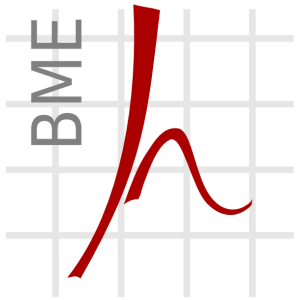
Rendezés. Generikus algoritmusok

A programozás alapjai I.



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

2021. november 30.

Tartalom

1. fejezet

Rendezés

Rendezés

Rendezni érdemes . . .

- . . . mert rendezett N elemű tömbben $\log_2 N$ lépésben megtalálunk egy elemet (vagy megtudjuk, hogy nincs benne)
- . . . mert rendezett N elemű listában $N/2$ lépésben megtalálunk egy elemet (vagy megtudjuk, hogy nincs benne)

Rendezni költséges . . .

- . . . de tipikus, hogy ritkán rendezünk, és rengetegszer keresünk

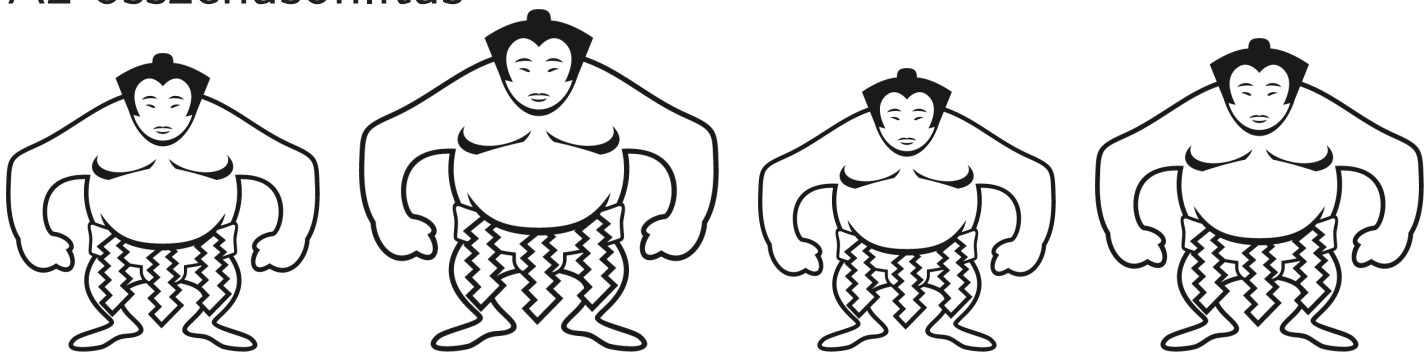
Mibe kerül a rendezés? . . .

