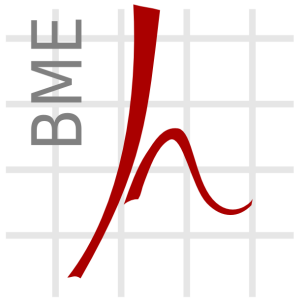
Generikus algoritmusok. Bitszintű operátorok

A programozás alapjai I.



Hálózati Rendszerek és Szolgáltatások Tanszék
Farkas Balázs, Fiala Péter, Vitéz András, Zsóka Zoltán

2020. november 30.

Tartalom

1. fejezet

Generikus algoritmusok

Motiváció

Rendezzünk 2D pontokat buborékalgoritmussal!

```
1  typedef struct { double x, y; } point;
```

```
1  void swap(point *px, point *py)
2  {
3    point tmp = *px;
4    *px = *py;
5    *py = tmp;
6  }
```

x koordinátájuk szerint növekvő (ascending) sorrendbe

```
1  void bubble_point_by_x_asc(point t[], int n)
2  {
3    int iter, i;
4    for (iter = 0; iter < n-1; ++iter)
5      for (i = 0; i < n-iter-1; ++i)
6        if (t[i].x > t[i+1].x)
7          swap(t+i, t+i+1);
8  }
```

Motiváció

Határ a csillagos ég...

```
1  void bubble_point_by_x_asc(point t[], int n);
2  void bubble_point_by_x_desc(point t[], int n);
3  void bubble_point_by_y_asc(point t[], int n);
4  void bubble_point_by_y_desc(point t[], int n);
5  void bubble_point_by_abs_asc(point t[], int n);
6  void bubble_point_by_abs_desc(point t[], int n);
7  void bubble_point_by_angle_asc(point t[], int n);
8  void bubble_point_by_angle_desc(point t[], int n);
```

...és ezek még csak a 2D pontok ...

- Írjuk meg a buborékalgoritmust függetlenül a rendezendő adatoktól és az összehasonlítási szemponttól!
- Generikus algoritmus.

Analízis

Mi a rendezés?

- Olyan algoritmus, amely
  - összehasonlításokból és
  - cserékből épül fel
- Ezek a rendező algoritmusok primitívjei.
- A primitívek dolgoznak közvetlenül az adatokkal, nekik kell ismerniük az adatok típusát
- A rendező algoritmus a primitívek hívási sorrendjét határozza meg, független az adattól.

Generikus algoritmus: I. lépés:

- Emeljük ki függvényként a primitíveket!
  - A cserével már megtettük (`swap` függvény)

Generikus rendezés

Emeljük ki az összehasonlítást külön függvénybe!

```
1  int comp_x_asc(point *a, point *b)
2  {
3    return a->x > b->x;
4  }
```

```
1  void bubble_point_by_x_asc(point t[], int n)
2  {
3    int iter, i;
4    for (iter = 0; iter < n-1; ++iter)
5      for (i = 0; i < n-iter-1; ++i)
6        if (comp_x_asc(t+i, t+i+1))
7          swap(t+i, t+i+1);
8  }
```

Ezzel még nem spóroltunk meg semmit, a különböző primitíveket különböző rendező függvények hívják.

Generikus rendezés

Minden összehasonlító primitív azonos típusú:

```
int comp_by_???(point *a, point *b);
```

Definiáljunk ilyen függvényekre mutató pointer típust

```
typedef int (*comp_fp)(point*, point*);
```

Adjuk át az összehasonlító primitívet is paraméterként

```
void bubble_point(point t[], int n, comp_fp comp)
{
  int iter, i;
  for (iter = 0; iter < n-1; ++iter)
    for (i = 0; i < n-iter-1; ++i)
      if (comp(t+i, t+i+1))
        swap(t+i, t+i+1);
}
```

A hívásnál választjuk ki az aktuális primitívet

```
bubble_point(points, 8, comp_x_asc);
```

Generikus rendezés

- Minden rendezési szemponthoz meg kell írnunk a két pontot összehasonlító primitívet
- Az egyszer megírt buborékrendező függvény paraméterként kapja az összehasonlítási elvet is

- Tud a `bubble_point` függvény macskákat rendezni életkor szerint?
- Sajnos, még nem.
- De majd mindjárt igen!

