Anotace

- Grafové algoritmy.
- Problémy řešitelné "vyplněním tabulky":
  - Nejdelší rostoucí podposloupnost,
  - počet korektních uzávorkování pomocí n párů závorek,
  - nalezení všech Youngových tabulek,
  - Pascalův trojúhelník,
  - pořadí násobení matic.

Souvislost grafu

Graf je souvislý právě když se lze z jednoho jeho vrcholu dostat do všech ostatních

```
for i in vrcholy do
      nenavstiv(i); {jeste jsme nic nenavstivili}
i:=startovni_vrchol;
fronta:={i};{na dosažitelné vrcholy}
while nonempty(fronta) do begin
      navstiv(i);
      fronta:=fronta+nenavstivene_sousedy(i);
end;
souvisly:=true;
for i in vrcholy do begin
      if nenavstiveny(i) then
            souvisly:=false;
```

Analýza algoritmu

- for-cyklus proběhne nejvýš n-krát.

Analýza algoritmu

- for-cyklus proběhne nejvýš n-krát.
- while-cyklus proběhne pro každý vrchol nejvýše jednou a podívá se na sousedy současného vrcholu.

Analýza algoritmu

- for-cyklus proběhne nejvýš n-krát.
- while-cyklus proběhne pro každý vrchol nejvýše jednou a podívá se na sousedy současného vrcholu.
- Složitost bude záviset na reprezentaci (jak rychle umíme najít sousedy vrcholu).

Analýza algoritmu

- for-cyklus proběhne nejvýš n-krát.
- while-cyklus proběhne pro každý vrchol nejvýše jednou a podívá se na sousedy současného vrcholu.
- Složitost bude záviset na reprezentaci (jak rychle umíme najít sousedy vrcholu).
- Složitost bude $\Omega(m)$ (na každou hranu musíme kouknout).

Analýza algoritmu

- for-cyklus proběhne nejvýš n-krát.
- while-cyklus proběhne pro každý vrchol nejvýše jednou a podívá se na sousedy současného vrcholu.
- Složitost bude záviset na reprezentaci (jak rychle umíme najít sousedy vrcholu).
- Složitost bude $\Omega(m)$ (na každou hranu musíme kouknout).
- Pokud máme seznam hran u vrcholu, bude složitost $O(m)$,

Analýza algoritmu

- for-cyklus proběhne nejvýš n-krát.
- while-cyklus proběhne pro každý vrchol nejvýše jednou a podívá se na sousedy současného vrcholu.
- Složitost bude záviset na reprezentaci (jak rychle umíme najít sousedy vrcholu).
- Složitost bude $\Omega(m)$ (na každou hranu musíme kouknout).
- Pokud máme seznam hran u vrcholu, bude složitost $O(m)$,
- pokud máme matici sousednosti, bude složitost $O(n^2)$,

Analýza algoritmu

- for-cyklus proběhne nejvýš n-krát.
- while-cyklus proběhne pro každý vrchol nejvýše jednou a podívá se na sousedy současného vrcholu.
- Složitost bude záviset na reprezentaci (jak rychle umíme najít sousedy vrcholu).
- Složitost bude $\Omega(m)$ (na každou hranu musíme kouknout).
- Pokud máme seznam hran u vrcholu, bude složitost $O(m)$,
- pokud máme matici sousednosti, bude složitost $O(n^2)$,
- máme-li matici incidence, složitost může být i $\Theta(mn^2)$.

Analýza algoritmu

- for-cyklus proběhne nejvýš n-krát.
- while-cyklus proběhne pro každý vrchol nejvýše jednou a podívá se na sousedy současného vrcholu.
- Složitost bude záviset na reprezentaci (jak rychle umíme najít sousedy vrcholu).
- Složitost bude $\Omega(m)$ (na každou hranu musíme kouknout).
- Pokud máme seznam hran u vrcholu, bude složitost $O(m)$,
- pokud máme matici sousednosti, bude složitost $O(n^2)$,
- máme-li matici incidence, složitost může být i $\Theta(mn^2)$.
- Při vhodné reprezentaci je tedy složitost $\Theta(m+n)$.

