Курс лекций по олимпиадной информатике (C)

Лекция 4. Алгоритмы сортировки

Михаил Густокашин, 2009

Конспекты лекций подготовлены для системы дистанционной подготовки, действующей на сайте **informatics.mccme.ru**.

При нахождении ошибок или опечаток просьба сообщать по адресу

gustokashin@gmail.com

Версия C от 16.11.2009

1 Сортировка пузырьком

Условимся считать массив отсортированным, если элементы расположены в порядке неубывания (т.е. каждый элемент не меньше предыдущего). Все примеры будем рассматривать на типе `int`, однако он может быть заменен любым другим сравнимым типом данных.

Рассмотрение методов сортировки начнем с сортировки пузырьком (BubbleSort).

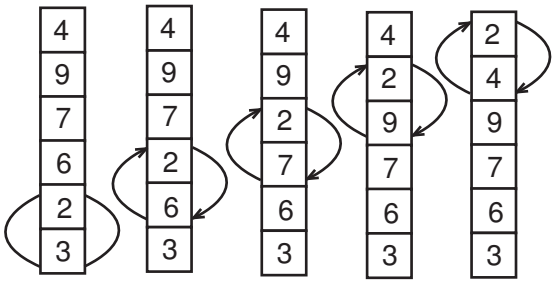

Рис. 1: Нулевой проход. Сравниваемые пары и обмены выделены

Это один из простейших методов сортировки, который обычно входит в школьный курс программирования. Название метода отражает его сущность: на каждом шаге самый «легкий» элемент поднимается до своего места («всплывает»). Для этого мы просматриваем все элементы снизу вверх, берем пару соседних элементов и, в случае, если они стоят неправильно, меняем их местами.

Вместо поднятия самого «легкого» элемента можно «топить» самый «тяжелый».

Т.к. за каждый шаг на свое место встает ровно 1 элемент (самый «легкий» из оставшихся), то нам потребуется выполнить N шагов.

Текст функции сортировки можно записать так:

```
void bubble_sort(int a[], int n)
{
  int i, j, k;
  For(i, n)
  for (j=n-1; j>i; j--)
    if (a[j-1] > a[j]) {
      k = a[j-1];
      a[j-1] = a[j];
      a[j] = k;
    }
}
```

Напомним, что здесь мы использовали макроподстановку

```
#define For(a,b) for(a=0; a<b; a++)
```

которая избавляет нас от необходимости писать длинные команды и уменьшает количество возможных опечаток и ошибок.

Алгоритм не использует дополнительной памяти, т.е. все действия осуществляются на одном и том же массиве.

Сложность алгоритма сортировки пузырьком составляет $O(N2)$, количество операций сравнения: $N \times (N-1)/2$. Это очень плохая сложность, но алгоритм имеет два плюса.

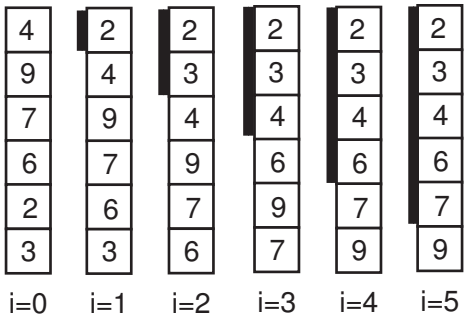

Рис. 2: Номер прохода. Отсортированная часть выделена полосой

Во-первых, он легко реализуется, а значит, может и должен применяться в тех случаях, когда требуется однократная сортировка массива. При этом размер массива не должен быть больше 10000, т.к. иначе алгоритм сортировки пузырьком не будет укладываться в отведенное время.

Во-вторых, сортировка пузырьком использует только сравнения и перестановки соседних элементов, а значит, может использоваться в тех задачах, где явно разрешен только такой обмен и для сортировки, например, списков.

Существуют разнообразные «оптимизации» сортировки пузырьком, которые усложняют (а нередко и увеличивают время работы алгоритма), но не приносят выгоды ни в плане сложности, ни в плане быстродействия.

На этом плюсы сортировки пузырьком заканчиваются. В дальнейшем мы еще более сузим область применения сортировки пузырьком.

