Anotace

- Dynamické programování,

Problémy řešitelné vyplněním tabulky

- Faktoriál,
- přednášející jde do M1,
- počet platných uzávorkování pomocí n párů závorek,
- počet rozkladů přirozeného čísla na součet nerostoucích kladných celých čísel,
- Pascalův trojúhelník,
- nejdelší rostoucí podposloupnost,
- pořadí násobení matic.

Přednášející jde (už poněkolikáté) do M1 (a při troše štěstí dnes naposledy):

- Rekurence $T(n) = T(n-1) + T(n-2)$,
- vedla na program:

```
function schody(a:integer):longint;
begin
if a=1 then schody:=1;
else  if a=2 then schody:=2;
                else  schody:=schody(a-1)+schody(a-2);
```

- Proto jsme si pořídili pole, kde už byly staré výsledky.
- Pole ani nebylo potřeba, pokud jsme začali "odpředu" a pamatovali si poslední dvě hodnoty.

Varianta s "cache"

```
program nic;
const MAX=150;
var cache:array[1..MAX] of longint;
function schody(a:integer):longint;
begin
      if cache[a]<>0 then schody:=cache[a]
      else
      begin
            if a= 1 then cache[a]:=1
            else if a=2 then cache[a]:=2
            else cache[a]:=schody(a-2)+schody(a-1);
            schody:=cache[a];
      end;
end;
```

Počet korektních uzávorkování

- Jak budeme řešit?
- Pomocí rekurze podle rostoucího počtu přidaných závorek.
- Uděláme funkci, která:
- zkusí přidat otevírací závorku (rekurze),
- zkusí přidat zavírací závorku (rekurze),
- pokud jsou použity všechny závorky, zvyš počet uzávorkování o 1.

Počet korektních uzávorkování

```
var paru,celkem:longint;
procedure pridej_zavorku(levych,pravych:integer);
begin
     if levych>pravych then
pridej_zavorku(levych,pravych+1);
     if paru>levych then
pridej_zavorku(levych+1,pravych);
     if (levych=pravych) and (paru=levych) then
inc(celkem);
end;
```

Jaký problém má toto řešení? Jaký problém má toto řešení?
Počítáme pořád to samé!

Závorkování – cache

- Jak z pasti? Položíme si správnou otázku:
- Záleží výsledek funkce `pridej_zavorku` na něčem jiném, než na parametrech `levych` a `pravych`?
- Nezáleží. Proč si potom nezapamatujeme, kolik závorkování přidá tato funkce pro konkrétní hodnoty `levych` a `pravych`:
- `cache:array[1..MAX,1..MAX] of longint`.
- Pole na počátku inicializujeme nulami,
- je-li `cache[levych,pravych]` =0, spustíme výpočet (výsledek ještě neznáme) a na konci funkce si ho uložíme do pole.
- Je-li `cache[levych,pravych]`<>0, připočteme tolik platných uzávorkování.
- Rozdíl mezi variantou "rekurze" a "cachovaná rekurze" je rozdíl mezi nepoužitelným a dobrým algoritmem!

Nalezení všech rozkladů zadaného čísla na součet

- Příklad: $2 = 2$ nebo $2 = 1 + 1$; $3 = 3$, $3 = 2 + 1$ anebo $3 = 1 + 1 + 1$.
- Sčítance chceme vždy v nerostoucím pořadí. Nápady jak řešit?
- Jako obvykle rekurzí. Budeme si pamatovat, kolik ještě zbývá rozdělit a kolik je maximum. A zkusíme všechno od maxima, až k jedné.
- Nezajímají nás samotné rozklady, ale jen jejich počet!

Rekurzívní funkce:

```
procedure rozloz(kolik,maximum:integer);
var i:integer;
begin
     if kolik:=0 then inc(pocet)
     else
          for i:=maximum downto 1 do
               rozloz(kolik-i,i);
end;
```

Jaký je problém?
Pořád ten samý
(počítáme pořád to samé).

