1.1. Najväčší spoločný deliteľ (Greatest common divisor)

Definujme najväčší spoločný deliteľ celých čísel *a*, *b* ako (*a*,*b*) (niekedy píšeme aj nsd(*a*,*b*)$_{(\gcd(a,b))}$) ako najväčšie také prirodzené číslo *n*, také, že *n* delí *a* aj *b* bezo zvyšku.

1.1.1. Euklidov algoritmus

Efektívnu metódu výpočtu NSD vynašli už staroveký Gréci pred vyše 2000 rokmi. Zakladá sa na jednoduchom pozorovaní, že najväčší spoločný deliteľ čísel *a* a *b* (*a*>*b*) je rovnaký ako najväčší spoločný deliteľ čísel *b* a *a*-*b*. Toto nám postačuje na skonštruovanie programu.

1.1.2. Realizácia v programovacom jazyku

Vymyslieť riešenie problému je jedna vec, dotiahnuť ho do konca, aby bolo funkčné, je vec druhá, a preto je dôležité sa naučiť sa písať prehľadné a jednoduché programy. S jazykom Pascal ste sa asi už stretli a preto tu prezentujeme riešenie napísané v ňom. Medzi vami sa nachádzajú aj priaznivci jazyka C, na učebné účely sa považuje za nevhodný jazyk, avšak v dnešnej dobe sa považuje za akúsi „angličtinu“ medzi jazykmi. Na napísanie efektívneho programu vám postačuje malá znalosť jeho syntaxe.

```
var x,y:integer;
function gcd(u,v:integer):integer;
  var t:integer;
begin
  repeat
    if u<v then
      begin t:=u; u:=v; v:=t end;
    u:=u-v;
  until u=0;
  gcd:=v;
end;
begin
while not eof do
  begin
  readln(x,y);
  if (x>0) and (y>0) then
    writeln(x,y,gcd(x,y));
  end;
end.
```

```
#include <stdio.h>
int gcd(int u,int v)
{
  int t;
  while(u > 0)
  { if (u < v)
    { t = u; u = v; v = t; }
    u = u-v;
  }
  return v;
}
main()
{
  int x,y;
  while (scanf("%d %d",&x,&y) != EOF)
    if (x>0 && y>0)
      printf("%d %d %d\n",x,y,gcd(x,y));
}
```

Keď si porovnáte program v jazyku Pascal a C, objavíte len nepatrné rozdiely.

1.1.3. Majstrovská implementácia

Vždy keď implementujeme nejakú myšlienku algoritmu, treba myslieť nato ako to efektívne „preložiť“ do reči programovacieho jazyka. Niekedy treba zvoliť určitý kompromis v rýchlosti, aby sme dosiahli čistú a zrozumiteľnú implementáciu, ktorá možno niekedy ide len nepatrne pomalšie, avšak čas potrebný na je napísanie je obyčajne výrazne nižší [1] . Nasledovná implementácia funkcie `gcd` je presne takýto príklad(predpokladáme, že parametre `u`, `v` sú nezáporné čísla):

```
function gcd(u,v:integer):integer;
begin
  if v=0 then
    gcd:=u
  else
    gcd:=gcd(v,u mod v);
end;
```

```
int gcd(int u,int v)
{
  if(v==0)
    return u;
  return gcd(v,u%v);
}
```

[1] Niekedy sa to v označuje aj ako princíp bozku (KISS principle – Keep It Simple Stupid), teda treba vedieť, čo možno najrýchlejšie, spraviť jednoduché „hlúpe riešenie“, ktoré funguje pre daný problém.

1.2. Zásobník, Fronta (Pushdown Stack, Queue)

V tejto časti nahliadneme na základné spôsoby organizácie dát v programoch. V mnohých aplikáciách je hlavným problémom práve výber vhodnej dátovej štruktúry. Keď sa už spraví toto rozhodnutie, ďalej si to vyžaduje len jednoduché algoritmy. Pre rovnaké dáta, rôzne dátové štruktúry vyžadujú rôzne množstvo miesta v pamäti; pre rovnaké operácie, rôzne dátové štruktúry vedú k rôzne efektívnym algoritmom.

Zväčša sa zaujímame o dátové štruktúry, ktoré môžu k dátam ľubovoľne pristupovať (prípadne ich ľubovoľne pridávať a mazať). Teraz sa začneme zaoberať so štruktúrami, ktoré majú prístup k jednotlivým prvkom nejako obmedzený. Uvidíme, že nami definované obmedzenia nie sú až také vážne, a vlastne, že práve tieto obmedzenia nám umožňujú jednoduchšiu realizáciu týchto dátových štruktúr v počítači.

