Beispiele für komplexere Rekursionen

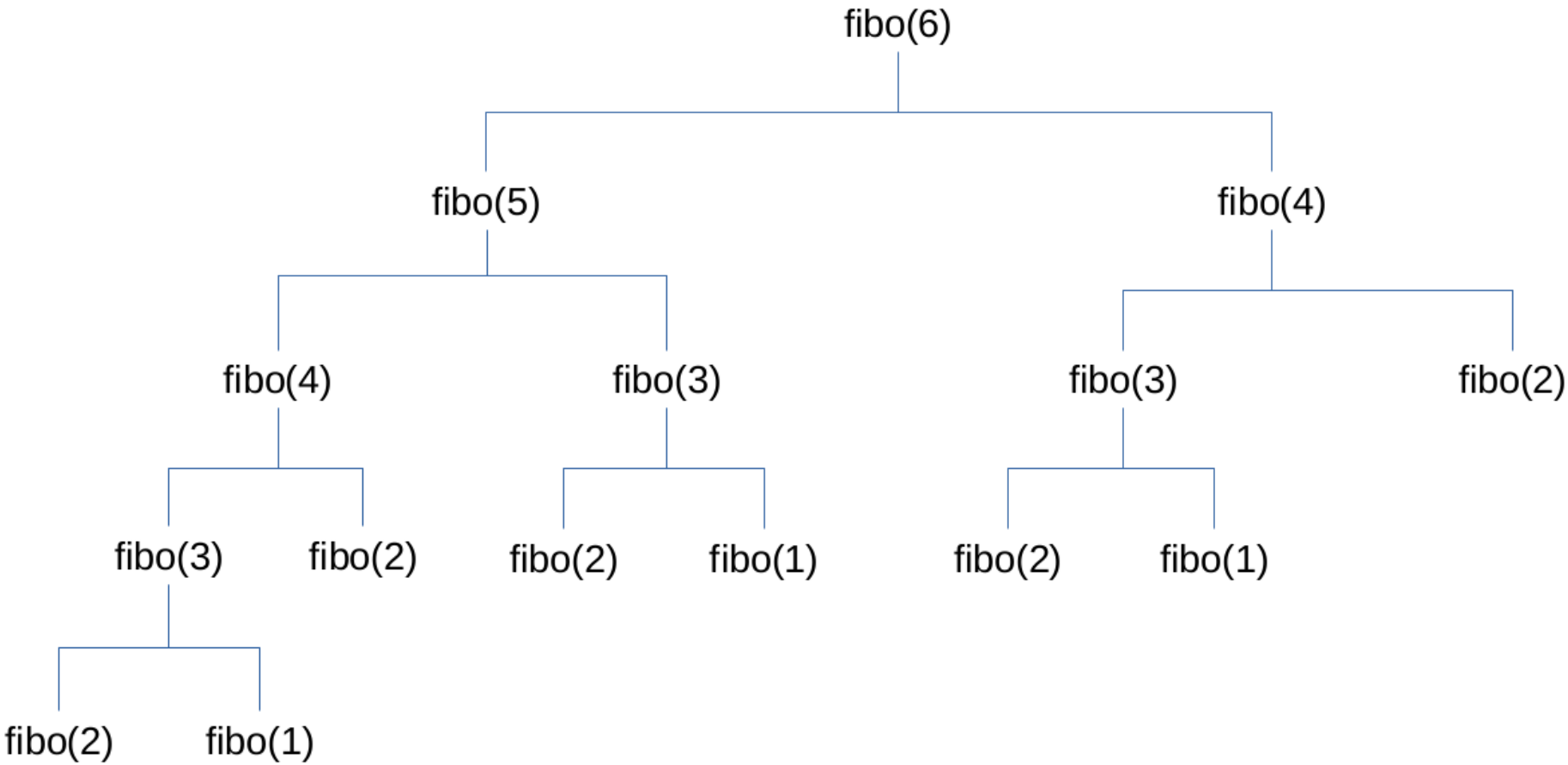
Beispiel Fibonacci-Folge:

