Algoritmizácia a programovanie
7. prednáška

Ján Grman

Obsah

1. Ukazovatele - pokračovanie
2. Algoritmy (vyhľadávanie, usporiadanie)

Referenčný operátor &

- Jedinou úlohou ukazovateľa je ukazovať na premenné určitého typu
- Každá premenná je uložená v pamäti na nejakej adrese
- Ak má ukazovateľ ukazovať na premennú, musíme poznať jej adresu
- Adresu ľubovoľnej premennej získame pomocou referenčného operátora **&**
- príkazom:

```
p = &i;
```

sme adresu premennej **i** uložili do premennej **p** – teda ukazovateľ **p** ukazuje na premennú **i**

Deferefenčný operátor *

- Hodnotu premennej na ktorú ukazovateľ ukazuje sprístupníme pomocou unárneho operátora *

***p**

Keď ukazovateľ neukazuje nikam

- Nulový ukazovateľ: **`NULL`**
- **`NULL`** - symbolická konštanta definovaná v **`stdio.h`**:
  - **`#define NULL 0`**
  - **`#define NULL ((void *) 0)`**
- Je možné priradiť ho ukazovateľom na ľubovoľný typ

```
if (p_i == NULL)
    ...
```

Prečo je potrebné určovať typ smerníka

- Všetky ukazovatele majú rovnakú veľkosť (sú to pamäťové miesta na uchovanie adresy)
- Prečo musí byť určený typ smerníka, keď všetky sú rovnako veľké?
- Ak nasmerujeme smerník na určitú adresu, vieme že na tejto adrese začína a ďalej spojito pokračuje cez toľko bajtov, koľko je potrebných na uloženie príslušného typu premennej.

Pointer na void

- Niekedy vznikne situácia, kedy potrebujeme použiť ukazovateľ, ale nemusíme vedieť na aký typ ukazuje
- Použijeme generický ukazovateľ

```
void *p;
```

- Môže ukazovať na ľubovoľný typ

Konverzia ukazovateľov

- Vyhnúť sa jej!
- Ak sa nedá vyhnúť – explicitne pretypovávať

```
int *p_i;
char *p_c;

p_c = p_i;
p_c = (char *)p_i;
```

nevhodné

vhodnejšie

Príklad ukazovateľa na typ void

```
int i;
float f;
void *p_void = &i;

*(int *) p_void = 2;
p_void = &f;
*(float *) p_void = 3.5;
```

p_void ukazuje na i

nastavenie i na 2

p_void ukazuje na f

nastavenie f na 3.5

- zastavme sa !
- pri priraďovaní je potrebné uviesť typ

Príklad: ukazovateľ na typ void

program vypíše načítané číslo
pričom použije ukazovateľ na int a
na void ako ukazovatele na celé
číslo

```
#include <stdio.h>

 int main() {
 int i, *p_int = &i;
   void *p_void = &i;

   printf("Zadajte cele cislo: ");
   scanf("%d", &i);

   printf("i: %d, p_int: %d, p_void: %d\n",
      i, *p_int, (*(int *)p_void));
   return 0;
}
```

Ukazovateľová aritmetika

- S ukazovateľmi sa dajú robiť niektoré aritmetické operácie:
  - Súčet ukazovateľa a celého čísla
  - Rozdiel ukazovateľa a celého čísla
  - Porovnávanie ukazovateľov rovnakého typu
  - Rozdiel dvoch ukazovateľov rovnakého typu
- Majú zmysel len v rámci bloku dynamicky vytvorenej pamäte (POLIA)
- Ostatné operácie nedávajú zmysel