1.2.1. Zásobník (Pushdown Stack)

Najdôležitejšia dátová štruktúra s obmedzeným prístupom je zásobník. Vyžadujeme dve základné operácie: *polož*$_{(push)}$ prvok na zásobník (vlož ho na začiatok) a *vyber*$_{(pop)}$ prvok zo zásobníka (zober ho zo začiatku). Zásobník funguje podobne ako zaneprázdnený „úradník v škatuľke“. Práca sa mu hromadí na stole do vysokej kopy papierov, vždy, keď je pripravený už konečne niečo spraviť, zoberie si prácu zvrchu kopy. Toto môže, žiaľ, znamenať, že niečo zostane dlho nepovšimnuté naspodku kopy, avšak dobrý úradník, predpokladajme, zvládne periodicky dokončovať svoju kopu práce na stole.

Možno je to pre nás prekvapivé, mnohé počítačové programy organizujú svoju prácu rovnako, prerušujú prácu na niektorých úlohách, zatiaľ čo vykonávajú iné, aby sa niekedy neskôr zasa vrátili k tým úlohám, ktoré prerušili. Presne v týchto situáciách je zásobník dôležitá dátová štruktúra.

Existujú rôzne implementácie zásobníku, pokiaľ nevieme zhora ohraničiť množstvo, ktoré budeme spracovávať, tak je vhodné využiť spájaný zoznam$_{(linked-list)}$ (dynamicky alokovať pamäť pre každý nový prvok) zásobníku. Tento postup je síce teoreticky správny, ale v praktických podmienkach nie veľmi vhodný. Ak vieme maximálnu veľkosť zásobníku, ktorú budeme potrebovať, určiť vopred, vhodná je nasledujúca statická implementácia:

```
var stack:array[1..Max]of integer;
    head:integer;

procedure push(v:integer);
begin { pridaj }
  stack[head]:=v; inc(head);
end;
function pop:integer;
begin { zober }
  dec(head); pop:=stack[head];
end;

procedure stack_init;
begin { vyprázdni }
  head:=1;
end;
function stack_empty:boolean;
begin { prázdny zásobník? }
  stack_empty:=head=1;
end;
```

```
/* zásobník pre Max prvkov */
int stack[Max],head;

void push(int v)
{ /* pridaj */
  stack[head++]=v;
}
int pop(void)
{ /* zober */
  return stack[--head];
}

void stack_init(void)
{ /* vyprázdni */
  head=0;
}
int stack_empty(void)
{ /* prázdny zásobník? */
  return !head;
}
```

1.2.2. Fronta (Queue)

Ďalšia významná štruktúra s obmedzeným prístupom k prvkom, ktoré sú v nej, je fronta. Znovu vyžadujeme len dve operácie: *vlož*(insert) prvok do fronty na začiatok a *vyber*(remove) prvok z jej konca. Náš „úradník v škatuľke“ by vlastne mal pracovať ako fronta, potom by práca, ktorá k nemu príde skôr, bola skôr aj vykonaná. V zásobníku môže práca zapadnúť hlboko na spodok, ale vo fronte bude všetko spracované v poradí v akom to k nej prišlo!

Tieto princípy poradia sa niekedy označujú pre zásobník ako LIFO(last in, first out) – ten čo príde neskôr, bude skôr na rade. Pre frontu to je FIFO(first in, first out) – ten čo príde skôr, bude skôr na rade. V našej praxi sa stretávame častejšie s frontami ako so zásobníkmi, pretože zásobníky sú nám mnohokrát skryté za rekurzívne volania funkcii. Obe tieto štruktúry však tvoria veľký základ dátových štruktúr.

Rovnako ako pre zásobníky, existujú rôzne implementácie fronty. Jedna z nich, využívajúca spájané zoznamy, je pre naše potreby nevhodná. Aj tu zostaneme pri statickej implementácií pomocou poľa, ktoré budeme pre lepšiu efektívnosť cykliť:

```
var queue:array[1..Max]of integer;
    head,tail:integer;

procedure put(v:integer);
begin { pridaj }
  queue[tail]:=v; inc(tail);
  if tail > Max then
    tail:=1;
end;
function get:integer;
begin { vyber }
  get:=queue[head]; inc(head);
  if head > Max then
    head:=1;
end;

procedure queue_init;
begin { vyprázdni }
  head:=1; tail:=1;
end;
function queue_empty:boolean;
begin { prázdna fronta? }
  queue_empty:=head=tail;
end;
```

```
/* fronta pre Max-1 prvkov */
int queue[Max],head,tail;

void put(int v)
{ /* pridaj */
  queue[tail++]=v;
  if(tail >= max)
    tail=0;
}
int get(void)
{ /* vyber */
  int t=queue[head++];
  if(head >= Max)
    head=0;
  return t;
}
void queue_init(void)
{ /* vyprázdni */
  head=tail=0;
}
int queue_empty(void)
{ /* prázdna fronta? */
  return head==tail;
}
```

