Dve chuťovky

Na úvod sa zoznámime s dvoma „chuťovkami“, ktoré nám poslúžia ako vstupná brána do sveta dátových štruktúr – transpozíciou matice a štruktúrou Union-Find. Prvý problém, transpozícia matice, sa na prvý pohľad môže javiť ako triviálny. Veď čo už by sa na ňom dalo pokaziť alebo vylepšiť? Ukážeme si, že tento zdanlivo jednoduchý problém skrýva zaujímavé prekvapenia, ktoré vyjdú na povrch najmä pri praktickej implementácii.

S druhým problémom preskočíme do teoretickej roviny. Predstavíme si elegantnú a veľmi jednoduchú dátovú štruktúru s názvom Union-Find (alebo aj disjunktné množiny). Hoci jej implementácia zaberá zopár riadkov zdrojového kódu, analýza časovej zložitosti patrí medzi zložitejšie kapitoly z dátových štruktúr. Ukážeme si úplne nový prístup, ako možno nazerať na analýzu časovej zložitosti.

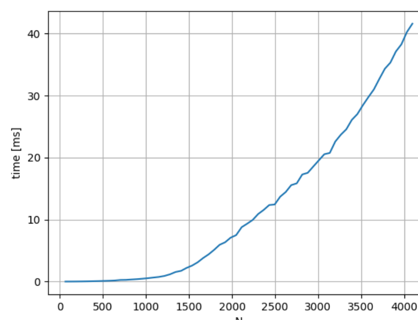
Transpozícia

Majme štvorcovú maticu rozmeru $N \times N$ a našou úlohou je transponovať ju, teda „preklopiť“ podľa hlavnej diagonály. Výsledkom transpozície je nová matica, v ktorej sa prvky na pozíciách (i, j) a (j, i) navzájom vymenia. Inak povedané, riadky sa stanú stĺpcami a stĺpce riadkami.

Otázka znie: ako na to?

Tu je priamočiare riešenie:

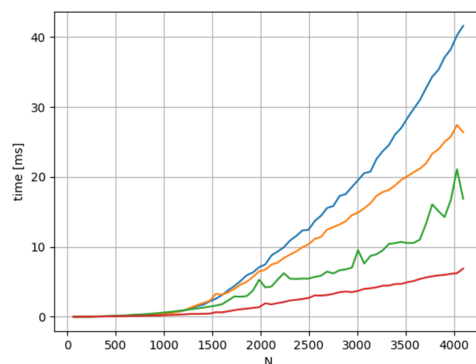
```
const int N = 1000;
int A[N][N];

void transpose() {
  for (int i=0; i<N; ++i)
    for (int j=i+1; j<N; ++j)
      swap(A[i][j], A[j][i]);
}
```

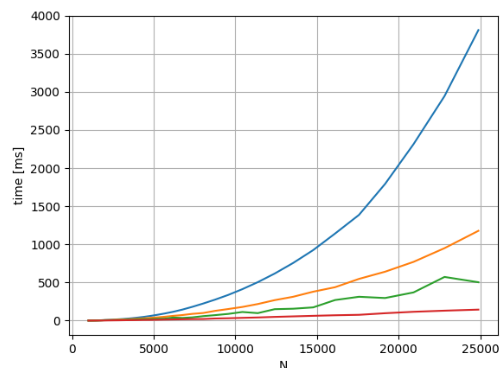


Časová zložitosť je zjavne $\Theta(N^2)$, takže graf bude vyzerať nejak takto, však? (Však?)

Taktiež je zjavné, že musíme premiesniť $\Theta(N^2)$ prvkov. Akýkoľvek algoritmus musí spraviť $\Theta(N^2)$ čítaní a zápisov do pamäte, takže lepšie to nejde.

Šok #1: existujú lepšie algoritmy:

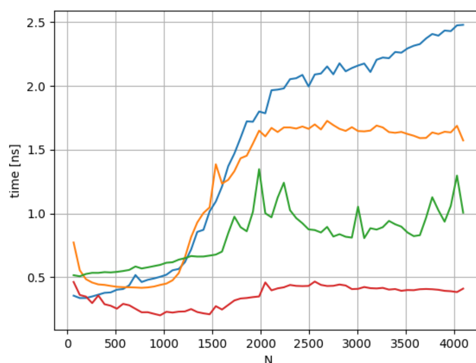


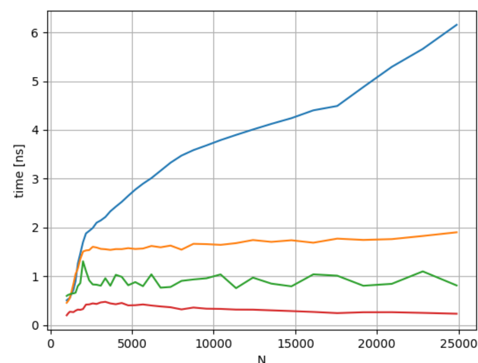


Vľavo je graf pre stredne veľké $N \leq 4\,000$, vpravo je ten istý graf pre veľké N až po 25 000. Čas je v milisekundách. Náš kód je modrá čiara – teda to najpomalšie riešenie. Ako sa to dá lepšie?

Mimochodom, tvrdili sme, že časová zložitosť nášho riešenia rastie ako druhá mocnina N. Graf naozaj pripomína parabolu, ale najlepší spôsob ako sa o tom presvedčiť je, že do grafu nanesieme „čas deleno N^2“, a vyjde nám konštanta. (Však?)

Šok #2: Čas deleno N^2 nie je konštanta?

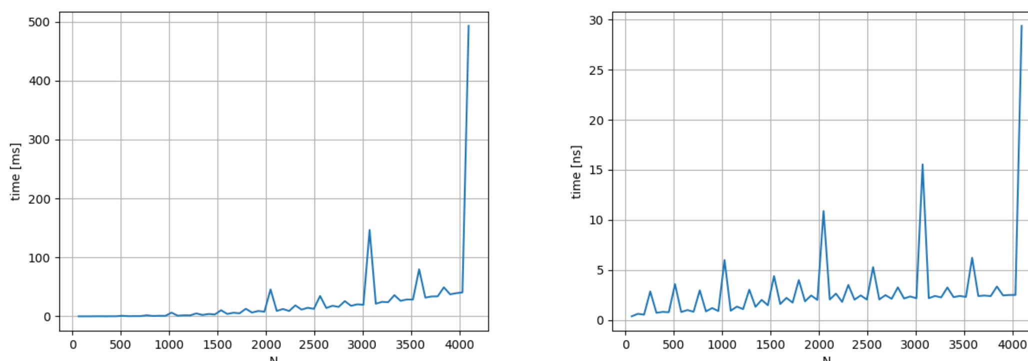




Tu vidíme čas deleno N^2, čo môžeme interpretovať ako čas v prepočte na jeden prvok matice, v nanosekundách.

