Einführung in die Programmierung

Bachelor of Science

Prof. Dr. Rethmann

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

WS 2009/10

Übersicht

- *dynamische Programmierung*
- Lösen durch Ausprobieren
- Erweiterungen im C99-Standard

Motivation

rekursive Funktionen, also Funktionen die sich selbst wieder aufrufen, allerdings mit geänderten Parameterwerten, sind oft ineffizient

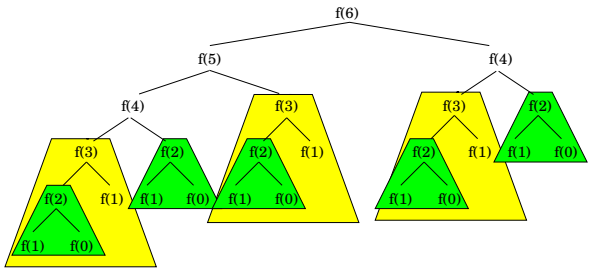
- *dynamische Programmierung:* verwende eine Tabelle, um bereits berechnete Werte zu speichern, und baue aus diesen Werten sukzessive eine größere Lösung zusammen
- *Backtracking:* allgemeine Problemlösung, bei der wir eine Lösung durch Ausprobieren ermitteln
  Was soll man machen, wenn einem nichts besseres einfällt?!

C99-Erweiterungen: C lebt! Daher gibt es in gewissen Zeitabständen Erweiterungen, die letzten waren 1990 und 1999.

Fibonacci-Zahlen

```
#include <stdio.h>

long fibo(int n) {
    if (n <= 1)
        return n;
    return fibo(n-1) + fibo(n-2);
}

void main() {
    int n;

    for (n = 0; n < 50; n++)
        printf("fib(%d) = %ld\n", n, fibo(n));
}
```


Fibonacci-Zahlen



Da viele rekursive Aufrufe mehrmals ausgeführt werden, können wir das Problem effizienter lösen, indem wir bereits berechnete Zwischenlösungen in einer Tabelle speichern.

Fibonacci-Zahlen

```
#include <stdio.h>
long fibs[50] = {0, 0, 0, 0, ...};

long fibo(int n) {
    if (n <= 1)
        return n;
    if (fibs[n-1] == 0)
        fibs[n-1] = fibo(n-1);   // memorieren
    return fibs[n-1] + fibs[n-2];
}

void main(void) {
    int n;

    printf("Wert? ");
    scanf("%d", &n);
    printf("fib(%d) = %ld\n", n, fibo(n));
}
```


Fibonacci-Zahlen

Wir können die Lösung auch von unten her aufbauen:

```
#include <stdio.h>
long fibs[50];

void main(void) {
    int n;

    fibs[0] = 0;
    fibs[1] = 1;
    for (n = 2; n < 50; n++) {
        fibs[n] = fibs[n-1] + fibs[n-2];
        printf("fib(%d) = %ld\n", n, fibs[n]);
   }
}
```


Fibonacci-Zahlen

Anmerkungen:

- Die rekursiven Verfahren arbeiten top down.
- Das letzte, iterative Verfahren arbeitet bottom up.
- dynamische Programmierung: anwenden des Bottom-Up-Prinzips auf Optimierungsprobleme

dynamische Programmierung

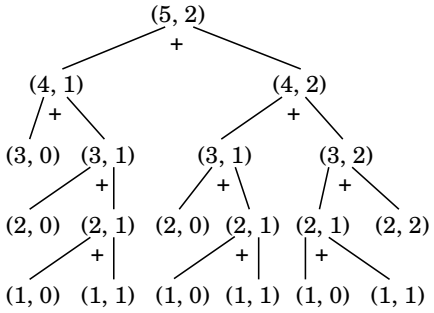
- berechne direkt die optimalen Lösungen der kleinsten Teilprobleme
- setze diese Teillösungen geeignet zu einer Lösung eines nächstgrößeren Teilproblems zusammen, und so weiter
- berechnete Teilergebnisse werden in einer Tabelle gespeichert, auf die immer wieder zurückgegriffen werden kann
- dynamische Programmierung vermeidet kostspielige Rekursionen, weil bekannte Teilergebnisse wiederverwendet werden

Binomialkoeffizienten

```
#include <stdio.h>

int bin(int n, int k) {
   if (n < k || k < 0)
      return -1;
   if (k == 0 || k == n)
      return 1;
   return bin(n-1, k-1) + bin(n-1, k);
}

void main() {
   int n, k;

   printf("n, k? ");
   scanf("%d, %d", &n, &k);
   printf("bin(%d,%d) = %d\n", n, k, bin(n, k));
}
```


Binomialkoeffizienten

Programmablauf:



Auch hier können wir die Technik des Memorierens oder die Bottom-Up-Methode anwenden!