1.2.3. Abstraktný dátový typ

Ako sme videli vyššie, je často pohodlné popisovať algoritmy a dátové štruktúry len podľa operácií ktoré nám umožňujú vykonávať, ako podľa detailov ich implementácie. Keď dátovú štruktúru popisujeme len podľa operácií, ktoré má byť schopná vykonávať, voláme ju *abstraktný dátový typ*.

Hlavný dôvod prečo by sme sa mali na dátové štruktúry pozerať pod týmto uhlom je ten, že nám to poskytuje mechanizmus na lepšie pochopenie veľkých programov, v ktoré sme potom schopný jednoduchšie upravovať.

1.3. Prioritná Fronta (Priority Queue)

V mnohých aplikáciách spracúvame dáta v určitom poradí, nevyžadujeme, aby sme vedeli presné poradie všetkých prvkov; nie vždy spracúvame všetky prvky naraz. Často, dostaneme nejaké prvky, spracujeme najväčší prvok, dostaneme ďalšie prvky, zasa spracujeme najväčší atď. V takýchto situáciách sa nám hodí dátová štruktúra, ktorá podporuje operácie:

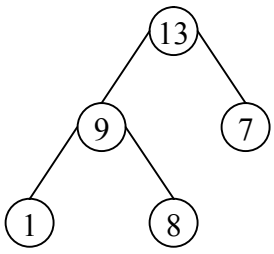
- pridanie prvku
- zmazanie najväčšieho

Takáto dátová štruktúra, ktorú môžeme prirovnávať k zásobníku alebo fronte sa nazýva prioritná fronta. Prioritnú frontu si môžeme predstaviť aj ako zovšeobecnenie zásobníka, fronty a iných jednoduchých dátových štruktúr, pretože tieto štruktúry sa dajú realizovať prioritnými frontami s vhodnými nastaveniami priorít.

1.3.1. Halda(Heap)

Najjednoduchší spôsob ako si reprezentovať prioritnú frontu je v poli. Do takejto štruktúry síce vieme rýchlo pridávať prvky v konštantnom čase, ale zmazať najväčší by nám trvalo až lineárny čas. Pri implementácií prioritnej fronty treba dbať nato, aby časy pre jednotlivé operácie boli vyvážené, pričom stále krátke.

Efektívny postup predstavuje halda. Halda je úplný binárny strom, v ktorom priorita každého vrcholu je väčšia ako priorita jeho detí. Nasleduje príklad haldy o N prvkoch, $N=5$:



Haldu napĺňame prvkami zľava doprava úroveň po úrovni, t.j. strom je úplný, až na svoju najnižšiu úroveň), a teda si ju tiež môžeme uchovávať v poli veľkosti N, v ktorom sú zapamätané jednotlivé úrovne postupne. Najskôr koreň (prvok s najvyššou prioritou), potom jeho deti (v poradí zľava doprava – ako na obrázku), potom deti ich detí, atď.

Táto reprezentácia je prirodzená aj v tom zmysle, že pre daný vrchol v_i vieme presne(pokiaľ existujú), že jeho rodič je $v_{\lfloor \frac{i}{2} \rfloor}$ a jeho deti sú v_{2i}, v_{2i+1}.

Všetky algoritmy na haldách pracujú na nejakej ceste z koreňa do listu – z najvyššej úrovne do najnižšej. Jednoducho vidíme, že v halde o N prvkoch je každá cesta dlhá najviac $O(\log N)$. Všetky jednoduchšie operácie na halde budú teda trvať logaritmický čas od počtu prvkov N.

Zostáva nám implementovať operácia pridania prvku a vybrania najväčšieho. Už sme si spomínali tzv. podmienku haldy(heap condition) − každý vrchol je väčší ako jeho deti. Obe tieto operácie najskôr narušia túto vlastnosť a potom ju znovu obnovia - zmeny, ktoré sme urobili budú zapracované do korektnej haldy.

