Informace k obhajobě 2. projektu, rekurze, vyhledávání a řazení

IZP-cv09

Ing. Jakub Husa Ph.D.

Vysoké Učení Technické v Brně, Fakulta informačních technologií
Božetěchova 1/2. 612 66 Brno - Královo Pole
ihusa@fit.vut.cz

22. listopadu 2024

Informace k obhajobě 2. projektu

Obhajoba 2. projektu

Hodnocení druhého projektu bude rozděleno na dvě části:

- Funkčnost – bude hodnocena automatickým skriptem od doktora Smrčky.
- Obhajoba – bude hodnocena cvičícím, a bude se konat 12. týden, místo cvičení.
- Cílem obhajoby je cvičícího během cca 5 minut přesvědčit že odevzdanému kódu rozumíte a prokázat tím, že jste jeho autorem.
- Pokud svoje autorství neprokážete, bude celý projekt hodnocen 0 body a může s vámi být zahájeno disciplinární řízení!

Faktory které budu při obhajobě hodnotit:

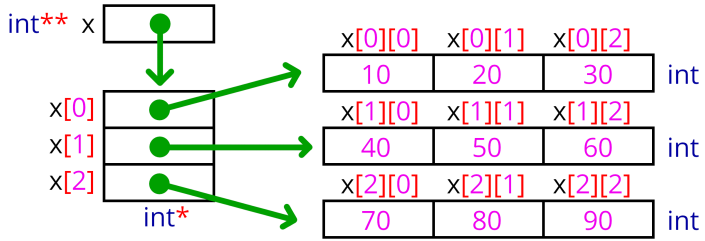
- Přehlednost (1 bod) – je kód přehledný a dobře zarovnaný? Dávají jména funkcí a proměnných smysl? Obsahuje kód pouze standardní znaky?
- Struktura (1 bod) – je kód dobře strukturovaný? Jsou funkce deklarovány? Je kód dobře dekomponován? Mají všechny funkce přijatelnou délku?
- Komentáře (1 bod) – Je u funkcí vysvětlen význam parametrů a návratové hodnoty? Jsou vysvětleny používané cykly, podmínky a magické konstanty?
- Vysvětlení (1 bod) – Dokážete sami shrnout jak váš program funguje?
- Odpovědi na otázky (3 body) – dokážete odpovědět na dotazy týkající se detailů vámi odevzdaného kódu?

Alokace 2D pole

Dvourozměrné (2D) pole je polem několika jednorozměrných (1D) polí:

- Každé z polí vyžaduje samostatné volání funkce malloc (a free).
- K prvkům pole přistupujeme dvojitými hranatými závorkami ([] []).

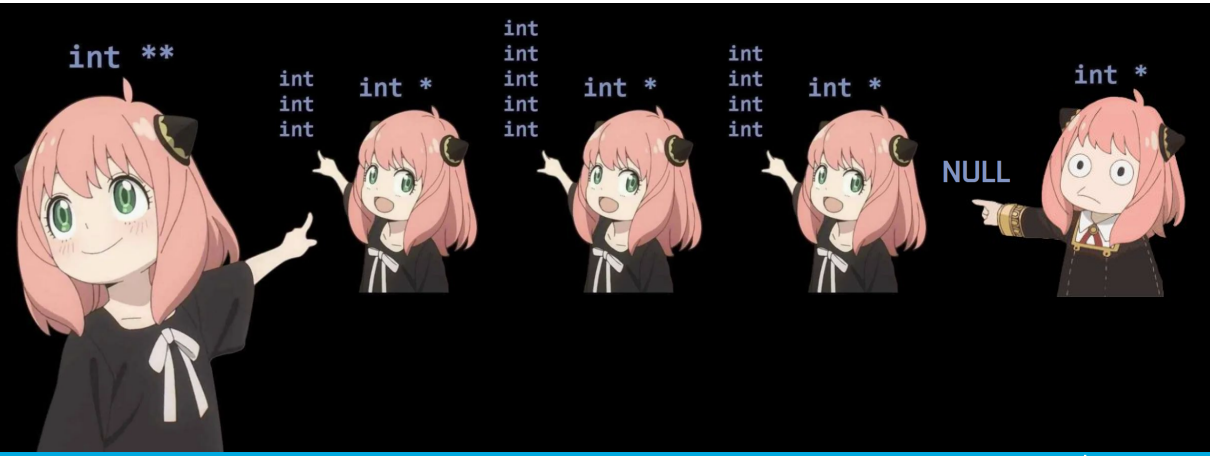
```
1  int rows = 3, cols = 3;                    //pocet radku a sloupcu pole
2  int** x = malloc(sizeof(int*) * rows);     //alokujeme 2D pole (radky)
3  for(int i = 0; i < rows; i++)              //pro kazdy radek
4      x[i] = malloc(sizeof(int) * cols);     //alokujeme 1D pole (sloupce)
5
6  for(int i = 0; i < rows; i++)              //pro kazdy radek
7      for(int j = 0; j < cols; j++)          //pro kazdy sloupece
8          x[i][j] = (i*cols+j)*10+10;        //prvkum pole pocitame hodnotu
```