2 Сортировка прямым выбором

Рассмотрим еще один квадратичный алгоритм, который, однако, является оптимальным по количеству присваиваний и может быть использован, когда по условию задачи необходимо явно минимизировать количество присваиваний.

Суть метода заключается в следующем: мы будем выбирать минимальный элемент в оставшейся части массива и приписывать его к уже отсортированной части. Повторив эти действия N раз, мы получим отсортированный массив.

```
void select_sort(int a[], int n)
{
  int i, j, k;
  For (i,n) {
    k=i;
    for(j=i+1; j<n; j++)
      if (a[j]<a[k]) k=j;
    j=a[k]; a[k]=a[i]; a[i]=j;
  }
}
```

Количество сравнений составляет $O(N^2)$, а количество присваиваний всего $O(N)$. В целом это плохой метод, и он должен быть использован только в случаях, когда явно необходимо минимизировать количество присваиваний.

3 Пирамидальная сортировка

Начнем рассмотрение эффективных алгоритмов сортировки (работающих за $O(N \log N)$) с пирамидальной сортировки, в которой используются знакомые нам идеи кучи.

Мы будем выбирать из кучи самый большой элемент, и записывать его в начало уже отсортированной части массива (сортировка выбором в обратном порядке). Т.е. отсортированный массив будет строиться от конца к началу. Такие ухищрения необходимы, чтобы не было необходимости в дополнительной памяти и для ускорения работы алгоритма — куча будет располагаться в начале массива, а отсортированная часть будет находиться после кучи.

Напомним свойство кучи максимумов: элементы с индексами $i+1$ и $i+2$ не больше, чем элемент с индексом i (естественно, если $i+1$ и $i+2$ лежат в пределах кучи). Пусть n — размер кучи, тогда вторая половина массива (элементы от $n/2+1$ до n) удовлетворяют свойству кучи. Для остальных элементов вызовем функцию «проталкивания» по куче, начиная с $n/2$ до 0.

```
void down_heap(int a[], int k, int n)
{
  int  temp=a[k];
  while (k*2+1 < n) {
    y=k*2+1;
    if (y < n-1 && a[y] < a[y+1]) y++;
    if (temp >= a[y]) break;
    a[k]=a[y];
    k=y;
  }
  a[k]=temp;
}
```

Эта функция получает указатель на массив, номер элемента, который необходимо протолкнуть и размер кучи. У нее есть небольшие отличия от обычных функций работы с кучей. Номер минимального предка хранится в переменной y, если необходимость в обменах закончена, то мы выходим из цикла и записываем просеянную переменную на предназначенное ей место.

Сама сортировка будет состоять из создания кучи из массива и N переносов элементов с вершины кучи с последующим восстановлением свойства кучи:

```
void heap_sort(int a[], int n) {
  int i, temp;
  for(i=n/2-1; i >= 0; i--) down_heap(a, i, n);
  for(i=n-1; i > 0; i--) {
    temp=a[i]; a[i]=a[0]; a[0]=temp;
    down_heap(a, 0, i);
```

```
  }
}
```

Всего в процессе работы алгоритма будет выполнено $3 \times N/2 - 2$ вызова функции `down_heap`, каждый из которых занимает $O(\log N)$. Таким образом, мы и получаем искомую сложность в $O(N \log N)$, не используя при этом дополнительной памяти. Количество присваиваний также составляет $O(N \log N)$.

Пирамидальную сортировку следует осуществлять, если из условия задачи понятно, что единственной разрешенной операцией является «проталкивание» элемента по куче, либо в случае отсутствия дополнительной памяти.

4 Быстрая сортировка

Мы уже рассматривали идеи, которые используются в быстрой сортировке, при поиске порядковых статистик. Точно так же, как и в том алгоритме, мы выбираем некий опорный элемент и все числа, меньшие его перемещаем в левую часть массива, а все числа большие его — в правую часть. Затем вызываем функцию сортировки для каждой из этих частей.

Таким образом, наша функция сортировки должна принимать указатель на массив и две переменные, обозначающие левую и правую границу сортируемой области.