$=$ összehasonlítások száma $\times$ egy összehasonlítás költsége

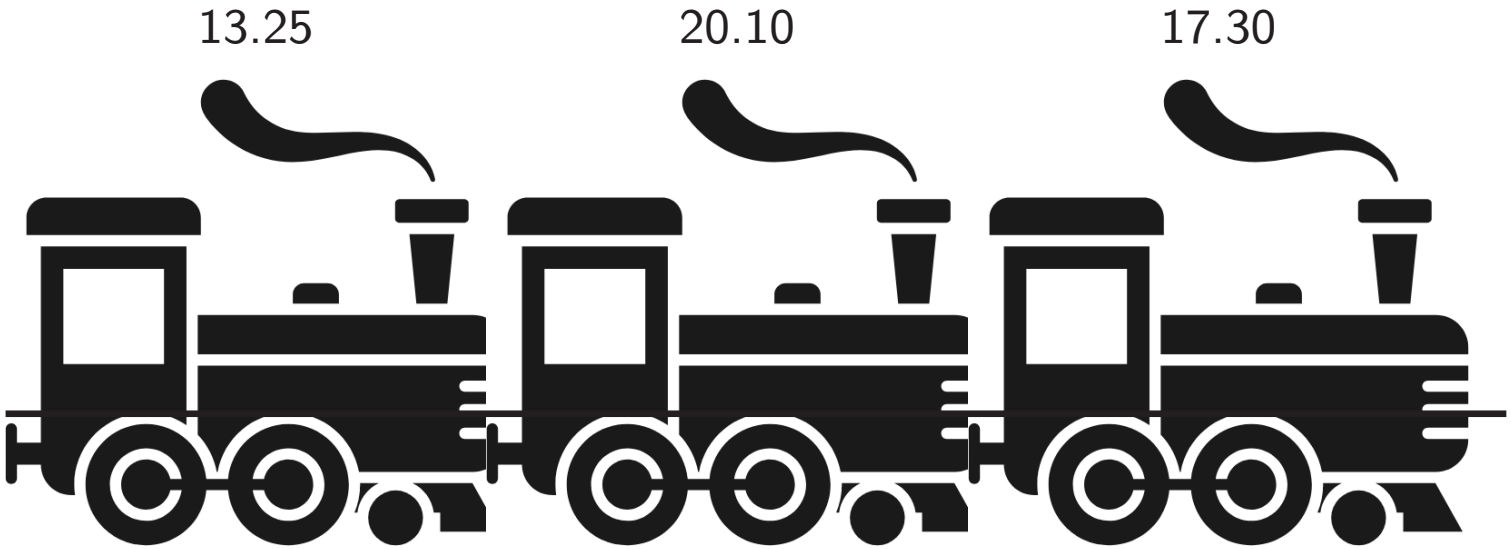
$+$ mozgatások (cserék) száma $\times$ egy mozgatás költsége

Mi kerül sokba?

- Az összehasonlítás



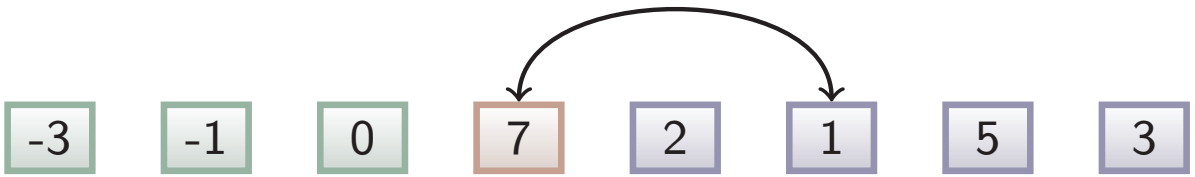
- A mozgatás



Nincs legjobb rendező módszer

Rendezés közvetlen kiválasztással

```
Cseréld ki a 0. elemmel a tömb minimumát
Cseréld ki az 1. elemmel az utolsó N-1 elem minimumát
Cseréld ki a 2. elemmel az utolsó N-2 elem minimumát
...
Cseréld ki az N-2. elemmel az utolsó 2 elem minimumát
```




```
MINDEN i-re 0-tól N-2-ig
  iMin ← i
  MINDEN j-re i+1-től N-1-ig
    HA t[j] < t[iMin]
      iMin ← j;
  t[i] ↔ t[iMin];
```

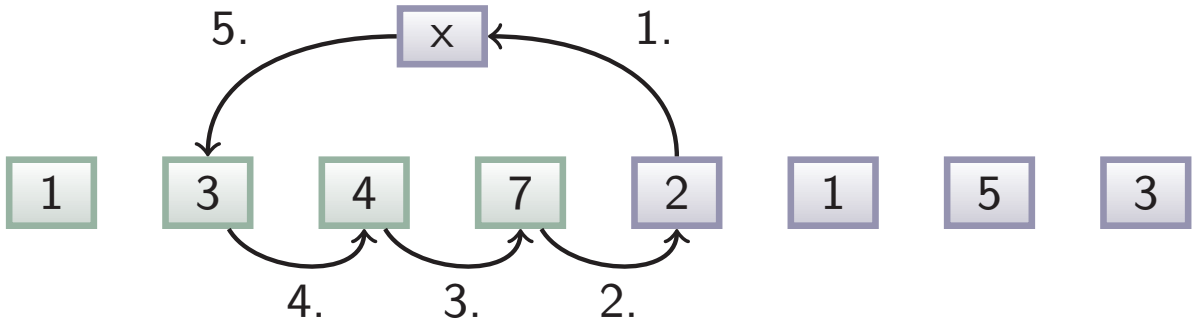
```
1  for (i=0; i<N-1; ++i) {
2    iMin = i;
3    for (j=i+1; j<N; ++j)
4      if (t[j] < t[iMin])
5        iMin = j;
6    swap(t+i, t+iMin);
7  }
```