Problémy s alokací 2D pole

Řádky dynamicky alokovaného pole nemusejí mít stejnou délku:

```
1  int** x = malloc(sizeof(int*) * 4); //2D pole ctyr 1D poli
2  x[0] = malloc(sizeof(int) * 3);     //1D pole tri  celych cisel
3  x[1] = malloc(sizeof(int) * 5);     //1D pole peti celych cisel
4  x[2] = malloc(sizeof(int) * 4);     //1D pole ctyr celych cisel
5  x[3] = malloc(sizeof(int) * -1);    //1D pole ktere se nepodari alokovat
```



Ošetření alokace 2D pole

Pokud kterákoliv z alokací selže, všechny předcházející alokace budeme muset uvolnit:

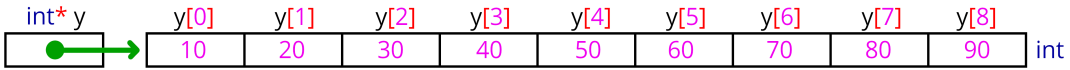
```
1  int rows = 3, cols = 3;                          //pocet radku a sloupcu pole
2
3  int** x = malloc(sizeof(int*) * rows);           //alokujeme 2D pole (radky)
4  if (x != NULL)                                   //pokud alokace neselhala
5  {
6      for(int i = 0; i < rows; i++)                //pro kazdy radek
7      {
8          x[i] = malloc(sizeof(int)*cols);         //alokujeme sloupce
9          if (x[i] == NULL)                        //pokud alokace selhala
10         {
11             for (int j = i-1; j >= 0; j--)       //pro vsechny predchozi radky
12             {
13                 free(x[j]);                      //uvolnujeme radek
14             }
15             free(x);                             //uvolnujeme 2D pole
16             break;                               //koncime cyklus alokace
17         }
18     }
19 }
```

Falešné 2D pole

2D můžeme nahradit dlouhým 1D polem se složitějším indexováním:

- K alokaci stačí jedno volání funkce malloc (a free).
- K prvkům pole přistupujeme jedněmi hranatými závorkami ([]).
- Jako kombinovaný index používáme index řádku * počet sloupců + index sloupce.

```
1  int rows = 3, cols = 3;                    //pocet radku a sloupcu pole
2  int* y = malloc(sizeof(int)*rows*cols);    //alokujeme jedno velke 1D pole
3
4  for(int i = 0; i < rows; i++)              //pro kazdy radek
5      for(int j = 0; j < cols; j++)          //pro kazdy sloupec
6          y[i*cols+j]=(i*cols+j)*10+10;      //prvkum pole pocitame hodnotu
```



Rekurze

Rekurze

Rekurze je programovací technika při které funkce volají sebe sama:

- Používáme ji pro řešení problémů které lze jednoduše rozdělit na podproblémy.