Operátor sizeof

- Na vysvetlenie aritmetických operácií s ukazovateľmi potrebujeme operátor **`sizeof()`**:
  - zistí veľkosť dátového typu v Bytoch
  - vyhodnotí sa v čase prekladu (nezdržuje beh)

```
int i, *p_i;
i = sizeof(p_i);
```

počet Bytov potrebných na uloženie ukazovateľa na `int` – nevyužíva sa

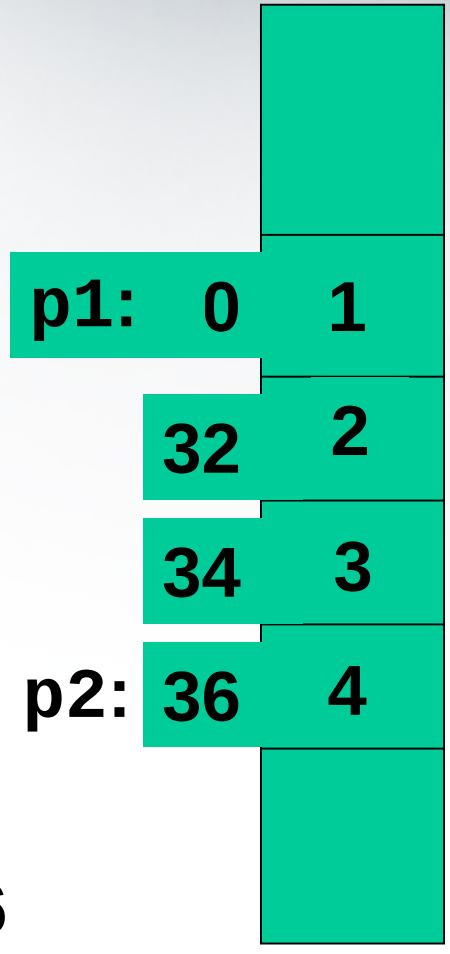
```
int i, *p_i;
i = sizeof(*p_i);
```

počet Bytov potrebných na uloženie typu `int` – využíva sa často

Súčet ukazovateľa a celého čísla

```
int n, *p1, *p2;
...
p2 = p1 + n;
```

(n=3)
sizeof(*p1)==2




```
p2 = (int *) p1 + sizeof(*p1)*n;
```

p2 = 30 + 2 * 3 = 36

Súčet ukazovateľa a celého čísla - príklady

```
char *p_c = 10;
int *p_i = 20;
float *p_f = 30;
```

Predpokladajme:

```
sizeof(char)  == 1
sizeof(int)   == 2
sizeof(float) == 4
```

Potom platí:

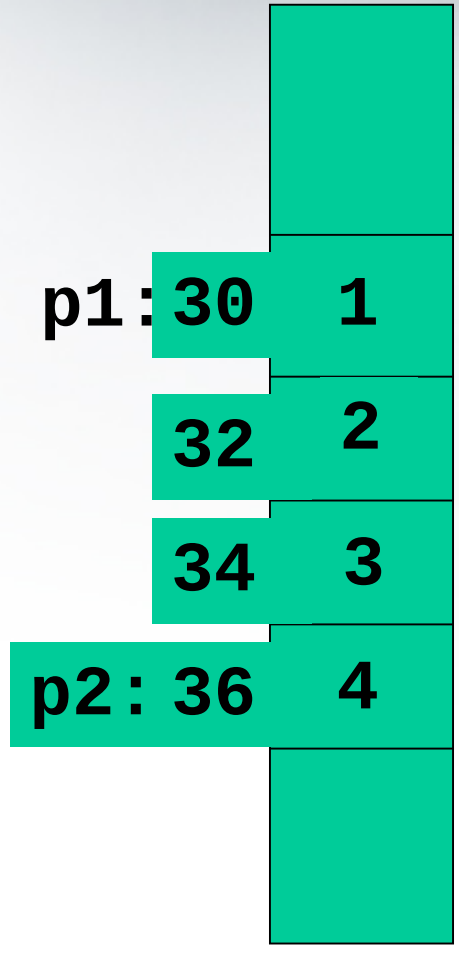
```
p_c + 1 == 11
p_i + 1 == 22
p_f + 1 == 34
```

POZOR ! Na tomto slajde ide o ADRESY

Rozdiel ukazovateľa a celého čísla

```
int n, *p1, *p2;
...
p1 = p2 - n;
```

(n=3)
sizeof(*p2)==2




```
p1 = (int *) p2 - sizeof(*p2)*n;
```

p2 = 36 - 2 * 3 = 30

Porovnávanie ukazovateľov

- operátory: **< <= > >= == !=**
- porovnávanie má zmysel len keď ukazovatele:
  - sú rovnakého typu
  - ukazujú na ten istý úsek pamäte
- výsledok porovnania:
  - ak je podmienka splnená: 1
  - inak: 0

