

Основы программирования

Простые алгоритмы поиска и сортировки

Задача и результаты поиска в массиве

Задача поиска:

Задан массив **A** из **n** элементов и некоторое значение **p** (поисковое). Требуется найти такой номер **i**, что **A[i]=p**.

Возможные результаты поиска:

- существует **единственный элемент** с номером **i**, для которого **A[i]=p**
- **A[i]≠p** при любых **i=0, 1, ..., n-1**
- существует **несколько элементов** с номерами **i1, i2, ...** таких, что **A[i1]=p, A[i2]=p, ...**

Поиск одного элемента в неупорядоченном массиве

```
int find_int(int *A, int n, int p)
{
    for (int i = 0; i < n; i++)
        if (A[i] == p) return i;
    return -1;
}
```

```
int find_double(double *A, int n, double p,
                double eps)
{
    for (int i = 0; i < n; i++)
        if (abs(A[i]-p) < eps) return i;
    return -1;
}
```

Трудоёмкость в наилучшем: $T(n) = O(1)$

Трудоёмкость в наихудшем: $T(n) = O(n)$

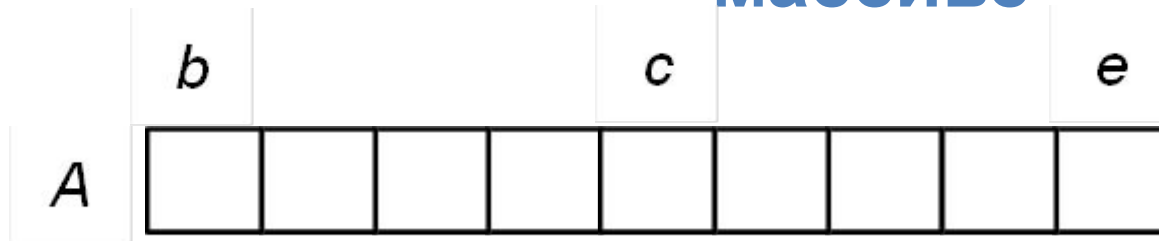
Простой поиск одного элемента в упорядоченном массиве

```
int find_sort_int(int *A, int n, int p)
{
    for (int i = 0; i < n && a[i] <= p; i++)
        if (A[i] == p) return i;
    return -1;
}
```

Трудоемкость в наилучшем: $T(n) = O(1)$

Трудоемкость в наихудшем: $T(n) = O(n)$

Дихотомический поиск в упорядоченном массиве



На каждом шаге алгоритма выделяется **область поиска**:
 $A[b], A[b+1], \dots, A[e]$

(начальные границы области: $b = 0, e = n-1$).

Поисковое значение p сравнивается с элементом $A[c]$,
где $c = (b+e)/2$. Возможны два исхода:

- $A[c] < p$, искомый элемент среди $A[c+1], \dots, A[e]$,
новая нижняя граница области поиска $b = c+1$
- $A[c] \geq p$, искомый элемент среди $A[b], \dots, A[c]$,
новая верхняя граница области поиска $e = c$

Поиск продолжается, пока $e > b$.

Алгоритм дихотомического поиска

```
int bin_search_first(int *A, int n, int p)
{
    int b = 0, e = n-1, c;
    while (b < e)
    {
        c = (b + e) / 2;
        if (A[c] < p) b = c+1;
        else e = c;
    }
    if (A[b] == p) return b;
    return -1;
}
```

Алгоритм дихотомического поиска

```
int bin_search_last(int *A, int n, int p)
{
    int b = 0, e = n-1, c;
    while (b < e)
    {
        c = (b + e + 1) / 2;
        if (A[c] <= p) b = c;
        else e = c-1;
    }
    if (A[b] == p) return b;
    return -1;
}
```

Трудоёмкость алгоритма

Пусть $2^{m-1} < k \leq 2^m$, (*)

где k – размер области поиска.

Вначале $k = n$.

При четном k размеры области уменьшаются ровно в два раза, а при нечетном – в два с округлением, неравенство (*) сохраняется.

После $m = \lceil \log_2 n \rceil$ шагов: $k = 1$.