1, 1, 2, 3, 5, 8, 13, 21, …

d.h.
für die ersten beiden Zahlen gilt Fibo(1)=Fibo(2)=1
für alle anderen gilt Fibo(N)=Fibo(N-1) + Fibo(N-2)

```
function fibo(n:word):word;
begin
  if n<=2 then fibo:=1
          else fibo:=fibo(n-1)+fibo(n-2)
end;
```

Auf jeder Rekursionsstufe werden **zwei** erneute Aufrufe der Funktion getätigt. Die Abbruchbedingung wird beispielsweise mit dem Aufruf N=6 insgesamt 8 mal erreicht und somit werden 8 Einsen addiert (fibo(6)=8). Das lässt sich anhand eines Baumes verdeutlichen:



Beispiel: Permutationen von Buchstaben in einer Zeichenkette der Länge n:

1. n=2, start='a', ende='b'

'aa', 'ab', 'ba', 'bb' Anzahl=4

2. n=1, start='a', ende='z'

'a', 'b', 'c', 'd', … , 'z' Anzahl=26

3. n=3, start='a', ende='k'

'aaa', 'aab', 'aac', 'aad', … , 'kkk' Anzahl=1331

$$\text{allg.: Anzahl} = (\text{ord}(\text{ende}) - \text{ord}(\text{start}) + 1)^n$$

