

Programmeringsteknik med C och Matlab

Kapitel 10: Slutet är nära

Henrik Björklund

Umeå universitet

20 oktober 2009

En hel del studenter har missat att anmäla sig till tentan.

De som ändå vill skriva:

Skicka epost till `studentexp@cs.umu.se`

IDAG!

Gäller även dem som pratat med mig.

Summera en lista igen

C-kod

```
int intListSum(intList list){
    if(list == NULL){
        return 0;
    }else{
        return list->value + intListSum(list->next);
    }
}
```

Summera en lista igen

C-kod

```
int intListSum(intList list) {
    if(list == NULL) {
        return 0;
    } else {
        return list->value + intListSum(list->next);
    }
}
```

1. Vi kollar om vi är i **basfallet** (ett fall så enkelt att vi kan hantera det direkt).
2. Om inte: Dela upp problemet i
 - 2.1 en **liten del vi kan hantera direkt** (här: plocka fram värdet ur första elementet)
 - 2.2 en **större del som kan skötas rekursivt** (här: summera resten av listan).

Kombinera sedan ihop resultatet av de två delarna (här: summera de två resultaten).

När kan man ana att det är lämpligt att använda 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: tom lista.)
2. Mer komplicerade fall går att **definiera i termer av** fall som ligger närmare de enklaste fallen. (Ex. ta hand om första elementet och ta hand om resten av listan.)
3. Genom att tillämpa denna omdefiniering **varje gång** funktionen anropas reduceras problemet till slut **helt och hållet** till de enklaste fallen.

När kan man ana att det är lämpligt att använda 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: tom lista.)
2. Mer komplicerade fall går att **definiera i termer av** fall som ligger närmare de enklaste fallen. (Ex. ta hand om första elementet och ta hand om resten av listan.)
3. Genom att tillämpa denna omdefiniering **varje gång** funktionen anropas reduceras problemet till slut **helt och hållet** till de enklaste fallen.

Hmm, en liten aning abstrakt kanske. Vi tittar på ett till exempel.

Fibonacci strikes back

Repetition: Definition av Fibonaccitalen.

Fibonaccital n , för ett naturligt tal n , definieras som

$$f(n) = \begin{cases} 0 & \text{om } n = 0 \\ 1 & \text{om } n = 1 \\ f(n-1) + f(n-2) & \text{om } n \geq 2. \end{cases}$$

Fibonacci strikes back

Repetition: Definition av Fibonaccitalen.

Fibonaccital n , för ett naturligt tal n , definieras som

$$f(n) = \begin{cases} 0 & \text{om } n = 0 \\ 1 & \text{om } n = 1 \\ f(n-1) + f(n-2) & \text{om } n \geq 2. \end{cases}$$

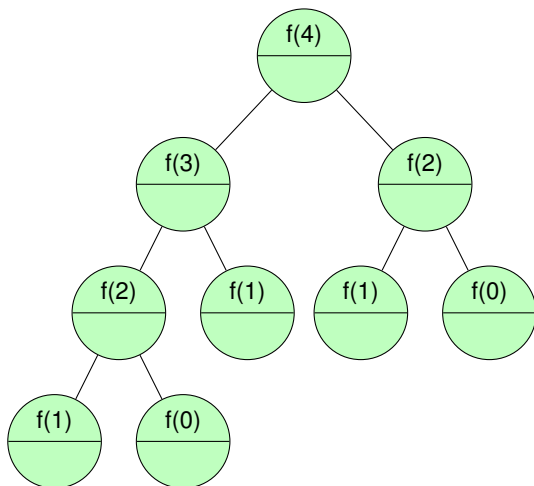
Detta är en **rekursiv definition** (funktionen f ingår i sin egen definition).

Den har alla egenskaper vi frågar efter:

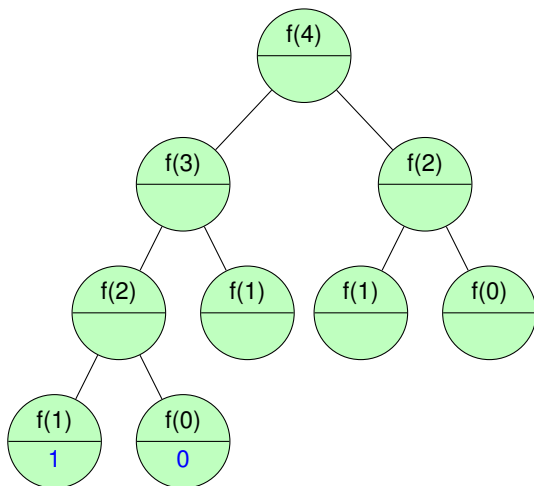
1. Två enkla fall ($n = 0$ och $n = 1$) som vi kan lösa direkt.
2. Övriga fall går att definiera **i termer av** fall som ligger närmare de enkla fallen.
3. Om vi hela tiden använder denna omdefinition kommer vi till slut bara att behöva lösa de enkla fallen och addera.

