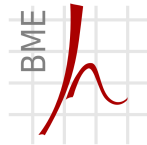




Rendezés. Rekurzió

A programozás alapjai I.



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

2020. november 16.

1. fejezet

Rendezés

Tartalom

1 Rendezés

- Bevezetés
- Közvetlen kiválasztás
- Közvetlen beszúrás
- Buborékrendezés
- Összevetés

■ Indextömbök

2 Rekurzió

- Definíció
- A rekurzió megvalósítása
- Rekurzió vagy iteráció
- Alkalmazások
- Közvetett rekurzió

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

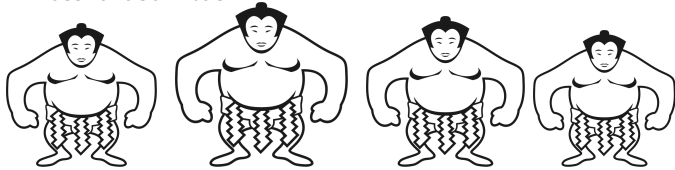
Mibe kerül a rendezés? ...

- = összehasonlítások száma \times egy összehasonlítás költsége
- + mozgatások (cserék) száma \times egy mozgatás költsége



Mi kerül sokba?

Az összehasonlítás



A mozgatus



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 }
```

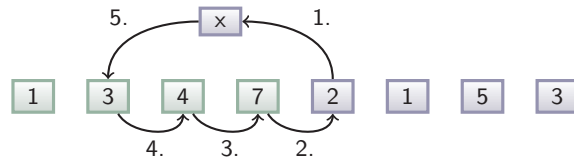
Összehasonlítások száma: $\mathcal{O}(N^2) \approx N^2/2$

Cserék száma: $\mathcal{O}(N) \quad N-1$



Közvetlen beszúrás

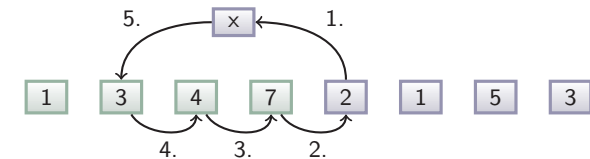
- A tömb egy $i(=4)$ hosszú rendezett szakaszból és egy $N-i$ hosszú rendezetlen szakaszból áll.



- 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. $(N^2/2)$ mozgatus)



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 }
```

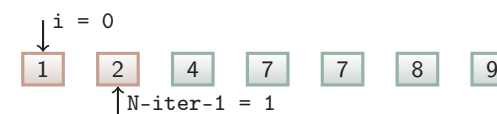


Buborékredezés

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              swap(t+i, t+i+1);
```



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

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



Javított buborékredezés 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              swap(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	mozgítások 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 algoritmus¹.

¹csak legrosszabb



Indextömbök

- Az adatmozgatások száma jelentősen csökkenthető, ha nem a tömelemeket, 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	1	2	3	4	5	6
---	---	---	---	---	---	---

rendezés

0	6	1	4	3	2	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

2. fejezet

Rekurzió



Rekurzió – definíció

Sok matematikai problémát rekurzívan fogalmazunk meg

- a_n sorozat összege

$$S_n = \begin{cases} S_{n-1} + a_n & n > 0 \\ a_0 & n = 0 \end{cases}$$

- Faktoriális

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

- Fibonacci-számsorozat

$$F_n = \begin{cases} F_{n-2} + F_{n-1} & n > 1 \\ 1 & n = 1 \\ 0 & n = 0 \end{cases}$$



Rekurzió – definíció

Sok hétköznapi problémát rekurzívan fogalmazunk meg

■ Felmenő-e Dózsa György?

$$\text{Felmenő-e?} = \begin{cases} \text{Apám/anyám felmenője-e?} \\ \text{Apám-e?} \\ \text{Anyám-e?} \end{cases}$$

■ Általában

$$\text{Probléma} = \begin{cases} \text{Egyszerűbb, hasonló problém(ák)} \\ \text{Triviális eset(ek)} \end{cases}$$



Rekurzió – kitekintés

■ Sokminden lehet rekurzív

Bizonyítás pl. teljes indukció

Definíció pl. Fibonacci-sorozat

Algoritmus pl. útvonalkeresés labirintusban

Adatszerkezet pl. láncolt lista, számítógép könyvtárstruktúrája

Geometria konstrukció pl. fraktál

■ Mi rekurzív adatszerkezetekkel és rekurzív algoritmusokkal foglalkozunk



Rekurzív algoritmusok C-ben

■ Faktoriális

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

Másoljuk be C-be!