Prečo to stúpa? Prečo to nie je konštanta??

Šok #3: Klamal som. V skutočnosti graf pre náš kód vyzerá takto:



Vľavo celkový čas v milisekundách, vpravo delené N^2 v nanosekundách.

Čo majú, do čerta, znamenať tie zuby??? Čo sú tie špicaté výkyvy v čase behu?

(Nie, nie je to chyba merania.)

Prvým zlepšením základného Union-Find algoritmu je technika nazývaná Union by Rank. Jej cieľom je udržiavať stromy čo najplytšie, aby operácia find(x) bola rýchla.

Zavedieme pojem rank pre každý vrchol, ktorý slúži ako odhad výšky stromu. Na začiatku má každý vrchol rank rovný nule — keďže každý je v samostatnej jednoprvkovej množine (strom výšky 0).

Operácia union(x, y) potom funguje nasledovne:

Najprv nájdeme korene stromov, do ktorých patria x a y.

Potom pripojíme strom s menším rankom pod strom s väčším rankom.

Ak majú oba stromy rovnaký rank, jeden z nich (ľubovoľný) sa stane koreňom a jeho rank sa zvýši o 1.

Dúfam, že vás tieto výsledky aspoň trochu prekvapili – a možno aj zmiatli. To bol cieľ. Zatiaľ vám odpovede neprezradím a nechám vás trochu sa potrápiť s vlastnými hypotézami. Prejdime teraz k druhej „chuťovke“ a na záver sa k transpozícii vrátime a uzavrieme celú kapitolu s lepším pochopením toho, čo sa vlastne deje „pod kapotou“.

Union-find

Chceme si udržiavať *neorientovaný graf*, ktorý sa dynamicky vyvíja – postupne doň *pridávame hrany* a zároveň chceme *rýchlo zisťovať súvislosť medzi vrcholmi*. Chceli by sme teda dátovú štruktúru, ktorá podporuje nasledujúce operácie:

- *join*(x, y): prepojí vrcholy x a y hranou,
- *connected*(x, y): zistí, či medzi vrcholmi x a y existuje cesta.

Takáto funkcionalita je napríklad presne to, čo potrebujeme v Kruskalovom algoritme na hľadanie najlacnejšej kostry v grafe (problému minimálnej kostry sa ešte budeme viac venovať v časti o haldách) alebo pri unifikácii.

Každú *množinu reprezentujeme ako strom*, pričom *koreň stromu slúži ako reprezentant celej množiny.* Všetky prvky v množine „ukazujú smerom nahor" – teda smerom ku koreňu.

Operácie potom vyzerajú nasledovne:

- *find*(x): Prechádzame od vrcholu x nahor, až kým nenájde koreň, teda reprezentanta množiny obsahujúcej prvok x.
- *union*(x, y): Najprv nájdeme korene stromov, v ktorých sa nachádzajú x a y. Ak ide o ten istý strom (teda už patria do tej istej množiny), neurobíme nič. Inak napojíme jeden strom pod druhý, konkrétne jeden koreň nastavíme ako syna toho druhého.

Toto riešenie je veľmi jednoduché, má však jednu zásadnú nevýhodu: Operácia find môže trvať čas úmerný výške stromu – a keďže bez ďalších optimalizácií sa môžu stromy „natiahnuť" do jednej dlhej reťaze, v najhoršom prípade môže byť časová zložitosť až lineárna.

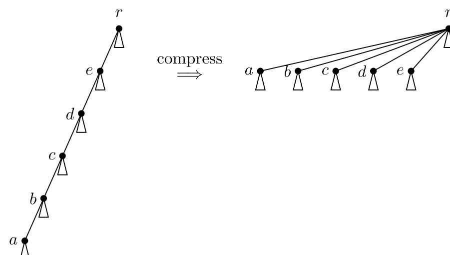
Vylepšenie #1: Union by Rank. Prvým vylepšením základného algoritmu je technika nazývaná *Union by Rank.* Cieľom je udržiavať stromy čo najplytšie, aby bola operácia find(x) rýchla.

Zavedieme pojem rank pre každý vrchol, ktorý slúži ako odhad výšky stromu. Na začiatku má každý vrchol rank rovný nule – keďže každý je v samostatnej jednoprvkovej množine (strom výšky 0). Pri operácia union(x, y) potom vždy pripojíme strom s menším rankom pod strom s väčším rankom. Ak majú oba stromy rovnaký rank, jeden z nich (ľubovoľný) sa stane koreňom a jeho rank sa zvýši o 1.

Indukciou ľahko dokážeme, že:

- strom s rankom r má výšku najviac r
- strom s rankom r má aspoň 2^r vrcholov
- teda rank r môže mať najviac $n/2^r$ vrcholov
- z toho vyplýva, že maximálny rank je $\lg n$, čo je zároveň maximálna hĺbka vrcholov a teda časová zložitosť klesne na $O(\lg n)$

Vylepšenie #2: Path Compression. Druhé vylepšenie, ktoré dramaticky znižuje časovú zložitosť operácií, sa nazýva path compression, teda „stlačenie cesty". Princíp je jednoduchý: *Po tom ako počas operácie find(x) nájdeme koreň, sa vrátime naspäť a všetky vrcholy, ktoré sme navštívili po ceste, napojíme priamo pod tento koreň.* Týmto spôsobom sa výrazne zníži hĺbka stromu pre všetky budúce volania *find* nielen pre x, ale aj pre skoro všetky vrcholy na ceste ku koreňu a vrcholy v ich podstromoch.



Na prvé počutie to môže znieť ako komplikovaný algoritmus, avšak jeho implementácia je veľmi jednoduchá. Ako prvé si stačí uvedomiť, že na rozdiel od iných štruktúr (napr. vyhľadávacie stromy alebo písmenkové stromy), kde si potrebujeme pre každý vrchol pamätať všetkých jeho synov, v union-finde prechádzame stromy výlučne smerom nahor, takže si stačí pamätať pre každý vrchol smerník na jeho otca. Dokonca môžeme vrcholy očíslovať $1, 2, 3, \ldots, n$ a v obyčajnom poli si zapamätať pre každý vrchol x jeho otca $p[x]$ (pre korene si môžeme poznačiť napr. $p[x] = 0$).

