

Лекция 8. Сортировки, продолжение

Сортировка слиянием.

```

void MergeSort(int n, double *a, double *b)
{
    int k = n/2
    if (n<2) return;
    MergeSort(k, a);
    MergeSort(n-k, a+k);
    Merge(k, a, n-k, a+k, b);
    Copy(n, a, b);    // a <= b
}

void MergeParts(int n, double *a, int partlen, double *b)
{ // слияние упорядоченных фрагментов
    int k;
    int npairs = n / (2*partlen);
    int lastlen = n % (2*partlen);

    if (partlen > n) return;
    for (k = 0; k < npairs; k++) {
        Merge(partlen, a + k*partlen,
              partlen, a + (k+1)*partlen, b + k*partlen);
    }
    if (lastlen > partlen) Merge(partlen, a + k*partlen,
                                  lastlen, a + (k+1)*partlen, b + k*partlen);

    Copy(n, a, b);
}

void MergeSort(int n, double *a, double *b)
{
    int k;
    for (k=1; k<n; k*=2) {
        MergeParts(n, a, k, b);
    }
}

```

Слияние без дополнительной памяти — возможно ли?

Да, за $O(n \log n)$ т.е. трудоемкость сортировки будет $O(n \log^2 n)$

Да, за $O(n)$, но способ очень извращенный и практического смысла не имеет

Практическая страна вопроса с сортировками:

1. — тестирование быстродействия:

генерация массива длины N
замер времени на сортировку
проверка правильности упорядочивания

```
#include <time.h>

int n = ... ;
clock_t t1, t2;
double seconds;

double *a = (double*)malloc(n*sizeof(double));
FillArray (n, a);
t1 = clock();
Sort(n, a);
t2 = clock();
seconds = (double)(t2-t1) / CLOCKS_PER_SEC;
printf("sorting time %f\n", seconds);
printf("%s\n", (TestSort(n, a)) ? "success" : "failure");
...

bool TestSort(int n, double *a)
{
    int i;
    for (i=1; i<n; i++) {
        if (a[i-1] > a[i]) return false;
    }
    return true;
}
\end{verbatim}
```

{\bf 2.} --- универсальность.

Указатель на функцию.

```
\begin{verbatim}
int (*f)(int);
bool (*ts)(int, double*);
void (*s)(double);
int (*cmp)(double, double);
int (*cmpv)(const void *, const void *);

double CalculateSomething (double x, double (*f)(double))
{
\end{verbatim}
```

```

    return f(x) + f(2*x);
}

double MyFun (double x)
{
    return x*x;
}

z = CalculateSomething(2.5, MyFun);
z = CalculateSomething(1.0, sin);
z = CalculateSomething(5.0, sqrt);

```

Для универсальности процедуры сортировки сравнение элементов можно задавать отдельной функцией

```

int Compare(double a, double b)
{
    return (a<b) ? -1 : (a>b) ? 1 : 0;
}
int Compare1(double a, double b)
{
    return (a>b) ? -1 : (a<b) ? 1 : 0;
}
int Compare2(double a, double b)
{
    if (a*b>=0) > 0 return 0;
    if (a<0 && b>0) return 1;
    if (a>0 && b<0) return -1;
}

void Sort (int n, double *a, int (*cmp)(double, double))
{
    .....
    if ( cmp(a[i],a[j]) > 0 ) ..... // if (a[i] > a[j]) ....
    .....
}

Sort(n, a, Compare);
Sort(n, a, Compare1);
Sort(n, a, Compare2);

```

3. Стандартная функция qsort

```
#include <stdlib.h>
```

```
                size_t                size_t
void qsort (void *array, unsigned int n_elems, unsigned int elem_size,
            int (*cmp)(const void *, const void *));
```

Функция сравнения заготовлена на произвольные ситуации. Нам надо ее реализовать в конкретном случае, т.е. для типов элементов нашего массива.

```
double *a;
int n;
.....
qsort (a, n, sizeof(double), CmpDoubleAscending);
.....

int CmpDoubleAscending (const void *a, const void *b)
{
    const double *pa = (const double *)a;
    const double *pb = (const double *)b;
    return (*pa<*pb) ? -1 : (*pa>*pb) ? 1 : 0;
}
```