Datastrukturer, Algoritmer och Design Patterns

Algoritmer

- att lista ut hur man löser ett problem är inte samma sak som att programmera lösningen
- en *algoritm* är en serie instruktioner som löser ett problem
- instruktionerna har inget med syntax att göra, en algoritm bryr sig inte om vilket språk man använder

Algoritmer ex.

- ta reda på hypotenusan på en rät triangel:
- pythagoras sats, vs. mäta med linjal
- ta sig till västerås från liljeholmen:
- tvärbana+tåg, vs tunnelbana+tåg
- bli rik:
- utveckla minecraft först, vs. råna en bank

Algoritmer ex. 2

- räkna ut 2x5:
- 2+2+2+2+2, vs multiplikationstabell
- ta reda på hur många minor som närligger cell:
- 1: gå igenom alla minor och räkna vilka som är intill
- 2: eller gå igenom alla närliggande celler och undersök vilka som har minor
- 2 kräver 2D-array

Algoritmer forts.

- olika lösningar kan vara *olika bra*, och bra på olika saker!
- därmed relevant när man vill *optimera*
- ex, antal operationer (tid), hur mycket minne som krävs, hur generaliserbar, hur lättläst
- de första tre beror på *storleken av inputen*
- ex, om man vill räkna antal siffror i en sträng, så tar det mera tid ju längre strängen

Algoritmer forts. 2

- hur listar jag ut en algoritm för ett problem jag har?
- vi kan lära oss syntax, men det finns ingen algoritm för att ta reda på vilken algoritm som löser ett problem
- enda vi kan göra är att öva, känna igen, och vara smart

Rekursion

- (tidigare nämnt som extrauppgift)
- rekursion är ett sätt att upprepa kod utan *iterering*
- dvs. använder sig inte av for, while, eller do-while
- utförs genom att en funktion *anropar sig själv*, tills något anrop såsmåningom bestämmer att den inte längre ska det

Rekursion ex.

```
int fakultet(int n){
    if (n == 0)
        return 1;
    return n * fakultet(n-1);
}
```

Rekursion 2

- alla rekursiva funktioner har *alltid*:
- (åtminstone) ett basfall, och något rekursivt anrop
- basfallet är vad som händer i *det enklaste fallet*
- det rekursiva anropet anropar funktionen med *en mindre del av inputten*
- detta garanterar att basfallet såsmåningom nås och att anropen slutar

Stockholms Tekniska Institut

Rekursion 3

- notera att det första anropet körs först, men avslutas sist. den måste få tillbaka ett värde från sitt rekursiva anrop innan den själv kan returnera
- basfallet *anropas* därmed sist, men returnerar först! (till funktionen som ropade på basfallet)
- rekursion bygger därmed en kedja av funktionsanrop. alla måste läggas på stacken i programmet innan de kan börja utvärderas
- (stacken är vart funktionsanrop och liknande tar upp minne i datorn)

Rekursion 4

- går att göra funktioner som gör mer än ett rekursivt anrop!
- ett typexempel på detta är en (naiv) rekursiv implementation som räknar ut ett steg av fibonaccisekvensen
- dvs. den returnerar fib(n-1) + fib(n-2)

Rekursion 5

- går att göra funktioner med mer än en rekursiv parameter!
- ett typexempel på detta är en (bra) rekursiv implementation som räknar ut ett steg av fibonaccisekvensen
- dvs. parameterlistan är något i stil med (int a, int b)
- sådana funktioner har oftast en *wrapper*, som anropas vanligt (dvs. ex. fib(5)), som endast anropar den rekursiva delen med två argument

Rekursion tips

- rekursion kan se knåpigt och svårt ut. det är det inte!
- skriv returtypen och alla parametrar först
- skriv sedan basfallet
- försök sedan lista ut det *näst* enklaste argumentet funktionen kan ta och lista ut hur det tillsammans med basfallet ska ge rätt svar

Rekursion 6

- oftast är iteration bättre, snabbare, och tar mindre plats
- rekursion skapar mer overhead, då det tar tid att göra själva anropet
- varför då använda rekursion?
- en del problem är mycket lättare att lösa med rekursion (ja, faktiskt)
- ofta kort kod (och snygg!)

Matte

- ordspråk: programmerare behöver kunna fyra matteoperationer:
- modulus, %
- exponenter, ^
- fakultet, !
- logaritmer, log()
- (fakultet kommer inte användas mycket idag, men är vanligt när man håller på med kombinatorik)

repetition exponenter

- 2^5 = 2 x 2 x 2 x 2 x 2
- 5^2 = 5 x 5
- x^0 = 1
- vad är störst, 6 ^ 3 eller 3 ^ 6?

repetition logaritmer

- logaritmer är inverser till exponenter
- dvs. de gör tvärtom
- log10(1000) = x *är ekvivalent till* 10^x = 1000
- (vilket tal ska 10 höjas upp till för att bli 1000)
- i datavetenskap blir det ofta bas 2

Komplexitet

- olika algoritmer kan vara olika snabba. kan vi prata om detta mera formellt?
- ja, med hjälp av “big-o notation”
- skrivs som O(n), (kan vara annat förutom n)
- uttalas “ordo n”
- grundidén är att det handlar om hur antalet operationer växer baserat på storleken av problemet
- kallas ibland tidskomplexitet

Komplexitet ex. 1

```
void foo1(int n){
    for(int i = 0; i<n; i++)
        printf("%d\n", i);
}
```

- hur många gånger körs printf() beroende av n?

