Föreläsning 15

Rekursion

TDDD86: DALP

Utskriftsversion av föreläsning i *Datastrukturer, algoritmer och programmeringsparadigm*
2 november 2015

Tommy Färnqvist, IDA, Linköpings universitet

15.1

Innehåll

Innehåll

15.2

1 Introduktion

Rekursion

- **rekursion**: Definition av operation i termer av sig själv
  - Att lösa ett problem med rekursion beror på att lösa mindre instanser av samma problem

- **rekursiv programmering**: Skriva funktioner som anropar sig själva för att lösa ett problem rekursivt
  - Ett lika kraftfullt substitut för *iteration* (loopar)
  - Lämpar sig särskilt väl för att lösa vissa typer av problem

15.3

Varför lära sig rekursion?

- **"kulturupplevelse"**: Ett annat sätt att tänka på problemlösning
- **kraftfullt**: kan lösa vissa typer av problem bättre än iteration
- Leder till elegant, simplistisk, kort kod (om använd på rätt sätt)
- Många (funktionella språk som Scheme, ML och Haskell) programmeringsspråk använder rekursion exklusivt (inga loopar)
- En nyckelkomponent i flera av de återstående labbarna i kursen

15.4

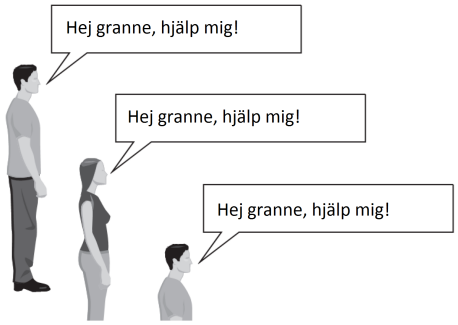
Övning

- (Till en student på främsta raden) Hur många studenter totalt sitter rakt bakom dig i din "kolumn" av föreläsningssalen?
  - Du har dålig syn så du kan bara se folk precis bredvid dig. Du kan alltså inte bara titta bakåt och räkna.
  - Men du får ställa frågor till personer nära dig.
  - Hur kan vi lösa det här problemet (rekursivt)

15.5

Idén

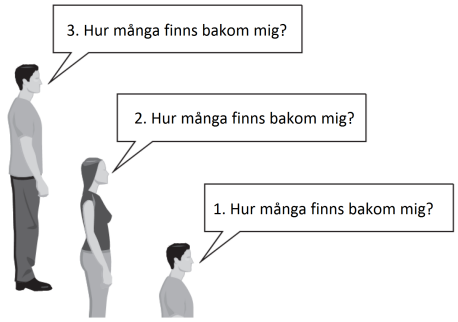
- Rekursion handlar om att bryta ned ett stort problem i mindre instanser av samma problem.
  - Varje person kan lösa en liten del av problemet.
    - Vad utgör en mindre version av problemet som skulle vara enklare att besvara?
    - Vilken information från en granne skulle kunna hjälpa mig?



15.6

Rekursiv algoritm

- Antalet människor bakom mig:
  - Om det finns någon bakom mig, fråga hen hur många människor som finns bakom hen.
    - När de svarar med ett värde N så svarar jag $N + 1$
  - Om det inte finns någon bakom mig svarar jag 0.



15.7

Rekursion och falluppdelning

- Varje rekursiv algoritm involverar minst 2 fall:
  - **basfall**: En enkel instans som kan lösas direkt.
  - **rekursivt fall**: En mer komplicerad instans av problemet som inte kan besvaras direkt, men som istället kan beskrivas i termer av mindre instanser av samma problem.
  - Vissa rekursiva algoritmer har mer än ett basfall eller rekursivt fall, men alla har minst ett av varje.
  - En nyckel till rekursiv programmering är att identifiera dessa fall.

15.8

En annan rekursiv uppgift

- Hur kan vi ta bort exakt hälften av alla M&M i en stor skål utan att hälla ut allihop och utan att kunna räkna dem?
  - Skulle det hjälpa om flera personer fick hjälpa till att lösa problemet? Kan varje person göra en liten del av arbetet?
  - Finns det något antal M&M som är lätt att fördubbla utan att kunna räkna?
    - (Vad är "basfallet"?)