Összehasonlítások száma: $\mathcal{O}(N^2)$ $\approx N^2/2$

Cserék száma: $\mathcal{O}(N)$ $N-1$

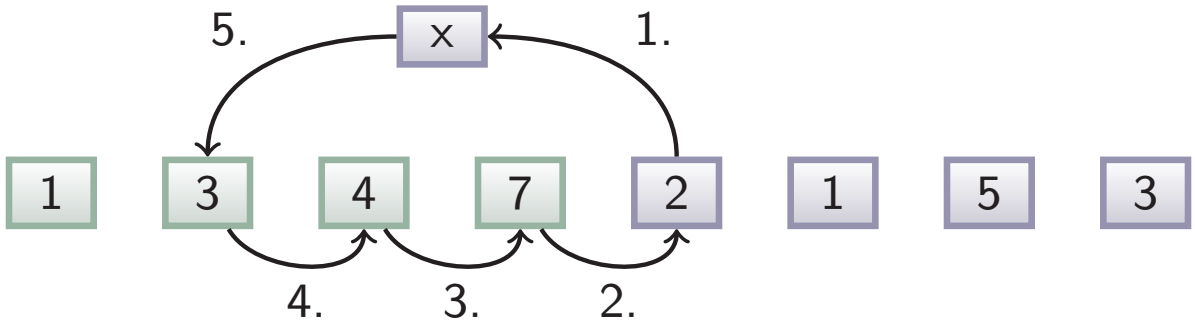
Közvetlen beszúrás

- A tömb egy $i(=4)$ hosszú rendezett szakaszból és egy $N-i$ hosszú rendezetlen szakaszból áll.



- A rendezetlen rész első elemét szúrjuk be a rendezett részbe, a megfelelő pozícióba
- Ezzel a rendezett szakasz hossza eggyel nőtt
- Kezdetben $i = 1$, az egyelemű tömb ugyanis rendezett

Közvetlen beszúrás



- A rendezett részben az új elem helyét $\log_2 i$ lépésben megtaláljuk

  Összehasonlítások száma: $\mathcal{O}(N \cdot \log_2 N)$
- A beszúráshoz átlagosan $i/2$ elemet el kell húzni

  Mozgatások száma: $\mathcal{O}(N^2)$ (max. $(N^2/2)$ mozgatás)

Közvetlen beszúrás

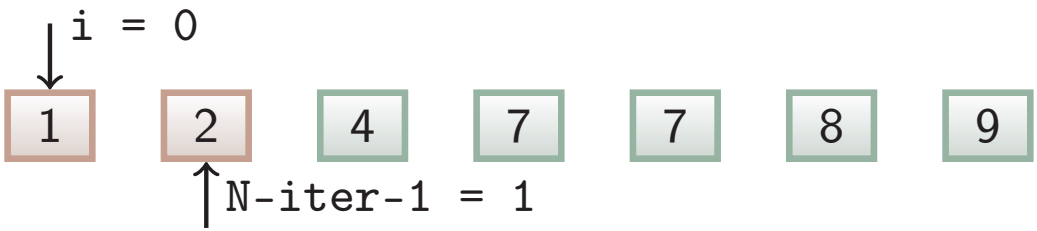
A közvetlen beszúrás C-kódja

```
1  for (i=1; i<N; i++)
2  {
3    s = t[i];                  /* beszúrandó elem */
4    for (a=0,f=i; a<f;)        /* log keresés 0 i között */
5    {
6      k = (a+f)/2;
7      if (t[k] < s)
8        a = k+1;
9      else
10       f = k;
11   }
12   for (j=i; j>a; j--) /* résztömb húzása */
13     t[j] = t[j-1];
14   t[a]=s;                    /* beszúrás */
15 }
```

