Jarné sústredenie UFO a Prask

17. 4. – 23. 4. 2016, Lúčka-Potoky

Zborník – informatika

Obsah

Prednášky

Princípy počítačov

Prednášajúci: Syseľ

Abstrakt

Rozumieť tomu, ako funguje počítač, od elektrického impulzu pri kliknutí myši až po vykreslenie explodujúcej hlavy zombíka, ktorého ste práve zastrelili, je celkom psycho. V tejto prednáške sa vám však pokúsim predstaviť aspoň základy týchto princípov, ktoré sú nevyhnutné na to, aby ste mohli programovať.

Jazyky, kompilácia a interpretácia

Od sedemnásteho storočia, kedy sa tkáčske stroje programovali dierami na správnych miestach v štítkoch, ubehlo veľa času. Dnes je proces vyrábania programov trochu komplikovanejší. Ako a prečo?

Počítač je stroj, ktorý spraví miliardy sčítaní, násobení a presunutí čísel za sekundu. Každá z týchto **operácií** je vykonávaná nejakou súčiastkou. (Áno, existuje súčiastka, do ktorej nejako zadáte dve čísla a ona vám nejako vypľuje ich súčin.) Aby však počítač robil niečo zmysluplné, musíme určiť, aké operácie sa majú vykonávať. Každá operácia má nejaké označenie, nazývané **inštrukcia**. Zoznam inštrukcií sa nazýva **program**. Nie každá postupnosť inštrukcií dáva zmysel. Súbor pravidiel, ktoré určujú, aké inštrukcie vieme používať a ako ich môžeme spájať sa nazýva **programovací jazyk**.

Pri prvých počítačoch zodpovedali inštrukcie priamo operáciám procesora. Keď ste si kúpili počítač, dostali ste k nemu aj manuál so sadou inštrukcií a pravidlami, ako ich spájať. Takýto programovací jazyk sa nazýva **strojový kód**. Ak ste si teda kúpili nejaký program, musel byť napísaný priamo pre váš počítač. A ani samotné písanie nebolo najpríjemnejšie.

Netrvalo dlho a zistilo sa, že existuje všeobecný postup, ako možno zobrať zápis programu v jednom jazyku a preložiť ho do iného jazyka. Tento postup sa nazýva **kompilácia**. Kompiláciu môžeme vykonávať aj ručne, avšak vieme písať programy (**kompilátory**), ktoré to spravia za nás. Vďaka tomu začali vznikať nové jazyky, nezávislé od konkrétnych počítačov, v ktorých sa programy písali ľahšie a po napísaní sa proste skompilovali do strojového kódu.

S rastúcim výkonom počítačov sa zjavili ďalšie možnosti. Zrazu bolo možné napísať program, ktorý simuloval výpočet iného počítača. A pri tejto simulácii postupoval podľa inštrukcií simulovaného (virtuálneho) počítača. Dokázali sme teda spustiť strojový kód iného (aj neexistujúceho) počítača. Tento proces sa nazýva **interpretácia** a program, ktorý ju vykonáva, je **interpreter**.

V dnešnej dobe existujú stovky programovacích jazykov. Sú medzi nimi strojové kódy, kompilované aj interpretované jazyky. Niektoré jazyky sa kompilujú viac krát a niektoré sa najskôr skompilujú a potom interpretujú. Všetko sa to deje preto, aby sa zjednodušil život programátorovi a zvýšila sa prenositeľnosť programov do iných systémov.

Operačný systém a systémové volania

Počítač v dnešnej dobe plní množstvo úloh. Musí sledovať, čo stláčate na klávesnici a kam klikáte, obsluhovať webkameru a tlačiareň, prehrávať hudbu, sťahovať z internetu fotky z poslednej akcošky a popri tom vám vykreslovať náročnú scenériu hry, ktorú hráte. Všetky tieto veci sa dejú naraz a vykonávajú

ich rôzne programy. Keďže máme ale iba jeden procesor, je treba rozhodnúť, ktorý program kedy beží. Taktiež treba obmedziť prístup programu k jednotlivým zariadeniam. Inak by si vás mohla náhodná hra nakamerovať, odsledovať heslo do internet bankingu alebo vytlačiť 3000 strán reklamy. Program, ktorý toto všetko zabezpečuje, sa nazýva nazýva **operačný systém**.

Operačný systém, je prvý program, ktorý sa spustí po zapnutí počítača. Zistí, aké všetky zariadenia máte pripojené a pripraví ich na používanie. Keď je všetko pripravené, vyzve používateľa, aby sa prihlásil, načíta jeho nastavenia a umožní mu spustiť iné programy. Počas behu iných programov dohliada, aby sa striedali, poskytuje im prístup k zariadeniam a notifikuje ich, keď sa niečo stane (stlačíte kláves, zasunieme USB).

Bežný program si vykonáva svoje výpočty a o nič iné sa nestará. Keď potrebuje spraviť niečo s nejakým zariadením, požiada o to operačný systém. Občas sa môže spýtať systému, či sa niečo nestalo (dokončila sa tlač, používateľ klikol myšou, ...). Takéto požiadavky a otázky na operačný systém sa nazývajú **systémové volania**.

Pomocou systémových volaní vieme prijímať ínformácie z myši a klávesnice alebo kresliť na obrazovku a vďaka tomu interagovať s používateľom. Všetky takéto volania sú však komplikované a teda pre začiatočníkov je odporúčané používať iný spôsob komunikácie s programom.

Tento spôsob využíva **štandardný vstup** a **štandardný výstup** a je možné si to predstaviť ako četovanie si s programom. Vy mu niečo napíšete a on vám odpíše. Takýto spôsob komunikácie je možný s každým programom, u väčšiny je však skrytý, pretože sa v dnešnej dobe interakcie a multimédií nepoužíva. Pre nás však bude prvým odrazovým mostíkom do sveta programovania. Budeme teda používať dve systémové volania: zisti, čo napísal používateľ a vypíš niečo používateľovi.

Reprezentácia dát v počítači

Pamäť počítača, ako je všeobecne známe, pozostáva z jednotiek a núl. Jeden takýto kus informácie sa nazýva bit, skupina ôsmich bitov sa nazýva bajt. Ako pomocou bitov a bajtov dokážeme vyjadriť čísla alebo texty?

Základom je reprezentácia čísiel. Ak ste už počuli o dvojkovej sústave, čítajte ďalej, ak nie, budete si ju musieť naštudovať z externého zdroja, pretože tento text je na jej vysvetlenie príliš krátky. Kladné celé čísla sa reprezentujú priamo ako ich zápis v dvojkovej sústave. Zvyčajne sa používa pevný počet bitov a preto je veľkosť takéhoto zápisu obmedzená. Ak chceme reprezentovať celé číslo, vyhradíme si jeden bit na znamienko – nula znamená kladné, jednotka záporné. Reálne číslo sa dá približne zapísať spojením dvoch celých čísel. Jedno určí niekoľko prvých nenulových cifier a to druhé určí pozíciu desatinnej čiarky vhľadom na poslednú z týchto cifier.

Text si vieme zapamätať, keď sa dohodneme na tabuľke, ktorá priradí jednotlivé symboly malým číslam. Postupnosť takýchto malých čísel potom zapíše text. Spomínaná tabuľka sa nazýva kódovanie. V dnešnej dobe má zmysel poznať ASCII kódovanie, ktoré obsahuje základné symboly používané v anglických textoch, a Unicode, ktoré zahŕňa väčšinu symbolov z väčšiny rozumne používaných jazykov.

Pole

Prednášajúci: Syseľ a Roman, text písal Roman

Abstrakt

Potrebovali ste niekedy veľa premenných, ktoré boli všetky "na jedno kopyto" a líšili sa iba hodnotou? Napríklad $a1, a2, a3, a4$ ... Tak to ste potrebovali *pole*! My si teraz povieme, ako ho v našom

programe vytvoriť a vykonávať s ním jednoduché operácie – pristupovať k jednotlivým prvkom a prechádzať ním v cykle.

Definícia 1. *Pole* (anglicky *array*) je dátová štruktúra, ktorá nám umožňuje pracovať s veľkým množstvom prvkov rovnakého typu.

Vytvárame pole

Príklad 1. Na vstupe sú 3 prirozedné čísla. Vypíšte ich v opačnom poradí.

Listing programu (C++)

```
#include <iostream>
using namespace std;

int main() {
    int a0, a1, a2;
    cin >> a0 >> a1 >> a2;
    cout << a2 << '\n' << a1 << '\n' << a0 << '\n';
}
```

Postačia nám 3 premenné, do ktorých načítame čísla zo vstupu a vypíšeme v opačnom poradí. Teraz sa pozrieme, ako by sa úloha dala riešiť poľom.

Listing programu (C++)

```
#include <iostream>
using namespace std;

int main() {
    int pole[3];
    cin >> pole[0] >> pole[1] >> pole[2];
    cout << pole[2] << '\n' << pole[1] << '\n' << pole[0] << '\n';
}
```

Na začiatku môžeme vidieť, ako sa pole definuje:
`int pole[3];`
Tento riadok vytvorí pole čísel (intov) s názvom `pole` a dlžkou 3.

Vieme, že príkaz `cin` slúži na načítavanie vstupu. Za ním by mal ale nasledovať názov premennej, nie? My však máme pole s názvom `pole`, ktoré obsahuje tri premenné. Musíme preto určit, ktorú z tých troch premenných chceme použiť.

Preto za názov poľa budeme písať hranaté zátvorky, v ktorých bude poradie žiadanej premennej. Výraz `pole[0]` nám teda dovolí pristupovať k prvému prvku poľa, a podobne môžeme pristupovať aj k druhému a tretiemu prvku (`pole[1]` a `pole[2]`). Všimnite si, že prvky sa v poli (v C++) **číslujú od 0**. Takéto výrazy sú teda naozaj premenné, a môžeme s nimi vykonávať také isté operácie:
`pole[0] = 2;` //priraď číslo 2 do prvej premennej poľa
`cin >> pole[2];` //načítaj hodnotu do tretieho prvku poľa
`cin >> pole[3];` //CHYBA – naše pole má iba 3 prvky na pozíciach 0, 1 a 2

Pozícia prvku v poli sa nazýva *index*. Ak si teda spravíme pole veľkosti n, jeho indexy sú čísla od 0 po $n-1$. Problém môže nastať, ak sa snažíme pristúpiť k prvku, ktorý neexistuje. Či už použijeme záporný index, alebo index, ktorý je priveľký, ako napríklad vo vyššie uvedenom prípade.

