Rendezés. Rekurzió

A programozás alapjai I.

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

2020. november 16.

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

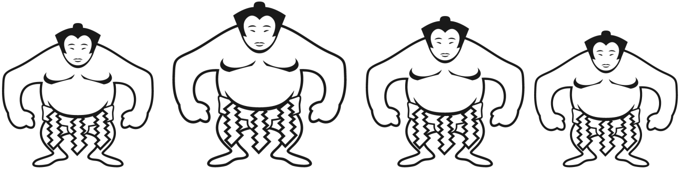
Mibe kerül a rendezés? ...

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

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

Mi kerül sokba?

- Az összehasonlítás



- A mozgatás

13.25 20.10 17.30

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
```

-3	-1	0	7	2	1	5	3

```
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];
```

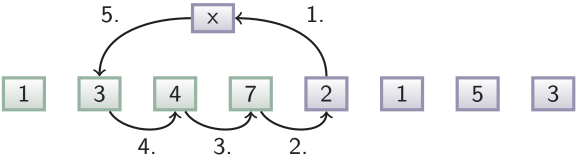
```c
for (i=0; i<N-1; ++i) {
  iMin = i;
  for (j=i+1; j<N; ++j)
    if (t[j] < t[iMin])
      iMin = j;
  swap(t+i, t+iMin);
}
```

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

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

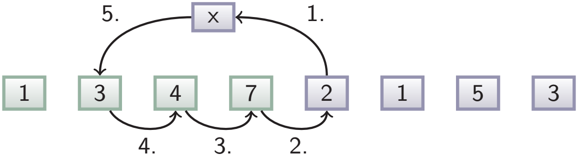
Közvetlen beszúrás

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

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

Közvetlen beszúrás

A közvetlen beszúrás C-kódja

```
for (i=1; i<N; i++)
{
  s = t[i];              /* beszúrandó elem */
  for (a=0,f=i; a<f;)    /* log keresés 0 i között */
  {
    k = (a+f)/2;
    if (t[k] < s)
      a = k+1;
    else
      f = k;
  }
  for (j=i; j>a; j--) /* résztömb húzása */
    t[j] = t[j-1];
  t[a]=s;             /* beszúrás */
}
```

Buborékrendezés

Szomszédos elemeket vizsgálunk. Ha rossz sorrendben állnak, csere

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

↓ `i = 0`

1	2	4	7	7	8	9

↑ `N-iter-1 = 1`

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

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

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

Javított buborékrendezés cserék figyelésével

```c
stop = n-1;
while (stop != 0) {
  nextstop = 0; /* utolsó csere indexe */
  for (i = 0; i < stop; ++i)
    if (t[i] > t[i+1]) {
      swap(t+i, t+i+1)
      nextstop = i;
    }
  stop = nextstop;
}
```

`i = 0`

1	2	4	7	7	8	9

`stop = 0`
`nextstop = 0`

- 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		0
1		6
2		1
3	$\xrightarrow{\text{rendezés}}$	4
4		3
5		2
6		5

név szerint rendező indextömb

```c
for (i = 0; i < n; ++i) /* névsor */
  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

- Felmenőm-e Dózsa György?

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

- 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
  - Geometriai 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!

```
unsigned factorial(unsigned n)
{
  if (n > 0)
    return factorial(n-1) * n;
  else
    return 1;
}
```

- A függvény hívása

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

Kitérő

- Hogyan képzejük el?

```
unsigned f0(void) { return 1; }
unsigned f1(void) { return f0() * 1; }
unsigned f2(void) { return f1() * 2; }
unsigned f3(void) { return f2() * 3; }
unsigned f4(void) { return f3() * 4; }
unsigned f5(void) { return f4() * 5; }
...
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?

```
/*
  faktoriális rekurzív függvény
 */
unsigned factorial(unsigned n)
{
  if (n > 0)
    return factorial(n-1) * n;
  else
    return 1;
}

int main(void)
{
  ...
   factorial(4);
  ...
}
```

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

```c
unsigned fact_rec(unsigned n)
{
  if (n == 0)
    return 1;
  return fact_rec(n-1) * n;
}
```

link

és iterációval – „fapados”, de hatékony

```c
unsigned fact_iter(unsigned n)
{
  unsigned f = 1, i;
  for (i = 2; i <= n; ++i)
    f *= i;
  return f;
}
```

link

Rekurzió vagy iteráció – Fibonacci

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

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

link

és iterációval – „fapados", de hatékony

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

link

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)

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

Lista bejárása rekurzívan

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

Csak elvileg érdekesek, ezen esetekben is az iteráció hatékonyabb

Szám kiírása adott számrendszerben

rekurzívan

```c
void print_base_rec(unsigned n, unsigned base)
{
  if (n >= base)
    print_base_rec(n/base, base);
  printf("%d", n%base);
}
```

link

iterációval

```c
void print_base_iter(unsigned n, unsigned base)
{
  unsigned d; /* n-nél nem nagyobb base-hatvány */
  for (d = 1; d*base <= n; d*=base);
  while (d > 0)
  {
    printf("%d", (n/d)%base);
    d /= base;
  }
}
```

link

Amikor a rekurzió már egyértelműen hasznos

Az alábbi tömb egy labirintust tárol

```c
char lab[9][9+1] = {
    "+-------+",
    "|       |",
    "+-+ ++ ++",
    "|       |",
    "| + +-+ |",
    "| | |   |",
    "+-+ +-+ |",
    "|     | |",
    "+-----+-+"
};
```

link

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

```c
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

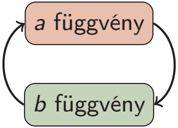
```
void traverse(char lab[][9+1], int x, int y)
{
  if (lab[x][y] != ' ')
    return;
  lab[x][y] = '.';          /* itt jártam */
  traverse(lab, x-1, y);
  traverse(lab, x+1, y);
  traverse(lab, x, y-1);
  traverse(lab, x, y+1);
}
```

link

Iterációval embert próbáló – de nem lehetetlen – feladat lenne

Közvetett rekurzió

Közvetett rekurzió: Függvények „körbehívják egymást”

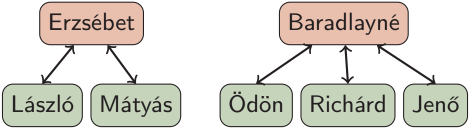



```
/* elődeklaráció */
void b(int); /* név, típus, paraméterek típusai */

void a(int n) {
  ...
  b(n); /* b hívható elődeklaráció miatt */
  ...
}

void b(int n) {
  ...
  a(n);
  ...
}
```

Elődeklaráció – kitekintés



Elődeklaráció közvetve rekurzív adatszerkezetek esetén is szükséges

```c
/* elődeklaráció */
struct child_s;

struct mother_s { /* anya típus */
  char name[50];
  struct child_s *children[20]; /*gyerekek ptrtömbje*/
};

struct child_s { /* gyerek típus */
  char name[50];
  struct mother_s *mother;        /*mutató anyára*/
};
```

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)

```
void qsort(int array[], int n)
{
  int pivot = array[n/2];     /* kritikus! */
  int i = 0, j = n;
  while (i < j) {
    if (array[i] < pivot)
      i++;
    else {
      j--;
      if (array[j] <= pivot)
        swap(array+i, array+j); /* fv-nyel */
    }
  }

  if (i > 1)
    qsort(array, i);              /* rekurzív hívások */
  if (n-i-1 > 1)
    qsort(array+i, n-i);
}
```

link

Köszönöm a figyelmet.