Binomialkoeffizienten

```
#include <stdio.h>
int koeff[30][30] = {{0,0,0, ...}, ...,
                     {0,0,0, ...}};

int bin(int n, int k) {
   if (k == 0 || k == n)
      return 1;

   if (koeff[n-1][k-1] == 0)
      koeff[n-1][k-1] = bin(n-1, k-1);
   if (koeff[n-1][k] == 0)
      koeff[n-1][k] = bin(n-1, k);
   return koeff[n-1, k-1] + koeff[n-1, k];
}

void main(void) {
   printf("bin(%d,%d) = %d\n", 7, 3, bin(7,3));
}
```


Binomialkoeffizienten

```
#include <stdio.h>
int koeff[30][30];

// berechne Pascalsches Dreieck
void main(void) {
   int n, k;

   for (n = 0; n < 30; n++) {
      koeff[n][0] = 1;
      for (k = 1; k < n; k++)
         koeff[n][k] = koeff[n-1][k-1]
                     + koeff[n-1][k];
      koeff[n][n] = 1;
   }
   printf("bin(%d,%d) = %d\n", 7, 3,
          koeff[7][3]);
}
```


Geld sammeln

- gegeben ist ein Schachbrett und eine Spielfigur
- jedem Feld ist ein Geldbetrag zugeordnet, der beim Passieren des Feldes gutgeschrieben wird
- Startfeld: links oben
- Ende: Spielfigur steht in der rechten Spalte

Geld sammeln

Rekursionsformel? Dynamische Programmierung?

0	5	12	23	16	11	27	13	16
1	2	24	25	22	8	29	16	23
2	6	1	16	6	27	18	1	0
3	9	2	12	0	9	12	22	5
4	1	13	9	16	2	12	9	14
5	17	16	1	23	20	29	3	8
6	23	27	2	19	2	25	21	7
7	4	25	7	20	13	7	29	14
	0	1	2	3	4	5	6	7

Variante: In welchem Feld der ersten Spalte muss ich starten, um möglichst viel Geld zu sammeln?

Übersicht

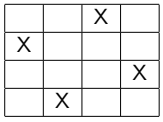
- dynamische Programmierung
- *Lösen durch Ausprobieren*
- Erweiterungen im C99-Standard

Backtracking

allgemeine Problemlösung: finde Lösung eines Problems

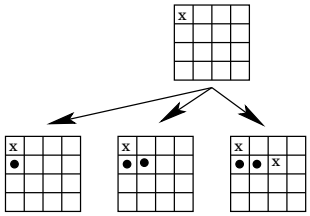
- nicht durch befolgen einer Berechnungsvorschrift,
- sondern durch Versuchen und Nachprüfen
- engl.: trial and error

Beispiel: n-Damen-Problem
Positioniere auf einem $n \times n$-Schachbrett n Damen so, dass keine Damen sich gegenseitig schlagen.

