Föreläsning 7

Pseudokod
Analys av algoritmer
Rekursiva algoritmer

För att beskriva algoritmer kommer vi använda oss av en *pseudokod* (låtsas programspråk) definierad i kursboken Appendix C. Vi går igenom denna.

Det viktigaste med detta avsnitt är att vissa algoritmer i kap 8-9 kan bli tydligare beskrivna i pseudokod och att vi kommer beskriva deras beräkningstider.

Det ingår inte att kunna bevisa att algoritmer är korrekta.

Tilldelningsoperatorn
$'{:=}'$ Inte samma som vanligt $=$
(Kursboken skriver bara $=$)
Om $x = 3$ och $y = 7$ så kommer kommandot
$x := y$
ha effekten att x får värdet 7, men y fortfarande har värdet 7.

Vanligt $=$ skrivs i pseudokoden som $==$
Och 'inte lika med' skrivs som $\neg =$.
I övrigt används logiska symboler $\wedge$, $\vee$, och
Symboler för algebraiska operationer $+ - */$ och symboler för olikheter som vanligt
Kommentarer till koden skrivs efter //

if satser
if(villkor)
{steg}
Om villkor är uppfyllt utförs steget innanför klamrarna.

Exempel:
if$(x \geq 0)\{$
$x := x + 1$
$\}$

om invärdet är $x = 1$ får vi ut $x = 2$ om invärdet är $x = -1$ får vi ut $x = -1$.

if else satser
if(villkor)
{steg 1}
else
{steg 2}
Om villkoret är sant utförs steg 1, annars utförs steg 2.

Exempel:
if$(x \geq 0)\{$
$x := x + 1$
$\}$
else$\{$
$x := x - 1$
$\}$

Om invärdet är $x = 1$ så får vi utvärdet $x = 2$, om invärdet är $x = -1$ får vi utvärdet $x = -2$

while loop
while(villkor)
{steg}
Steget utförs så länge villkoret är uppfyllt. OBS! Det är viktigt att villkoret bara kan vara uppfyllt ett ändligt antal gånger.

Exempel: $s_1, \ldots s_n$ en följd av tal är inputvärden. En algoritm som bestämmer det minsta talet av dessa.
$small := s_1$
$i := 2$
while$(i \leq n)\{$
  if$(s_i < small)\{$
    $small := s_i\}$ // small är alltid minsta värdet av $s_1 \ldots s_i$.
  $i := i + 1$

}

outputvärdet är *small* som kommer vara det minsta värdet av talen när algoritmen är klar.

En **while** loop som i exemplet ovan, där variabeln i får vara alla heltal från 2 till n kan också skrivas som en **for** loop

$small := s_1$
for($i = 2$ to n){
  if($s_i < small$){
    $small := s_i$}
}

Om exvis koden ovan är en del av en längre kod kan vi skriva den som en funktion, med inputvärden s, n (s är följden $s_1, \ldots, s_n$ och n antalet element i följden) och outputvärde *small*.

Då kan vi som en del av en större kod skriva exvis $x := min(s, n)$

$min(s, n)$\{
$small := s_1$
  for($i = 2$ to n){
    if($s_i < small$){
      $small := s_i$}
   return *small* // return ger outputvärden
  }
}

Sökalgoritm

Ger exempel på en sökalgoritm som letar rätt på första tillfället av mönstret (ordet) $p = p_1 \ldots p_m$ av längd m i texten $t = t_1 \ldots t_n$ av längd n.

Invärden är p, m, t, n utvärde är i där i är indexet för 'första bokstaven' i p i den första förekomsten av p i texten. Om p inte förekommer i texten ges i värdet 0.

$textsearch(p, m, t, n)$\{
  for($i = 1$ *to* $n - m + 1$){ // Om p finns i t börjar den senast på index $n - m + 1$.

```
    j = 1
    while(t_{i+j-1} == p_j){ //Jämför om bokstävernalika på aktuellt index
      j := j + 1 // fortsätter genom hela ordet
      if(j > m) // om j = m + 1 har vi hittat hela p
        return i
  }
}
return 0
}
```

Antalet steg som krävs för att utföra en algoritm är mycket viktigt.

