F10 - rekursion och sortering

Programmeringsteknik med C och Matlab, 7,5 hp

Niclas Börlin
niclas.borlin@cs.umu.se

Datavetenskap, Umeå universitet

2023-10-13 Fre

Rekursion

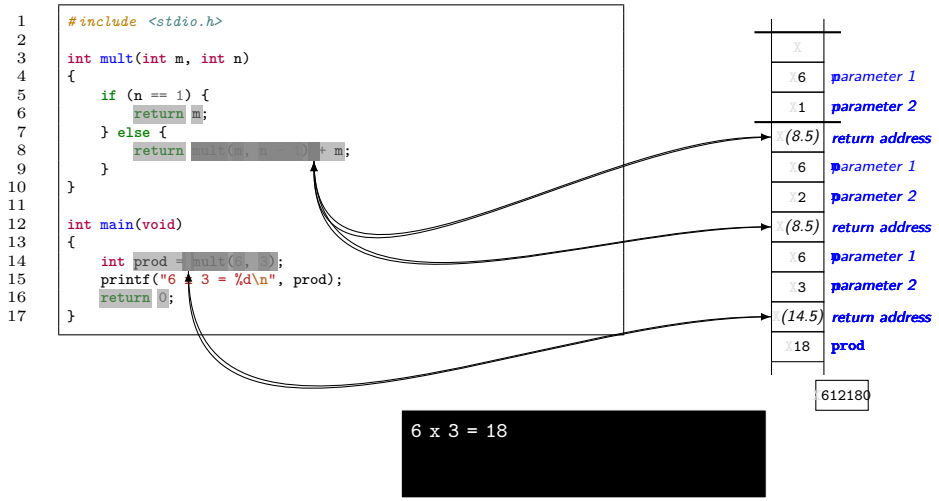
Rekursion

- Det finns inget som hindrar att en funktion anropar sig själv
  - Detta kallas rekursion
- För att rekursionen skall terminera (avslutas), måste det
  1. finnas ett eller flera stoppvillkor (basfall) och
  2. varje rekursivt anrop måste ta oss minst ett steg närmare ett stoppvillkor

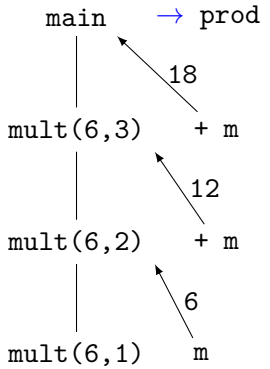
Ett exempel (2)

- En multiplikation går att se som en sekvens av additioner
  - $m \cdot n = \underbrace{m + m + \cdots + m}_{n}$
- En rekursiv algoritm `mult(m, n)` skulle kunna se ut så här:
  1. Om `n = 1`
     1.1 Returnera `m`
  2. annars
     2.1 Returnera `mult(m, n - 1) + m`
- Basfallet är `n = 1`
- I det rekursiva fallet anropar vi `mult` med värdena `m` och `n - 1` (och adderar sedan `m`)
  - Vi kommer ett steg närmare basfallet

mult, körning



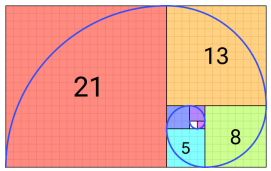
Anropsträd



Ett exempel till

- Fibonacci-sekvensen är definierad som en rekursiv sekvens:
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$
- Sekvensen blir
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

  och den dyker upp på många ställen i naturen[1]



