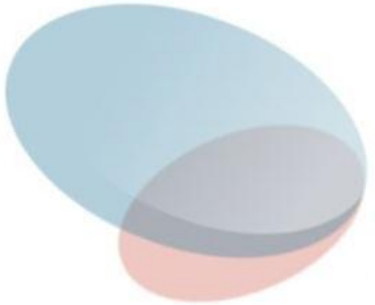


Алгоритмы сортировки
Лекция 3

1.1 Сортировка методом Шелла

Сортировка Шелла (англ. Shell sort) — разработана Дональдом Л. Шеллом в 1959 году. **Идея алгоритма состоит в сравнении элементов, стоящих не только рядом, но и на расстоянии друг от друга.** Иными словами это сортировка вставками с предварительными «грубыми» проходами.

При сортировке Шелла **сначала сравниваются и сортируются между собой ключи, отстоящие один от другого на некотором расстоянии h. Полученная последовательность элементов в списке называется отсортированной по h.** После этого процедура повторяется **для некоторых меньших значений h**, а **завершается** сортировка Шелла упорядочиванием элементов **при h = 1** (то есть, обычной сортировкой вставками).

Сортировка Шелла во многих случаях медленнее, чем быстрая сортировка, но она имеет ряд **преимуществ**:

- отсутствие потребности в памяти под стек;
- отсутствие деградации при неудачных наборах данных – в этом случае быстрая сортировка (qsort) легко деградирует до $O(n^2)$, что хуже, чем худшее гарантированное время для сортировки Шелла.

1.2 Сортировка методом Шелла

Пример

Пусть дан список A = (32,95,16,82,24,66,35,19,75,54,40,43,93,68), в качестве значений h выбраны 5, 3, 1.

На первом шаге сортируются подсписки A, составленные из всех элементов A, различающихся на 5 позиций, то есть подсписки A5,1 = (32,66,40), A5,2 = (95,35,43), A5,3 = (16,19,93), A5,4 = (82,75,68), A5,5 = (24,54).

В полученном списке на втором шаге вновь сортируются подсписки из отстоящих на 3 позиции элементов.

Процесс завершается обычной сортировкой вставками получившегося списка.

Исходный массив	32 95 16 82 24 66 35 19 75 54 40 43 93 68	
После сортировки с шагом 5	32 35 16 68 24 40 43 19 75 54 66 95 93 82	6 обменов
После сортировки с шагом 3	32 19 16 43 24 40 54 35 75 68 66 95 93 82	5 обменов
После сортировки с шагом 1	16 19 24 32 35 40 43 54 66 68 75 82 93 95	15 обменов

1.3 Сортировка методом Шелла

Пример. Сортировка Шелла

СОРТИРОВКА ШЕЛЛА СО СМЕЩЕНИЯМИ 8, 4, 2, 1

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

Сортировка через 8:

503 087 154 061 612 170 765 275 653 426 512 509 908 677 897 703

Сортировка через 4:

503 087 154 061 612 170 512 275 653 426 765 509 908 677 897 703

Сортировка через 2:

154 061 503 087 512 170 612 275 653 426 765 509 897 677 908 703

Сортировка через 1:

061 087 154 170 275 426 503 509 512 612 653 677 703 765 897 908

1.4 Сортировка методом Шелла

Пример. Сортировка Шелла

СОРТИРОВКА ШЕЛЛА СО СМЕЩЕНИЯМИ 7, 5, 3, 1

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

Сортировка через 7:

275 087 426 061 509 170 677 503 653 512 154 908 612 897 765 703

Сортировка через 5:

154 087 426 061 509 170 677 503 653 512 275 908 612 897 765 703

Сортировка через 3:

061 087 170 154 275 426 512 503 653 612 509 765 677 897 908 703

Сортировка через 1:

061 087 154 170 275 426 503 509 512 612 653 677 703 765 897 908

1.5 Сортировка методом Шелла

Среднее время работы алгоритма зависит от длин промежутков h, на которых будут находиться сортируемые элементы исходного массива ёмкостью n на каждом шаге алгоритма. Существует несколько подходов к выбору этих значений.

