32. Rekursion

Rekursive funktioner er uundværlige til bearbejdning af rekursive datastrukturer. Rekursive datastrukturer forekommer ofte - f.eks. både som lister og træer. Rekursiv problemløsning via del og hersk algoritmer repræsenterer en vigtig ide i faget. Derfor er det vigtigt for alle, som skal programmere, at kunne forholde sig til rekursion.

32.1. Rekursion

Lektion 8 - slide 2

Herunder udtrykker vi den essentielle indsigt som gør rekursion attraktiv. Vi indså allerede i afsnit 11.6 at del og hersk problemløsning er af stor betydning. I en nødeskal går del og hersk problemløsning ud på at opdele et stort problem i mindre problemer, løse disse mindre problemer, samt at kombinere løsningerne på de små problemer til en løsning af det oprindelige, store problem.

Anvendelse af del og hersk princippet involverer altså *problemopdelning* og *løsningskombination*.

> *Rekursion* er en problemløsningside, som involverer løsning af delproblemer af samme slags som det overordnede problem

Vi omgiver os med rekursive strukturer - både i vores hverdag og når vi arbejder på en computer. Blade i naturen udviser rekursive strukturer. Filer og kataloger på en computer udviser rekursive strukturer. Vi vil se på et antal endnu mere hyppige hverdagseksempler i afsnit 32.2 og afsnit 32.3.

Vi argumenter herunder for sammenhængnen mellem rekursive (data)strukturer og rekursive processer. Rekursive processer kan f.eks. programmeres med rekursive funktioner i C.

- Rekursive strukturer
  - I en rekursiv struktur kan man genfinde helheden i detaljen
  - I datalogisk sammenhæng arbejder vi ofte med rekursive datastrukturer
- Rekursive processer
  - Rekursive strukturer processeres naturligt rekursivt
  - Rekursive processer programmeres med rekursive procedurer eller funktioner

32.2. Hverdagsrekursion (1)

Lektion 8 - slide 3

I dette og næste afsnit ser vi på mere eller mindre naturlige rekursive fænomener som er kendt fra vores hverdag.

Man kan opnå et rekursivt billede hvis et spejl spejler sig et i et andet spejl

Tilsvarende rekursive billeder fåes hvis man filmer et TV med kamera, som samtidigt viser signalet fra kameraet

Hvis man er så heldig at have et badeværelse med to spejle på modsat rettede vægge kan man næsten ikke undgå at observere et rekursivt billede.

Hvis man ejer et videokamera og viser signalet på et fjernsyn, som samtidig filmes med kameraet giver dette også et rekursivt billede. Ganske som billedet vist i figur 32.1.

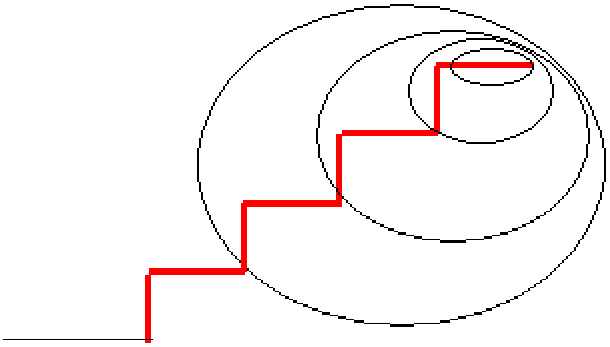
Figur 32.1 *Et billede af en del af en computerskærm, optaget med et web kamera, hvis billede samtidigt vises på skærmen. Det rekursive billede opstår fordi vinduet på skærmen filmes af kameraet, som viser det optagede billede som en del af vinduet, som hermed påvirker hvad kameraet filmer, som hermed påvirker billedet, osv. Efter nogle få niveuaer fortoner effekten sig i et meget forvrænget billede.*

32.3. Hverdagsrekursion (2)

Lektion 8 - slide 4

Når jeg går op af en trappe kan jeg vælge at tænke og gå rekursivt! Jeg tager et skridt, og går dernæst op af trappen som udgøres af de resterende trin foran mig. Jeg er med på at man nok skal være (dat)faglig idiot for at bestige trapper på denne måde. Men i princippet kunne det være en mulig - og langt op ad trappen - en fornuftig tankegang...

Det er muligt - men næppe naturligt - at opfatte en trappe som en rekursiv struktur



Figur 32.2 *En trappe opfattet som en rekursiv struktur.*

- En rekursiv trappe:
  - En trappe kan være 'tom' - flad gulv
  - En trappe kan bestå af et trin og en tilstødende trappe

I afsnit 41.1 ser vi på lineære lister, som i realiteten er ensdannede med trappen ovenfor. Mange centrale liste funktioner programmeres rekursivt, ud fra en forestilling om at en liste består af et hoved (svarende til et trin) og en hale (en resterende del af trappen) som selv er en liste (trappe).

32.4. Referencer

[-] ECIU materiale om rekursion - bedst i IE med SVG, Flash og Java plugins
http://www.cs.auc.dk/~normark/eciu-recursion/html/recit.html

33. Basal rekursion i C

I dette afsnit ser vi på det mere tekniske aspekt af rekursive funktioner i C. Dette er for øvrigt stort set det perspektiv som dyrkes i kapitel 11 af *C by Dissection*.