Aufgabe

Lösung für n=3, start='a', ende='c' ?

Mögliche Lösung:

```
const n=3; start='a';ende='c';

procedure permut_iter;
var i,j,k:char;
begin
  for i:=start to ende do
    for j:=start to ende do
      for k:=start to ende do
        writeln(i+j+k)
end;
```

Problem dieser Lösung:

Das Unterprogramm ist zwar allgemein formuliert für den Unterbereich der Buchstaben (start bis ende), aber **nicht für n**.

Für n=4 müsste beispielsweise eine for-Schleife ergänzt werden.

Lösung mit rekursiven Ansatz:

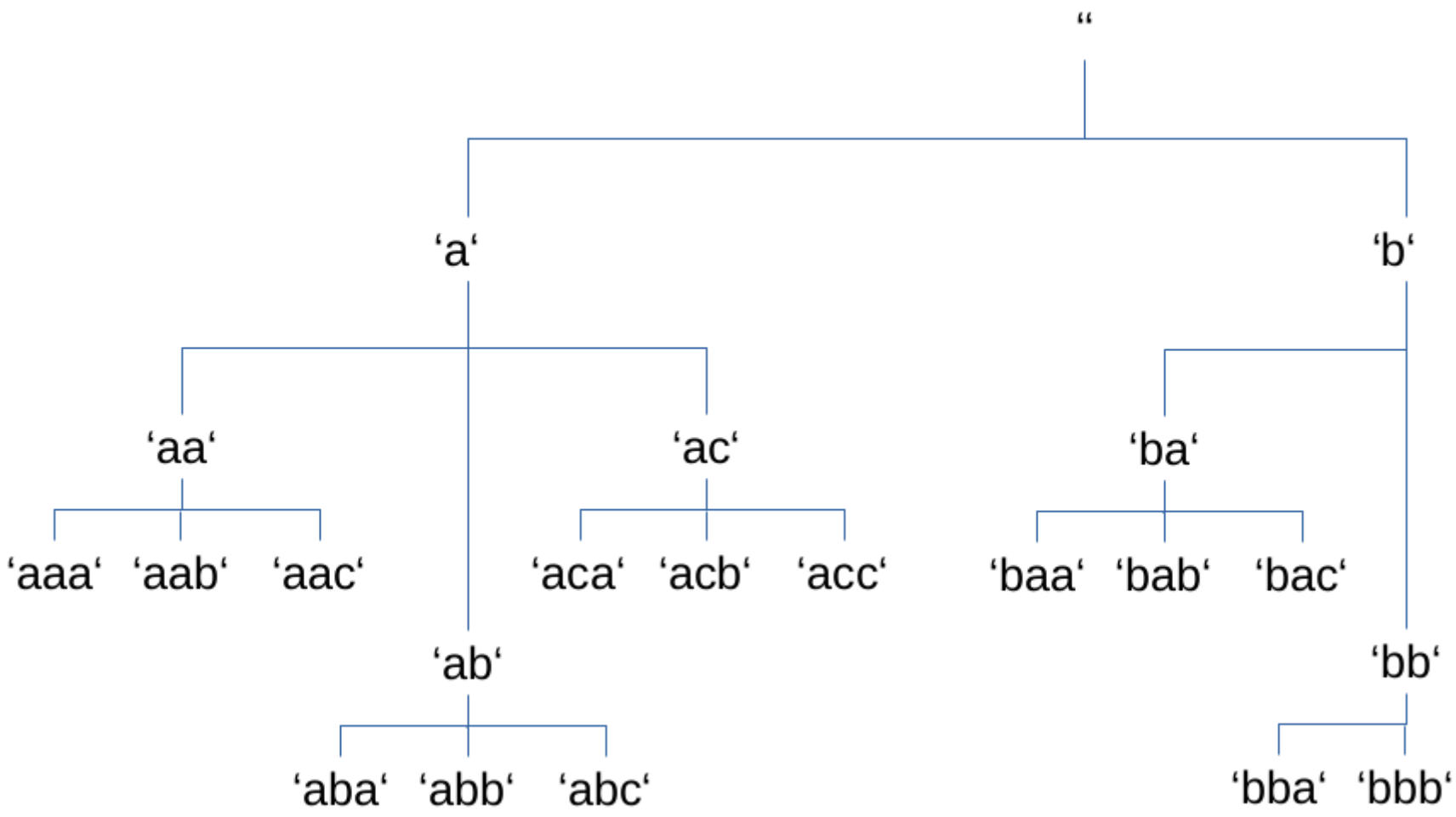
```
const n=3; start='a';ende='c';

procedure permut_reku(permutstring:string;ebene:byte);
var i:char;

begin
  if ebene=n
    then writeln(permutstring)
  else
    for i:=start to ende do
      permut_reku(permutstring+i,ebene+1)
end;

begin
  permut_reku('',0)
end.
```

