Сортировка
Эффективные алгоритмы

Общий подход к ускорению сортировки

Рассмотренные ранее базовые алгоритмы сортировки работают за время $O(N^2)$. Такая сравнительно высокая сложность резко ограничивает область применения базовых методов до сортировки небольших (не более нескольких тысяч элементов) массивов. На практике для внутренней сортировки больших объёмов данных используются так называемые эффективные алгоритмы сортировки.

Одним из основных недостатков базовых методов является маленькая скорость перемещения элементов на свои места. Например, в пузырьковой сортировке за каждый обмен элемент перемещается только на одну позицию. В то же время было показано, что в среднем при сортировке элемент должен передвинуться на $N/3$ позиций. Рассматриваемые ниже алгоритмы при каждом обмене перемещают элементы на значительные расстояния, именно этим во многом объясняется их высокое быстродействие.

Сортировка Шелла

Сортировка Шелла – алгоритм, являющийся улучшением сортировки вставками. Основная идея сортировки Шелла заключается в предварительных «грубых» и быстрых проходах по массиву, которые позволяют привести массив в почти упорядоченное состояние. Полученный после нескольких проходов почти упорядоченный массив окончательно сортируется методом вставок.

На практике эта идея реализуется следующим образом. Сначала каждая группа элементов, отстоящих друг от друга на k_1 позиций, сортируется независимо методом вставок. Назовём этот процесс k_1-сортировкой. После первого прохода рассматриваются группы элементов, отстоящих друг от друга на k_2 ($k_2 < k_1$) позиций, и эти группы тоже сортируются по отдельности. Этот процесс называется k_2-сортировкой. Так проводится несколько итераций с различными постоянно уменьшающимися шагами. Завершающим этапом является традиционная сортировка вставками с шагом равным единице. Очевидно, что метод приводит к отсортированному массиву. Также очевидно, что допустима любая последовательность расстояний, лишь бы последнее из них было равно единице.

Встаёт резонный вопрос: не превысят ли возможную экономию затраты на выполнение нескольких проходов, в каждом из которых сортируются все элементы? Выигрыш сортировки Шелла заключается в том, что сортировка одной группы элементов либо имеет дело с относительно небольшим их числом, либо элементы уже частично упорядочены благодаря предыдущим итерациям. В любом случае количество перестановок сравнительно мало.

Приведём пример алгоритма для фиксированного количества проходов. Всего будет пять расстояний $h_1, h_2, \ldots, h_5$, удовлетворяющих условиям: $h_5 = 1, h_{i+1} < h_i$.

Листинг 1

```
procedure ShellSort;
```

```
const T = 5;
var
  i, j, k, m: integer;
  x: integer;
  h: array [1 .. 5] of integer;
begin
  h[1] := 31; h[2] := 15; h[3] := 7; h[4] := 3; h[5] := 1;
  for m := 1 to T do
  begin
    k := h[m];
    for i := k + 1 to N do
    begin
      x := a[i];
      j := i - k;
      while (j > k) and (x < a[j]) do
      begin
        a[j + k] := a[j];
        j := j - k;
      end;
      if (j > k) or (x >= a[j]) then
        a[j + k] := x
      else
      begin
        a[j + k] := a[j];
        a[j] := x;
      end;
    end;
  end;
end;
```

Анализ этого алгоритма – сложная математическая задача, у которой до сих пор нет полного решения. В настоящий момент неизвестно, какая последовательность расстояний даёт наилучший результат, но известно, что расстояния не должны быть кратными друг другу. Это необходимо, чтобы очередной проход не объединял две последовательности, до того никак не пересекавшиеся. Наоборот, желательно, чтобы последовательности пересекались как можно чаще. Это связано со следующей теоремой: если последовательность после k_i сортировки подвергается k_j сортировке, то она при этом остаётся k_i сортированной.

Дональд Кнут предлагает в качестве последовательно уменьшающихся расстояний использовать одну из следующих последовательностей (приведены в обратном порядке): $1,4,13,40,121, \dots$, где $h_{i-1} = 3 * h_i + 1$ или $1,3,7,15,31, \dots$, где $h_{i-1} = 2 * h_i + 1$. В последнем случае исследования показывают, что при сортировке N элементов алгоритмом Шелла временные затраты пропорциональны $O(N^{1.2})$. Этот результат значительно лучше, чем обеспечиваемая базовыми сортировками сложность $O(N^2)$, но ниже будут рассмотрены ещё более эффективные алгоритмы.