33.1. Basal rekursion (1)

Lektion 8 - slide 6

Programmerne i program 33.1 og program 33.2 er næsten funktionelt identiske. Begge gentager - på en forholdsvis besværlig måde - udskriften 'f: i', hvor i er et lille heltal.

I program 33.2 benytter vi en funktion, som kalder sig selv rekursivt. I program 33.1 har vi håndlavet et antal versioner af `f`, som kalder hinanden. I forhold til både program 33.1 og program 33.2 ville det være lettere at skrive en simpel for-løkke. Men programmerne illustrerer hvordan vi kan bruge rekursion til simple former for gentagelser i stedet for at bruge løkker.

En funktion i C er rekursiv hvis den i nogle programtilstande kalder sig selv direkte eller indirekte

```c
#include <stdio.h>

void f1(int i){
  printf("f1: %i\n", i);
}

void f2(int i){
  printf("f2: %i\n", i);
  f1(i-1);
}

void f3(int i){
  printf("f3: %i\n", i);
  f2(i-1);
}

int main(void) {
  printf("main\n");
  f3(3);
  return 0;
}
```

Program 33.1 *Funktionen* **`main`** *som kalder en funktion* **`f3`**, *som kalder* **`f2`**, *og som kalder* **`f1`**.

```c
#include <stdio.h>

void f(int i){
  if (i >= 1){
    printf("f: %i\n", i);
    f(i-1);
  }
}

int main(void) {
  printf("main\n");
  f(3);
  return 0;
}
```

Program 33.2 *En tilsvarende kæde af rekusive kald.*

For at sikre at programmet afsluttes, skal der eksistere et grundtilfælde (en programtilstand) hvor funktionen undlader at kalde sig selv rekursivt

Grundtilfældet i en rekursiv definition svarer til et delproblem, som ofte er så simpelt¨, at det kan løses umiddelbart uden yderlig problemopsplitning.

33.2. Basal rekursion (2)

Lektion 8 - slide 7

I dette afsnit vil vi i større detalje undersøge hvad der sker når en funktion kalder sig selv rekursivt. Vi vil se på stakken af aktiveringer (activation records), og vi vil i den forbindelse se på kommandoer som udføres før og efter de rekursive kald.

Vi viser her en variation af programmet, som afslører flere interessante aspekter af rekursion

I program 33.3 ser vi `main`, som kalder den rekursive funktion `f`. Kaldene af `f`, herunder det rekursive kald, svarer til de blå dele i programmet.

I funktionen `f` indlæses der et tegn med `scanf` inden det rekursive kald af `f`. Efter det rekursive kald af f udskrives det indlæste tegn.

```
#include <stdio.h>

void f(int i){
  char ch, skipch;
  if (i >= 1){
    printf("Enter a character: ");
    scanf("%c%c", &ch, &skipch);
    f(i-1);
    printf("We read: '%c'\n", ch);}
  else
    printf("No more recursive calls\n");
}

int main(void) {
  f(4);
  return 0;
}
```

Program 33.3 *Læsning på vej op ad bakken/stakken - Skrivning på vej ned.*

Herunder, i program 33.4 viser vi en dialog med program 33.3 hvor tegnene `'a'`, `'b'`, `'c'` og `'d'` indlæses i den lokale variabel `ch`. Hert nyt tegn indlæses i et separat rekursivt kald af f. Bemærk at der på denne måde kan sameksistere et antal `ch` variable - én pr. kald af `f`.

```
Enter a character: a
Enter a character: b
Enter a character: c
Enter a character: d
Enter a character: e
No more recursive calls
We read: 'e'
We read: 'd'
We read: 'c'
We read: 'b'
We read: 'a'
```

Program 33.4 *Input til og output fra programmet.*

Hvis du skifter til den slide, som svarer til dette afsnit, kan du følge udvikling af kaldstakken, som modsvarer kørslen af program 33.3.

Ved at følge udviklingen af kaldstakken skal du bide mærke i hvordan variablene `i` og `ch` optræder i alle kald af `f`. Man kan kun tilgå variable i den øverste ramme på stakken. Variable under toppen af stakken kan først tilgås når kaldene svarende til de øvre rammer er kørt til ende.

Læg også mærke til at tegnene `a`, `b`, `c` og `d`, som indlæses *på vej op ad stakken* udskrives i modsat rækkefølge *på vej ned ad stakken*.

34. Simple eksempler

I dette afsnit vil vi se på række simple eksempler på rekursion. Vi repeterer fakultetsfunktionen, rodsøgning og strengsammenligningen fra tidligere lektioner. Som nye eksempler vil vi se på en Fibonaccital funktion og en funktion til potensopløftning.

34.1. Eksempler fra tidligere lektioner

Lektion 8 - slide 9

Fakultetsfunktionen mødte vi i afsnit 14.1 i forbindelse med vores introduktion til rekursiv funktioner. I program 14.1 illustreres det hvordan den velkendte formel for fakultetsfunktionen (se teksten i afsnit 14.1) kan implementeres som en funktion skrevet i C.