```
1 unsigned factorial(unsigned n)
2 {
3     if (n > 0)
4         return factorial(n-1) * n;
5     else
6         return 1;
7 }
```

■ A függvény hívása

```
1 unsigned f = factorial(5); /* működik! */
2 printf("%u\n", f);
```



Kitérő

■ Hogyan képzejük el?

```
1 unsigned f0(void) { return 1; }
2 unsigned f1(void) { return f0() * 1; }
3 unsigned f2(void) { return f1() * 2; }
4 unsigned f3(void) { return f2() * 3; }
5 unsigned f4(void) { return f3() * 4; }
6 unsigned f5(void) { return f4() * 5; }
7 ...
8 unsigned f = f5();
```

■ Egyazon függvénynek sok különböző, egyszerre létező alakja

■ A paramétereik különböztetik meg őket



A rekurzió megvalósítása

Hogyan létezhet egy függvénynek egyszerre sok példánya?

```

1  /*
2   faktoriális rekurzív függvény
3   */
4  unsigned factorial(unsigned n)
5  {
6      if (n > 0)
7          return factorial(n-1) * n;
8      else
9          return 1;
10 }
11
12 int main(void)
13 {
14     ...
15     factorial(4);
16     ...
17 }
```

regiszter: 24



A rekurzió megvalósítása

- A C függvényhívási mechanizmusa eleve alkalmas a rekurzív függvényhívás megvalósítására
- A függvényt közvetve vagy direkt módon hívó függvények összes adatát (lokális változók, visszatérési cím) a veremben tároljuk
- A működés szempontjából közömbös, hogy egy függvény önmagát hívja vagy egy másik függvényt hív.
- A rekurzív hívások maximális mélysége: ami a verembe belefér



Rekurzió vagy iteráció – faktoriális

$n!$ számítása rekurzívan – elegáns, de pazarló

```

1  unsigned fact_rec(unsigned n)
2  {
3      if (n == 0)
4          return 1;
5      return fact_rec(n-1) * n;
6  }
```

és iterációval – „fapados”, de hatékony

```

1  unsigned fact_iter(unsigned n)
2  {
3      unsigned f = 1, i;
4      for (i = 2; i <= n; ++i)
5          f *= i;
6      return f;
7  }
```



Rekurzió vagy iteráció – Fibonacci

F_n számítása rekurzívan – elegáns, de kivárhatatlan!
A számítási idő n -nel exponenciálisan nő!

```

1  unsigned fib_rec(unsigned n)
2  {
3      if (n <= 1)
4          return n;
5      return fib_rec(n-1) + fib_rec(n-2);
6  }
```

és iterációval – „fapados”, de hatékony

```

1  unsigned fib_iter(unsigned n)
2  {
3      unsigned f[2] = {0, 1}, i;
4      for (i = 2; i <= n; ++i)
5          f[i%2] = f[(i-1)%2] + f[(i-2)%2];
6      return f[n%2];
7  }
```



Rekurzió vagy iteráció

- 1 Minden rekurzív algoritmus megoldható iterációval (ciklusokkal)
 - Nincs általános módszer az átírásra, sokszor igen nehéz
 - 2 Minden iterációval megoldható algoritmus megoldható rekurzívan
 - Könnyen automatizálható, általában nem hatékony
- A problémától függ, hogy melyik módszert érdemes használni



Iterációk rekurzívan

Tömb bejárása rekurzívan (for ciklus kiváltása)

```
1 void print_array(int array[], int n)
2 {
3     if (n == 0)
4         return;
5     printf("%3d", array[0]);
6     print_array(array+1, n-1); /* rekurzív hívás */
7 }
```

Lista bejárása rekurzívan

```
1 void print_list(list_elem *head)
2 {
3     if (head == NULL)
4         return;
5     printf("%3d", head->data);
6     print_list(head->next); /* rekurzív hívás */
7 }
```

Csak elvileg érdekesek, ezen esetekben is az iteráció hatékonyabb



Szám kiírása adott számrendszerben

rekurzívan