Generikus rendezés

Definiáljuk át a primitívek paraméterezését

```
int comp_by_???(point *array, int i, int j) { ... }
void swap_point(point *array, int i, int j) { ... }
```

A megfelelő függvénymutató-típusok:

```
typedef int (*comp_fp)(point*, int, int);
typedef void (*swap_fp)(point*, int, int);
```

Adjuk át a cserélő primitívet is paraméterként

```
void bubble_point(point *t, int n,
                  comp_fp comp, swap_fp xch) {
  int iter, i;
  for (iter = 0; iter < n-1; ++iter)
    for (i = 0; i < n-iter-1; ++i)
      if (comp(t,i,i+1))/* átkerült a ptraritmetika */
        xch(t, i, i+1);  /* innen is! */
}
```

Generikus rendezés

A mutatóaritmetika a bubble_point függvényből átkerült a primitívekbe!

Nem kell tudnia a tömbelemek méretét, csak a tömb kezdőcímét!

A kezdőcímet adjuk át void *-ként!

```
1  void bubble(void *t,int n, comp_fp comp,swap_fp xch) {
2    int iter, i;
3    for (iter = 0; iter < n-1; ++iter)
4      for (i = 0; i < n-iter-1; ++i)
5        if (comp(t, i, i+1))
6          xch(t, i, i+1);
7  }
```

A bubble függvény már nem tudja, hogy pontokat vagy macskákat rendez. Ekkor a primitívek is csak void *-ként kaphatják meg a tömböt. A megfelelő függvénymutató-típusok:

```
1  typedef int (*comp_fp)(void*, int, int);
2  typedef void (*swap_fp)(void*, int, int);
```

Generikus rendezés

A primitívek tudják, hogy milyen típusú adatokon dolgoznak
A `void *` kezdőcímet kényszerített típuskonverzióval átértelmezik

```
1  int comp_cat_by_age_asc(void *t, int i, int j)
2  {
3    cat *c = (cat *)t; /* mutatókonverzió */
4    return c[i].age > c[j].age;
5  }
```

```
1  void swap_cat(void *t, int i, int j)
2  {
3    cat *c = (cat *)t; /* mutatókonverzió */
4    cat tmp = c[i];
5    c[i] = c[j];
6    c[j] = tmp;
7  }
```

link

A hívás immár teljesen általános

```
1  bubble(cats, 8, comp_cat_by_age_asc, swap_cat);
2  bubble(dogs, 24, comp_dog_by_name_desc, swap_dog);
```

Összefoglalás

Generikus vektoralgoritmus

- Az algoritmust megvalósító függvény a tömböt `void *`-ként deklarált kezdőcímmel kapja meg
- Az általános algoritmus nem indexel, nem végez mutatóaritmetikát, csak a tömbindexekkel játszik
- A specializált primitívek `void *`-ként kapják meg a tömböt, és kényszerített mutatókonverzió után dolgoznak rajta.