Rodsøgningsfunktionen fra kapitel 13 er et andet godt og simpelt eksempel som appellerer til rekursiv tankegang. Den iterative løsning på rodsøgningsproblemet fra program 13.1 er vist som en tilsvarende rekursiv løsning i program 14.3. Personlig finder jeg at den rekursive løsning er mere elegant, og på et lidt højere niveau end den iterative løsning.

I program 34.1 viser vi en rekursiv løsning på strengsammenligningsproblemet, jf. afsnit 28.1, som har været en øvelse i en tidligere lektion af kurset. I en if-else kæde håndterer vi først en række randtilfælde samt situationerne hvor det første tegn i `s1` og `s2` er forskellige. I det sidste tilfælde kalder vi funktionen rekursivt (det røde sted). Vi håndterer her et delproblem, svarende til det oprindelige problem, på de dele af `s1` og `s2` som ikke omfatter de første tegn i strengene.

```
int mystrcmp(const char* s1, const char* s2){
  int result;

  if (*s1 == '\0' && *s2 == '\0')
    result = 0;
  else if (*s1 == '\0' && *s2 != '\0')
    result = -1;
  else if (*s1 != '\0' && *s2 == '\0')
    result = 1;
  else if (*s1 < *s2)
    result = -1;
  else if (*s1 > *s2)
    result = 1;
  else       /* (*s1 == *s2) */
    result = mystrcmp(s1+1, s2+1);

  return result;
}
```

Program 34.1 *Rekursiv udgave af* ***strcmp*** *- fra opgaveregningen i forrige lektion.*

I det følgende vil vi se på endnu flere eksempler som involverer rekursion.

34.2. Fibonacci tal (1)

Lektion 8 - slide 10

Der er et antal 'sikre hits', som næsten altid findes blandt eksemplerne, som bruges til at illustrere rekursive løsninger. Beregningen af Fibonacci tal er en af disse. Et Fibonaccital er summen af de to foregående Fibonacci tal. Starten kan i princippet være vilkårlig. I vore eksempler er fib(0) = 0 og fib(1) = 1.

Det er ikke svært at programmerer en funktion, som returnerer det n't Fibonacci tal. I program 34.1 viser vi en flot rekursiv udgave, som er en ganske direkte nedskrivning af definitionen givet herover.

```
long fib(long n){
  long result;

  if (n == 0)
    result = 0;
  else if (n == 1)
    result = 1;
  else
    result = fib(n-1) + fib(n-2);

  return result;
}
```

Program 34.2 *Funktionen fib der udregner det n'te Fibonaccital.*

Læg mærke til at hvert kald af `fib` forårsager to nye kald af `fib`. Denne observation er kritisk når vi skal forstå, hvorfor et kald af `fib`, f.eks. `fib(45)`, tager adskillige minutter at gennemføre.

På slide udgaven af dette materiale linker vi til en komplet program, som udskriver de 100 første Fibonaccital. Programmet er også her. Dette program vil aldrig køre til ende på grund af ovenstående observation. Køretiden af `fib` udvikler sig eksponentielt. I praksis betyder dette, at der er en øvre grænse for hvilke kald af `fib` der overhovedet kan beregnes med metoden i program 34.2.

Vi viser en let modificeret udgave af `fib` funktionen i program 34.3 herunder. I denne udgave tæller vi antallet af additioner, som udføres af et kald af `fib` samt alle de deraf afledte rekursive kald. Læg mærke til at tælleren `plus_count` er en global variabel (det røde sted). Variablen tælles op for hver addition (det blå sted), og den nulstilles for hvert nyt ydre kald af `fib` (det brune sted).

```
#include <stdio.h>

long plus_count = 0;

long fib(long n){
  long result;

  if (n == 0)
    result = 0;
  else if (n == 1)
    result = 1;
  else {
    result = fib(n-1) + fib(n-2);
    plus_count++;
  }

  return result;
}

int main(void) {
  long i, fib_res;

  for(i = 0; i < 42; i++){
    plus_count = 0;
    fib_res = fib(i);
    printf("Fib(%li) = %li (%li)\n", i, fib_res, plus_count  );
  }


  return 0;
}
```

Program 34.3 *En udgave af programmet som holder regnskab med antallet af additioner.*

Når vi kører program 34.3 ser vi at værdien af variablen `plus_count` vokser i samme takt som værdierne af `fib` funktionen anvendt på større og større inputværdier. Herunder viser vi et udsnit af output fra program 34.3.

```
Fib(0) = 0 (0)
Fib(1) = 1 (0)
Fib(2) = 1 (1)
Fib(3) = 2 (2)
Fib(4) = 3 (4)
Fib(5) = 5 (7)
Fib(6) = 8 (12)
Fib(7) = 13 (20)
Fib(8) = 21 (33)
...
Fib(36) = 14930352 (24157816)
Fib(37) = 24157817 (39088168)
Fib(38) = 39088169 (63245985)
Fib(39) = 63245986 (102334154)
Fib(40) = 102334155 (165580140)
Fib(41) = 165580141 (267914295)
```

Vi cutter listen på det sted, hvor udførelsen af `fib` bliver virkelig langsom. Vi ser og mærker tydeligt, at arbejdet i `fib` vokser eksponentielt i forhold til parameteren `n`.