```
1 void print_base_rec(unsigned n, unsigned base)
2 {
3     if (n >= base)
4         print_base_rec(n/base, base);
5     printf("%d", n%base);
6 }
```

iterációval

```
1 void print_base_iter(unsigned n, unsigned base)
2 {
3     unsigned d; /* n-nél nem nagyobb base-hatvány */
4     for (d = 1; d*base <= n; d*=base);
5     while (d > 0)
6     {
7         printf("%d", (n/d)%base);
8         d /= base;
9     }
10 }
```



Amikor a rekurzió már egyértelműen hasznos

Az alábbi tömb egy labirintust tárol

```
1 char lab[9][9+1] = {
2     "+-----+",
3     "|           |",
4     "+-+ +-+ +-+",
5     "|           |",
6     "| + +-+ |",
7     "| | | |",
8     "+-+ +-+ |",
9     "|           |",
10    "+-----+",
11 };
```

Járjuk be a teljes labirintust adott (x,y) kezdőpozícióból

```
1 traverse(lab, 1, 1);
```

Minden lehetséges irányban elindulunk, és bejárjuk a még be nem járt labirintusrészeket.

Amikor a rekurzió már egyértelműen hasznos



A megoldás rekurzióval pofonegyszerű

```

1 void traverse(char lab[][9+1], int x, int y)
2 {
3     if (lab[x][y] != ' ')
4         return;
5     lab[x][y] = '.'; /* itt jártam */
6     traverse(lab, x-1, y);
7     traverse(lab, x+1, y);
8     traverse(lab, x, y-1);
9     traverse(lab, x, y+1);
10 }

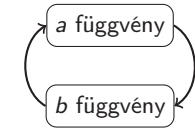
```

Iterációval embert próbáló – de nem lehetetlen – feladat lenne

Közvetett rekurzió



Közvetett rekurzió: Függvények „körbehívják egymást”

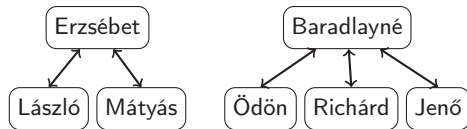


```

1 /* elődeklaráció */
2 void b(int); /* név, típus, paraméterek típusai */
3
4 void a(int n) {
5     ...
6     b(n); /* b hívható elődeklaráció miatt */
7     ...
8 }
9
10 void b(int n) {
11     ...
12     a(n);
13     ...
14 }

```

Elődeklaráció – kitekintés



Elődeklaráció közvetve rekurzív adatszerkezetek esetén is szükséges

```

1 /* elődeklaráció */
2 struct child_s;
3
4 struct mother_s { /* anya típus */
5     char name[50];
6     struct child_s *children[20]; /*gyerekek ptrtömbje*/
7 };
8
9 struct child_s { /* gyerek típus */
10     char name[50];
11     struct mother_s *mother; /*mutató anyára*/
12 };

```

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;
AMÍG i < j
    HA t[i] < pivot
        i ← i+1;
    EGYÉBKÉNT
        j ← j-1;
    HA t[j] >= pivot
        t[i] ↔ t[j]

```

i = 0 ↓ j = 8

adatok:

2	-1	-3	4	-2	3	-5	4
---	----	----	---	----	---	----	---



Gyorsrendezés (quick sort)

- Az n elemű tömböt n lépésben szétválogattuk „kis elemek” – „nagy elemek” tömbökre
- Válogassuk szét külön-külön az i elemű „kis elemek” tömböt és az $n - i$ elemű „nagy elemek” tömböt ugyanezzel a módszerrel!
- A rekurzió leállási feltétele: Az egyelemű tömb rendezett.
- Az algoritmus lépésszáma
 - első kör: n lépés
 - második kör: $i + (n - i) = n$ 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     int pivot = array[n/2];      /* kritikus! */
4     int i = 0, j = n;
5     while (i < j) {
6         if (array[i] < pivot)
7             i++;
8         else {
9             j--;
10            if (array[j] <= pivot)
11                swap(array+i, array+j); /* fv-nyel */
12        }
13    }
14
15    if (i > 1)
16        qsort(array, i);           /* rekurzív hívások */
17    if (n-i-1 > 1)
18        qsort(array+i, n-i);
19 }
```