Katedra Informatiky
Fakulta Matematiky, Fyziky a Informatiky
Univerzity Komenského, Bratislava

Triediace algoritmy

(Bakalárska práca)

Juraj Zemianek

Odbor: Informatika 9.2.1
Vedúci: doc. RNDr. Juraj Procházka, CSc. Bratislava, 2007

Čestne prehlasujem, že som túto bakalársku prácu
vypracoval samostatne len s použitím uvedenej literatúry.

_____

Ďakujem svojmu vedúcemu bakalárskej práce, doc. RNDr. Jurajovi Procházkovi, CSc. za cenné rady a pripomienky pri písaní tejto práce.

Obsah

Úvod

Jedným zo základných problémov, ktorými sa informatika zaoberá, je utriedenie zoznamu prvkov. Postupnosť krokov, ktorá tento problém rieši, je triediaci algoritmus. Triediaci algoritmus je teda algoritmus, ktorého vstupom je zoznam prvkov a výstupom je ten istý zoznam, ale utriedený podľa hodnoty prvkov. Triedenie má široké použitie v praxi a je súčasťou mnohých algoritmov.
V mojej práci sa zaoberám najznámejšími triediacimi algoritmami. V prvej kapitole sa nachádza rozdelenie algoritmov a popis každého z nich, v druhej kapitole sa nachádza hlavný cieľ tejto práce, ktorým je porovnať triediace algoritmy z hľadiska výpočtovej zložitosti. Druhá kapitola je rozdelená do troch častí. V prvej časti sa zaoberám mnohonásobným triedením zoznamov malého rozsahu, v druhej časti sa zaoberám porovnaním algoritmov Countingsort a Quicksort, v tretej časti sa zaoberám porovnaním Quicksortu a triedením, ktoré vznikne kombináciou Quicksortu a Insertsortu.

Kapitola 1

Triediace algoritmy

1.1 Prehľad triediacich algoritmov

Pri triediacich algoritmoch si všímame ich vlastnosti a na základe nich ich porovnávame. Najdôležitejšou vlastnosťou je, koľko času na usporiadanie prvkov spotrebujú. Samotné algoritmy sa delia podľa relatívneho času, tzv. *O*- notáciou, čo je asymptotické ohraničenie zhora. Zadefinujeme *O*- notáciu:

Pre danú funkciu $g(n)$ označujeme $O(g(n))$ množinu funkcií

$O(g(n)) = \{ f(n)$: existujú kladné konštanty c a n_0 také, že $0 \leq f(n) \leq c\, g(n)$ pre všetky $n \geq n_0\}$

Skutočnosť, že funkcia $f(n)$ je prvkom množiny $O(g(n))$ označujeme $f(n) = O(g(n))$.

Algoritmy rozdeľujeme na tri skupiny: algoritmy pracujúce v čase $O(n^2)$, $O(n \log n)$ a $O(n)$.

Najznámejšie algoritmy prvej skupiny, $O(n^2)$, sú Bubblesort, Insertsort a Selectsort.

Algoritmy tejto skupiny majú svoje spoločné vlastnosti. Všetko ide o časovo pomalšie algoritmy. Majú jednoduchú myšlienku, sú pomerne ľahko naprogramovateľné. Nevyžadujú pomocné dátové štruktúry a pomocnú pamäť. Sú vhodné najmä pre dáta a zoznamy s menším počtom prvkov. Pri väčších zoznamoch majúcich nad 1000 prvkov už nie sú vhodné, pretože čas rastie rádovo kvadraticky. Pre takéto zoznamy je vhodné použiť algoritmy triediace v čase $O(n \log n)$.

Najznámejšie algoritmy druhej skupiny, $O(n \log n)$, sú Quicksort, Mergesort a Heapsort.

Aj táto skupina má svoje špecifické vlastnosti. Všetko ide o časovo rýchlejšie algoritmy. Myšlienka je u väčšiny algoritmov zložitejšia, pretože mnohé vyžadujú rekurziu alebo znalosť pokročilých dátových štruktúr. Sú naprogramovateľné ťažšie, než v prípade predošlej skupiny. Pracujú veľmi rýchlo a odporúčajú na triedenia veľkých zoznamov.

Najznámejšie algoritmy tretej skupiny, $O(n)$, sú Countingsort, Radixsort a Bucketsort.

Pre algoritmy tejto skupiny platí, že ich môžeme uplatniť len vtedy, ak niečo vieme o tom, ako vyzerajú vstupné údaje. Napríklad Countingsort počíta výskyty každej hodnoty a potom tieto hodnoty zoradí. To predpokladá mať na vstupe celé čísla nízkeho rozsahu. Myšlienka zvykne byť jednoduchá. Často sa požaduje prídavná pamäť. Pri správnom použití týchto algoritmov dostaneme mnohonásobne rýchlejšie výsledky, než v prípade druhej skupiny.

1.2 Algoritmy triediace v čase $O(n^2)$

1.2.1 Bubblesort

Popis:

Bubblesort pracuje veľmi jednoducho. Porovnáva každý prvok zoznamu s nasledujúcim a ak je to potrebné, navzájom ich vymení. Tento proces pokračuje dovtedy, dokiaľ po prejdení poľa bol vymenený aspoň jeden prvok. Ak už nebolo vymenené nič, algoritmus končí.

Možné prístupy naprogramovania:

Ľavá časť poľa je neutriedená, pravá utriedená. Postupne prebiehame neutriedenou časťou poľa od začiatku do konca. Ak sa stane, že prvok väčšej hodnoty sa nachádza pred prvkom nižšej hodnoty, vymeníme tieto prvky. Takto sa ľavá neutriedená časť každým krokom zmenší o jeden prvok. Máme istotu, že vždy najväčší prvok v neutriedenej časti sa dostane na jej koniec.
Opakujúco prechádzame poľom vždy od začiatku do konca a všímame si počet výmen, ktoré sme museli urobiť. Robíme tak dovtedy, dokiaľ je počet výmen nenulový.

Časový odhad:

V najhoršom prípade potrebuje algoritmus v každom kroku n-1 výmen, počet prvkov je n, z čoho plynie zložitosť $O(n^2)$. V najlepšom prípade, kedy je pole utriedené, prejde poľom len raz a končí, z čoho plynie zložitosť $O(n)$.

Porovnanie:

Bubblesort je najpomalší algoritmus prvej skupiny, pretože potrebuje veľký počet výmen, v najhoršom prípade sa počet výmen rovná počtu prechodov po poli (ak je pôvodné pole usporiadané naopak). Ostatné algoritmy, ako Insertsort, Selectsort, sú rýchlejšie, pretože nevyžadujú taký veľký počet výmen. V praxi sa Bubblesort neodporúča.

Pseudokód:

```
zmena ← false;
while (zmena = true) do
begin
     zmena ← false;
     for i ← 1 to length(A) do
     if (A[i] > A[i+1] ) then
     begin
          vymen( A[i], A[i+1] );
          zmena ← true;
     end;
end;
```

1.2.2 Insertsort

Popis:

Triedenie vkladaním je jednoduchý algoritmus. Vezme prvok a vloží ho podľa jeho vlastnosti (hodnoty, abecedného poradia...) do výstupnej postupnosti. Pokračuje dovtedy, dokiaľ nie je každý prvok utriedený.

Možné prístupy naprogramovania:

Prvky vyberáme sekvenčne zo vstupného poľa a vkladáme ich do výstupného poľa. Výstupné pole udržujeme v každom kroku utriedené, po skončení algoritmu je výsledkom utriedená postupnosť.
Pre ušetrenie pamäti sa používa iný prístup. Vstupné pole a pomyslene rozdelí na dve časti – začiatočnú a konečnú. Prvky vyberáme sekvenčne a vkladáme ich do utriedenej postupnosti. V každom kroku sa začiatočná časť zväčší a konečná zmenší o jeden prvok.

Časový odhad:

V najhoršom prípade potrebuje algoritmus $n - 1$ výmen, počet prvkov je n, zložitosť je teda $O(n^2)$. Ak je pole utriedené, ide o najlepší prípad, lebo nepotrebuje žiadnu výmenu a zložitosť je $O(n)$.

Porovnanie:

V porovnaní s algoritmami prvej skupiny dáva Insertsort dobré výsledky. Je približne dvakrát rýchlejší, ako Bubblesort a o 40% rýchlejší, ako Selectsort ([5]). Je vhodný pre zoznamy, ktoré majú veľkosť do 1000 prvkov a pre zoznamy, ktoré sú takmer usporiadané.

Pseudokód:

```
for i ← 2 to length(A) do
begin
     pomocna ← A[i];
     j ← i;
     while (j > 1) and (A[j-1] >= pomocna) do
     begin
          A[j] ← A[j-1];
          j ← j-1;
     end;
     A[j] ← pomocna;
end;
```

1.2.3 Selectsort

Popis:

Selectsort vyberá vždy neutriedený prvok najnižšej hodnoty a vloží ho na koniec utriedeného zoznamu. Algoritmus končí vtedy, keď sú všetky prvky usporiadané.

Možné prístupy naprogramovania:

Máme dané vstupné pole a výstupné. V každom kroku zistíme hodnotu minimálneho neoznačeného prvku, označíme ho a vložíme na koniec výstupného poľa.
Kvôli šetreniu pamäti sa používa iný prístup. Podobne ako pri Insertsorte, pomyslene rozdelíme pole na utriedenú (je na začiatku) a neutriedenú časť. Z neutriedenej časti nájdeme minimálny prvok a vymeníme ho s prvkom, ktorý je na konci utriedenej časti. Takto sa v každom kroku utriedená časť zväčší a neutriedená zmenší.

Časový odhad:

V najhoršom prípade beží algoritmus v čase $O(n^2)$, v najlepšom prípade – utriedenej postupnosti, beží v lineárnom čase $O(n)$.

Porovnanie:

Selectsort je približne o 60% rýchlejší, ako Bubblesort ([5]), ale pomalší ako Insertsort. Je výhodné ho použiť, ak je operácia výmeny drahá, pretože si len pamätá index najmenšieho prvku a v každom kroku tak nastane nanajvýš jedna výmena.