Poznámky

- Použijeme-li frontu, jedná se o algoritmus vlny (prohledávání do šířky) z jednoho vrcholu.

Poznámky

- Použijeme-li frontu, jedná se o algoritmus vlny (prohledávání do šířky) z jednoho vrcholu.
- Můžeme použít i zásobník, v tom případě se jedná o prohledávání do hloubky

Poznámky

- Použijeme-li frontu, jedná se o algoritmus vlny (prohledávání do šířky) z jednoho vrcholu.
- Můžeme použít i zásobník, v tom případě se jedná o prohledávání do hloubky
- Výhody a nevýhody:

Poznámky

- Použijeme-li frontu, jedná se o algoritmus vlny (prohledávání do šířky) z jednoho vrcholu.
- Můžeme použít i zásobník, v tom případě se jedná o prohledávání do hloubky
- Výhody a nevýhody:
- Při hledání do hloubky můžeme použít rekurzi a nemusíme si aktuálně pamatovat sousedy prohledávaného vrcholu.

Poznámky

- Použijeme-li frontu, jedná se o algoritmus vlny (prohledávání do šířky) z jednoho vrcholu.
- Můžeme použít i zásobník, v tom případě se jedná o prohledávání do hloubky
- Výhody a nevýhody:
- Při hledání do hloubky můžeme použít rekurzi a nemusíme si aktuálně pamatovat sousedy prohledávaného vrcholu.
- Hledání do šířky navštíví vrchol po nejkratší cestě.

Hledání kružnice

Graf má kružnici, pokud se při prohledávání grafu vrátíme do už navštíveného vrcholu.

```
kruznice:=false; {zatim zadna}
for i in vrcholy do nenavstiv(i);
for i in vrcholy do
    if nenavstiveny(i) then{nova komponenta}
    begin fronta:={i};
        while(nonempty(fronta)) do
        begin prvni prvek vyrad z fronty a prirad do i;
            if(navstiveny(i)) then
                kruznice:=true;
            else  for j in sousedy(i) do
                begin fronta:=fronta+{j};
                    smaz_hranu({i,j});
                end;
    end; end;
```

Strom

- Budeme testovat, zda je graf souvislý a bez kružnic, tedy použijeme oba předešlé algoritmy.

Strom

- Budeme testovat, zda je graf souvislý a bez kružnic, tedy použijeme oba předešlé algoritmy.
- Anebo budeme testovat, zda je bez kružnic a má jen jednu komponentu.

Strom

- Budeme testovat, zda je graf souvislý a bez kružnic, tedy použijeme oba předešlé algoritmy.
- Anebo budeme testovat, zda je bez kružnic a má jen jednu komponentu.
- Anebo otestujeme souvislost (či kružnice) a správný počet hran.

Nejkratší cesta

Hledáme-li v grafu nejkratší cestu z vrcholu do vrcholu, záleží na reprezentaci:

- Prolezeme graf do šířky (máme-li seznam vrcholů a hran),

Theorem

V matici ${A_G}^k$ hodnota na pozici i, j určuje počet sledů délky k z vrcholu i do vrcholu j.

Corollary

V matici $(A_G + I)^k$ určuje hodnota na pozici i, j počet sledů délky nejvýše k z i do j.

Nejkratší cesta

Hledáme-li v grafu nejkratší cestu z vrcholu do vrcholu, záleží na reprezentaci:

- Prolezeme graf do šířky (máme-li seznam vrcholů a hran),
- mocníme matici sousednosti, pokud máme maticovou reprezentaci.

Theorem

V matici ${A_G}^k$ hodnota na pozici i, j určuje počet sledů délky k z vrcholu i do vrcholu j.