Okrem poľa otcov si ešte potrebujeme pamätať ranky. Mohli by sme si ich pamätať v samostatnom poli, avšak keď si uvedomíme, že ranky potrebujeme poznať iba pre korene (a korene nemajú otca), môžeme informáciu o rankoch napchať do toho istého poľa. Použijeme nasledovný trik: pre vrcholy, ktoré nie sú koreň, bude $p[x] > 0$ číslo ich rodiča; pre korene si v $p[x]$ uložíme hodnotu $-\operatorname{rank}(x) \leq 0$.

Tu je jednoduchá implementácia na ∼15 riadkov:

```
int p[MAX]; // ak p[x] > 0, p[x] je otec x
            // ak p[x] <= 0, tak -p[x] je rank(x)

int find(int x) {
  if (p[x] <= 0) {
    return x; // koren
  } else {
    int root = find(p[x]); // rekurzivne
    p[x] = root;
  }
}

void union(int x, int y) { link(find(x), find(y)); }

void link(int rx, int ry) {  // predpokladame, ze rx, ry su korene
  if (rx == ry) return;                       // a)
  if (p[rx] == p[ry]) { p[rx]=ry; p[ry]--; } // b)
  else if (-p[rx] < -p[ry]) p[rx]=ry;        // c)
  else p[ry] = rx;                            // d)
}
```

a) rovnaké stromy – nerobíme nič

b) r_x aj r_y majú rovnaký rank; napojíme r_x pod r_y a r_y zvýšime rank (pripomeňme, že v poli p máme uloženú hodnotu $-rank$, takže túto hodnotu znížime o 1)

c) r_x má menší rank, takže napojíme r_x pod r_y

d) r_y má menší rank, takže napojíme r_y pod r_x

A tu je triková randomizovaná implementácia na dva riadky (s tým rozdielom, že pre otca koreňa si poznačíme 0 a pri rozhodovaní, ktorý strom podpojíme pod ktorý pri unione, si hodíme mincou):

```
int f(int x) { return p[x] ? p[x] = f(p[x]) : x;  }
void u(int x, int y) { rand() % 2 ? p[f(x)] = f(y) : p[f(y)] = f(x); }
```

Analýza union-findu

Všimnime si, že každá operácia union pozostáva z dvoch volaní find (na nájdenie koreňov príslušných množín), plus prelinkovania, čo je však len konštantne veľa práce. Preto sa pri analýze časovej zložitosti zameriame iba na operáciu find.

Ďalej si uvedomme, že časová zložitosť operácie *find*(x) je úmerná dĺžke cesty od vrcholu x ku koreňu, teda počtu „skokov“, ktoré musíme vykonať, aby sme sa ku koreňu dostali.

Dokážeme:

Veta 1. *Ak máme najviac $n < 2^{65\,535}$ prvkov, potom ľubovoľná postupnosť m volaní* find *vykoná najviac $6m + 2n$ skokov.*

Vo všeobecnosti platí, že ľubovoľná postupnosť m operácií union a find má celkovú časovú zložitosť $O((m+n)\log^* n)$, kde $\log^*$ je tzv. iterovaný logaritmus.

Len pre porovnanie: milión je $10^6 < 2^{20}$, miliarda je $10^9 < 2^{30}$. Počet atómov v pozorovateľnom vesmíre sa odhaduje na menej ako 2^{300} (niekde medzi 10^{78} a 10^{82}). Inými slovami, *v praxi* nikdy nebudeme mať viac ako $2^{65\,535}$ prvkov. Ani náhodou. A teda pre všetky praktické účely je zložitosť union-findu konštantná (v priemere na jednu operáciu).

A v teórii? V teórii sa, samozrejme, zaujímame, ako sa algoritmus chová pre n rastúce do nekonečna a po $2^{65\,535}$ ešte nasleduje veľa čísiel. V skutočnosti zložitosť union-findu (v priemere na jednu operáciu) *nie je* konštantná, ale rastie veľmi veľmi pomaly, pomalšie ako iterovaný logaritmus.

Čo je to iterovaný logaritmus? Zoberte číslo n a zadajte ho do kalkulačky. Potom opakovane stláčajte tlačidlo „`log`“, kým výsledok neklesne na hodnotu menšiu alebo rovnú 1. Počet stlačení logaritmu, ktoré ste museli vykonať, je práve $\log^* n$.

Napríklad: $\log^* 65\,536 = 4$, pretože:

$$65\,536 \to 16 \to 4 \to 2 \to 1;$$

potrebovali sme stlačiť „`log`“ štyrikrát. Kolko je $\log^* 2^{65\,536}$? 5, pretože z $2^{65\,536}$ sa dostaneme na 65 536 už po jednom logaritme, a ďalej pokračujeme ako v predchádzajúcom príklade. Takže celkovo potrebujeme iba 5 stlačení.

Iterovaný logaritmus pre $n \to \infty$ rastie do nekonečna:

$$\log^* 2^{2^{65\,536}} = \log^* 2^{2^{2^{2^{2}}}} = 6, \qquad \log^* 2^{2^{2^{65,536}}} = \log^* 2^{2^{2^{2^{2^{2}}}}} = 7, \text{ atď.}$$

Každým pridaním ďalšieho poschodia zväčšíme iterovaný logaritmus o 1, takže ako nám bude vežička exponentov rásť do nekonečna, bude iterovaný logaritmus rásť donekonečna. Hoci nepredstaviteľne pomaly.

■ **Dôkaz.** Začnime niekoľkými jednoduchými pozorovaniami:

- Vrcholy s vysokým rankom sú zriedkavé. Pripomeňme si, že strom s rankom r obsahuje aspoň 2^r vrcholov. To znamená, že najviac $n/2^r$ vrcholov môže dosiahnuť rank (a teda aj výšku) r. Napríklad iba 16-tina vrcholov môže mať rank aspoň 4 a iba 65 536-tina vrcholov môže dosiahnuť rank 16.
- Rank rodiča je vždy aspoň o 1 vyšší ako rank jeho detí. To znamená, že ak ideme po ceste od ľubovoľného vrcholu ku koreňu, hodnoty rankov striktne narastajú.
- Každý vrchol začína s rankom 0. Rank sa môže zvyšovvať len pokým je daný vrchol koreňom. Akonáhle vrchol pripojíme pod iný, jeho rank sa už nikdy nezmení. Jeho rodičovi však môže rank ďalej rásť.
- Pre všetky vrcholy okrem koreňov definujeme hodnotu $gap(x)$ ako rozdiel $gap(x) = rank(parent(x)) - rank(x)$. Pri každom použití path compression (t.j. stlačení cesty) sa všetky vrcholy okrem koreňa a jeho bezprostredného dieťaťa presunú pod „vyššieho“ rodiča, teda pod vrchol s vyšším rankom. Výsledkom je, že $gap(x)$ sa pre všetky tieto vrcholy zvýši.