1. Первоначально **Шеллом** использовалась **последовательность длин промежутков 1, 2, 4, 8, 16, 32 и т.д**. В обратном порядке она вычисляется по формулам: $h_1=n/2$, $h_i=h_{i-1}/2$, $h_0=1$, **В худшем случае сложность алгоритма составит O(n^2)**.

1.6 Сортировка методом Шелла

2. Гораздо **лучший вариант** предложил **Роберт Седжвик**. Его последовательность имеет вид (самая быстрая из известных на сегодня)

$$inc[s] = \begin{cases} 9*2^s - 9*2^{s/2} + 1, \text{ если } s \text{ четно} \\ 8*2^s - 6*2^{(s+1)/2} + 1, \text{ если } s \text{ нечетно} \end{cases}$$

При использовании таких приращений **среднее количество операций имеет порядок O(n^(7/6)), в худшем случае - порядок O(n^(4/3))**.

Последовательность вычисляется в порядке, противоположном используемому: inc[0] = 1, inc[1] = 5, ...19, 41, 109 и т.д. Формула дает сначала меньшие числа, затем все большие, при этом расстояние между сортируемыми элементами, наоборот, должно уменьшаться. Поэтому массив приращений inc вычисляется перед запуском собственно сортировки до максимального расстояния между элементами, которое будет первым шагом в сортировке Шелла. Потом его значения используются в обратном порядке.

При использовании формулы Р. Седжвика следует остановиться на значении inc[s-1], если 3*inc[s] > n;

1.7 Сортировка методом Шелла

3. **Хиббардом** предложена последовательность: все значения (2^i-1)/2<=n/2, i ∈ N; такая последовательность шагов приводит к алгоритму со сложностью **O(n^3 / 2)**;
4. **Праттом** предложена последовательность: все значения 2^i 3^j <= n/2, i, j ∈ N; в таком случае сложность алгоритма составляет **O(n(log n)^2);**
5. **Эмпирическая последовательность Марцина Циура** (последовательность A102549 в OEIS **(On-Line Encyclopedia of Integer Sequences)):** h ∈ {1, 4, 10, 23, 57, 132, 301, 701, 1750}; является одной из лучших, для сортировки массива ёмкостью приблизительно до 4000 элементов;
6. **Эмпирическая последовательность, основанная на числах Фибоначчи**: h ∈ {Fn};
7. Для достаточно больших массивов рекомендуемой считается **последовательность, предложенная в 1969 году Дональдом Кнутом**: 1, 4, 13, 40, 121 и т.д. (каждое последующее значение на единицу больше, чем утроенное предыдущее $h_{i+1} = 3h_i+1$, а $h_1 = 1$). Начинается процесс с h_m, что $h_m \geq [n/9]$. Для списков средних размеров Кнут оценил быстродействие для среднего случая как O(n^(5/4)), а для худшего случая как O(n^(3/2)).

В настоящее время не известна последовательность h_i, h_{i-1}, h_{i-2}, ...,h_1, оптимальность которой доказана.

1.8 Сортировка методом Шелла

```
procedure ShellSort(n: integer; var A: intarray);
{Процедура сортировки Шелла с последовательностью Шелла}
Var  h, i, j, Tmp: integer;
Begin
   {Вычисляем величину h}
   h := n div 2;
   {Собственно сортировка}
   while h > 0 do begin
      for i := 1 to n-h do begin
         j := i;
         while j > 0 do begin
         {Сравниваем элементы, отстоящие друг от друга}
         {на расстояние, кратное h}
         if A[j] > A[j+h] then begin {Меняем элементы}
           Tmp := A[j+h]; A[j+h] := A[j]; A[j] := Tmp;
            j := j – h;
         end else j := 0;{Досрочно завершаем цикл с параметром j}
      end;
    end;
    {Уменьшаем размер серии}
    h := h div 2;
  end;
end;
```

2.1 Сортировка методом прочесывания

Сортировка расчёской или методом прочесывания (англ. comb sort) – это довольно упрощённый алгоритм сортировки, изначально спроектированный Влодзимежом Добосиевичем в 1980 г. Позднее он был переоткрыт и популяризован в статье Стефана Лэйси и Ричарда Бокса в журнале Byte Magazine в апреле 1991 г.

