                          // O(n^2)
  for row <- 0 until n; column <- 0 until n do        // O(n^2)
    var v = 0.0                                       // O(n^2)
    for i <- 0 until n do                             // O(n^3)
      v += this(row, i) * that(i, column)             // O(n^3)
    result(row, column) = v                           // O(n^2)
  end for
  result                                              // O(1)
end *
```

Koodiin perustuva analyysi – matriisin kertolasku

- $\mathcal{O}(n^2) + \mathcal{O}(n^2) + \mathcal{O}(n^2) + \mathcal{O}(n^3) + \mathcal{O}(n^3) + \mathcal{O}(n^2) + \mathcal{O}(1) = ?$

```
/** Returns a new matrix that is the product of this and that */
def *(that: Matrix): Matrix =
  val result = new Matrix(n)                          // O(n^2)
  for row <- 0 until n; column <- 0 until n do        // O(n^2)
    var v = 0.0                                       // O(n^2)
    for i <- 0 until n do                             // O(n^3)
      v += this(row, i) * that(i, column)             // O(n^3)
    result(row, column) = v                           // O(n^2)
  end for
  result                                              // O(1)
end *
```

Koodiin perustuva analyysi – matriisin kertolasku

- $\mathcal{O}(n^2) + \mathcal{O}(n^2) + \mathcal{O}(n^2) + \mathcal{O}(n^3) + \mathcal{O}(n^3) + \mathcal{O}(n^2) + \mathcal{O}(1) = \mathcal{O}(n^3)$

```
/** Returns a new matrix that is the product of this and that */
def *(that: Matrix): Matrix =
  val result = new Matrix(n)                          // O(n^2)
  for row <- 0 until n; column <- 0 until n do        // O(n^2)
    var v = 0.0                                       // O(n^2)
    for i <- 0 until n do                             // O(n^3)
      v += this(row, i) * that(i, column)             // O(n^3)
    result(row, column) = v                           // O(n^2)
  end for
  result                                              // O(1)
end *
```

Koodiin perustuva analyysi – peukalosäännöt

- **Sisäkkäiset silmukat yli ongelman (datan) koon kasvattavat kertaluokkaa**
  - Koska kaikki silmukan sisällä pitää tehdä *n* kertaa