2 Rekursion i C++

Rekursion i C++

- Betrakta följande funktion för att skriva ut en rad stjärnor

```
// Prints a line containing the given number of stars.
// Precondition: n >= 0
void printStars(int n) {
   for (int i = 0; i < n; i++) {
      cout << "*";
   }
   cout << endl; // end the line of output
}
```

- Skriv en rekursiv version av funktionen (som anropar sig själv).
  - Lös problemet utan att använda loopar.
  - Tips: Din lösning bör skriva ut endast en stjärna i taget.

Använda rekursion på rätt sätt

- Kondensera de rekursiva fallen till ett enda fall:

```
void printStars(int n) {
   if (n == 1) {
      // base case; just print one star
      cout << "*" << endl;
   } else {
      // recursive case; print one more star
      cout << "*";
      printStars(n - 1);
   }
}
```


"Rekursions-zen"

- Det riktiga, ännu enklare, basfallet är när n är 0, inte 1:

```
void printStars(int n) {
   if (n == 0) {
      // base case; just end the line of output
      cout << endl;
   } else {
      // recursive case; print one more star
      cout << "*";
      printStars(n - 1);
   }
}
```

Övning - printBinary

- Skriv en rekursiv funktion `printBinary` som tar in ett heltal och skriver ut det heltalets representation i bas 2 (binärt)
  - Exempel: `printBinary(7)` skriver ut `111`
  - Exempel: `printBinary(12)` skriver ut `1100`

plats	10	1
värde	**4**	**2**

32	16	8	4	2	1
1	**0**	**1**	**0**	**1**	**0**

  - Exempel: `printBinary(42)` skriver ut `101010`
  - Skriv funktionen rekursivt och utan loopar

15.13

Fallanalys

- Rekursion handlar om att lösa en liten del av ett stort problem.
  - Vad blir 69743 i bas 2?
    - Vet vi *någonting* om dess representation i bas 2?
  - Fallanalys:
    - Vad/vilka är enkla tal att skriva ut i bas 2?
    - Kan vi uttrycka ett större tal i termer av (flera) mindre tal?

15.14

Att hitta mönstret

- Antag att vi undersöker något godtyckligt heltal N.
  - Om representationen av N i bas 2 är
  - så är representationen av $(N/2)$
  - och representationen av $(N\%2)$ är
    - Vad kan vi dra för slutsats av det förhållandet?

10010101011
1001010101
1

15.15

Lösning - printBinary

```
// Prints the given integer's binary representation.
// Precondition: n >= 0
void printBinary(int n) {
   if (n < 2) {
      // base case; same as base 10
      cout << n;
   } else {
      // recursive case; break number apart
      printBinary(n / 2);
      printBinary(n % 2);
   }
}
```

15.16

Övning - reverseLines

- Skriv en rekursiv funktion `reverseLines` som tar in en filström och skriver ut raderna i filen i omvänd ordning.

Exempelindatafil:

```
Roses are red,
Violets are blue.
All my base
Are belong to you.
```

→

Förväntat utdata:

```
Are belong to you.
All my base
Violets are blue.
Roses are red,
```

  - Vilka fall har vi att beakta?
    - Hur kan vi lösa en liten del av problemet i taget?
    - Hur ser en fil ut som är lätt att reversera?

15.17

Pseudokod för reversering

- Reversera raderna i en fil:
  - Läs en rad R från filen.
  - Skriv ut resten av raderna i omvänd ordning.
  - Skriv ut R.
- Om vi bara hade ett sätt att reversera resten av raderna i filen...

15.18

Lösning - reverseLines

```
void reverseLines(ifstream& input) {
   string line;
   if (getline(input, line)) {
      // recursive case
      reverseLines(input);
      cout << line << endl;
   }
}
```

- Var är basfallet?

15.19

2.1 Implementation av rekursion

Kom ihåg: användning av stack – funktionsanrop

- Så implementerar kompilatorn funktioner
  - Funktionsanrop: *push:a* lokal omgivning och returnadress
  - Return: *pop:a* returnadress och lokal omgivning
  - Detta möjliggör rekursion.