Rozklad na sčítance

- Jak z pasti tentokrát?
- Stejně jako u závorkování:
- Uděláme dvourozměrné pole cache a budeme si do něj značit, kolika způsoby lze rozložit KOLIK, je-li první sčítanec nejvýše MAXIMUM:

```
procedure rozloz(kolik,maximum:integer);
var i,nazacatku:integer;
     if cache[kolik,maximum]<>0 then
           rozloz:=cache[kolik,maximum];
     else
     begin  nazacatku:=pocet;
          if kolik= 0 then inc(pocet);
          else  for i:=maximum downto 1 do
               rozloz(kolik-i,i);
          cache[kolik,maximum]:=pocet-nazacatku;
     end;
end;
```

Pascalův trojúhelník

- Obsahuje kombinační čísla,
- n-tý řádek konkrétně obsahuje hodnoty $\binom{n}{0}, \binom{n}{1}, \ldots, \binom{n}{n}$,
- při výpočtu využíváme toho, že $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$,
- pro jednoduchost chceme spočítat jen číslo $\binom{n}{k}$.

Pascalův trojúhelník rekurzívní řešení

- Získali jsme rekurenci $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$,
- z této snadno sestrojíme rekurzívní program:

```
function kombin(n,k:integer):longint;
begin
      if (k=0) or (k=n) then kombin:=1;
      else  kombin:=kombin(n-1,k-1)+kombin(n-1,k);
end;
begin
      kombin(100,50);
end.
```

Mále stejný problém, pořád počítáme to samé mockrát.
Opět přidáme cache na výsledky.

Pascalův trojúhelník

```
const MAX=100;
var cache:array[0..MAX,0..MAX] of longint;
function kom(n,k:integer):longint;
begin
    if cache[n,k]=0 then
    begin if (k=0) or (k=n) then cache[n,k]:=1
       else cache[n,k]:=kom(n-1,k-1)+kom(n-1,k);
    end;
    kom:=cache[n,k];
end;
var n,k:integer;
begin
    read(n,k);
    writeln(kom(n,k));
end
```

Nejdelší rostoucí podposloupnost

- Utvoř posloupnost dvojic (třeba pomocí pole recordů),
- první prvek obsahuje příslušnou hodnotu, druhý ukazuje délku nejdelší rostoucí podposloupnosti končící dotyčným prvkem.
- Vyplňuj zleva doprava, pro každý prvek najdi mezi prvky jemu předcházejícími takový menší prvek, ve kterém končí nejdelší dostupná podposloupnost.
- Podposloupnost najdi od konce:
- Najdi prvek nabízející největší možnou délku,
- poznamenej si HODNOTU a DÉLKU.
- postupuj od konce a pokud najedeš na prvek nabízející délku DÉLKA s hodnotou nejvýše tolik, kolik HODNOTA, prvek si poznamenej, sniž DÉLKU o jedna a HODNOTU na hodnotu nalezeného prvku.
- Nalezenou posloupnost otoč.

Nejdelší rostoucí podposloupnost – výpočet (vyplnění tabulky)

```
for i:=1 to n do begin
     maximum:=0; maxindex:=0;
     for j:=i-1 downto 1 do begin
          if pole[j].hodnota<pole[i].hodnota
                         and pole[j].delky>maximum then
          begin
               maximum:=pole[j].delky;
               maxindex:=j;
          end;
     pole[i].delky:=maximum+1;
end;
```

Násobení matic

- Násobení matic není komutativní, ale je asociativní.
- Máme-li vynásobit několik matic, nemůžeme jejich pořadí měnit,
- můžeme součin všelijak závorkovat.
- Různá uzávorkování jsou různě výhodná.
- Které z nich je to nejvýhodnější?
- Předpokládáme, že máme abstraktní funkce zjistící ze zadaných rozměrů složitost součinu matic správných tvarů.