Beispiel: N=3, start=‘a‘, ende=‘c‘; Ausschnitt aus dem entstehenden Baum



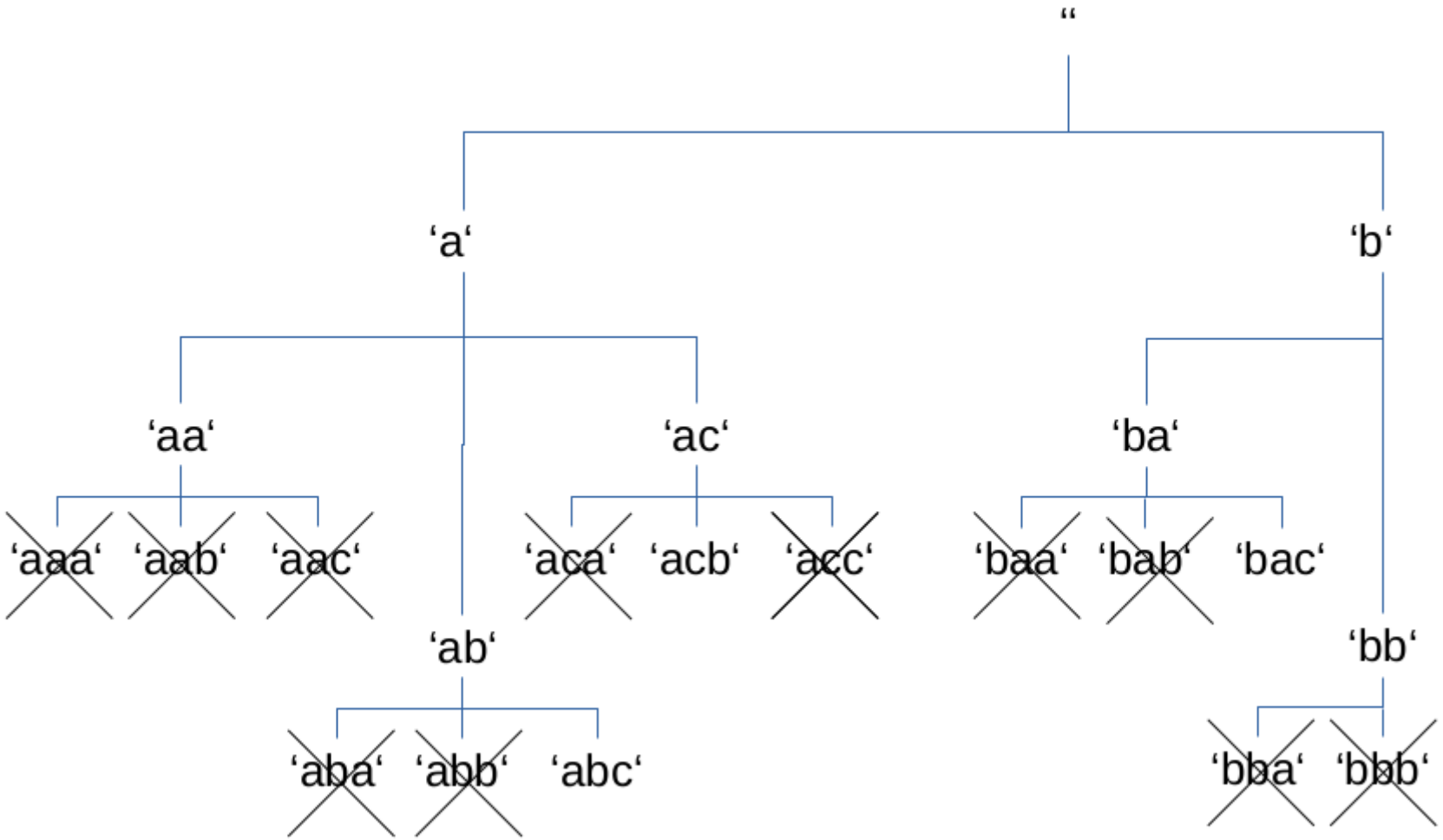
Nehmen wir an, dass wir nur nach irgendeinem Kriterium **gültige** Lösungen haben möchten (z.B. könnte gefordert werden, dass jeder Buchstabe maximal einmal vorkommen darf):

```
const n=3; start='a';ende='c';

function gueltig(s: string): boolean;
...

procedure permut_reku(permutstring: string; ebene: byte);
var i: char;
 begin
  if ebene = n then
    if gueltig(permutstring) then
      writeln(permutstring)
    else
    begin
      // leer; else gehört immer zur letzten If-Anweisung, in der es
      // keinen Else-Zweig gibt, d.h. es gäbe sonst ein Problem mit dem
      // nächsten else
    end
  else
    for i := start to ende do
       permut_reku(permutstring + i, ebene + 1);
end;
```