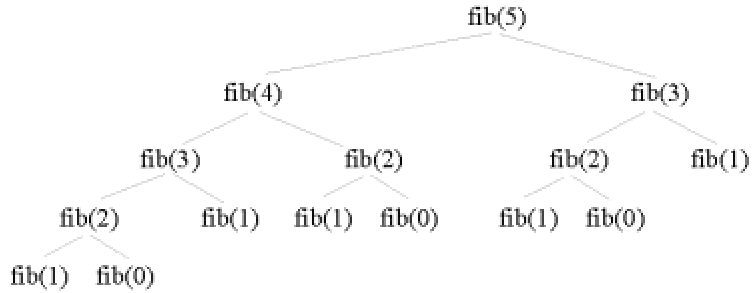
34.3. Fibonacci tal (2)

Lektion 8 - slide 11

Vi vil nu i yderligere detalje forstå hvorfor `fib` er så langsom. I figur 34.1 ser vi kaldene af `fib(5)` . Hvert kald af `fib` resulterer i to nye kald, på nær kald af `fib(0)` og `fib(1)`. Antallet af knuder et træet `fib(n)` bliver astronomisk stor når `n` vokser. Da der udføres én addition pr. indre knude i træet vokser beregningsbyrden tilsvarende. Reelt er det dog vigtigt at forstå, at langt de fleste beregninger er *genberegninger* af allerede beregnede værdier af `fib(n)`.

Funktionen **`fib`** foretager store mængder af unødvendige genberegninger

Antallet af additioner vokser eksponentielt i forhold til parameteren **`n`**



Figur 34.1 *En illustration af beregningsprocessen af fib(5)*

- Mere effektive løsninger
  - En simpel iterativ løsning
  - En rekursiv løsning, som kortslutter genberegningerne

Vi ser på to mere effektive løsninger i næste afsnit.

34.4. Fibonacci tal (3)

Lektion 8 - slide 12

Den mest oplagte - men også den mindst interessante - effektivisering er programmering af `fib` med en forløkke. Denne udgave er vist i program 34.4.

```
#include <stdio.h>

long fib(long n){
  long small, large, temp, m;

  for (small = 0, large = 1, m = 0;
       m < n;
       m++){
    temp = small;
    small = large;
    large = large + temp;
  }

  return small;
}

int main(void) {
  long i;

  for(i = 0; i < 100; i++){
    printf("Fib(%li) = %li\n", i, fib(i) );
  }


  return 0;
}
```

Program 34.4 *En iterativ udgave af fib programmeret med en forløkke.*

I program 34.4 har vi variablene `small` og `large`, som på et hvert tidspunkt indeholder hhv. `fib(n)` og `fib(n+1)` for et eller andet heltal `n`. Variablene `small` og `large` opdateres stort set på samme måde som de to variable `p` og `q` i proceduren `swap`, jf. program 23.2.

Herunder ser vi en udgave af `fib`, som minder meget om program 34.2. Forskellene fremgår af de blå, røde og lilla programsteder. Vi har indført et array, `memo`, som indeholder allerede beregnede værdier af `fib(n)`. I stedet for at genberegne `fib(n)` hentes det ud af tabellen `memo` (det røde sted). Vi skal naturligvis også huske at putte nyberegnede værdier af `fib(n)` ind i `memo` (det lilla sted). Bemærk at rækkefølgen af tilfældene i if-else kæden er vigtig at iagttage.

```c
#include <stdio.h>


long fib(long n){
  long result;
  static long memo[100];   /* elements pre-initialized to 0 */

  if (n == 0)
    result = 0;
  else if (n == 1)
    result = 1;
  else if (memo[n] != 0)
    result = memo[n];
  else {
    result = fib(n-1) + fib(n-2);
    memo[n] = result;
  }

  return result;
}

int main(void) {
  long i;

  for(i = 0; i < 100; i++)
    printf("Fib(%li) = %li\n", i, fib(i));

  return 0;
}
```

Program 34.5 *En memoriseret udgave af fib.*

Hermed afslutter vi vores interesse for Fibonacci tal.

34.5. Potensopløftning (1)

Lektion 8 - slide 13

Potensopløftning handler i bund og grund om at gange et tal med sig selv et antal gange. Vi starter med at se på en simpel, ligefrem og rekursiv potensopløfningsfunktion. I afsnit 34.6 ser vi på en meget mere interessant og meget bedre udgave.

Den beregningsmæssige idé er vist herunder. Bemærk at vi f.eks. tillægger 2^{-3} betydning, nemlig som $1/2^3$.

- Beregningsidé:
  - **potens > 0:** $tal^{potens} = tal^{potens-1} \cdot tal$
  - **potens = 0:** $tal^{0} = 1.0$
  - **potens < 0:** $1.0 / tal^{-potens}$

Idéen er direkte og meget ligefrem implementeret i program 34.6 herunder.

```
double power(double number, int pow){
  double result;

  if (pow == 0)
    result = 1.0;
  else if (pow > 0)
    result = number * power(number, pow - 1);
  else
    result = 1.0 / power(number, -pow);

  return result;
}
```

Program 34.6 *Den simple power funktion.*

34.6. Potensopløftning (2)

Lektion 8 - slide 14

Løsningen i program 34.6 var ganske naiv. Det er måske overraskende at der kan udtænkes en rekursiv løsning som er meget mere effektiv uden at være ret meget mere kompliceret. Ideen i den nye løsning er beskrevet på matematisk vis herunder.