Je potrebné povedať, že takýto program by sa skompiloval a tváril sa bezchybne. Kompilátor nás neupozorní na podobné chyby, takže pri pristupovaní do poľa sa vždy treba zamyslieť, či sa nesnažíme pracovať s tým, čo nám nepatrí. Takáto chyba sa však (väčšinou) prejaví počas behu programu. Na testovači to spoznáte hláškou `EXC`, u seba väčšinou hláškou `Segmentation fault`.

Definícia 2. Všeobecne pre definíciu poľa platí štruktúra: `typ nazov[dlzka];`

Pristupovať k jednotlivým prvok na pozíciách 0 až $dlzka - 1$ môžeme vďaka *indexom*. Jednoducho povedané, index je číslo v hranatej zátvorke, ktoré jednoznačne určuje pozíciu v poli – $mena[0]$, $prvky[5]$.

Prechádzame poľom

Príklad 2. Teraz zovšeobecníme predošlú úlohu. Počet čísel neprezradíme v zadaní, ale označíme si ho n. Platí však $0 \leqslant n \leqslant 1\,000$.

Listing programu (C++)

```
#include <iostream>
using namespace std;

int main() {
    int pole[1000], n;
    cin >> n;
    for (int i = 0; i < n; ++i) {
        cin >> pole[i];
    }

    for (int i = n-1; i >= 0; --i) {
        cout << pole[i] << '\n';
    }
}
```

Riešenie je priamočiare, ukázali sme si ale, že index nemusí byť len konštatné číslo, ale aj premenná, alebo výraz. Môžeme tak polia jednoducho prechádzať pomocou cyklu.

Príklad 3. Nájdi pozíciu najmenšieho čísla v poli.

Listing programu (C++)

```
#include <iostream>
using namespace std;

int main() {
    int pole[1000], n, minPoz;
    cin >> n;
    for (int i = 0; i < n; ++i) {
        cin >> pole[i];
    }

    minPoz = 0;
    for (int i = 1; i < n; ++i) {
        if (pole[i] < pole[minPoz]) {
            minPoz = i;
        }
    }

    cout << "Najmensi prvok je na pozicii " << minPoz + 1 << endl;
}
```

Prejdeme všetky prvky poľa a počas prechodu si v premennej *minPoz* budeme pamätať pozíciu najmenšieho čísla, ktoré sme doposiaľ videli. Na začiatku, bude premenná *minPoz* ukazovať na prvý prvok (bude mať preto hodnotu 0).

Ďalej nasleduje cyklus, v ktorom postupne prejdeme všetky ostatné prvky poľa. Vždy sa spýtame, či je hodnota na aktuálnej pozícii menšia ako hodnota na pozícii *minPoz*. Ak áno, znamená to, že sme narazili na menší prvok ako naše doterajšie minimum. Do *minPoz* preto priradíme i, teda aktuálnu pozíciu.

Po skončení cyklu sa v *minPoz* nachádza pozícia najmenšieho prvku v poli.

Aplikácie v úlohách

Príklad 4. Zoraď vzostupne (rastúco) pole čísel s dĺžkou n pomocou hľadania najmešieho prvku.

Listing programu (C++)

```cpp
#include <iostream>
using namespace std;

int main() {
    int pole[1000], pom, minPoz;, n;
    cin >> n;
    for (int i = 0; i < n; ++i) {
        cin >> pole[i];
    }

    for (int i = 0; i < n; ++i) {
        minPoz = i;
        for (int j = i + 1; j < n; ++j) {
            if (pole[j] < pole[minPoz]) minPoz = j;
        }

        pom = pole[i];
        pole[i] = pole[minPoz];
        pole[minPoz] = pom;
    }
}
```

Myšlienka: Aký prvok bude na prvej pozícii v zoradenej postupnosti? Najmenší. Nájdeme ho preto a vymeníme s aktuálne prvým prvkom. Aký prvok bude teraz na druhej pozícii? Predsa najmenší z prvkov na pozíciách 1 až $n-1$, lebo prvok na pozícii 0 je na správnom mieste. Nájdeme ho a vymeníme s druhým. Všeobecne – ak už máme zoradených prvých i prvkov (pozície 0 až $i-1$), na i-tom mieste sa bude nachádzať najmenší prvok zo zvyšných čísel, teda tých na pozíciách i až $n-1$.

Príklad 5. Žabiak Jozef sa rozhodol skákať po leknách. Z každého lekna sa dá skočiť práve na jedno iné. Jazero si budeme reprezentovať ako postupnosť n rôznych čísel z rozsahu 1 až n. Jazero veľkosti 6 by teda mohlo vyzerať takto: $(4, 2, 5, 3, 1, 6)$. Čísla predstavujú jednotlivé lekná. Prvé lekno obsahuje číslo 4, čo znamená, že z neho Jozef môže skočiť na štvrté lekno. Zo štvrtého môže ďalej pokračovať na tretie, atď.

Zaujímalo by ho teraz, či vie preskákať z lekna na pozícii a na lekno na pozícii b. Vstup začína troma číslami n, a, b. Ďalej nasleduje n rôznych čísel reprezentujúcich jednotlivé lekná.

Listing programu (C++)

```cpp
#include <iostream>
using namespace std;

int main() {
    int lekna[1000], n, a, b, akt;
    cin >> n >> a >> b;
    for (int i = 0; i < n; ++i)
        cin >> lekna[i];

    akt = lekna[a-1];
    while (akt != a && akt != b)
        akt = lekna[akt-1];

    if (akt == a) cout << "neexistuje" << endl;
    if (akt == b) cout << "existuje" << endl;
}
```

V riešení jednoducho simulujeme skákanie z pozície a. Ak sa dostaneme na pozíciu b, určite cesta existuje. Ak neobjavíme b a dokončíme celý cyklus (sme znovu na začiatku v a), cesta neexistuje. Musíme si ešte dať pozor na indexovanie od 0.

Časová zložitosť. Vector.

Prednášajúci: Baklažán

Abstrakt

V prvej časti prednášky si ukážeme spôsob, ktorým sa dá popísať rýchlosť algoritmu – asymptotickú časovú zložitosť. V druhej časti si ukážeme, ako sa dá naprogramovať pole premenlivej veľkosti, ktoré nebude oveľa pomalšie než obyčajné pole fixnej veľkosti.

Asymptotická časová zložitosť

Poznámka 1. Ak nie je povedané inak, písmenom n v tejto prednáške značím veľkosť vstupu.

Definícia 1.

- *Algoritmus* je postupnosť niekoľkých dobre definovaných inštrukcií – úkonov, ktorá slúži na vykonanie nejakej úlohy.
- *Program* je skupina inštrukcií v nejakom konkrétnom programovacom jazyku. Programy typicky reprezentujú nejaké algoritmy, jeden algoritmus sa však dá reprezentovať mnohými rôznymi programami (ktoré môžu byť v rôznych programovacích jazykoch).
- *Implementácia* algoritmu je ľubovoľný program, ktorý tento algoritmus vykonáva.

To, ako dlho algoritmus beží, závisí od viacerých faktorov: od rýchlosti počítača, na ktorom beží, od konkrétnej implementácie, od použitého programovacieho jazyka, veľkosti vstupu atď.. Väčšina z týchto faktorov nezávisí iba od daného algoritmu, jednu vec však často vieme povedať aj čisto na základe algoritmu: ako bude čas behu rásť, keď bude rásť veľkosť vsupu. Konkrétne nás bude zaujímať odpoveď na otázku: "Keď zväčším veľkosť vstupu x-krát, ako sa zväčší čas behu programu?".

Definícia 2.

- Ak pre algoritmus platí, že ak x-krát zväčšíme vstup, algoritmus bude bežať x-krát dlhšie, hovoríme, že časová zložitosť algoritmu je *lineárna*, značíme $O(n)$. Ak teda algoritmu s lineárnou časovou zložitosťou zväčšíme vstup dvakrát, jeho čas behu sa zväčší dvakrát.

Príklad 1. Nasledujúci program načíta n čísel zo vstupu a vypíše ich v opačnom poradí. Jeho časová zložitosť je lineárna.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n;
    cin >> n;
    vector<int> pole(n);
    for(int i=0; i<n; i++) cin >> pole[i];
    for(int i=n-1; i>=0; i--) cout << pole[i] << "␣";
    cout << endl;
}
```

Definícia 3.

- Ak pre algoritmus platí, že ak x-krát zväčšíme vstup, algoritmus bude bežať x^2-krát dlhšie, hovoríme, že časová zložitosť algoritmu je *kvadratická*, značíme $O(n^2)$. Ak teda algoritmu s kvadratickou zložitosťou zväčšíme vstup dvakrát, bude bežať štyrikrát dlhšie.
- Ak bude algoritmus po zväčšení vstupu x-krát bežať x^3 krát dlhšie, jeho zložitosť je $O(n^3)$ (*kubická*), ak x^4 krát dlhšie, jeho zložitosť je $O(n^4)$, atď..

Príklad 2. Tento program načíta číslo n a vypíše štvorec $n \times n$ hviezdičiek. Jeho časová zložitosť je kvadratická.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n;
    cin >> n;
    for(int y=0; y<n; y++) {
        for(int x=0; x<n; x++) {
            cout << "*";
        }
        cout << endl;
    }
}
```

Tvrdenie 1. *Ak má algoritmus viac (konštante veľa) častí s rovnakou časovou zložitosťou, časová zložitosť celého algoritmu je rovnaká ako zložitosť jednej časti.*

Ak máme dva algoritmy riešiace ten istý problém, pričom jeden z nich je kvadratický a druhý je lineárny, potom od istej veľkosti vstupu bude lineárny algoritmus rýchlejší. Je to tak preto, lebo keď budeme zväčšovať vstup, čas behu kvadratického algoritmu bude rásť rýchlejšie ako čas behu lineárneho, teda aj keby bol pre malé vstupy kvadratický algoritmus rýchlejší, pri väčších vstupoch jeho čas behu "prerastie"čas behu lineárneho algoritmu.

Ak bol napríklad kvadratický algoritmus na nejakom malom vstupe 10-krát rýchlejší než lineárny, stačí nám pozrieť sa na 10-krát väčší vstup. Na takomto vstupe bude kvadratický algoritmus bežať 100-krát pomalšie než na pôvodnom a lineárny bude bežať 10-krát dlhšie než na pôvodnom vstupe, teda oba budú bežať rovnako rýchlo. Na ešte väčších vstupoch už bude lineárny algoritmus rýchlejší.

Preto lineárne algoritmy vo všeobecnosti považujeme za rýchlejšie než kvadratické, čo sa niekedy značí aj ako $O(n) << O(n^2)$. Podobne platí $O(n^2) << O(n^3) << O(n^4) << \ldots$.