Buborékrendezés

Szomszédos elemeket vizsgálunk. Ha rossz sorrendben állnak, csere

```
1  for (iter = 0; iter < n-1; ++iter)
2    for (i = 0; i < N-iter-1; ++i)
3      if (t[i] > t[i+1])
4        xchg(t+i, t+i+1);
```



Összehasonlítások száma:	$\mathcal{O}\left(N^2\right)$	$N^2/2$
Cserék száma:	$\mathcal{O}\left(N^2\right)$	max. $\left(N^2/2\right)$

- Az utolsó három körben nem cseréltünk semmit. Nem derül ez ki korábban?

Javított buborékrendezés cserék figyelésével

```
1  stop = n-1;
2  while (stop != 0) {
3    nextstop = 0; /* utolsó csere indexe */
4    for (i = 0; i < stop; ++i)
5      if (t[i] > t[i+1]) {
6        xchg(t+i, t+i+1)
7        nextstop = i;
8      }
9    stop = nextstop;
10 }
```

- Az összehasonlítások száma csökkent
- A cserék száma maradt

Rendező algoritmusok összehasonlítása

$N = 100\,000$	összehasonlítások	mozgatások száma
közvetlen kiválasztás	4 999 950 000	299 997
közvetlen beszúrás	1 522 642	2 499 618 992
buborék	4 999 950 000	7 504 295 712
javított buborék	4 999 097 550	7 504 295 712
gyorsrendezés	3 147 663	1 295 967

összehasonlító program

Nincs legjobb algoritmus[1].

[1] csak legrosszabb

Indextömbök

- Az adatmozgatások száma jelentősen csökkenthető, ha nem a tömbelemeket, hanem azok indexeit rendezzük

0	ABC123	Aladár
1	QE8BZX	Dzsenifer
2	S45FDO	Kristóf
3	KJ967F	Gyöngyvér
4	FEK671	Éva
5	F34K98	Mihály
6	D678EF	Berci

eredeti adatvektor

0	rendezés →	0
1		6
2		1
3		4
4		3
5		2
6		5

név szerint rendező indextömb

```
1  for (i = 0; i < n; ++i) /* névsor */
2    printf("%s\n", data[index[i]].name);
```

- Indexek helyett rendezhetünk mutatókat is, ha az eredeti tömb (vagy lista) a memóriában van

Rendezés több szempont szerint

- Több kulcs szerint rendezés indextömbökkel
- Gyors keresés érdekében érdemes az indextömbökben a kulcsokat is tárolni, és az indextömböket kulcs szerint rendezve tartani

0	ABC123	Aladár
1	QE8BZX	Dzsenifer
2	S45FDO	Kristóf
3	KJ967F	Gyöngyvér
4	FEK671	Éva
5	F34K98	Mihály
6	D678EF	Berci

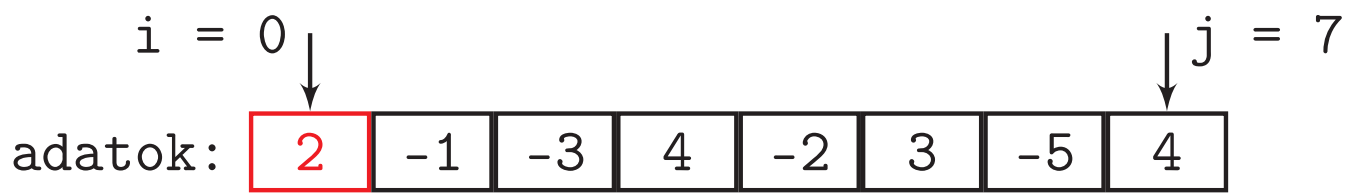
Aladár	0
Berci	6
Dzsenifer	1
Éva	4
Gyöngyvér	3
Kristóf	2
Mihály	5