Турнирная и пирамидальная и сортировки

Базовая сортировка выбором основана на выборе наименьшего ключа среди N элементов, затем среди оставшихся $N-1$ элементов и так далее, пока не останется один элемент. Эту сортировку можно улучшить, если после каждого просмотра сохранять не только значение минимального элемента в массиве, но и другие сведения о порядке элементов. Например, $N/2$ сравнений позволяют определить меньший элемент в каждой паре, следующие $N/4$ сравнений дадут меньший элемент в каждой паре из уже найденных меньших элементов и так далее. Пользуясь такими попарными сравнениями можно за $N-1$ шагов построить соответствующее дерево, в

корне которого будет находится минимальный элемент всего сортируемого массива. Например, для массива $[100,18,94,86,25,7,2,12]$ дерево будет выглядеть как показано на Рис. 1:

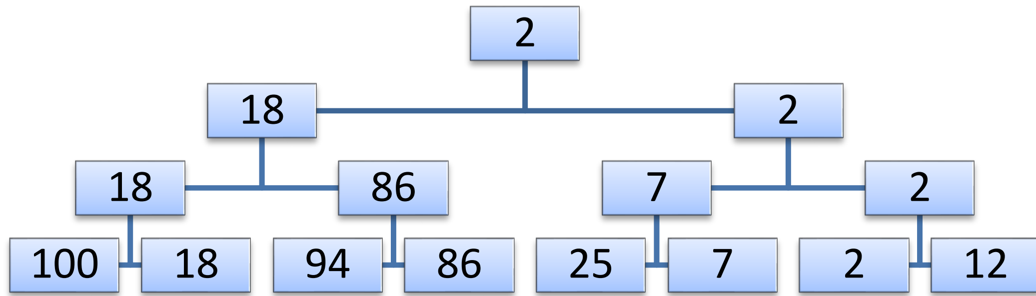

Рис. 1. Дерево попарных сравнений

Это дерево может служить основой для сортировки выбором: минимальный элемент находится в корне дерева, а значит именно он должен попасть в последовательность-приёмник. Затем, за количество шагов, равное количеству уровней в дереве ($\log_2 N$), можно удалить уже добавленный в приёмник элемент (см. Рис. 2)

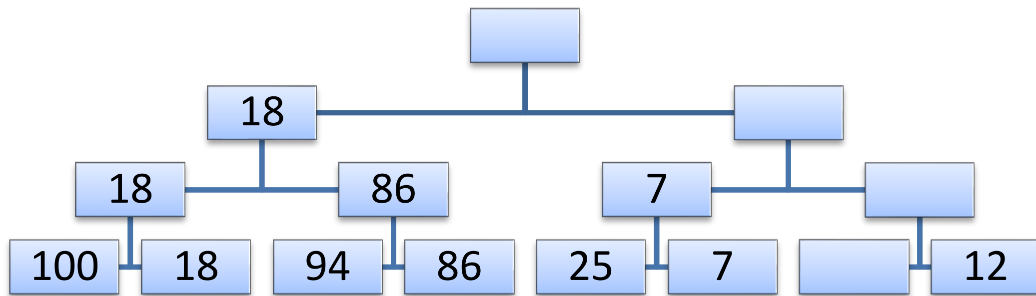

Рис. 2. Дерево попарных сравнений с удалённым минимальным элементом

И ещё за $\log_2 N$ шагов можно восстановить дерево таким образом, чтобы в корне снова был минимальный элемент. Для этого надо повторно выполнить сравнения в той ветви дерева, где был удалён элемент (см. Рис. 3)