Tvrdenie 2. *Ak má algoritmus viac častí, pričom ich časové zložitosti sú rôzne, za časovú zložitosť celého algoritmu považujeme zložitosť najpomalšej z nich.*

Príklad 3. Minsort:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n;
    cin >> n;
    vector<int> pole(n);
    for(int i=0; i<n; i++) {
        cin >> pole[i];
    }

    for(int i=0; i<n; i++) {
        int mini = 1023456789, kde_je = -1;
        for(int j=i; j<n; j++) {
            if(pole[j] < mini) {
                mini = pole[j];
                kde_je = j;
            }
        }
        swap(pole[i], pole[kde_je]);
    }

    for(int i=0; i<n; i++) {
        cout << pole[i] << "␣";
    }
    cout << endl;
}
```

Ďalšie triedy zložitosti

Definícia 4.

- Ak čas behu algoritmu nezávisí na veľkosti vstupu, hovoríme, že jeho časová zložitosť je $O(1)$ (*konštantná*).

- Ak sa čas behu algoritmu pri zväčšení vstupu o jeden prvok zdvojnásobí, hovoríme, že jeho časová zložitosť je $O(2^n)$ (*exponenciálna* – aj keď týmto slovom sa označujú aj iné zložitosti).
- Ak sa čas behu algoritmu pri zdvojnásobení veľkosti vstupu zväčší iba o nejakú konštantu, jeho časová zložitosť je $O(\log n)$ (*logaritmická*).
- Vo všeobecnosti, ak je čas behu algoritmu priamo úmerný nejakej funkcii f od veľkosti vstupu, hovoríme o časovej zložitosti $O(f)$.

Nakoniec ešte uvedieme usporiadanie všetkých spomínaných tried podľa "rýchlosti". Toto usporiadanie má nasledovný zmysel: Ak máme dva algoritmy z rôznych tried, algoritmus z rýchlejšej triedy bude od istej veľkosti vstupu bežať rýchlejšie než algoritmus z pomalšej triedy.

$$O(1) << O(\log(n) << O(n) << O(n^2) << \cdots << O(n^k) << O(2^n))$$

Vector

Obyčajné polia v C++ potrebujú už počas kompilácie vedieť, aké budú veľké. To je dosť nepraktické v situáciách, keď veľkosť poľa, ktoré potrebujeme, závisí od veľkosti vstupu. Takáto situácia v praxi nastáva veľmi často. Príkladom môžu byť programy na úpravu rastrových obrázkov (Skicár, Gimp, Photoshop), ktoré si otvorený obrázok potrebujú pamätať v poli, pričom veľkosť obrázka sa dozvedia, až keď ho začnú otvárať.

Poznámka 2. Nasledujúcu techniku vám odporúčam nepoužívať, hlavne kým poriadne nerozumiete tomu, ako presne funguje. Na tejto prednáške sa to nedozviete a v skutočnosti ju ani nepotrebujete, ak sa naučíte používať `vector`.

Našťastie, v C++ existuje aj spôsob, ako vytvoriť počas behu programu pole zadanej veľkosti:

```
int n;
cin >> n;
int *pole;
pole = new int[n];
```

Aj v tomto prípade však musíme veľkosť poľa poznať skôr, než ho začneme používať. V prípade obrázkových editorov nám tento spôsob stačí, niekedy však potrebujeme vedieť pole zväčšiť už po tom, ako sme doňho zapísali nejaké prvky.

Na toto existuje v C++ vylepšenie poľa s názvom `vector`. S vectorom sa dá robiť všetko čo s poľom (je to hŕba premenných, do ktorej sa dá indexovať), navyše však dokáže (okrem iného) nasledovné veci:

1. Vytvoriť sa s požadovanou veľkosťou n (ktorá mohla byť zistená až počas behu programu), v čase $O(n)$.
2. Zväčšiť svoju veľkosť o 1 prvok, v *amortizovanej*[1] časovej zložitosti $O(1)$.

Prvú z týchto vecí dokáže aj obyčajné dynamicky alokované pole, druhú však už nie.

Definícia 5.

- Za *amortizovaný* čas vykonávania operácie považujeme *priemer* z časov, za ktoré sa táto operácia vykoná počas celého algoritmu. Napríklad, ak má operácia amortizovanú časovú zložitosť $O(1)$ a počas celého behu programu sa vykoná k krát, časová zložitosť vykonania všetkých týchto operácií bude $O(k)$.

[1] tento pojem si vysvetlíme neskôr

Okrem toho má vector ešte hromadu ďalších užitočných vychytávok, tie sú ale nad rámec tejto prednášky.

Príklad práce s vectorom:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n;
    cin >> n;
    vector<int> pole(n); //vyrobi vector velkosti n (ktoru sme nacitali zo vstupu)
    for(int i=0; i<n; i++) cin >> pole[i];
    vector<int> velke; //vyrobi vector velkosti 0
    for(int i=0; i<n; i++) {
        if(pole[i] > 1000) {
            velke.push_back(pole[i]); //zvacsi vector velke o 1 prvok, hodnotu tohto prvku nastavi na pole[i]
        }
    }
    cout << "Pocet prvkov vacsich ako 1000 je " << velke.size() << endl;
    //velke.size() nam da pocet prvkov vectoru velke
    for(int i=0; i<velke.size(); i++) {
        cout << velke[i] << " ";
    }
    cout << endl;
}
```

Úlohy

V nasledujúcich príkladoch je úlohou určiť časovú zložitosť zadaného algoritmu, programu alebo časti programu.

Úloha 1 (Floyd-Warshall)

```
int n;
vector<vector<int> > dist(n, vector<int> (n));
...
for(int k=0; k<n; k++) {
    for(int i=0; i<n; i++) {
        for(int j=0; j<n; j++) {
            if(dist[i][j] > dist[i][k] + dist[k][j]) {
                dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
}
```

Úloha 2

Nasledujúci kus kódu skontroluje, či je v poli nejaká nula a ak áno, prepíše všetky prvky na 47.

```
int n;
vector<int> pole(n);
...
for(int i=0; i<n; i++) {
    if(pole[i] == 0) {
        for(int j=0; j<n; j++) {
                pole[1] = 47;
        }
    }
}
```

Úloha 3 (binárne vyhľadávanie)

```
int n, hladam;
vector<int> pole(n);
...
//predpokladame, ze pole je teraz uz utriedene
int malo = 0, vela = n;
while(malo < vela-1) {
    int stredne = (malo + vela)/2;
    if(pole[stredne] > hladam)vela = stredne;
    else malo = stredne;
}
```

Úloha 4 (neefektívna konštrukcia intervalového stromu)

```
int n;
vector<int> pole(2*n);
...
for(int i=n; i<2*n; i++) {
    for(int j = i; j>1; j/= 2) {
        pole[j/2] += pole[j];
    }
}
```

Riešenia úloh

Riešenie 1
$O(n^3)$

Riešenie 2
Napriek dvom vnoreným for-cyklom, algoritmus beží v čase $O(n)$.

Riešenie 3
$O(\log n)$

Riešenie 4
$O(n \cdot \log n)$

Rekurzia

Prednášajúci: Žaba a Roman

Abstrakt

Rozmýšlali ste niekedy, čo sa stane, ak zavoláte funkciu samú v sebe? Neskončí vtedy vesmír? My si teraz ukážeme, ako sa niečo také dá využiť pri riešení reálnych problémov. Najskôr si zopakujeme, čo je to funkcia a potom na pár príkladoch vysvelíme, ako rekurzia vlastne funguje. Vyriešime napríklad *Úlohu o 8 vežiach* a tiež si na *Fibonacciho postupnosti* ukážeme, že sa nám občas oplatí zapamätať si pri výpočtoch niečo naviac – využijeme *memoizáciu*.

O funckiách

Definícia 1. *Funckia* (anglicky *function*) je postupnosť príkazov, ktoré riešia určitý podproblém. Môže mať rôzne *argumenty*, ktoré predstavujú vstup pre náš podproblém. Na konci funckia *vracia hodnotu*, ktorá predstavuje výsledok.

Nasledujúca funckia napríklad vypočíta obsah obĺžnika so stranami a a b.

Listing programu (C++)

```
#include <iostream>
using namespace std;

int obsah(int a, int b) {
    int S = a*b;
    return S;
}

int main() {
    cout << obsah(8, 5) << endl;    //vypise 40
    cout << obsah(10, 10) << endl;  //vypise 100
}
```

Strany a a b prijme funckia ako argumenty. Obsah S vráti ako výsledok. Všimnite si, že aj samotný *main()* je vlastne funckia, ktorá sa zavolá pri spustení programu. Vracia celé číslo a nemá (zvyčajne) žiadne argumenty.

Čo je rekurzia?

Skúsme si teraz odpovedať na otázku z úvodu – čo sa stane, ak zavoláme funkciu samu v sebe? Zamyslime sa nad nasledujúcou ukážkou.

Listing programu (C++)

```
#include <iostream>
using namespace std;

int makaj(int n) {
    return makaj(n-1);
}

int main() {
    cout << makaj(3) << endl;
}
```

Funckia `makaj()` má jeden argument n, pričom vracia hodnotu, ktorú vráti funckia `makaj()` zavolaná s argumentom $n - 1$. Odsimulujme si teraz volanie tejto funckie s hodnotou 3.

Listing programu (Text)

```
makaj(3)        //zavolane v maine; makaj(3) potrebuje hodnotu makaj(2), ktoru chce vratit
    makaj(2)    //makaj(2) potrebuje hodnotu makaj(1)
        makaj(1)        //makaj(1) potrebuje hodnotu makaj(0)
            makaj(0)    //makaj(0) potrebuje hodnotu makaj(-1)
                ....
                    ....
```

Tuším sme sa zacyklili. Zjavne naše nekonečné volanie nemá čo zastaviť. Čo ak by sme ale do našej funckie pridali nejakú šikovnú *podmienku*?

Listing programu (C++)

```
#include <iostream>
using namespace std;

int makaj(int n) {
    if (n == 0)
        return 42;

    return makaj(n-1);
}

int main() {
    cout << makaj(3) << endl;
    return 0;
}
```

Znovu sa pozrieme, ako bude prebiehať volanie našej funckie.

Listing programu (Text)

```
makaj(3)        //zavolane v maine; makaj(3) potrebuje hodnotu makaj(2), ktoru chce vratit
    makaj(2)
        makaj(1)
            makaj(0)   //v tejto funckii sa n rovna 0; splni sa teda podmienka a my sa dalej nevoláme, ale vratime 42
        makaj(1)        //uz pozname hodnotu makaj(0), takze ju vratime
    makaj(2)    //pozname makaj(1)
makaj(3)        //pozname makaj(2)

//vysledok volania makaj(3) je 42
```

Vidíme, že sa už funckia nezacyklí – keď sa n rovnalo 0, nevolali sme sa ďalej, ale vrátili sme hodnotu 42. Mohli sme teda vidieť prvú skutočnú rekurziu, ktorá niečo aj počítala.