  - Mutta silmukat voi olla dynaamisia (käsitellään vain osaa)
- **Sama rekursiolle, missä pitää olla tarkkana sen kanssa, kuinka ’syvälle’ rekurssio menee**
  - Palaamme tähän myöhemmin
- **Metodien/funktioiden kutsut voivat piilottaa monimutkaisuutta**
  - Tunne käyttämiesi metodien tehokkuus

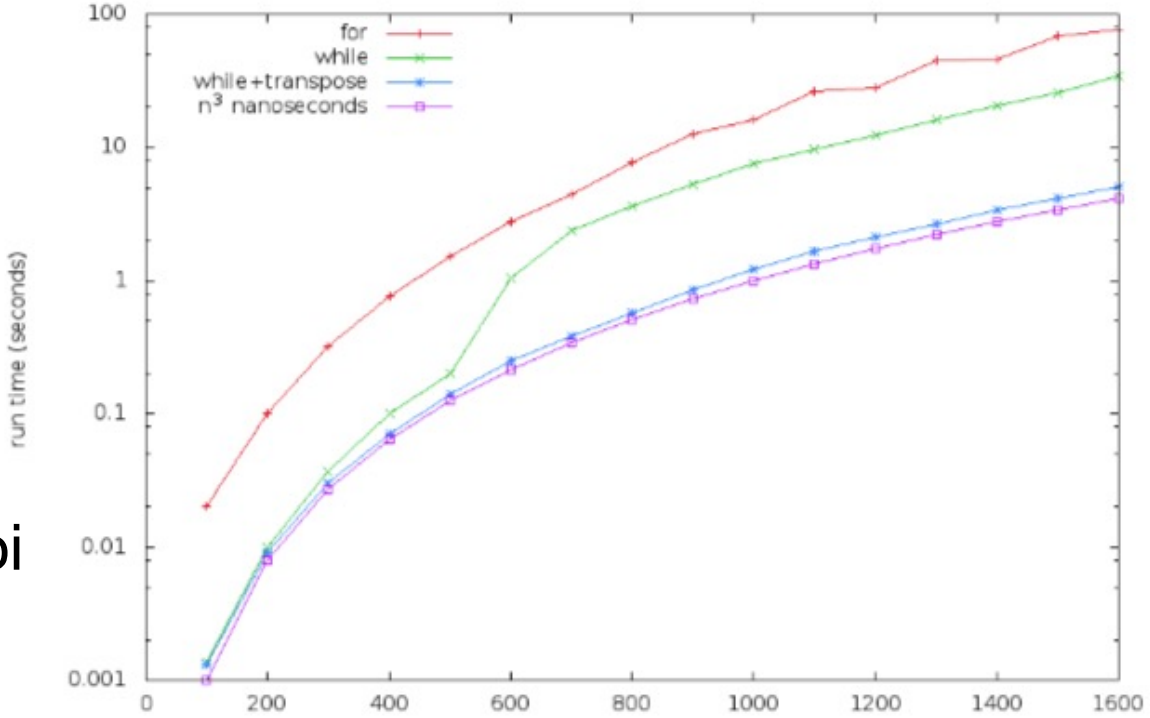
Koodiin perustuva analyysi – peukalosäännöt

- **Tietorakenteet merkitsevät!**
  - Esim. Matriisin elementit Array- tai List-rakenteessa
  - Siksi varmista aina, että tiedät mitä tietorakennetta käytetään
- **Dokumentoi aina suorituskykyominaisuudet, kun luot kirjaston tai paketin**
  - Kuten Scalan kokoelmissa oli täällä: https://docs.scala-lang.org/overviews/collections-2.13/performance-characteristics.html

- **Kysymys: kuinka tehokas meidän Matrix-toteutus olisi, jos käyttäisimme `Array`n sijaan `ListBuffer`ia alkioiden tallentamiseen?**

Vakiotekijän (eli koodin) optimointi

- **Big-O sivuttaa vakiotekijät**
- **Algoritmien ja tieto-rakenteiden skaalautuvuus on erittäin tärkeää tehokkaalle koodille**
  - Mutta käytännössä vakiotekijöiden optimointi voi johtaa suureen ajan (ja energian) säästöön
  - Algoritmi määrää muodon!
- **Esim. Matriisioperaatioiden optimointi: lähes 30-kertainen kasvu, mutta silti vielä $\mathcal{O}(n^3)$**



- **Vakiotekijän optimointi vähentää luettavuutta, eli tee vain jos oikeasti tarpeen**
- **Tarkista eka algoritmisi!**

https://a1120.cs.aalto.fi/current/notes/round-efficiency--bigoh.html#matrix-multiplication-reloaded-constants-do-matter-too

Mutta onko tehokkuudella väliä? Kyllä!

- **Käytännössä esim.**
  - Jättimäiset datasetit ⇒ jättimäinen n
  - Miljoonia käyttäjiä ⇒ sovellus ajetaan miljoonia kertoja
  - Energiankulutus* ⇐ laskenta yleisesti vaatii energiaa (Landauerin periaate)
- **Teoriassa**
  - Laskennan kompleksisuus (computational complexity) on ala, joka tutkii kuinka vaikeita ongelmat ovat ja mitä pohjimmiltaan voimme laskea

* Historiallisesti teknologiset innovaatiot ovat kattaneet tämän kasvun. Laskenta / kWh on tuplaantunut joka 1,6 vuosi 1950 luvulta asti (Koomeyn laki), mutta n 10 vuotta sitten se on hidastunut jonkin verran (2,5), ja on avointa, kuinka kauan trendi voi jatkua.