Pseudokód:

```
for i ← 1 to length(A) do
begin
     min ← i;
     for j ← i+1 to length(A) do
     if (A[j] < A[min]) then min ← j;
     vymen(A[i],A[min]);
end;
```

1.3 Algoritmy triediace v čase $O(n \log n)$

1.3.1 Quicksort

Popis:

Quicksort je v priemernom prípade najrýchlejší algoritmus druhej skupiny. Čo sa myšlienky týka, je jednoduchý, ale náročný na naprogramovanie. Využíva metódu „Rozdeľuj a panuj“. Ide o rekurzívny algoritmus. Rekurzia pozostáva zo štyroch krokov:

1. Ak je počet prvkov v poli najviac jedna, je pole utriedené a algoritmus nerobí nič.
2. Vyberie prvok v poli. Tento prvok sa nazýva pivot.
3. Rozdelí pole do dvoch častí. V prvej časti sú prvky menšie alebo rovné, ako pivot, v druhej časti sú prvky väčšie ako pivot.
4. Rekurzívne sa zavolá pre obidve časti.

Možné prístupy naprogramovania:

Pretože rozdeľujeme pole na menšie časti, je potrebné pamätať si index pivota, index prvého a index posledného prvku. V časti rozdeľovania poľa postupujeme tak, že prvky, ktoré sú v nesprávnej časti, medzi sebou vymieňame. Pri rekurzívnom volaní upravíme hranice pre začiatok a koniec podpoľa. Rozdiely v algoritme môžu byť jedine v súvislosti s výberom pivota. V ideálnom prípade sa pole rozdelí na dve rovnako dlhé polovice. V priemernom prípade to tak nie je. Ako pivot sa obyčajne vyberie najľavejší alebo najpravejší prvok. Takýto prístup je však nevhodný, ak je vstupné pole utriedené. Z tohto dôvodu sa v mnohých implementáciach vyberá stredný prvok, alebo náhodný prvok z intervalu od začiatku do konca.

Časový odhad:

V najhoršom prípade pracuje Quicksort v čase $O(n^2)$, v priemernom aj najlepšom prípade pracuje v čase $O(n^2)$.

Porovnanie:

Quicksort dáva v priemernom prípade z hľadiska času najlepšie výsledky. Je rýchlejší ako Heapsort, aj ako Mergesort. Pre množinu všeobecných prvkov je to najrýchlejší algoritmus vôbec.

Pseudokód:

```
procedure qsort(zaciatok,koniec:integer);
begin
     // vyber pivota – mame tri moznosti...

     //p:=A[z];
     //p:=random(zaciatok..koniec);
     p:=A[(zaciatok+koniec) div 2];

     i:=zaciatok-1;
     j:=koniec+1;

     opakuj
          opakuj i:=i+1; pokiaľ A[i]>=p;
          opakuj j:=j-1; pokiaľ A[j]<=p;
          ak i<j potom vymen(A[i],A[j]);
     pokiaľ i>=j;

     ak i=j potom
     begin
          i:=i+1;
          j:=j-1;
     end;

     ak zaciatok<=j potom qsort(zaciatok,j);
     ak koniec>=i potom qsort(i,koniec);
end;
```

1.3.2 Mergesort

Popis:

Mergesort je ďalší algoritmus triedenia, ktorý triedi v čase $O(n \log n)$ a využíva metódu „Rozdeľuj a panuj“. Myšlienkou je rekurzívne deliť pole na dve podpolia rovnakých veľkostí a pri návrate z rekurzie zlučovať utriedené podpolia do väčších utriedených podpolí. Rekurzia sa skladá z týchto častí:

1. Rozdelíme pole na dve podpolia (približne) rovnakej veľkosti
2. Ak je počet prvkov podpoľa väčší ako 1, zavoláme procedúru na toto podpole
3. Zlúčime podpolia, ktoré sú utriedené, do väčšieho podpoľa, ktoré je utriedené

Možné prístupy naprogramovania:

Mergesort má rôzne modifikácie. Spoločné pre ne je, že majú dve časti – v prvej sa rozdeľuje pole na dve časti (*Mergesort*) a v druhej zlučuje utriedené polia (*Merge*). Rozdielnosť sa prejavuje v procedúre *Merge*, kde máme na vstupe dve utriedené polia a máme z nich dostať jedno utriedené pole.

Časový odhad:

V najlepšom, priemernom i najhoršom prípade má mergesort zložitosť $n \log n$. Rekurzia sa dá napísať ako $T(n) = 2T(n/2) + n$, kde $T(n/2)$ je rekurzívne volanie na obe podpolia a n je časová zložitosť zlúčenia podpolí. Podľa Master Theorem ([2]) zistíme, že takáto rekurzia beží v čase $O(n \log n)$.

Porovnanie:

Čo sa týka času, v priemernom prípade je mergesort pomalší, ako quicksort, ale rýchlejší, ako heapsort. Čo sa týka pamäťovej zložitosti, mergesort požaduje prídavnú pamäť, čo je jeho nevýhoda v porovnaní s heapsortom, ktorý prídavnú pamäť nepotrebuje. Mergesort však dáva dobré výsledky, a je odporúčaný najmä pre spájané zoznamy, pretože tam stačí konštantná prídavná pamäť.

Pseudokód:

```
procedure Merge_sort(zaciatok, koniec);

begin
     if (koniec-zaciatok > 0) then
     begin
          p ← ⌊(zaciatok + koniec)/2⌋;

          Merge_sort(zaciatok, p);
          Merge_sort(p+1, koniec);

          for i ← p downto zaciatok do B[i] ← A[i];
          for j ← p+1 to koniec do B[koniec+p+1-j] ← A[j];
          for k ← zaciatok to koniec do
          begin
```

```
                if (B[i] < B[j]) then
                begin
                     A[k] ← B[i];
                     i ← i+1;
                end
                else
                begin
                     A[k] ← B[j];
                     j ← j-1;
                end;
           end;
     end;
end;
```

1.3.3 Heapsort

Popis:

Heapsort je algoritmus, ktorý triedi prvky poľa pomocou binárnej haldy. Binárna halda je pole, na ktoré sa je možné pozerať ako na úplný binárny strom. Prvý prvok poľa je koreň tohto stromu, ako jediný nemá rodiča. Každý iný vrchol rodiča má. Pre všetky vrcholy platí, že majú buď dvoch potomkov (pravého a ľavého), jedného, alebo žiadneho potomka. Vrcholy bez potomkov sa nazývajú listy. Ak má vrchol index *i*, jeho rodič má index $\lfloor i/2 \rfloor$, jeho ľavý syn má index 2**i* a pravý syn má index 2**i* + 1. Dôležité je, že halda musí spĺňať vlastnosť haldy, teda že pre každý vrchol *i* rôzny od koreňa platí, že hodnota vrchola jeho rodiča je väčšia alebo rovná, ako hodnota vo vrchole *i*.

Triedenie heapsort pracuje s touto haldou. Najprv ju vybuduje a potom pravidelne vymieňa koreň a posledný prvok poľa, pričom udržuje stále vlastnosť haldy. Realizuje sa to pomocou dvoch procedúr.

Prvá sa nazýva *heapify*. Táto procedúra predpokladá, že pravý i ľavý podstrom koreňa má vlastnosť haldy, no v koreni je hodnota, ktorá je menšia ako hodnota v pravom alebo ľavom synovi. To porušuje vlastnosť haldy. Úlohou procedúry *heapify* je premiestniť prvok z koreňa na také miesto, aby spĺňal vlastnosť haldy. To sa dosiahne opakovanou vzájomnou výmenou hodnôt aktuálneho vrchola a jeho rodiča.
Druhá procedúra je *build-heap*, ktorá urobí z poľa haldu. Prvky v podpoli s indexami od $\lfloor n/2 \rfloor + 1$ až *n* už tvoria jednoprvkovú haldu. Na zvyšné prvky sa v poradí od najväčšieho indexu po najmenší zavolá procedúra *heapify*. Poradie, v ktorom procedúra prechádza vrcholy, zabezpečuje, aby podstromy s koreňmi v synoch vrchola boli haldy predtým, ako sa na tomto vrchole začne vykonávať procedúra *heapify*.

Po vybudovaní haldy sa opakuje nasledujúci krok: Pamätá sa veľkosť haldy. Posledný prvok poľa, ktorý je na indexe rovnajúcom sa veľkosti haldy, sa vymení s prvým prvkom poľa, teda koreňom. Potom sa zavolá procedúra *heapify* na koreň haldy. Takto sa prvok z koreňa dostane na iné miesto haldy tak, aby bola vlastnosť haldy udržaná. Nakoniec sa veľkosť hlady sa zníži o jednu.
Tento krok sa opakuje dovtedy, dokiaľ nie je veľkosť haldy rovná jednej. Takto sa stane to, že vo výslednom poli máme utriedenú postupnosť. Samozrejme, procedúru *heapify* musíme upraviť tak, aby jej rekurzívne volanie záviselo od aktuálnej veľkosti hlady, lebo prvky na konci poľa sú už utriedené a nesmie sa už s nimi hýbať.

Možné prístupy naprogramovania:

Procedúry *heapify* a *build-heap* sú jasne špecifikované. Rozdielnosť môže byť však v implementácii haldy. V popise algoritmu je halda reprezentovaná cez pole, (index *i* má rodiča *i*/2 a synov 2**i*, 2**i* + 1), môže byť však reprezentovaná aj cez triedu *Vrchol*, ktorá má premenné *info, left* a *right*, kde *info* je hodnota vrchola a *left*, *right* sú synovia vrchola (tiež sú typu *Vrchol*). Podľa toho, ako reprezentujeme haldu, píšeme algoritmy procedúr *heapify* a *build-heap*.

Časový odhad:

Trvanie procedúry *heapify* je nanajvýš veľkosť hĺbky haldy, teda jej trvanie je $O(\log n)$. Trvanie procedúry *build-heap* je v čase $O(n \log n)$, pretože tu na $n/2$ prvkov zavoláme procedúru *heapify*. Samotné triedenie sa skladá z vybudovania haldy a n násobného volania procedúry *heapify*, teda celkový čas triedenia je $O(n \log n) + n * O(\log n) = O(n \log n)$. Časová zložitosť je v najlepšom, priemernom i najhoršom prípade $O(n \log n)$.

Porovnanie:

Heapsort je v priemernom prípade pomalší, ako quicksort a mergesort, no jeho výhodou je skutočnosť, že potrebuje len konštantnú prídavnú pamäť. Takže je ho výhodné použiť, ak chceme triediť rýchlo pri minimálnej spotrebe pamäti.

