2. Сортировка элементов массива(внутренняя сортировка)

2.1. Начальные сведения

Массивы данных широко используются в языках программирования для представления объектов, состоящих из определенного числа элементов массива одного и того же типа(класса).

Рассмотрим алгоритмы сортировки массивов, являющиеся одной из фундаментальных алгоритмических задач программирования. В общем случае сортировку понимают как процесс перегруппировки, заданного множества объектов в определенном порядке. Обычно под алгоритмом сортировки подразумевают алгоритм упорядочивания множества элементов по возрастанию или убыванию.

При наличии элементов с одинаковыми значениями, в упорядоченной последовательности они располагаются рядом друг за другом в любом порядке. Однако иногда бывает полезно сохранять первоначальный порядок элементов с одинаковыми значениями.

В алгоритмах сортировки лишь часть данных используется в качестве ключа сортировки. Ключом сортировки называется атрибут (или несколько атрибутов), по значению которого определяется порядок элементов. Таким образом, при написании алгоритмов сортировок массивов следует учесть, что ключ полностью или частично совпадает с данными.

Практически каждый алгоритм сортировки можно разбить на 3 части:

1. сравнение, определяющее упорядоченность пары элементов;
2. перестановку, меняющую местами элементы пары;
3. собственно сортирующий алгоритм, который осуществляет сравнение и перестановку элементов до тех пор, пока все элементы множества не будут упорядочены.

Алгоритмы сортировки имеют большое практическое применение. Их можно встретить там, где речь идет об обработке и хранении больших объемов информации. Некоторые задачи обработки данных решаются проще, если данные заранее упорядочить.

2.2. Оценка алгоритмов сортировки

Проблема сортировки породила огромное количество разнообразных решений. Универсального, наилучшего алгоритма сортировки на данный момент не существует. Однако, имея приблизительные характеристики входных данных, можно подобрать метод, работающий оптимальным образом. Для этого необходимо знать параметры, по которым будет производиться оценка алгоритмов. Перечислим параметры оценки алгоритмов:

- *Время сортировки* – основной параметр, характеризующий быстродействие алгоритма.
- *Память* – один из параметров, который характеризуется тем, что некоторые алгоритмы сортировки требуют выделения дополнительной памяти под временное хранение данных. При оценке используемой памяти не будет учитываться место,

которое занимает исходный массив данных и независящие от входной последовательности затраты, например, на хранение кода программы.

- *Устойчивость* – это параметр, который отвечает за то, что сортировка не меняет взаимного расположения равных элементов.
- *Естественность поведения* – параметр, которой указывает на эффективность метода при обработке уже отсортированных, или частично отсортированных данных. Алгоритм ведет себя естественно, если учитывает эту характеристику входной последовательности и работает лучше.

2.3. Классификация алгоритмов сортировки

Все разнообразие алгоритмов сортировки можно классифицировать по различным признакам, например, по устойчивости, по поведению, по использованию операций сравнения, по потребности в дополнительной памяти, по потребности в знаниях о структуре данных, выходящих за рамки операции сравнения, и другие.

Наиболее подробно рассмотрим классификацию алгоритмов сортировки по сфере применения. В данном случае основные типы упорядочивания делятся следующим образом.

- *Внутренняя сортировка* – это алгоритм сортировки, который в процессе упорядочения данных использует только оперативную память (ОЗУ) компьютера. В зависимости от конкретного алгоритма и его реализации данные могут сортироваться в той же области памяти, либо использовать дополнительную оперативную память.
- *Внешняя сортировка* – это алгоритм сортировки, который при проведении упорядочения данных использует внешнюю память, как правило, жесткие диски. Внешняя сортировка разработана для обработки больших объемов данных, которые не помещаются в оперативную память. Обращение к различным носителям накладывает некоторые дополнительные ограничения: в каждый момент времени можно считать или записать только элемент, следующий за текущим; объем данных не позволяет им разместиться в ОЗУ.