https://en.wikipedia.org/wiki/Computational_complexity
https://en.wikipedia.org/wiki/Landauer%27s_principle
https://en.wikipedia.org/wiki/Koomey%27s_law

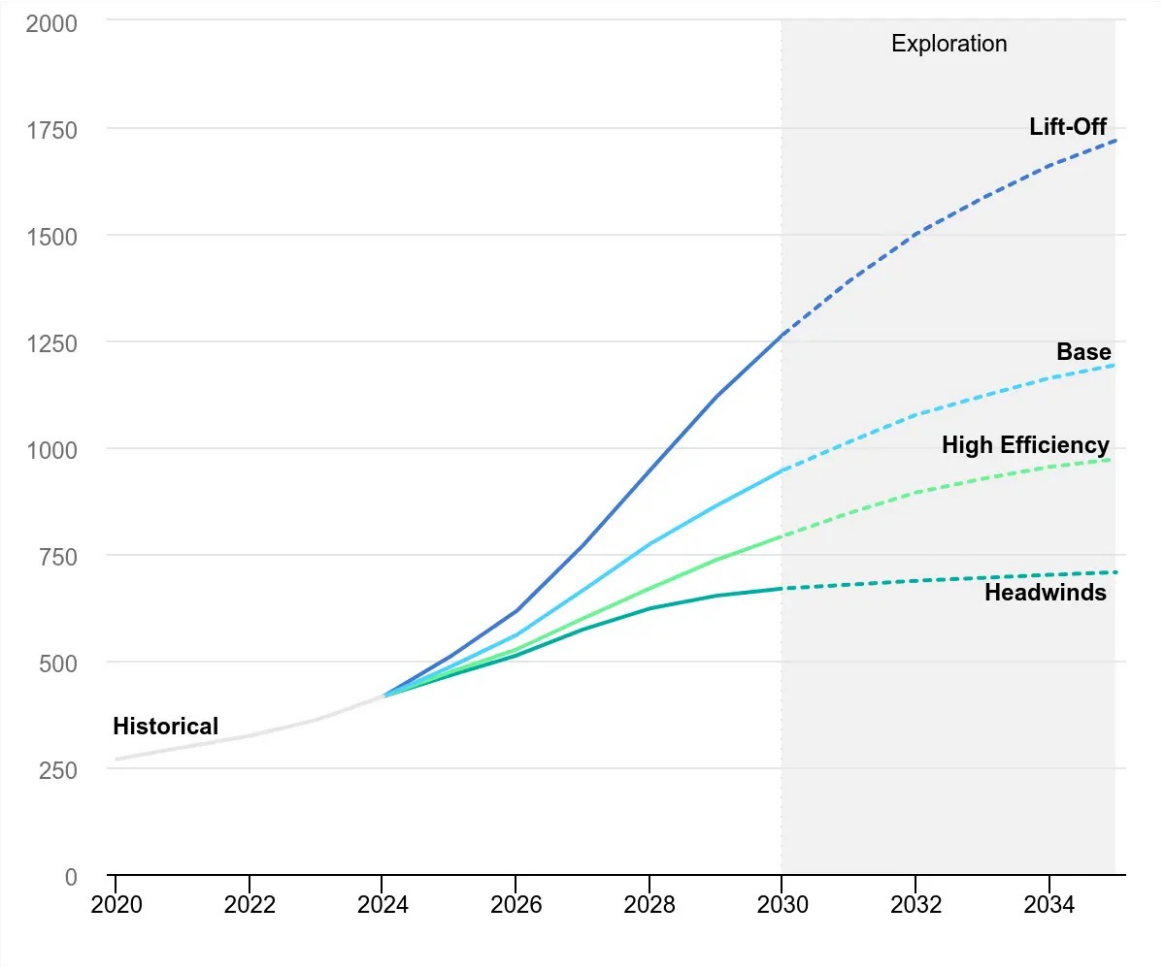
Mutta onko tehokkuudella väliä? Kyllä!