Pseudokód:

```
procedure Heapify(A,i);
begin
     l ← Left(i)
     r ← Right(i)
     if (l ≤ heap-size[A]) and (A[l] > A[i])
     then max ← l else max ← i
     if (r ≤ heap-size[A]) and (A[r] > A[max])
     then max ← r
     if max ≠ i then vymena(A[i], A[max])
     Heapify(A, max)
end;

procedure Build-Heap(A)
begin
      heap-size[A] ← length[A]
      for i:= ⌊length[A]/2⌋ downto 1 do
      Heapify(A, i)
end;

procedure Heapsort(A);
begin
      Build-Heap(A);
      for i:= length(A) downto 2 do
      vymena(A[1], A[i])
      heap-size[A] = heap-size[A] – 1
      Heapify(A, 1)
end;
```

1.4 Algoritmy triediace v čase $O(n)$

1.4.1 Countingsort

Popis:

Všetky algoritmy v čase $O(n)$ predpokladajú, že vstupné prvky majú nejakú vlastnosť. Countingsort predpokladá, že každý z n vstupných prvkov je celé číslo z intervalu $<1; k>$ pre nejakú celočíselnú konštantu k.
Hlavnou myšlienkou algoritmu je pre každý vstupný prvok a určiť počet prvkov, ktoré sú menšie, alebo rovné, ako prvok a. Táto informácia je potom použitá pre priame umiestnenie prvku vo výstupnom poli. Počet prvkov jednej a tej istej hodnoty môže byť viac, preto musí algoritmus vyriešiť aj to, aby sa tieto prvky nedostali vo výstupe na to isté miesto. Countingsort je stabilné triedenie, to znamená, že prvky rovnakej hodnoty na vstupe budú vo výstupe v rovnakom poradí.

Možné prístupy naprogramovania:

Predpokladajme, že vstupné pole je $A[1..n]$. K nemu potrebujeme ešte dve ďalšie polia. Pole $C[1..k]$, ktoré najprv počíta výskyt prvku danej hodnoty a potom určí, koľko prvkov je menších alebo rovných, ako prvku danej hodnoty a pole $B[1..n]$, ktoré slúži na utriedený výstup.
Najprv nastavíme všetky prvky poľa C na nulu. Potom prejdeme postupne vstupné pole A. Ak je hodnota prvku vstupného poľa i, zvýšime $C[i]$ o jedna. Takto dosiahneme, že v poli C bude počet výskytov každej z hodnôt intervalu od 1 po k. V ďalšom kroku prejdeme pole C od 2 do k a do každého prvku poľa priradíme súčet $C[i] + C[i+1]$. Takto dosiahneme, že v poli C bude pre každé $i = 1..k$ informácia, koľko prvkov je menších alebo rovných ako i. Nakoniec umiestnime každý prvok z A na jeho správnu pozíciu vo výstupnom poli B. Pokiaľ sú všetky prvky rôzne, potom je pre každé $A[i]$ hodnota $C[A[i]]$ správnou pozíciou $A[i]$ vo výstupnom poli. Pretože všetky prvky nemusia byť navzájom rôzne, vždy, keď umiestnime hodnotu $A[i]$ do poľa B, znížime hodnotu $C[A[i]]$ o jedna. To zabezpečí, aby ďalší vstupný prvok s hodnotou rovnou $A[i]$, pokiaľ existuje, bol umiestnený na pozíciu pred $A[i]$ vo výstupnom poli.

Časový odhad:

Inicializácia poľa C – nastavenie prvkov na 0 a jeden ďalší prechod týmto poľom trvajú $O(k)$, algoritmus prejde navyše dvakrát pole A, to trvá $O(n)$. Takto je celkový beh v čase $O(n + k)$. Predpokladá sa, že rozsah $1..k$ nie je väčší než n, teda možno uvažovať, že $k = O(n)$ a teda je potom celkový čas behu countingsortu $O(n)$.

Porovnanie:

Countingsort beží za ideálnych predpokladov (počet prvkov je podstatne menší než ich rozsah) niekoľkonásobne rýchlejšie ako algoritmy bežiace v čase $O(n \log n)$. Jeho čas behu je porovnateľný s časom behu iných algoritmov triediacich v čase $O(n)$.

Pseudokód:

```
procedure Counting_Sort;
begin
      for i ← 1 to k do C[i] ← 0;
      for j ← 1 to length(A) do C[A[j]] ← C[A[j]] + 1;
      // C[i] teraz obsahuje počet výskytov prvka i v poli A
      for i ← 2 to k do C[i] ← C[i] + C[i – 1];
      // C[i] teraz obsahuje počet prvkov menších alebo rovných ako i
      for j ← length(A) downto 1 do
      begin
      B[C[A[j]]] ← A[j];
      C[A[j]] ← C[A[j]] – 1;
      end;
end;
```

1.4.2 Radixsort

Popis:

Radixsort triedi čísla tak, že ich utriedi podľa ich poslednej číslice, potom podľa predposlednej, takto pokračuje ďalej, až po prvú číslicu. Ak majú čísla na vstupe rôzne dlhý zápis, je potrebné doplniť tieto čísla zľava nulami. Pri triedení podľa jednotlivých cifier musíme použiť algoritmus, ktorý je stabilný. Obyčajne je výhodné použiť Countingsort alebo Bucketsort.

Možné prístupy naprogramovania:

Vo všeobecnosti možno použiť dva prístupy. Prvý z nich je spomenutý v popise, je to LSD – least significant digit. To znamená, že sa ide od najmenej významnej číslice po najvýznamnejšiu, teda sprava doľava. Druhý z nich je MSD – most significant digit. V tomto prípade sa začína triediť od najvýznamnejšej číslice po najmenej významnú, teda zľava doprava. Oba prístupy pracujú analogicky.
V rámci nich je možné použiť dve metódy medzitriedenia prvkov. Buď ich budeme deliť na skupiny podľa hodnoty cifry na aktuálnej pozícii cifry, alebo použijeme na to rady.

Časový odhad:

Záleží od algoritmu, ktorý je použitý ako pomocný. Obyčajne nebývajú číslice veľkého rozsahu a používa sa Countingsort. Ak je počet cifier d a počet prvkov n, potom trvá každý prechod $O(n + k)$. Počet prechodov je d, potom je celkový čas $O(nd + kd)$. Keďže d je konštanta a $k = O(n)$, beží radixsort v lineárnom čase.

Porovnanie:

Radixsortom sa triedia rýchlo čísla, ktoré nemajú veľa cifier, podobne je možné triediť dátumy a slová. Je približne rovnako rýchly, ako iné algoritmy, ktoré triedia v čase $O(n)$. Na rozdiel od nich však musí algoritmus radixsort použiť iné, pomocné triedenie.

Pseudokód:

```
procedure Radix-Sort(A, d);

begin
     for i ← 1 to d do
     utrieď pole A stabilným triedením podľa cifry poradia i
end;
```

1.4.3 Bucketsort

Popis:

Bucketsort je ďalšie z triedení, ktoré beží v priemernom prípade v lineárnom čase. O vstupe predpokladá, že na vstupe sú rovnomerne generované čísla z intervalu <0; 1). Hlavnou myšlienkou je rozdeliť interval <0; 1) na *n* rovnako veľkých disjunktných podintervalov. Tieto podintervaly sa nazývajú buckety. Do bucketov potom rozmiestni *n* vstupných čísel. Nakoniec prejde všetkými bucketami a utriedi čísla v každom z nich.

Možné prístupy naprogramovania:

Buckety sa reprezentujú pomocou spájaných zoznamov. Uvažujeme preto pole *B* [0..*n*-1], ktorého prvkami sú spájané zoznamy. Každý bucket bude zhromažďovať prvky z intervalu veľkosti 0,1, prvý bude obsahovať čísla intervalu <0; 0,1), druhý bude obsahovať čísla z intervalu <0,1; 0,2), a tak ďalej, až desiaty interval bude obsahovať čísla z intervalu <0,9; 1). Predpokladáme, že máme k dispozícii mechanizmus na prácu so spájanými zoznamami. Pomocou neho vytvoríme zoznamy. Potom každý zo zoznamov utriedime pomocou insertsortu. Nakoniec tieto zoznamy, teda *B*[0], *B*[0], … , *B*[9], zreťazíme.

Časový odhad:

Vkladanie do zoznamu prebieha v čase $O(n)$. Pravdepodobnostnými metódami možno ukázať, že čas behu insertsortu použitého v bucketsorte je $O(n)$. Takto je čas celého algoritmu bucketsort $O(n)$.

Porovnanie:

Na rozdiel od Countingsortu očakáva bucketsort na vstupe reálne čísla z istého intervalu. Možno ho však použiť všade, kde sa dajú vstupné údaje rozdeliť na malý počet skupín. Oproti iným algoritmom triediacich v čase $O(n)$ využíva spájané zoznamy. Pri správnom použití je približne rovnako rýchly, ako Countingsort a Radixsort.

Pseudokód:

```
procedure Bucket_Sort(A);
begin
     n ← length[A];
     for i ← 1 to n do vlož A[i] do zoznamu B[⌊nA[i]⌋];
     for i ← 1 to n do utrie+d zoznam B[i] pomocou insertsortu
     zreťaz zoznamy B[0]...B[n-1] v tomto poradí
end;
```

Kapitola 2

Porovnanie algoritmov z hľadiska výpočtovej zložitosti

2.1 Triedenie zoznamov s malým počtom prvkov

2.1.1 Zadanie úlohy

Pri triedení zoznamov s malým počtom prvkov vzniká zaujímavá otázka. Tá otázka znie, že ktoré algoritmy sú rýchlejšie v závislosti od počtu prvkov zoznamu. Z teoretického hľadiska by mali byť rýchlejšie algoritmy zo skupiny $O(n \log n)$, než algoritmy zo skupiny $O(n^2)$. Prakticky je to však inak. Keďže algoritmy skupiny $O(n \log n)$ používajú rekurziu a pokročilé dátové štruktúry, zatiaľ čo algoritmy skupiny $O(n^2)$ sú jednoduché (používajú len porovnávanie), pre zoznamy s malým počtom prvkov sú algoritmy skupiny $O(n^2)$ rýchlejšie. Vynára sa však ďalšia otázka. A síce, ktoré algoritmy sú výhodné a pre aký počet prvkov. Totiž ak sa počet prvkov zväčšuje, časy algoritmov skupiny $O(n \log n)$ sa zlepšujú, až sú od istého počtu prvkov rýchlejšie, ako algoritmy skupiny $O(n^2)$. Mojou úlohou v tejto časti bude preskúmať, aké rýchle sú algoritmy a pre aký počet prvkov.

V našom pokuse nebudeme uvažovať algoritmy bežiace v lineárnom čase. Budeme porovnávať šesť algoritmov, tri zo skupiny $O(n \log n)$ – Quicksort, Mergesort a Heapsort a tri zo skupiny $O(n^2)$ – Bubblesort, Insertsort a Selectsort. Ako prvky zoznamu budeme uvažovať typ longint, teda celé znamienkové 4 bajtové čísla. V skutočnosti pôjde vždy o kladné čísla z intervalu od 1 do 1 000 000. Meranie časov behu algoritmov budeme realizovať pomocou programu.