15.20

2.2 Svansrekursion

Svansrekursion

Ett rekursivt anrop är *svansrekursivt* omm första instruktionen efter att kontrollflödet kommit tillbaka efter anropet är **return**.

- stacken behövs inte: allting på stacken kan kastas direkt
- svansrekursiva funktioner kan skrivas om till iterativa funktioner

Det rekursiva anropet i FACT är *inte* svansrekursivt:

function FACT(n)
  if $n = 0$ **then return** 1
  else return $n \cdot$FACT$(n-1)$

Första instruktionen efter retur från det rekursiva anropet är *multiplikation* $\Rightarrow n$ *måste behållas på stacken*

15.21

En svansrekursiv funktion

function BINSEARCH($v[a,\ldots,b],x$)
  if $a < b$ **then**
    $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$
    if $v[m].key < x$ **then**
      return BINSEARCH($v[m+1,\ldots,b],x$)
    else return BINSEARCH($v[a,\ldots,m],x$)
  if $v[a].key = x$ **then return** a
  else return 'not found'

De två rekursiva anropen är *svansrekursiva*. 15.22

Eliminering av svansrekursion

De två rekursiva anropen kan elimineras:

1: **function** BINSEARCH($v[a,\ldots,b],x$)
2: **if** $a < b$ **then**
3: $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$
4: **if** $v[m].key < x$ **then**
5: $a \leftarrow m+1$ {var: **return** BINSEARCH($v[m+1,\ldots,b],x$)}
6: **else** $b \leftarrow m$ {var: **return** BINSEARCH($v[a,\ldots,m],x$)}
7: **goto** *(2)*
8: **if** $v[a].key = x$ **then return** a
9: **else return 'not found'**

15.23

Svansrekursiv fakultetsfunktion

$fact$ kan skrivas om genom att introducera en hjälpfunktion:

function FACT(n)
  return FACT2($n,1$)

function FACT2(n,f)
  if $n = 0$ **then return** f
  else return FACT2($n-1, n \cdot f$)

FACT2 är *svansrekursivv* $\Rightarrow$ *minnesanvändningen efter rekursionseliminering blir* $O(1)$ 15.24

2.3 En till övning

Övning - pow

- Skriv en rekursiv funktion `pow` som tar in en heltalsbas och en exponent och returnerar basen upphöjt till exponenten.
  - Exempel: `pow(3, 4)` returnerar 81
  - Lös problemet rekursivt och utan att använda loopar

15.25

Lösning - pow

```
// Returns base ^ exponent.
// Precondition: exponent >= 0
int pow(int base, int exponent) {
   if (exponent == 0) {
      // base case; any number to 0th power is 1
      return 1;
   } else {
      // recursive case: x^y = x * x^(y-1)
      return base * pow(base, exponent - 1);
   }
}
```

15.26

En optimering?

- Notera följande matematiska egenskap:

$3^{12} = 531441 = 9^6$

$= (3^2)^6$

$531441 = (9^2)^3$

$= ((3^2)^2)^3$

  - När fungerar det här "tricket"?
  - Hur kan vi dra nytta av den här optimeringen i vår kod?
  - Varför bry sig om tricket när koden redan fungerar?

15.27

Lösning 2 - pow

```
// Returns base ^ exponent.
// Precondition: exponent >= 0
int pow(int base, int exponent) {
   if (exponent == 0) {
      // base case; any number to 0th power is 1
      return 1;
   } else if (exponent % 2 == 0) {
      // recursive case 1: x^y = (x^2)^(y/2)
      return pow(base * base, exponent / 2);
   } else {
      // recursive case 2: x^y = x * x^(y-1)
      return base * pow(base, exponent - 1);
   }
}
```

15.28

3 Algoritmanalys

Analys av algoritmer

Vad ska vi analysera?

- Korrekthet (inte i denna kurs)
- Terminering (inte i denna kurs)
- Effektivitet, resursförbrukning, komplexitet

Tidskomplexitet — hur lång tid tar algoritmen i värsta fallet?