Haldy si budeme pamätať v poli na pozíciách *1,...,Max* vrátane. V jazyku C teda musíme vytvoriť pole o jeden prvok (nulu) väčšie. Program, využívajúci haldu zvykne ďalej volať už len funkcie *insert* a *remove*.

Nasleduje implementácia v jazykoch Pascal a C:

```
var heap:array[1..Max] of integer;
    n:integer;

{ obnov haldu smerom hore }
procedure up(i:integer);
var k:integer;
begin
  while i>1 do
  begin
    if heap[i div 2] < heap[i] then
      break;
    k:=heap[i];
    heap[i]:=heap[i div 2];
    heap[i div 2]:=k;
    i:=i div 2;
  end;
end;
{ vlož prvok }
procedure insert(v:integer);
begin
  inc(n); heap[n]:=v;
  up(n);
end;
{ obnov haldu smerom dole }
procedure down(i:integer);
var j,k:integer;
begin
  while 2*i <= n do
  begin
    j:=2*i;
    if j+1 <= n then
      if heap[j+1] < heap[j] then
        inc(j); { väčšie z detí }
    if heap[i] < heap[j] then
      break;
    k:=heap[i]; heap[j]:=heap[i];
    heap[j]:=k;
    i:=j;
  end;
end;
{ vyber najväčší }
function remove:integer;
begin
  remove:=heap[1];
  heap[1]:=heap[n]; dec(n);
  down(1);
end;
```

```
int heap[Max+1],n;

/* obnov haldu smerom hore */
void up(int i)
{
  int k;

  while (i>1)
  {
    if(heap[i/2]<heap[i])
      break;
    k=heap[i];
    heap[i]=heap[i/2];
    heap[i/2]=k;
    i=i/2;
  }
}
/* vlož prvok */
void insert(int v)
{
  heap[++n]=v;
  up(n);
}
/* obnov haldu smerom dole */
void down(int i)
{
  int j,k;
  while(2*i<=n)
  {
    j=2*i;
    if(j+1<=n)
      if(heap[j+1]<heap[j])
        j++; /* väčšie z detí */
    if(heap[i]<heap[j])
      break;
    k=heap[i];heap[j]=heap[i];heap[j]=k;
    i=j;
  }
}
/* vyber najväčší */
int remove(void)
{
  int k=heap[1];
  heap[1]=heap[n--];
  down(1);
  return k;
}
```

1.3.2. Heapsort

Elegantná a efektívna metóda triedenia funguje presne na princípe haldy. Idea je jednoduchá, postavme s daných N prvkov, ktoré chceme utriediť, haldu a potom ich v utriedenom poradí postupne s haldy vyberme. Pre N prvkov nám triedenie bude trvať čas $O(N \log N)$. Na prvý pohľad sa zdá, že algoritmus vyžaduje pre túto haldu $O(N)$ pamäti; dá sa to však realizovať aj bez ďalšej pomocnej pamäti, čo nám dáva asymptoticky optimálny triediaci algoritmus. V praxi sa však ukáže, že existuje algoritmus, ktorý je schopný utriediť rovnaké množstvo prvkov asi dvakrát rýchlejšie.

1.4. Quicksort (Quicksort)

Daných je *N* prvkov, potrebujeme ich utriediť do vzostupného poradia. Úlohy tohto typu riešia triediace algoritmy. Budeme sa zaoberať pravdepodobne najpoužívanejším triediacim algoritmom, *Quicksort*om. Základný algoritmus navrhol v roku 1960 C. A. R. Hoare a odvtedy bol skúmaný mnohými ľudmi. *Quicksort* je populárny, pretože nie je ťažký na implementáciu, všeobecne funguje dobre a vyžaduje menšie množstvo zdrojov ako akýkoľvek iný triediaci algoritmus.

Dôležitými vlastnosťami *Quicksort*u je, že je to triedenie na mieste(in-place) - vyžaduje len malú pomocnú pamäť, vyžaduje okolo *N* log *N* operácií na utriedenie *N* prvkov a má extrémne krátku vnútornú slučku. Nevýhodou je, že je rekurzívny (ťažko sa implementuje, keď nemožno použiť rekurziu), potrebuje okolo N^2 operácií v najhoršom prípade a je príliš „zraniteľný“ v tom, že už malá chybička v implementácií môže spôsobiť, zlú funkčnosť len v niektorých prípadoch.