Ak by sme chceli odmerať čas pre jedno triedenie poľa, pre hocijaký algoritmus by sme dostali nič nehovoriaci čas 00:00.00 (*minúty* : *sekundy* . *stotiny sekundy*). Z tohto dôvodu je potrebné vykonať triedenie viackrát. Aby boli výsledky skutočne viditeľné a údaje porovnateľné, budeme pre daný algoritmus a daný počet prvkov zoznamu triediť zoznam miliónkrát. Našim cieľom je zistiť, ako sa správajú algoritmy pre istý počet prvkov zoznamu. Preto budeme každým z algoritmov triediť pole s počtom prvkom rovnajúcim sa $5*i$, kde i je celé číslo z intervalu od 1 do 16. Inými slovami, počty prvkov zoznamu budú násobky piatich od 5 do 80.

2.1.2 Tvorba programu

Dostávame sa k písaniu programu. Program je napísaný tak, aby sa dali vstupné údaje ľahko meniť. Vstup je daný pomocou konštánt. Konštanta *vstupy* (nastavená na 16) určuje počet triedení pre každý zo šiestich algoritmov, konštanta *max_pocet* (nastavená na 80) určuje maximálny počet prvkov jedného poľa, konštanta *opakovania* (nastavená na 1 000 000) určuje, koľkokrát sa triedenie každým algoritmom a každým počtom prvkov zoznamu vykoná. Konštantné pole *pocet_prvkov* má veľkosť rovnajúcu sa konštante *vstupy* a obsahuje počty prvkov zoznamu, ktoré budú algoritmy triediť. Jeho hodnoty sú nastavené na násobky piatich od 5 do 80. Polia, ktoré sa majú utriediť, sú typu *pole*, typ *pole* je definovaný ako pole s rozsahom prvkov od 0 do *max_pocet* dátového typu longint.

Na začiatku programu sa vytvorí súbor *vystup.out*, do ktorého sa budú zapisovať výsledky meraní. Príkazom *randomize* sa iniciuje generátor náhodných čísel. V ďalších dvoch príkazoch sa iniciujú polia *tried* a *vzor*, potrebné pri meraní časov. Polia, ktoré sa budú triediť, budú v poli *tried*. Toto pole má milión prvkov a každým prvkom je pole, ktoré sa má utriediť. Aby každý z algoritmov triedil vždy rovnakú vzorku polí, na začiatku sa uložia hodnoty do pola *vzor*. Toto pole má tiež milión prvkov a každý prvok je pole, ktoré má byť utriedené. Do pola *tried* sa pred každým triedením priradí pole *vzor*. Nasleduje *for* cyklus. Na jeho začiatku sa do pola *vzor* vygenerujú prvky. Každý prechod cyklu sa líši počtom prvkov polí, ktoré sa majú utriediť. Počet prvkov sa udržuje v premennej *n*. Potom pre každý z algoritmov Quicksort, Mergesort, Heapsort, Bubblesort, Insertsort a Selectsort sa deje nasledovné:

1. Do poľa *tried* sa priradí pole *vzor*
2. Do premennej *start* sa uloží systémový čas pred triedením
3. Mnohonásobne sa triedi pole *tried* príslušným algoritmom. Koľkokrát to je, to záleží od hodnoty konštanty *opakovania*
4. Do premennej *finish* sa uloží systémový čas na konci triedenia
5. Do premennej *rozdiel* sa uloží celkový čas merania v stotinách sekundy
6. Pomocou funkcie *spracuj* vracajúcej reťazec sa upraví čas do formátu *minúty: sekundy. stotiny sekundy* a uloží sa do premennej *vys*
7. Výsledok sa zapíše do tabuľky *tab*, ktorá je typu reťazec

Nakoniec sa volaním procedúry *zapis_do_sub* zapíšu namerané údaje do súboru.

Merania časov boli uskutočnené pri minimálnej záťaži procesora. Program bol spustený viackrát. Výsledky pre jednotlivé počty prvkov poľa a algoritmy boli v jednotlivých razoch buď rovnaké, alebo veľmi podobné.

Nasledujúca tabuľka ukazuje priemerné hodnoty meraní:

n \ sort	QS	MS	HS	BS	IS	SS
5	00:00.35	00:00.42	00:00.37	00:00.30	00:00.23	00:00.30
10	00:00.78	00:00.95	00:00.92	00:00.89	00:00.43	00:00.68
15	00:01.31	00:01.58	00:01.50	00:01.87	00:00.69	00:01.24
20	00:01.81	00:02.22	00:02.22	00:03.03	00:00.87	00:01.78
25	00:02.34	00:02.97	00:02.97	00:04.53	00:01.19	00:02.41
30	00:02.89	00:03.57	00:03.80	00:06.31	00:01.61	00:03.14
35	00:03.47	00:04.29	00:04.69	00:08.56	00:02.18	00:04.22
40	00:04.45	00:05.69	00:06.52	00:13.27	00:03.30	00:06.20
45	00:05.95	00:07.83	00:08.67	00:18.69	00:04.48	00:08.31
50	00:07.36	00:09.53	00:10.65	00:24.19	00:05.69	00:10.29
55	00:08.50	00:10.80	00:12.18	00:29.50	00:06.81	00:12.15
60	00:09.46	00:11.92	00:13.70	00:34.97	00:07.99	00:14.05
65	00:10.32	00:13.03	00:15.08	00:40.70	00:09.21	00:15.95
70	00:11.21	00:14.19	00:16.55	00:46.91	00:10.50	00:17.96
75	00:12.06	00:15.31	00:17.97	00:53.56	00:11.91	00:20.19
80	00:12.97	00:16.31	00:19.58	01:00.70	00:13.37	00:22.43

2.1.3 Výsledky a závery pokusu

Zo zistených meraní možno vyčítať nasledovné:

Algoritmy triediace v čase $O(n^2)$:

Bubblesort je rýchlejší ako algoritmy zo skupiny O (n log n) len v prípade, keď je počet prvkov do 10. Ak je zoznam väčší, je nevýhodný a rastúcim počtom prvkov sa časy triedenia zhoršujú veľmi rýchlo. Potvrdilo sa teda aj tu, že bubblesort je vo všeobecnom prípade najpomalšie triedenie. Aj v prípade zoznamu s malým počtom prvkov je výhodnejšie použiť iný algoritmus bežiaci v kvadratickom čase, napríklad Selectsort alebo Insertsort.

Selectsort dáva oproti Bubblesortu dobré časové výsledky. Ak je počet prvkov zoznamu do 30, je prakticky použiteľný. V prípade počtu prvkov 30 možno vidieť, že Quicksort je už o niečo rýchlejší, no Mergesort a Heapsort sú stále pomalšie. Rastúcou hodnotou počtu prvkov sa časy Selectsortu nezhoršujú príliš rýchlo. V porovnaní s Insertsortom je však Selectsort pomalší.

Insertsort vyšiel v tomto prípade ako víťaz. Potvrdilo sa, že je v prípade zoznamov s malým počtom prvkom najrýchlejší. Je ďaleko rýchlejší, ako Bubblesort a niečo rýchlejší, ako Selectsort. 75 až 80 prvkov, to je interval, kde sa stáva Insertsort pomalší, ako Quicksort. Pre 75 prvkov a menej je najrýchlejší, pri 80 prvkoch je už rýchlejší Quicksort. Pri 80 prvkoch je však stále rýchlejší, ako Mergesort, či Heapsort. Rastúcim počtom prvkov sa časy príliš nezhoršujú, aspoň keď ide o malý počet prvkov.

Algoritmy triediace v čase $O(n \log n)$:

Quicksort je podľa zistených časov v porovnaní so svojimi kolegami triediacimi v čase $O(n \log n)$ najrýchlejším triedením, a to pre akýkoľvek počet prvkov. Pre počet prvkov 80 predbieha aj najrýchlejšie triedenie skupiny $O(n^2)$. Časy sa zväčšujúcim počtom prvkov zväčšujú pomaly, pomalšie ako v prípade algoritmov zo skupiny $O(n^2)$.

Mergesort je pre počet prvkov do 15 horší ako Heapsort, pre počet od 15 do 25 je približne rovnaký a od 30 prvkov je rýchlejší, ako Heapsort. Avšak v porovnaní s Quicksortom je horší pre ľubovoľný počet prvkov. V porovnaní s algoritmami skupiny $O(n^2)$ je lepší ako Bubblesort od 15 prvkov a lepší ako Selectsort od približne 40 prvkov. V rozsahu tabuľky nie je v žiadnom z prípadov lepší, ako Insertsort. Pretože nechcem odhadovať, od akého počtu prvkov je Mergesort rýchlejší, ako Insertsort, urobím ešte jedno meranie. Jeho popis je uvedený v nasledujúcej časti.

Heapsort je pre počet prvkov do 15 lepší, ako Mergesort, pre počet od 15 do 25 približne rovnaký a od 30 prvkov pomalší, ako Mergesort. Pre ľubovoľný počet prvkov je horší, ako Quicksort. Teda od 30 prvkov je najpomalší spomedzi algoritmov skupiny $O(n \log n)$. Čo sa týka ostatných algoritmov, od 15 prvkov je rýchlejší, ako Bubblesort a približne od 60 prvkov je lepší, ako Selectsort. Ani v prípade Heapsortu nie je z tabuľky zrejmé, od akého počtu prvkov je lepší, než najrýchlejší algoritmus skupiny $O(n^2)$. Odpoveď na túto otázku zistíme nasledujúcim programom.

Všeobecný záver:

Bez ohľadu na to, pre aký počet prvkov je Mergesort, resp. Heapsort rýchlejší, než Insertsort, môžeme poskytnúť praktický záver. Týmto záverom je, že ak je počet prvkov zoznamu malý a triedi sa veľmi veľa krát, potom rýchlejšie nie sú algoritmy triediace v čase $O(n \log n)$, ale algoritmy triediace v čase $O(n^2)$. Tak či onak, Quicksort dosahuje veľmi slušné výsledky, no ak je počet prvkov 70 a menej, je najlepšou voľbou Insertsort. Ak je počet prvkov väčší, je najlepšou voľbou Quicksort.