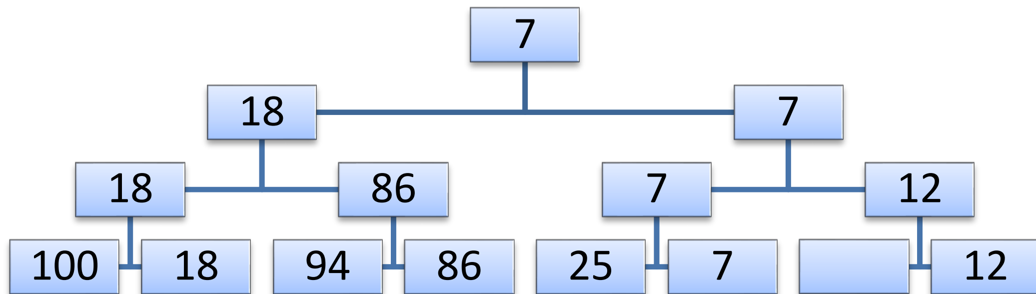

Рис. 3. Дерево попарных сравнений с удалённым минимальным элементом

После N таких шагов в дереве не останется ни одного узла с хранящимся там элементом, и процесс сортировки завершится. Этот алгоритм называется турнирной сортировкой. Оценить временную сложность такого алгоритма довольно легко: N шагов требуется на построение дерева, а затем выполняется N шагов сложностью $O(\log_2 N)$. Таким образом вся процедура требует порядка $O(N * \log_2 N)$ шагов. Это значительное улучшение не только по сравнению с простыми методами, требующими $O(N^2)$ шагов, но и с сортировкой Шелла, требующей $N^{1.2}$

шагов. Существенным недостатком рассмотренного алгоритма является необходимость в выделении дополнительной памяти $O(N)$ под хранение дерева.

Проблема расхода памяти была решена в алгоритме пирамидальной сортировки, который является существенным улучшением турнирной сортировки. Для рассмотрения пирамидальной сортировки надо дать определение структуре данных, называемой пирамидой. Пирамидой (heap) называется массив, для которого выполняются следующие условия: $A[i] \leq A[2i]$ и $A[i] < A[2i+1]$ $i \in [1..\frac{R}{2}]$. Любую пирамиду можно представить в виде дерева, в вершине которого минимальный элемент массива. Дерево, соответствующее пирамиде $[6,9,12,23,48,15,27]$, показано на Рис. 4

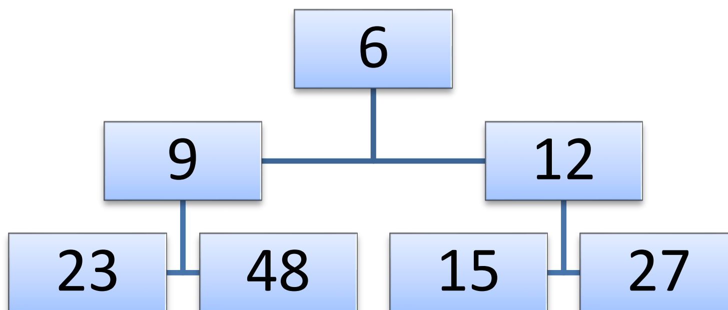

Рис. 4. Дерево попарных сравнений с удалённым минимальным элементом

Алгоритм добавления нового элемента в пирамиду сравнительно прост: сначала добавляемый элемент ставится на вершину пирамиды и, если он больше хотя бы одного своего потомка, просеивается вниз, меняясь местами с наименьшим из своих потомков. Поясним процедуру добавления элемента в пирамиду на конкретном примере: добавим число 19. Массив будет выглядеть следующим образом: $[19,6,9,12,23,48,15,27]$, соответствующая пирамида приведена на Рис. 5.

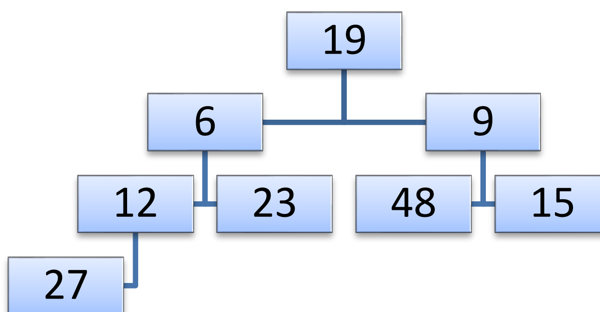

Рис. 5. Дерево попарных сравнений с удалённым минимальным элементом

Новый элемент сначала поменяется местами с элементом 6 (см. Рис. 6), а затем с элементом 12 (см. Рис. 7). После этого дальнейшее просеивание не выполняется, так как все потомки элемента 19 больше него.