Definícia 2. *Rekurzia* (anglicky *recursion*) je názov postupu, kedy funkciu definujeme pomocou seba samej. Pri tejto definícii musí existovať tzv. *základný prípad*, ktorý zaručí, že sa nám funckia nezacyklí.

Základný prípad (angliky *base case*) je také volanie funkcie, pre ktoré už dopredu poznáme riešenie. Nevoláme pri ňom už ďalej našu funckiu, ale vrátime priamo toto riešenie.

Môžeme si všimnúť, že v našom príklade vyššie nastal základný prípad vtedy, keď sa n rovnalo 0. Povedali sme si už dopredu, že pre tento prípad bude výsledok 42.

Príklad 1. Vytvorte funkciu, ktorá vráti súčet prvých n prirodzených čísel.

Listing programu (C++)

```
#include <iostream>
using namespace std;

int sucet(int n) {
    if (n == 0)
        return 0;
    return n + sucet(n-1);
}

int main() {
    cout << sucet(3) << endl;
    cout << sucet(10) << endl;
}
```

(Na chvíľu predstierajme, že neexistuje vzorec pre súčet prvých n členov)
Myšlienka: Vieme súčet čísel od 0 po n vypočítať hneď? Nie. Vieme ale problém *zmenšiť* – môžeme povedať, že tento súčet je rovný `n + sucet(n - 1)`. Základným prípadom bude 0 – vtedy vrátime súčet 0.

Rekurziu môžeme využívať v prípadoch, kedy vieme problém po častiach zmenšovať, až kým nenarazíme na najjednoduchší problém, pre ktorý vieme odpoveď hneď a ten sa stane naším základným prípadom.

Backtracking

Príklad 2. Koľkými možnými spôsobmi vieme rozložiť 8 veží na šachovnici tak, aby sa navzájom neohrozovali?

Listing programu (C++)

```
#include <iostream>
using namespace std;

int mriezka[8][8] = {0}, pocet = 0;

bool jeBezpecne(int s) {
    //kontrola v stlpci
    for (int i = 0; i < 8; ++i) {
        if (mriezka[i][s] == 1) return false;
    }
    //ak sme az tu, presli sme testami
    return true;
}
//postupne zaplname z horneho riadku - r
void rozlozenia(int r) {
    //ak sme prisli do posledneho radu, nasli sme riesenie
    if (r == 8) {
        ++pocet;
        return;
    }
    for (int i = 0; i < 8; ++i) {
        if (jeBezpecne(i)) {
            mriezka[r][i] = 1;
            rozlozenia(r+1);
            mriezka[r][i] = 0;
        }
    }
}

int main() {
```

```
    rozlozenia(0);
    cout << pocet << endl;
}
```

Program je trochu dlhší, ale jeho myšlienka je jednoduchá – skúsime do riadka r postupne položiť na každú pozíciu vežu. Ak sa nám to podarí (neohrozuje sa s ostatnými), posunieme sa o riadok nižšie a skúsime ukladať ďalšiu vežu tam. Riešenie nájdeme, ak sa dostaneme za posledný riadok – položili sme vežu do každého riadku. Vtedy sa vrátime o jedno vnorenie vyššie a skúšame nájsť ďalšie riešenia.

Pri implementácii sme použili jeden trik – globálne 2D pole `mriezka[][]`, kde si pamätáme šachovnicu. Vždy, keď pokladáme vežu na pozíciu (r, s), najskôr sa pozrieme, či v stĺpci s už neleží iná veža. Ak nie, zaznačíme si do nášho poľa na pozíciu `mriezka[r][s]` hodnotu 1 a rekurzívne sa zavoláme na ďalší riadok. Toto sme spravili preto, aby sme v ďalšom riadku vedeli zistiť, že v stĺpci s už veža je. Keď sa skončí rekurzívne volanie musíme si však dať pozor, aby sme hodnotu `mriezka[r][s]` nastavili na 0. Skončenie rekurzívneho volania totiž znamená, že už sme skúsili všetky ďalšie možnosti, keď bola veža na políčku (r, s) a teraz ju ideme vyskúšať dať na riadku r do iného stĺpca.

Technika, ktorú sme práve použili sa nazýva back-tracking.

Definícia 3. *Back-tracking*, alebo aj "metóda pokusov a opráv" je spôsob riešenia problémov, pri ktorom prechádzame stavmi, resp. stavovým stromom. Je lepší ako skúšanie všetkých možností hrubou silou (*bruteforce*), pretože často môžeme vylúčiť veľké množsvo potenciálnych riešení bez priameho vyskúšania.

Definícia znie možno trochu ťažkopádne, preto si ju vysvetlíme na predošlom príklade: Stavy predstavovala naša šachovnica. Vždy keď sme na nej pridali, alebo zobrali vežu, prešli sme do iného stavu. O niektorých stavoch sme vedeli povedať, že nevedú k riešeniu (ak sa práve položená veža ohrozovala s inou) – vtedy sme nepustili rekurzívne našu funkciu, ale tento stav sme rovno prehlásili za zlý a nepridávali sme doňho ďalšie veže. Vďaka tomu náš program bežal výrazne rýchlejšie ako hrubá sila, ktorá sa pozrie na všetky možné rozloženia ôsmich veží a pre každé zvlášť určí, či je platné.

Memoizácia

Príklad 3. Zistite n-tý člen Fibonacciho postupnosti. *Fibonacciho postupnosť* je postupnosť čísel začínajúca 0 a 1. Každý nasledujúci člen je súčtom predchádzajúcich dvoch. Prvých 10 členov Fibonacciho postupnosti vyzerá: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.

Listing programu (C++)

```
#include <iostream>
using namespace std;

int fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}

int main() {
    cout << fibonacci(6) << endl;
    return 0;
}
```

Tento príklad priam volá po rekurzívnej implementácii, keďže samotná postupnosť je definovaná rekurzívne – pomocou menších členov. Program je podobný ako úvodný, namiesto jednej podmienky ale máme dve, pretože funckiu voláme aj s hodnotu $n - 2$. Pri n rovnému 1 by sme sa tak pýtali na -1 člen, ktorý samozrejme neexistuje.

Ak však otestujeme náš program, zistíme, že už výpočet 40-teho člena trvá akosi dlho. Výpočet nového člena predstavuje iba spočítanie dvoch predošlých čísel, čo potom nášmu programu trvá tak dlho?

Skúsme si ho odsimulovať.

Listing programu (Text)

```
fibonacci(6)    //zavola 5 a 4
    fibonacci(5)    //4 a 3
        fibonacci(4)    //3 a 2
            fibonacci(3)    //2 a 1
                fibonacci(2)    //1 a 0
                    fibonacci(1)
                    fibonacci(0)
                fibonacci(1)
            fibonacci(2)    //2 a 1
                fibonacci(1)
                fibonacci(0)
        fibonacci(3)    //2 a 1
            fibonacci(2)    //1 a 0
                fibonacci(1)
                fibonacci(0)
            fibonacci(1)
    fibonacci(4)    //3 a 2
        fibonacci(3)    //2 a 1
            fibonacci(2)    //1 a 0
                fibonacci(1)
                fibonacci(0)
            fibonacci(1)
        fibonacci(2)    //1 a 0
            fibonacci(1)
            fibonacci(0)
```

Zjavne mnoho vecí počítame zbytočne veľakrát. Napríklad `fibonacci(6)` potrebuje zistiť 5-ty a 4-tý člen. Zavolá na ne preto príslušné funkcie. Piaty člen ale tiež potrebuje štvrtý člen a rovnaká funkcia na výpočet 4-tého člena sa zavolá dvakrát. A tak sa dvakrát zavolajú všetky volania funkcie potrebné na výpočet tohoto člena. Funkciu `fibonacci(2)` dokopy zavoláme až 5 krát.

Pritom funkcia `fibonacci(2)` vráti vždy číslo 1 bez ohľadu na to, koľkokrát ju zavoláme. Náš program teda robí naozaj veľa vecí navyše. Dalo by sa ukázať, že od člena, ktorý chceme vypočítať, tieto veci závisia až exponenciálne.

Čo ak by sme si ale pri každom vypočítanom člene zapamätali, aký výsledok sme vypočítali? Ak by sme ho potom potrebovali znovu, jednoducho by sme vrátili zapamätanú hodnotu. Nemuseli by sme teda opäť počítať všetky menšie členy.

Listing programu (C++)

```
#include <iostream>
#define MAX 1000
using namespace std;

int memoizacia[MAX];

int fibonacci(int n)
{
    if (memoizacia[n] != -1) return memoizacia[n];
    if (n == 0) return 0;
    if (n == 1) return 1;

    memoizacia[n] = fibonacci(n-1) + fibonacci(n-2);
    return memoizacia[n];
}

int main() {
    //-1 bude znamenat, ze tento clen sme este nepocitali
    for (int i = 0; i < MAX; ++i) memoizacia[i] = -1;

    cout << fibonacci(45) << endl;
    return 0;
}
```

Veľmi jednoduchá úprava kódu náš program teraz výrazne zrýchlila. Od člena, ktorý chceme vypočítať už nezávisí exponenciálne, ale lineárne. Každé možné volanie funkcie `fibonacci()` pre čísla menšie ako n totiž vypočítame **práve raz** ako súčet dvoch čísel.

Definícia 4. *Memoizácia* je optimalizáčná technika, pri ktorej si ukladáme výsledky volaných funckií pre ďalšie použitie – pri rovnakých hodnotách sa teda funkcia nemusí znovu volať. Časová zložitosť sa dá potom vypočítať ako počet možných volaní funkcie krát čas na výpočet jednej hodnoty.

Grafy a BFS

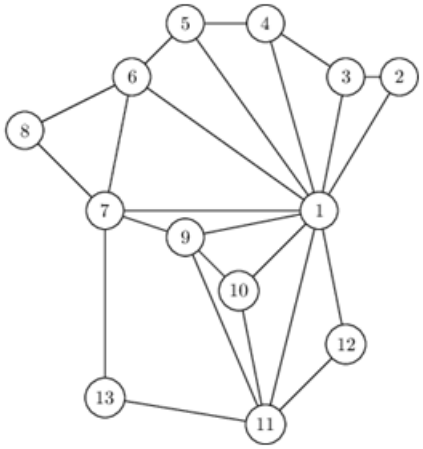
Prednášajúci: Žaba a Andrej

Abstrakt

Na tejto prednáške si najprv povieme čo to graf je. Zistíme aké typy grafov poznáme, preberieme jeho možné reprezentácie v pamäti počítača, ich výhody aj nevýhody. Na koniec si ukážeme algoritmus prehľadávania do šírky (BFS) čo je jeden zo základných, často používaných grafových algoritmov.

