# Algoritme:

En veldefineret computer procedure der omdanner input til output.

Der kan være flere rigtige algoritmer til hver opgave

Kriterier: for hvert input giver den det korrekte output

## Kompleksitet:

Space: hvor meget hukommelse bruger den

Tid: hvor lang tid tager algoritmen

God eller dårlig algoritme

Vil måske afhænge af input: best, worst og average case.

Incorrect algoritme: giver måske ikke output på input, eller giver forkert output

Brugbar i tilfælde hvor der ikke er nogen bedre korrekt algoritme, men man kender fejlprocenten og margin.

## Lineær søgning:

Defineret af størrelse på input

Kompleksitet som funktion af input størrelse n

Space used: array størrelse

Best case: A[1] = q, running time =1, konstant tid

Worst case: A[n] = q, running time = n, lineær tid

Avarage case: q er forventet I midten af sekvensen, running time = n/2, stadig lineær tid

Ikke godt hvis der er rigtig mange data. Big Data æra

## Binær søgning

Virker hvis data er sorterede.

Ideer bag: del og hersk (divide and conqure), en af nøgle design teknikkerne.

Rækkefølgen af data sørger for at elementer man leder efter ikke er ignoreret.

**Sorterings problem:**

Input sekvens er kaldt en instans af sorterings problem.

Der kan være uendeligt mange input

Nogle algoritmer er hurtigere end andre.

**Fremgangsmåde**

1. Analyser specifikationer på forholdet mellem input og output
2. Design algoritmer og datastrukturer
3. Implementer algoritmer i rigtigt programmeringssprog

**Algoritme mål**

* Korrekthed
* Effektivitet

Datastruktur:

Gemmer og organiserer date for at få tilgang til og

Datastruktur: Hvordan organiserer vi data der skal håndteres

Algoritme: Beskriv processen hvor data er håndteret

# Ram model:

Hver instruktion tager en konstant tid betinget af maskinen den kører på.

Instruktioner er operationer som aritmetiske, data bevægelse, kontrol(return fx), sammenligning.

Kompleksitet handler om hvor mange instruktioner ved et givent input.

Husk alle sammenligninger – også i if og for.

tj er afhængig af hvor meget j er i et forloop.

Best case vil altid være 1.

**Average case**: svær at prædefinerer, handler om hvilke input er mest sandsynlige. Bliver kun talt om i Quick sort(i kurset).

**Worst case:** giver en højeste grænse, som garanti for at det ikke bliver ofte. Kan ske ofte. Avarage er ofte lige så slemt som worst.

# Asymptotic notation

Bruges til at simplificere analyse af køretid ved at fjerne ligegyldige detaljer, 3\*n^2 = n^2. Ignorer makine afhængig tid.

For at lave Theta Θ som ingenør: ignorer forstående konstanter fx 3x, ignorer 3. Find det stærkeste led. Det er det der siger noget.

Konstant c< poly-logarithm, lgkn<polynomial, na< exponential, bn

Man skal finde punktet n0 hvor f(x) altid vil være større end g(x)

Sammenlign to algoritmer. Hvor hurtigt er g(x) vokset højere end f(x). Hvilken der er bedst ændrer sig i forhold til x.

O notation = asymptotic upper bound. Worst case running time is no greater than O(f(x))

Θ notation = asymptotic tight bound. Der er en konstant der altid vil gøre f(x) højere end g(x), og der ere n konstant der altid vil gøre f(x) lavere end g(x).

Ω notation = asymptotic lower bound. F(x) gror asymptotically hurtigere end g(n) hvis f(n) = Ω(g(n))

**Concrete complexity:**

Refererer til resultat af ram analyse, med mange detaljer

**Abstrakt kompleksitet:**

Refererer til resultat af saymptotic analyse, theta, Big-O…

# Commun time compleksitet:

## Perfekt

Konstant tid  Θ(i)

## Feasible alogrithmes – godt

Log time Θ(lg n)

Linear time Θ(n)

Log linear time Θ(n log n)

## Polynomial time – okay til små

Quadratic time Θ(n^2)

Cubic time Θ(n^3)

## Infeasible – duer slet ikke, umuligt for computer

Eksponential Θ(2^n)

# Monotenicity:

En function er “monotonically increasing” hvis m er mindre end eller lig n også betyder at f(m) er mindre end eller lig f(n): m <= n, f(m) <= f(n)

En funktion er ”monotonically decreasing” hvis m er mindre end eller lig n betyder at f(m) er større end eller lig f(n): m <= n, f(m) >= f(n)

En funktion er ”strictly increasing” hvis m er mindre end n betyder at f(m) er mindre end f(n)

En funktion er ”stricktly decreasing” hvis m er mindre end n betyder at f(m) er større end f(n)

# Algoritmer:

## Sortering:

Insertion sort(s. 16).

# Design af algoritmer

(s. 29)

Forskellige typer design teknikker. Der er incremental fremgangsmåde.

## Divide and conquer

Ofte let at bestemme running time. Deler sig op i forskellige dele der ligner hovedproblemet, men er mindre. De gør dette rekursivt så mange af delene bliver ordnet hver for sig, og samler sig så til sidst og løser problemet.

Tre step i hver rekursions niveau:

**Divide:** Del problemet ind i x antal underproblemer der hver er små instanser af samme problem.

**Conquer:** Løs underproblemerne rekursivt. Hvis underproblemerne er små nok skal man bare løse dem ligefremt.

**Combine:** Kombiner e forskellige underproblmer til løsningen af det originale problem.

(s. 34)

Når en algoritme har et rekursivt kald til sig selv, kan man ofte beskrive dets running time med ”recurrence equation” eller ”recurrence”. Denne beskriver den overordnede køretid af en algoritme med størrelsen n ud fra køretiden på mindre input.

(side 65)

Når et rekursivt problem bliver så småt at man ikke kan dele det rekursivt igen, siger man at ”the recursion bottoms out” og at man er kommet ned til ”the base case”.

Nogle gange skal man løse underproblemer der ikke er helt de samme som det originale problem.

Her er beskrevet tre metoder til at løse rekursisioner:

* Substitution method: gætter på en grænse og bruger matematisk induktion til at bevise at det er rigtigt
* Recursion tree method: Laver rekursionen om til et træ hvis grene repræsenterer de forskellige niveauer af rekursionen. Der bruges teknikker til ”bounding summations (midt s. 66) til at løse denne rekursion.
* Master method: giver grænser for rekursioner i formen T(n) = aT(n/b)+f(n), hvor a >= 1, b > 1 og f(n) er en given funktion.

Hvis man kun kan bestemme upper bound af en funktion bruges O(n), hvis man kun kan bestemme lower bruges Ω.

## Maksimum subarray problem

(side 68)

Hvis man har et array og finder det underarray med den højeste sum, er det ”maximum subarray”. Der kan være flere i et array. Dette er kun interessant når der er negative indgange i arrayet.