Anotace

- Síla předvýpočtu
- Rekurze podruhé
- Pole jako parametr
- Definice vlastních datových typů (výčtové datové typy)
- Konstrukce `case ... of ...`
- Základní třídící algoritmy
- Direktivy překladače
- Soubory (textové)

Maximální jedničková podmatice

Problém: V matici tvaru $m \times n$ vyplněné nulami a jedničkami máme najít největší podmatici (souvislou) obsahující pouze jedničky.

Naivní algoritmus

- Pro každý možný levý horní a pravý dolní roh prohlédni vnitřek matice.

Naivní algoritmus

- Pro každý možný levý horní a pravý dolní roh prohlédni vnitřek matice.
- Algoritmus funguje, ale s jakou složitostí?

Naivní algoritmus

- Pro každý možný levý horní a pravý dolní roh prohlédni vnitřek matice.
- Algoritmus funguje, ale s jakou složitostí?
- $\Theta(mn)$ levých horních rohů, $\Theta(mn)$ pravých dolních rohů, $\Theta(mn)$ prvků uvnitř (proč?), celkem tedy $\Theta(m^3n^3)$.

Naivní algoritmus

- Pro každý možný levý horní a pravý dolní roh prohlédni vnitřek matice.
- Algoritmus funguje, ale s jakou složitostí?
- $\Theta(mn)$ levých horních rohů, $\Theta(mn)$ pravých dolních rohů, $\Theta(mn)$ prvků uvnitř (proč?), celkem tedy $\Theta(m^3n^3)$.
- Nápady na zlepšení?

Předvýpočet

- Pro každou jedničku si spočítáme, kolik jedniček leží bezprostředně pod ní (tedy v řadě nepřerušené nulou).

Předvýpočet

- Pro každou jedničku si spočítáme, kolik jedniček leží bezprostředně pod ní (tedy v řadě nepřerušené nulou).
- Zkoušíme každou matici identifikovat pomocí levého a pravého horního rohu:

Předvýpočet

- Pro každou jedničku si spočítáme, kolik jedniček leží bezprostředně pod ní (tedy v řadě nepřerušené nulou).
- Zkoušíme každou matici identifikovat pomocí levého a pravého horního rohu:
  - Ke kandidátu na levý horní roh zkoušej všechny možnosti pravého horního rohu (v jeho řadě).

Předvýpočet

- Pro každou jedničku si spočítáme, kolik jedniček leží bezprostředně pod ní (tedy v řadě nepřerušené nulou).
- Zkoušíme každou matici identifikovat pomocí levého a pravého horního rohu:
    - Ke kandidátu na levý horní roh zkoušej všechny možnosti pravého horního rohu (v jeho řadě).
    - Tyto nesmějí být odděleny nulou (tedy "žijí" v souvislém bloku jedniček),

Předvýpočet

- Pro každou jedničku si spočítáme, kolik jedniček leží bezprostředně pod ní (tedy v řadě nepřerušené nulou).
- Zkoušíme každou matici identifikovat pomocí levého a pravého horního rohu:
    - Ke kandidátu na levý horní roh zkoušej všechny možnosti pravého horního rohu (v jeho řadě).
    - Tyto nesmějí být odděleny nulou (tedy "žijí" v souvislém bloku jedniček),
    - Jelikož máme posčítané počty jedniček směrem dolů, stačí jako druhý rozměr vzít minimum z těchto posčítaných jedniček.

Předvýpočet

- Pro každou jedničku si spočítáme, kolik jedniček leží bezprostředně pod ní (tedy v řadě nepřerušené nulou).
- Zkoušíme každou matici identifikovat pomocí levého a pravého horního rohu:
  - Ke kandidátu na levý horní roh zkoušej všechny možnosti pravého horního rohu (v jeho řadě).
  - Tyto nesmějí být odděleny nulou (tedy "žijí" v souvislém bloku jedniček),
  - Jelikož máme posčítané počty jedniček směrem dolů, stačí jako druhý rozměr vzít minimum z těchto posčítaných jedniček.
  - Zbytek je násobení a porovnávání.

Předvýpočet

- Pro každou jedničku si spočítáme, kolik jedniček leží bezprostředně pod ní (tedy v řadě nepřerušené nulou).
- Zkoušíme každou matici identifikovat pomocí levého a pravého horního rohu:
  - Ke kandidátu na levý horní roh zkoušej všechny možnosti pravého horního rohu (v jeho řadě).
  - Tyto nesmějí být odděleny nulou (tedy "žijí" v souvislém bloku jedniček),
  - Jelikož máme posčítané počty jedniček směrem dolů, stačí jako druhý rozměr vzít minimum z těchto posčítaných jedniček.
  - Zbytek je násobení a porovnávání.
- Složitost: Předvýpočet $O(mn)$, výpočet $O(m^2n)$.