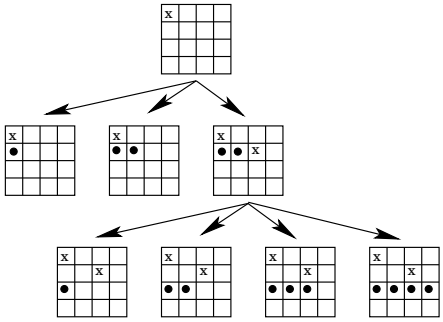


4-Damen-Problem

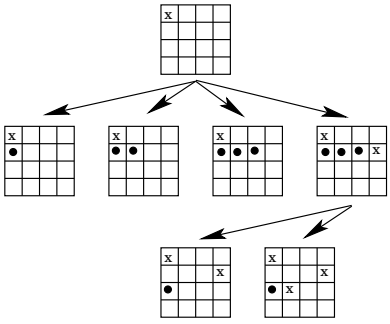
4-Damen-Problem



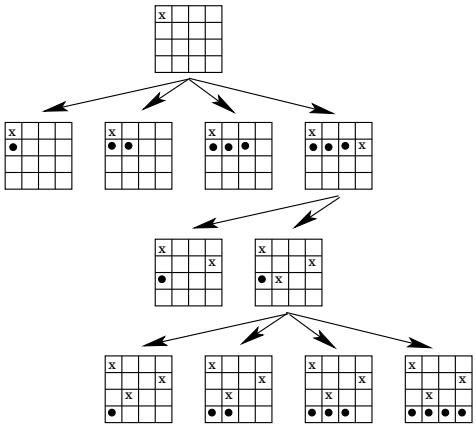
4-Damen-Problem



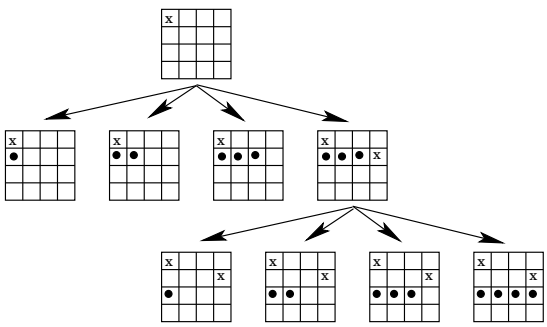
4-Damen-Problem



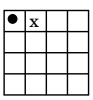
4-Damen-Problem



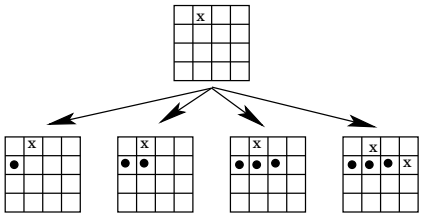
4-Damen-Problem



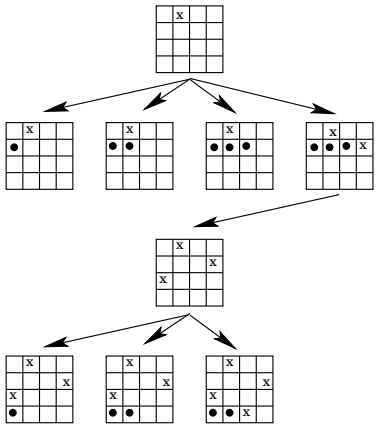
4-Damen-Problem



4-Damen-Problem



4-Damen-Problem



n-Damen-Problem

```
#include <stdio.h>
#include <stdlib.h>

char stateOk(int n);
char findPosition(int n, int dim);
void output(int n);

int *pos;

void main(void) {
    int n = 8;

    pos = (int *) calloc(n, sizeof(int));
    if (findPosition(0, n))
        output(n);
    else printf("%d: no solution\n", n);
    free(pos);
}
```


n-Damen-Problem

```
char findPosition(int n, int dim) {
    int i;
    char found = 0;

    for (i = 0; i < dim && !found; i++) {
        pos[n] = i;
        if (stateOk(n)) {
            if (n == dim - 1)
                return 1;
            found = findPosition(n + 1, dim);
        }
    }
    return found;
}
```


n-Damen-Problem

```
char stateOk(int n) {
    int z1, z2, s1, s2, dx, dy;

    for (int z1 = 0; z1 < n; z1++) {
        for (int z2 = z1 + 1; z2 <= n; z2++) {
            s1 = pos[z1];
            s2 = pos[z2];
            dx = s1 - s2;
            dy = z1 - z2;

            if (s1 == s2)
                return 0;
            if (dx == dy || dx == -dy)
                return 0;
        }
    }
    return 1;
}
```


n-Damen-Problem

```
void output(int dim) {
    int i, j;

    printf("\nn = %d\n", dim);
    for (i = 0; i < dim; i++) {
        for (j = 0; j < dim; j++) {
            if (pos[i] == j)
                printf("x");
            else printf("-");
        }
        printf(" | %3d\n", pos[i] + 1);
    }
    printf("\n");
}
```


Übersicht

- dynamische Programmierung
- Lösen durch Ausprobieren
- *Erweiterungen im C99-Standard*

Erweiterungen im C99 Standard

inline functions: Compiler-Hinweis, jeder Aufruf der Funktion ist durch Einfügen des Codes der Anweisungen des Funktionsrumpfs zu ersetzen.

Beispiel:

```
inline int max(int i, int j) {
    return (i > j) ? i : j;
}
```

⇒ spart das Erzeugen von Variablen sowie das Kopieren von Werten für Argumente und Funktionswert!