Внутренняя сортировка является базовой для любого алгоритма внешней сортировки – отдельные части массива данных сортируются в оперативной памяти.

Следует отметить, что внутренняя сортировка значительно эффективней внешней, так как на обращение к оперативной памяти затрачивается намного меньше времени, чем к носителям.

2.4. Алгоритмы внутренней сортировки

Методы внутренней сортировки делятся на группы:

1. Сортировка выбором.
2. Сортировка вставками.
3. Обменная сортировка.
4. Улучшенные методы сортировки.

Рассмотрим методы сортировок каждой группы

2.4.1.Сортировка выбором.

Идея метода состоит в том, чтобы создавать отсортированную последовательность путем присоединения к ней элементы один за другим в правильном порядке.

Будем строить готовую последовательность, начиная с левого конца массива. Алгоритм состоит из *n* последовательных шагов, начиная от нулевого и заканчивая*(n-1)*-м, где *n – число сортируемых элементов*

На *i*-м шаге выбираем наименьший из элементов *a[i] ... a[n]* и меняем его местами с *a[i]*.

Алгоритм сортировки числового массива x (size- размер массива):

```
for( i=0; i < size; i++) { // i - номер текущего шага

k=i; x=a[i];

for( j=i+1; j < size; j++) // цикл выбора наименьшего элемента

if ( a[j] < x ) {

k=j; x=a[j]; // k - индекс наименьшего элемента

}

a[k] = a[i]; a[i] = x; // меняем местами наименьший с a[i]

}
```

Для нахождения наименьшего элемента из n+1 рассматриваемых алгоритм выполняет n сравнений. С учетом того, что количество рассматриваемых на очередном шаге элементов уменьшается на единицу, общее количество операций:

n + (n-1) + (n-2) + (n-3) + ... + 1 = 1/2 * (n2+n) =O(n2).

Так как число обменов всегда будет меньше числа сравнений, время сортировки растет квадратично относительно количества элементов.

Алгоритм не использует дополнительной памяти: все операции происходят "на месте".

Метод неустойчив. Даже если входная последовательность почти упорядочена, то сравнений будет столько же, то есть алгоритм ведет себя неестественно.

2.4.2. Сортировка вставками.

2.4.2.1. Сортировка простыми вставками.

Одним из наиболее простых и естественных методов внутренней сортировки является сортировка простыми включениями или вставками. Идея алгоритма очень проста. Пусть задан массив ключей *a[1], a[2], ..., a[n]*. Для каждого элемента массива, начиная со второго, производится сравнение с элементами с меньшим индексом (элемент *a[i]* последовательно сравнивается с элементами *a[i-1], a[i-2]*...) и до тех пор, пока для очередного элемента *a[j]* выполняется соотношение *a[j] > a[i], a[i]* и *a[j]* меняются местами. Если удается встретить такой элемент *a[j]* ,что *a[j] <= a[i]*, или если достигнута нижняя граница массива, производится переход к обработке элемента *a[i+1]* (пока не будет достигнута верхняя граница массива).

Аналогичным образом делаются проходы по части массива, и аналогичным же образом в его начале "вырастает" отсортированная последовательность.

Однако в сортировке «пузырьком» или «выбором» можно было четко заявить, что на *i*-м шаге элементы *a[0]...a[i]* стоят на правильных местах и никуда более не переместятся. Здесь же подобное утверждение будет более слабым: последовательность *a[0]...a[i]* упорядочена. При этом по ходу алгоритма в нее будут вставляться все новые элементы.

На *i*-м шаге работы алгоритма последовательность разделена на две части: готовую *a[0]...a[i]* и неупорядоченную *a[i+1]...a[n]*.