- **Ajoaikavertailu, olettaen että f(*n*) vie 1 ns**
  - (d on päiviä, a vuosia; vrt. Maailmankaikkeuden ikä 13,8-26.7x10^9)

n	$\log n$	n^2	n^3	2^n	$n!$
10	3.4ns	0.1μs	1μs	1μs	3.7ms
20	4.4ns	0.4μs	8μs	1.1ms	77.2a
50	5.7ns	2.5μs	125μs	13.1d	9.6×10^{47}a
100	6.7ns	10 μs	1ms	40×10^{12}a	
1000	10ns	1 ms	1s		
10000	13.3ns	0.1s	16.7min		
100000	16.7ns	10s	11.6d		
10^6	20.0ns	16.7min	31.8a		
10^9	29.9ns	31.8a	32×10^9a		

Mutta onko tehokkuudella väliä? Kyllä!

- IEA (2025): globaali datakeskusten energiakulutus 2020-2035, IEA,Paris https://www.iea.org/data-and-statistics/charts/global-data-centre-electricity-consumption-by-sensitivity-case-2020-2035, Licence: CC BY 4.0



Järjestäminen ja haku

Aalto-yliopisto
Perustieteiden
korkeakoulu

Haku (searching) – yleinen tehtävä

- **Elementin löytäminen kokoelmasta on hyvin yleinen tehtävä/ongelma**
- **Tähän liittyy kokoelman elementtien läpikäynti kunnes**
  - Elementti löytyy, tai
  - Tiedämme, että elementti ei ole kokoelmassa
- **Koska hakeminen (haku, searching) on erittäin yleistä, sen tehokkuus (efficiency) on tärkeää**

Epäjärjestys ja lineaarinen haku

- **Jos kokoelma on epäjärjestyksessä (disordered) tai emme tiedä siitä mitään, paras tapa on lineaarinen haku (linear search)**
  - Esim. Etsi `19` listasta `(4,24,7,11,4,7,21,23,8,19,1,30)`
  - Käy kokoelma läpi alusta loppuun kunnes elementti löytyy tai saavutetaan loppu

```
def linearSearch[T](s: IndexedSeq[T], k: T): Boolean =
  var i = 0
  while(i < s.length) do
    if(s(i) == k) then return true // found k at position i
    i = i + 1
  end while
  false // no k in sequence
end linearSearch
```

Epäjärjestys ja lineaarinen haku (jatkuu)

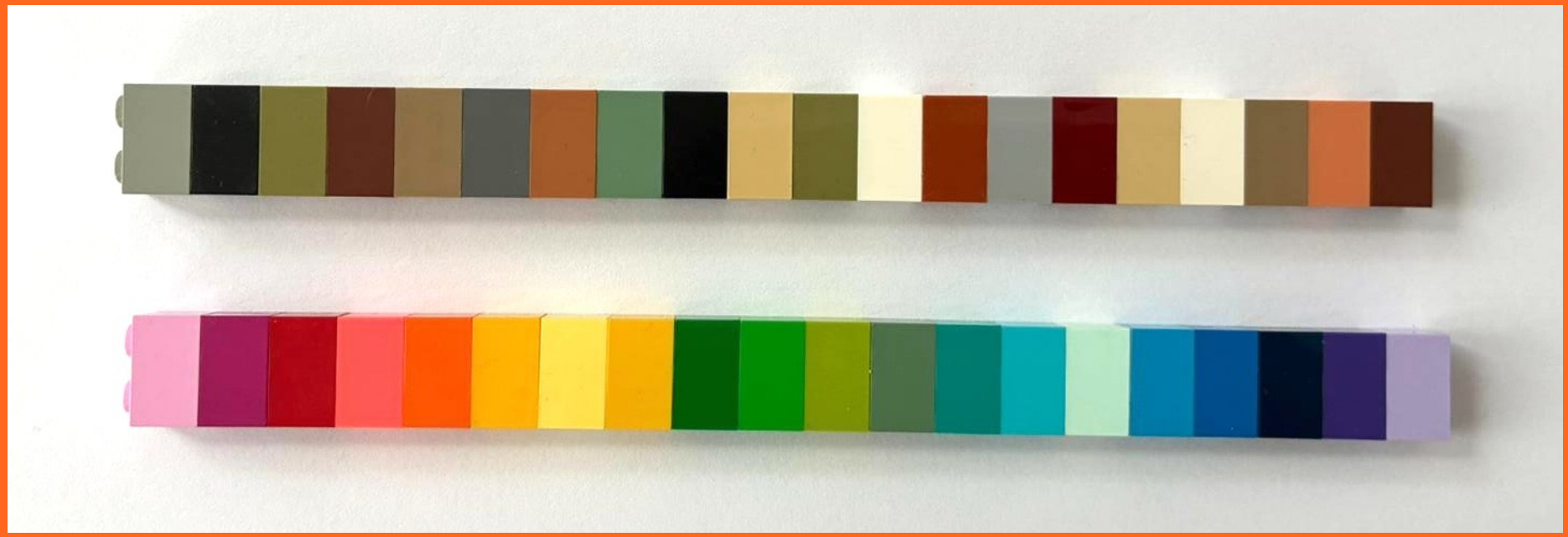
```
def linearSearch[T](s: IndexedSeq[T], k: T): Boolean =