⇒ nur sinnvoll für Funktionen, deren Rumpf nur wenige Anweisungen enthalten

Erweiterungen im C99 Standard

Variablen-Deklaration sind nun an vielen Stellen im Code erlaubt, nicht nur zu Beginn eines Blocks

Beispiel:

```
#include <stdio.h>

void main(void) {
    for (int i = 0; i < 10; i++)
        printf("%2d: Hello, world!\n", i);

    int x = 1;
    while (x < 10)
        printf("x = %3d\n", x++);
}
```

⇒ kein sinnvolles Feature, Code wird schlechter lesbar

Erweiterungen im C99 Standard

neue Datentypen:

- `long long int` schließt Lücke zwischen 32- und 64-Bit
- `complex` Darstellung komplexer Zahlen + Arithmetik

```
#include <stdio.h>
#include <complex.h>

void main(void) {
   complex c = 1.0f + 1.0f * _Complex_I;

   printf("+ %f+i%f\n", creal(c+c), cimag(c+c));
   printf("- %f+i%f\n", creal(c-c), cimag(c-c));
   printf("* %f+i%f\n", creal(c*c), cimag(c*c));
   printf("/ %f+i%f\n", creal(c/c), cimag(c/c));
}
```


Erweiterungen im C99 Standard

erweiterte/neue Bibliotheken

- `stdint.h`
  Definition von `long long int`
- `complex.h`
  Darstellung komplexer Zahlen + Arithmetik
- `snprintf(char *str, size_t size, char *fmt, ...)`
  schreibt maximal `size` Zeichen nach `str`, Formatierung erfolgt anhand `fmt`
- `va_copy(va_list dest, va_list src)`
  (siehe variabel lange Parameterlisten)

Erweiterungen im C99 Standard

heute üblich: *Unicode*-Zeichensatz

- C90 definert *breite Zeichen*: Datentyp `wchar_t`
- C99 definiert alle Bibliotheksfunktionen über Zeichenketten auch in einer Version für breite Zeichen

```
#include <wchar.h>

int main(void) {
   wchar_t ustr[20] = L"abcde";
   int len;

   wprintf(L"%ls\n", ustr);
   swprintf(ustr, 20, L"%4i", 1234);
   len = wcslen(ustr);

   return 0;
}
```


Erweiterungen im C99 Standard

- *einzeilige Kommentare wie in C++:*

```
// Hauptprogramm
int main(int argc, char *argv[]) {
   int upper;  // obere Grenze
   int lower;  // untere Grenze
   ...
```

- *Arrays variabler Länge:*

```
   int len = atoi(argv[1]);
   int arr[len];   // nach ISO-C90 verboten
```


Variabel lange Parameterlisten

Motivation: Wir haben bereits (unbewusst?) variabel lange Parameterlisten angewendet:

```
printf("Ergebnis: %d\n", erg);
printf("fib[%d]= %d\n", i, fib[i]);
printf("bin[%d][%d]= %d\n", i, j, bin[i][j]);

scanf("%d", &no);
scanf("%d, %f", &no, &epsilon);
```

Wie ist das eigentlich realisiert?

Variabel lange Parameterlisten

Motivation: Bei Datenbankzugriffen mittels SQL können die Daten nach verschiedenen Kriterien sortiert werden.

```
select * from student
   order by matrikelnr;
select * from student
   order by fachbereich, gebdat;
select * from student
   order by name, vorname, gebdat;
```

Wie sähe entsprechender C-Code aus?

```
stud_t arr[100];
...
sortBy(arr, "matrikelnr");
sortBy(arr, "fachbereich", "gebdat");
sortBy(arr, "name", "vorname", "gebdat");
```


Variabel lange Parameterlisten

Die Standardbibliothek `stdarg` bietet die Möglichkeit, eine Liste von Funktionsargumenten abzuarbeiten, deren Länge und Datentypen nicht bekannt sind.

Beispiele:

```
int printf(const char *format, ...)
int scanf(const char *format, ...)
```

Parameterliste endet mit `...` → die Funktion darf mehr Argumente akzeptieren als Parameter explizit beschrieben sind

`...` darf nur am Ende einer Argumentenliste stehen

Variabel lange Parameterlisten

Beispiel: `int fkt(char *fmt, ...);`

Mit dem *Typ* `va_list` definiert man eine Variable, die der Reihe nach auf jedes Argument verweist.

```
va_list vl;
```

Das *Makro* `va_start` initialisiert `vl` so, dass die Variable auf das erste unbenannte Argument zeigt. Das Makro muss einmal aufgerufen werden, bevor `vl` benutzt wird.

Es muss mindestens einen Parameter mit Namen geben, da `va_start` den letzten Parameternamen benutzt, um anzufangen.

```
va_start(vl, fmt);
```


Variabel lange Parameterlisten

Jeder Aufruf des *Makros* `va_arg` liefert ein Argument und bewegt `vl` auf das nächste Argument.

`va_arg` benutzt einen Typnamen, um zu entscheiden, welcher Datentyp geliefert und wie `vl` fortgeschrieben wird.

```
ival = va_arg(vl, int);
sval = va_arg(vl, char *);
```

Vorsicht: Der Typ des Arguments wird nicht automatisch erkannt. Um den korrekten Typ angeben zu können, wird ein Format-String wie bei `printf(const char *, ...)` benutzt.