Lze to ještě urychlit?

Ku podivu ano – a ještě k tomu znovu předvýpočtem:

- Spočítej blok jedniček směrem dolů, ($\rightarrow B$)

Lze to ještě urychlit?

Ku podivu ano – a ještě k tomu znovu předvýpočtem:

- Spočítej blok jedniček směrem dolů, ($\rightarrow B$)
- spočítej blok jedniček směrem nahoru, ($\rightarrow C$)

Lze to ještě urychlit?

Ku podivu ano – a ještě k tomu znovu předvýpočtem:

- Spočítej blok jedniček směrem dolů, ($\to B$)
- spočítej blok jedniček směrem nahoru, ($\to C$)
- zkusíme hledat "levý kritický konec", tedy místo, kde matice "najede na nulu", tedy $a_{i,j} = 1$ a $a_{i,j-1} = 0$ nebo $j = 1$ ($a_{i,j-1}$ leží mimo matici).

Lze to ještě urychlit?

Ku podivu ano – a ještě k tomu znovu předvýpočtem:

- Spočítej blok jedniček směrem dolů, ($\rightarrow B$)
- spočítej blok jedniček směrem nahoru, ($\rightarrow C$)
- zkusíme hledat "levý kritický konec", tedy místo, kde matice "najede na nulu", tedy $a_{i,j} = 1$ a $a_{i,j-1} = 0$ nebo $j = 1$ ($a_{i,j-1}$ leží mimo matici).
- Zkus všechny možné kandidáty na pravý okraj (v příslušném řádku).

Analýza složitosti

- Předvýpočty (stavba matic B a C): $O(mn)$,
- ačkoliv se zdá, že složitost výpočtu bude stejná jako dříve, není tomu tak,
- protože každý prvek matice zkoušíme jako kandidát na "pravý okraj" jen jednou!
- Celkem tedy $O(mn)$; jelikož složitost problému je $\Omega(mn)$, máme algoritmus (až na konstantu) optimální.

Rekurze podruhé

- Rekurze je metoda řešení problému spočívající v tom, že problém "zmenšíme"a z řešení menší instance odvodíme řešení větší instance.

Rekurze podruhé

- Rekurze je metoda řešení problému spočívající v tom, že problém "zmenšíme"a z řešení menší instance odvodíme řešení větší instance.
- Příklady: faktoriál, přednášející jde do M1...

Rekurze podruhé

- Rekurze je metoda řešení problému spočívající v tom, že problém "zmenšíme"a z řešení menší instance odvodíme řešení větší instance.
- Příklady: faktoriál, přednášející jde do M1...
- Dnes: Výpis všech čísel v zadané číselné soustavě (o dané délce),

Rekurze podruhé

- Rekurze je metoda řešení problému spočívající v tom, že problém "zmenšíme" a z řešení menší instance odvodíme řešení větší instance.
- Příklady: faktoriál, přednášející jde do M1...
- Dnes: Výpis všech čísel v zadané číselné soustavě (o dané délce),
- Problém batohu

Hlavní program

```
program q;
const MAX=10;
var cif,zakl,pocet:integer;
      pole:array[1..MAX] of integer;
begin
      write('Zadej pocet cifer: ');
      readln(cif);
      if(cif>MAX) then
            halt;{Moc dlouhe cislo}
      write('Zadej zaklad soustavy: ');
      readln(zakl);
      if zakl>10 then
            halt;{moc vysoky zaklad soustavy}
      vypln(1);
end.
```

Jádro rekurze

```
procedure vypln(odkud:integer);
var i:integer;
begin
      if(odkud<=cif) then
            for i:=0 to zakl-1 do
            begin
                  pole[odkud]:=i;
                  vypln(odkud+1);
            end
      else vypis;
end;
```

Procedura vypis

```
procedure vypis;
var i:integer;
start:boolean;
begin
      start:=true;
      for i:=1 to cif do
            if((not start) or (pole[i]<>0)) then
            begin
                  start:=false;
                  write(pole[i]);
            end;
      if start then write(0);
      writeln;
end;
```

Problém batohu

- Vykrademe klenotnictví a chceme si odnést co největší lup – krademe jen zlato, tedy hmotnost odpovídá ceně.
- Problém je těžký (NP-úplný) pro neomezené hmotnosti,
- pokud se hmotnosti jednotlivých předmětů dostatečně liší, lze problém řešit polynomiálně,
- tedy například jsou-li hmotnosti celočíselné.