Například – výpočet faktoriálu definovaného rekurzivní rovnicí:

$$0! = 1 \qquad n! = n * (n-1)!$$

```
1  int rekurzeFaktorial(int n)             //vypocet pomoci rekurze
2  {
3      if (n == 0)                         //pokud pocitame faktorial nuly
4          return 1;                       //vysledek je jedna
5      return n * rekurzeFaktorial(n-1); //jinak vracime rekuzi
6  }
7
8  int cyklusFaktorial(int n)              //vypocet pomoci cyklu
9  {
10     int x = 1;                          //pocatecicni hodonota produktu
11     for (int i = n; i > 0; i--)         //pro vsechan cisla od n do 1
12         x = x * i;                      //produk nasobime cislem
13     return x;                           //vracime vysledny produkt
14 }
```

Ošetření rekurze

Při každém volání se na zásobník ukládá nová kopie parametrů funkce:

- Rekurzivní volání vždy musí mít nějakou ukončovací podmínku.
- Nekonečná rekurze vždy havaruje na nedostatek paměti.

```
1  void foo(int n)          //rekurzivni funkce s ukoncujici podminkou
2  {
3      printf("%i\n", n); //funkce vypisuje hodnotu "N"
4      if (n > 0)          //pokud N neni nula (ukoncujici podminka)
5          foo(n-1);       //funcke vola sebe sama pro "N-1"
6      printf("%i\n", n); //funkce vypisuje hodnotu "N"
7  }
8
9  void bar(int n)          //rekurzivni funkce bez ukoncujici podminky
10 {
11     printf("%i\n", n); //funkce vypisuje hodnotu "N"
12     bar(n+1);           //CHYBA -- po cca 43000 volanich rekuze havaruje
13     printf("%i\n", n); //a tento radek se nikdy neprovede
14 }
```

Každou rekurzi lze převést na cyklus, a každý cyklus lze převést na rekurzi:

```c
void vypisCyklem(char str[])  //vypis znaku retezce pomoci cyklu
{
    for (int i = 0; str[i] != '\0'; i++) //dokud znak na indexu i neni NUL
        printf("%c", str[i]);            //vypis znak z indexu i
}

void vypisRekurzi(char str[]) //vypis znaku retezce pomoci rekurze
{
    if (str[0] == '\0')       //pokud je retezec prazdny
        return;               //rekurze konci
    printf("%c", str[0]);     //vytiskni prvni znak retezce
    vypisRekurzi(&str[1]);    //vytiskni zbytek retezce
}

int main()                    //zacatek programu
{
    vypisCyklem("Ahoj\n");    //vypis znaku retezce pomoci cyklu
    vypisRekurzi("Ahoj\n");   //vypis znaku retezce pomoci rekurze
    return 0;                 //konec programu
}
```

Vyzkoušejte si:

- Napište funkce které pomocí rekurze a cyklu spočítají hodnotu n-tého členu Fibonacciho posloupnosti definované rekurzivní rovnicí:

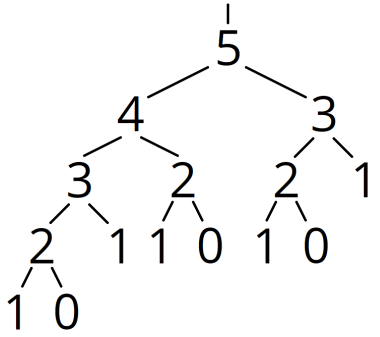
$$\text{fib}(0) = 0 \qquad \text{fib}(1) = 1 \qquad \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$$

```
1 int rekurzeFibonacci(int n);
2 int cyklusFibonacci(int n);
```

Například:

- Rekurze f(5) = 5
  Rekurze f(10) = 55
  Rekurze f(15) = 610
- Cyklus f(5) = 5
  Cyklus f(10) = 55
  Cyklus f(15) = 610

Strom rekurzivních volání a cyklus výpočtu:



0 + 1= 1
1 + 1= 2
1 + 2= 3
2 + 3= 5
3 + 5= 8
5 + 8=13

```c
int rekurzeFibonacci(int n);
int cyklusFibonacci(int n);

int main()        //zacatek programu
{
    int a = 5;    //paty      clen
    int b = 10;   //desaty    clen
    int c = 15;   //patnacty  clen

    //volani vypoctu pomoci rekurze
    printf("Rekurze f(%2i) = %3i\n", a, rekurzeFibonacci(a));
    printf("Rekurze f(%2i) = %3i\n", b, rekurzeFibonacci(b));
    printf("Rekurze f(%2i) = %3i\n", c, rekurzeFibonacci(c));
    //volani vypoctu pomoci cyklu
    printf("Cyklus  f(%2i) = %3i\n", a, cyklusFibonacci(a));
    printf("Cyklus  f(%2i) = %3i\n", b, cyklusFibonacci(b));
    printf("Cyklus  f(%2i) = %3i\n", c, cyklusFibonacci(c));

    return 0;     //konec programu
}
```

Vyhledávání

Sekvenční vyhledávání

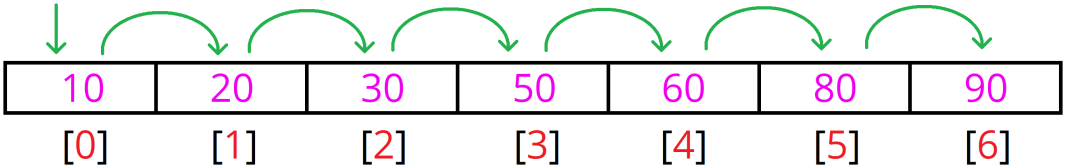
U algoritmů nás zajímá jejich časová (nebo paměťová) asymptotická složitost:

- Udává maximální počet kroků potřebných pro zpracování n položek.
- Více se o složitosti algoritmů budete učit v předmětu IAL (3. semestr).

Například – sekvenční vyhledávání prvku v poli má lineární časovou složitost O(n):

- Algoritmus projde přes všechny prvky pole a porovná je s hledanou hodnotou.
- Prohledat 1 000 000 prvků zabere 1 000 000 kroků.

```
1  int sekvencniVyhledavani(int delka, int pole[], int cislo)
2  {
3      for(int i = 0; i < delka; i++) //prochazime vsemi prvky pole
4          if (cislo == pole[i])      //pokud najdeme hledany prvek
5              return i;              //vratime jeho index
6      return -1;                     //pokud jsme prvek nenasli vracime -1
7  }
```



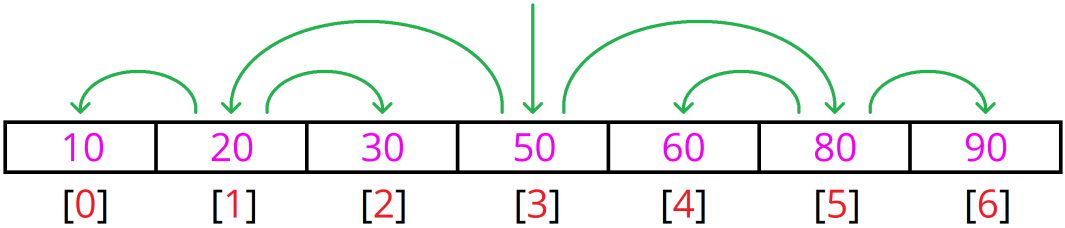
Binární vyhledávání

Pokud je pole seřazené můžeme použít algoritmus binárního vyhledávání:

- Binární vyhledávání začíná prostředním prvkem pole.
- Pokud je pole prázdné, tak prvek v poli není, a vrátíme -1.
- Pokud jsme hledaný prvek našli, tak vrátíme jeho index.
- Pokud jsme hledaný prvek nenašli, tak hledání rekurzivně opakujeme v levé nebo v pravé polovině zbytku pole.

Binární vyhledávání má logaritmickou časovou složitost $O(\log(n))$:

- Prohledat 1 000 000 prvků zabere pouze 20 kroků!