[1] https://en.wikipedia.org/wiki/Fibonacci_number

Fibonacci-sekevensen

- Följande kod beräknar det n:te talet i Fibonacci-sekvensen:

code/fib.c

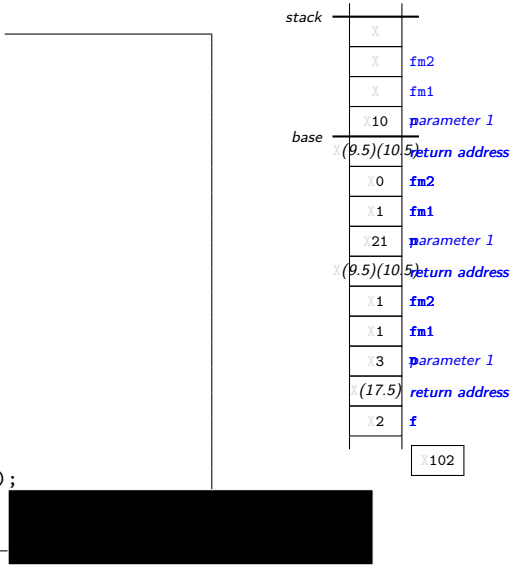
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```

- Basfallen är `n = 0` och `n = 1`
- I de rekursiva fallen anropar vi `fib` med värdena `n-1` och `n-2`
  - Vi kommer minst ett steg närmare basfallen

fib, körning

code/fib.c

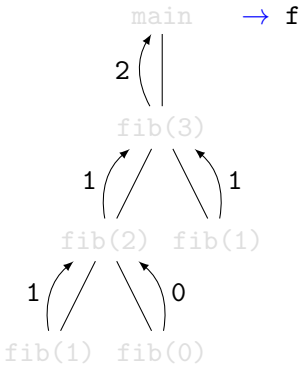
```
1  #include <stdio.h>
2
3  int fib(int n)
4  {
5      int fm1, fm2;
6      if (n < 2) {
7          return n;
8      } else {
9          fm1 = fib(n - 1);
10         fm2 = fib(n - 2);
11         return fm1 + fm2;
12     }
13 }
14
15 int main(void)
16 {
17     int f = fib(3);
18     printf("fib(3) = %d\n", f);
19     return 0;
20 }
```



fib, anropsträd

code/fib.c

```
1  #include <stdio.h>
2
3  int fib(int n)
4  {
5      int fm1, fm2;
6      if (n < 2) {
7          return n;
8      } else {
9          fm1 = fib(n - 1);
10         fm2 = fib(n - 2);
11         return fm1 + fm2;
12     }
13 }
14
15 int main(void)
16 {
17     int f = fib(3);
18     printf("fib(3) = %d\n", f);
19     return 0;
20 }
```



När är det lämpligt med rekursion?

- Rekursion är ofta en bra lösning om följande villkor är uppfyllda:
  1. Ett eller flera enkla fall har en enkel, icke-rekursiv lösning
     - Ex: tomt fält
  2. Mer komplicerade fall går att definiera i termer av fall som ligger närmare de enklaste fallen
     - Ex: ta hand om första halvan av fältet och sedan andra halvan av fältet
- Genom att tillämpa denna omdefiniering varje gång funktionen anropas reduceras problemet till slut helt och hållet till de enklaste fallen

Rekursion vs iteration

- Fördelar med rekursion är att
  - Det blir ofta eleganta, kompakta kodlösningar
  - Det lämpar sig oerhört väl för problem som är rekursiva i sin natur, tex att söka information i ett träd
- Nackdelar är
  - Ineffektivitet: Vid varje funktionsanrop skall en massa data sparas undan, etc.
  - Olämpligt för applikationer som skall iterera för alltid

Vanliga fel vid rekursion

- Det vanligaste problemet är att något basfall inte nås
  - Endera så saknas basfall...
  - ...eller så tar inte det rekursiva fallet problemet närmare ett basfall

Binärsökning igen

- Algoritm med vanliga ord
  1. Jämför med elementet närmast mitten i sekvensen
     - 1.1 Om likhet — klart
     - 1.2 Om det sökta värdet kommer före elementet närmast mitten, sök i den vänstra delsekvensen, hoppa till steg 1
     - 1.3 Om det sökta värdet kommer efter elementet närmast mitten, sök i den högra delsekvensen, hoppa till steg 1
- Rekursiv!