  var i = 0
  while(i < s.length) do
    if(s(i) == k) then return true // found k at position i
    i = i + 1
  end while
  false // no k in sequence
end linearSearch
```

- **Lineaarinen haku on $\mathcal{O}(n)$ olettaen vakioajan yhtäsuuruuden testaamiselle ja indeksoidulle elementin saamiselle (`s(i)`)**
- **Ei paha, mutta koodissa usein haetaan samasta kokoelmasta useita kertoja toistuvasti**
  - Haetaan jokainen elementti kertaalleen → silloin koodi on $\mathcal{O}(n^2)$
  - Yksi n toistetusta hausta * n itse haku

Indeksointi – rakenteen määrittäminen

- **Jos tiedämme, että kokoelmasta tullaan hakemaan toistuvasti, voi olla kannattavaa analysoida se ensin**
- **Tätä prosessia kutsutaan yleisesti indeksoinniksi (indexing)**
- **Yleisin indeksoinnin muoto on järjestäminen (sorting)**
  - Tämä vaatii sen, että meillä on joku idea, miten elementit voidaan jotenkin järkevästi järjestää
- **Indeksointi tehdään kerran, joten jos sen kustannus ei ole kohtuuttoman suuri, sen tekeminen voi olla kannattavaa**

Binäärihaku



Aalto-yliopisto
Perustieteiden
korkeakoulu

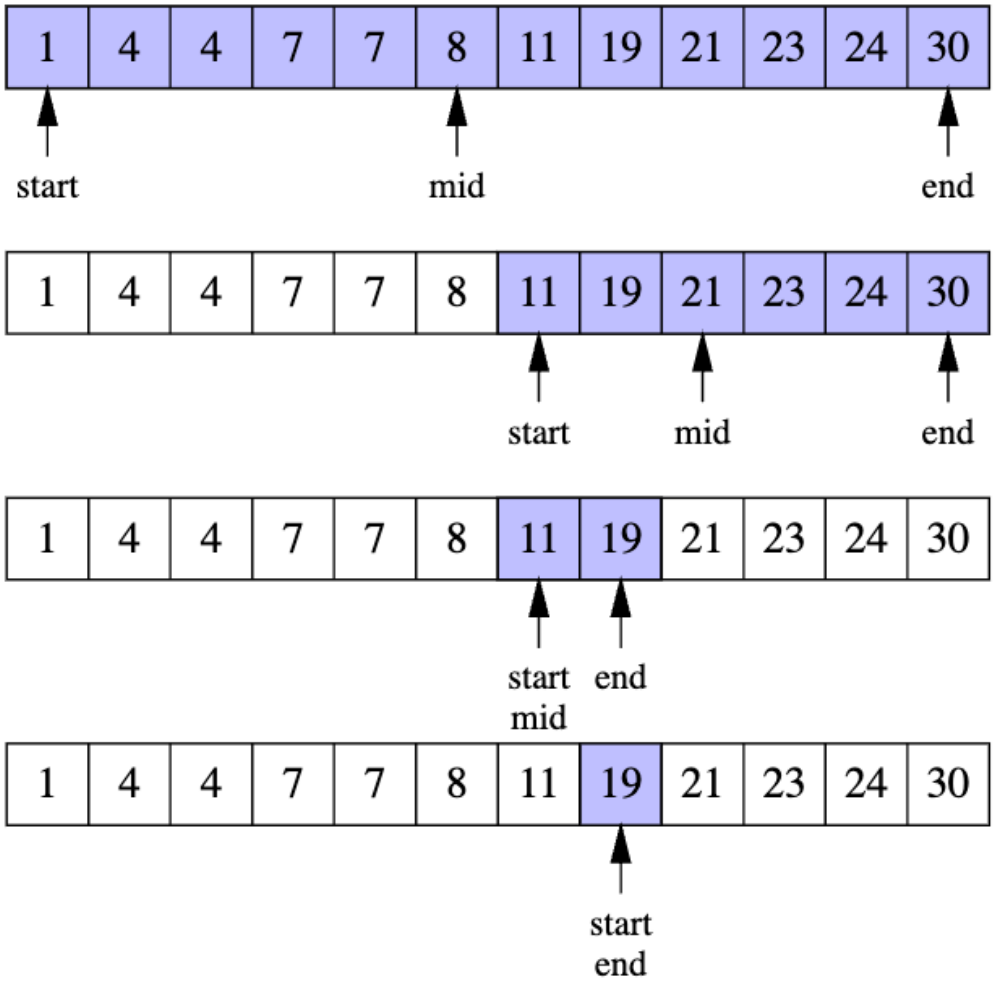
Järjestys (order) ja binäärihaku (binary search)

- **Oletetaan, että kokoelman data on järjestetty (ordered)**
  - Esim. Etsi `19` listasta `(1,4,4,7,7,8,11,19,21,23,24,30)`
- **Auttaako rakenne (structure) meitä tekemään lineaarista hakua paremman haun?**
- **Kyllä – binäärihaku on tehokkaampi**

Binäärihaku

- **Oletetaan, että meidän pitää löytää elementti `k` kokoelmasta `s`**
- **Oletetaan, että `s` on järjestetty nousevaan järjestykseen**
- **Jos `s` on tyhjä, elementtiä ei voi löytää. Lopeta.**
- **Olkoon `m` keskimmäinen (alaspäin pyöristettynä) elementti kokoelmassa `s`:**
  - Jos `k` = `m`, elementti löytyi. Lopeta.