Problém batohu

- Jak vyřešit rekurzí?
- Budeme vždycky zkoušet: Buďto prvek přidáme, nebo ne.
- Tedy načteme vstup (do pole), začneme od první položky a každou postupně zkusíme:
  1. přidat,
  2. nepřidat.

Hlavní program

```
program knapsack;
const MAX=10;
var kap,pocet,i:integer;
      polozky:array[0..MAX] of integer;
      pridano:array[0..MAX] of boolean;
begin {nacteni}
      readln(kap); readln(pocet);
      polozky[0]:=0;
      for i:=1 to pocet do
      begin
            readln(polozky[i]);
            pridano[i]:=false;
      end;
      pridej(0);
end.
```

procedure pridej(odkud:integer);

```
var i:integer;
begin if kap>0 then
            for i:=odkud+1 to pocet do
            begin pridano[i]:=true;
                  kap:=kap-polozky[i];
                  pridej(i);
                  kap:=kap+polozky[i];
                  pridano[i]:=false;
            end
      else if kap=0 then
            begin for i:=1 to MAX do
                    if pridano[i] then write(i,', ');
                  writeln;
            end;
end;
```

Jak předat pole jako parametr funkci?

- V klasickém Pascalu je třeba definovat vlastní typ (demonstrovat, proč nejde naivní přístup).
- Turbo Pascal (a Free Pascal) umí tzv. open arrays.

Definice vlastního typu:

- Klíčové slovo `type` umožňuje definovat vlastní typ.
- triviální použití: `type int=integer;`
- použití: `type x=array[1..10] of integer;`

Příklad

```
program nic;
type pole=array[1..10] of integer;
var p:pole;
procedure vypis(a:pole);
var i:integer;
begin
    for i:=1 to 10 do
        writeln(a[i]);
end;
begin
...vypis(p);
end.
```

Open arrays

- V Turbo Pascalu i Free Pascalu,
- uvedeme, že parametr je pole nějakého typu, ale neřekneme meze.
- Příklad: `procedure vypis(a:array of integer);`
- Je předáno jako pole od 0 do N.
- Velikost můžeme zjistit pomocí funkce `high`.

Příklad na open arrays

```
procedure vypis(a:array of integer);
var i:integer;
begin
      for i:=0 to high(a) do
            writeln(a[i]);
end;
```

Další využití tvorby vlastních typů

Chceme počítat dny v týdnu. Jak to uděláme?

- Očíslujeme si dny takto: Pondělí=1, Úterý=2,...

Další využití tvorby vlastních typů

Chceme počítat dny v týdnu. Jak to uděláme?

- Očíslujeme si dny takto: Pondělí=1, Úterý=2,...
- Jenže já to přečísluju: Pondělí=0, Úterý=1,...

Další využití tvorby vlastních typů

Chceme počítat dny v týdnu. Jak to uděláme?

- Očíslujeme si dny takto: Pondělí=1, Úterý=2,...
- Jenže já to přečísluju: Pondělí=0, Úterý=1,...
- Přijde američan a očísluje: Neděle=1, Pondělí=2,...

Další využití tvorby vlastních typů

Chceme počítat dny v týdnu. Jak to uděláme?

- Očíslujeme si dny takto: Pondělí=1, Úterý=2,...
- Jenže já to přečísluju: Pondělí=0, Úterý=1,...
- Přijde američan a očísluje: Neděle=1, Pondělí=2,...
- ... anebo Neděle=0, Pondělí=1,...

Další využití tvorby vlastních typů

Chceme počítat dny v týdnu. Jak to uděláme?

- Očíslujeme si dny takto: Pondělí=1, Úterý=2,...
- Jenže já to přečísluju: Pondělí=0, Úterý=1,...
- Přijde američan a očísluje: Neděle=1, Pondělí=2,...
- ... anebo Neděle=0, Pondělí=1,...
- Proto raději uděláme zvláštní typ indexovaný dny v týdnu a čísla necháme na překladači.