Vi tar en titt på hur vi skulle kunna räkna ut fibonaccital 4.

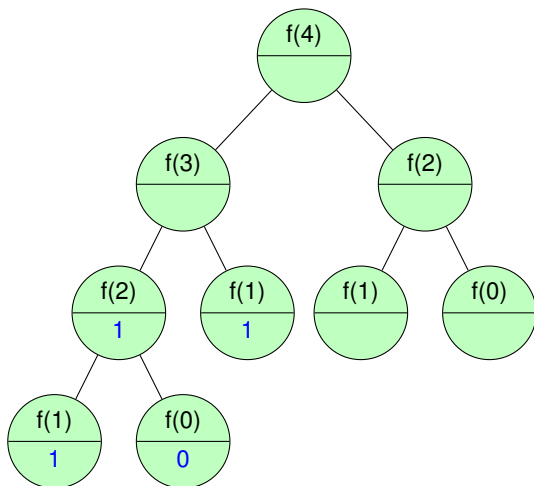
Fibonacci 4



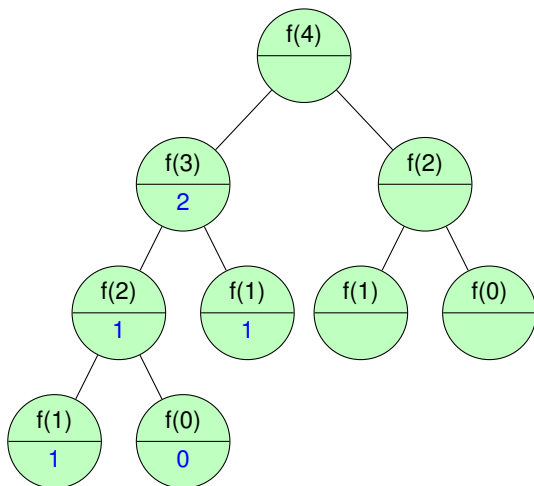
Fibonacci 4



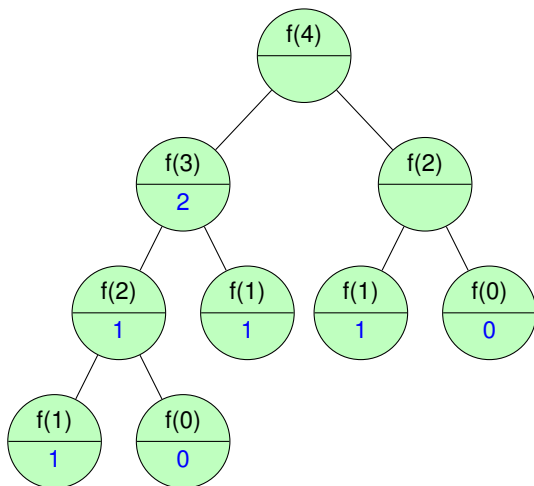
Fibonacci 4



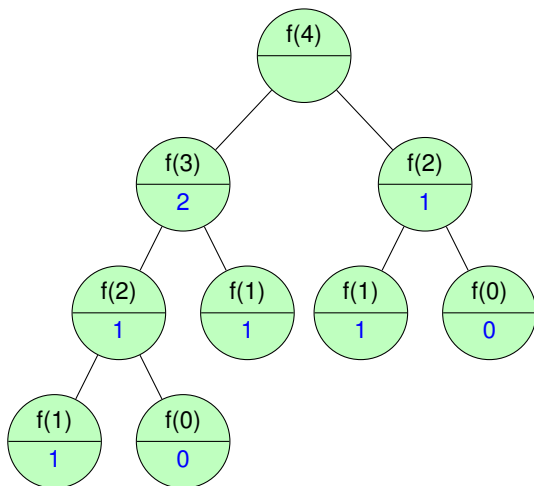
Fibonacci 4



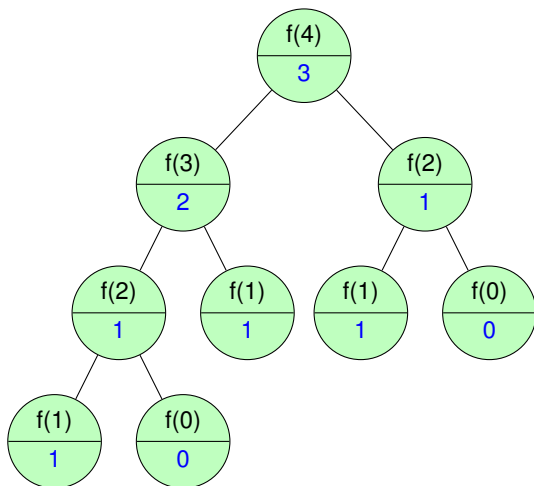
Fibonacci 4



Fibonacci 4



Fibonacci 4



En rekursiv fibonaccifunktion

Fibonaccitalen kan också beräknas rekursivt:

C-kod

```
int fibonacci(int n) {  
    if(n <= 1) return n;  
    return fibonacci(n-1) + fibonacci(n-2);  
}
```

En rekursiv fibonaccifunktion

Fibonaccitalen kan också beräknas rekursivt:

C-kod

```
int fibonacci(int n) {  
    if(n <= 1) return n;  
    return fibonacci(n-1) + fibonacci(n-2);  
}
```

Lägg märke till:

1. Eftersom funktionen är definierad för alla naturliga tal har vi **två basfall** (precis som i fibonaccitalens definition).
2. Extrem **kodekonomi**.
3. Inte särskilt effektivt i praktiken (många delproblem löses flera gånger).

Åter till listorna

Vi tittar på hur vi kan använda rekursion för att infoga nya element på rätt plats i en **sorterad** länkad lista.

För kodekonomin skull introducerar vi först en liten hjälpfunktion:

C-kod

```
intList intListCreateElement(int newValue) {
    intList newElement =
        (intList) malloc(sizeof(struct intListElement));
    newElement->value = newValue;
    newElement->next = NULL;
    return newElement;
}
```

Rekursivt infogande i en sorterad lista

C-kod

```
intList intListInsertSortedRec(intList oldList, int newValue){
    intList newElement;
    if(oldList == NULL){ /* Basfall 1 */
        return intListCreateElement(newValue);
    }else if(oldList->value >= newValue){ /* Basfall 2 */
        newElement = intListCreateElement(newValue);
        newElement->next = oldList;
        return newElement;
    }else{ /* Rekursivt fall */
        oldList->next =
            intListInsertSortedRec(oldList->next, newValue);
        return oldList;
    }
}
```

Flera returvärden från funktioner

Funktioner i C kan bara returnera **ett värde**. Värdet måste vara av funktionens **returtyp**.

Så vad gör vi om vi vill returnera flera värden från en funktion?

Det finns två alternativ:

1. Returnera en sammansatt datatyp (en struktur).
2. Förse funktionen med s.k. returparametrar.

Returnera en sammansatt datatyp

Antag att vi vill skriva en funktion som hittar det största och minsta talet i en `int`-array.

Funktionen ska alltså returnera två tal.

Det kan vi göra genom att returnera `en` sammansatt datatyp som innehåller `två` tal.

C-kod

```
typedef struct {  
    int min;  
    int max;  
} minMaxReturn;
```

En min-max-funktion

C-kod

```
minMaxReturn arrayMinMax(int * a, int length){
    minMaxReturn minMaxValues;
    int i;

    if(length <= 0){
        minMaxValues.min = 0;
        minMaxValues.max = 0;
        return minMaxValues;
    }
    minMaxValues.min = a[0];
    minMaxValues.max = a[0];
    for(i = 1 ; i < length ; i++){
        if(a[i] < minMaxValues.min) minMaxValues.min = a[i];
        if(a[i] > minMaxValues.max) minMaxValues.max = a[i];
    }
    return minMaxValues;
}
```

Att anropa vår min-max-funktion

C-kod

```
int main(void) {  
  
    int a [] = {1, -3, 5, 2, -5, 3, 7};  
    minMaxReturn minMaxValues = arrayMinMax(a, 7);  
    printf("Min: %d\tMax: %d\n", minMaxValues.min, minMaxValues.max);  
  
    return 0;  
}
```

Returparametrar

Om vi vill kan vi i stället förse funktionen med returparametrar.

1. Vi deklarerar variabler för returvärdena i **de funktioner som anropar vår funktion**.
2. Vi skickar med **pekare** till returvariablerna till funktionen.
3. Funktionen **sparar** returvärden i returvariablerna, så att den **anropande** funktionen kan läsa dem.

En ny min-max-funktion

C-kod

```
void arrayMinMax(int * a, int length, int * min, int * max) {
    int i;

    if(length <= 0) {
        *min = 0;
        *max = 0;
        return;
    }
    *min = a[0];
    *max = a[0];
    for(i = 1 ; i < length ; i++) {
        if(a[i] < *min) *min = a[i];
        if(a[i] > *max) *max = a[i];
    }
}
```

Att anropa vår nya funktion

C-kod

```
int main(void) {  
  
    int a [] = {1, -3, 5, 2, -5, 3, 7};  
    int min, max;  
    arrayMinMax(a, 7, &min, &max);  
    printf("Min: %d\tMax: %d\n", min, max);  
  
    return 0;  
}
```

På fredag

På fredag blir det **repetition**.

Ta med er

1. frågor och
2. åsikter om vad ni vill repetera.

Och igen: anmäl er till tentan via epost (om ni inte redan gjort det)!