Porovnávanie ukazovateľov: príklad - výpis reťazca

```
...
char *p1, *p2 , str[N];

strcpy(str, "ahoj");
p1 = str;
p2 = p1;
while(p2 < p1+ N && *p2 != '\0')
    printf("%c", *p2++);
```

`str`: pole s `N` znakmi,

`p1, p2`: ukazovatele

POZOR !
Vysvetliť krok po kroku

vyisuje znaky pokiaľ:

- nepresiahne dĺžku pridelenej pamäte premennej `str` a
- pokým nedosiahne koniec zapísaného slova

Porovnávanie ukazovateľov s konštantou NULL

- bez explicitného pretypovávania
- **`p = NULL`**:
  - neukazuje na žiadne zmysluplné miesto v pamäti

```
int n, *p;
...
if (n >= 0)
     p = alokuj(n);
else
    p = NULL;
...
if (p != NULL)
  ...
```

Rozdiel dvoch ukazovateľov rovnakého typu

```
int n, *p1, *p2;
...
n = p1 - p2;
```

POZOR !
Vysvetliť krok po kroku

```
n = ((int *) p1 - (int *) p2) / sizeof(*p1);
```

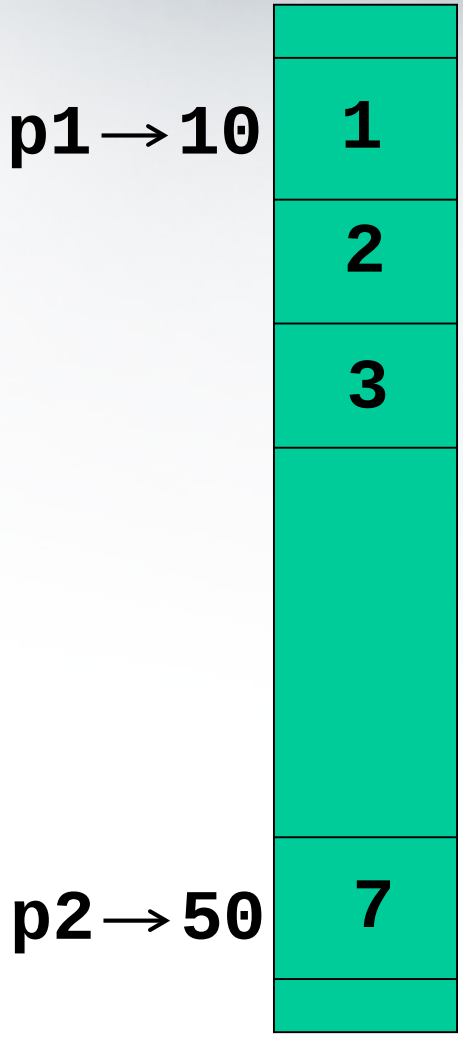
príklad:

ak je v bloku pamäte '?', vypíše pozíciu, inak -1

```
char *p1, *p2 , str[N];
p1 = str;
for (p2=p1; p2<p1+N && *p2 != '?'; p2++)
    ;
printf("%d", (p2 < p1+N) ? (p2-p1+1) :-1);
```

Ukazovateľová aritmetika

- aritmetické operácie:
  - Súčet ukazovateľa a celého čísla
  - Rozdiel ukazovateľa a celého čísla
  - Porovnávanie ukazovateľov rovnakého typu
  - Rozdiel dvoch ukazovateľov rovnakého typu
- majú zmysel len vtedy, keď:
  - sú ukazovatele na rovnaký typ
  - ukazujú na ten istý úsek pamäte (OS nezaručí, že neskôr alokovaný blok bude na vyššej adrese)



Algoritmy

Vyhľadávanie v usporiadanom poli

program načíta do poľa usporiadanú postupnosť čísel a hodnotu, ktorú chce v postupnosti (v poli) vyhľadať (nájsť jej index): použije *sekvenčné* a *binárne vyhľadávanie*