Výčtový datový typ

- Definujeme v sekci `type`,
- jednotlivé hodnoty klademe do závorek a oddělujeme čárkou.
- Příklad: `type dnyvtydnu=(pondeli,utery,streda, ctvrtek,patek, sobota,nedele);`
- Anebo definujeme přímo proměnnou tohoto typu: `var kal:(pondeli,utery,streda,ctvrtek,patek, sobota,nedele);`

Příklad

- Chceme vyrobit jednoduchý "kalendář" na rok 2010, tedy vypsat datum a údaj o dni v týdnu.
- Pro jednoduchost předpokládejme, že každý měsíc má 30 dnů...
- Zdrojový kód je na webu (kam.mff.cuni.cz/ perm/programovani/enum.pas).
- V příkladu vidíme, že funkce `write` neumí vypsat příslušné názvy, bylo by proto pěkné v závislosti na čísle dne v týdnu vypsat příslušný text. Nápady?
- Buďto mnoho klauzulí `if`, nebo: `case proměnná of ...`

Konstrukce `case ... of ...`

- Umožňuje vytvořit mnoho větví programu v závislosti na obsahu jedné proměnné.
- Syntax:

  `case` jméno proměnné `of`
      `hodnota1:` příkaz nebo blok
      `hodnota2:` příkaz nebo blok
      `else` příkaz nebo blok
  `end;`
- Provede se jen větev označená aktuální hodnotou proměnné, `else`-větev je pro ostatní (explicitně neuvedené) případy.
- Klauzule `else` nemusí být přítomna!
- Je-li poslední klauzule blok, jde `end` dvakrát po sobě (první uzavře blok posledních příkazů, druhý uzavře blok `case`).

Příklad – kalendář s jmény dnů

je na adrese
`kam.mff.cuni.cz/ perm/programovani/case_of.pas`.

Problém třídění – motivace

- Máme načtena data (například čísla),
- chceme je zpracovat například v rostoucím pořadí.
- Jak to udělat? Setřídíme, zpracujeme.
- Předpokládejme, že data jsou v poli.

Problém třídění – jednoduché třídící algoritmy

- Bublinkové třídění (BubbleSort),
- zatřiďování alias třídění přímým vkládáním (InsertSort),
- třídění výběrem (SelectSort),
- QuickSort.

Bublinkové třídění

- Geometrická interpretace:
  Bublinky v kapalině jdou zpravidla vzhůru
- Myšlenka: Porovnáváme po sobě jdoucí čísla (ve smyslu sousední v zadaném poli) od prvního k poslednímu, jsou-li v nesprávném pořadí, prohodíme je.
- Prvky "probublávají""správným"směrem.
- Opakujeme bublání, dokud se prohazuje.

Bubblesort v pseudokódu

- `prohazovalose:=true;`
- `while prohazovalose:=true do`
  `begin`
  - `for i:=1 to pocet - 1 do`
    `begin`
    - `prohazovalose:=false;`
    - `if pole[i]>pole[i+1] then`
      ```
      begin prohod(pole[i],pole[i+1]);
            prohazovalose:=true;
      end;
      ```
  - `end;`
- `end;`

Složitost bubble-sortu

- Kolikrát se provede vnější `while`-cyklus?

Složitost bubble-sortu

- Kolikrát se provede vnější `while`-cyklus?
- V *i*-tém kroku dojede *i*-tý největší na své místo!

Složitost bubble-sortu

- Kolikrát se provede vnější `while`-cyklus?
- V *i*-tém kroku dojede *i*-tý největší na své místo!
- Stačí tedy *n*-krát probublat, jedno probublání porovná po sobě jdoucí dvojice, tedy má složitost lineární.

Složitost bubble-sortu

- Kolikrát se provede vnější `while`-cyklus?
- V *i*-tém kroku dojede *i*-tý největší na své místo!
- Stačí tedy *n*-krát probublat, jedno probublání porovná po sobě jdoucí dvojice, tedy má složitost lineární.
- Složitost `BubbleSortu` je tedy $O(n^2)$.

Složitost bubble-sortu

- Kolikrát se provede vnější `while`-cyklus?
- V i-tém kroku dojede i-tý největší na své místo!
- Stačí tedy n-krát probublat, jedno probublání porovná po sobě jdoucí dvojice, tedy má složitost lineární.
- Složitost `BubbleSortu` je tedy $O(n^2)$.
- Implementaci, kdy se střídavě bublá z jedné strany na druhou a z druhé na první se říká `ShakeSort` a funguje pro něj stejný odhad složitosti.

Třídění přímým výběrem a zatřiďováním

Přímý výběr:

- Opakuj, dokud není tříděné pole prázdné:
- Najdi v poli minimum a přesuň ho na konec setříděného pole.

