Anotace

- Memoizace, dynamické programování, tedy prostředky zvyšování efektivity rekurze a jejich implementace.

Problémy, které byly

- Přednášející jde tentokrát do S3,
- 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,
- problém batohu.

Přednášející do posluchárny!

Opakuji, pan přednášející se dostaví do posluchárny!

- Známe ze zimy: Přednášející jde po schodišti a buďto šlápne vždy na následující schod, nebo jeden překročí.
- Úloha vede na Fibonacciho čísla,
- v zimě jsme odvodili rekurzi $f(i) = f(i-1) + f(-2)$.
- ```
  static int f(int i)
  {   if(i<3) return i;
      return f(i-1)+f(i-2);
  }
  ```
- Jakou má toto řešení nevýhodu?
- Ano, exponenciální složitost. Co s problémem?
  ```

Co kazí složitost?

To, že funkce počítá pořád totéž!

- Funkce počítá pořád totéž
- exponenciálně velké číslo načítá po jedničkách (a dvojkách).
- Všimneme si, že hodnota funkce závisí jen na parametrech
- a vyrobíme si cache na výsledky.

Cache

spasí výpočet

```
static int[]c=new int[MOOOOC];
static int f(int i)
{   if(c[i]==0)
    {   if(i<3) c[i]=i;
        else c[i]=f(i-1)+f(-2);
    }
    return c[i];
}
```

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

```
static int paru,celkem=0;
static void pridej_zavorku(int lev,int prav)
{
      if(lev>prav)
            pridej_zavorku(lev,prav+1);
      if(paru>lev)
            pridej_zavorku(lev+1,prav);
      if((lev==prav)&&(paru==lev))
            celkem++;
}
```

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`:
- `static int[,]cache=new int [`MAX, MAX`]`.
- Pole na počátku inicializujeme nulami,
- je-li `cache[lev,prav]` $==0$, spustíme výpočet (výsledek ještě neznáme) a na konci funkce si ho uložíme do pole.
- Je-li `cache[lev,prav]` $!=0$, připočteme tolik platných uzávorkování.
- Rozdíl mezi variantou "rekurze" a "cachovaná rekurze" je rozdíl mezi nepoužitelným a dobrým algoritmem!

Nalezení všech Youngových tabulek

- Youngova tabulka je tabulka sestávající z řádků postupně nerostoucí délky.
- Samotný řádek je Youngova tabulka.
- Jeden sloupec taktéž.
- "Normální" tabulka $m \times n$ také.
- Nesmí se jen objevit širší řádek za užším.
- Pro zadané n vypište všechny Youngovy tabulky s n políčky.
- Nejde o nic jiného, než zjistit, kolika způsoby lze rozložit číslo na sčítance v nerostoucím pořadí.
- Zajímá nás jen jejich počet, nechceme tabulky vypisovat!

Youngovy tabulku – k řešení

- Jak úlohu vyřešit?
- Jako obvykle rekurzí. Budeme si pamatovat, kolik okének ještě zbývá a kolik jich nejvýše smíme dát do jednoho řádku.
  A zkusíme všechno od maxima, až k jedné.

Rekurzívní funkce:

```
static void pridej_radek(int kolik,int maximum)
{ int i;
     if(kolik==0) pocet++;
     else
          for(i=maximum i>=1;i--)
               pridej_radek(kolik-i,i);
}
```

Jaký je problém?

Pořád ten samý

(počítáme pořád to samé).

Youngovy tabulky

- 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 povolená šířka řádku nejvýše MAXIMUM:

```
static void rozloz(int kolik,int maximum);
{ int i,nazacatku;
     if(cache[kolik,maximum]!=0)
          pocet+=cache[kolik,maximum];
     else
     {     nazacatku=pocet;
          if(kolik==0) pocet++;
          else  for(i=maximum;i>=1;i--)
               rozloz(kolik-i,i);
          cache[kolik,maximum]=pocet-nazacatku;
     }
}
```

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:

```
static int kombin(int n,int k)
{     if ((k==0)||(k==n)) return 1;
      return kombin(n-1,k-1)+kombin(n-1,k);
}
static void Main(string[] a)
{     ... kombin(100,50);...
}
```

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

Pascalův trojúhelník

```
static const int MAX=100;
static int[,]cache=new int[MAX,MAX]; static int
kom(int n, int k)
{   if(cache[n,k]==0)
    {   if((k==0)||(k==n)) cache[n,k]=1;
        else cache[n,k]=kom(n-1,k-1)+kom(n-1,k);
    }
    return cache[n,k];
}
static void Main(string[]a)
{   Console.WriteLine(kom(Convert.toInt32(
        Console.ReadLine()),Convert.toInt32(
        Console.ReadLine()))); }
```

Nejdelší rostoucí podposloupnost

pro začátek rekurzí

- Nejdelší rostoucí podposloupnost někde končí.
- Zkusíme všechny kandidáty.
- Poslední prvek se pozná tak, že před ním je nejdelší rostoucí posloupnost končící před ním posledním prvkem menším (než je tento) $\Rightarrow$ jasný tah k rekurzi podle posledního prvku.
- Rekurze zkusí všechny podposlounposti.

Nejdelší rostoucí podposloupnost

s cachí

- Nejdelší rostoucí podposloupnost končící v i. prvku je pořád stejná.
- Nebudeme ji tudíž počítat pokaždé znovu, ale tento údaj nacachujeme.
- Vznikne cache parametrizovaná jednotlivými prvky.
- Tuto cache není třeba vyplňovat rekurzí.
- Délka nejdelší rostoucí podposloupnosti je určena maximální hodnotou v cachi.

Nejdelší rostoucí podposloupnost

- Utvoř posloupnost dvojic (pole struktur/objektů),
- 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;i<=n;i++)
{    maximum:=0; maxindex:=0;
     for(j=i-1;j>=1;j--)
     {    if(pole[j].hodnota<pole[i].hodnota
                         && pole[j].delky>maximum)
          {    maximum=pole[j].delky;
               maxindex=j;
          }
     pole[i].delky=maximum+1;
}
```

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.

Násobení matic

- Některé násobení provedeme jako poslední.
- To úlohu rozdělí na (nejvýše) dvě další instance.
- V každé (menší) instanci opět nějaké násobení provedeme jako poslední.
- Ideální podhoubí pro rekurzi!

Násobení matic

- static int slozitost(int od,int az_do)
- for(i=od;i<=az_do-1;i++)
- slozitost=slozitost(od,i)+
  slozitost(i+1,az_do)+samotne_nasobeni;

Opět budeme zbytečně násobit stále to samé.
Opět se toho zbavíme stejně.

Násobení matic

- int slozitost(int od,int az_do);
- if(cache[od,az_do]==0)
- for(i=od; i<=az_do-1;i++)
- cache[od,az_do]=slozitost(od,i)+
  slozitost(i+1,az_do)+samotne_nasobeni;
- return cache[od,az_do];

Metoda

- Všechny tyto problémy jsme řešili stejně:
- Navrhli jsme rekurzívní algoritmus,
- ten počítal často to samé, proto jsme mu doplnili cache.
- Pokud se podíváme pořádně, na správném místě cache najdeme výsledek.
- Potřebujeme tam tedy tu rekurzi?
- Ne a můžeme se jí zbavit za předpokladu, že zjistíme, jak vyplňovat cache, tedy tabulku.
- Tomu říkáme dynamické programování (probírá se na ADS).

Odstranění rekurze

- Přednášející jde do posluchárny:

```
a[1]=1;
a[2]=2;
for(i=3;i<=n;i++) a[i]=a[i-1]+a[i-2];
```

- Pascalův trojúhelník:

```
for(i=0;i<=n;i++)
        for(j=0;i<=i;j++)
                p(i,j)=p(i-1,j-1)+p(i-1,j);
```

  a okrajové případy zvlášť!

- Závorkování: Cvičení