Sekvenčné vyhľadávanie

- najjednoduchšie vyhľadávanie:
  - od začiatku poľa postupne zväčšuje index pokým nepríde na hodnotu, ktorá je väčšia alebo rovná alebo pokým nepríde na koniec poľa
- neefektívne

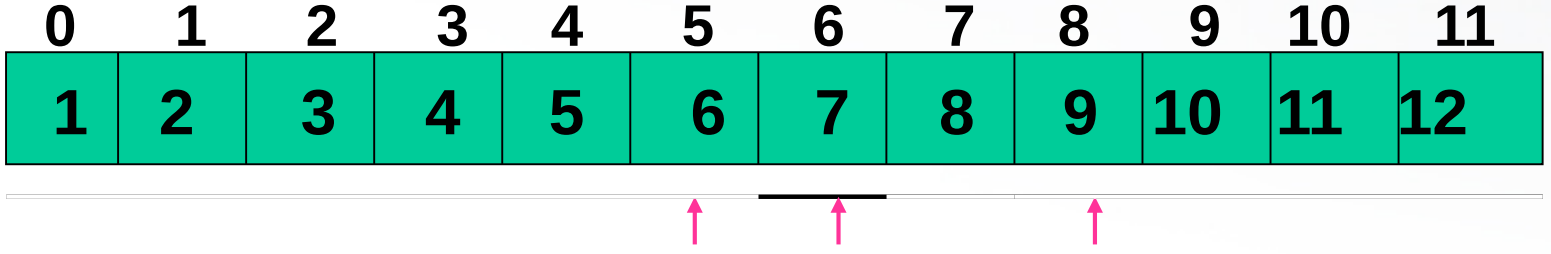
Sekvenčné vyhľadávanie

```
int sekvencne(int pole[], int n, int x)
{
    int i=0;

    while(i < n && pole[i] < x)
        i++;
    if(i<n && pole[i] == x)
        return i;
    return -1;
}
```

Binárne vyhľadávanie

- nájdenie stredu intervalu - ak je hľadaná hodnota menšia ako hodnota stredného prvku → hľadanie v ľavej polovici, inak v pravej polovici



hľadáme pozíciu hodnoty 7

```
int binarne(int pole[], int n, int x)
{
   int m, l = 0, r = n-1;

   while (l <= r) {
      m = (l + r) / 2;

      if (x == pole[m])
         return m;
      if (x < pole[m])
         r = m - 1;
      else
         l = m + 1;
   }

   if (pole[m] == x)
      return m;
   else
      return -1;
}
```

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

int sekvencne(int pole[], int n, int x);
int binarne(int pole[], int n, int x);

int main()
{
   int p[100], i, n, x, vysl;
   char c;

   printf("Zadaj pocet prvkov pola (<100): ");
   scanf("%d", &n);

   if (n >= 100) {
      printf("Prilis velky pocet prvkov...\n");
      return 1;
   }

   for (i=0; i<n; i++) {
      printf("p[%d]: ", i);
      scanf("%d", &p[i]);
   }
```

```
   printf("Zadajte hodnotu, ktora sa ma vyhladat: ");
   scanf("%d", &x);
   while(getchar() != '\n');
   printf("Vyhladavat Binarne alebo Sekvencne? [b/s]: ");
   c = getchar();

   printf("%c\n", c);
   if ((c = tolower(c)) == 's')
      vysl = sekvencne(p, n, x);
   else if (c == 'b')
      vysl = binarne(p, n, x);
   else {
      printf("Zadali ste nespravnu hodnotu.\n");
      return 1;
   }

   if (vysl > -1)
      printf("Index hladanej hodnoty je %d.\n",vysl);
   else
      printf("Hladana hodnota sa v poli nenachadza.\n");
   return 0;
}
```

Príklad: spojenie usporiadaných reťazcov