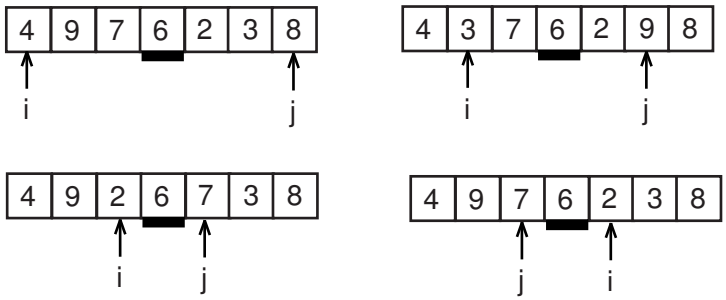

Рис. 3: Первый проход быстрой сортировки

Остановимся более подробно на выборе опорного элемента. В некоторых книгах рекомендуется выбирать случайный элемент между левой и правой границей. Хотя теоретически это красиво и правильно, но на практике следует учитывать, что функция генерации случайного числа достаточно медленная и такой метод заметно ухудшает производительность алгоритма в среднем.

Наиболее часто используется середина области, т.е. элемент с индексом $(l + r)/2$. При таком подходе используются быстрые операции сложения и деления на два, и в целом он работает достаточно неплохо. Однако в некоторых задачах, где сутью является исключительно сортировка, хитрое жюри специально подбирает тесты так, чтобы «завалить» стандартную быструю сортировку с выбором опорного элемента из середины. Стоит заметить, что это очень редкая ситуация, но все же стоит знать, что можно выбирать произвольный элемент с индексом m так, чтобы выполнялось неравенство $l \leq m \leq r$. Чтобы это условие выполнялось, достаточно выбрать произвольные два числа x и y и выбирать m исходя из следующего соотношения: $m = (x \times l + y \times r)/(x + y)$.

В целом такой метод будет незначительно проигрывать выбору среднего элемента, т.к. требует двух дополнительных умножений.

Приведем текст функции быстрой сортировки с выбором среднего элемента в качестве опорного:

```
void quick_sort(int a[], int left, int right)
{
  int i = left, j = right, temp, p;
  p = a[(left+right)/2];
  do {
    while (a[i] < p) i++;
    while (a[j] > p) j--;
    if (i <= j) {
      temp = a[i]; a[i] = a[j]; a[j] = temp;
      i++; j--;
    }
  } while (i <= j);
  if (j > left) quick_sort(a, left, j);
  if (i < right) quick_sort(a, i, right);
}
```

Чтобы воспользоваться быстрой сортировкой, необходимо передать в функцию левую и правую границы сортируемого массива (т.е., например, вызов для массива `a` будет выглядеть как `quick_sort(a, 0, n-1)`.

Алгоритм быстрой сортировки в среднем использует $O(N \log N)$ сравнений и $O(N \log N)$ присваиваний (на практике даже меньше) и использует $O(\log N)$ дополнительной памяти (стек для вызова рекурсивных функций). В худшем случае алгоритм имеет сложность $O(N^2)$ и использует $O(N)$ дополнительной памяти, однако вероятность возникновения худшего случая крайне мала: на каждом шаге вероятность худшего случая равна $2/N$, где N — текущее количество элементов.

Рассмотрим возможные оптимизации метода быстрой сортировки.

Во-первых, при вызове рекурсивной функции возникают накладные расходы на хранение локальных переменных (которые нам не особо нужны при рекурсивных вызовах) и другой служебной информацией. Таким образом, при замене рекурсии стеком мы получим небольшой прирост производительности и небольшое снижение требуемого объема дополнительной памяти.

Во-вторых, как мы знаем, вызов функции — достаточно накладная операция, а для небольших массивов быстрая сортировка работает не очень хорошо. Поэтому, если при вызове функции сортировки в массиве находится меньше, чем K элементов, разумно использовать какой-либо нерекурсивный метод, например, сортировку вставками или выбором. Число K при этом выбирается в районе 20, конкретные значения подбираются опытным путем. Такая модификация может дать до 15% прироста производительности.

Быструю сортировку можно использовать и для двусвязных списков (т.к. в ней осуществляется только последовательный доступ с начала и с конца), но в этом случае возникают проблемы с выбором опорного элемента — его приходится брать первым или последним в сортируемой области. Эту проблема можно решить неким набором псевдослучайных перестановок элементов списка, тогда даже если данные были подобраны специально, эффект нейтрализуется.

