Лекция 9. Специальные сортировки

Битовые операции

В языке C есть еще операции с отдельными битами целых чисел. Выполняются побитово.

```
Операции   &   |   ^   ~              (&=  |=  ^=)      a = b & c;
                                                        a |= b;   a = a | b;
```

and	or	xor	not
0 & 0 = 0	0 | 0 = 0	0 ^ 0 = 0	~0 = 1
0 & 1 = 0	0 | 1 = 1	0 ^ 1 = 1	~1 = 0
1 & 0 = 0	1 | 0 = 1	1 ^ 0 = 1	
1 & 1 = 1	1 | 1 = 1	1 ^ 1 = 0	

Операции сдвига << >> (<<= >>=) b >> k a << 2
<< влево, дополняется нулями
>> вправо, дополняется 0 для unsigned и старшим разрядом для signed
Удобно работать вместе с шестнадцатеричными константами (маска)

```
int x = 0x2F;   00..00101111
x |= (1<<4);    00..00111111
x &= ~(1<<2);   00..00111011

int  SetBit (int x, int k)
{
    return x | (1<<k);
}
int  ClearBit (int x, int k);
{
    return x & ~(1<<k);
}
bool TestBit(int x, int k)
{
   return x & (1<<k);
}

void SwapBytes (int *x)             _A__ _B__ _C__ _D__     _A__ _B__ _D__ _C__
{
    unsigned b0 = *x & 0xff;
    unsigned b1 = *x & 0xff00;
    *x &= 0xffff0000;
    *x |= (b0 << 8) | (b1 >> 8);
}
```

Правило сдвига — умножение/деление на 2 в степени k

```
 3 << 2 есть 12
 12 >> 2 есть 3
-12 >> 2 есть -3
```

Сортировка подсчетом

Сортировка подсчетом (вариант с "генерацией" + вариант с перестановками).

```
  a  - исходный массив длины na
  b  - массив количеств или позиций длины nb=p
  c  - массив результата

void CountSort (int *a, int *b, int na, int nb)
{   int i,j=0;
    for (i=0; i<nb; i++) b[i] = 0;
    for (i=0; i<na; i++) b[a[i]]++;
    for (i=0; i<nb; i++) for ( ; b[i]!=0; b[i]--) a[j++] = i;
}

void CountSort2 (int *a, int *b, int *c, int na, int nb)
{   int i,s=0, s1;
    for (i=0; i<nb; i++) b[i] = 0;
    for (i=0; i<na; i++) b[a[i]]++;
    for (i=0; i<nb; i++) { s1 = s; s+=b[i]; b[i] = s1; }
    for (i=0; i<na; i++) c[b[a[i]]++] = a[i];
}
```

Трудоемкость $O(na + nb)$, память $nb + na$

Напрямую редко применяется, но как составная часть может быть весьма эффективна.

Поразрядные сортировки (radix sort)

Рассмотрим как тестовый пример, сортировку массива неотрицательных целых чисел. Применим идею быстрой сортировки, разделяя массив на части по значению k-го бита в представлении числа. Ниже мы обозначим номер бита, который принимается во внимание, через ibit.

```
int QuickBitSortPartition (int n, unsigned int *a, int ibit)
{
    int i, j;
    while(true) {
```

```
        for ( i=0; i<n && !TestBit(a[i], ibit); i++);
        for ( j=n-1; j>=0   &&  TestBit(a[j], ibit); j--);
        if (i==n || j==-1) return 0;
        if (i<j) {
           swap(a+i, a+j);
           ++i, --j;
        } else {
           return j+1;
        }
     }
  }
```

Далее схема быстрой сортировки.

Определение. Сортировка называется устойчивой (или монотонной), если элементы с одинаковыми значениями располагаются в отсортированном массиве в том же порядке относительно друг друга, что и в исходном массиве.

Пузырьковые сортировки устойчивые.

Перестанока максимума будет устойчивой, если всегда выбирать последний максимум.

Быстрая сортировка неустойчивая

Слияние — устойчивая (если правильно сливать)

Сортировка кучей — ???

Сортировка подсчетом — устойчивая.

Классическая radix сортировка

Рассматриваем число как набор “цифр”, например по 4 бита (всего 16 значений).

Сортируем подсчетом по младшей “цифре”.

Потом сортируем подсчетом по следующей “цифре” и т.д.

В силу устойчивости сортировки подсчетом в результате получим отсортированный массив.

В качестве “цифры” можно даже взять один двоичный разряд.

```
  void  BitRadixSort (unsigned int *a, int n, unsigned int *b)
  {
      int i, k, j0, j1, n0, n1;

      for (k=0; k<sizeof(int)*8; k++) {
         n0 = n1 = 0;
         // подсчет 0 и 1
         for (i=0; i<n; i++) {
            if (TestBit(a[i],k)) { n1++; } else { n0++; }
         }
         // сортировка в новый массив
         j0 = 0;
```

```
        j1 = n0;
        for (i=0; i<n; i++) {
            if (TestBit(a[i],k)) {
               b[j1++] = a[i];
            } else {
               b[j0++] = a[i];
            }
        }
        // копирование в исходный массив
        for (i=0; i<n; i++) { a[i] = b[i]; }
    }
}
```

Теорема. Любой метод сортировки, основанный на попарных сравнениях и перестановках элементов, не может иметь гарантированную трудоемкость лучше, чем $O(n \log n)$, где n — количество элементов в массиве.

Доказательство. Идея:

Дерево алгоритма (вершины — сравнения, ребра — переход с другой паре элементов).

В этом дереве $n!$ концевых вершин (листьев) так как каждая перестановка на входе должна в итоге прийти к отсортированному состоянию.

Формула Стирлинга $n! \approx \sqrt{2\pi n}(n/e)^n$

Можно доказать такое утверждение:

Любой метод сортировки, основанный на попарных сравнениях и перестановках элементов, не может иметь гарантированную трудоемкость лучше, чем $O(np)$, где n — количество элементов в массиве, а p — количество битов в представлении каждого числа из массива.