На следующем, каждом *(i+1)*-м шаге алгоритма берем *a[i+1]* и вставляем на нужное место в готовую часть массива. Поиск подходящего места для очередного элемента входной последовательности осуществляется путем последовательных сравнений с элементом, стоящим перед ним. В зависимости от результата сравнения элемент либо остается на текущем месте (вставка завершена), либо они меняются местами и процесс повторяется. Таким образом, сортировка выглядит так:

```
for(i=0;i<size;i++) { // цикл проходов,i- номер прохода

x=a[i]; // поиск места элемента a[i] в готовой последовательности

for ( j=i-1; j>=0 && a[j] > x; j--)

a[j+1] = a[j]; // сдвигаем элемент направо, пока не дошли

// место найдено, вставить элемент

a[j+1] = x;

}
```

Аналогично сортировке выбором, среднее, а также худшее число сравнений и пересылок оцениваются $O(n^2)$, дополнительная память при этом не используется.

Хорошим показателем сортировки является весьма естественное поведение: почти отсортированный массив будет доотсортирован очень быстро. Именно эти качества алгоритма делают его широко применимым на практике.

2.4.2.2. Сортировка Шелла.

Эта сортировка является модификацией алгоритма сортировки простыми вставками. Быстродействие достигается за счет возможности обмена элементов, далеко отстоящих друг от друга.

Общая схема сортировки Шелла

1. Происходит упорядочивание элементов n/2 пар (x_i,$x_{n/2+i}$) для 1<i<n/2.
2. Упорядочиваются элементы в n/4 группах из четырех элементов (x_i,$x_{n/4+i}$,$x_{n/2+i}$,$x_{3n/4+i}$) для 1<i<n/4.
3. Упорядочиваются элементы уже в n/8 группах из восьми элементов и т.д.

На последнем шаге упорядочиваются элементы сразу во всем массиве x_1,x_2,...,x_n.

```
for (h = n/2 ; h > 0 ; h = h/2)
    for (i = 0 ; i < n-h ; i++)
      for (j = i ; j >= 0 ; j = j - h)
       if (x[j] > x[j+h])
          Exchange (j, j+h, x);// обмен элементов j, j+h
        else j = 0;
}
```

В настоящее время неизвестна последовательность hi,hi-1,hi-2,...,h1, оптимальность которой доказана. Для достаточно больших массивов рекомендуемой считается такая последовательность, что hi+1=3hi+1, а h1=1. Начинается процесс с hm, что hm>[n/9]. Иногда значение h вычисляют проще: hi+1=hi/2,h1=1,hm=n/2. Это упрощенное соотношение для вычисления h и будем использовать далее.

Метод, предложенный Дональдом Л. Шеллом, является неустойчивой сортировкой. Эффективность метода Шелла объясняется тем, что сдвигаемые элементы быстро попадают на нужные места. Среднее время для сортировки Шелла равняется , для худшего случая оценкой является $O(n^{1.5})$.

2.4.3. Обменная сортировка

2.4.3.1. Метод пузырька.

Идея метода: шаг сортировки состоит в проходе снизу вверх по массиву. По пути просматриваются пары соседних элементов. Если элементы некоторой пары находятся в неправильном порядке, то они переставляются.

После нулевого прохода по массиву "вверху" оказывается самый "легкий" элемент - отсюда аналогия с пузырьком. Следующий проход делается до второго сверху элемента, таким образом, второй по величине элемент поднимается на правильную позицию.

Делаем проходы по все уменьшающейся нижней части массива до тех пор, пока в ней не останется только один элемент. На этом сортировка заканчивается, так как последовательность упорядочена по возрастанию.

Среднее число сравнений и обменов имеет квадратичный порядок роста: $O(n^2)$, отсюда можно заключить, что алгоритм «пузырька» очень медленный и малоэффективный. Тем не менее, он прост и его можно улучшить следующим образом:

Рассмотрим ситуацию, когда на каком-либо из проходов не произошло ни одного обмена. Это значит, что все пары расположены в правильном порядке, так что массив уже отсортирован, продолжать процесс не имеет смысла.

Приведем фрагмент кода:

```
for( i=0; i < size; i++) { // i - номер прохода

for( j = size-1; j > i; j-- ) { // внутренний цикл прохода

if ( a[j-1] > a[j] ) {

x=a[j-1]; a[j-1]=a[j]; a[j]=x;

}

}
```

Итак, первое улучшение алгоритма заключается в запоминании, факта обмена на данном проходе. Если нет - алгоритм заканчивает работу.