2.1.4 Dodatok

V ďalšom našom pokuse sme teda zvedaví, kedy sú algoritmy Mergesort a Heapsort rýchlejšie ako Insertsort v závislosti od počtu prvkov. Zisťovať to budeme pomocou programu, bude veľmi podobný, ako v predošlej časti. Rozdiely budú len vo vstupe a výstupe. Pre porovnanie budeme zaznamenávať aj časy algoritmov Quicksort a Selectsort. Na rozdiel od predošlého programu vynecháme Bubblesort, pretože bez neho bude bežať program podstatne rýchlejšie. Počty prvkov zoznamu budú násobky piatich od 70 do 160. Z dôvodu zvýšenia počtu prvkov som zmenil aj počet opakovaní z 1 000 000 na 500 000.

Výstupom je teda tabuľka s 19 riadkami a 5 stĺpcami:

n \ sort	QS	MS	HS	IS	SS
70	00:03.95	00:05.11	00:05.89	00:03.64	00:06.28
75	00:04.30	00:05.42	00:06.55	00:04.10	00:07.07
80	00:04.59	00:05.75	00:06.97	00:04.60	00:07.80
85	00:04.92	00:06.34	00:07.50	00:05.11	00:08.60
90	00:05.26	00:06.74	00:08.08	00:05.67	00:09.47
95	00:05.63	00:07.37	00:09.36	00:07.36	00:12.49
100	00:07.39	00:09.69	00:11.97	00:09.54	00:15.65
105	00:08.70	00:11.23	00:13.85	00:11.18	00:18.04
110	00:09.51	00:12.17	00:15.05	00:12.53	00:19.90
115	00:10.16	00:12.90	00:15.90	00:13.60	00:21.44
120	00:10.58	00:13.36	00:16.58	00:14.59	00:22.85
125	00:11.00	00:13.81	00:17.25	00:15.65	00:24.37
130	00:11.36	00:14.35	00:17.94	00:16.72	00:25.83
135	00:11.75	00:14.89	00:18.60	00:17.79	00:27.37
140	00:12.11	00:15.31	00:19.38	00:19.03	00:29.09
145	00:12.57	00:15.72	00:20.11	00:20.27	00:30.95
150	00:13.09	00:16.48	00:20.95	00:21.52	00:32.67
155	00:13.44	00:16.93	00:21.52	00:22.75	00:34.37
160	00:13.86	00:17.33	00:22.36	00:24.07	00:36.22

Z tabuľky vidíme, že Mergesort je pri 100 prvkoch rýchlejší, ako Insertsort, pri 105 prvkoch ide o tesný rozdiel, pri 110 prvkoch je už Mergesort rýchlejší. Záverom je, že približne pre 100 prvkov je Insertsort pri mnohonásobnom triedení zoznamu rýchlejší, ako Mergesort, pre vyšší počet prvkov je už potom rýchlejší Mergesort.

V prípade porovnania Heapsortu a Insertsortu je možné z tabuľky vyčítať, že pri 135 a 140 je ešte rýchlejší Insertsort, pri 145 a 150 je už rýchlejší Heapsort. V tomto prípade je záver, že 140 prvkov je hranica, kedy je Insertsort pri mnohonásobnom triedení malého zoznamu ešte rýchlejší, ako Heapsort.

2.2 Porovnanie Countingsortu a Quicksortu

2.2.1 Zadanie úlohy

V nasledujúcej časti sa budeme zaoberať tým, ktorý algoritmus je v závislosti od vstupných údajov rýchlejší, než ostatné. Algoritmy, ktoré majú zložitosť $O(n^2)$, uvažovať nebudeme. Z praktického hľadiska sú totiž pomalé pre veľký počet vstupných dát. Z hľadiska teoretického môžu mať zaujímavé vlastnosti. Napríklad je faktom, že ak je na vstupe utriedená postupnosť, potom je jeden z časovo najhorších algoritmov – bubblesort, rýchlejší, než quicksort, alebo countingsort, pretože mu stačí prejsť pole len raz a bez jedinej vzájomnej výmeny prvkov skončí. Týmito okrajovými prípadmi sa však bližšie zaoberať nebudeme. Budeme sa zaoberať takými vstupmi, ktoré sa môžu vyskytnúť v bežnej praxi.

Vstupné dáta budú prirodzené čísla, ktoré budú náhodne zoradené. Bude dopredu daný ich počet. V prípade Countingsortu bude známy aj rozsah – interval, v ktorom sa budú dané čísla nachádzať. Zameriame sa na algoritmy, ktoré sa považujú za najrýchlejšie.

Ak sú na vstupe prirodzené čísla menšieho rozsahu, je veľmi výhodné použiť Countingsort, triediaci v lineárnom čase. Ovšem aj Quicksort, považovaný v priemernom prípade za najrýchlejšie triedenie v čase $O(n \log n)$, je dosť rýchly. Ak je rozsah malý, je Countingsort podstatne a citeľne rýchlejší, než Quicksort. Ak rozsah zväčšíme, rozdiel sa zmenší. Ak pri rovnakom počte prvkov vstupu rozsah ešte zväčšíme, rozdiel bude zanedbateľný. Nakoniec, ak je počet prvkov oveľa menší, než rozsah údajov, je Quicksort rýchlejší. Našou úlohou bude preskúmať pomocou programu, za akých podmienok, čo sa týka počtu prvkov vstupu a veľkosti rozsahu, je Countingsort stále rýchlejší, než Quicksort.

2.2.2 Tvorba prvého programu

Porovnanie Countingsortu a Quicksortu budeme realizovať pomocou dvoch programov.

Prvý program bude mať za úlohu ukázať tri skutočnosti. Prvou z nich je, že Quicksort je pre veľký počet prvkov a malý rozsah pomalší, než Countingsort. Druhou z nich bude ukázať a uvážiť, ako sa správajú tieto triediace algoritmy, ak je rozsah a počet prvkov rovnaký. Treťou z nich bude ukázať, že pri malom počte prvkov veľkého rozsahu je Quicksort rýchlejší, než Countingsort.

Druhý program bude zameraný na zistenie, pre aký rozsah je pre pevný počet prvkov Countingsort rýchlejší, ako Quicksort. Pevný počet prvkov bude z množiny $A = \{10^n ; n \in \langle 3;7 \rangle\}$.

Dostávame sa k tvorbe prvého programu. Jeho vstup je zadaný napevno na jeho začiatku v deklarácii konštánt. Konkrétne ide o počet testovacích vstupov a dve polia – prvé z nich je rozsah prvkov vstupu a druhé z nich je počet prvkov vstupu. Program potom vytvorí dáta pre testovanie. Dáta zapíše a uloží prostredníctvom volania procedúry *makefile* do súboru *vstup.in*. Potom inicializuje dva súbory. Prvý z nich je určený pre zápis, volá sa *vystup.out*. Sem sa budú zapisovať výsledky pre jednotlivé vstupy. Druhý z nich je určený na čítanie, je to *vstup.in*, ktorý bol vytvorený procedúrou *makefile*. Aby boli časy čo najpresnejšie, je potrebné

inicializovať polia, či už tie, ktoré sa budú triediť, alebo pomocné polia. Countingsort potrebuje pre svoj beh dve pomocné polia. V programe je to pole *c*, ktoré počíta počet výskytov prvkov a pole *b*, do ktorého sa ukladá utriedené pole. Tieto sa inicializujú, nastavia sa na nulu. Potom sa dostávame k cyklu, jeho veľkosť je určená počtom vstupov. V každom cykle sa vykonáva nasledovné:

1. Načíta sa počet prvkov vstupu
2. Do poľa *testqs* sa načítajú prvky zo súboru
3. Do poľa *testcs* sa priradí po prvkoch pole *testqs*
4. Do premennej *start* sa uloží systémový čas pred triedením Quicksortom
5. Vykoná sa triedenie Quicksortom
6. Do premennej *finish* sa uloží systémový čas po triedení Quicksortom
7. Do premennej *rozdielqs* sa uloží rozdiel časov pred a po triedení
8. Body 4 až 7 sa vykonajú analogicky pre Countingsort
9. Do premenných *vysqs* a *vyscs* sa uloží čas trvania v čitateľnom formáte, ktorý zabezpečí funkcia *spracuj*
10. Na obrazovku i do súboru *vystup.out* sa zapíše výsledok. Vypíše sa číslo vstupu, počet prvkov, rozsah a časy trvania oboch triedení

Pre čo najpresnejšie meranie je potrebné, aby pri behu programu nebol procesor zaťažený ničím iným. Program neošetruje prípad, keď sa začne beh algoritmu pred polnocou a skončí po polnoci, resp. pred a po zmene zimného času na letný a opačne. Výstupom sú časy behu Countingsortu a Quicksortu pre daný počet prvkov a rozsah. Tieto čísla závisia od rýchlosti procesoru počítača, na ktorom program beží. Bez ohľadu na to, aké hodnoty vyjdú, je možné zistiť porovnanie medzi triedeniami.

2.2.3 Výsledky prvého programu

Nasledujúce tri tabuľky ukazujú výsledky, ktoré boli výstupom po troch behoch programov bežiacom na mojom počítači:

Prvý beh programu:

Vstup č.	Počet prvkov	Rozsah prvkov	Čas Quicksortu	Čas Countingsortu
1.	20 000 000	10	4,36 s	0,43 s
2.	10 000 000	10	2,03 s	0,20 s
3.	1 000 000	10	0,19 s	0,01 s
4.	1 000	1 000	0,00 s	0,00 s
5.	100 000	100 000	0,02 s	0,01 s
6.	10 000 000	10 000 000	2,89 s	4,09 s
7.	10	1 000 000	0,00 s	0,00 s
8.	10	10 000 000	0,00 s	0,11 s
9.	10	20 000 000	0,00 s	0,20 s

Druhý beh programu:

Vstup č.	Počet prvkov	Rozsah prvkov	Čas Quicksortu	Čas Countingsortu
1.	20 000 000	10	4,18 s	0,44 s
2.	10 000 000	10	2,06 s	0,20 s
3.	1 000 000	10	0,17 s	0,02 s
4.	1 000	1 000	0,00 s	0,00 s
5.	100 000	100 000	0,03 s	0,01 s
6.	10 000 000	10 000 000	2,88 s	4,09 s
7.	10	1 000 000	0,00 s	0,02 s
8.	10	10 000 000	0,00 s	0,11 s
9.	10	20 000 000	0,00 s	0,20 s

Tretí beh programu:

Vstup č.	Počet prvkov	Rozsah prvkov	Čas Quicksortu	Čas Countingsortu
1.	20 000 000	10	4,20 s	0,42 s
2.	10 000 000	10	2,07 s	0,20 s
3.	1 000 000	10	0,17 s	0,03 s
4.	1 000	1 000	0,00 s	0,00 s
5.	100 000	100 000	0,02 s	0,01 s
6.	10 000 000	10 000 000	2,86 s	4,09 s
7.	10	1 000 000	0,00 s	0,02 s
8.	10	10 000 000	0,00 s	0,10 s
9.	10	20 000 000	0,00 s	0,21 s

2.2.4 Závery z výsledkov prvého programu

Čo možno usúdiť zo zistených meraní?