Rozdeľme ranky do troch úrovní:

- Úroveň 0: ranky od 0 do 3,
- Úroveň 1: ranky od 4 do $2^4 - 1 = 15$,
- Úroveň 2: ranky od 16 do $2^{16} - 1 = 65\,536$,
- . . . a mohli by sme takto pokračovať, ale pre $n < 2^{65\,536}$ viac úrovní nebude.

Poďme teraz spočítať celkový počet „skokov“ vykonaných počas všetkých m operácií find dokopy. Všetky skoky rozdelíme do štyroch kategorií:

1. **Úroveň 0 (ranky 0–3):** Na tejto úrovni strávime najviac 3 skoky pri každej operácií, teda spolu najviac $3m$ skokov.
2. **Finálne skoky ku koreňu:** Pre každý find(x) si osobitne započítame posledný skok do koreňa. To je najviac m skokov.

3. **Skoky medzi úrovňami:** Keď počas find preskočíme z jednej úrovne do druhej, každý takýto „medziúrovňový“ skok započítame zvlášť. Keďže máme len 3 úrovne, týchto skokov je najviac $2m$.

4. Zostáva spočítať **skoky vo vnútri úrovní 1 a 2, ktoré nie sú finálne.** A tu prichádzame k pointe dôkazu (prečo skoky delíme na kategórie, ktoré rátame zvlášť):

   (a) **Všetky skoky v rámci úrovne 1:** sú také skoky, z vrcholu x do jeho rodiča $y = parent(x)$, že oba vrcholy x aj y majú ranky v rozmedzí $4 \leq rank(x) < rank(y) < 16$. To znamená, že $gap(x) = rank(parent(x)) - rank(x)$ je najviac 11. Avšak keďže počítame iba nefinálne skoky, y nie je koreň a má otca z s ešte väčším rankom. A tu využijeme, že používame kompresiu cesty: Pri každom finde sa pri kompresii cesty x napojí na koreň, t.j. na z alebo ešte vyššie. Tým pádom hodnota $gap(x)$ vzrastie aspoň o 1. To znamená, že po 11-tich findoch, ktoré prechádzajú cez x a skok z x na otca nie je finálny, narastie $gap(x)$ aspoň na 12 a to znamená, že x už bude napojený na vrchol na vyššej úrovni. To znamená, že všetky ďalšie findy prechádzajúce cez x už budú medziúrovňové a tie tu nepočítame (už sme ich započítali v bode 3.).

   Zhrnutie: Cez každý vrchol spravíme najviac 11 nefinálnych skokov v rámci úrovne 1. Zároveň však platí, že na úroveň 1 sa dostane iba 16-tina vrcholov. Z toho vyplýva, že celkový počet skokov v rámci úrovne 1 je najviac $11 \times (n/16) < n$.

   (b) **Všetky skoky v rámci úrovne 2:** spočítame podobne: Ranky sú tu od 16 do $2^{16} - 1$, takže $gap(x)$ musí byť menej ako 2^{16}. Po 2^{16} findoch cez x narastie $gap(x)$ tak, že otec x už nutne patrí na úroveň 3 alebo vyššie.

   Takže spravíme menej ako 2^{16} operácií na každý vrchol na 2. úrovni. Avšak 2. úroveň znamená rank aspoň 16 a iba 2^{16}-tina všetkých vrcholov tento rank niekedy dosiahne. Spolu to je teda menej ako $2^{16} \times (n/2^{16}) = n$ skokov.

Takto dostávame, že všetkých skokov je najviac $6m + 2n$ (ak $n < 2^{65\,535}$). Skúste si rozmyslieť, ako treba dôkaz upraviť pre všeobecné n.[1] □

Aká je skutočná zložitosť tejto dátovej štruktúry? Ukázali sme, že union-find so spájaním podľa ranku a kompresiou cesty trvá najviac $O((m +$

[1] Pre všeobecné n by sme namiesto 3 úrovní mali $\log^* n$ úrovní: Každá ďalšia úroveň by mala ranky v rozsahu $[k, 2^k)$, takže na i-tej úrovni by boli ranky od $\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{i+1}$ po $\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{i+2} - 1$.

V kategóriach 1. a 2. je rovnako veľa skokov, ale v kategórií 3. bude $O((\log^* n) \times m)$ medziúrovňových skokov a 4. v rámci každej úrovne spravíme najviac n nefinálnych skokov, ale máme $\log^* n$ úrovní, takže spolu $O((\log^* n) \times n)$.

$n) \log^* n)$, teda $\log^* n$ na jednu operáciu (ak $m \geq n$). Ale aká je *skutočná* zložitosť tejto dátovej štruktúry? Možno si poviete: *„Nie je v skutočnosti tá zložitosť konštantá? Nie je toto všetko len slabá analýza? Keby sme sa viac snažili, nevedeli by sme dokázať lepšiu hornú hranicu?“*

Odpoveď znie: Áno a Nie. R. E. Tarjan 1975; R. Tarjan a Van Leeuwen 1984 dokázal lepší odhad: že m union-find operácií trvá $O(m\alpha(m,n))$, kde α je funkcia, ktorá súvisí s inverznou Ackermannovou funkciou, ktorá síce rastie do nekonečna, ale ešte oveľa pomalšie ako iterovaný logaritmus. Pre všetky praktické odhady môžeme predpokladať, že $\alpha(m,n) \leq 3$. Fascinujúce je, že v zápätí ukázal aj dolný odhad (R. E. Tarjan 1979), tzn. našiel takú postupnosť operácií union a find, pri ktorej spravíme aspoň $\Omega(m\alpha(m,n))$ skokov! To znamená, že *skutočná* zložitosť *nie je* konštantná a naozaj v najhoršom prípade rastie do nekonečna, hoci veľmi veľmi pomaly.