Общее количество сравнений C в наихудшем случае: $C = \lceil \log_2 n \rceil + 1 = O(\log_2 n)$

Рекурсивная функция поиска

```
int bin_search_rec(int *A, int p,
                  int b, int e)
{
    int c;
    if (b == e)
        if (A[b] == p) return p;
        else return -1;
    else
    {
        c = (b + e) / 2;
        if (A[c] < p)
            return bin_search_rec(A, p, c+1, e);
        else
            return bin_search_rec(A, p, b, c);
    }
}
```

Вызов рекурсивной функции поиска

Функция `bin_search_rec` должна получать в качестве параметров не длину массива, а номера начального и конечного элемента области поиска. Для удобства пользователя можно создать **функцию-«обертку»**, которая будет только вызывать `bin_search_rec` :

```
int bin_search(int *A, int n, int p)
{
    return bin_search_rec(A, p, 0, n-1);
}
```

Пользователь будет вызывать функцию `bin_search`, не зная деталей реализации поиска.

Задача сортировки элементов массива

Задача сортировки (упорядочения) массива по возрастанию:

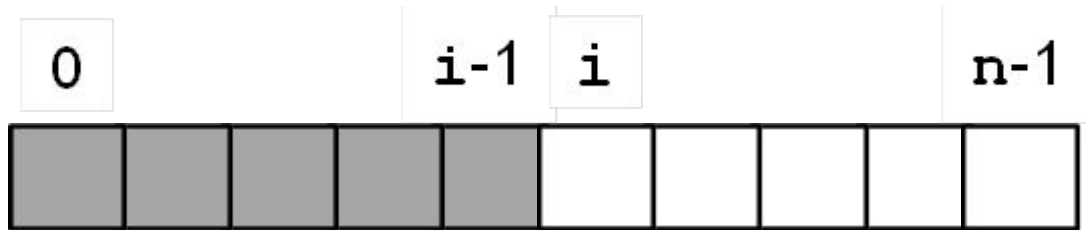
Задан произвольный массив **A** из **n** элементов. Требуется переставить его элементы таким образом, чтобы условие $A[i] \leq A[i+1]$ выполнялось для всех $i=0, \dots, n-2$.

При сортировке по убыванию меняется условие:

$A[i] \geq A[i+1]$ для всех $i=0, \dots, n-2$.

Набор значений в массиве **A** должен оставаться неизменным.

Алгоритм обменной сортировки



```
void exchange_sort(double *A, int n)
{
    int i, j; double z;
    for (i = 1; i < n; i++)
        for (j=i-1; j>=0 && A[j]>A[j+1];j--)
        {
            z=A[j]; A[j]=A[j+1]; A[j+1]=z;
            // или swap(A[j], A[j+1]);
        }
}
```

Алгоритм сортировки вставками

Данный алгоритм очень похож на предыдущий. Разница заключается в замене обмена элементов сдвигом:

- значение $z=A[i]$ запоминается
- $A[j]>z$, $j<i$, сдвигаются на одну позицию вправо
- z ставится на свое место в последовательности

```
void insert_sort(double *A, int n)
{
    int i, j; double z;
    for (i = 1; i < n; i++)
    { z = A[i];
      for (j=i-1; j>=0 && A[j]>z; j--)
          A[j+1] = A[j];
      A[j+1] = z;
    }
}
```

Трудоёмкость алгоритмов обменной сортировки и сортировки вставками

Общее количество выполнений внутреннего цикла в наихудшем случае:

$$1 + 2 + \dots + (n - 1) = n \cdot (n - 1) / 2,$$

Трудоёмкость в наихудшем $O(n^2)$.

Если массив **уже упорядочен**, то внутренний цикл ни разу не будет исполняться, и тогда трудоёмкость обоих алгоритмов (трудоёмкость **в наилучшем**) будет иметь порядок $O(n)$.

Алгоритм сортировки выбором

Среди m начальных элементов массива ищем максимальный и **меняем его местами** с последним ($m-1$). Выполняем эти действия для всех $m=n...2$.

```
void select_sort(double *A, int n)
{
    int i, m, nmax; double z;
    for (m = n; m > 1; m--)
    {
        for (nmax = 0, i = 1; i < m; i++)
            if (A[nmax] < A[i]) nmax = i;
        z=A[nmax]; A[nmax]=A[m-1]; A[m-1]=z;
    }
}
```

Трудоёмкость алгоритма сортировки выбором

Внутренний цикл выполняется $m-1$ раз, а m уменьшается во внешнем цикле от n до 2.