Beispiel: N=3, start=‘a‘, ende=‘c‘; Ausschnitt aus dem entstehenden Baum



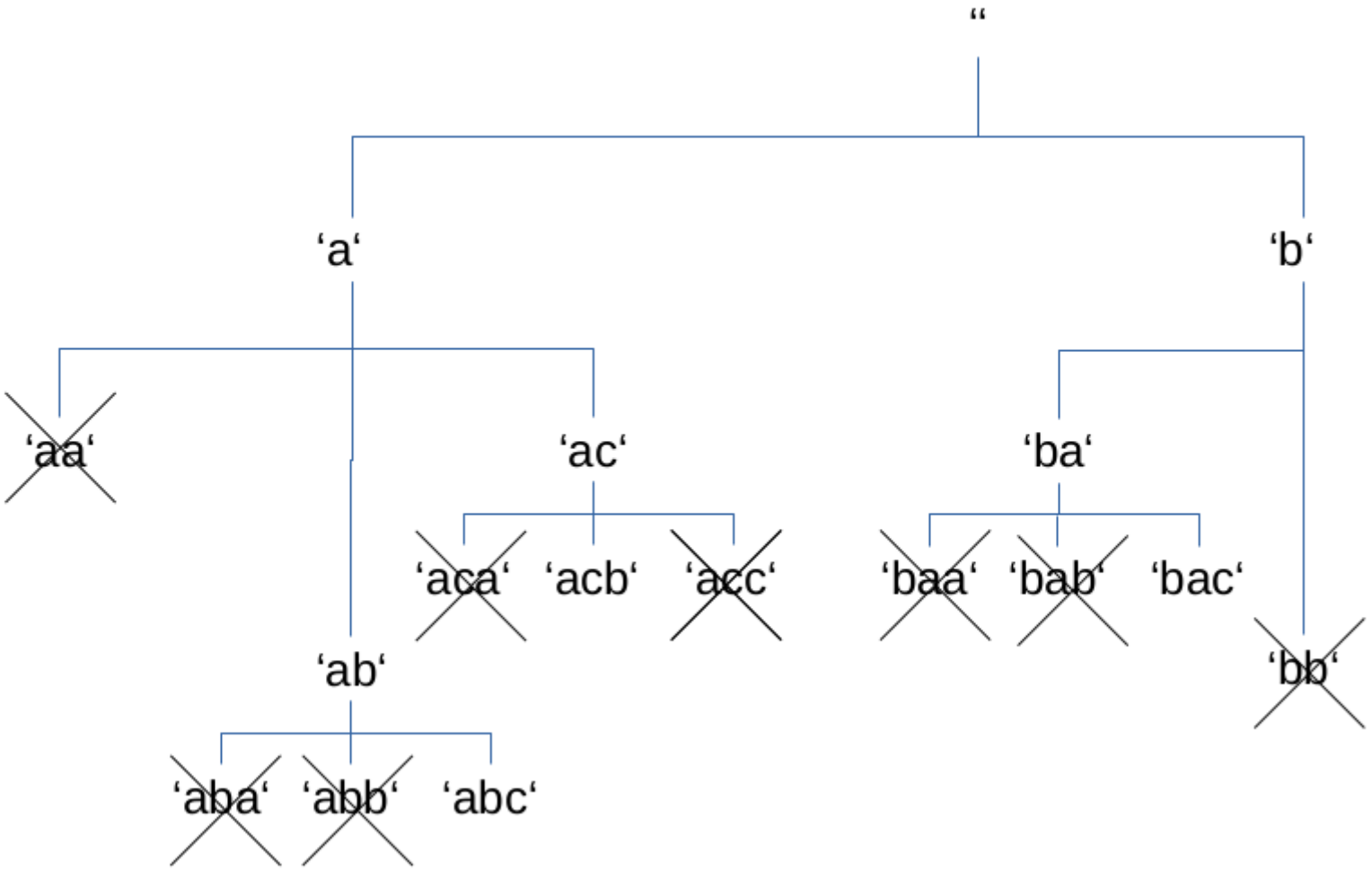
Laufzeittechnisch wäre es sicherlich besser, wenn wir ungültige Lösungen **früher erkennen** und dann den Baum an dieser Stelle nicht weiter verfolgen:

```
const n=3; start='a';ende='c';

function gueltig(s: string): boolean;
  …

procedure permut_reku(permutstring: string; ebene: byte);
  var i: char;
  begin
    if (ebene = n) or not gueltig(permutstring) then
        if gueltig(permutstring) then writeln(permutstring)
        else
        begin
        end
    else
      for i := start to ende do
        permut_reku(permutstring + i, ebene + 1);
  end;
```

Beispiel: N=3, start=‘a‘, ende=‘c‘; Ausschnitt aus dem entstehenden Baum



Zusätzlich möchten wir aus den gültigen Lösungen nur eine (nach irgendeinem Algorithmus ermittelte) **punktbeste** Lösung (genauer: die erste von evtl. mehreren besten Lösungen), erhalten und als **Parameter** zusammen mit den maximalen Punkten an den Aufrufer zurückgeben:

```
const n=3; start='a';ende='c';

function punkte(s: string): integer;
...

procedure permut_reku(permutstring: string; ebene: byte; var maxPunkte:
                      integer; var besterString: string);
var i: char;

begin
  if (ebene = n) or not gueltig(permutstring) then
    if gueltig(permutstring) and (punkte(permutstring) > maxPunkte) then
    begin
      besterString := permutstring;
      maxpunkte := punkte(permutstring);
    end
    else
    begin
    end
  else
```

```
      for i := start to ende do
         permut_reku(permutstring + i, ebene + 1, maxPunkte,
            besterString);
  end;

var
  maxPunkte: integer;
  besterString: string;

begin
  besterString := '';
  maxPunkte := 0;
  permut_reku('', 0, maxPunkte, besterString);
  writeln(besterString,' ',maxpunkte)
  ...
```