Процесс улучшения можно продолжить, если запоминать не только сам факт обмена, но и индекс последнего обмена *k*. Действительно, все пары соседних элементов с индексами, меньшими *k*, уже расположены в нужном порядке. Дальнейшие проходы можно заканчивать на индексе *k*, вместо того чтобы двигаться до установленной заранее верхней границы.

Качественно другое улучшение алгоритма можно получить, учтя следующее наблюдение. Хотя легкий «пузырек» снизу поднимется наверх за один проход, тяжелые «пузырьки» опускаются с минимальной скоростью.

Чтобы избежать подобного эффекта, можно менять направление следующих один за другим проходов. Получившийся алгоритм иногда называют *шейкерной сортировкой*.

2.4.3.2. Шейкерная сортировка

Приведем фрагмент кода.

```
long lb=1, ub = size-1; // границы неотсортированной части массива
do{ // проход снизу вверх
for( j=ub; j>0; j-- ) {
if ( a[j-1] > a[j] ) {
x=a[j-1]; a[j-1]=a[j]; a[j]=x;
k=j;
}
}
lb = k+1;
for(j=1;j<=ub;j++) { // проход сверху вниз
if ( a[j-1] > a[j] ) {
x=a[j-1]; a[j-1]=a[j]; a[j]=x;
k=j;
}
}
ub = k-1;
} while ( lb < ub );
```

Среднее количество сравнений, хоть и уменьшилось, но остается O(n2), в то время как число обменов не изменилось. Среднее (оно же худшее) количество операций остается квадратичным.

Дополнительная память не требуется. Поведение усовершенствованного (но не начального) метода довольно естественное, почти отсортированный массив будет отсортирован намного быстрее случайного. Сортировка «пузырьком» устойчива, шейкерная сортировка - нет.

На практике метод «пузырька», даже с улучшениями, работает слишком медленно, поэтому почти не применяется.

2.5. Улучшенные методы сортировки

2.5.1. Быстрая сортировка Хоара.

Метод быстрой сортировки был впервые описан Ч.А.Р. Хоаром в 1962 году. Быстрая сортировка – это общее название ряда алгоритмов, которые отражают различные подходы к получению критичного параметра, влияющего на производительность метода.

При общем рассмотрении алгоритма быстрой сортировки, отметим, что он основывается на последовательном разделении сортируемого набора данных на блоки меньшего размера таким образом, что между значениями разных блоков обеспечивается отношение упорядоченности (для любой пары блоков все значения одного из этих блоков не превышают значений другого блока).

Опорным (ведущим) элементом называется некоторый элемент массива, который выбирается определенный образом. С точки зрения корректности алгоритма выбор опорного элемента безразличен. С точки зрения повышения эффективности алгоритма выбираться должна медиана, но без дополнительных сведений о сортируемых данных ее обычно невозможно получить. Необходимо выбирать постоянно один и тот же элемент (например, средний или последний по положению) или выбирать элемент со случайно выбранным индексом.

Алгоритм быстрой сортировки Хоара

Пусть дан массив x[n] размерности n.

Шаг 1. Выбирается опорный элемент массива.

Шаг 2. Массив разбивается на два – левый и правый – относительно опорного элемента. Реорганизуем массив таким образом, чтобы все элементы, меньшие опорного элемента, оказались слева от него, а все элементы, большие опорного – справа от него.

Шаг 3. Далее повторяется шаг 2 для каждого из двух вновь образованных массивов. Каждый раз при повторении преобразования очередная часть массива разбивается на два меньших и т. д., пока не получится массив из двух элементов.

Быстрая сортировка стала популярной прежде всего потому, что ее нетрудно реализовать, она хорошо работает на различных видах входных данных и во многих случаях требует меньше затрат ресурсов по сравнению с другими методами сортировки.