- som funktion av vad?
- vad är ett tidssteg?

Minneskomplexitet — hur stort minne behöver algoritmen i värsta fallet?

- som funktion av vad?
- mätt i vad?
- tänk på att funktions- och proceduranrop också tar minne

15.29

Hur man kan jämföra effektivitet

- Studera exekveringstid (eller minnesåtgång) som en funktion av storleken på indata.
- När är två algoritmer "lika effektiva"?
- När är en algoritm bättre än en annan?

Jämförelse med några elementära funktioner

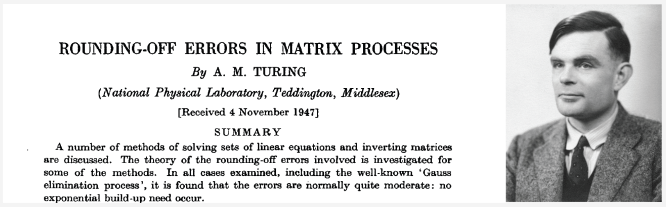
n	$\log_2 n$	n	$n \log_2 n$	n^2	2^n
2	1	2	2	4	4
16	4	16	64	256	$6.5 \cdot 10^4$
64	6	64	384	4096	$1.84 \cdot 10^{19}$

$1.84 \cdot 10^{19} \mu$sekunder $= 2.14 \cdot 10^8$ dagar $= 583.5$ årtusenden

15.30

Förenkla beräkningarna

> *" It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings. "* — Alan Turing



ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING

(*National Physical Laboratory, Teddington, Middlesex*)

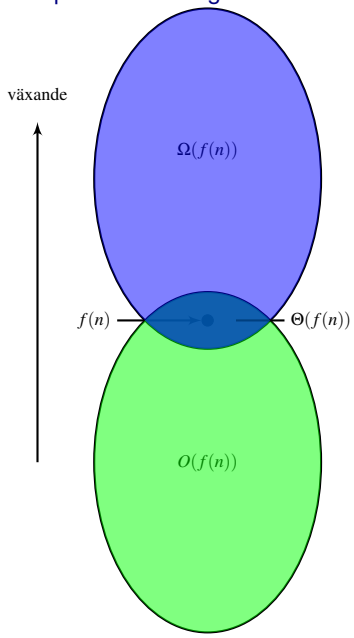
[Received 4 November 1947]

SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known 'Gauss elimination process', it is found that the errors are normally quite moderate: no exponential build-up need occur.

15.31

Hur komplexitet kan anges



- Hur ändras komplexiteten för växande storlek n på indata?
- Asymptotisk komplexitet — vad händer när n växer mot oändligheten?
- Mycket enklare om vi bortser från konstanta faktorer.

- $O(f(n))$ – växer högst lika snabbt som $f(n)$
- $\Omega(f(n))$ – växer minst lika snabbt som $f(n)$
- $\Theta(f(n))$ – växer lika snabbt som $f(n)$

15.32

Ordo-notation

f, g: växande funktioner från $\mathbb{N}$ till $\mathbb{R}^+$

- $f \in O(g)$ omm det existerar $c > 0, n_0 > 0$ sådana att $f(n) \leq c \cdot g(n)$ för alla $n \geq n_0$ Intuition: Bortsett från konstanta faktorer växer f inte snabbare än g
- $f \in \Omega(g)$ omm det existerar $c > 0, n_0 > 0$ sådana att $f(n) \geq c \cdot g(n)$ för alla $n \geq n_0$ Intuition: Bortsett från konstanta faktorer växer f minst lika fort som g
- $f(n) \in \Theta(g(n))$ omm $f(n) \in O(g(n))$ och $g(n) \in O(f(n))$ Intuition: Bortsett från konstanta faktorer växer f och g lika snabbt

NOTERA: Ω är motsatsen till O, dvs $f \in \Omega(g)$ omm $g \in O(f)$.