Corollary

V matici $(A_G + I)^k$ určuje hodnota na pozici i, j počet sledů délky nejvýše k z i do j.

Dijkstrův algoritmus

Hledá nejkratší cestu z daného vrcholu (do všech ostatních)

Vstup: Graf s `nezáporně` ohodnocenými hranami.

- Udržujeme "frontu" vrcholů setříděnou podle dosud nejkratší cesty do nich.
- Na začátku zinicializujeme vzdálenosti do všech vrcholů kromě startovního nekonečnem (tedy dost vysokou hodnotou) a vzdálenost do startu nulou.
- Startovní vrchol přidáme do "fronty" dosažitelných vrcholů.
- Vrchol, do kterého se dostaneme nejkratší cestou z fronty odstraníme a pokusíme se cestu jdoucí z něj rozšířit do jeho sousedů.
- Toto opakuj, dokud je "fronta" neprázdná.

Rozšíření cesty

Rozšíření cesty vypadá tak, že pro vrchol v ve vzdálenosti $d(v)$ zkusíme pro každou hranu $\{v, w\}$, zda

$$d(w) > d(v) + delka(\{v, w\}).$$

Pokud ano, $d(w) := d(v) + delka(\{v, w\})$ a oprav pozici vrcholu w ve "frontě".

Analýza

- Algoritmus je popsán a dokázán v mnoha knihách (a skriptech), kupř. Kapitoly z diskrétní matematiky nebo Algebraické algoritmy (Kučera, Nešetřil)...

Analýza

- Algoritmus je popsán a dokázán v mnoha knihách (a skriptech), kupř. Kapitoly z diskrétní matematiky nebo Algebraické algoritmy (Kučera, Nešetřil)...
- Konečnost: V každé iteraci odstraníme z "fronty" jeden vrchol, který se v ní už neobjeví
  (protože mezi vrcholy ve frontě měl nejmenší vzdálenost od startu a vzdálenosti jsou nezáporné).

Analýza

- Algoritmus je popsán a dokázán v mnoha knihách (a skriptech), kupř. Kapitoly z diskrétní matematiky nebo Algebraické algoritmy (Kučera, Nešetřil)...
- Konečnost: V každé iteraci odstraníme z "fronty" jeden vrchol, který se v ní už neobjeví
  (protože mezi vrcholy ve frontě měl nejmenší vzdálenost od startu a vzdálenosti jsou nezáporné).
- Parciální správnosti pomůže invariant:
  V každém kroku evidujeme nejkratší cesty ze startu používající pouze vrcholy již odstraněné z "fronty".

Analýza

- Algoritmus je popsán a dokázán v mnoha knihách (a skriptech), kupř. Kapitoly z diskrétní matematiky nebo Algebraické algoritmy (Kučera, Nešetřil)...
- Konečnost: V každé iteraci odstraníme z "fronty" jeden vrchol, který se v ní už neobjeví
  (protože mezi vrcholy ve frontě měl nejmenší vzdálenost od startu a vzdálenosti jsou nezáporné).
- Parciální správnosti pomůže invariant:
  V každém kroku evidujeme nejkratší cesty ze startu používající pouze vrcholy již odstraněné z "fronty".
- Z invariantu plyne korektnost.

Analýza

- Algoritmus je popsán a dokázán v mnoha knihách (a skriptech), kupř. Kapitoly z diskrétní matematiky nebo Algebraické algoritmy (Kučera, Nešetřil)...
- Konečnost: V každé iteraci odstraníme z "fronty" jeden vrchol, který se v ní už neobjeví
  (protože mezi vrcholy ve frontě měl nejmenší vzdálenost od startu a vzdálenosti jsou nezáporné).
- Parciální správnosti pomůže invariant:
  V každém kroku evidujeme nejkratší cesty ze startu používající pouze vrcholy již odstraněné z "fronty".
- Z invariantu plyne korektnost.
- Jde jen o modifikovaný algoritmus vlny, tedy hledání do šířky!