- Beregningsidé:
  - ***potens > 0, potens er lige:*** $tal^{potens} = (tal^{potens/2})^{2}$
  - ***potens > 0, potens er ulige*** $tal^{potens} = tal^{potens-1} \cdot tal$
  - ***potens = 0:*** $tal^{0} = 1.0$
  - ***potens < 0:*** $1.0 / tal^{-potens}$

Den essentielle indsigt er at opløftning i en lige potens *p* kan udføres ved at opløfte til potensen *p/2* efterfulgt at en kvadrering. I stedet for at udføre *p* multiplikationer skal vi nu kun udføre arbejdet forbundet ved opløfning i *p/2* potens efterfulgt af en kvadrering. Hvis *p/2* igen er lige kan vi igen halvere. Hvis *p/2* er ulige ved vi at *(p/2)-1* er lige. Så "tricket" kan typisk gentages flere gange - rekursivt. Kvadrering består naturligvis af en enkelt multiplikation.

Programmet herunder er en direkte og ligefrem implementation af ovenstående formler.

```
double power(double number, int pow){
  double result;

  printf("power(%lf,%i)\n", number, pow);
  if (pow == 0)
    result = 1.0;
  else if (pow > 0 && even(pow))
    result = sqr(power(number,pow/2));
  else if (pow > 0 && odd(pow))
    result = number * power(number, pow - 1);
  else
    result = 1.0 / power(number, -pow);

  return result;
}
```

Program 34.7 *Den hurtige power funktion.*

På en separate slide i materialet leverer vi dokumentation for fordelene ved program 34.7 i forhold til program 34.6. Det sker ved et billedserie som viser de rekursive kald i `power(2.0,10)`. Vi viser også et komplet program som kalder både `power` ala program 34.6 og `power` ala program 34.6. Udskriften af programmet viser antallet af multiplikationer i en lang række kald. Det fremgår klart, at antallet af multiplikationer i den "smarte udgave" vokser meget langsomt i forhold til potensen. Væksten viser sig at være logaritmisk, idet vi i mindst hver andet kald halverer potensen.

35. Hvordan virker rekursive funktioner?

Før det næste eksempel indskyder vi her et ganske kort afsnit, som i tekst gør rede for hvordan rekursive funktioner virker internt.

35.1. Implementation af rekursive funktioner

Lektion 8 - slide 17

De tre første punkter gør rede for stakken af activation records, som vi har set i både udviklingen af de simple (og kunstige) rekursive kald af f fra afsnit 33.2 og udviklingen af power(2.0,10) omtalt i afsnit 34.6.

- Hvert funktionskald kræver afsættelse af en *activation record* - lager til værdier af aktuelle parametre, lokale variable, og noget "internt bogholderi".
- En kæde af rekursive funktionskald kræver én *activation record* pr. kald/inkarnation af funktionen.
- Man skal være varsom med brug af rekursion som kan afstedkomme mange (tusinder) af rekursive kald.
    - Dette kan føre til 'run time stack overflow'.
- Der findes en speciel form for rekursion hvor lagerforbruget kan optimaliseres

- Denne form kaldes halerekursion
- Halerekursion er tæt forbundet med iterative løsninger (brug af while eller for kontrolstrukturer).

Det sidste punkt (og de to underpunkter) nævner *halerekursion* (tail recursiveness på engelsk). Vi går ikke i detaljer med halerekursion i dette materiale. Jeg henviser til materialet Functional Programming in Scheme som diskuterer emnet forholdsvis grundigt.

Enhver løkke (ala while, do, for) kan let omprogrammeres med en (hale)rekursiv funktion.

Nogle former for rekursion kan kun meget vanskeligt omprogrammeres med løkker.

36. Towers of Hanoi

Ligesom Fibonacci tal er Towers of Hanoi en absolut klassiker i diskussion og introduktion af rekursion.

Towers of Hanoi illustrerer en beregningsmæssig opgave med eksponentiel køretid. Derved ligner dette program vores første program til beregning af Fibonacci tal (afsnit 34.2). Ligheden rækker dog ikke langt. Det var helt enkelt at finde en effektiv algoritme til udvikling af talrækken af Fibonacci tal. Der er ingen genveje til løsning af Towers of Hanoi problemet.

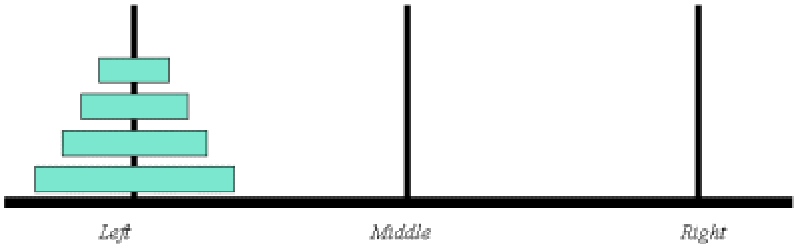
36.1. Towers of Hanoi (1)

Lektion 8 - slide 19

Problemet bag Towers of Hanoi er at flytte skiver fra et tårn til en anden med en tredje som 'mellemstation'. De nærmere regler er beskrevet herunder.