Om invärdena är en mängd med n element, där n kan variera så beror antalet steg (oftast) av n. Om antalet steg växer mycket snabbt när n växer kan algoritmen bli obrukbar även för små värden på n.

worst-case tiden för en algoritm är det största antalet steg som kan krävas för n st invärden.

best-case tiden för en algoritm är det minsta antalet steg som kan krävas för n st invärden

average-case tiden för en algoritm är det genomsnittliga antalet steg som krävs för n st invärden

När beräkningstider för algoritmer beskrivs så är man mest intresserad av storleksordningen, det exakta värdet är inte så viktigt.

$f(n) = O(g(n))$ 'f är av ordning högst g'
om det finns konstant $C_1 > 0$ så att
$|f(n)| \leq C_1|g(n)|$ för alla utom ett ändligt antal n.

$f(n) = \Omega(g(n))$ 'f är av ordning lägst g'
om det finns konstant $C_2 > 0$ så att
$|f(n)| \geq C_2|g(n)|$ för alla utom ett ändligt antal n.

$f(n) = \Theta(g(n))$ 'f är av ordning g'
om $f = O(g(n))$ och $f = \Omega(g(n))$ dvs
$C_2|g(n)| \leq |f(n)| \leq C_1|g(n)|$ för alla utom ett ändligt antal n.

Sats Ett polynom $p(n) = a_k n^k + \ldots a_1 x + a_0$ växer som n^k.
dvs $p(n) = \Theta(n^k)$
Några vanliga typer av tillväxt, i storleksordning
$\Theta(\ln n)$, $\Theta(n^k)$, $\Theta(c^n)$, $\Theta(n!)$

Om $f(n) = \Theta(n^k)$ för något heltal $k \geq 1$ sägs f ha polynomiell tillväxt.

Polynomiell tillväxt är bra för en algoritm, ($\Theta(\ln n)$ naturligtvis ännu bättre). En algoritm där worst-case time har polynomiell tillväxt (eller bättre) kallas effektiv.

Om $f(n) = \Theta(c^n)$ för något $c > 1$ sägs f ha exponentiell tillväxt.

Exponentiell tillväxt är inte bra för en algoritm, $\Theta(n!)$ ännu värre. Även för relativt små värden på n krävs ett alldeles för stort antal beräkningar.

Exempel på bestämning av beräkningstider ges på lektion

Rekursiva algoritmer

En rekursiv algoritm är en algoritm som innehåller en funktion (i pseudokodsmening) som anropar sig själv.

Exempel: Algoritm för att beräkna $n!$. Använder att $n! = n * (n-1)!$.

```
factorial(n){
  if(n == 0){
     return 1}
   return n * factorial(n - 1)
}
```

Exempel (som ger upphov till viktig följd):

En robot kan gå steg av 1 meters resp 2 meters längd. På hur många sätt kan roboten gå n meter. Kallar detta antal $walk(n)$

Om $n = 1$ kan det göras på ett sätt, om $n = 2$ kan det göras på två sätt (1+1 eller 2).

Om $n > 2$ kan roboten antingen avsluta med ett två meters kliv, ger $walk(n-2)$ möjligheter eller med ett en meters kliv. Ger $walk(n-1)$ möjligheter. Totalt är $walk(n) = walk(n-1) + walk(n-2)$.

Vi kan skriva en rekursiv algoritm som beräknar $walk(n)$.

Invärde n
$walk(n)\{$
  if$(n == 1 \vee n == 2)\{$
    return $n\}$
    return $walk(n-1) + walk(n-2)$
$\}$

Definition Fibonacci följden $\{f_n\}_{n=1}^{\infty}$ definieras genom
$f_1 = 1$
$f_2 = 1$
$f_n = f_{n-1} + f_{n-2}$, $n > 2$
Mycket viktig följd som dyker upp i ett förbluffande stort antal sammanhang.

Eftersom $walk(1) = f_2$ och $walk(2) = f_3$ och
$walk(n) = walk(n-1) + walk(n-2)$
så är $walk(n) = f_{n+1}$.