15.33

3.1 Rekursiva algoritmer

Exekveringstid för rekursiva algoritmer

- Karakterisera exekveringstiden med en rekurrensrelation
- Finn en lösning (på sluten form) till rekurrensrelationen
- Om du inte känner igen rekurrensrelationen kan du
  - "Rulla upp" relationen några gånger för att få fram en hypotes om en möjlig lösning: $T(n) = \ldots$
  - Bevisa hypotesen om $T(n)$ med matematisk induktion. Om det inte går bra, modifiera hypotesen och prova igen...

15.34

Exempel: Fakultetsfunktionen

function FACT(n)
  if $n = 0$ **then return** 1
  else return $n \cdot$FACT($n-1$)

Exekveringstid:

- tid för jämförelse: t_c
- tid multiplikation: t_m
- tid för anrop och return försummas

Total exekverings tid $T(n)$. $T(0) = t_c$ $T(n) = t_c + t_m + T(n-1)$, om $n > 1$ Alltså, för $n > 0$:

$$T(n) = (t_c + t_m) + (t_c + t_m) + T(n-2) =$$

$$= (t_c + t_m) + (t_c + t_m) + (t_c + t_m) + T(n-3) = \ldots =$$

$$= \underbrace{(t_c + t_m) + \ldots + (t_c + t_m)}_{n \text{ ggr}} + t_c = n \cdot (t_c + t_m) + t_c \in O(n)$$

15.35

Exempel: Binärsökning

function BINSEARCH($v[a,\ldots,b], x$)
  if $a < b$ **then**
    $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$
    if $v[m].key < x$ **then**
      return BINSEARCH($v[m+1,\ldots,b], x$)
    else return BINSEARCH($v[a,\ldots,m], x$)
  if $v[a].key = x$ **then return** a
  else return 'not found'

Låt $T(n)$ vara tiden, i värsta fall, att söka bland n tal med BINSEARCH.

$$T(n) = \begin{cases} \Theta(1) \text{ om } n = 1 \\ T\left(\lceil \frac{n}{2} \rceil\right) + \Theta(1) \text{ om } n > 1 \end{cases}$$

Om $n = 2^m$ får vi

$$T(n) = \begin{cases} \Theta(1) \text{ om } n = 1 \\ T\left(\frac{n}{2}\right) + \Theta(1) \text{ om } n > 1 \end{cases}$$

Vi kan då sluta oss till att $T(n) = \Theta(\log n)$.

15.36

Mästarmetoden

Sats 1 ("Master theorem")**.** Om $a \geq 1, b > 1, d > 0$ så har rekurrensrelationen

$$\begin{cases} T(n) = aT\left(\frac{n}{b}\right) + f(n) \\ T(1) = d \end{cases}$$

den asymptotiska lösningen

- $T(n) = \Theta(n^{\log_b a})$ om $f(n) = O(n^{\log_b a - \varepsilon})$ för något $\varepsilon > 0$
- $T(n) = \Theta(n^{\log_b a} \log n)$ om $f(n) = \Theta(n^{\log_b a})$
- $T(n) = \Theta(f(n))$ om $f(n) = \Omega(n^{\log_b a + \varepsilon})$ för något $\varepsilon > 0$ och $af\left(\frac{n}{b}\right) \leq c \cdot f(n)$ för någon konstant $c < 1$ för alla tillräckligt stora n.

15.37

3.2 Vanliga tillväxttakter

Vanliga tillväxttakter

tillväxttakt	typisk kod	beskrivning	exempel	$T(2n)/T(n)$
1	`a = b + c`	sats	lägga ihop två tal	1
$\log_2 n$	`while (n > 1) { n = n / 2; ...}`	dela på hälften	binärsökning	≈ 1
n	`for (int i = 0; i < n, i++) { ... }`	loop	hitta maximum	2
$n \log_2 n$	se föreläsning om mergesort	divide and conquer	mergesort	≈ 2
n^2	`for (int i = 0; i < n, i++) for (int j = 0; j < n, j++) { ... }`	dubbel loop	kolla alla par	4
n^3	`for (int i = 0; i < n, i++) for (int j = 0; j < n, j++) for (int k = 0; k < n, k++) { ... }`	trippel-loop	kolla alla tripplar	8
2^n	se nästa föreläsning	total-sökning	kolla alla delmängder	$T(n)$

15.38