Variabel lange Parameterlisten

Vorsicht: Das Ende der Liste kann nicht anhand eines `NULL`-Wertes erkannt werden.

So nicht!

```
while (vl != NULL) {
    val = va_arg(vl, int);
    ...
}
```

⇒ Die Anzahl und die Datentypen der Parameter müssen bekannt sein. Beides kann mittels eines Format-Strings wie bei `printf` erreicht werden.

Variabel lange Parameterlisten

Beispiel: Format-String

```
for (; *fmt; fmt++) {
   switch(*fmt) {
   case 'd': ival = va_arg(vl, int);    break;
   case 'f': fval = va_arg(vl, double); break;
   case 's': sval = va_arg(vl, char *); break;
   }
   ...
```

Eventuell notwendige Aufräumarbeiten erledigt `va_end`.

```
va_end(vl);
```


Beispiel stdarg

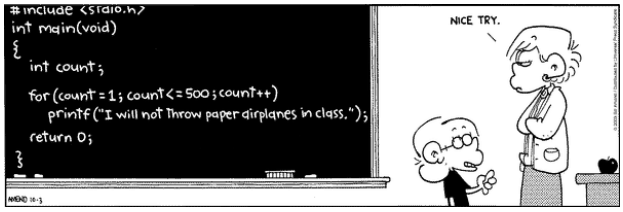
```
#include <stdio.h>
#include <stdarg.h>

void main(void) {
    fkt("sfdsd", "eins", 2.0, 3, "vier", 5);
    fkt("fdsd", 6.0, 7, "acht", 9);
}

int fkt(const char *fmt, ...) {
    int z;
    va_list l;

    va_start(l, fmt);
    for (z = 0; *fmt; fmt++, z++)
        getAndPrintNextValue(*fmt, &l);
    va_end(l);
    return z;
}
```


Beispiel stdarg

```
void getAndPrintNextValue(char c, va_list *l) {
    char *sval;
    int ival;
    double fval;

    if (c == 'd') {
        ival = va_arg(*l, int);
        printf("%d (int)\n", ival);
    } else if (c == 'f') {
        fval = va_arg(*l, double);
        printf("%f (double)\n", fval);
    } else if (c == 's') {
        sval = va_arg(*l, char *);
        printf("%s (char *)\n", sval);
    }
}
```


Was noch bleibt

Anwenden!!!



Evolution!!!

- Grundschule:

```
10 PRINT "HELLO WORLD"
20 END
```

- 5. Klasse Gymnasium:

```
program Hello(input, output);
begin
  writeln('Hello World');
end.
```

- Abiturjahrgang:

```
(defun hello<br>
(print<br>
(cons 'Hello (list 'World))))
```

Evolution!!!

- 1. Jahr Uni:

```c
#include <stdio.h>

void main(void) {
    char *message[] = {"Hello ", "World"};
    int i;

    for(i = 0; i < 2; ++i)
        printf("%s", message[i]);
    printf("\n");
}
```


Evolution!!!

- Hacker-Anfänger:

```perl
#!/usr/local/bin/perl
$msg="Hello, world.\n";
if ($#ARGV >= 0) {
    while(defined($arg=shift(@ARGV))) {
        $outfilename = $arg;
        open(FILE, ">".$outfilename)
            || die "Can't write $arg:$!\n";
        print (FILE $msg);
        close(FILE)
            || die "Can't close $arg: $!\n";
    }
} else {
    print ($msg);
}
```


Evolution!!!

- erfahrener Hacker:

```c
#include <stdio.h>
#define S "Hello, World\n"
main(){exit(printf(S) == strlen(S) ? 0 : 1);}
```

- Senior-Hacker:

```
% cc -o a.out ~/src/misc/hw/hw.c
% a.out
```

- Guru-Hacker:

```
% cat
Hello, world.
^D
```


Evolution!!!

- Neuer Manager:

```
10 PRINT "HELLO WORLD"
20 END
```

- Älterer Manager:

```
mail -s "Hello, world." bob@b12
Bob, could you please write me a program
that prints "Hello, world."?
I need it by tomorrow. ^D
```

- Senior-Manager:

```
% zmail jim
I need a "Hello, world." program by
this afternoon.
```


Evolution!!!

- Chef:

```
% letter
letter: Command not found.
% mail
To: ^X ^F ^C
% help mail
help: Command not found.
% damn!
!: Event unrecognized
% logout
```