  - Jos `k` < `m`, elementti voi löytyä ainoastaan kokoelman alkupuoliskosta
    - Etsitään rekursiivisesti vain alkupuoliskosta
  - Jos `k` > `m`, elementti voi löytyä ainoastaan kokoelman loppupuoliskosta
    - Etsitään rekursiivisesti vain loppupuoliskosta

Binäärihaku – esimerkki: etsi 19



Binäärihaku

```
def binarySearch[T](s: IndexedSeq[T], k: T)
  (using Ordering[T]) : Boolean = // Split to rows to fit a slide

  import math.Ordered.orderingToOrdered // To use the given Ordering
  def inner(start: Int, end: Int): Int =
    if !(start < end) then start
    else
      val mid = (start + end) / 2
      val cmp = k compare s(mid)
      if cmp == 0 then mid                          // k == s(mid)
      else if cmp < 0 then inner(start, mid-1)      // k < s(mid)
      else inner(mid+1, end)                        // k > s(mid)
    end if
  end inner
  if s.length == 0 then false
  else s(inner(0, s.length-1)) == k
end binarySearch
```


Kts koko koodi: https://a1120.cs.aalto.fi/2026/notes/round-efficiency--binarysearch.html

Binäärihaun tehokkuus

- **Oletetaan edelleen että vertailu (=, <, >) ja alkion saaminen sekvenssistä vie vakioajan**
- **Jokainen ‘askel’ binäärihaun algoritmissa sisältää vain $\mathcal{O}(1)$ operaatiota**
- **Mutta sitä kutsutaan rekursiivisesti**
- **Mikä on maksimi määrä kertoja, jota sitä kutsutaan?**
  - Pysähtyy, kun elementti on löydetty, tai kutsuttu 0-pituiselle jonolle
  - Jokaisessa rekursiivisessa kutsussa jono puolitetaan
- **Jos alkuperäisen jonon pituus on n, niin rekursiivisten kutsujen pituus on $\frac{n}{2^1}, \frac{n}{2^2}, \ldots$ kunnes jokin $\left\lfloor \frac{n}{2^k} \right\rfloor = 0$**
  - Eli $k = \log n$ rekursiivista kutsua
- **Eli binäärihaulla on $\mathcal{O}(\log n)$**

Tehokkuus?