További egyszerűsítés

- A cserélő primitív bájtonként cserél, nem is kell megírnunk minden típusra, elég az elemméretet átadnunk
- Gyorsrendező függvény az <stdlib.h>-ban

```
1 void qsort(void *t, size_t n, size_t elem_size,
2            int (*comp)(void*, void*));
```

Megjegyzés

- A `void *`-os pointerkonverzió „már-már durva hekkelés” kategóriába tartozik
- Ez is lefut figyelmeztetés nélkül:

```
1 Dalmatian doggies[101]; /* 101 kiskutya */
2 bubble(doggies, 101, comp_train_by_length,
3                      swap_city);
```

- A határokat feszegetjük, nagyon kell figyelnünk!
- Jövő félévben sokkal szebb módszert tanulunk, csak más nyelven

2. fejezet

Bitszintű operátorok

Bitszintű operátorok

művelet	szintaxis
bitenkénti negálás	`~ <egész kif.>`
léptetés (shift)	`<egész kif.> >> <nemneg. egész kif.>`
	`<egész kif.> << <nemneg. egész kif.>`

```
1  unsigned char a = 92; /* 0 1 0 1 1 1 0 0 */
2  unsigned char c;
3  c = ~a;               /* 1 0 1 0 0 0 1 1 */
4  c = a>>2;             /* 0 0 0 1 0 1 1 1 */
5  c = a<<3;             /* 1 1 1 0 0 0 0 0 */
```

Bitszintű operátorok

művelet	szintaxis
bitenkénti és	<egész kif.> & <egész kif.>
bitenkénti vagy	<egész kif.> | <egész kif.>
bitenkénti kizáró vagy	<egész kif.> ^ <egész kif.>

```
1  unsigned char a = 92;  /* 0 1 0 1 1 1 0 0 */
2  unsigned char b = 233; /* 1 1 1 0 1 0 0 1 */
3  unsigned char c;
4  c = a&b;               /* 0 1 0 0 1 0 0 0 */
5  c = a|b;               /* 1 1 1 1 1 1 0 1 */
6  c = a^b;               /* 1 0 1 1 0 1 0 1 */
```

Alkalmazások

Kifejezés, mely igaz, ha a b bájt 3. bitje be van állítva

```
b & (1 << 3)
/* b:           0 1 0 0 1 0 1 1          */
/* (1 << 3):    0 0 0 0 1 0 0 0    maszk */
```

Ha b 3. bitje 1, akkor a kifejezés értéke maga a maszk, vagyis IGAZ.

328 BCD-ben (binary coded decimal)

```
bcd = (3 << 8) | (2 << 4) | 8;
/* 0011 0010 1000 */
```

Visszaalakítás[1]:

```
value =        bcd      & 0xF  + /* 0xF a maszk */
         10*((bcd>>4) & 0xF) +
        100*((bcd>>8) & 0xF);
```

[1] intenzív zárójelezés jó szokás

Értékadó operátorok

művelet	szintaxis
viszonyított értékadás	`<egész balérték> &=<egész kif.>`
	`<egész balérték> |=<egész kif.>`
	`<egész balérték> ^=<egész kif.>`
	`<egész balérték> >>=<egész kif.>`
	`<egész balérték> <<=<egész kif.>`

Maszkolás – folytatás

Kifejezés, mely beállítja a `b` bájt 3. bitjét

```
1  b |= (1 << 3)
2  /* b (előtte): 0 1 0 0 0 0 1 1          */
3  /* (1 << 3):   0 0 0 0 1 0 0 0    maszk */
4  /* b (utána):  0 1 0 0 1 0 1 1          */
```

Kifejezés, mely törli a `b` bájt 3. bitjét

```
1  b &= ~(1 << 3)
2  /* b (előtte): 0 1 0 0 1 0 1 1                  */
3  /* (1 << 3):   0 0 0 0 1 0 0 0    maszk        */
4  /* ~(1 << 3):  1 1 1 1 0 1 1 1    inverz maszk */
5  /* b (utána):  0 1 0 0 0 0 1 1                  */
```

Kifejezés, mely megfordítja a `b` bájt 3. bitjét

```
1  b ^= (1 << 3)
2  /* b (előtte): 0 1 0 0 0 0 1 1                  */
3  /* (1 << 3):   0 0 0 0 1 0 0 0    maszk        */
4  /* b (utána):  0 1 0 0 1 0 1 1                  */
```

Köszönöm a figyelmet.