Čo je to graf?

Definícia 1. Graf je množina vrcholov pospájaných hranami.



Obr. 1: Príklad grafu.

Graf si môžeme dobre predstaviť ako cestnú sieť. Mestá, predstavujú vrcholy, hrany predstavujú cesty medzi nimi. Hrana medzi dvoma vrcholmi A a B nám reprezentuje, že sa vieme z vrcholu A dostať do vrcholu B a opačne. Ak sa vrátime k pohľadu na graf ako na cestnú sieť, hrana značí, že sa vieme priamo z mesta A dostať do mesta B po nejakej ceste.

Susedné sa dva vrcholy nazývajú pokiaľ medzi nimi existuje hrana, ktorá ich spája.

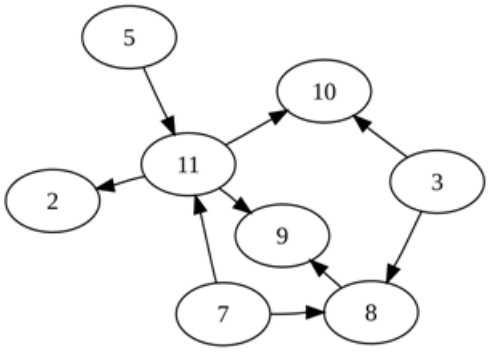
Stupeň vrcholu je počet hrán, ktoré vychádzajú z príslušného vrcholu.

Postupnosť hrán, ktorými vieme prejsť z jedného vrcholu do iného nazývame **cesta**. Počas cesty by sa nemali opakovať hrany ani vrcholy, ktoré sme prešli.

Samotné hrany môžu mať naviac číselné ohodnotenie. To môže hovoriť akú má daná hrana dĺžku, aký je potrebný čas na jej prejdenie, ale aj koľko zebier stretneme po ceste touto hranou. Informáciu, ktorú toto číslo predstavuje si volíme my na základe toho, aký problém práve riešime. Takýto graf voláme **ohodnotený**.

Graf môže byť takisto **orientovaný**. V orientovanom grafe má každá hrana priradený smer (takzvanú orientáciu), v ktorom je možné ňou prechádzať. Ak teda vedia hrana z vrchola A do vrchola B, nemôžeme sa touto istou hranou vracať aj opačným smerom. Ak sa chceme dostať opäť do A, musíme použiť inú hranu.

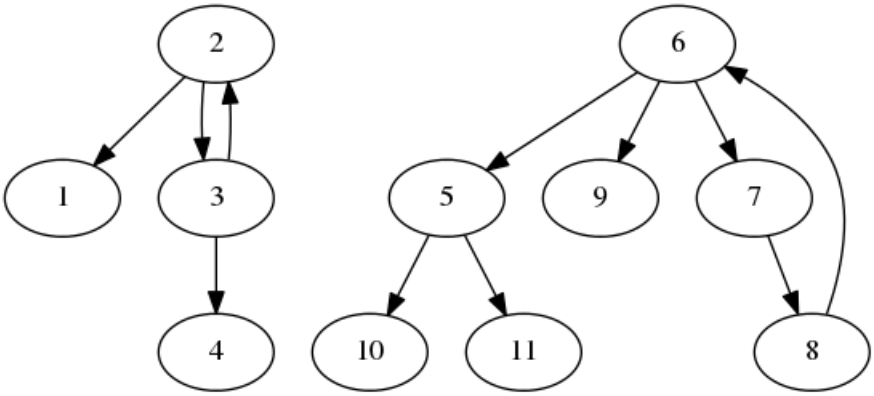
Uvedomme si, že nie v každom grafe existuje hrana medzi každými dvoma vrcholmi a dokonca, medzi niektorými vrcholmi nemusí existovať ani cesta. Zahnané do extrému, graf s 10 vrcholmi a 0 hranami je stále graf, akurát sa v ňom nedá príliš pohybovať.



Obr. 2: Príklad orientovaného grafu.

Definícia 2. Ak existujú v grafe podmnožiny vrcholov medzi ktorými neexistuje cesta nazývame takéto množiny komponenty. Viacero komponentov môže byť súčasťou jedného grafu.

Tu vidíme graf, ktorý má 2 komponenty, hovoríme však stále o jednom grafe.



Obr. 3: Príklad grafu s dvomi komponentami.

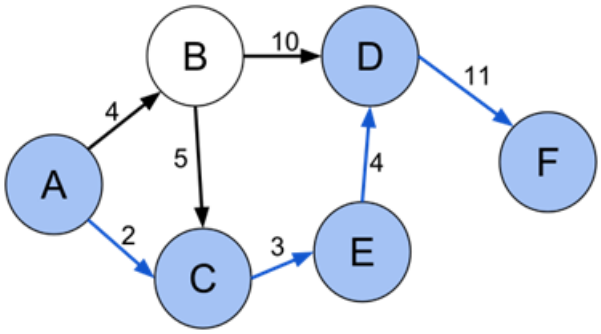
Využitie

Teraz prichádza dôležitá otázka : Na čo je toto všetko dobré?

Ako sme spomínali, grafom sa dobre reprezentuje napriklad cestná sieť. Predstavme si vymyslenú cestnú sieť ako na obrázku. Aplikovaním známych algoritmov vieme v tomto grafe nájsť napríklad najkratšiu cestu medzi vrcholmi A a F, ktoré môžu predstavovať napríklad mestá Bratislava a Košice. Grafom vieme reprezentovať však omnoho viac, jednotlivé vrcholy nám môžu napríklad predstavovať ľuďí a hrany medzi nimi jednotlivé priateľstvá na Facebooku. Takto by sme jednoducho vedeli zistiť spoločný počet priateľov ľubovoľných dvoch ľudí. Zamyslite sa ako ...

Reprezentácia grafu v pamäti

Keď už vieme čo ten graf vlastne je, naskytá sa dôležitá otázka – ako si graf zapamatať v našom programe, teda ako ho reprezentovať v pamäti počítača?



Obr. 4: Graf ako cestná sieť.

My si predstavíme dva rôzne spôsoby na zapamätanie si grafu. Na oba je nám potrebná iba dátová štruktúra `vector<>`, teda klasické pole. Najprv si oba spôsoby predstavíme a potom ich navzájom porovnáme.

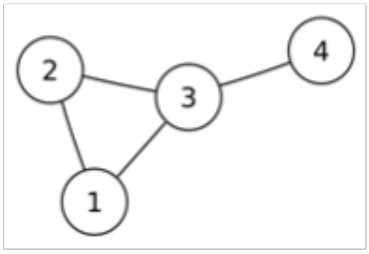
Prvým, jednoduchším spôsobom je zapamatať si graf v matici. Matica je dvojrozmerný `vector`, ktorý má rozmery $n \times n$, kde n je počet vrcholov. Nazvime si toto dvojrozmerné pole G. Na pozícii `G[a][b]` si budeme značiť, či existuje hrana medzi vrcholmi a a b. Ak existuje, na túto pozíciu dáme číslo 1, inak tam bude číslo 0. Všimnite si, že ak používame neorientované grafy, tak `G[a][b] = G[b][a]`.

Navyše, ak by bol graf ohodnotený, vieme si pamatäť na jednotlivých miestach, nie len či hrana existuje ale aj jej váhu, resp. jej ohodnotenie. Neexistujúcu hranu si označíme ako čislo, ktoré nemôže reprezentovať reálny údaj, napríklad -1.

Teraz si predstavíme druhý spôsob uloženia grafu v pamäti a to zoznam susedov. V zozname susedov máme pre každý vrchol určený jeden `vector`, v ktorom si pamätáme všetkých susedov, teda vrcholy, do ktorých sa dá z našeho vrcholu dostať po jednej hrane(pre orientované grafy samozrejme len v povolenom smere).

Pre ohodnotené grafy by sme si museli pamätať dvojice $(sused, vaha)$.

G	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1
4	0	0	1	0



1	2	3	
2	3	1	
3	2	4	1
4	3		

Obr. 5: V strede obrázka je graf. Naľavo je jeho maticová reprezentácia, napravo reprezentácia cez zoznam susedov.

Matica vs. zoznam susedov

Určite by vás teraz zaujímalo, ktorý z týchto prístupov je lepší. Odpoveď znie: oba spôsoby sú iné a každý sa hodí niekedy inokedy. My si teraz predstavíme výhody a nevýhody týchto reprezentácií a ukážeme si na čo je ktorá lepšia.

Príklad 1. Zistite či existuje hrana medzi vrcholmi a a b.

Matica: Pozrieme sa na hodnotu `G[a][b]`. Riešenie zistíme v konštantnom čase $O(1)$, podľa hodnoty tejto premennej.

Zoznam susedov: Prejdeme susedov vrcholu a. Ak hrana existuje určite budem medzi susedmi vrchol b. Časová zložitosť je $O(\textit{stupen vrcholu } a)$.

Príklad 2. Zistite počet všetkých susedov vrcholu a.

Matica: Pozrieme sa na riadok vrcholu a v matici, prejdeme ním a spočítame si počet existujúcich hrán v čase $O(n)$.

ZS: Zistíme veľkosť vectora, v ktorom si pamätáme susedov vrcholu a. To ide v čase $O(1)$.

Príklad 3. Odstráňte hranu medzi vrcholmi a a b.

Matica: Jediné dve políčka, ktoré nesú informáciu o hrane medzi A a B sú `G[a][b]` a `G[b][a]`, tieto vieme zmeniť v konštantnom čase $O(1)$.

ZS: Prejdeme susedov a aj susedov b a šikovne odstránime hranu v čase $O(\textit{stupeb } a + \textit{stupen } b)$.

Príklad 4. Pridajte hranu z vrcholu a do vrcholu b.

Matica: Hodnotu `G[a][b]` a `G[b][a]` zmeníme na 1 v čase $O(1)$.

ZS: V konštantnom čase pridáme b medzi susedov vrcholu a a naopak v čase $O(1)$.

Príklad 5. Aká je pamäť spotrebovaná na zapamätanie grafu?

Matica: Musíme si pamätať maticu $n \times n$, teda $O(n^2)$.

ZS: Každú hranu si pamätáme dvakrát, raz pre každý koniec. Preto potrebujeme iba $O(m)$ pamäte.