```
void merge(int A[], int m, int B[], int n) {

        while(m > 0 && n > 0){
            if(A[m-1] > B[n-1]){
                A[m+n-1] = A[m-1];
                m--;
            }else{
                A[m+n-1] = B[n-1];
                n--;
            }
        }

        while(n > 0){
            A[m+n-1] = B[n-1];
            n--;
        }
    }
```

Program spojí usporiadané polia celých čísel A a B do A tak, aby výsledok bol opäť usporiadaný (predpokladáme, že sa do A sa všetky hodnoty zmestia).

Agoritmy usporiadania

- usporiadanie poľa čísel (od najmenšieho po najväčšie)
- rôzne algoritmy (rôzne efektívne):
  - bublinkové usporadúvanie (BubbleSort)
  - usporadúvanie výberom (MaxSort)
  - triedenie vsúvaním (InsertSort)
  - (rýchle usporadúvanie (QuickSort))
  - ...

BubbleSort

- porovnávanie hodnôt dvoch susedných buniek poľa
  - do bunky s nižším indexom menšie z nich
  - do bunky s vyšším indexom väčšie z nich
- po jednom prechode poľom sa určite maximálny prvok dostane na koniec poľa, potom ostáva usporiadať N-1 prvkov poľa (posledný je už na svojom mieste)
- najmenej efektívne

BubbleSort: implementácia

```
void bubblesort(int a[], int n)
{
   int i,j;

   for (i = n; i > 1; i--)
      for (j = 1; j < i; j++)
         if (a[j-1] > a[j])
            vymen(&a[j-1], &a[j]);
}
```

v úseku 0...i "vybubláme" najväčší prvok nakoniec

```
void vymen(int *x, int *y)
{
   int pom = *x;
   *x = *y;
   *y = pom;
}
```

MaxSort

- nájdi v úseku maximálny prvok, vymeň ho s posledným prvkom, skráť usporiadavané pole o 1- pokým nie je jednoprvkové
- porovnanie s BubbleSort-om:
  - rovnako porovnávaní
  - menej výmen

MaxSort: implementácia

```
void maxsort(int a[], int n)
{
   int i, j, max;

   for (i = n-1; i > 0; i--) {
     max = 0;
     for (j = 1; j <= i; j++)
       if (a[j] > a[max])
           max = j;
     if (i != max)
       vymen(&a[max], &a[i])
   }
}
```

v úseku 0...i nájdeme maximum

ak maximum z úseku 0...i nie i-ty prvok, vymeníme ich

InsertSort

- časť poľa je usporiadaná a vsunie sa do nej prvok tak, aby pole zostalo usporiadané

InsertSort: implementácia

```
void insertsort(int a[], int n)
{
   int i, j, pom;

   for (i = 1; i < n; i++) {
      pom = a[i];
      j = i-1;
      while (j >= 0  && a[j] > pom) {
         a[j+1] = a[j--];
      }
      a[j+1] = pom;
   }
}
```

úsek 0..i-1 je usporiadaný, vsunieme a[i] tak, aby zostal usporiadaný

hľadanie vhodného miesta pre prvok a[i]

QuickSort

- rozdeľuj a panuj:
  - z poľa sa vyberie pivot (napr. prvý prvok poľa)
  - podľa pivota rozdelíme vstupné pole na tri časti obsahujúce
    1. čísla menšie a rovné ako pivot - tú ešte rozdelíme na čísla menšie ako (1.) pivot a (2.) pivot
    3. väčšie ako pivot
  - potom rovnakým spôsobom usporadúvame časť (1.) a časť (3.)
- rekurzia
- rýchle, oplatí sa pre veľké polia

```
void quickSort(int a[], int l, int r) {
   int j;

   if (l < r) {
      j = rozdel(a, l, r);
     quickSort(a, l, j-1);
     quickSort(a, j+1, r);
   }
}
```

```
int rozdel(int a[], int l, int r) {
   int i, j;
   int pivot = a[l];

   i = l; j = r+1;
   do {
      do ++i; while (a[i] <= pivot && i <= r);
      do --j; while (a[j] > pivot);
      if (i < j)
         vymen(&a[i], &a[j]);
   } while (i < j);
   vymen(&a[l], &a[j]);
   return j;
}
```

Príklad: vloženie prvku do poľa

Program načíta do poľa celé čísla a
vloží zadané číslo na zadanú pozíciu