Analýza

- Algoritmus je popsán a dokázán v mnoha knihách (a skriptech), kupř. Kapitoly z diskrétní matematiky nebo Algebraické algoritmy (Kučera, Nešetřil)...
- Konečnost: V každé iteraci odstraníme z "fronty" jeden vrchol, který se v ní už neobjeví
  (protože mezi vrcholy ve frontě měl nejmenší vzdálenost od startu a vzdálenosti jsou nezáporné).
- Parciální správnosti pomůže invariant:
  V každém kroku evidujeme nejkratší cesty ze startu používající pouze vrcholy již odstraněné z "fronty".
- Z invariantu plyne korektnost.
- Jde jen o modifikovaný algoritmus vlny, tedy hledání do šířky!
- Složitost významně závisí na reprezentaci grafu a na reprezentaci "fronty"!

Poznámky

- Pokud jde o graf neohodnocený (délka všech hran je 1), potom se z Dijkstrova algoritmu stane obyčejný algoritmus vlny.

Poznámky

- Pokud jde o graf neohodnocený (délka všech hran je 1), potom se z Dijkstrova algoritmu stane obyčejný algoritmus vlny.
- Aplikace grafových algoritmů: Zde začíná teoretická informatika. :-)

Poznámky

- Pokud jde o graf neohodnocený (délka všech hran je 1), potom se z Dijkstrova algoritmu stane obyčejný algoritmus vlny.
- Aplikace grafových algoritmů: Zde začíná teoretická informatika. :-)
- Příklady využití grafových algoritmů:

Poznámky

- Pokud jde o graf neohodnocený (délka všech hran je 1), potom se z Dijkstrova algoritmu stane obyčejný algoritmus vlny.
- Aplikace grafových algoritmů: Zde začíná teoretická informatika. :-)
- Příklady využití grafových algoritmů:
- Thesseus a Minotaurus,

Poznámky

- Pokud jde o graf neohodnocený (délka všech hran je 1), potom se z Dijkstrova algoritmu stane obyčejný algoritmus vlny.
- Aplikace grafových algoritmů: Zde začíná teoretická informatika. :-)
- Příklady využití grafových algoritmů:
- Thesseus a Minotaurus,
- Král (či jiné figurky) na šachovnicích různých tvarů s různě pozakazovanými políčky,

Poznámky

- Pokud jde o graf neohodnocený (délka všech hran je 1), potom se z Dijkstrova algoritmu stane obyčejný algoritmus vlny.
- Aplikace grafových algoritmů: Zde začíná teoretická informatika. :-)
- Příklady využití grafových algoritmů:
- Thesseus a Minotaurus,
- Král (či jiné figurky) na šachovnicích různých tvarů s různě pozakazovanými políčky,
- ...

Další grafové problémy/algoritmy

- Minimální kostra, aneb jak natáhnout elektrické vedení?
- Eulerovský graf, aneb lze všechny hrany grafu nakreslit jedním tahem (abychom žádnou hranou neprojeli dvakrát)?
- Hamiltonskost, aneb kružnice, která navštíví všechny vrcholy (každý právě jednou),
- Klika, aneb maximální úplný podgraf,
- Barevnost, aneb jaký je nejmenší počet barev takový, abychom mohli obarvit vrcholy grafu tak, že sousední vrcholy nedostanou stejnou barvu?
- Hranová barevnost, aneb jaký je nejmenší počet barev takový, abychom mohli obarvit hrany grafu tak, aby žádná dvojice hran přiléhající ke společnému vrcholu neměla stejnou barvu?
- ...

Nejdelší rostoucí podposloupnost

- Jak řešit?

Nejdelší rostoucí podposloupnost

- Jak řešit?
- První nápad: Rekurze. Ale podle čeho?