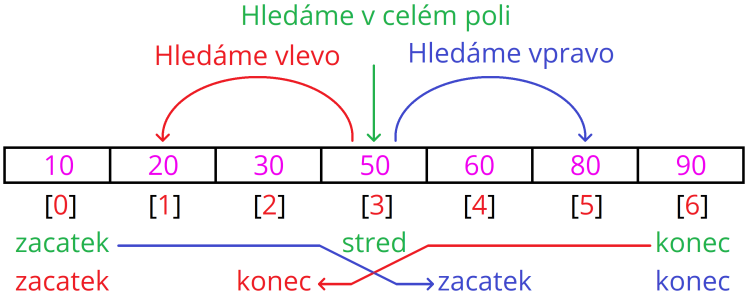
Vyzkoušejte si:

- Napište funkci která pomocí rekurze implementuje binární vyhledávání, a vrátí index hledaného čísla, nebo hodnotu -1 pokud pole dané číslo neobsahuje.
- Funkce jako parametry dostává: index začátku prohledávané části pole, index konce prohledávané části pole, pole ve kterém vyhledává a hledané číslo.

```
1 int binarniVyhledavani(int zacatek, int konec, int pole[], int cislo);
```

Například:

- Cislo 20 je na indexu 1
  Cislo 60 je na indexu 4
  Cislo 70 je na indexu -1



```c
int binarniVyhledavani(int zacatek, int konec, int pole[], int cislo);

int main()            //zacatek programu
{
    int d = 20;       //prvni hledane cislo
    int e = 60;       //druhe hledane cislo
    int f = 70;       //treti hledane cislo
    int arr[7] = {10,20,30,50,60,80,90}; //prohledavane pole (serazene)

    //volani hledani pomoci rekurze
    printf("Cislo %i je na indexu %i\n",d, binarniVyhledavani(0,6,arr,d));
    printf("Cislo %i je na indexu %i\n",e, binarniVyhledavani(0,6,arr,e));
    printf("Cislo %i je na indexu %i\n",f, binarniVyhledavani(0,6,arr,f));
    return 0;         //konec programu
}
```

Řazení

Řazení

Pro řazení prvků existuje mnoho algoritmů s různými vlastnostmi:

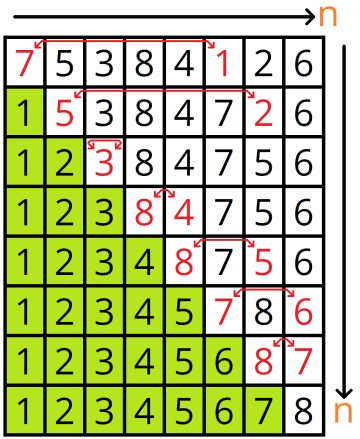
- Pomalé řadící algoritmy (Bubble-sort, Select-sort, Insert-sort, ...) používají dvojitý cyklus, mají kvadratickou časovou složitost $O(n^2)$, a seřadit 1000 prvků jim zabere 1 000 000 kroků.
- Rychlé řadící algoritmy (Quick-sort, Merge-sort, Heap-sort, ...) používají rekurzi a cyklus, mají linearitmickou časovou složitost $O(n * log(n))$, a seřadit 1000 prvků jim zabere jen 10 000 kroků.

Například – implementace algoritmu bublinkového řazení (Bubble-sort):

```
1  void bubbleSort(int delka, int pole[])    //bublinkove razeni
2  {
3      for(int i = 0; i < delka; i++)        //pro kazdy prvek
4          for(int j = 0; j < delka-1; j++) //projdi vsechny prvky
5              if(pole[j] > pole[j+1])       //pokud jsou dva sousedni
6              {                             //prvky ve spatnem poradi
7                  int temp = pole[j];       //vymen jejich hodnoty
8                  pole[j] = pole[j+1];
9                  pole[j+1] = temp;
10             }
11 }
```

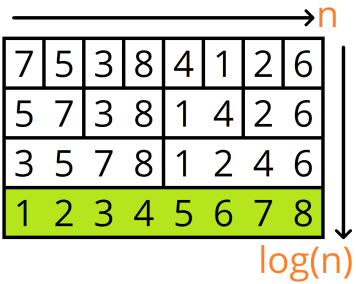
Řazení výběrem (Select-sort) n-krát zopakuje:

- V neseřazené části pole najdi minimum.
- Vyměň ho s prvním prvkem v neseřazené části pole.