```
#include<stdio.h>

   int main(){
   int a[50], velkost, cislo, i, poz;

   printf("Zadajte velkost pola: ");
   scanf("%d", &velkost);
   printf("Zadajte %d prvkov pola:\n", velkost);
   for(i=0; i<velkost; i++)
      scanf("%d", &a[i]);

   printf("Zadajte poziciu a cislo na vlozenie: ");
   scanf("%d %d", &poz, &cislo);

   i = velkost++;
   while(i > poz) {
      a[i] = a[i-1];
      i--;
   }
   a[i] = cislo;

   printf("Pole po vlozeni prvku:\n");
   for(i=0; i<velkost; i++)
      printf(" %d",a[i]);
   return 0;
}
```

Príklad: zmazanie prvku poľa

Program načíta do poľa celé čísla a vymaže číslo zo zadanej pozície

```
#include<stdio.h>

int main(){
   int a[50], i, poz, velkost;
   printf("Zadajte pocet prvkov pola (<=50): ");
   scanf("%d", &velkost);

   printf("Zadajte %d prvkov pola:\n", velkost);
   for(i=0; i<velkost; i++)
      scanf("%d", &a[i]);

   printf("Zadajte poziciu na vymazanie prvku: ");
   scanf("%d",&poz);

   i = poz;
   while(i < velkost-1){
      a[i] = a[i+1];
      i++;
   }

   velkost--;

   printf("Pole po vymazani prvku:\n");
   for(i=0; i<velkost; i++)
      printf("%d ",a[i]);

  return 0;
}
```

Hodnota polynómu v danom bode

program načíta koeficienty polynómu a hodnotu premennej x a vypočíta hodnotu polynómu v bode x.

Hodnota polynómu v danom bode

- Efektívny algoritmus znižujúci na minimum počet násobení je známy ako *Hornerova schéma*:

$$a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1} + a_nx^n =$$
$$a_0 + x\,(a_1 + x\,(a_2 + x\,(\ldots + x\,(a_{n-1} + a_nx)\ldots)))$$

- polynóm stupňa N tak môže byť vyhodnotený len s použitím:
  - N-1 operácií násobenia,
  - N operácií sčítania

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

float horner(float p[], int n, float x);
void nacitaj(float p[], int n);

int main()
{
   int n, i;
   float x, p[100];

   printf("Zadajte stupen polynomu: ");
   scanf("%d", &n);
   n++;

   if(n > 100) return 0;
   nacitaj(p, n);
   printf("Zadajte hodnotu premennej x: ");
   scanf("%f", &x);
   printf("hodnota polynomu je: %f\n", horner(p, n, x));
   return 0;
}
```

Hodnota polynómu v danom bode: Hornerova schéma

```
float horner(float p[], int n, float x)
{
    int i;
    float v = p[n-1];

    for (i = n-2; i >= 0; i--)
        v = x * v + p[i];

    return v;
}
```

```
void nacitaj(float p[], int n)
{
    int i;
    for (i=0; i<n; i++) {
        printf("p[%d]: ", i);
        scanf("%f", &p[i]);
    }
}
```

Príklady na doma

Mergesort

- Rozdelenie n-prvkového zoznamu na polovice (rekurzívne) – až na n 1-prvkových zoznamov
  - 1-prvkové zoznamy sú usporiadané
- Opakovanie spájania podzoznamov a vytváranie tak dlhších usporiadaných podzoznamov, až pokým nezostane 1 zoznam, ktorý je usporiadaný

Mergesort

Prekopírovanie obsahu poľa B do A od iMin po iMax-1

```
void CopyArray(int B[], int iMin, int iMax, A[]) {
    for(k = iMin; k < iMax; k++)
        A[k] = B[k];
}

void MergeSort(int A[], int B[], int n) {
    SplitMerge(A, 0, n, B);
}
```

Aby sa dal algoritmus volať rekurzívne pre rôzne podčasti poľa, potrebujeme aj dolný index (pre celé pole voláme s 0)

Mergesort

Pole od indexu iMin po iMax-1

```
void SplitMerge(int A[], int iMin, int iMax, int B[]) {
    if(iMax - iMin < 2)
        return;

    iStred = (iMax + iMin) / 2;
    SplitMerge(A, iMin,  iStred, B);
    SplitMerge(A, iMiddle,    iMax, B);
    Merge(A, iMin, iStred, iMax, B);
    CopyArray(B, iMin, iMax, A);
}
```