Príklad 6. Vypíšte všetkých susedov vrcholu a.

Matica: Pozrieme sa na riadok vrcholu a v matici, prejdeme ním a vypíšeme všetky i, kde `G[a][i]` sa rovnalo 1. To bude trvať čas $O(n)$.

ZS: Vypíšeme všetky prvky vectora, v ktorom si pamätáme susedov vrcholu a. To zaberie čas $O(\textit{stupen } a)$.

Práve posledný príklad je dôvod, prečo si častokrát zvolíme radšej reprezentáciu pomocou zoznamu susedov. Keď totiž prechádzame susedov nejakého vrcholu v matici, pozeráme sa na veľa zbytočných vrcholov, s ktorými hranu nemá. Pri zozname sa pozeráme len na tých, ktorých potrebujeme. Najviac sa to prejaví, ak má napríklad každý vrchol pomerne malý stupeň, poprípade ak chceme prejsť susedov všetkých vrcholov. Pri matici by sme totiž museli urobiť $O(n^2)$ operácií, ale pri zozname iba $O(n + m)$. Rozmyslite si prečo.

Listing programu (C++)

```
#include<iostream>
#include<vector>
using namespace std;

vector<vector<int> > G;

int vypis_susedov(int v) {
    for (int i = 0; i < G[v].size(); ++i) if (G[v][i] == 1) cout << i << endl;
}

void pridaj_hranu(int a, int b) {
```

```
    G[a][b] = G[b][a] = 1;
}

void zmaz_hranu(int a, int b) {
    G[a][b] = G[b][a] = 0;
}

int main(){
    int n, m, a, b;
    cin >> n >> m;
    G.resize(n,vector<int>(n,0));
    for (int i = 0; i < m; ++i) {
        cin >> a >> b;
        G[a][b] = G[b][a] = 1;
    }
}
```

Listing programu (C++)

```
#include<iostream>
#include<vector>
using namespace std;

vector<vector<int> > G;

int vypis_susedov(int v) {
    for (int i=0; i<G[v].size(); ++i) cout << G[v][i] << endl;
}

void pridaj_hranu(int a, int b) {
    G[a].push_back(b);
}

void zmaz_hranu(int a, int b) {
    for (int i = 0; i < G[a].size(); ++i) {
        if (G[a][i] == b)swap(G[a][i], G[a].back());
    }
    G[a].pop_back();
}

int main() {
    int n, m, a, b;
    cin >> n >> m;
    G.resize(n);
    for (int i = 0; i < m; ++i) {
        cin >> a >> b;
        G[a].push_back(b);
        G[b].push_back(a);
    }
}
```

Prehľadávanie do hĺbky (BFS)

Predstavme si, že sa začne z jedného vrcholu a šíriť po grafe voda. Na začiatku, akoby okamžite, zaplaví tento vrchol a. Keďže simulujeme vodu, ktorá sa rozlieva rovnomerne, ďalšiu sekundu sa rovnomerne rozleje do všetkých okolitých vrcholov. Je na nás si uvedomiť, že po prvej sekunde sú zaplavené práve vrcholy, ktoré nazývame susedmi vrcholu a. Po ďalšej sekunde sú zaplavený zatiaľ nezaplavený susedia, susedov atď. Teraz si ukážeme ako vieme takúto myšlienku využiť.

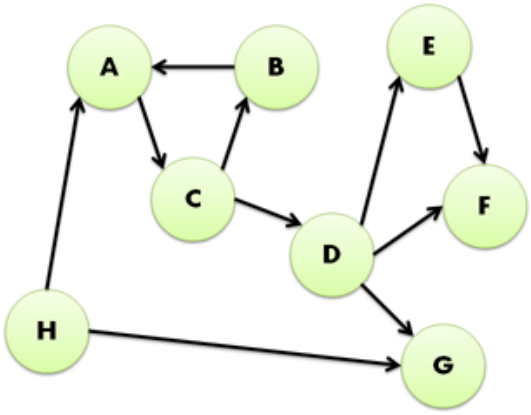
Príklad 7. Máme neohodnotený orientovaný graf. Nájdite najkratšiu cestu medzi vrcholmi H a F.

Chceme teda zistiť, na koľko tzv. krokov, sa vieme dostať z vrcholu H do vrcholu F.

Predstavme si, že sa začne z vrcholu H šíriť po grafe voda rýchlosťou hrana/sekundu. V "nultej sekunde" bude zaplavený vrchol H, v prvej sekunde všetky vrcholy na vzdialenosť 1 od H, takže všetci jeho susedia G a A. V druhej sekunde, susedia jeho susedov (C). V tretej sekunde bude zaplavený vrchol B a D a v štvrtej sekunde vrcholy E a F.

Takto sa voda postupne dostala do všetkých vrcholov, do ktorých sa dá z vrcholu H dostať. Všimnime si teraz, že algoritmus zaplaví v k-tej sekunde všetky vrcholy vo vzdialenosti k od nášho pôvodného vrcholu, čo znamená, že akonáhle je vrchol objavený, vieme na koľko najmenej krokov je možné ho dosiahnuť. Riešením je teda spustiť toto prehľadavanie z vrcholu H a skončiť vtedy, keď objavíme vrchol F.

Tento algoritmus sa nazýva prehľadávanie do hĺbky alebo aj BFS z anglického breadth-first search.



Obr. 6: Neohodnotený orientovaný graf.

Teraz niečo k implementácii tohto algoritmu:

Samotné prehľadávanie funguje tak nejak systematicky. Najprv chceme spracovať náš prvý vrchol H, ten označíme za prehľadaný a nastavíme mu vzdialenosť na 0. Všetky jeho susedné vrcholy si dáme zatiaľ na niečo ako *čakaciu listinu*, ale o všetkých už vieme, že najkratšia cesta vedúca z H má dĺžku 1, preto si to o nich poznačíme.

Keďže sme skončili s vrcholmi, ktoré majú vzdialenosť 0 tak môžeme pokračovať ďalej. Zoberieme prvého suseda z čakacej listiny, označme ho v, a všetkých jeho zatiaľ nespracovaných susedov pridáme na čakaciu listinu. Sused je zatiaľ nespracovaný, ak sme mu ešte neurčili vzdialenosť od H. Naviac však týmto vrcholom vieme nastaviť vzdialenosť od H. Bude to o jedna viac, ako je od H vzdialený vrchol v.

Tu ale pozor, dôležité je všetky tieto vrcholy dať na čakaciu listinu až za pôvodných ešte nespracovaných susedov a to z dôvodu, že všetci susedia nášho práve spracovávaného vrcholu už majú vzdialenosť 2. Takto budeme pokračovať pokým nespracujeme všetky vrcholy z čakacej listiny. Ako našu čakaciu listinu použijeme nástroj z STL nazývaný fronta alebo aj queue.

Čo sa týka *časovej zložitosti*, nemusí byť zrejmé akú zložitosť má tento algoritmus. Avšak si môžeme všimnúť, že vrchol na čakaciu listinu dáme práve raz a potom ho práve raz odoberieme, časová zložitosť je teda lineárna od počtu vrcholov ale aj od počtu hrán, keďže na každú hranu sa pozrieme 2 krát. Časová zložitosť je preto $O(n + m)$.

Pamäťová zložitosť: Počas algoritmu si pre každý vrchol musíme pamätať vzdialenosť od začiatku, čakaciu listinu a pôvodný graf. Na prvé dve veci nám stačí pole veľkosti n. No a graf si zapamätáme ako zoznam susedov, lebo musíme pre každý vrchol prejsť všetkých jeho susedov, čo zvláda táto reprezentácia lepšie. Výsledná pamäť je teda $O(n + m)$.

Listing programu (C++)

```
#include<iostream>
#include<vector>
#include<queue>
using namespace std;

vector<vector<int> > G;

void bfs(int od, int kam) {
    queue<int> Q;
    vector<int> vzd(G.size(), -1);
    vzd[od] = 0;
    Q.push(od);
    while(!Q.empty()) {
        int spracuj = Q.front();
        Q.pop();
        for (int i = 0; i < G[spracuj].size(); ++i) {
            if (G[spracuj][i] != -1) continue;
```

```
            int dalsi = G[spracuj][i];
            vzd[dalsi] = vzd[spracuj] + 1;
            Q.push(dalsi);
        }
    }
    if(vzd[kam] == -1) cout << "Z␣" << od << "␣sa␣do␣" << kam << "␣neda␣dostat." << endl;
        else cout << "Z␣" << od << "␣sa␣do␣" << kam << "␣da␣dostat␣na␣" << vzd[kam] << endl;
}

int main() {
    int n, m, a, b, k, l;
    cin >> n >> m;
    G.resize(n);

    for(int i = 0; i<m; ++i) {
            cin >> a >> b;
            G[a].push_back(b);
    }
    cin >> k >> l;
    bfs(G, k, l);
}
```

Príklad 8. Daná je mriežka rozmerov $r \times s$. Políčko je označené jedným zo symbolov: '.', 'x', 'S' alebo 'C'. Nájdite najkratšiu cestu z 'S' do 'C' pričom platí, že z každého políčka sa viete hýbať do 4 smerov a nemôžete vstúpiť na políčka označené 'x'.

V tomto príklade na prvý pohľad nevidíme žiadne vrcholy ani hrany, vstup vôbec nevyzerá ako graf a teda by sme povedali, že nejde o grafovú úlohu.

Predstavme si teraz na chvíľu, že políčka sú vrcholy grafu. Z každého políčka (vrcholu) sa môžeme pohnúť na 4 susedné políčka (vrcholy). Medzi takýmito vrcholmi môžeme teda pridať hrany. A zrazu sme si z mriežky spravili pekný graf. Políčka označené 'x' vyriešime tak, že príslušné políčka (vrcholy) aj s ich hranami odstránime.

Riešením je teda použiť prehľadávanie do šírky. Začneme na políčku s označením 'S', tomu nastavíme vzdialenosť 0, teraz si uvedomme, že každý bod mriežky, je daný dvoma číslami. Do našej fronty budeme preto hádzať práve tieto dvojice, pomocou nej totiž vieme presne určiť, kam sa vieme pohnúť ďalej.

Susedia políčka so súradnicami (y, x) sú políčka so súradnicami $(y-1, x)$ (sused o riadok vyššie), $(y, x+1)$ (sused napravo), $(y+1, x)$(sused o riadok nižšie) a $(y, x-1)$(sused naľavo). Jediné čo nám ešte treba ošetriť je aby sme nevystúpili z mriežky, čo riešime šikovne v implementácii.

Menším trikom vieme jednoducho určiť všetky políčka okolo. Predpripravíme si pole so zmenou súradníc.