1.4.1. Základný princíp

Quicksort je založený na tzv. metóde rozdeľuj-a-panuj(divide-and-conquer). Aby utriedil *N* prvkov, rozdelí si ich na dve časti – obsahujúce menšie a väčšie prvky – ktoré potom utriedi. Ako uvidíme neskôr, samotné miesto predelu závisí na spracovaných prvkov, takže algoritmus má nasledovnú rekurzívnu štruktúru:

```
Quicksort(A,l,r)
{
   if l<r then
   {
      i ← Rozdeľ(A,l,r)
      Quicksort(A,l,i);
      Quicksort(A,i+1,r);
   }
}
```

Rozdeľuj:
Prvky poľa *A[l...r]* sú preusporiadané do menších polí *A[l…i]* a *A[i+1…r]* tak, že každý prvok *A[l…i]* je menší alebo rovný ako všetky prvky poľa *A[i+1...r]*. Index *i* sa vypočíta pri procese rozdeľovania.

Panuj:
Menšie polia *A[l…i]* a *A[i+1…r]* sú rekurzívne utriedené. Pôvodné pole *A[l…r]* bude teda nakoniec utriedené.

Zrejme, kľúčová vec algoritmu je spôsob rozdelenia daného poľa *A[l...r]*. Všimnime si i-ty prvok. Prvok *A[i]* bude v koncovom utriedení na správnom mieste. Chceme teraz prvky poľa *A* poprehadzovať tak, aby všetky prvky *A[1],…,A[i-1]* boli menšie alebo rovné ako *A[i]* a všetky prvky *A[i+1],…,A[r]* boli väčšie alebo rovné ako *A[i]*.

1.4.2. Rozdelenie

Správne rozdelenie nájdeme pomocou jednoduchej stratégie. Najskôr, ľubovoľne, vyberme prvok *A[r]* ako prvok, ktorý pôjde na svoje správne miesto. Teraz hľadajme z ľavého konca poľa prvok väčší ako *A[r]*, a z pravého konca poľa prvok menší ako *A[r]*. Tieto dva prvky sú zrejme v nesprávnom poradí, vymeňme ich.

Pokračujúc týmto spôsobom (ľavý bežec - zľava hľadáme prvky väčšie ako deliaci prvok, pravý bežec - sprava prvky menšie) budú všetky prvky naľavo od ľavého bežca menšie ako *A[r]* a napravo od pravého bežca väčšie ako *A[r]*. Keď sa bežci stretnú deliaci proces je skoro u konca. Zostáva vymeniť prvok *A[r]* s najľavejším prvokom pravého poľa (vymedzeného pravým bežcom).

Zjavne proces rozdeľovania nie je stabilný, pretože prvok môže byť pri výmene presunutý cez veľké množstvo rovnakých prvkov.

1.4.3. Implementácia

Nasledujúci program implementuje *Quicksort* v programovacom jazyku Pascal a C. Program najskôr demonštratívne vygeneruje 1000 náhodných čísel, ktoré následne utriedi:

```
program Quicksort;
{$R-,S-}
{ vypni Range-Check a Stack-Check }

const
  Max = 1000;
type
  List = array[1..Max] of Integer;
var
  Data: List;
  i: Integer;

procedure QuickSort(var A: List; left,
right: Integer);

procedure Sort(l, r: Integer);
var
  i, j, x, y: integer;
begin
  i := l; j := r; x := a[(l+r) DIV 2];
  repeat
    while a[i] < x do i := i + 1;
    while x < a[j] do j := j - 1;
    if i <= j then
    begin
      y := a[i]; a[i] := a[j]; a[j] := y;
      i := i + 1; j := j - 1;
    end;
  until i > j;
  if l < j then Sort(l, j);
  if i < r then Sort(i, r);
end;
begin {Quicksort};
  Sort(left,right);
end;

begin
  { 1000 náhodných čísel }
  Randomize;
  for i := 1 to Max do
    Data[i] := Random(30000);
  { triedime }
  QuickSort(Data, 1, Max);
  for i := 1 to Max do Write(Data[i]:8);
end.
```

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define Max 1000

int Data[Max];

void QuickSort(int a[], int l,int r)
{
  int v,i,j,t;

  if(r > l)
  {
    v=a[r]; i=l-1;j=r;
    for(;;)
    {
      while(a[++i]<v);
      while(a[--j]>v);
      if(i>=j)
        break;
      t=a[i];a[i]=a[j];a[j]=t;
    }
    t=a[i]; a[i]=a[r]; a[r]=t;
    QuickSort(a,l,i-1);
    QuickSort(a,i+1,r);
  }
}

int main(void)
{
  int i;