Общее количество элементарных шагов:

$$(n-1) + (n-2) + \dots + 1 = n \cdot (n-1) / 2 \approx n^2 / 2.$$

Трудоёмкость алгоритма и в наилучшем, и в наихудшем составляет $O(n^2)$.

Алгоритм пузырьковой сортировки

Для m начальных элементов массива проводим сравнение всех пар соседних элементов $A[i]$ и $A[i+1]$, $i=0\dots m-2$, и меняем их местами, если $A[i] > A[i+1]$. В результате максимальный элемент встает на последнее место ($m-1$). Повторяем эти действия для всех $m=n\dots 2$.

```
void bubble_sort(double *A, int n)
{
    int i, m;
    for (m = n; m > 1; m--)
        for (i = 0; i < m-1; i++)
            if (A[i] > A[i+1])
                swap(A[i], A[i+1]);
}
```

Улучшенный алгоритм пузырька

Алгоритм сортировки пузырьком можно улучшить. Если во внутреннем цикле не производится ни одного обмена, то массив уже отсортирован. Для отметки этого используем **флаг** – переменную **sorted**.

```
void bubble_sort_2(double *A, int n)
{
    int i, m; bool sorted = false;
    for (m = n; m > 1 && !sorted; m--)
        for (sorted=true, i = 0; i < m-1; i++)
            if (A[i] > A[i+1])
                { swap(A[i], A[i+1]); sorted=false; }
}
```

Косвенная упорядоченность в массиве

При косвенной сортировке исходный массив **A** не **изменяется**. Вместо этого формируется такой массив **Ind** из **n** индексов элементов **A**, что выполняется:
$$A[Ind[0]] \leq A[Ind[1]] \leq \dots \leq A[Ind[n-1]]$$
т.е. **Ind[i]** хранит номер элемента, который в упорядоченном массиве **A** стоял бы на **i**-м месте.

Модификация алгоритма сортировки для косвенной упорядоченности:

- 1) перед началом сортировки элементам **Ind** присваиваются начальные значения:
$$Ind[0]=0, \quad Ind[1]=1, \quad \dots, \quad Ind[n-1]=n-1$$
- 2) везде, где элемент массива **A[j]** используется в операции **сравнения**, заменить **A[j]** на **A[Ind[j]]**
- 3) везде, где элемент массива **A[j]** используется в **присваивании**, заменить **A[j]** на **Ind[j]**.

Косвенная сортировка алгоритмом пузырька

При косвенной сортировке необходимо сформировать целочисленный индексный массив `Ind` длины `n`.

Вариант 1: уже существующий массив `Ind` передается в функцию

```
void bubble_sort(double *A, int *Ind, int n)
{
    int i, m;
    for (i = 0; i < n; i++) Ind[i] = i;
    for (m = n; m > 1; m--)
        for (i = 0; i < m-1; i++)
            if (A[Ind[i]] > A[Ind[i+1]])
                swap(Ind[i], Ind[i+1]);
}
```

Косвенная сортировка алгоритмом пузырька

Вариант 2: динамический индексный массив `Ind` создается и формируется в функции, функция возвращает его адрес (указатель)

```
int* bubble_sort(double *A, int n)
{
    int i, m, *Ind;
    Ind = new int [n];
    for (i = 0; i < n; i++) Ind[i] = i;
    for (m = n; m > 1; m--)
        for (i = 0; i < m-1; i++)
            if (A[Ind[i]] > A[Ind[i+1]])
                swap(Ind[i], Ind[i+1]);
    return Ind;
}
```

Дихотомический поиск в косвенно упорядоченном массиве

```
int bin_search_first(int *A, int *Ind,
                    int n, int p)
{
    int b = 0, e = n-1, c;
    while (b < e)
    {
        c = (b + e) / 2;
        if (A[Ind[c]] < p) b = c+1;
        else e = c;
    }
    if (A[Ind[b]] == p) return b;
    return -1;
}
```