5 Сортировка слияниями

Сортировка слияниями также основывается на идее, которая уже была нами затронута при рассмотрении алгоритма поиска двух максимальных элементов. В этом алгоритме мы сначала разобьем элементы на пары и упорядочим их внутри пары. Затем из двух пар создадим упорядоченные четверки и т.д.

3	7	8	2	4	6	1	5	Последовательности длины 1
3 7		2 8		4 6		1 5		Слияние до упорядоченных пар
2 3 7 8				1 4 5 6				Слияние пар в упорядоченные четверки
1 2 3 4 5 6 7 8								Слияние пар в упорядоченные четверки

Интерес представляет сам процесс слияния: для каждой из половинок мы устанавливаем указатели на начало, смотрим, в какой из частей элемент по указателю меньше, записываем этот элемент в новый массив и перемещаем соответствующий указатель.

Опишем функцию слияния следующим образом:

```
void merge(int a[], int b[], int c, int d, int e)
{
  int p1=c, p2=d, pres=c;
  while (p1 < d && p2 < e)
    if (a[p1] < a[p2])
      b[pres++] = a[p1++];
    else
      b[pres++] = a[p2++];
  while (p1 < d)
    b[pres++]=a[p1++];
  while (p2 < e)
    b[pres++]=a[p2++];
}
```

Здесь a — исходный массив, b — массив результата, c и d — указатели на начало первой и второй части соответственно, e — указатель на конец второй части.

Далее опишем довольно хитрую нерекурсивную функцию сортировки слиянием:

```
void merge_sort(int a[], int n)
{
  int *temp, *a2=a, *b=(int*)malloc(n*sizeof(int)), *b2;
  int c, k = 1, d, e;
  b2=b;
  while (k <= 2*n) {
    for (c=0; c<n; c+=k*2) {
      d=c+k<n?c+k:n;
      e=c+2*k<n?c+2*k:n;
      merge(a2, b, c, d, e);
    }
    temp = a2; a2 = b; b = temp;
    k *= 2;
```

```
  }
  for (c=0; c<n; c++)
    a[c] = a2[c];
  free(b2);
}
```

Рекурсивная реализация сортировки слияниями несколько проще, но обладает меньшей эффективностью и требует $O(\log N)$ дополнительной памяти.

Алгоритм имеет сложность $O(N \log N)$ и требует $O(N)$ дополнительной памяти.

В оригинале этот алгоритм был придуман для сортировки данных во внешней памяти (данные были расположены в файлах) и требует только последовательного доступа. Этот алгоритм применим для сортировки односвязных списков.

6 Сортировка подсчетом

Это сортировка может использоваться только для дискретных данных. Допустим, у нас есть числа от 0 до 99, которые нам следует отсортировать. Заведем массив размером в 100 элементов, в котором будем запоминать, сколько раз встречалось каждое число (т.е. при появлении числа будем увеличивать элемент вспомогательного массива с индексом, равным этому числу, на 1). Затем просто пройдем по всем числам от 0 до 99 и выведем каждое столько раз, сколько оно встречалось. Сортировка реализуется следующим образом:

```
for (i=0; i<MAXV; i++)
  c[i] = 0;
for (i=0; i<n; i++)
  c[a[i]]++;
k=0;
for (i=0; i<MAXV; i++)
  for (j=0; j<c[i]; j++)
    a[k++]=i;
```

Здесь $MAXV$ — максимальное значение, которое может встречаться (т.е. все числа массива должны лежать в пределах от 0 до $MAXV - 1$).

Алгоритм использует $O(MAXV)$ дополнительной памяти и имеет сложность $O(N + MAXV)$. Его применение дает отличный результат, если $MAXV$ намного меньше, чем количество элементов в массиве.

7 Поразрядная сортировка

Алгоритм сортировки подсчетом чрезвычайно привлекателен своей высокой производительностью, но она ухудшается при возрастании $MAXV$, также резко возрастают требования к дополнительной памяти. Фактически, невозможно осуществить сортировку подсчетом для переменных типа `unsigned int` ($MAXV$ при этом равно 2^{32}).

В качестве развития идеи сортировки подсчетом рассмотрим поразрядную сортировку. Сначала отсортируем числа по последнему разряду (единиц). Затем повторим

то же самое для второго и последующих разрядов, пользуюсь каким либо устойчивым алгоритмом сортировки (т.е. если числа с одинаковым значением в сортируемом разряде шли в одном порядке, то в отсортированной последовательности они будут идти в том же порядке).