ABC123	0
D678EF	6
FEK671	4
F34K98	5
KJ967F	3
QE8BZX	1
S45FDO	2

Gyorsrendezés (quick sort)

- Helyben szétválogatáson alapul
  - *n* elemű tömböt helyben szétválogatunk úgy, hogy adott tulajdonságú elemek a tömb elejére kerülnek
  - Kerüljenek a tömb elejére azok az elemek, melyek a rendezetlen tömb egy tetszőleges „pivot” (vezér) eleménél kisebbek

```
i ← 0; j ← n-1;
AMÍG i < j
  AMÍG t[i] <= t[0]
    i ← i+1;
  AMÍG t[j] > t[0]
    j ← j-1;
  HA i < j
    t[i] ↔ t[j]
t[0] ↔ t[j]
```



Gyorsrendezés (quick sort)

- Az n elemű tömböt n lépésben szétválogattuk „kis elemek” – vezérelem – „nagy elemek” tömbökre
- Válogassuk szét külön-külön a j elemű „kis elemek” tömböt és az $n-(j+1)$ elemű „nagy elemek” tömböt ugyanezzel a módszerrel!
- A rekurzió leállási feltétele: Az egyelemű tömb rendezett.
- Az algoritmus lépésszáma
  - első kör: n lépés
  - második kör: $(j)+(n-(j+1))=n-1$ lépés
  - A körök száma $\approx \log_2 n$ (átlagosan minden tömböt sikerül felezni)
- Rendezés $n \log_2 n$ lépésben!
- Ideális, ha pivot a felező (medián) elem
- Okosan (de gyorsan) kell kiválasztani

Gyorsrendezés (quick sort)

```
1  void qsort(int array[], int n)
2  {
3    if (n <= 1) return;
4    int pivot = 0, i = 0, j = n-1;
5    while (i < j)
6    {
7      while (i < n-1 && array[i] <= array[pivot])
8        i++;
9      while (array[j] > array[pivot])
10       j--;
11     if (i < j)
12       swap(array + i, array + j);
13   }
14   swap(array + pivot, array + j);
15
16   qsort(array, j); /* rekurzív hívások */
17   qsort(array + j+1, n - (j+1));
18 }
```

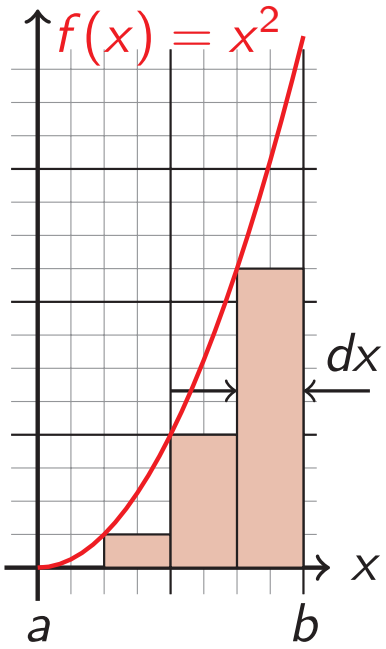
link

2. fejezet

Függvénymutatók

Bemelegítő feladat

Írjunk függvényt, amely az $f(x) = x^2$, ill. x^3 függvény görbéje alatti területet közelíti az $[a, b]$ intervallumon, n darab téglalappal




```
1  double xsquare(double x)
2  {
3    return x*x;
4  }
```

```
1  double integral_x2(double a, double b,
2                     unsigned n)
3  {
4    double dx = (b-a)/n, sum=0.0, x;
5    unsigned i;
6
7    for (x=a, i=0; i<n; x+=dx, i++)
8      sum += xsquare(x) * dx;
9
10   return sum;
11 }
```