Слияние двух упорядоченных массивов

```
void merge(double *A, int n, double *B,  
           int m, double *C)  
{  
    int i=0, j=0, k=0;  
    while (i < n && j < m)  
        if (A[i] <= B[j]) C[k++] = A[i++];  
        else C[k++] = B[j++];  
    while (i < n) C[k++] = A[i++];  
    while (j < m) C[k++] = B[j++];  
}
```

На каждом шаге трех циклов в C переносится один элемент, поэтому трудоемкость составляет $O(n+m)$

Слияние двух массивов: вариант 2

```
void merge(double *A, int n, double *B,  
           int m, double *C)  
{  
    int i=0, j=0, k=0;  
    while (i < n || j < m)  
        if (j >= m) C[k++] = A[i++];  
        else if (i >= n) C[k++] = B[j++];  
        else if (A[i] <= B[j]) C[k++] = A[i++];  
        else C[k++] = B[j++];  
}
```


Рекурсивный алгоритм сортировки слиянием

При каждом вызове рекурсивной функции параметры задают границы текущей области сортировки: **b** – номер начального элемента, **e** – номер конечного. При первом вызове **b=0**, **e=n-1** (для массива длины **n**)

Идея алгоритма (исходный массив **A**, рабочий – **D**):

- вычисляется **c = (b+e) / 2** – центральный элемент области сортировки
- элементы **A[b...c]** сортируются рекурсивно
- элементы **A[c+1...e]** сортируются рекурсивно
- серии **A[b...c]** и **A[c+1...e]** сливаются в **D[b...e]**
- элементы **D[b...e]** копируются назад в **A[b...e]**

Рекурсивный алгоритм сортировки слиянием

```
void merge_rec(double *A, int b, int e,  
              double *D)  
{  
    int c = (b + e) / 2;  
    if (b < c) merge_rec(A, b, c, D);  
    if (c+1 < e) merge_rec(A, c+1, e, D);  
    merge_series(A, b, c, e, D); // слияние серий  
    for (int i = b; i <= e; i++)  
        A[i] = D[i];  
}
```

Слияние серий в сортировке слиянием

```
int i = b, j = c+1, k;  
for (k = b; k <= e; k++)  
    if (j > e) D[k] = A[i++];  
    else if (i > c) D[k] = A[j++];  
    else if (A[i] <= A[j]) D[k]=A[i++];  
    else D[k] = A[j++];
```

Вызов рекурсивной функции

Функция-обертка для рекурсивной функции **merge_rec** выделяет и освобождает память для динамического рабочего массива и вызывает **merge_rec**:

```
void merge_sort(double *A, int n)
{
    double *D = new double[n];
    merge_rec(A, 0, n-1, D);
    delete [] D;
}
```

Глубина рекурсии и трудоемкость

Пусть размер n упорядочиваемого массива

$$2^{m-1} < n \leq 2^m,$$

тогда не более чем за m последовательных делений размер фрагмента массива станет равным 1, поэтому

глубина рекурсии также не превысит $m = \lceil \log_2 n \rceil$

Рекуррентное соотношение для трудоемкости $T(n)$:

$$\begin{cases} T(1) = 1, \\ T(n) = 2T(n/2) + cn, \quad n = 2, 3, \dots, \end{cases}$$

где c – константа.

Полагая $2^{m-1} < n \leq 2^m$, получим:

$$\begin{aligned} T(n) &= 2T(n/2) + cn = \\ &= 2^2 T(n/4) + cn + cn = \dots \leq cnm = cn \lceil \log_2 n \rceil = O(n \log_2 n)_{29} \end{aligned}$$

Функции трудоемкости

n	$\lceil \log_2 n \rceil$	$n \lceil \log_2 n \rceil$	n^2	n^3	2^n
4	2	8	16	64	16
10	4	40	100	1000	1024
20	5	100	400	8000	$\approx 10^6$
40	6	240	1600	64000	$\approx 10^{12}$
100	7	700	10000	10^6	$\approx 10^{30}$
1000	10	10000	10^6	10^9	$\approx 10^{300}$
10^6	20	$20 \cdot 10^6$	10^{12}	10^{18}	$\approx 10^{300000}$
10^9	30	$30 \cdot 10^9$	10^{18}	10^{27}	$\approx 10^{300000000}$

На выполнение 10^{18} действий при скорости 10^9 действий в 1 секунду потребуется более 30 лет.