Späť k transpozícii matíc

Poďme si konečne prezradiť, čo sa to vlastne porobilo pri transpozícii matíc. Odpovede na všetky záhady súvisia s pamäťovým systémom moderných počítačov a vyrovnávacou pamäťou (cache).

Moderné procesory sú extrémne rýchle – vykonávajú miliardy operácií za sekundu. Naproti tomu hlavná pamäť (RAM) je relatívne pomalá – čas na čítanie/písanie do RAM je rádovo sto-krát pomalšie ako je rýchlosť samotného procesora. To znamená, že ak by procesor musel na každý údaj čakať, kým dorazí z RAM, väčšinu času by sa nudil. Aby sa tento problém vyriešil, moderné procesory majú viacúrovňový systém cache pamätí – malých, ale extrémne rýchlych pamäťových blokov, ktoré uchovávajú najčastejšie používané údaje. Súčasné počítače mávajú tri úrovne cache: tzv. L1, L2, a L3, každá ďalšia úroveň je väčšia, zato pomalšia. Napríklad počítač, na ktorom píšem túto kapitolu má 256KiB L1d dátovej cache, 256KiB inštrukčnej L1i cache (sem idú inštrukcie, ktoré procesor vykonáva), 8MiB L2 cache, 16MiB L3 cache a 58GiB RAM. Na tejto architektúre má každý procesor svoju vlastnú L1 a L2 cache, ale L3 a RAM sú spoločné.

Keď chce procesor načítať nejaký údaj z pamäti, najskôr sa pozrie do L1 cache, ak tam údaj nie je (tzv. L1 cache miss), pozrie sa do L2 cache, potom do L3, až nakoniec, ak sa údaj nenachádza ani v L3, načíta ho z RAM. Pri načítaní sa zároveň údaje prekopírujú do nižších úrovní cache (napr. z RAM do L3, z L3 do L2, z L2 do L1), tak, aby opakovaný prístup k nim už bol rýchly. A tu sa dostávame k podstatnej časti: nebolo by efektívne, keby sme pri načítaní skopírovali len 1 bajt, alebo 1 int (4 bajty). Pri každom prístupe do pamäte sa prenáša celý *blok pamäte*, tzv. cache line, typicky veľkosti 64 bajtov[2]. To má veľmi citeľné praktické dôsledky!

Vyskúšajte si nasledujúci jednoduchý experiment: Zoberte si maticu $N \times N$ (napríklad typu int) a zmerajte, koľko trvá, ak budete jej prvky čítať riadok po riadku; potom to porovnajte s prípadom, keď maticu čítate stĺpec po stĺpci;

[2]závisí od architektúry, najnovšie počítače už majú aj 128 bajtové cache line

nakoniec to porovnajte s prípadom, keď prečítate všetky prvky matice, ale v *náhodnom* poradí. V štandardnom teoretickom modeli, kde predpokladáme, že každá základná operácia procesora trvá konštantný čas, má každý algoritmus zložitosť N^2, avšak v praxi bude medzi týmito troma prechodmi obrovský rozdiel.

Pri čítaní po riadkoch využívame cache najefektívnejšie. Do jednej cache line sa zmestí 16 intov (32-bitových) a vždy po tom, ako prečítame jeden, prečítame aj ďalších 15 nasledujúcich, čo je takmer zadarmo. Naopak, ak čítame v náhodnom poradí, pri dostatočne veľkej matici, ktorá je mnohonásobne väčšia ako L3, bude takmer každý prístup do pamäte L3 cache miss a program pobeží rádovo sto-krát dlhšie.

Pri čítaní po stĺpcoch zakaždým prečítame 4 bajty zo 64 a prejdeme na ďalší riadok... Tu bude čas závisieť od uloženia v pamäti: máme maticu v jednom veľkom bloku pamäte, alebo ju máme uloženú ako pole riadkov, kde každý jeden riadok je samostatne naalokovaný blok pamäte? Napríklad v C++, máme statické pole `int A[N][N]`, alebo `vector<vector<int>> A`? V tom prvom prípade bude aj čítanie po stĺpcoch rýchle vďaka tomu, že procesor je v tomto prípade schopný predpovedať adresu nasledujúceho prístupu (políčko v rovnakom stĺpci ale o 1 nižšom riadku je o $4 \times N$ bajtov ďalej). Ďalší trik moderných procesorov je tzv. *prefetching* – ak vedia dopredu predpovedať, ktorú adresu v pamäti bude treba, môžu už dopredu začať načítavať. Naopak, v druhom prípade, môže byť každý riadok úplne inde a ťažké predpovedať ďalší prístup do pamäte.

Dodajme ešte, že keď už je cache plná, pri načítaní nových dát musíme vždy niečo vyhodiť, aby sme uvoľnili miesto pre nové dáta. Ktoré údaje sa vyhodia a vplyv na chovanie nášho programu sa ťažko predpovedá – treba merať.

Ale vráťme sa k transpozícií matíc. Šok #2 by sme už po tomto úvoďme mali vedieť ľahko vysvetliť: Čas deleno N^2 *je* konštanta, avšak zatiaľčo pre veľmi malé $N < 256$ sa celá matica zmestí do L1 cache, pre $N > 1500$ sa už nezmestí ani do L2 a pre $N > 2048$ ani do L3 cache. S rastúcim N pribúdajú cache missy a to, že čas na prvok matice stúpne „iba“ na $\approx$ 15-násobok (z 0.4ns na 6ns) je dôkaz, že cachovanie ešte stále trochu funguje. Ak by sme predsalen chceli použiť idealizovaný teoretický model, kde každá operácia má jednotkovú cenu, museli by sme do tej ceny započítať, že v najhoršom prípade pristupujeme až do RAM, čo sú desiatky nanosekúnd. (O prípade, že sa dáta nezmestia ani do RAM a pagingu niekedy inokedy.)

