MA1: L03

Tom Smedsaas

Introduktion till komplexitetsanalys

Beräkna x^n iterativt

$$x^n = \begin{cases} 1 & \text{om } n = 0 \\ x \cdot x \cdot x \cdot \ \ldots \ \cdot \mathrm{x} & \text{om } n > 0 \end{cases}$$

```
def power(x, n):
    result = 1
    for i in range(1, n+1):
        result *= x
    return result
```

Algoritmen gör *n* multiplikationer så tiden växer linjärt med *n* oberoende av *x.*

UPPSALA UNIVERSITET

Beräkna x^n *rekursivt*

```
def power(x, n):
    if n == 0:
        return 1
    else:
        return x*power(x, n-1)
```

Om vi låter $t(n)$ stå för tiden anropet `power(x, n)` tar så gäller

$$t(n) = \begin{cases} c & \text{om } n = 0 \\ d + t(n-1) & \text{om } n > 0 \end{cases}$$

$$t(n) = d + t(n-1) = d + d + t(n-2) = d + d + d + t(n-3) = \ \dots$$

$$\dots = d \cdot n + t(0) = d \cdot n + c$$

Effektivare beräkning av x^n

Antag att vi vill beräkna x^{16}.

Om vi börjar med att beräkna x^8 så räcker en kvadrering för att få x^{16}

och x^8 kan beräknas med att kvadrera x^4

och x^4 kan beräknas med att kvadrera x^2

och x^2 kan beräknas med en multiplikation.

Alltså, x^{16} kan beräkna med 4 multiplikationer i stället för 16.

Hur kan man formulera denna idé till en generell algoritm?

Effektivare beräkning av x^n

Antag att *n* är jämnt och ≥ 0:

$$x^n = \begin{cases} 1 & \text{om } n = 0 \\ (x^{n/2})^2 & \text{om } n > 0 \end{cases}$$

Om *n* är udda så är *n* -1 jämnt:

$$x^n = \begin{cases} 1 & \text{om } n = 0 \\ (x^{n/2})^2 & \text{om } n > 0,\ n \text{ jämnt} \\ x \cdot (x^{(n-1)/2})^2 & \text{om } n > 0,\ n \text{ udda} \end{cases}$$

Effektivare beräkning av x^n

```
def power(x, n):

    def sqr(x):
        return x*x

    if n<0:
        return 1./power(x, -n)
    elif n==0:
        return 1
    elif n%2==0:
        return sqr(power(x, n//2))
    else:
        return x*sqr(power(x, (n-1)//2))
```

$$\begin{cases} 1 \\ (x^{n/2})^2 \\ x \cdot (x^{(n-1)/2})^2 \end{cases}$$

Hur många multiplikationer gör denna algoritm?

Om *n* är en jämn 2-potens, dvs om $n = 2^k$?

Krävs $k = \log_2 n$ multiplikationer

Om *n* inte är en jämn 2-potens, dvs om $n = 2^k$?

Krävs högst en extra multiplikation innan problemet halveras
dvs högst $2\log_2 n$ multiplikationer.

Ungefär hur många multiplikationer krävs för att beräkna
x^{1000} respektive $x^{1000000}$?

Sökning i en lista

Att söka efter ett värde i en Python-lista görs normalt med operatorn in t. ex. med uttrycket `if x in lst`

Även om vi inte ser det så måste det bakom scenen finnas kod liknande den här:

```
def search(x, lst):
    for e in lst:
        if e == x:
            return True
    return False
```

Arbetet är således, åtminstone i värsta fall, proportionellt mot listans längd.

Operatorn `in` och funktionen `search` har samma *komplexitet* även om operatorn in säkert är mycket snabbare.

Effektivare sökning

Om data i listan är sorterade i storleksordning kan sökningen göras väsentligt effektivare:

om x < värdet i mitten:
sök i vänster halva
annars:
sök i höger halva

Metoden som kallas binär sökning är enkel i implementera både rekursivt och iterativt.

Komplexitet för binär sökning

Metoden halverar sökmängden för varje iteration eller rekursivt anrop.

$$n \longrightarrow \frac{n}{2} \longrightarrow \frac{n}{2^2} \longrightarrow \frac{n}{2^3} \longrightarrow \dots \longrightarrow \frac{n}{2^k}$$

När är $\frac{n}{2^k} = 1$?

$$n = 2^k \Leftrightarrow k = \log_2 n$$

Hur många iterationer/funktionsanrop behövs för att söka i en lista med $10^6, 10^9$ och 10^{12} element?

Algoritm- eller komplexitetsanalys

Studerar hur tiden för en algoritm beror på indata.

Resultat av typen:

tiden växer proportionellt mot kvadraten på antalet element

eller

tiden är konstant oberoende av input

eller

tiden växer exponentiellt med problemstorleken

The end