Выберем в качестве опорного элемент, расположенный на средней позиции.

```
void Quick_Sort(int left, int right, int *x){
```

```
  int i, j, m, h;
  i = left;
  j = right;
  m = x[(i+j+1)/2];
  do {
    while (x[i] < m) i++;
    while (x[j] > m) j--;
    if (i <= j) {
      Exchange(i,j,x);//обмен i, j элементов местами
      i++;
      j--;
    }
  } while(i <= j);
  if (left < j)
    Quick_Sort (left, j, x);
  if (i < right)
    Quick_Sort (i, right, x);
}
```

Эффективность быстрой сортировки в значительной степени определяется правильностью выбора опорных (ведущих) элементов при формировании блоков. В худшем случае трудоемкость метода имеет ту же сложность, что и пузырьковая сортировка, то есть порядка $O(n^2)$. При оптимальном выборе ведущих элементов, когда разделение каждого блока происходит на равные по размеру части, трудоемкость алгоритма совпадает с быстродействием наиболее эффективных способов сортировки, то есть порядка $O(n \log n)$. В среднем случае количество операций, выполняемых алгоритмом быстрой сортировки, определяется выражением $T(n) = O(1.4n \log n)$

Быстрая сортировка является наиболее эффективным алгоритмом из всех известных методов сортировки, но все усовершенствованные методы имеют один общий недостаток – невысокую скорость работы при малых значениях n.

Рекурсивная реализация быстрой сортировки позволяет устранить этот недостаток путем включения прямого метода сортировки для частей массива с небольшим количеством элементов. Анализ вычислительной сложности таких алгоритмов показывает, что если подмассив имеет девять или менее элементов, то целесообразно использовать прямой метод (сортировку простыми вставками).

2.5.2. Сортировка слиянием.

Слияние – это объединение двух или более упорядоченных массивов в один упорядоченный.

Сортировка слиянием является одним из самых простых алгоритмов сортировки (среди быстрых алгоритмов). Особенностью этого алгоритма является то, что он работает с элементами массива преимущественно последовательно. Кроме того, сортировка слиянием является алгоритмом, который может быть эффективно использован для сортировки таких структур данных, как связные списки.

Данный алгоритм применяется тогда, когда есть возможность использовать для хранения промежуточных результатов память, сравнимую с размером исходного массива. Он построен на принципе "разделяй и властвуй". Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Далее их решения комбинируются, и получается решение исходной задачи.

Процедура слияния требует два отсортированных массива. Заметим, что массив из одного элемента по определению является отсортированным.

Алгоритм сортировки слиянием.

Шаг 1. Разбить имеющиеся элементы массива на пары и осуществить слияние элементов каждой пары, получив отсортированные цепочки длины 2 (кроме, быть может, одного элемента, для которого не нашлось пары).

Шаг 2. Разбить имеющиеся отсортированные цепочки на пары, и осуществить слияние цепочек каждой пары.

Шаг 3. Если число отсортированных цепочек больше единицы, перейти к шагу 2.

```
k = 1;
while (k < n){
  t = 0;
  s = 0;
  while (t+k < n){
    Fin1 = t+k;
    Fin2 = (t+2*k < n ? t+2*k : n);
    i = t;
    j = Fin1;
    for ( ; i < Fin1 && j < Fin2 ; s++){
```

```
        if (x[i] < x[j]) {
          tmp[s] = x[i];
          i++;
        }
        else {
          tmp[s] = x[j];
          j++;
        }
      }
      for ( ; i < Fin1; i++, s++)
        tmp[s] = x[i];
      for ( ; j < Fin2; j++, s++)
        tmp[s] = x[j];
      t = Fin2;
    }
    k *= 2;
   for (s = 0; s < t; s++)
     x[s] = tmp[s];
  }
  delete(tmp);
}
```

Недостаток алгоритма заключается в том, что он требует дополнительную память размером порядка n (для хранения вспомогательного массива). Кроме того, он не гарантирует сохранение порядка элементов с одинаковыми значениями. Но его временная сложность всегда пропорциональна O(n log n).