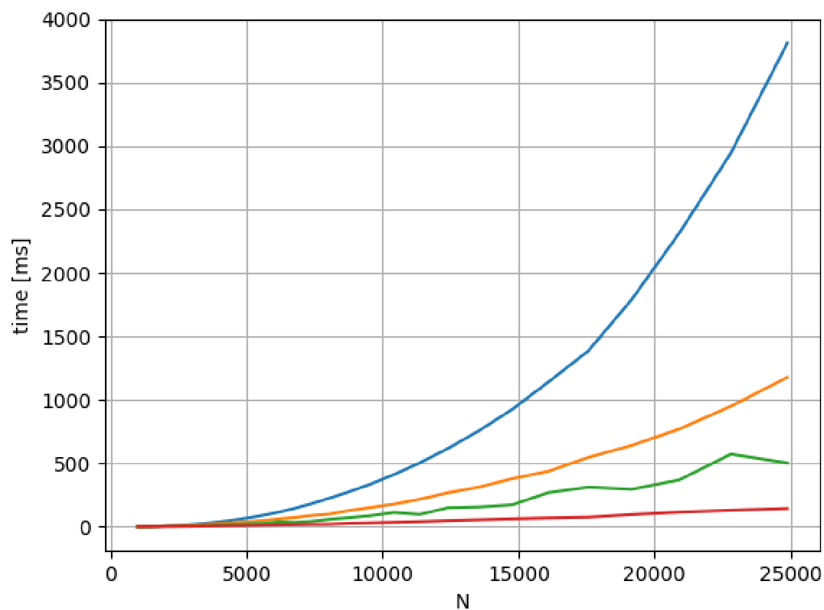
Skúsme teraz porozmýšľať, ako algoritmus na transpozíciu matice zlepšiť. Pamätajte, že je dané, ktoré dvojice políčok treba vymeniť, ale nie je dané, v akom poradí V skutočnosti je poradie prístupov do pamäte to jediné, čo budeme ďalej meniť. Pripomeňme, že v pôvodnom algoritme postupne prechádzame všetky dvojice (i, j), pričom vymieňame prvky $A[i][j] \leftrightarrow A[j][i]$. Políčka (i, j) síce prechádzame po riadkoch, čo je dobre, ale tým pádom pozície (j, i) prechádzame po stĺpcoch, čo je zle, lebo pri tom využijeme iba 4/64 cache line.

Riešenie 2: Rozdeľme maticu na bloky veľkosti $B \times B$. Postupne prechádzame blok po bloku po riadkoch a každý blok $A[i..i + B - 1][j..j + B - 1]$ vymeníme

s (transponovaným) blokom $A[j..j+B-1][i..i+B-1]$.

```
void transpose_block(vector<vector<int>> &A) {
  int N = A.size();
  int B = 64;
  for (int k = 0; k < N; k += B) {
    // transponuj blok [k..k+B][k..k+B] na uhlopriecke
    for (int i = k; i < k + B && i < N; ++i)
      for (int j = i + 1; j < k + B && j < N; ++j)
        swap(A[i][j], A[j][i]);
    for (int l = k + B; l < N; l += B)
      // transponuj blok [k..k+B][l..l+B] <-> [l..l+B][k..k+B]
      for (int i = k; i < k + B && i < N; ++i)
        for (int j = l; j < l + B && j < N; ++j)
          swap(A[i][j], A[j][i]);
  }
}
```

Prvé tri vnorené cykly riešia bloky na uhlopriečke (treba ošetriť zvlášť), nasledujúce cykly vymieňajú bloky so ľavými hornými rohmi $(k,l) \leftrightarrow (l,k)$. Najlepšiu hodnotu B som zvolil skusmo.



A teraz pozor... chvíľka napätia... tento algoritmus sa umiestnil až na treťom mieste! Oranžová čiara na grafe vyššie.

To znamená, že sa to dá ešte lepšie

Čo sa dá ešte zlepšiť? Nuž, zastavme sa pri otázke, akú veľkosť bloku B zvoliť. Je možné, že to záleží aj od toho, na ktorú úroveň cache sa pýtame – chceme minimalizovať počet L1 cache missov? alebo L2? alebo L3?

Riešenie 3: Skúsme použiť tú istú myšlienku ešte raz Maticu najskôr rozdelíme na veľké bloky veľkosti $B \times B$. Maticu budeme prechádzať postupne po veľkých blokoch. Keď však budeme vymieňať $A[x..x+B-1][y..y+B-1] \leftrightarrow A[y..y+B-1][x..x+B-1]$, spravíme to tak, že ich najskôr prerozdelíme na menšie bloky $b \times b$, a tie budeme postupne vymieňať.

Treba trochu zaťať zuby a voilà:

```
void transpose_block2(vector<vector<int>> &A) {
  int N = A.size();
  int B = 1040;
  int b = 4;
  for (int x = 0; x < N; x += B) {
    for (int k = x; k < x + B && k < N; k += b) {
      for (int i = k; i < k + b && i < N; ++i)
        // transponuj blok [k..k+B][k..k+B] na uhlopriecke
        for (int j = i + 1; j < k + b && j < N; ++j)
          swap(A[i][j], A[j][i]);
      for (int l = k + b; l < x + B && l<N; l += b)
        // velky blok na uhlopriecke, maly blok mimo
        for (int i = k; i < k + b && i < N; ++i)
          for (int j = l; j < l + b && j < N; ++j)
            swap(A[i][j], A[j][i]);
    }
    for (int y = x + B; y < N; y += B)
      for (int k = x; k < x + B && k<N; k += b)
        for (int l = y; l < y + B && l < N; l += b)
          // transponuj blok [k..k+B][l..l+B] <-> [l..l+B][k..k+B]
          for (int i = k; i < k + b && i < N; ++i)
            for (int j = l; j < l + b && j < N; ++j)
              swap(A[i][j], A[j][i]);
  }
}
```

Krása. 6 vnorených for-cyklov.

A čo sme dosiahli? Druhé miesto. Zelená čiara na grafe vyššie.

Až sa začínam báť, čo bude nasledovať. Ako povedal klasik: *„Do kedy my takto chceme? Dokéédýýý?“*

Riešenie 4: Nebojte sa, nejdem písať troj-úrovňové riešenie (veľké, stredné, malé bloky) s ôsmimi vnorenými for-cyklami. Kdeže. Keď už, tak ideme all in. Myšlienku delenia na bloky použijeme rekurzívne!