Towers of Hanoi er en gammel asiatisk overlevering om munke, der skulle flytte 64 skiver fra én søjle til en anden

Ifølge legenden ville templet og verden omkring det falde inden arbejdet kunne fuldføres



Figur 36.1 *En illustration af Towers of Hanoi med fire skiver.*

- Problemet:
  - Flyt stakken af skiver fra venstre stang til højre stang med brug af midterste stang som 'mellemstation'
  - Skiverne skal flyttes én ad gangen
  - En større skive må aldrig placeres oven på en mindre skive

36.2. Towers of Hanoi (3)

Lektion 8 - slide 21

Vi løser Towers of Hanoi problemet ved udpræget brug af en del og hersk strategi, jf. afsnit 11.6. Delproblemerne appellerer til rekursiv problemløsning, idet delproblemerne til forveksling ligner det oprindelige problem.

> Vi programmerer en løsning på problemet som kun viser hvilke flytninger der skal foretages.
>
> Bogen har en løsning, som også viser hvordan tårnene ser ud efter hver flytning.

Funktionen `hanoi` herunder flytter `n` skiver fra tårn `a` til tårn `b` med brug af tårn `c` som mellemstation. Som det fremgår af program 36.2 er typen `tower` en enumeration type som dækker over værdierne `left`, `middle` og `right`. Enumerationtyper blev behandlet i afsnit 18.3.

```
/* Move n discs from tower a to tower b via tower c */
void hanoi(int n, tower a, tower b, tower c){
  if (n == 1)
    move_one_disc(a,b);
  else {
    hanoi(n-1,a,c,b);
    move_one_disc(a,b);
    hanoi(n-1,c,b,a);
  }
}
```

Program 36.1 *Den centrale funktion i "Towers of Hanoi".*

Lad os nu forstå løsningen i program 36.1. Hvis `n` er lig med 1 er løsningen triviel. Vi flytter blot skiven fra tårn `a` til `b`. Dette gøres med `move_one_disc`. Hvis `n` er større end 1 flyttes først `n-1`

skiver fra `a` til hjælpetårnet `c` (det første røde sted). Dernæst flyttes den nederste store skive fra `a` til `b` med `move_one_disc` (det andet blå sted). Endelig flyttes de `n-1` skiver på fra `c` til `b`, nu med `a` som hjælpetårn (det andet røde sted).

Hele programmet er vist i program 36.2. Vi ser først enumeration typen `tower` og dernæst `hanoi` funktionen beskrevet herover. Funktionen `move_one_disc` rapporterer blot flytningen ved at udskrive data om flytning på standard output (skærmen). Variablen `count`, som er static i funktionen (jf. afsnit 20.3), tæller antallet af flytninger, blot for at tilfredsstille vores nysgerrighed om opgavens omfang. Funktionen `tower_out`, som kaldes fra `move_one_disc`, løser problemet med at udskrive en `tower` værdi. Denne problematik er tidligere illustreret i bl.a. program 18.7. Funktionen `main` prompter brugeren for antallet af skiver, og dernæst kaldes `hanoi`.

```
#include <stdio.h>

enum tower {left, middle, right};
typedef enum tower tower;

void move_one_disc(tower a, tower b);
char *tower_out(tower t);


/* Move n discs from tower a to tower b via tower c */
void hanoi(int n, tower a, tower b, tower c){
  if (n == 1)
    move_one_disc(a,b);
  else {
    hanoi(n-1,a,c,b);
    move_one_disc(a,b);
    hanoi(n-1,c,b,a);
  }
}

void move_one_disc(tower a, tower b){
  static long count = 0;
  count++;
  printf("%5i. Move disc from  %6s  tower to  %6s  tower\n",
         count, tower_out(a), tower_out(b));
}

char *tower_out(tower t){
  static char *result;
  switch (t){
    case left: result = "LEFT"; break;
    case middle: result = "MIDDLE"; break;
    case right: result = "RIGHT"; break;
  }
  return result;
}

int main(void) {
  int number_of_discs;

  printf("How many discs: ");
  scanf("%d", &number_of_discs);
```

```
  hanoi(number_of_discs, left, right, middle);

  return 0;
}
```

Program 36.2 *Hele programmet.*

En kørsel af program 36.2 med fire skiver giver følgende output:

```
 1. Move disc from    LEFT  tower to  MIDDLE  tower
 2. Move disc from    LEFT  tower to   RIGHT  tower
 3. Move disc from  MIDDLE  tower to   RIGHT  tower
 4. Move disc from    LEFT  tower to  MIDDLE  tower
 5. Move disc from   RIGHT  tower to    LEFT  tower
 6. Move disc from   RIGHT  tower to  MIDDLE  tower
 7. Move disc from    LEFT  tower to  MIDDLE  tower
 8. Move disc from    LEFT  tower to   RIGHT  tower
 9. Move disc from  MIDDLE  tower to   RIGHT  tower
10. Move disc from  MIDDLE  tower to    LEFT  tower
11. Move disc from   RIGHT  tower to    LEFT  tower
12. Move disc from  MIDDLE  tower to   RIGHT  tower
13. Move disc from    LEFT  tower to  MIDDLE  tower
14. Move disc from    LEFT  tower to   RIGHT  tower
15. Move disc from  MIDDLE  tower to   RIGHT  tower
```

Program 36.3 *Output fra programmet ved flytning af fire skiver.*

Som det tydeligt fremgår af næste afsnit, vil outputtet fra program 36.2 føles som uendelig langt hvis `hanoi` kaldes med et stort heltal `n` som input.