`dx[]=-1, 0, 1, 0`
`dy[]=0, 1, 0, -1`

Ak si očíslujeme tieto 4 políčka v smere hodinových ručičiek od 0 do 3 tak potom políčko k má súradnice $(y+dy[k], x+dx[k])$. To nám umožnuje v jednom `for`-cykle prejsť všetky okolité vrcholy – budeme meniť k od 0 do 3.

Listing programu (C++)

```
#include<iostream>
#include<vector>
#include<queue>
#include<string>
using namespace std;

int dx[] = {-1, 0, 1, 0};
int dy[] = {0, 1, 0, -1};

void bfs(vector<string>&vstup, vector<vector<int> >&navs, int z1, int z2) {
    pair<int, int>sprac;
    queue<pair<int, int> >Q;
    navs[z1][z2] = 0;
    Q.push({ z1, z2 });

    while (!Q.empty())
    {
        sprac = Q.front();
        Q.pop();
        for (int i = 0; i < 4; ++i)if (navs[sprac.first + dx[i]][sprac.second + dy[i]] == -1 && vstup[sprac.first + dx[i]][sprac.second +
        {
            navs[sprac.first + dx[i]][sprac.second + dy[i]] = navs[sprac.first][sprac.second] + 1;
```

```
            Q.push({sprac.first+dx[i], sprac.second + dy[i] });
        }

    }
}

int main()
{
    pair<int, int>z, k;
    int r, s;
    cin >> r >> s;

    vector<string>vstup(r+2);
    vector<vector<int> >navs(r+2, vector<int>(s+2, -1));
    string riadok;

    for (int i = 0; i < s + 2; ++i)
    {
        vstup[0].push_back('x');
        vstup[r+1].push_back('x');
    }

    for (int i = 1; i < r+1; ++i)
    {
        cin >> riadok;
        vstup[i].push_back('x');
        vstup[i] += riadok;
        vstup[i].push_back('x');
    }

    for (int i = 0; i < r + 2; ++i)for (int j = 0; j < s + 2; ++j)
    {
        if (vstup[i][j] == 'Z')
        {
            z.first = i;
            z.second = j;
        }
        else if (vstup[i][j] == 'K')
        {
            k.first = i;
            k.second = j;
        }
    }

    bfs(vstup, navs, z.first, z.second);

    if (navs[k.first][k.second] == -1)cout << "Do␣ciela␣sa␣nevieme␣dostat" << endl;
    else cout << "Do␣ciela␣sa␣vieme␣dostat␣na␣" << navs[k.first][k.second] << "␣krokov" << endl;
    return 0;
}
```

STL

Prednášajúci: Andrej a Syseľ, text písal Andrej

Abstrakt

Veľa bežných algoritmov je tak často používaných, že ľudom sa už nechce ich stále programovať dookola, potom ich debugovať a optimalizovať. Na tejto prednáške si predstavíme STL – Standard Template Library, čo je knižnica obsahujúca niektoré veľmi často používané algoritmy a tiež dátové štruktúry. Výhodou používania STL je, že nástroje v nej sú už odladené a optimalizované. Taktiež používanie týchto algoritmov skracuje kód a sprehľadnuje ho, nehovoriac o cennom ušetrenom čase. STL je teda vec, ktorá sa zíde každému (súťažnému aj nesúťažnému) programátorovi píšucemu v C++.

Niečo o stavbe STL

STL je sada knižníc, ktorá obsahuje 3 dôležité časti: *algoritmy*, *kontajnery* a *iterátory*. Vy ste pravdepodobne doteraz prišli do styku s knižnicou STL pri používaní `vector`-u. Vector je jeden z najpoužívanejších *kontajnerov*. Kontajner nám slúži na uloženie dát. Kontajnery sa líšia tým, ako údaje ukladajú v pamäti a tiež tým, akú sadu funkcií vedia rýchlo vykonávať. Napríklad, ako môžete vedieť, do vectoru ide rýchlo

prvky vkladať aj vector rýchlo prechádzať, horšie je to už s vyhľadávaním prvku vo vectore, to zväčša trvá veľmi dlho.

Niečo o iterátoroch

Teraz si na vectore vysvetlíme iterátory. Iterátor je objekt, ktorý ukazuje na nejaký prvok, ktorý je obsahom kontajnera. Kontajnery sú totiž rôzne a `vector` je jedným z najjednoduchších – prvky *vectoru* sú v pamäti uložené za sebou a teda vieme ľahko povedať, kde sa nachádza ďalší prvok. Niektoré kontajnery majú však svoje prvky rozhádzané po pamäti a pozícia ďalšieho prvku sa nedá zistiť tak jednoducho. Na to nám slúžia *iterátory*. *Vector*, má dva základné iterátory: iterátor začiatku a konca. Iterátor začiatku ukazuje na prvý prvok poľa, iterátor konca ukazuje na miesto hneď za posledným prvkom poľa.

Iterátor sa deklaruje nasledovne: `vector<int>:: iterator it;`

Ako prvé si je treba všimnúť, že iterátor má zadaný presný typ kontajnera, pre ktorý vie spĺňať svoju funkciu. Iterátorov je viac druhov, neskôr si povieme aký je medzi nimi rozdiel. Každý iterátor má príslušnosť ku kontajneru, pre ktorý bol definovaný.

Iterátoru sa dá priamo priradiť iterátor začiatku vectora: `it=vec.begin();`

Všimnime si, že `begin()` je funkcia vectora, ktorá vracia iterátor ukazujúci na začiatok vectora. Keďže sme si povedali, že iterátor je v podstate ukazovateľ na nejaký prvok, on sám nemá nejakú číselnú hodnotu. Číselnú hodnotu má adresa v pamäti na ktorú iterátor ukazuje, k tejto hodnote pristúpime pomocou `*it`. Vectorový iterátor je inkrementovateľný aj dekrementovateľný, ak zvýšime jeho hodnotu o jedna, posunie sa v pamäti akoby o jedno miesto ďalej, a čo je na tomto mieste? Predsa ďalší prvok poľa. Špecialitou vectora aj jeho iterátora je, že má možný tzv. random access v konštantnom čase. To znamená, že vieme v konštantnom čase zistiť, čo sa nachádza na 3. mieste vo vectore. Pozor, nie všetky dátové štruktúry STL disponujú touto výhodou. Na iterátoroch sa to prejavuje tak, že ide napísať napríklad `it=it + 5`; Tento príkaz spôsobí, že iterátor sa pohne o 5 miest ďalej v pamäti.

Keď už približne vieme ako iterátory fungujú, môžme sa trocha pohrať a pomocou iterátoru vypísať pole.

`for(vector<int>::iterator it=vec.begin();it!=vec.end();++it)cout<<(*it)<<endl;`

Tento kód najskôr inicializuje iterátor `it` na začiatok vectoru `vec`, potom iterátor `it` inkrementuje až pokým nenarazí na špeciálny iterátor `vec.end()`. Tento iterátor ukazuje **za posledný** prvok vectoru, po dosiahnutí tohto iterátora teda už chceme vypisovanie zastaviť.

Listing programu (C++)

```
#include<iostream>
#include<vector>
using namespace std;

int main() {
    vector<int>vec = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    vector<int>::iterator it;
    it = vec.begin();
    cout << (*it) << endl;  //*it je 1
    it++;
    cout<<(*it)<<endl;  //*it je 2
    it = it + 2;
    cout<<(*it)<<endl;  //*it je 4
    it = vec.begin()+3;  //(*it) je ekvivalentné vec[3]
    cout<<(*it)<<endl;
    for(vector<int>::iterator it=vec.begin();it!=vec.end();++it)
        cout<<(*it)<<endl;
}
```

Prečo sú iterátory také dôležité? Iterátory sa používajú ako parametre funkcií z STL.

Funkcie

Väčšina užitočných funkcií STL sa skrýva pod knižnicou `<algorithm>`, čo znamená, že ak chceme tieto funkcie použiť, musíme o tom dať kompilátoru vedieť pomocou direktívy `include<algorithm>`. Určite množstvo z vás už trápil problém usporadúvania čísel. Tento problém má veľmi veľa riešení, niektoré sa implementujú ťažšie iné jednoduchšie, avšak za cenu horšej časovej zložitosti. Našťastie STL prináša funkciu `sort()`. Funkcia `sort()` usporiada n prvkov v garantovanej časovej zložitosti $O(n \cdot \log(n))$. Jediné, čo musíme dať tejto funkcii je interval odkiaľ pokiaľ sa nachádzajú prvky, ktoré má usporiadať. Funkcia `sort()` si berie najmenej 2 parametre a to iterátor, ktorý ukazuje na začiatok oblasti, ktorá sa má usporiadať a iterátor ukazujúci tesne za túto oblasť. Pozor, nie na posledný prvok ale na miesto **za posledným** prvkom.

Listing programu (C++)

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;

int main() {
    vector<int>vec1 = {9, 8, 7, 6, 5, 4, 3, 2, 1};
    vector<int>vec2 = {9, 8, 7, 6, 5, 4, 3, 2, 1};
    sort(vec1.begin(), vec1.end());
    sort(vec2.begin(), vec2.begin()+3);
    cout << "Prve :"<<endl;
    for (vector<int>::iterator it = vec1.begin(); it != vec1.end(); ++it)
        cout<<*it << endl;
    cout << "Druhe :"<<endl;
    for (vector<int>::iterator it = vec2.begin(); it != vec2.end(); ++it)
        cout << *it << endl;
}
```

Všimnime si, čo spravilo prvé a čo druhé volanie funkcie `sort()`. Prvé volanie funkcie `sort()` dostalo usporiadať celý vector `vec1`. Druhé volanie malo usporiadať všetko od začiatku vectora až po index 2. Iterátor `vec2.begin()+3` síce ukazuje na index 3, funkcia sort si však ako druhý parameter berie, ako sme už spomínali, iterátor na prvý prvok, ktorý už **ne**usporadúva.

Niekoľko ďalších užiteľných funkcii v STL pracuje rovnako ako `sort()`, napríklad funkcia `reverse()`, ktorá otočí pole. Nebudeme sa im však špeciálne venovať. Iba poznamenáme, že všetky tieto funkcie berú ako parameter nejaký interval čísel daný začiatočným iterátorom a iterátorom za posledným prvkom tohto intervalu.

Šikovnými funkciami sú aj funkcie `swap()`, `min()` a `max()`. Všetky tieto funkcie berú dva parametre a a b. Funkcia `swap()` zamení ich hodnoty, `min()` vráti menšiu hodnotu z dvojice a a b a `max()` väčšiu.

STL obsahuje množstvo užitočných funkcií a my sme si predstavili iba zopár z nich. Preto odporúčame sa bližšie zoznámiť s ďalšími funkciami STL a naplno ich využívať pri programovaní. Vždy si však dajte pozor na to, akú zložitosť má daná funkcia.

Listing programu (C++)

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;

int main() {
    vector<int>vec1 = { 12, 13, 14, 15, 16, 17, 18, 1, 2 };
    vector<int>vec2=vec1, vec3=vec1;
    int a=1000, b=5;
    double k = 20.056, l = 21.1589;

    reverse(vec1.begin() + 3, vec1.begin() + 6);

    fill(vec2.begin(), vec2.end(), 5);

    sort(vec3.begin(), vec3.end());
    reverse(vec3.begin(), vec3.end());
```

```
    cout << "Prvy␣vector␣:";
    for (vector<int>::iterator it = vec1.begin(); it != vec1.end(); ++it)
        cout<<*it << "␣";
    cout << endl;

    cout << "Druhy␣vector␣:";
    for (vector<int>::iterator it = vec2.begin(); it != vec2.end(); ++it)
        cout << *it << "␣";
    cout << endl;

    cout << "Treti␣vector␣:";
    for (vector<int>::iterator it = vec3.begin(); it != vec3.end(); ++it)
        cout << *it << "␣";
    cout << endl;

    if (k == min(k, l))cout << "K␣je␣mensie" << endl;
    else cout << "L␣je␣mensie" << endl;

    swap(k, l);
    swap(a, b);
    cout << "a␣je␣teraz" << a << "␣a␣b␣je␣teraz␣" << b << endl;
}
```