Ako transponujeme $N \times N$ maticu A? Rozdelíme ju na štyri podmatice $\frac{N}{2} \times \frac{N}{2}$:

$$A = \begin{pmatrix} B & C \\ D & E \end{pmatrix} \qquad \text{a transponujeme:} \qquad A^T = \begin{pmatrix} B^T & D^T \\ C^T & E^T \end{pmatrix}.$$

Všimnite si, že podmatice B a E na diagonále sa iba transponujú, C a D sa transponujú a navzájom vymenia. Ako? Rekurzívne: opäť sa rozdelia na štyri podmatice a tie sa vymieňajú a transponujú. Takto pokračujeme v rekurzívnom delení, kým sa nedostaneme na dostatočne malé matičky, ktoré vymeníme a transponujeme klasicky dvoma for-cyklami.

```
void transpose_rec(const int N, vector<vector<int>> &A,
                   int B, int i0, int j0) {
  if (B <= 4) {
    if (i0 == j0) {
      for (int i = i0; i < i0 + B && i < N; ++i)
        for (int j = i + 1; j < i0 + B && j < N; ++j)
          swap(A[i][j], A[j][i]);
    } else {
      for (int i = i0; i < i0 + B && i < N; ++i)
        for (int j = j0; j < j0 + B && j < N; ++j)
          swap(A[i][j], A[j][i]);
    }
  } else {
    int h = B / 2;
    transpose_rec(N, A, h, i0, j0);
    transpose_rec(N, A, B - h, i0 + h, j0);
    if (i0 != j0) transpose_rec(N, A, B - h, i0, j0 + h);
    transpose_rec(N, A, B - h, i0 + h, j0 + h);
  }
}

transpose_rec(N, A, N, 0, 0, 0, 0);
```

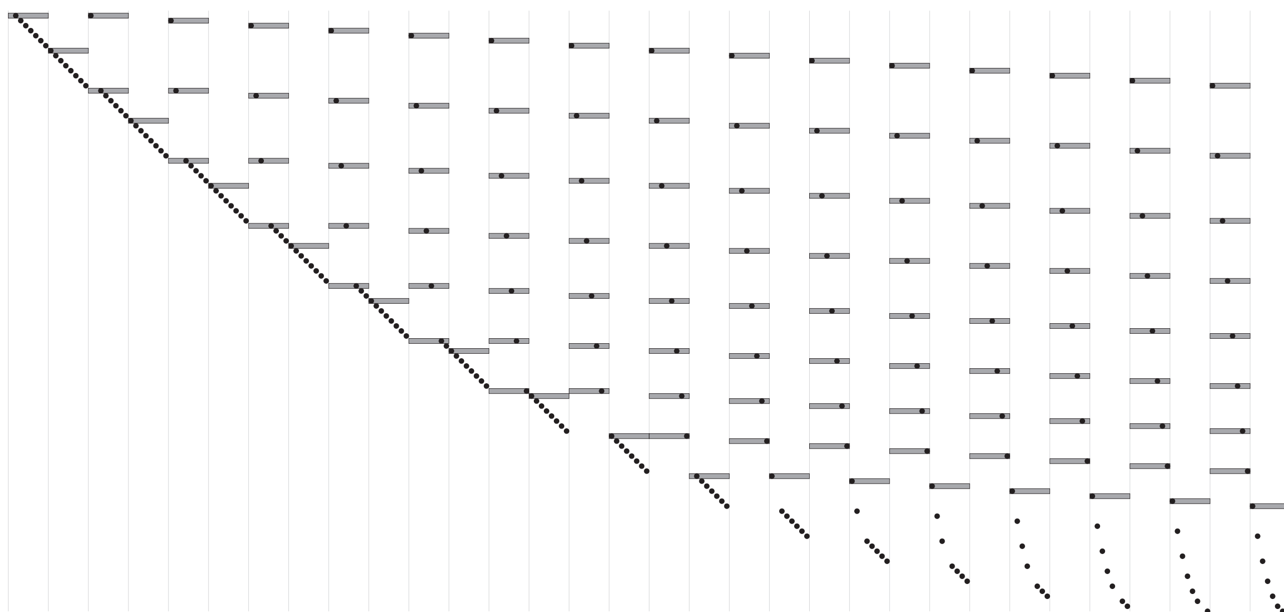
Dámy a páni, toto je naše víťazné riešenie, červená čiara na grafe vyššie.

Referencie

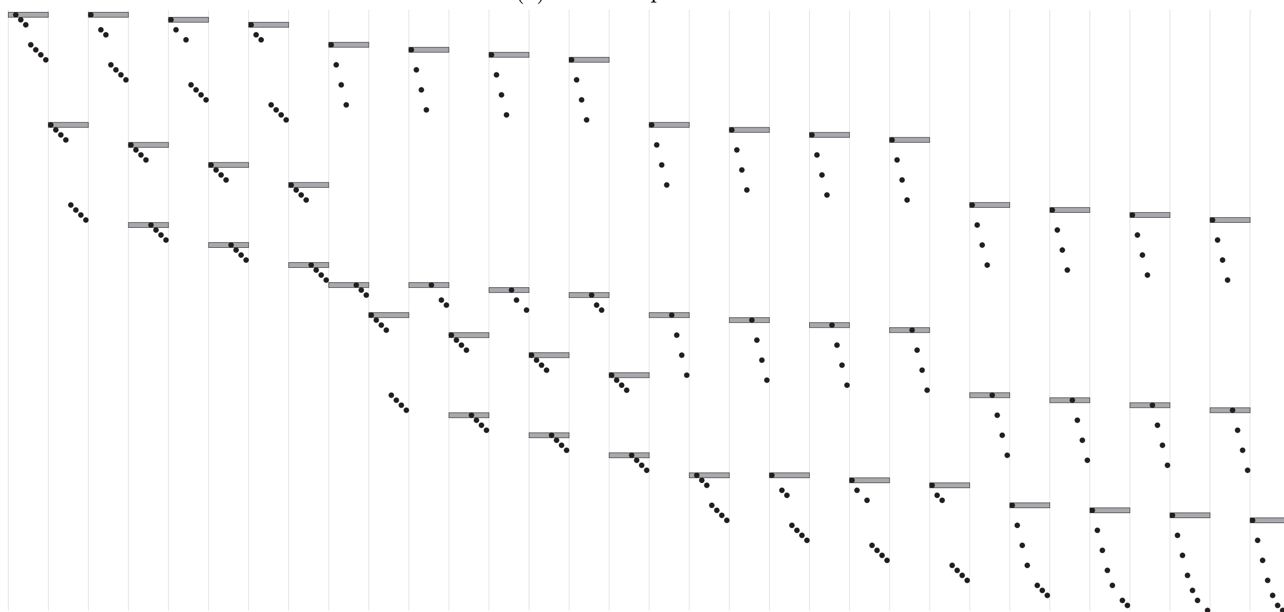
Tarjan, R.E. a J. Van Leeuwen (1984). „Worst-case analysis of set union algorithms“. In: *Journal of the ACM (JACM)* 31.2, s. 245–281.