Для примера приведем таблицу, в первом столбце которой расположены исходные данные, а в последующих — результат сортировки по разрядам.

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	837	657	839

Для самой сортировки будем использовать сортировку подсчетом. После этого будем переделывать полученную таблицу так, чтобы для каждого возможного значения разряда сохранялась позиция, начиная с которой идут числа с таким значением в соответствующем разряде (т.е. сколько элементов имеют меньшее значение в этом разряде). Назовем этот массив c.

После этого будем проходить по всему исходному массиву, смотреть на текущее значение разряда (i), записывать текущее число во вспомогательный массив (b) на позицию $c[i]$, а затем увеличивать $c[i]$ (чтобы новое число с таким же значением текущего разряда не легло поверх уже записанного).

Пусть количество знаков в числе равно k, а количество возможных значений равно m (система счисления, использованная при записи числа). Тогда количество присваиваний, производимое алгоритмом, будет равно $O(k \times N + k \times m)$, а количество дополнительной памяти — $O(N + k \times m)$.

Приведем эффективную реализацию поразрядной сортировки для беззнаковых 4-байтных чисел (`unsigned int`). Мы будем использовать 4 разряда, каждый из которых равен байту (система счисления с основанием 256). Эта реализация использует несколько хитростей, которые будут пояснены ниже.

```
void radix_sort(unsigned int a[], int n)
{
  unsigned int *t, *a2=a;
  unsigned int *b=(unsigned int*) malloc(n*sizeof(int));
  unsigned int *b2;
  unsigned char *bp;
  int i, j, npos, temp;
  int c[256][4];
  b2 = b;
  memset(c, 0, sizeof(int)*256*4);
  bp = (unsigned char*) a;
  For(i,n)
    For (j,4) {
      c[*bp][j]++;
      bp++;
    }
  For(j,4) {
    npos = 0;
    For(i,256) {
      temp = c[i][j];
      c[i][j] = npos;
```

```
      npos += temp;
    }
  }
  For(j,4) {
    bp = (unsigned char*) a2 + j;
    For(i,n) {
      b[c[*bp][j]++] = a2[i];
      bp += 4;
    }
    t = a2; a2 = b; b = t;
  }
  free(b2);
}
```

Функция `memset` используется для заполнения заданной области памяти нулями (обнуление массива), она находится в библиотеке `string.h`. Всю таблицу сдвигов (c) мы будем строить заранее для всех 4 разрядов. Для эффективно доступа к отдельным байтам мы будем использовать указатель bp типа `unsigned char *` (тип `char` как раз занимает 1 байт и может трактоваться как число). Затем мы формируем модифицированную таблицу и проводим собственно функцию расстановки чисел по всем 4 разрядам.

Внимательный читатель заметит в приведенной функции несколько мест, которые на первый взгляд кажутся ошибочными. Хотя в текстовом описании мы и говорили, что следует сортировать, начиная с последних разрядов, в реализации мы начинаем с первых байтов. Это объясняется тем, что в архитектуре **x86** числа хранятся в «перевернутом» виде — это было сделано для совместимости с младшими моделями.

Второе возможное место для ошибки — массив a не ставится в соответствие указателю отсортированного массива и не осуществляется копирование отсортированных элементов в него в конце работы функции. Это связано с четным количеством разрядов, так что результат в итоге и так оказывается в массиве a.

Вообще говоря, далеко не обязательно так сильно связывать поразрядную сортировку с аппаратными средствами. Более того, основное удобство поразрядной сортировки состоит в там, что с ее помощью можно сортировать сложные структуры, такие как даты, строки (массивы) и другие структуры со многими полями.

8 Сравнение производительности сортировок

Название	**Сравнений**	**Присваиваний**	**Память**	**Время**
Пузырьком	$O(N^2)$	$O(N^2)$	0	$---$
Выбором	$O(N^2)$	$O(N)$	0	$---$
Пирамидальная	$O(N \log N)$	$O(N \log N)$	0	16437
Быстрая	$O(N \log N)$	$O(N \log N)$	$O(\log N)$	5618
Слиянием	$O(N \log N)$	$O(N \log N)$	$O(N)$	7669
Подсчетом	0	$O(N)$	$O(MAXV)$	322
Поразрядная	0	$O(kN + km)$	$O(N + km)$	1784

Тестирование проводилось на 10^7 случайных чисел, которые не превышали 10^5. Для квадратичных алгоритмов тестирование не проводилось (поскольку это заняло

бы очень большое время). При тестировании использовались приведенные выше функции. В таблице k — количество «цифр», а m — количество значений, которое может принимать каждая цифра.