2.5.3.Пирамидальная сортировка

Пирамидальная сортировка является модификацией такого подхода, как сортировка выбором, с тем лишь отличием, что минимальный (или максимальный) элемент из неотсортированной последовательности выбирается за меньшее количество операций. Для такого быстрого выбора из неотсортированной последовательности строится структура, которая называется пирамидой.

Пирамида (сортирующее дерево, двоичная куча) –структура, вершиной которой является наименьший или наибольший элемент). Пирамиду можно представить в виде массива. Первый элемент пирамиды является наименьшим или наибольшим, что зависит вида сортировки.

Просеивание – это построение новой пирамиды по следующему алгоритму: новый элемент помещается в вершину структуры, далее он перемещается ("просеивается") по пути вниз на основе сравнения с дочерними элементами. Спуск завершается, если результат сравнения с дочерними элементами соответствует виду сортировки.

Последовательность чисел xi,xi+1,...,xi образует пирамиду, если для всех k=i, i+1,...,n/2 выполняются неравенства xk > x2k, xk > x2k+1 (или xk < x2k, xk < x2k+1). Элементы x2i и x2i+1 называются потомками элемента xi.

Алгоритм пирамидальной сортировки.

Шаг 1. Преобразовать массив в пирамиду.

Шаг 2. Использовать алгоритм сортировки пирамиды .

Алгоритм преобразования массива в пирамиду (построение пирамиды). Пусть дан массив x[1],x[2],...,x[n].

Шаг 1. Устанавливаем k=n/2.

Шаг 2. Перебираем элементы массива в цикле справа налево для i=k,k-1,...,1. Если неравенства xi > x2i, xi > x2i+1 не выполняются, то повторяем перестановки xi с наибольшим из потомков(x2i, x2i+1). Перестановки завершаются при выполнении неравенств xi > x2i, xi > x2i+1.

Алгоритм сортировки пирамиды.

Рассмотрим массив размерности n, который представляет пирамиду x[1],x[2],...,x[n].

Шаг 1. Переставляем элементы x[1] и x[n].

Шаг 2. Определяем n=n-1. Это эквивалентно тому, что в массиве из дальнейшего рассмотрения исключается элемент x[n].

Шаг 3. Рассматриваем массив x[1],x[2],...,x[n-1], который получается из исходного за счет исключения последнего элемента. Данный массив из-за перестановки элементов уже не является пирамидой. Но такой массив легко преобразовать в пирамиду. Это достигается повторением перестановки значения элемента из x[1] с наибольшим из потомков. Такая перестановка продолжается до тех пор, пока элемент из x[1] не окажется на месте элемента x[i] и при этом будут выполняться неравенства x[i] > x[2i], x[i] > x[2i+1]. Тем самым определяется новое место для значения первого элемента из x[1]

Шаг 4. Повторяем шаги 2, 3, 4 до тех пор, пока не получим n=1. Произвольный массив можно преобразовать в пирамиду

Теоретическое время работы этого алгоритма можно оценить, учитывая, что пирамидальная сортировка аналогична построению пирамиды методом просеивания (при этом не учитывается начальное построение пирамиды). Поэтому время работы алгоритма пирамидальной сортировки без учета времени построения пирамиды будет определяться по формуле $T1(n)=O(n \cdot \log n)$.

Построение пирамиды занимает $T2(n)=O(n)$ операций за счет того, что реальное время выполнения функции построения зависит от высоты уже созданной части пирамиды.

Тогда общее время сортировки (с учетом построения пирамиды) будет равно: $T(n)=T1(n)+T2(n)=O(n)+O(n \cdot \log n)=O(n \cdot \log n)$.

Пирамидальная сортировка не использует дополнительной памяти. Метод не является устойчивым: по ходу работы массив так "перетряхивается", что исходный порядок элементов может измениться случайным образом. Поведение неестественно: частичная упорядоченность массива никак не учитывается. Данная сортировка на почти отсортированных массивах работает также долго, выигрыш ее получается только при больших n.