Motiváció

Határ a csillagos ég...

```
1 double integral_x2(double a, double b, unsigned n);
2 double integral_x3(double a, double b, unsigned n);
3 double integral_sin(double a, double b, unsigned n);
4 double integral_sqrt(double a, double b, unsigned n);
```

- ha módosítjuk a számítást, minden ilyen függvényt át kell írnunk
- Helyette jó lenne a görbét megadó függvényt is átadnunk a számolónak

Függvénymutató

Függvénymutató

- a függvény is a memóriában van, ezért címe is képezhető → függvényre mutató pointernek is van értelme
- értékadásnál egy megfelelő típusú függvény azonosítóját kell megadnunk
  típus: paraméterek típusai + visszatérési érték típusa
- a mutatott függvényt a mutatón keresztül meghívhatjuk

```
1  double (*fx)(double);
2  fx = sqrt;
3  printf("%f\n", fx(5.0));   /* sqrt(5.0) */
4  fx = sin;                  /* <math.h> */
5  printf("%f\n", fx(1.57)); /* sin(1.57) */
```

Görbe alatti - bármire

```
13 double integral(double (*fx)(double),
14                 double a, double b, unsigned n)
15 {
16   double dx=(b-a)/n, sum=0.0, x;
17   unsigned i;
18
19   for(x=a, i=0; i<n; x+=dx, i++)
20     sum += fx(x) * dx;
21
22   return sum;
23 }
24
25 int main(void)
26 {
27   printf("%f\n", integral(xsquare, 1.0, 5.0, 100));
28   printf("%f\n", integral(xcube, 1.0, 5.0, 100));
29   return 0;
30 }
```

link

Függvény mint függvény paramétere

```
1  double integral(/* double (*fx)(double) */
2                  double fx(double),
3                  double a, double b, unsigned n) {
4  ...
5  sum += fx(x) * dx;
6  ...
7  }
8
9  printf("%f\n", integral(xsquare, 1.0, 5.0, 100));
```

- A függvényt mutatóval adjuk át
  Egyszerűsítés: a függvény fejlécét is írhatjuk
- A mutatót a függvényhívás operátorral () használjuk
- Aktuális paraméterként csak a függvény azonosítóját kell megadnunk, hasonló formában, mint a tömb átadásánál

Függvénymutatók tömbje

- Függvénypointerekből álló tömböt is képezhetünk

```
1 double (*(ftrig[2]))(double) ={sin, cos};
2
3 for (i = 0; i < 2; ++i)
4   printf("%f\n", ftrig[i](3.14));
```

- Könnyebben érthető a típus, ha `typedef`-et használunk

```
1 typedef double (*fvptr)(double);
2 fvptr fhyp[]={sinh, cosh};
```

- Megjegyzések
  - A függvénymutatókkal nem végezhetünk mutatóaritmetikai műveleteket
  - A függvénymutatók read-only területre mutatnak, a függvények nem írhatók felül

Menürendszer függvénypointerekkel

```c
void start_game(void) { printf("Ez jó volt\n"); }
void list_scores(void) { printf("Pityu: 10\n"); }
```

```c
typedef struct {
  char command[21];
  void (*func)(void);
} menu_t;
```

```c
menu_t menu[] = {
  {"játék",  start_game },
  {"pontok", list_scores},
  {"ment",   save_game  },
  {"",       NULL/*végjel*/}
};
```

```c
char command[21];
do {            /* soha többé nem kell hozzányúlni :) */
  unsigned i;
  printf("Válassz: ");
  scanf("%s", command);
  for (i = 0; menu[i].func != NULL; ++i)
    if (strcmp(command, menu[i].command)==0)
      menu[i].func();
} while (strcmp(command, "kilép"));
```

link

3. 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

Köszönöm a figyelmet.