36.3. Towers of Hanoi (4)

Lektion 8 - slide 22

Som allerede observeret forårsager hvert ikke-trivielt kald af `hanoi` to yderligere kald. Derved fordobles arbejdet for hver ekstra skive, som er involveret. Dette er eksponentiel vækst.

> Det er let at se at antallet af flytninger *F(n)* vokser eksponentielt med antallet af diske i tårnet.
>
> Mere præcist ser vi at $F(n) = 2^n - 1$.

Vi kan kun håndtere et arbejde, som vokser eksponentielt, for meget begrænsede problemstørrelser. I tabel 36.1 illustreres dette konkret. I søjlen 10^{-9} antager vi at vi kan flytte 1000000000 skiver pr. sekund. I søjlen 10^{-3} antager vi at vi kun kan flytte 1000 skiver pr. sekund. Uanset dette ser vi at arbejdet tager tilnærmelsesvis uendelig lang tid at udføre, når `n` går mod 100.

n	**2^n**	**$10^{-9} \cdot 2^n$**	**$10^{-3} \cdot 2^n$**
5	32	$3.2 \cdot 10^{-8}$ sek	$3.2 \cdot 10^{-2}$ sek
25	$3.4 \cdot 10^7$	0.03 sek	9.3 timer
50	$1.1 \cdot 10^{15}$	13 dage	35702 år
65	$3.7 \cdot 10^{19}$	1170 år	$1.17 \cdot 10^9$ år
80	$1.2 \cdot 10^{24}$	$3.8 \cdot 10^7$ år	$3.8 \cdot 10^{13}$ år
100	$1.2 \cdot 10^{39}$	$4.0 \cdot 10^{13}$ år	$4.0 \cdot 10^{20}$ år

Tabel 36.1 *En table der viser antallet af flytninger og køretider af hanoi funktionen. Køretiderne i tredje kolonne er angivet under antagelsen af vi kan flytte 10^9 skiver pr. sekund. Køretiderne i fjerde kolonne er angivet under antagelsen af vi kan flytte 10^3 skiver pr. sekund.*

37. Quicksort

Som det sidste indslag i lektionen om rekursion ser vi her på Quicksort. Quicksort sorterer et array, ligesom det var tilfældet med Bubblesort i afsnit 24.4.

Quicksort er en klassisk rekursiv sorteringsprocedure. Tilmed er Quicksort bredt anerkendt som én af de bedste sorteringsteknikker overhovedet.

Hvis man skal bruge Quicksort i C skal man anvende `qsort` fra standard C biblioteket. Programmet, som udvikles i denne lektion er funktionel, men ikke optimal i praktisk forstand. Det er svært at programmere Quicksort med henblik på god, praktisk anvendelighed.

37.1. Quicksort (1)

Lektion 8 - slide 24

I dette afsnit vil vi med ord beskrive den algoritmiske idé i Quicksort. I afsnit 37.2 viser vi den overordnede rekursive sorteringsfunktion i C. I afsnit 37.3 diskuterer vi den centrale detalje i algoritmen, som er en iterativ array opdelningsfunktion. Endelig, i afsnit 37.4 vurderer vi god Quicksort er i forhold til Bubblesort.

Quicksort er et eksempel på en rekursiv sorteringsteknik der typisk er meget mere effektiv end f.eks. boblesortering

- Algoritmisk idé:
  - Udvælg et vilkårligt *deleelement* blandt tabellens elementer
  - Reorganiser tabellen således at den består af to dele:
    - En venstre del hvor alle elementer er mindre end eller lig med deleelementet
    - En højre del hvor alle elementer er større end eller lig med deleelementet
  - Sorter både venstre og højre del af tabellen rekursivt, efter samme idé som netop beskrevet

Reorganiseringen af elementerne i punkt to herover kan opfattes som en *grovsortering* af tabellen i små og store elementer, relativ til deleelementet.

Quicksort er et andet eksempel på en rekursiv del og hersk løsning

En stor del af arbejdet består i opdelning af problemet i to mindre delproblemer, som egner sig til en rekursiv løsning

Sammensætningen af delproblemernes løsninger er triviel

37.2. Quicksort (2)

Lektion 8 - slide 25

Herunder viser vi C funktionen `quicksort`. Som det ses er der to rekursive kald `quicksort` i hver aktivering af `quicksort`. Baseret på vores erfaring med Fibonacci tal fra afsnit 34.2 og Towers of Hanoi fra afsnit 36.2 kunne det godt få alarmklokkerne til at ringe med tonerne af 'langsommelighed'. Det viser sig dog at være ubegrundet. Mere om dette i afsnit 37.4.

```
/* Sort the array interval a[from..to] */
void quicksort(element a[], index from, index to){
  index point1, point2;

  if (from < to){
    do_partitioning(a, from, to, &point1, &point2);
    partitioning_info(a, from, to, point1, point2);
    quicksort(a, from, point1);
    quicksort(a, point2, to);
  }
}
```

Program 37.1 *Den rekursive quicksort funktion.*

Det er ligetil at forstå program 37.1 ud fra opskriften - den algoritmiske idé - i afsnit 37.1. Kaldet af `do_partitioning` udvælger dele elementet og foretager opdelningen af arrayet i en venstre del

med små elementer og en højre del med store elementer. Med andre ord bestyrer `do_partitioning` grovsorteringen.