V prvých troch vstupoch bol počet prvkov oveľa väčší, než rozsah. Podľa očakávania, Countingsort bol oveľa rýchlejší, vo všetkých prípadoch približne desaťnásobne.

V druhých troch vstupoch bol počet prvkov a rozsah rovnaký. Pre malý počet prvkov (1 000) je čas zanedbateľný. Pre vyšší počet prvkov (100 000) je podľa výsledkov Quicksort o niečo pomalší, no pri veľkom počte prvkov je pomalší Countingsort. Preskúmať, pre aký počet prvkov akého rozsahu je ktoré triedenie výhodnejšie, bude našim cieľom v druhom programe.

V poslednej trojici vstupov je rozsah prvkov oveľa väčší, ako ich počet. Quicksort tieto vstupy utriedi v zanedbateľnom čase, Countingsort ich utriedi mnohonásobne pomalšie. Pri desiatich prvkoch by mali zanedbateľný čas 0,00 aj algoritmy triediace v čase $O(n^2)$. Z toho vyplýva záver, že pre veľmi malý počet prvkov je Countingsort vhodný len vtedy, ak je malý aj rozsah prvkov.

2.2.5 Tvorba druhého programu

Dostávame sa k tvorbe druhého programu.

V prvom programe sme overili dva zrejmé fakty:

1. Ak je počet prvkov veľký a ich rozsah malý, vtedy je Countingsort podstatne rýchlejší ako Quicksort.
2. Ak je počet prvkov malý a ich rozsah veľký, vtedy je Quicksort podstatne rýchlejší, ako Countingsort.

Teraz sa budeme bližšie zaoberať tým, ktorý algoritmus je ako rýchly v závislosti od počtu prvkov (*n*) a rozsahu prvkov (daným premennou *k*, pričom rozsah je od 1 do *k*). Budeme testovať 12 rôznych hodnôt pre *n* a 7 rôznych hodnôt pre *k*. Množina hodnôt pre *n*, nazvime ju *A*, je nasledovná: *A* = {10 000, 20 000, 50 000, 100 000, 200 000, 500 000, 1 000 000, 2 000 000, 5 000 000, 10 000 000, 20 000 000, 50 000 000}. Množina hodnôt pre *k*, nazvime ju *B*, je daná jednoducho: ide o mocniny čísla 10 od 1 do 7, teda $B = \left\{10^i ; i \in \langle 1;7 \rangle\right\}$. Pre každý prvok karteziánskeho súčinu *A* x *B*, teda pre každú dvojicu (*a*, *b*), kde $a \in A$ a $b \in B$, vykonáme nasledovné:

1. Určí sa rozsah prvkov vstupu
2. Určí sa počet prvkov vstupu
3. Do poľa *testqs* sa načítajú prvky zo súboru
4. Do poľa *testcs* sa priradí po prvkoch pole *testqs*
5. Do premennej *start* sa uloží systémový čas pred triedením Quicksortom
6. Vykoná sa triedenie Quicksortom
7. Do premennej *finish* sa uloží systémový čas po triedení Quicksortom
8. Do premennej *rozdielqs* sa uloží rozdiel časov pred a po triedení
9. Body 5 až 8 sa vykonajú analogicky pre Countingsort
10. Do premenných *vysqs* a *vyscs* sa uloží čas trvania v čitateľnom formáte, ktorý zabezpečí funkcia *spracuj*
11. Do premenných *tabqs* a *tabcs* (dvojrozmerné polia) sa uložia výsledné časy pre daný počet prvkov a daný rozsah

Nakoniec sa vypíše do súboru *vystup.out* celá tabuľka.

Podobne ako v prvom prípade, aj tu boli jednotlivé merania časov uskutočnené pri minimálnej záťaži procesora. Program bol spustený viackrát.

2.2.6 Výsledky druhého programu

Výsledky pre jednotlivé počty prvkov a rozsahy boli v jednotlivých razoch buď rovnaké, alebo veľmi podobné. Nasledujúca tabuľka ukazuje priemerné hodnoty meraní:

Výsledky pre Quicksort:

n \ k	10	100	1000	10000	100000	1000000	10000000
10000	00:00.00	00:00.00	00:00.00	00:00.00	00:00.00	00:00.00	00:00.00
20000	00:00.00	00:00.01	00:00.00	00:00.00	00:00.00	00:00.00	00:00.01
50000	00:00.00	00:00.01	00:00.02	00:00.02	00:00.01	00:00.02	00:00.02
100000	00:00.02	00:00.02	00:00.01	00:00.03	00:00.03	00:00.03	00:00.03
200000	00:00.05	00:00.03	00:00.05	00:00.03	00:00.05	00:00.05	00:00.05
500000	00:00.08	00:00.09	00:00.09	00:00.11	00:00.11	00:00.13	00:00.13
1000000	00:00.17	00:00.19	00:00.20	00:00.23	00:00.23	00:00.25	00:00.25
2000000	00:00.38	00:00.40	00:00.43	00:00.46	00:00.48	00:00.52	00:00.53
5000000	00:00.99	00:01.04	00:01.10	00:01.15	00:01.25	00:01.35	00:01.37
10000000	00:02.01	00:02.13	00:02.25	00:02.39	00:02.51	00:02.70	00:02.85
20000000	00:04.22	00:04.39	00:04.65	00:04.86	00:05.13	00:05.50	00:05.86
50000000	00:11.08	00:11.48	00:12.20	00:13.67	00:15.39	00:17.50	00:19.37

Výsledky pre Countingsort:

n \ k	10	100	1000	10000	100000	1000000	10000000
10000	00:00.00	00:00.00	00:00.00	00:00.00	00:00.00	00:00.01	00:00.11
20000	00:00.00	00:00.00	00:00.00	00:00.00	00:00.00	00:00.02	00:00.11
50000	00:00.00	00:00.00	00:00.00	00:00.00	00:00.00	00:00.02	00:00.12
100000	00:00.00	00:00.00	00:00.00	00:00.00	00:00.02	00:00.03	00:00.14
200000	00:00.00	00:00.00	00:00.00	00:00.01	00:00.03	00:00.08	00:00.17
500000	00:00.02	00:00.02	00:00.02	00:00.05	00:00.11	00:00.18	00:00.30
1000000	00:00.03	00:00.03	00:00.05	00:00.10	00:00.21	00:00.37	00:00.48
2000000	00:00.05	00:00.05	00:00.06	00:00.18	00:00.41	00:00.75	00:00.89
5000000	00:00.09	00:00.12	00:00.15	00:00.49	00:01.03	00:01.87	00:02.08
10000000	00:00.22	00:00.25	00:00.33	00:01.00	00:02.11	00:03.75	00:04.08
20000000	00:00.42	00:00.53	00:00.66	00:02.11	00:04.37	00:07.58	00:08.19
50000000	00:01.07	00:01.33	00:01.74	00:06.33	00:12.59	00:21.63	00:24.07

2.2.7 Závery z výsledkov druhého programu

Čo možno z výsledkov vyčítať:

Ako prvé spomeniem, že tu vznikla jedna veľmi zvláštna vec. Nehľadel som na typ premennej ako prvku vstupného poľa. Vždy som uvažoval, že je typu longint, teda štvorbajtové znamienkové číslo. Samozrejme, pri nižšom rozsahu (10, 100) sa dalo uvažovať

jednobajtové číslo, pri väčšom (1000, 10000) dvojbajtové, až pri iných rozsahoch štvorbajtové, čím by sa znížila veľkosť čísla reprezentovaná v počítači a malo by to za následok zrýchlenie behu pre polia údajov nižšieho rozsahu. Práve preto, aby nevznikali pri Quicksorte veľké rozdiely medzi jednotlivými rozsahmi, som tak neurobil. Môj odhad bol, že Quicksortu bude jedno, akého sú prvky rozsahu a že prípadné rozdiely budú mať veľkosť nanajvýš pol desatiny sekundy pre malý počet prvkov a nanajvýš dve desatiny sekundy pre veľký počet prvkov.

Ukázalo sa však, že to nie je pravda, rastúcim rozsahom čísel sú rastúce aj časy behu algoritmu. Pre veľký počet prvkov - od dvoch miliónov - ide o ostro rastúce hodnoty, čím je počet prvkov väčší, tým sa rozdiely zväčšujú. Pre väčší počet prvkov (500000 až 1000000) sú rozdiely malé, ale stále ide o rastúcu postupnosť. Pri menšom počte prvkov sú už rozdiely zanedbateľné, čo môže byť spôsobené tým, že Quicksort je pravdepodobnostný algoritmus.

Prvý záver je teda neočakávané tvrdenie, že čas behu Quicksortu sa zväčšením rozsahu prvkov zvyšuje. Sám neviem, ako je to možné, nedokážem si to vysvetliť.

Keď sa pozrieme na časy Countingsortu, vyšli tak, ako sa očakávalo, teda zväčšujúcim sa rozsahom prvkov sa zvyšuje pre pevný počet prvkov čas behu algoritmu. Z tabuľky sa dá vyčítať nasledovné:

- Pre nižší počet prvkov (do 200 000) a nižší rozsah (do 10 000) je čas zanedbateľný.
- Pre nižší počet prvkov (do 200 000) a vysoký rozsah je čas síce malý, no pre daný počet prvkov je príliš vysoký vzhľadom k tomu, že Countingsort je lineárne triedenie
- Pre vyšší počet prvkov (do 2 000 000) možno vidieť, že čas behu je pre malý rozsah (do 1 000) veľmi dobrý, pre vyšší rozsah je uspokojivý. Vidno pomerne veľký časový nárast medzi rozsahom prvkov 1 000 a 10 000, 10 000 a 100 000, 100 000 a 1 000 000.
- Pre vysoký počet prvkov (od 5 000 000) vidno, že pre malý rozsah sú časy veľmi dobré. Pri vyššom rozsahu možno nahliadnuť, že časy narastajú v závislosti od rozsahu. Zaujímavosťou je, že pre malé *k* (10, 100, 1 000) nie je veľký rozdiel medzi časmi, no pri prechode z 1 000 na 10 000 je vidno niekoľkonásobný rozdiel, podobne je to i pri prechode z 10 000 na 100 000 a pri prechode zo 100 000 na 1 000 000.