1-prvkové pole je usporiadané

Rekurzívna vetva:
- Rozdeľ na polovice
- Usporiadaj (Rozdeľ/spoj) ľavú polovicu
- Usporiadaj (Rozdeľ/spoj) pravú polovicu
- Spoj usporiadané polovice (do B)
- Prekopíruj usporiadané do A

Mergesort

- Spájanie (a zápis do poľa B)
  - Ľavej časti A - index i0: iMin – iStred-1
  - Pravej časti A – index i1: iStred – iMax-1

```
void Merge(int A[], int iMin, int iStred, int iMax, intB[])
{
    int i0, i1, j;
    i0 = iMin, i1 = iStred;

    for (j = iMin; j < iMax; j++) {
        if (i0 < iStred && (i1 >= iMax || A[i0] <= A[i1]))
            B[j] = A[i0];
            i0 = i0 + 1;
        else
            B[j] = A[i1];
            i1 = i1 + 1;
    }
}
```

Ak v ľavej časti ešte sú prvky a (buď v pravej časti už nie sú prvky, alebo v ľavej časti je menší alebo rovný prvok)

Príklad: nájdenie k-teho najmenšieho prvku poľa (Partial selection sort)

```
int select(int pole[], int n, int k) {
  int i, j, minIndex, minHod;

  for(i=0; i<k; i++) {
     minIndex = i;
     minHod = pole[i];
     for(j=i+1; j<=n; j++) {
        if(pole[j] < minHod) {
           minIndex = j;
           minHod = pole[j];
        }
     }
     vymen(&pole[i], &pole[minIndex]);
  return pole[k-1];
}
```

Binárne vyhľadávanie: rekurzívne

```
int binarneRek(int A[], int k, int iMin, int iMax)
{
   if (iMax < iMin)
      return -1;
   else {
      int iStred = (iMin + iMax)/2;
      if (A[iStred] > k)
         return binarneRek(A, k, iMin, iStred-1);
      else if (A[iStred] < key)
         return binarneRek(A, k, iStred+1, iMax);
      else
         return iStred;
    }
}
```

Príklad: kontrola sumy + ladenie

Program zistí, či prvé číslo v súbore **`suma.txt`** je súčtom čísel, ktoré sú za ním. Použitie základného a podrobného ladenia (príklad z cvičení).

```c
#include <stdio.h>
#include <stdlib.h>
#define LADENIE_ZAKLADNE
#define LADENIE_PODROBNE

int main() {
   FILE *fr;
   float cena, suma, sucet = 0.0;

   #if defined(LADENIE_ZAKLADNE) || defined(LADENIE_PODROBNE)
      printf("Otvorenie suboru\n");
   #endif

   if((fr = fopen("suma.txt", "r")) == NULL) {
      printf("Nepodarilo sa otvorit subor.\n");
      exit(1);
   }
   #ifdef LADENIE_PODROBNE
      printf("Subor otvoreny\n");
   #endif
```

```
#if defined(LADENIE_ZAKLADNE) || defined(LADENIE_PODROBNE)
   printf("Kontrola sumy\n");
#endif

fscanf(fr, "%f", &sucet);

while (fscanf(fr, "%f", &cena) != EOF) {
   suma += cena;
   #ifdef LADENIE_PODROBNE
      printf("Suma: %.2f\n", suma);
   #endif
}

if(suma == sucet)
   printf("Suma je spravna\n");
else
   printf("Suma je nespravna\n");

#ifdef LADENIE_PODROBNE
   printf("Suma skontrolovana\n");
#endif
```

```
    #if defined(LADENIE_ZAKLADNE) || defined(LADENIE_PODROBNE)
       printf("Zatvaranie suboru\n");
    #endif

    if(fclose(fr) == EOF) {
       printf("Nepodarilo sa zatvorit subor.\n");
       exit(1);
    }

    #ifdef LADENIE_PODROBNE
       printf("Subor zatvoreny\n");
    #endif

    return 0;
}
```