- **Sama järjestetty kokoelma ja tehtävä:**
  - Etsi `19` listasta `(1,4,4,7,7,8,11,19,21,23,24,30)`
- **Mikä on lineaarisen haun rekursiivisten kutsujen määrä?**
  - Entäpä jos alkiota ei ole listassa?
- **Entäpä binäärihaun kutsujen määrä? (muistutus:**
  - Jos alkuperäisen jonon pituus on n, niin rekursiivisten kutsujen pituus on $\frac{n}{2^1}, \frac{n}{2^2}, \ldots$ kunnes jokin $\left\lfloor \frac{n}{2^k} \right\rfloor = 0$
  - Eli k = log n rekursiivista kutsua
  - Eli binäärihaulla on $\mathcal{O}(\log n)$)

https://presemo.aalto.fi/o2fi2026

Järjestämisen + binäärihaun tehokkuus

- **Kun kokoelma on järjestetty, haku voidaan tehdä $\mathcal{O}(\log n)$**
- **Mutta entäpä järjestämisen kulut?**
  - Vertailupohjaiset järjestämisalgoritmit, kuten Scalan `sorted`-metodi, toimivat $\mathcal{O}(n \log n)$
- **Järjestämis+haku-tehokkuus on**
  $\mathcal{O}(n \log n) + \mathcal{O}(\log n) = \mathcal{O}(n \log n)$
- **Tämä on on huonompi kuin lineaarisen haun $\mathcal{O}$(n)!**
  - Kyllä, mutta teemme järjestämisen vain kerran!
- **Jos teemme n toistettu hakua, silloin:**
  - Binäärihaku $\mathcal{O}(n \log n) + \mathcal{O}(n) \times \mathcal{O}(\log n) = \mathcal{O}(n \log n)$
  - Lineaarinen haku $\mathcal{O}(n) \times \mathcal{O}(n) = \mathcal{O}(n^2)$

Peukalosääntö:

- **vain muutama haku: ei välttämättä kannata prosessoida**
- **Jos hakujen määrä on suuri: indeksointi kannattaa**

(comparison-based sorting algorithms
https://en.wikipedia.org/wiki/Sorting_algorithm)

Jatkokursseja

- **Paljon lisää järjestämis- ja hakualgoritmeista ja tehokkuudesta kurssilla CS-A1140 Tietorakenteet ja algoritmit syksyllä**
- **Jota seuraa ihan maisteritason pääaineen verran kursseja**
- **Tehokkuutta voi lisätä myös mm. rinnaisuuden avulla, lisää mm. CS-E4580 Programming Parallel Computers**

Tehtävät

1. **Mediaani ja prosenttipisteet (percentiles)**
2. **Big-O Quiz**
3. **Binäärihaku: juurten haku**
4. **Binäärihaku: alijoukon haku**
5. **Parien summa (pair sum)**
6. **Haastetehtävä: yksi teratavu**

- **Muista, että Scalan sorted-metodi toimii $\mathcal{O}(n \log n)$**
- **Tutki binäärihaun periaatteita ja sen Scala-toteutusta kurssimateriaalista**
  - Tästä on kaksi tehtävää
  - Piirrä kuva hajoita ja hallitse –algoritmin askeleista binäärihakutehtävissä
- **Testaile kynällä ja paperilla Pair sum –tehtävässä kehitellessäsi ideaa nopealle algoritmille**