Řazení slučováním (Merge-sort) problém seřazení celého pole rekurzivně rozdělí na "seřazení a sloučení levé a pravé poloviny":

- Pole o jednom prvku je vždy seřazené.
- Porovnej první prvky dvou sousedních polí a menší z nich přesuň do sloučeného pole.
- Opakuj dokud obě pole nejsou prázdná.
- Vyber další dvě pole, a postup opakuj dokud nejsou spojeny všechny prvky.



Vyzkoušejte si:

- Implementujte následující funkce pro práci s datovým typem osoba.

```
1 int porovnej(osoba A, osoba B);
2 void vymen(osoba* A, osoba* B);
3 int nejmensi(int delka, osoba pole[]);
4 void serad(int delka, osoba pole[]);
```

- porovnej – Porovná jména dvou osob a pokud jsou stejná tak porovná jejich věk.
  Pokud je jméno první osoby abecedně menší než druhé osoby, vrátí záporné číslo.
  Pokud je jméno první osoby abecedně větší než druhé osoby, vrátí kladné číslo.
  Pokud jsou jména stejná a první osoba je mladší, vrátí záporné číslo.
  Pokud jsou jména stejná a první osoba je starší , vrátí kladné číslo.
  Pokud jsou jména stejná a věk je stejný, vrátí nulu.
- vymen – vzájemně vymění dvě osoby předané odkazem.
- nejmensi – vrátí index osoby s nejmenší hodnotou (dle funkce porovnej).
- serad – pole osob seřadí od nejmenší po největší algoritmem Select-sort.

Na procvičení (2/2)

Například:

```
(Bob 23) (Cecil 23) (Bob 22) (Alice 24) (Daniela 21)
Ruzna jmena     (0,1) OK
Ruzny vek       (0,2) OK
(Daniela 21) (Cecil 23) (Bob 22) (Alice 24) (Bob 23)
Vymena   osob   (0,4) OK
Nejmensi osoba (3)    OK
(Alice 24) (Bob 22) (Bob 23) (Cecil 23) (Daniela 21)
```

```c
typedef struct Sosoba {                                          //struktura "Sosoba"
    char jmeno[10];                                              //pole znaku "jmeno"
    int vek;                                                     //cele cislo "vek"
} osoba;                                                         //datovy typ "osoba"

void vypis(int delka, osoba pole[]);                             //deklarace funkce "vypis"

int main()                                                       //zacatek programu
{                                                                //neserazene pole sosob
    osoba pole[5] ={{"Bob", 23},{"Cecil", 23},{"Bob", 22},{"Alice", 24},{"Daniela",21}};
    vypis(5, pole);                                              //vypis pole
    if (porovnej(pole[0], pole[1])<0) printf("Ruzna jmena (0,1) OK\n");   //test funkce porovnej
    if (porovnej(pole[0], pole[2])>0) printf("Ruzny vek (0,2) OK\n");     //test funkce porovnej
    vymen(&pole[0], &pole[4]);                                   //vymena dvou osob
    vypis(5, pole);                                              //vypis pole
    if (porovnej(pole[0], pole[4])>0) printf("Vymena osob (0,4) OK\n");   //test funkce vymen
    if (nejmensi(5, pole) ==3)        printf("Nejmensi osoba (3) OK\n");  //test funkce nejmensi
    serad(5, pole);                                              //test funkce serad
    vypis(5, pole);                                              //vypis pole
    return 0;                                                    //konec programu
}

void vypis(int delka, osoba pole[])                              //definice funkce "vypis"
{
    for(int i =0; i  < delka; i++)                               //pro vsechny prvky pole
        printf("(%s %i)  ", pole[i].jmeno, pole[i].vek);         //vypis jmeno a vek
    printf("\n");                                                //vypis konec radku
}
```