Tarjan, Robert Endre (1975). „Efficiency of a Good But Not Linear Set Union Algorithm“. In: *J. ACM* 22.2, s. 215–225.

— (1979). „A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets“. In: *J. Comput. Syst. Sci.* 18.2, s. 110–127.

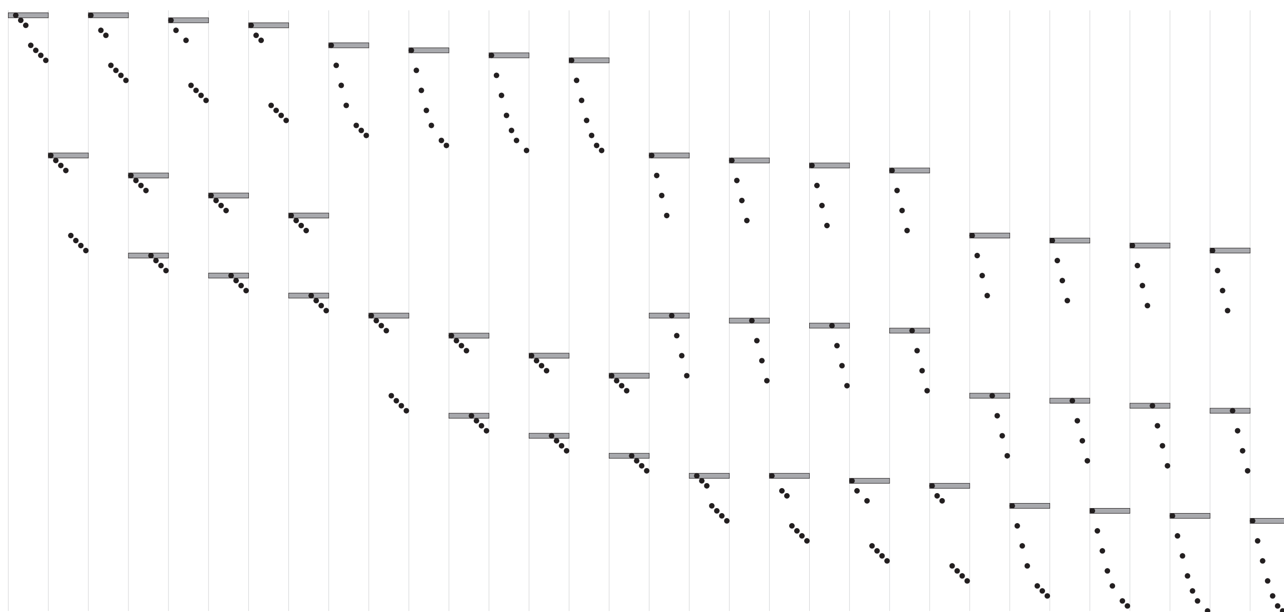


(a) Prechod po riadkoch.

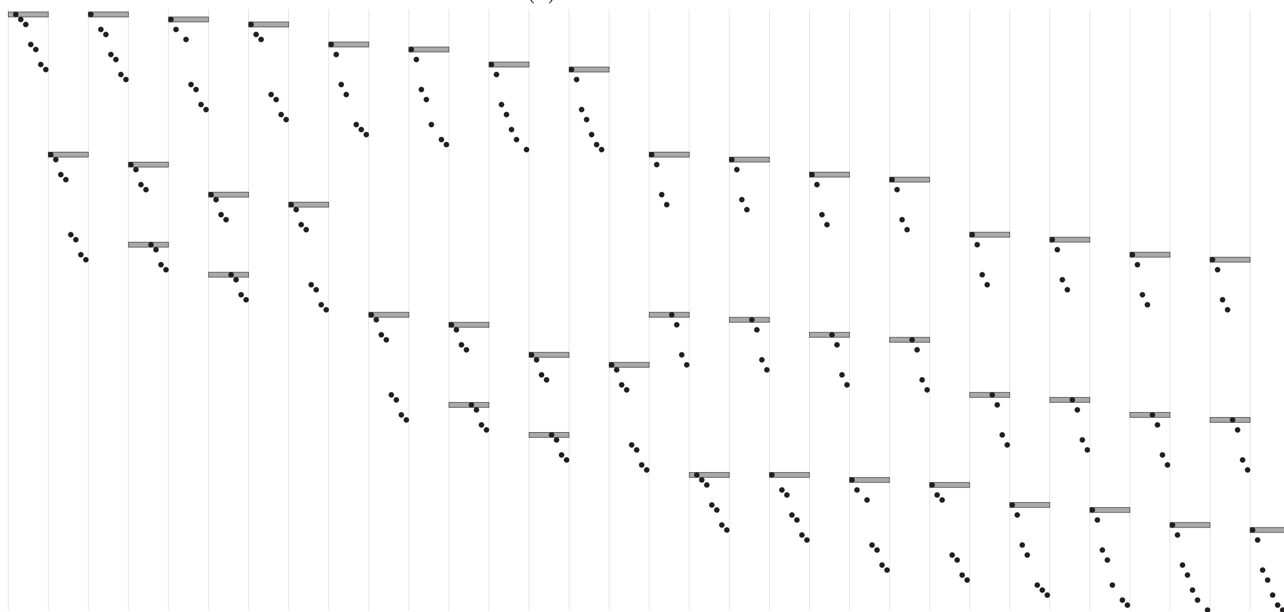


(b) Prechod po blokoch

Obr. 2: Transponovanie matice 16×16: a) po riadkoch a b) po blokoch 4×4. Os x zľava doprava predstavuje pamäť, os y zhora nadol predstavuje čas. Na obrázku je zobrazený priebeh algoritmu: každá čierna bodka znamená prístup do pamäte, šedý obdĺžnik znázorňuje cache miss a načítanie cache line. V tomto hračkárskom príklade máme cache line dĺžky 8 prvkov a cache má kapacitu 8 cache line. Jednotlivé riešenia spôsobia postupne 115 a 50 cache missov.



(a) Dve úrovne blokov.



(b) Rekurzívne riešenie.

Obr. 3: [pokračovanie] Transponovanie matice 16×16: c) dve úrovne blokov 4×4 a 8×8, d) rekurzívne. Tiero riešenia spôsobia postupne 46, respektíve 44 cache missov.