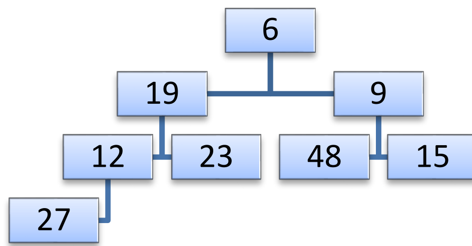

Рис. 6. Процесс просеивания элемента 19 (шаг 1)

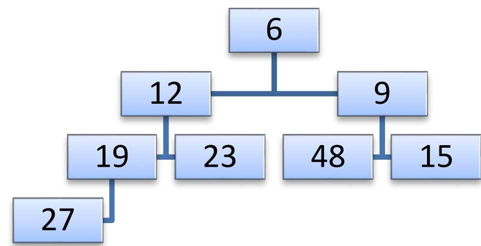

Рис. 7. Процесс просеивания элемента 19 (шаг 2)

Эффективная программная реализация основывается на процедуре `sift` (просеивание). Если задан массив $A[1] \dots A[N]$, то его элементы с номерами от $\frac{N}{2}$ до N уже образуют пирамиду (т.к. в этих элементах нет таких пар i, j, чтобы $j = 2 * i$ или $j = 2 * i + 1$). Важно понимать, что эти элементы не организуют пирамиду сами по себе, а составляют нижний уровень пирамиды в общем массиве. Затем начинается процесс расширения пирамиды. Причём за один шаг в неё будет включаться только один элемент, и этот элемент будет ставиться на своё место с помощью процедуры просеивания, код которой приведён ниже.

Листинг 2

```
procedure sift(L,R: integer);
var
i,j: integer;
x: integer;
begin
  i:=L;
  j:=2*i;
  x:=a[i];
  if (j<R) and (a[j]<a[j+1]) then
    j:=j+1;
  while (j<=R) and (x<a[j]) do
  begin
    a[i]:=a[j];
    i:=j;
    j:=2*j;
    if (j<R) and (a[j]<a[j+1]) then
      j:=j+1;
  end;
  a[i]:=x;
end;
```

Процесс получения пирамиды можно описать следующим образом:

Листинг 3

```
L:=(N div 2)+1;
R:=N;
while L>1 do
begin
  L:=L-1;
  sift(L,R);
end;
```

Пирамида, которую мы после этого получим, характеризуется тем, что в её вершине будет храниться наибольший (а не наименьший) ключ. Для того, чтобы добиться полной упорядоченности нужно выполнить ещё N просеиваний и после каждого из них снимать с вершины очередной (максимальный) элемент. Существует довольно красивое решение, позволяющее хранить снимаемый с вершины элемент непосредственно в массиве: нам

необходимо взять последний элемент пирамиды (будем называть его x), записать элемент с вершины пирамиды в позицию, освободившуюся из под x, а элемент x поставить сначала на вершину пирамиды, а затем и в правильную позицию очередным просеиванием. Этот процесс можно описать следующим образом:

Листинг 4

```
while R>1 do
begin
  x:=a[1];
  a[1]:=a[R];
  a[R]:=x;
  R:=R-1;
  sift(L,R);
end;
```

Таким образом, алгоритм пирамидальной сортировки можно представить следующим кодом:

Листинг 5

```
procedure HeapSort;
  procedure sift(L,R: integer);
  var
  i,j: integer;
  x: integer;
  begin
    i:=L;
    j:=2*i;
    x:=a[i];
    if (j<R) and (a[j]<a[j+1]) then
      j:=j+1;
    while (j<=R) and (x<a[j]) do
    begin
      a[i]:=a[j];
      i:=j;
      j:=2*j;
      if (j<R) and (a[j]<a[j+1]) then
        j:=j+1;
    end;
    a[i]:=x;
  end;
var
L,R: integer;
x: integer;
begin
  L:=(N div 2)+1;
  R:=N;
  while L>1 do
  begin
    L:=L-1;
    sift(L,R);
  end;
  while R>1 do
  begin
    x:=a[1];
    a[1]:=a[R];
    a[R]:=x;
    R:=R-1;
    sift(L,R);
```

```
  end;
end;
```

В этом алгоритме большие элементы сначала просеиваются влево, и только потом занимают окончательные позиции в правой части отсортированного массива. Из-за этого может показаться, что эффективность пирамидальной сортировки не очень высока, но на самом деле это не так. В худшем случае фаза создания пирамиды требует $O(N)$ шагов просеивания, затем фаза сортировки требует ещё $O(N)$ просеиваний с не более чем $O(\log_2 N)$ перестановок. Получается, что даже в наихудшем случае пирамидальная сортировка требует $O(N * \log_2 N)$ перестановок. Это свойство является важнейшей отличительной особенностью данного алгоритма.