Быстрая сортировка в худшем случае имеет $O(N^2)$ сравнений и присваиваний.

Как видно из таблицы, поразрядная сортировка опережает все остальные на «обычных» данных, но для практического применения больше подходит быстрая сортировка, т.к. разница уменьшается по мере уменьшения количества сортируемых элементов, а само по себе считывание 10^7 элементов занимает время намного большее, чем разница во времени между этими сортировками.

При решении той или иной задачи следует выбирать нужный тип сортировки исходя из таблицы и специфических условий применимости (минимизация количества сравнений или присваиваний, сортировка списков или последовательный доступ к данным).

Еще одна практическая рекомендация заключается в следующем: при сортировке сложных структур (например, строк) следует не производить обмены, а просто переставлять указатели (строка может состоять из нескольких тысяч символов, а указатель занимает 4 байта и процесс сортировки может ускориться во много раз).

9 Сканирующая прямая

Очень часто в олимпиадных задачах возникает необходимость обработать некоторые события по порядку. В этом случае события упорядочиваются по оси «времени» и сканирующая прямая движется вдоль этой прямой, переходя от события к следующему. Рассмотрим, например, такую классическую задачу: на прямой задано N отрезков двумя числами: координатами начала и конца отрезка. Требуется определить, какое максимальное количество отрезков покрывает некоторую точку.

Будем двигаться по «событиям» (в нашем случае событием считается достижение точки, которая является началом или конца отрезка) слева направо. Если очередная точка — начало отрезка, то прибавим к счетчику начавшихся, но не кончившихся отрезков единицу. Если же очередная точка — конец отрезка, то вычтем из счетчика единицу. В случае, если в одной точке происходит несколько событий, то будем учитывать сначала начала отрезков, а затем их концы: в нашей задаче точка считается покрытой отрезком, если она покрыта хотя бы одной его точкой, а значит в случае если в некоторой точке предыдущий отрезок кончается, а новый — начинается, то необходимо учесть, что точка покрыта двумя отрезками. В других задачах упорядочивания одновременных событий определяется из условия.

Для реализации такого решения, необходимо «слить» координаты начал и концов отрезков в один массив, при этом дополнив их признаком того, является ли точка началом или концом отрезка. Затем этот массив сортируется в первую очередь по координате, а в случае их совпадения — по признаку начала или конца. После этого необходим цикл, который проходит по всем упорядоченным точкам и прибавляет единицу к счетчику в случае если очередная точка имеет признак начала и вычитает единицу если очередная точка является концом отрезка. Достигнутый во время этого прохода максимум и будет являться ответом на задачу.

Задача становится интереснее, если кроме самого максимального количества отрезков, которыми покрыта точка, необходимо вывести и номера этих отрезков.

Рассмотрим решение «в лоб». Список начавшихся, но не закончившихся, отрезков

легко поддерживать в виде булевского массива. Однако при нахождении очередного максимума надо совершить проход по этому массиву чтобы сохранить ответ. Один проход по такому массиву займет $O(N)$ времени, ответ также может обновляться порядка N раз (когда все отрезки последовательно вложены друг в друга). Таким образом, сложность такого решения составит $O(N^2)$, что слишком много для большинства задач такого рода.

Более разумно сделать двухпроходное решение. Первый проход будет решать исходную задачу — искать максимальное количество отрезков, покрывающих точку. Второй проход будет незначительно отличаться от первого, а именно: когда значение счетчика начавшихся, но не закончившихся отрезков станет равно максимальному значению, следует прекратить дальнейший поиск и пройти по булевскому массиву, в котором помечены эти отрезки и вывести их. Сложность такого решения составит $O(N)$ (первый и второй проход займут порядка N операций, проход по массиву открытых отрезков также займет $O(N)$ операций и будет выполнен один раз.