  /* 1000 náhodných čísel */
  srand((unsigned int)time(NULL));
  for(i=0;i<Max;i++)
    Data[i]=rand()%30000;
  /* triedime */
  QuickSort(Data,0,Max-1);
  for(i=0;i<Max;i++)
    printf("%8d",Data[i]);
  return 0;
}
```

1.4.4. Degenerované prípady

Veľmi rušivá vlastnosť tohto triedenia je, že je veľmi neefektívne na jednoduché súbory. Predstavme si triediť súbor čísel, ktoré sú už v utriedenom poradí. Rozdelenia budú degenerované, a program sa zavolá *N* krát, vždy vyhodí len jeden prvok na volanie. To neznamená, že len čas potrebný na triedenie bude okolo

$N^2/2$, ale, že tiež množstvo potrebnej pamäte na rekurziu bude okolo N, čo je pre praktické použitie neprípustné. Existujú viaceré riešenia z tejto šlamastiky.

Predstavme si súbor s množstvom rovnakých prvkov, ukazuje sa, že je lepšie rozdeľovať prvky do troch menších celkov, pre menšie prvky, pre väčšie a pre rovné rozdeľovaciemu prvku. K tomuto nám dopomôže, keď bežcov, ktorý hľadajú prvky, ktoré sú v nesprávnom poradí budeme zastavovať aj na prvkoch, ktoré majú hodnotu rovnakú ako rozdeľovací prvok.

Pre malé súbor sa ukazuje, že nie je výhodný takýto rekurzívny postup. Testovanie ukázalo, že pre súbory do 8 prvkov je výhodnejšie namiesto rekurzie použiť triedenie priamym vkladaním(insertion sort).

Posledným úskalím je výber deliaceho prvku. Najbezpečnejšia cesta by bola vybrať náhodný prvok, potom by najhorší prípad nastal len s minimálnou pravdepodobnosťou. Toto je príklad pravdepodobnostného algoritmu, ktorý dosahuje dobré výsledky skoro vždy bez ohľadu na vstupné dáta. Niekedy sa zvykne vybrať niekoľko prvkov a z nich potom vybrať mediána (prostredný prvok) ako deliaci prvok. V praxi sa ale ukazuje, že výkonnosť algoritmu sa neovplyvní, nejakým príliš premyslením výberom, a používa sa hodnota prostredného prvku poľa, vďaka čomu degenerované prípady nastanú len vo veľmi podivných usporiadaniach vstupných čísel.

1.4.5. Vstavaná implementácia v jazyku C

Jazyk C už obsahuje triedenie *Quicksort* ako svoju základnú funkciu, demonštrujeme spôsob jej použitia v príklade:

```
int int_cmp(const void *va, const void *vb)
{
      int *ia=(int*)va,*ib=(int*)vb;
      return *ia-*ib;
}

/* triedime */
qsort(Data,Max,sizeof(int),int_cmp);
```

1.4.6. Hľadanie mediána

Medián postupnosti N prvkov je definovaný ako prvok, ktorý je menší alebo rovný ako jedna polovica všetkých prvkov a súčasne väčší alebo rovný ako druhá polovica všetkých prvkov. Napríklad mediánom postupnosti 16, 12, 99, 95, 18, 87, 10 je prvok 18. Metóda, ktorú si ukážeme sa zaoberá výberom k-tého najmenšieho prvku. Medián je potom špeciálnym prípadom pre $k = \frac{N}{2}$.

```
Select (A,l,r,k)
{
  if l<r then
  {
    i ← Rozdeľ(A,l,r)
   if i>l+k-1 then Select(A,l,i-1,k);
   if i<l+k-1 then Select(A,i+1,r,k-i);
  }
}
```

Táto procedúra preusporiada prvky poľa A tak, že prvky *A[1],...,A[k-1]* budú menšie alebo rovné ako *A[k]* a všetky prvky *A[k+1],...,A[r]* budú väčšie alebo rovné ako *A[k]*.

V priemernom prípade nájdeme týmto postupom k-ty najmenší prvok v čase $O(N)$. V najhoršom prípade, tento postup bude rovnako rýchly ako *Quicksort*. Pokiaľ použijeme základný postup na rozdeľovanie, na nájdenie najmenšieho prvku v utriedenom poli potrebujeme čas $O(N^2)$.