Komplexitet ex. 1 svar

```
void foo1(int n){
    for(int i = 0; i<n; i++)
        printf("%d\n", i);
}
```

- den körs n gånger
- linjär tidskomplexitet
- O(n)

Komplexitet ex. 2

```
void foo2(int n){
    for(int i = 0; i<100; i++)
        printf("%d\n", i);
}
```

- hur många gånger körs printf() beroende av n?

Komplexitet ex. 2 svar

```
void foo2(int n){
    for(int i = 0; i<100; i++)
        printf("%d\n", i);
}
```

- antalet körningar är okopplat till variabeln n
- det är *alltid* lika många körningar
- konstant tidskomplexitet, det alltid tar lika lång tid
- O(1)

Komplexitet ex. 3

```
void foo3(int n){
    for(int i = 0; i<n; i++)
        for(int i2 = 0; i2<n; i2++)
            printf("%d\n", i);
}
```

- hur många gånger körs printf() beroende av n?

Komplexitet ex. 3 svar

```
void foo3(int n){
    for(int i = 0; i<n; i++)
        for(int i2 = 0; i2<n; i2++)
            printf("%d\n", i);
}
```

- n * n, n^2 gånger
- geometrisk(kvadratisk) tidskomplexitet
- O(n^2)

Komplexitet ex. 4

```
void foo4(int n){
    for(int i = 1; i<n; i*=2)
        printf("%d\n", i);
}
```

- hur många gånger körs printf() beroende av n?

Komplexitet ex. 4 svar

```
void foo4(int n){
    for(int i = 1; i<n; i*=2)
        printf("%d\n", i);
}
```

- variabeln i dubblas vid varje iteration
- för varje extra körning på loopen måste vi dubbla n
- antalet körningar = log2(n)
- O(log n)
- *inte* O(log2 n), O ser ingen skillnad på logaritmbas

Komplexitet ex. 5

```
void foo5(int n){
    if (n == 0)
        return 1;
    return foo5(n-1) + foo5(n-1);
}
```

- hur många gånger körs foo5()?

Komplexitet ex. 5 svar

```
void foo5(int n){
    if (n == 0)
        return 1;
    return foo5(n-1) + foo5(n-1);
}
```

- för varje körning körs två körningar
- varje gång n ökar med 1 dubblas antal körningar
- exponentiell tidskomplexitet
- O(2^n)

Komplexitet ex. 6

```
void foo6(int n){
    for(int i = 0; i<100; i++)
        printf("%d\n", i);
    for(int i = 0; i<n; i++)
        printf("%d\n", i);
}
```

- hur många gånger körs printf() beroende av n?
- vad är komplexitetsklassen på foo6()?

Komplexitet ex. 6 svar

```
void foo6(int n){
    for(int i = 0; i<100; i++)
        printf("%d\n", i);
    for(int i = 0; i<n; i++)
        printf("%d\n", i);
}
```

- O(n)
- O bryr sig om hur många gånger som körs vid stora n
- loopen med hundra körningar blir då irrelevant
- dvs, O(1) + O(n) = O(n). störst bestämmer!
- detta gäller generellt för kod som körs i *serie*

Komplexitet ex. 7

```
void foo7(int n){
    for(int i = 0; i<n; i++)
        printf("%d\n", i);
}
void foo8(int n){
    for(int i = 0; i<n; i++)
        foo7(n);
}
```

- hur många gånger körs printf() beroende av n, om vi kör foo8()?
- vad är komplexitetsklassen på foo7()?
- vad är komplexitetsklassen på foo8()?

Komplexitet ex. 7 forts

```
void foo7(int n){
    for(int i = 0; i<n; i++)
        printf("%d\n", i);
}
void foo8(int n){
    for(int i = 0; i<n; i++)
        foo7(n);
}
```

- O(n) * O(n) = O(n^2)
- generellt när man utför loopar i varandra multiplicerar vi komplixiterna av beståndsdelarna

Stockholms Tekniska Institut

Komplexitet forts.

- det går att undersöka tidskomplexiteten av olika saker för samma algoritm, t.ex:
- average
- worst case
- best case
- i ex. sortering, komplexiteten ifall elementen redan är sorterade
- ifall elementen redan är bakåtsorterade
- i frågor angående sant/falskt, kan algoritmen ha olika komplexitet baserat på resultatet

Vanliga komplexitetsklasser

- O(1) //konstant
- O(log n) //logaritmisk
- O(n) //linjär
- O(n log n)
- O(n^k) //för något k //geometrisk
- O(k^n) //för något k //exponentiell

- generellt kan olika tidskomplexiteter ses som att varje är ekvivalent med någon ekvation för en graf:
- y = kx + m <=> O(n)
- y = m <=> O(1)
- y = kx^2 <=> O(n^2)
- y = k logx + m <=> O(log n)