Binärsökning - rekursiv

- Rekursiv algoritm:

```
1  // "Starter" function to be called from the outside
2  Algorithm binsearch(a: Array, n: Int, v: Value)
3    return binsearch_rec(a, 0, n - 1, v)
4
5  // Internal recursive function, not visible from the outside
6  Algorithm binsearch_rec(a: Array, left, right: Int, val: Value)
7    mid <- (left + right) / 2
8    if left > right then
9        return -1                              // Not found
10   else if val = a[mid] then
11       return mid                             // Found it
12   else if val < a[mid] then                 // Look left
13       return binsearch_rec(a, left, mid - 1, val)
14   else // val > a[mid]                      // Look right
15       return binsearch_rec(a, mid + 1, right, val)
```


Binärsökning - iterativ

- Iterativ algoritm igen:

```
1  Algorithm binsearch(a: Array, n: Int, v: Value)
2    left <- 0
3    right <- n - 1
4    while left <= right do
5      mid  <- (left + right) / 2   // Integer division
6      if v = a[mid] then
7        return mid                 // Found it
8      else if v < a[mid] then
9        right <-  mid - 1          // Look left
10     else
11       left <- mid + 1            // Look right
12
13   return -1 // Not found
```

Sortering

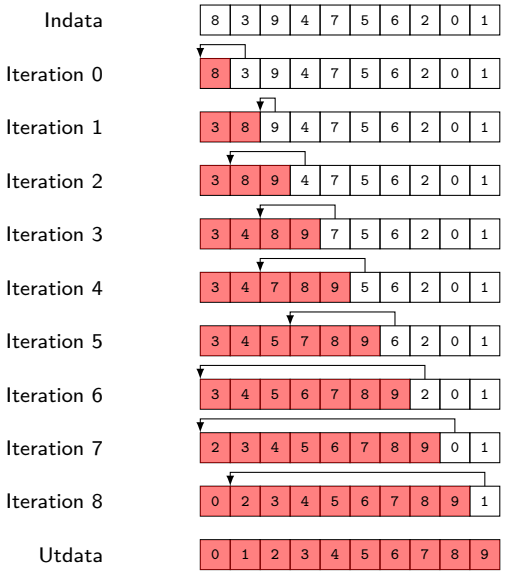
Sortering

- Varför ska man sortera?
  - Snabba upp andra algoritmer genom att vi vet mer
    - Sökning
    - Hantera stora datamängder
- Det finns flera olika algoritmer för sortering
- Vi kommer att titta på tre olika
  - Instickssortering — *Insertion Sort*
  - Bubbel-sortering — *Bubble Sort*
  - Samsortering — *Merge Sort*
- Syfte:
  - Förstå principerna, känna igen algoritmerna
  - Behöver inte kunna implementera

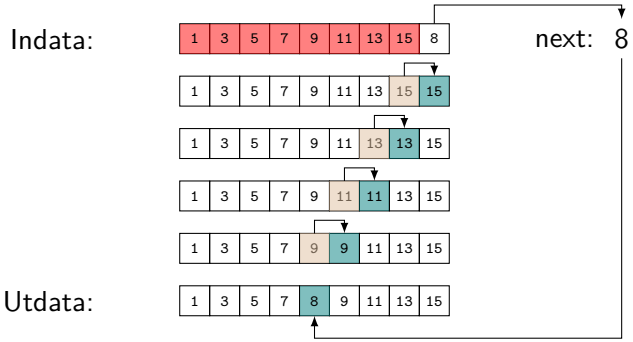
Insertion sort av fält

- Algoritmen i grova drag:
  - Börja med ett element (ett element är sorterat)
  - Ta sedan ett element i taget och sortera in på rätt plats bland de tidigare sorterade elementen