На практике для сортировки небольшого числа элементов лучше воспользоваться другими алгоритмами (например, сортировкой Шелла), но для больших массивов эта сортировка очень эффективна.

Быстрая сортировка

Сортировка Шелла и пирамидальная сортировка – улучшения алгоритмов вставки и выбора. Аналогично, быстрая сортировка – улучшение пузырьковой сортировки. Что интересно, если пузырьковая сортировка была самым медленным из базовых алгоритмов, то её усовершенствованная версия – один из лучших методов внутренней сортировки.

Как уже упоминалось выше, для построения эффективного алгоритма необходимо обменивать максимально удалённые между собой элементы. Рассмотрим для начала вырожденный случай: задан массив из N элементов и все элементы расположены по убыванию. Тогда, чтобы отсортировать элементы по возрастанию, достаточно выполнить всего лишь $N/2$ перестановок, сначала обменяв крайний левый и крайний правый элементы, а потом постепенно продвигаясь внутрь массива с обеих сторон. Разумеется, этот метод работает, только если элементы в массиве упорядочены по убыванию. Но, тем не менее, именно на этой идее построена быстрая сортировка.

Попробуем реализовать такой алгоритм: случайно выберем любой элемент из массива (назовём его x). Будем просматривать массив слева, пока не найдём элемент $A[i] > x$, а затем справа, пока не найдём элемент $A[j] < x$. Выполним обмен двух найденных элементов и будем продолжать такой процесс, пока оба просмотра не встретятся где-то в середине массива. В результате получим массив, разделённый точкой встречи индексов на две части: левую, с ключами, меньшими или равными x, и правую с ключами, большими или равными x. Сформулируем процесс разделения в виде процедуры:

Листинг 6

```
procedure partition;
var
i,j: integer;
w,x: integer;
begin
  i:=1;
  j:=n;
  // выбрать наугад значение x
  REPEAT
    while a[i]<x do
      i:=i+1;
```

```
    while x<a[j] do
      j:=j-1;
    if i<=j then
    begin
      w:=a[i];
      a[i]:=a[j];
      a[j]:=w;
      i:=i+1;
      j:=j-1;
    end;
  UNTIL i>j;
end;
```

После выполнения процедуры разделения массив стал чуть более упорядоченным, но наша цель – отсортировать его полностью. На самом деле, от процедуры разделения до сортировки всего один небольшой шаг: после разделения массива нужно применить тот же процесс разделения к обеим получившимся частям, затем к частям частей и т.д., пока каждая часть не будет состоять из одного элемента.

Листинг 7

```
procedure sort(L,R: integer);
var
i,j: integer;
w,x: integer;
begin
  i:=L;
  j:=R;
  x:=a[(L+R) div 2];
  REPEAT
    while a[i]<x do
      i:=i+1;
    while x<a[j] do
      j:=j-1;
    if i<=j then
    begin
      w:=a[i];
      a[i]:=a[j];
      a[j]:=w;
      i:=i+1;
      j:=j-1;
    end;
  UNTIL i>j;
  if L<j then
    sort(L,j);
  if i<R then
    sort(i,R);
end;
procedure QuickSort;
begin
  sort(1,N);
end;
```

Чтобы изучить эффективность быстрой сортировки, нужно сначала исследовать поведение процесса разделения. После выбора разделяющего значения просматривается весь массив, поэтому выполнится точно N сравнений. Если повезёт и в качестве границы всегда будет выбираться медиана массива, то каждая итерация разбивает массив строго пополам, и число

необходимых проходов равно $\log_2 N$. Тогда полное число сравнений равно $N * \log_2 N$, а полное число обменов оценивается как $O(N * \log_2 N)$.