Pair

Pair je jedna zo základných dátových štruktúr v STL. Je veľmi jednoduchá pretože je to len dvojica čísel v jednej premennej. Deklaruje sa `pair<typ1, typ2> x;` Pair v sebe vlastne nesie dve premenné – jednu typu `typ1` a druhú typu `typ2`. K prvej premennej vieme pristúpiť pomocou kľúčového slova *first*. `x.first` je hodnota prvej premennej typu `typ1`. Hodnota `x.second` je hodnota druhej premennej typu `typ2`. Pair môže vyzerať napríklad takto: `pair<int, int>`. Samozrejme, že ide vytvoriť pole pair-ov. Pomocou neho si vieme napríklad dobre reprezentovať body v rovine. Ak na pair aplikujeme porovnanie, tak sa najskôr porovná prvý prvok a podľa toho sa rozhodne, ktorý pair je menší. Ak nastane na prvom prvku rovnosť, rozhoduje druhý prvok.

Listing programu (C++)

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;

int main() {
    pair<int, int>a, b;
    a.first = 5;
    a.second = 10;
    b = { 12, 4 };
    vector<pair<int, int> >pole;
    pole.push_back({ 1, 3 });
    pole.push_back({ 5, 0 });
    pole.push_back({ 5, 1 });
    pole.push_back({ 12, -3 });

    sort(pole.begin(), pole.end());
    for (int i = 0; i < pole.size(); ++i)
        cout << i << "-ty␣bod␣je␣[" << pole[i].first << ";" << pole[i].second << "]" << endl;
    swap(a, b);
}
```

Set

Set, jeden z najužitočnejších kontajnerov v STL. Implementáciou sa diametrálne odlišuje od vectora, to ale my nebudeme riešiť a predstavíme si jeho funkcie bez toho, aby sme zisťovali, ako presne set pracuje. Set funguje ako čierna krabička, do ktorej ide prvky vkladať a vyberať. Tieto dve operácie však netrvajú konštantý čas $O(1)$ ako pri vectore ale $O(log(n))$, kde n je počet prvkov v sete. Treba teda počítať, že vkladanie a vyberanie prvkov do a zo setu môže trvať dosť dlho.

Načo nám je dátová štruktúra, ktorá má tieto dve operácie pomalšie ako vector? Nestačí nám teda iba vector? Set ukrýva jednu veľmi silnú funkciu a to `find()`. Find vie odpovedať na otázky typu: Nachádza

sa v set-e prvok `x`? v čase $O(log(n))$. Je treba si uvedomiť, že vector vie na takéto otázky odpovedať iba v čase $O(n)$, čo je pre veľké n obrovský rozdiel. Naviac, vie set odpovedať na otázky typu: Nájdi prvý väčší/menší prvok ako `x` v danom set-e. Okrem toho sú v sete prvky stále usporiadané, keď ho teda budeme prechádzať, vieme vypisovať tieto prvky rovno vzostupne usporiadané. Jedinou nevýhodou je, že iterátor set-u nemá random access (ale stále je inkrementovateľný), čo v preklade znamená, že nevieme v konštantnom čase povedať, aký je k-ty najmenší prvok v set-e.

Keďže set reprezentuje množinu prvkov, nie je možné v ňom mať prvky duplicitne, preto každý prvok môže byť v sete iba raz. Ak sa pokúsime pridať do set-u prvok, ktorý už v set-e je, nebude tam tento prvok po našom pridaní 2 krát ale stále len raz! Ak by vám táto vlastnosť nevyhovovala, môžete použiť kontajner `multiset`, ktorý sa tiež nachádza v STL.

Dôležitými sú ešte funkcie `lower_bound()` a `upper_bound()`. Obidve tieto funkcie si ako parameter berú prvok x rovnakého typu ako sú prvky v sete, pričom `upper_bound()` vracia iterátor ukazujúci na prvý prvok vačší ako x. `lower_bound()` vracia iterátor na prvý prvok menši alebo rovný ako x. Netreba zabúdať, že tieto dve funkcie vracajú iterátor a teda ak chceme zistiť reálnu hodnotu skrývajúcu sa za týmito iterátormi, musíme použiť operátor `*`.

Listing programu (C++)

```
#include<iostream>
#include<set>
#include<algorithm>
using namespace std;

int main() {
    int x=5;
    set<int>S;
    S.insert(6); S.insert(7); S.insert(5); S.insert(8); S.insert(8);
    cout << "V set-e sa nachadzaju prvky : ";
    for (set<int>::iterator it = S.begin(); it != S.end(); ++it)cout << *it << " ";
    cout << endl;

    if (S.find(x) != S.end())cout << x << "sa v set-e nachadza!" << endl;  //S.find() pri nenájdení prvku x vracia iterátor na koniec set
    else cout << x << "sa v set-e nenachadza!" << endl;

    //lower a upper bound
    cout << *S.upper_bound(7) << endl;
    cout << *S.lower_bound(7) << endl;
}
```

Všimnime si najmä, že set má rovnaký iterátor ukazujúci na začiatok a rovnaký iterátor ukazujúci na koniec ako vector.

Map

Map je veľmi špecifická dátová štruktúra, je to množina dvojíc [kľúč, hodnota]. Map je vo svojej implementácii podobná setu. Ukladá si kľúče a pomocou kľúča vieme pristupovať k hodnote. Nájsť v sebe hodnotu prislúchajúcu kľúču vie v čase $O(log(n))$. Map si môžeme predstaviť ako špeciálny vector – vo vectore sú kľuče indexy, teda čísla, v mape to može byt aj napr. `char` alebo `string`. Za indexom, resp. v mape za kľúčom, sa ukrýva nejaká premenná (`int`, `char`, `string`, ...).

Príklad 1. Máme daný vector obsahujúci n čísiel z rozsahu 1 až 2 000 000 000. Zistite pre každé číslo, koľko krát sa v poli vyskytuje. Platí $n \leqslant 100\,000$.

Riešenie: Každé číslo bude kľúčom, za ktorým sa ukrýva hodnota, ktorá hovorí, koľko krát sa dané číslo vyskytuje v poli.

Listing programu (C++)

```
vector<int>vec;//do tohto vectora nacitame vstup
...
map<int, int> x;
for (int i = 0; i<vec.size(); ++i)x[vec[i]]+=1;
for (map<int, int>::iterator it = x.begin(); it != x.end(); ++it)
{
    cout << it->first << "␣sa␣v␣poli␣nachadza␣" << it->second << "krat" << endl;
}
...
```

Queue

Queue alebo aj fronta je dynamická dátová štruktúra, v ktorej ide prvky odoberať z predu a pridávať na koniec. Pripomína teda rad v supermarkete: kto skôr prišiel, ten sa aj skôr dostane k pokladni a odíde. Queue vie v čase $O(1)$ pridať prvok na koniec a v rovnakom čase odobrať prvok zo začiatku. Pritom na začiatku je vždy ten prvok, ktorý je v queue najdlhšie. Táto dátová štruktúra má 2 dôležité funkcie `push()` a `pop()`. Funkcia `pop()` vymaže prvý prvok z fronty a neberie si žiadne parametre. Metóda `push(x)` si berie ako parameter x a ten vloží na koniec za všetky prvky ležiace momentálne vo fronte. Poslednou užitočnou funkciou je funkcia `front()`, ktorá vráti hodnotu prvku, ktorý je práve vo fronte prvý.

Listing programu (C++)

```
#include<iostream>
#include<queue>
using namespace std;

int main() {
    queue<int>Q;
    Q.push(5);
    Q.push(7);
    Q.push(8);
    while (!Q.empty()) {
        cout << "Na␣vrchu␣je␣prave␣" << Q.front() << endl;
        Q.pop();
    }
}
```

Priority queue

Táto dátová štruktúra je známa aj pod menom halda. Funguje na podobnom princípe ako fronta: tiež do nej môžeme vkladať prvky a vyberať prvky z vrchu. Ako však napovedá už názov, na vrchu `priority_queue` nie je prvok, ktorý je tam najdlhšie, ale prvok s najväčšou prioritou. Pre známe dátové typy ako čísla a podobne je vždy na vrchole najvačší prvok. Vloženie prvku do haldy trvá čas $O(log(n))$ a v rovnakom čase sa z haldy dá aj vyberať. Halda je veľmi používaná, ak nás príliš nezaujíma, aké prvky sú práve v kontajneri ale chceme vedieť, ktorý je najväčší alebo najmenší. Keď sa s haldou spoznáte trošku bližšie, naučíte sa písať si svoje funkcie, ktoré budú určovať, ktorý prvok má najväčšiu prioritu. Môže to byť napríklad aj ten najmenší (ak vás zaujíma minimum).

Listing programu (C++)

```
#include<iostream>
#include<queue>
using namespace std;

int main() {
    priority_queue<int>Q;
    Q.push(5);Q.push(7);
    Q.push(8);Q.push(9);
    Q.push(2); Q.push(-5);
    while (!Q.empty()) {
        cout << "Na␣vrchu␣je␣prave␣" << Q.top() << endl;
        Q.pop();
    }
}
```