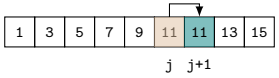
Insertion sort — exempel



Insertion sort — Sidospår: insättning (1)



Insertion sort — Sidospår: insättning (2)



Insertion sort — algoritm

- Algoritm:

```
1  Algorithm insertion_sort(a: Array, n: Int)
2    // i indicates first unsorted element in a
3
4    for i <- 1 to n - 1 do
5
6      // new value to insert in sorted part of a
7      next <- a[i]
8
9      // start with last sorted element
10     j <- i - 1
11
12     // as long as new element is smaller and
13     // we're inside the array
14     while j >= 0 and next < a[j] do
15
16       // shift element right
17       a[j + 1] <- a[j]
18
19       // continue to the left
20       j <- j - 1
21
22     // insert new value in its sorted place
23     a[j+1] <- next
24
25   return a
```


Bubble Sort

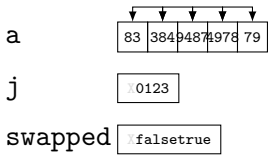
- Algoritmen i grova drag:
  - Upprepa följande tills ingen förändring sker:
    - Jämför alla elementen ett par i taget
      - Börja med element 0 och 1, därefter 1 och 2, osv
    - Om elementen är i fel ordning, byt plats på dem

Bubble Sort — algoritm

- Algoritm:

```
1  Algorithm bubble_sort(a: Array, n: Int)
2    do
3      // so far no swap has taken place
4      swapped <- false
5
6      // for each adjacent pair in a...
7      for j <- 0 to n - 2 do
8
9        // if the elements are in the wrong order...
10       if a[j] > a[j + 1] then
11
12         // ...swap the elements
13         tmp <- a[j]
14         a[j] <- a[j + 1]
15         a[j + 1] <- tmp
16
17         // remember that a swap has taken place
18         swapped <- true
19
20   while swapped = true
21
22   return a
```


Bubble Sort exempel



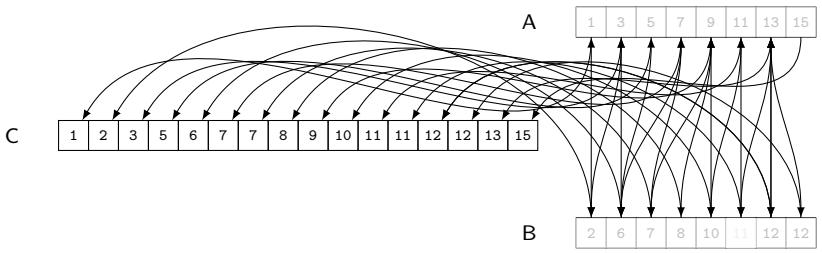
Merge Sort

- Algoritmen i grova drag
  - Om sekvensen har ett element
    - Returnera sekvensen (den är redan sorterad)
  - annars
    - Dela sekvensen i två ungefär lika stora delsekvenser
    - Sortera delsekvenserna rekursivt
    - Slå samman delsekvenserna (*Merge*)
    - Returnera den sammanslagna sekvensen

Merge

- *Merge Sort* använder en delalgoritm — *Merge*
- Algoritm för att slå samman två redan sorterade sekvenser:
  - Så länge bägge sekvenserna har element:
    - Jämför första (=minsta) elementet i vardera sekvensen
    - Flytta det minsta av de två elementen till utsekvensen
  - Flytta över alla element som finns kvar i sekvenserna

Merge — exempel



Merge

- Algoritm för *Merge*:

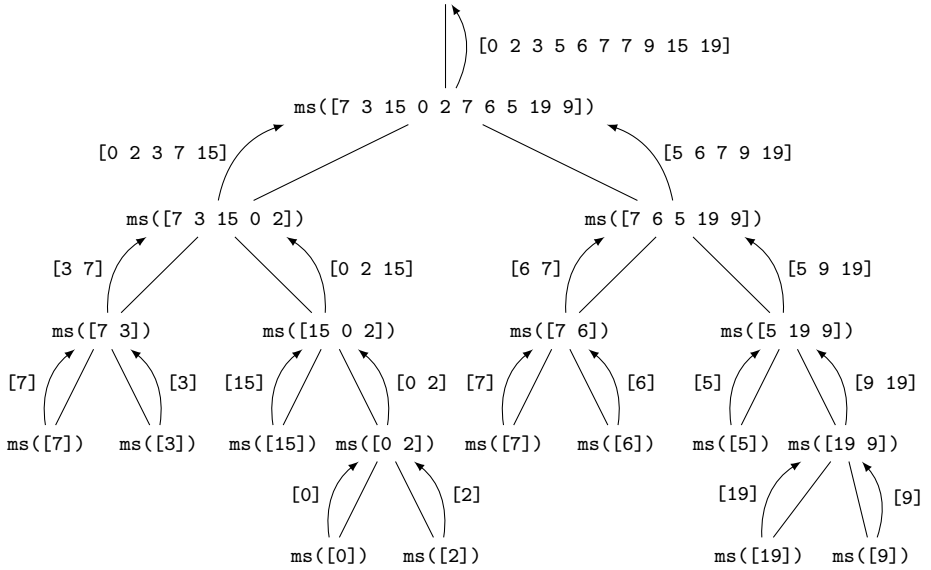
```
1  Algorithm merge(A, B: Array, na, nb: Int)
2      C <- create_array(na + nb)
3
4      ia <- 0 // Where to read from in A
5      ib <- 0 // Where to read from in B
6      ic <- 0 // Where to write to in C
7
8      // While there are elements in both A and B...
9      while ia < na and ib < nb do
10         if A[ia] <= B[ib] then    // Smallest in A...
11             C[ic] <- A[ia]        // ...copy from A
12             ia <- ia + 1          // ...advance in A
13         else                      // Smallest in B...
14             C[ic] <- B[ib]        // ...copy from B
15             ib <- ib + 1          // ...advance in B
16
17         ic <- ic + 1              // Advance in C
18
19     // While there are elements in A...
20     while ia < na do
21         C[ic] <- A[ia]            // ...copy from A
22         ia <- ia + 1              // ...advance in A and C
23         ic <- ic + 1
24
25     // While there are elements in B...
26     while ib < nb do
27         C[ic] <- B[ib]            // ...copy from B
28         ib <- ib + 1              // ...advance in B and C
29         ic <- ic + 1
30
31     return C
```


Merge Sort — algoritm

- Algoritm för *Merge Sort*:

```
1  Algorithm merge_sort(a: Array, n: Int)
2    if n < 2 then
3      // Already sorted
4      return a
5
6    // Split a in two parts
7    (left, right) <- split(a, n/2)
8
9    // Lengths of left and right parts, respectively
10   nl <- floor(n/2)
11   nr <- n - nl
12
13   // Sort left half recursively
14   left <- merge_sort(left, nl)
15
16   // Sort right half recursively
17   right <- merge_sort(right, nr)
18
19   // Merge sorted arrays
20   a <- merge(left, right, nl, nr)
21
22   return a
```


`merge sort`, anropsträd



Sortering, summering

- Varför olika algoritmer?
  - Olika effektivitet
  - Olika problemlösningsstrategier
    - *Insertion Sort* — instickssortering
    - *Bubbel Sort* — utbytessortering
    - *Merge Sort* — samsortering
- Det finns fler algoritmer än dessa
  - Mer på *Datastrukturer och Algoritmer*-kursen
  - Sök på "hungarian folk dance sorting" på youtube
    - https://www.youtube.com/watch?v=dENca26N6V4
    - https://www.youtube.com/watch?v=EdIKIf9mHk0&list=PLOmdoKois7_FK-ySGwHBkltzB11snW7KQ