Конечно, нельзя ожидать, что с выбором разделяющего элемента всегда будет так везти. Строго говоря, вероятность этого $1/N$. Но показано, что ожидаемая эффективность быстрой сортировки хуже оптимальной только на множитель $2 * ln(2)$.

Даже у алгоритма быстрой сортировки есть недостатки. Остаётся проблема наихудшего случая. Предположим, что каждый раз в качестве разделяющего элемента выбирается наибольшее значение в разделяемом сегменте. Тогда каждый шаг будет разделять фрагмент из N элементов на две последовательности из 1 и $N - 1$ элементов соответственно. Очевидно, что в наихудшем случае поведение оценивается величиной $O(N^2)$. Чтобы избежать наихудшего случая, было предложено выбирать разделяющий элемент как медиану из трёх значений разделяемого сегмента. Такое дополнение не ухудшает алгоритм в общем случае, но сильно улучшит его поведение в наихудшем случае.

Сравнение эффективных алгоритмов внутренней сортировки

В Табл. 1 показана зависимость времени сортировки массива случайных чисел от размера массива и используемого алгоритма. Все измерения производились на обычном персональном компьютере.

Табл. 1. Зависимость времени сортировки от размера случайного массива и используемого алгоритма

Размер массива	**Сортировка**		
	Шелла	**Пирамидальная**	**Быстрая**
256	0,001	<мс	<мс
512	0,001	<мс	<мс
1024	0,001	<мс	<мс
2048	0,001	<мс	0,001
4096	0,001	0,003	0,001
8192	0,001	0,004	0,001
16384	0,004	0,003	0,002
32768	0,008	0,007	0,005
65536	0,021	0,022	0,01
131072	0,037	0,032	0,021
262144	0,089	0,087	0,052
524288	0,218	0,203	0,102
1048576	0,567	0,558	0,207
2097152	1,128	0,967	0,472
4194304	2,502	2,109	0,942
8388608	5,574	5,076	2,03
16777216	12,833	11,238	4,19
33554432	30,066	25,894	8,445
67108864	70,343	57,031	17,346

Полученные результаты позволяют убедиться, что рассмотренные эффективнее алгоритмы сортировки работают значительно быстрее, чем рассмотренные на прошлой лекции базовые методы.

Кроме того, Табл. 1 позволяет сделать некоторые выводы об относительной скорости работы изученных методов. Быстрейшим со значительным отрывом оказался алгоритм быстрой сортировки. На втором месте находится пирамидальная сортировка, и замыкает список эффективных сортировок сортировка Шелла.

Помимо сравнения производительности на случайных данных довольно интересно сравнить эффективность сортировок на уже упорядоченных массивах и массивах, упорядоченных в обратном порядке.

Табл. 2. Зависимость времени сортировки от размера упорядоченного массива и используемого алгоритма

Размер массива	Сортировка		
	Шелла	Пирамидальная	Быстрая
4096	<мс	0,001	<мс
8192	<мс	0,001	<мс
16384	0,001	0,003	0,001
32768	0,002	0,005	0,002
65536	0,003	0,018	0,006
131072	0,006	0,04	0,004
262144	0,014	0,046	0,01
524288	0,028	0,15	0,021
1048576	0,122	0,298	0,045
2097152	0,143	0,547	0,123
4194304	0,292	0,981	0,205
8388608	0,662	2,333	0,53
16777216	1,362	4,477	1,348
33554432	2,894	10,007	3,121
67108864	6,262	21,544	6,06

Табл. 3. Зависимость времени сортировки от размера упорядоченного в обратном порядке массива и используемого алгоритма

Размер массива	Сортировка		
	Шелла	Пирамидальная	Быстрая
1024	<мс	0,001	<мс
2048	0,001	0,001	<мс
4096	0,001	0,001	<мс
8192	0,001	0,001	<мс
16384	0,001	0,002	<мс
32768	0,002	0,005	0,001
65536	0,005	0,01	0,002
131072	0,009	0,024	0,006
262144	0,021	0,063	0,027
524288	0,068	0,217	0,023
1048576	0,1	0,22	0,047
2097152	0,199	0,432	0,135
4194304	0,446	0,911	0,217
8388608	0,967	2,244	0,608
16777216	2,315	5,004	1,262
33554432	4,381	9,238	2,946
67108864	10,198	21,69	6,874