Zatřiďování:

- Opakuj, dokud není tříděné pole prázdné:
- Vyjmi z něj první prvek a zatřiď do cílového pole, tedy: najdi pozici, kam prvek patří, přidej ho tam a zbytek setříděného pole posuň (o jedna dál).

Analýza složitosti: $n$krát opakujeme proces, který trvá nejvýše n kroků, tedy také $O(n^2)$.

Quicksort – třídění za pomoci rekurze – idea:

- Pokud třídíme jedno číslo, nic nedělej (posloupnost je setříděna),
  tedy vrať posloupnost tak, jak jsme ji dostali.
- V poli POLE vyber jeden prvek (dále pivot).
- Rozděl POLE na pole A obsahující prvky menší než pivot
- a na pole B obsahující prvky větší nebo rovné pivotu.
- Pomocí sebe sama setřiď pole A,
- pomocí sebe sama setřiď pole B,
- Vypiš: pole A, pivot, pole B.

Directivy překladače

- Překladač kontroluje plno věcí, například:
- zda nekoukáme za konec pole,
- zda nám nepřetekl zásobník,
- anebo zda nenastala chyba na vstupu/výstupu...
- Většinou je užitečné mít kontroly zapnuté, někdy však "víme, co děláme".
- V tom případě můžeme na nezbytnou dobu chování překladače změnit pomocí tzv. *direktiv překladače.*
- Direktivy vypadají jako komentář, tedy jsou ve složených závorkách, ovšem začínají znakem string ($), jméno je zpravidla 1znakové a následuje přepínač $+/-$.

Direktivy překladače:

- Příklad: $\{\$R-\}$ – vypni *range-checking*.
- Nejdůležitější:
  - $\$Q$ – overflow-checking,
  - $\$R$ – range-checking,
  - $\$I$ – test vstupu a výstupu,
  - Úplný seznam najdete v helpu (některé direktivy se liší podle překladačů).

Soubory a práce s nimi

- V tomto semestru budou pouze soubory textové. Ačkoliv existují i soubory binární, ty budou předmětem výuky až v létě!
- Textový soubor ovládáme pomocí proměnné typu `Text`.
- Příslušnou proměnnou "napojíme" na daný soubor pomocí funkce `Assign`,
- otevřeme pomocí funkce `Reset`, `Rewrite` nebo `Append`,
- soubor čteme pomocí funkce `Read` (resp. `Readln`), které jako první argument předáme příslušnou proměnnou typu `Text`, zapisujeme analogicky pomocí funkcí `Write` a `Writeln`.
- Nakonec soubor uzavřeme pomocí funkce `Close`.

Práce se souborem – syntax (1)

- `var f:Text;`
- `Assign(f,'soubor.txt');` – asociuj proměnnou `f` se souborem `soubor.txt`.
- `Reset(f);` – otevři soubor `f` (pro čtení).
- `Rewrite(f);` – otevři soubor `f` a jeho dosavadní obsah znič.
- `Append(f);` – otevři soubor `f` pro zápis za jeho dosavadní konec.

Práce se souborem – syntax (2)

- `Writeln(f,'Zapiseme text do souboru');` – zapiš do souboru příslušný text.
- `Read(f,a);` – načti ze souboru proměnnou a.
- `Close(f);` – uzavři soubor (už s ním nebudeme pracovat).
- `eof(f);` – funkce, která sdělí, zda jsme na konci souboru.
- `eof;` – funkce oznamující konec standardního vstupu (z klávesnice).
- Existuje mnoho dalších funkcí jako `Rename`, `Erase`, ...

Potíže se soubory

- Často se stane, že otevíraný soubor neexistuje.
- Tato událost vyvolá input/output error.
- Nechceme-li při zapnuté této direktivě překladače soubor zničit (pomocí `Rewrite`, které sice neexistující soubor založí, ale existující přemaže), použijeme direktivu překladače a zda nastala chyba zjistíme pomocí funkce `IOResult`.

Příklad

```
Assign(f,'soubor.txt');
{$I−} {Vypni test na vstupně-výstupní chyby}
Reset(f);
{$I+} {Zapni test vstupně-výstupních chyb}
if IOResult<>0 then
begin writeln('Chyba!'); halt;
end;
while not eof(f) do begin
      readln(f,s);
      writeln(s);
end;
```

Pozor, `IOResult` je funkce a po zavolání ztratí hodnotu, nelze ji tedy číst opakovaně a její výsledek je případně třeba uložit do proměnné!