Givet en veludført opdelning kan sorteringsopgaven fuldføres ved at sortere de små elementer i den første røde del. Dernæst sorteres de store elementer i den anden røde del. Dette gøres naturligvis rekursivt, fordi denne opgave er af præcis samme natur som det oprindelige problem. Bemærk her, at når de to rekursive sorteringer er fuldført, er der ikke mere arbejdet at der skal udføres i den oprindelige sortering. Der er altså intet kombinationsarbejde, som vi ofte ser i forbindelse med del og hersk problemløsning.

37.3. Quicksort (3)

Lektion 8 - slide 26

Vi vil nu se på den centrale 'detalje' i `quicksort`, nemlig grovsorteringen. Dette udføres af et kald af `do_partitioning`, som vises i program 37.2.

Funktionen `do_partitioning` opdeler den del af arrayet `a` som er afgrænset af indekserne `from` og `to`. Resultatet af opdelningen er to indekser, der føres tilbage fra `do_partitioning` via to call by reference parametre `point_left` og `point_right`.

```
/* Do a partitioning of a, and return the partitioning
   points in *point_left and *point_right */
void do_partitioning(element a[], index from, index to,
                     index *point_left, index *point_right){
 index i,j;
 element partitioning_el;

 i = from - 1; j = to + 1;
 partitioning_el = a[from];
 do{
   do {i++;} while (a[i] < partitioning_el);
   do {j--;} while (a[j] > partitioning_el);
   if (i < j) swap(&a[i],&a[j]);
 } while (i < j);

 if (i > j) {
   *point_left = j; *point_right = i;}
 else {
   *point_left = j-1; *point_right = i+1;}
}
```

Program 37.2 *Funktionen do_partitioning der opdeler tabellens elementer op i store og små.*

Lad os løbe hen over `do_partitioning` i nogen detalje. På den tilknyttede slide findes der en billedserie, som støtter forståelsen af opdelningen. Den brune del udvælger et deleelement blandt tabellens elementer. Et hvilket som helst element kan vælges. Vi vælger blot `a[from]`, altså det første. Indekserne `i` og `j` gennemløber nu tabellen fra hver sin side i den sorte while-løkke. `i` tvinges frem af den røde do-løkke, og `j` tvinges tilbage af den blå do-løkke. For hvert gennemløb af

den ydre while-løkke foretages en ombytning af en *uorden* i tabellen. Den lilla del forneden sørger for justeringen af de endelige delepunkter.

Som man kan se er der en del detaljer at tage sig af i `do_partitioning`. Det er svært at tune disse, så man får en god, praktisk og velfungerende implementation af Quicksort ud af det.

37.4. Quicksort (4)

Lektion 8 - slide 27

I dette afsnit vil vi kort indse, at vores bekymring om køretiden fra afsnit 37.2 ikke er begrundet. Selv om Quicksort kalder sig selv to gange er vi meget langt fra at udføre et eksponentielt arbejde.

En god implementation plejer at være blandt de allerbedste sorteringsmetoder - men der er ingen garanti

- Køretid i bedste tilfælde:
  - I størrelsesordenen $n \cdot log_2\,(n)$ sammenligninger og ombytninger.
- Køretid i værste tilfælde
  - I størrelsesordenen n^2 sammenligninger og ombytninger.

Argumentet for bedste tilfælde køretiden på $n \cdot log(n)$ er følgende. Hvert kald af `do_partitioning` har en køretid proportional med afstanden `to - from`. Hvis vi derfor kalder `do_partitioning` på et array af størrelsen n udfører vi et arbejde som svarer til n. Vi siger også undertiden at arbejdet er $O(n)$.

Antag nu, at `do_partitioning` deler tabellen på midten i hvert rekursivt kald. Disse opdelninger kan foretages $log(n)$ gange. Samlet set (over alle kald på et givet niveau) udføres et arbejde i `do_partitioning` på n på hvert af de $log(n)$ niveauer. Derfor er køretiden af Quicksort samlet set $n \cdot log(n)$.

Hvis `do_partitioning` skævdeler tabellen i 1 og $(n-1)$ elementer opnås køretiden i værste tilfælde. Dermed får vi n niveauer hver med et samlet opdelningsarbejde på n. Med dette bliver køretiden af Quicksort på $n \cdot n$.

n	n^2	$n \cdot log_2\,(n)$
100	10.000	664
1000	1.000.000	9.966
10.000	100.000.000	132.887
100.000	10.000.000.000	1.660.964

Tabel 37.1 *En tabel der illustrerer forskellen på væksten af n^2 og $n \cdot log_2\,(n)$*

Det kan vises at en sortering med i størrelsesordenen $n \cdot log\ (n)$ sammenligninger og ombytninger er optimal.