Nejdelší rostoucí podposloupnost

- Jak řešit?
- První nápad: Rekurze. Ale podle čeho?
- Podle délky vstupní posloupnosti.

Nejdelší rostoucí podposloupnost

- Jak řešit?
- První nápad: Rekurze. Ale podle čeho?
- Podle délky vstupní posloupnosti.
- `function zkus_posledni(i:integer):integer;`

Nejdelší rostoucí podposloupnost

- Jak řešit?
- První nápad: Rekurze. Ale podle čeho?
- Podle délky vstupní posloupnosti.
- `function zkus_posledni(i:integer):integer;`
- Funkce zkusí najít nejdelší rostoucí posposloupnost končící na pozici `i`.

Nejdelší rostoucí podposloupnost

- Jak řešit?
- První nápad: Rekurze. Ale podle čeho?
- Podle délky vstupní posloupnosti.
- `function zkus_posledni(i:integer):integer;`
- Funkce zkusí najít nejdelší rostoucí posposloupnost končící na pozici `i`.
- Nevýhoda: Pořád počítáme to samé. Co s tím?

Nejdelší rostoucí podposloupnost

- Jak řešit?
- První nápad: Rekurze. Ale podle čeho?
- Podle délky vstupní posloupnosti.
- `function zkus_posledni(i:integer):integer;`
- Funkce zkusí najít nejdelší rostoucí posposloupnost končící na pozici `i`.
- Nevýhoda: Pořád počítáme to samé. Co s tím?
- Vytvoříme cache na výsledky.

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;
```

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(lev,prav:integer);
begin
     if lev>prav then
          pridej_zavorku(lev,prav+1);
     if paru>lev then
          pridej_zavorku(lev+1,prav);
     if (lev=prav) and (paru=lev) then
          inc(celkem);
end;
```

Počet korektních uzávorkování

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

Jaký problém má toto řešení?

Počet korektních uzávorkování

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

Jaký problém má toto řešení?

Počítáme pořád to samé!

Závorkování – cache

- Jak z pasti? Položíme si správnou otázku:

Závorkování – cache

- Jak z pasti? Položíme si správnou otázku:
- Záleží výsledek funkce `pridej_zavorku` na něčem jiném, než na parametrech `levych` a `pravych`?

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`:

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`.

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,

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[lev,prav] =0`, spustíme výpočet (výsledek ještě neznáme) a na konci funkce si ho uložíme do pole.

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[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.

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[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:

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

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

Youngovy tabulky

- Jak z pasti tentokrát?

Youngovy tabulky

- Jak z pasti tentokrát?
- Stejně jako u závorkování:

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:

```
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,

Pascalův trojúhelník

- Obsahuje kombinační čísla,
- n-tý řádek konkrétně obsahuje hodnoty $\binom{n}{0}, \binom{n}{1}, \ldots, \binom{n}{n}$,

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 $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$,

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
```

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.

Násobení matic

- Některé násobení provedeme jako poslední.
- To úlohu rozdělí na (nejvýše) dvě další instance.

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.

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

- function slozitost(od,az_do:integer):longint;
- for i:=od to az_do-1 do begin
- 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

```
function slozitost(od,az_do:integer):longint;
if cache[od,az_do]=0 then
    for i:=od to az_do-1
        cache[od,az_do]:=slozitost(od,i)+
            slozitost(i+1,az_do)+samotne_nasobeni;
slozitost:=cache[od,az_do];
```

Násobení matic – poučení

- Tomuto říkáme *dynamické programování.*
- Zpravidla implementujeme pouze fázi vyplňování tabulky.
- Na tom je nepříjemné určovat, odkud vyplňování zahájit.
- Není tudíž špatné navrhnout si aspoň na počátku rekurzívní řešení a až pak rekurzi "rozbít":
- `cache[od,az_do]:=min{cache[od,i]+cache[i,az_do]+ samotne_nasobeni}`
- což stačí udělat cyklem.

Konec

Děkuji za pozornost...