Pozrime sa na porovnanie Quicksortu a Countingsortu.

Moja domnienka na začiatku pokusu bola, že Countingsort je približne rovnako rýchly, ako Quicksort, ak počet prvkov *n* a rozsah prvkov *k* sú približne rovnaké. V prípade, že je rozsah prvkov nižší, než ich počet, myslel som si, že je rýchlejší Countingsort. Výsledky ukázali, že moja domnienka nebola správna. Pre malý počet prvkov je totiž Countingsort rýchlejší ako Quicksort aj vtedy, ak je rozsah väčší, než počet. Naopak, ak je počet prvkov veľmi veľký, potom aj pri menšom rozsahu, než je počet, je rýchlejší Quicksort. Z týchto dôvodov nie je možné presne stanoviť, v akom vzájomnom pomere má byť počet a rozsah prvkov tak, aby sme mohli povedať, že práve tu je hranica, kedy je Countingsort rovnako rýchly, ako Quicksort. Je teda potrebné stanoviť určité hranice pre počet a rozsah prvkov a tak zovšeobecniť výsledok nášho pokusu.

Zovšeobecnenie vyzerá nasledovne:

- pre malý počet prvkov, do 100 000, dáva Quicksort dobré výsledky, Countingsort však dáva lepšie výsledky, aj keď rozsah prvkov je o niečo väčší, než počet prvkov (približne desaťnásobne). Jedine v prípade, že je rozsah prvkov podstatne väčší (stonásobne), je Countingsort pomalší, ako Quicksort.
- pre väčší počet prvkov, od 200 000 do 2 000 000, platí, že pre malý rozsah vedie Countingsort, no pri rozsahu rovnajúcemu sa počtu prvkov je Quickosort približne rovnako rýchly, pri hodnotách okolo milióna prvkov je dokonca v tomto prípade rýchlejší Quicksort
- pre veľký počet prvkov, od 5 000 000, je Countingsort rýchlejší len pre malý rozsah (v porovnaní s počtom prvkov), približne do 100 000. Ak je rozsah vyšší (od milióna), možno si všimnúť, že rýchlejší je Quicksort, čo je podľa môjho názoru prekvapujúce.

Celé to možno zhrnúť tak, že Countingsort je výhodné použiť najmä vtedy, ak je rozsah dostatočne malý. Obecne je výhodné pri malom počte prvkov použiť vždy Quicksort, prípadne iné triedenie triediace v čase $O(n \log n)$. Pri väčšom počte prvkov treba zvážiť, ktoré triedenie je výhodnejšie, pri veľkom počte prvkov je potrebné uvážiť, o aký rozsah ide. Ako som zistil, ak aj je rozsah ďaleko menší, než počet prvkov, môže byť Quicksort prekvapujúco rýchlejší, než Countingsort.

2.3 Porovnanie Quicksortu a Quicksortu s Insertsortom

2.3.1 Zadanie úlohy

Ak sú na vstupe prvky rôzneho charakteru (celé čísla, reálne čísla, reťazce) a nevieme dopredu ich vlastnosti, nemožno uvažovať nad použitím algoritmov so zložitosťou $O(n)$. Je potrebné použiť niektorý algoritmus, ktorý beží v čase $O(n \log n)$. Vo všeobecnosti dáva najlepšie výsledky Quicksort. Poznáme však dve verzie Quicksortu, klasický, založený len na rekuzii a kombinovaný, ktorý uvažuje, že pre nejaké k utriedi podpostupnosť veľkosti menšej, alebo rovnej ako k, nejakým algoritmom, ktorý beží v čase $O(n^2)$, napríklad Insertsortom. Takto sa vyhne nepríjemnému rekurzívnemu volaniu kvôli podpostupnosti malého rozsahu a výsledkom bude zrýchlenie algoritmu. Našou ďalšou úlohou bude preskúmať, pre aké k je kombinovaný Quicksort rýchlejší, alebo pomalší, ako jeho čisto rekurzívna verzia.

V tejto časti budeme teda skúmať, pre aké k a n je kombinácia Quicksortu a Insertsortu (pre zjednodušenie budeme ďalej hovoriť o QIsorte) rýchlejšia, ako samotný Quicksort. Podobne ako v predošlom prípade, n je počet prvkov. Premenná k tu bude znamenať niečo iné, ako rozsah. Bude to najväčší počet prvkov podpostupnosti, pre ktoré sa zavolá pri QIsorte Insertsort. Inými slovami, ak je počet prvkov (rekurzívne zavolanej) podpostupnosti viac ako k, zavolá sa Quicksort, inak sa zavolá Insertsort.

2.3.2 Tvorba prvého programu

Podobne ako v predošlej časti, na zisťovanie časov behu algoritmov použijeme program. Program, pomocou ktorého som pokus urobil, je veľmi podobný tomu v predošlej časti. Preto sa mi zdá byť zbytočné komentovať ho podrobne znova, upozorním len na hlavné skutočnosti a na to, čo je iné.

Množina hodnôt pre *n*, nazvime ju *A*, je nasledovná: *A* = {10 000, 20 000, 50 000, 100 000, 200 000, 500 000, 1 000 000, 2 000 000, 5 000 000, 10 000 000, 20 000 000, 50 000 000}. Množina hodnôt pre *k*, nazvime ju *B*, je daná jednoducho: ide o násobky čísla 10 od 10 do 70, teda $B = \{10 * i; i \in \langle 1;7 \rangle\}$. Pre každú dvojicu (*a*, *b*), kde $a \in A$ a $b \in B$, vykonáme meranie času pre Quicksort a QIsort. Algoritmus Insertsortu sa nachádza v procedúre *isort*. Bolo potrebné použiť implementáciu, ktorá utriedi len časť poľa (teda na vstupe dostane index začiatku a konca podpoľa), nie celé od prvého po posledný prvok. Polia, ktoré triedime tu, sa volajú *testqs* a *testcqs*. Prvky sú vygenerované náhodne, v intervale od 1 do 1 000 000. Výsledky ukladáme do dvojrozmerných polí *tabqs* a *tabcqs* a vypíšeme ich do súboru *vystup.out*. Program bol spustený viackrát.

2.3.3 Výsledky prvého programu

Priemerné výsledky sú zaznamenané v tabuľke:

Výsledky pre Quicksort:

n \ k	10	20	30	40	50	60	70
10000	00:00.00	00:00.00	00:00.00	00:00.00	00:00.00	00:00.00	00:00.00
20000	00:00.00	00:00.00	00:00.00	00:00.00	00:00.00	00:00.00	00:00.02
50000	00:00.01	00:00.00	00:00.02	00:00.02	00:00.00	00:00.01	00:00.00
100000	00:00.04	00:00.03	00:00.03	00:00.02	00:00.02	00:00.01	00:00.03
200000	00:00.04	00:00.05	00:00.04	00:00.05	00:00.05	00:00.05	00:00.04
500000	00:00.11	00:00.13	00:00.13	00:00.13	00:00.12	00:00.11	00:00.13
1000000	00:00.26	00:00.27	00:00.25	00:00.27	00:00.27	00:00.26	00:00.27
2000000	00:00.51	00:00.53	00:00.53	00:00.53	00:00.53	00:00.53	00:00.53
5000000	00:01.36	00:01.37	00:01.36	00:01.36	00:01.36	00:01.38	00:01.36
10000000	00:02.82	00:02.80	00:02.78	00:02.77	00:02.77	00:02.77	00:02.81
20000000	00:05.66	00:05.64	00:05.72	00:05.64	00:05.75	00:05.97	00:06.30
50000000	00:17.89	00:18.16	00:18.81	00:18.98	00:19.02	00:19.09	00:19.03

Výsledky pre kombináciu Quicksortu a Insertsortu:

n \ k	10	20	30	40	50	60	70
10000	00:00.00	00:00.00	00:00.00	00:00.00	00:00.00	00:00.00	00:00.00
20000	00:00.01	00:00.00	00:00.02	00:00.00	00:00.01	00:00.00	00:00.00
50000	00:00.00	00:00.01	00:00.00	00:00.01	00:00.02	00:00.00	00:00.01
100000	00:00.01	00:00.02	00:00.01	00:00.03	00:00.03	00:00.04	00:00.04
200000	00:00.04	00:00.03	00:00.05	00:00.04	00:00.03	00:00.03	00:00.05
500000	00:00.11	00:00.09	00:00.09	00:00.11	00:00.10	00:00.11	00:00.11
1000000	00:00.21	00:00.20	00:00.20	00:00.20	00:00.20	00:00.22	00:00.22
2000000	00:00.46	00:00.46	00:00.45	00:00.45	00:00.46	00:00.47	00:00.48
5000000	00:01.23	00:01.20	00:01.20	00:01.22	00:01.25	00:01.27	00:01.29
10000000	00:02.65	00:02.59	00:02.60	00:02.62	00:02.65	00:02.70	00:02.77
20000000	00:05.81	00:05.74	00:05.78	00:05.75	00:06.12	00:06.48	00:06.91
50000000	00:22.27	00:23.53	00:24.20	00:24.42	00:24.60	00:24.88	00:25.02

2.3.4 Závery z výsledkov prvého programu

Zo zistených meraní možno vyčítať nasledovné:

Na rozdiel od predošlého pokusu nevznikli pri Quicksorte žiadne nečakané odchýlky, pretože Quicksort premenná *k* naň nemá vôbec žiaden vplyv. Možno vidieť len drobné odchýlky pri malých hodnotách *n* (do piatich stotín sekundy) a o niečo väčšie (do približne sekundy) pri veľkých hodnotách *n*. Je to spôsobené tým, že Quicksort je pravdepodobnostný algoritmus.

Pri QIsorte, kde už premenná *k* hrá svoju úlohu, si je možné všimnúť určité rozdiely v jednotlivých stĺpcoch (hodnotách pre *k*). Pre malé hodnoty *n* sa prejavuje pravdepodobnosť algoritmu Quicksort a nemožno presne vyčítať, ako sa QIsort správa pre dané *k*. Ak je počet prvkov od 500 000 do 20 000 000, je možné si všimnúť, že pre $k = 20$ a $k = 30$ sú časy najlepšie v porovnaní s ostatnými hodnotami pre *k*. Čím je počet prvkov vyšší, tým sú vyššie aj rozdiely. Zvláštnosťou však je, že pri počte prvkov 50 000 000 neplatí, že najlepšie časy sú pre hodnoty $k = 20$ a $k = 30$, tam je to rastúce a najlepší čas je tak pre $k = 10$. Sám si to vysvetľujem tak, že pri takom veľkom počte prvkov sa triedenie Insertsortom vykonáva častejšie, než pri menších hodnotách prvkov poľa a algoritmus má tak viac charakter algoritmu v čase $O(n^2)$, ako pri nižších počtoch prvkov.

Pokiaľ chceme porovnať Quicksort a QIsort, vychádza nám pekne nahliadnuteľný výsledok. Už pri menšom počte prvkov (do 500 000) vidíme, že QIsort je v priemernom prípade rýchlejšia, než Quicksort. Rozdiely sú nepatrné, lebo oba algoritmy utriedia malý počet prvkov rýchlo. Lepšie viditeľné rozdiely sú pri počte prvkov od milióna do desiatich miliónov, miestami aj dve desatiny sekundy. Zvláštne je, že pre dvadsať a päťdesiat miliónov to už neplatí a pre hocijaké *k* víťazí Quicksort. Ešte pre 20 000 000 je rozdiel malý, do 20 stotín sekundy, pri $n = 50\,000\,000$ je už podstatne väčší, až 6 sekundový. Je možné, že pri týchto hodnotách má QIsort charakter Insertsortu natoľko veľký, že je to na úkor celkového času.

Záver teda je, že ak je počet prvkov približne do 10 000 000 miliónov, je dobré použiť kombináciu Quicksortu a Insertsortu. Pre o niečo vyšší počet prvkov je to približne rovnaké, pre veľký počet prvkov (20 000 000 a viac) je dobré ostať pri Quicksorte.

V uvedenom závere sme neuviedli, akú hodnotu *k* treba použiť. Ako som naznačil, zdá sa, že najrýchlejší je QIsort pre hodnoty *k* = 20 a *k* = 30. (uvažujeme počet prvkov 10 000 000 a menej). Pre nižšie hodnoty ako *k* = 20, alebo väčšie, ako *k* = 30, je QIsort pomalší.

2.3.5 Tvorba druhého programu

Ďalší program bude mať za úlohu preskúmať, ako sa správa Quicksort a QIsort pre hodnoty *k* z intervalu od 21 do 29. Popis programu robiť netreba, je úplne identický, ako predošlý, jediné, čo sa mení, je množina *B*, teda množina hodnôt pre premennú *k*. Množina *B* je v tomto programe teda *B* = {21, 22, 23, 24, 25, 26, 27, 28, 29}. Samozrejme, mení sa počet stĺpcov tabuľky zo 7 na 9. Vzhľadom k tomu, že 9 stĺpcov by sa ťažko zmestilo na túto stránku, uvediem tabuľku hodnôt pre *k* od 21 do 27. Budeme vidieť, že to bude pre naše potreby stačiť. Program je urobený tak, aby porovnal Quicksort a QIsort. Toto sme však už urobili v predošlom programe a je zbytočné preto uvádzať časy pre Quicksort. Bude nám stačiť len tabuľka pre QIsort.

2.3.6 Výsledky druhého programu

Uvedená tabuľka je tu:

n \ k	21	22	23	24	25	26	27
10000	00:00.02	00:00.00	00:00.00	00:00.00	00:00.00	00:00.00	00:00.00
20000	00:00.00	00:00.01	00:00.00	00:00.02	00:00.00	00:00.01	00:00.00
50000	00:00.02	00:00.02	00:00.00	00:00.00	00:00.00	00:00.00	00:00.02
100000	00:00.01	00:00.02	00:00.02	00:00.01	00:00.01	00:00.02	00:00.01
200000	00:00.04	00:00.05	00:00.05	00:00.03	00:00.03	00:00.04	00:00.03
500000	00:00.10	00:00.10	00:00.10	00:00.10	00:00.10	00:00.10	00:00.11
1000000	00:00.20	00:00.21	00:00.20	00:00.20	00:00.20	00:00.20	00:00.21
2000000	00:00.44	00:00.43	00:00.43	00:00.42	00:00.44	00:00.43	00:00.42
5000000	00:01.17	00:01.16	00:01.17	00:01.16	00:01.18	00:01.15	00:01.19
10000000	00:02.53	00:02.52	00:02.48	00:02.47	00:02.48	00:02.51	00:02.50
20000000	00:05.72	00:06.00	00:06.30	00:06.54	00:06.75	00:06.93	00:07.04
50000000	00:23.64	00:23.89	00:24.09	00:24.12	00:24.20	00:24.20	00:24.30

2.3.7 Závery z výsledkov druhého programu

Z tabuľky vidno, že je pomerne ťažké určiť presnú hodnotu *k*. Prejavuje sa tu náhodnosť algoritmu a výsledné časy sú podobné, aspoň pokiaľ ide o počet prvkov 10 miliónov a menej. Je možné prehlásiť to, že ak je *k* v intervale od 20 do 30, je čas behu QIsortu najrýchlejší. Ak

by sme chceli byť úplne presní, je možné z uvedených čísel vyčítať, že pre $k = 24$ sú časy lepšie, než pre iné hodnoty. Ide však len o veľmi malé odchýlky.

Aj v tomto pokuse sa potvrdilo, že ak triedime pole s počtom prvkov 20, resp. 50 miliónov, potom sú časy utriedenia ostro rastúce. Teda aj tu platí záver, že pre veľmi veľký počet prvkov možno odporučiť samotný Quicksort.

2.3.8 Dodatok

Z tabuliek sa dá nahliadnuť, že pre veľmi veľký počet prvkov, 20 000 000 a viac, je Quicksort rýchlejší, ako QIsort. Aspoň tak to platí pre hodnoty *k* od 10 do 70. Avšak čím je hodnota *k* menšia, tým je čas behu QIsortu lepší.

Z tohto dôvodu preskúmame pre veľký počet prvkov, aké časy má QIsort pre *k* z intervalu od 3 do 9. Program je opäť veľmi podobný, ako v predošlom prípade, len sa mení množina *A* a množina *B*. Množina *A* je definovaná ako $A = \{i * 10^7 ; i \in \langle 2;6 \rangle\}$ a množina *B* je definovaná ako $B = \{3, 4, 5, 6, 7, 8, 9\}$. Keďže v tomto prípade opäť porovnávame Quicksort a QIsort, merajú sa časy pre obe z triedení. Výsledky ukazujú nasledujúce dve tabuľky:

Výsledky pre Quicksort:

n \ k	3	4	5	6	7	8	9
20000000	00:05.57	00:05.51	00:05.50	00:05.53	00:05.54	00:05.58	00:05.56
30000000	00:10.39	00:10.86	00:10.63	00:10.27	00:10.86	00:10.45	00:10.47
40000000	00:15.53	00:15.81	00:15.98	00:15.84	00:15.81	00:15.86	00:15.83
50000000	00:19.96	00:19.81	00:19.92	00:19.81	00:19.78	00:19.75	00:19.71
60000000	00:23.78	00:23.73	00:23.67	00:23.75	00:23.77	00:23.84	00:23.86

Výsledky pre kombináciu Quicksortu a Insertsortu:

n \ k	3	4	5	6	7	8	9
20000000	00:05.84	00:05.80	00:05.78	00:05.77	00:05.77	00:05.76	00:05.72
30000000	00:11.24	00:11.48	00:11.58	00:11.72	00:12.34	00:12.75	00:12.90
40000000	00:17.94	00:18.35	00:18.85	00:18.74	00:18.78	00:18.78	00:18.82
50000000	00:23.81	00:24.00	00:24.61	00:24.62	00:24.83	00:24.72	00:24.82
60000000	00:29.36	00:29.83	00:30.74	00:30.92	00:31.33	00:31.63	00:31.71

Výsledky ukázali, že pre 20 000 000 prvkov sa časy QIsortu pre zväčšujúce *k* zlepšujú, hoci oproti časom Quicksortu sú horšie, i keď rozdiel veľký nie je. Pre 30 000 000 a viac prvkov tu pre časy QIsortu platí, že zväčšujúcim *k* sa časy zhoršujú a čím je počet prvkov väčší, tým viac je horší aj rozdiel medzi časmi QIsortu a Quicksortu. Záverom je teda tvrdenie, ktoré som vyslovil už v predošlej časti, a síce, že pre veľké zoznamy prvkov je výhodné použiť Quicksort, pretože kombinácia Quicksortu a Insertsortu nie je rýchlejšia ani pre malé hodnoty *k*.

Kapitola 3

Záver

Výsledkom mojej práce je prehľad základných triediacich algoritmov a niekoľko poznatkov, ktoré som zistil pomocou programov.

Pri mnohonásobnom triedení prvkov malého rozsahu som zistil, že pokiaľ je počet prvkov menší, alebo rovný, ako 70, je výhodné použiť Insertsort, v prípade, že je prvkov viac, ako 70, je výhodné použiť Quicksort.

Pri porovnávaní Countingsortu a Quicksortu som prišiel na to, že Countingsort možno použiť vždy, keď je rozsah prvkov malý. Countingsort je výhodný aj vtedy, ak nie je veľký rozsah a počet prvkov (do 1 000 000). V prípade, že je počet prvkov veľký, odporúčam použiť Quicksort.

Pri porovnávaní Quicksortu a kombinácie Quicksortu a Insertsortu som prišiel na to, že kombinácia Quicksortu a Insertsortu je pre malý a väčší počet prvkov (do 10 000 000) rýchlejšia, než Quicksort. Najlepšie výsledky dosahuje kombinácia triedení pre približné hodnoty k rovné 24, kde k je najväčší počet prvkov podpostupnosti, ktorá sa triedi Insertsortom. V prípade, že je počet prvkov veľmi veľký, je podľa mojich zistení lepší samostatný Quicksort, než kombinácia Quicksortu a Insertsortu.

Podrobnejšie závery sú uvedené v častiach jednotlivých kapitol, konkrétne 2.1.3, 2.2.4, 2.2.7, 2.3.4 a 2.3.7.

Pri tvorbe práce som si zopakoval a podrobnejšie preštudoval triediace algoritmy. Pretože som zisťoval časy behu algoritmov pomocou programov, precvičil som si svoje programátorské schopnosti a pozrel sa na triediace algoritmy aj z hľadiska praktického. Výsledky namerané pomocou programov sú zaujímavé a niektoré z nich ma prekvapili. Verím, že moja práca zaujme každého jej čitateľa.

Literatúra

[1] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman: The design and analysis of computer algorithms, Addison – Wesley, 1974

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest: Introduction to algorithms, MIT Press, Cambridge, 1990

[3] Niklaus Wirth: Algoritmy a štruktúry údajov, ALFA, Bratislava, 1987

[4] http://en.wikipedia.org/wiki/Sorting_algorithm

[5] http://linux.wku.edu/~lamonml/algor/sort/sort.html