Сортировка расчёской улучшает сортировку пузырьком, и конкурирует с алгоритмами, подобными быстрой сортировке. Фактически **он использует пузырьковую сортировку таким же образом, как сортировка Шелла использует сортировку методом вставок**.

В сортировке пузырьком, когда сравниваются два элемента, промежуток (расстояние между элементами) равен 1. **Основная идея сортировки расчёской в том, что этот промежуток может быть гораздо больше, чем единица** (сортировка Шелла также основана на этой идее, но она является модификацией сортировки вставками, а не сортировки пузырьком).

2.2 Сортировка методом прочесывания

Алгоритм

Как и в методе Шелла, вначале выбирается последовательность расстояний $h=(h_1, h_2, h_3, \ldots, h_m)$, в которой $h_i > h_{i+1}$, например, для массива из 13 элементов, можно выбрать 8, 6, 4, 3, 2, 1.

На первом шаге при $h_1=8$ сравниваются и, в случае необходимости, переставляются местами элементы с номерами j и $j+h_1$, то есть 1-й и 9-й элементы, затем – 2-й и 10-й, 3-й и 11-й, 4-й и 12-й, 5-й и 13-й и т.д. до конца массива, то есть элементы, отстоящие друг от друга на 8 позиций.

На следующем шаге сравниваются и переставляются пары элементов с номерами j и $j+h_2$: (1, 7), (2, 8), (3, 9), (4, 10), (5,11), (6, 12), (7, 13) и т. д., то есть элементы, отстоящие друг от друга на 6 позиций.

Далее выполняется проход по массиву для элементов, отстоящих друг от друга на 4 позиции, затем на 3 и 2 позиции.

На последнем шаге выполняется стандартная пузырьковая сортировка, которую можно рассматривать как продолжение предыдущего алгоритма для соседних элементов.

Алгоритм сортировки методом прочесывания требует всего два цикла: один для уменьшения размера “прыжков” – расстояний между элементами, второй – для выполнения разновидности пузырьковой сортировки.

2.3 Сортировка методом прочесывания

Выбор длины прыжка

Разработчики алгоритма **эмпирическим** путем пришли к выводу, что **значение каждого следующего расстояния прыжка должно быть получено в результате деления предыдущего на 1.3**. Этот коэффициент дает наилучшее время сортировки.

Эмпирическим путем также было установлено, что **значения расстояний 9 и 10** между сравниваемыми элементами **являются неоптимальными**, и если они присутствуют в последовательности, их лучше **заменять на 11**. В этом случае сортировка будет выполняться гораздо быстрее.

В качестве **начального расстояния** берется **целая часть от деления количества элементов n на 1.3.**

В худшем случае сложность алгоритма составит O(n^2).

2.4 Сортировка методом прочесывания

```
procedure TDCombSort(aList : TList;
                     aFirst : integer;
                     aLast : integer;
                     aCompare : TtdCompareFunc);
var
  i, j : integer;
  Temp : pointer;
  Done : boolean;
  Gap : integer;
begin
  TDValidateListRange(aList, aFirst, aLast, 'TDCombSort');
  {начать с расстояния, равного количеству элементов}
  Gap := succ(aLast - aFirst);
  repeat
    {предположить, что сортировка будет выполнена на этом проходе}
    Done := true;
    {calculate the new gap}
    Gap := (longint(Gap) * 10) div 13; {Gap := Trunc(Gap / 1.3);}
    if (Gap < 1) then
      Gap := 1
    else if (Gap = 9) or (Gap = 10) then
      Gap := 11;
```

2.5 Сортировка методом прочесывания

```
    {упорядочить два элемента, отстоящих друг от друга на Gap элементов}
    for i := aFirst to (aLast - Gap) do begin
      j := i + Gap;
      if (aCompare(aList.List^[j], aList.List^[i]) < 0) then begin
        {поменять местами элементы с индексами j и (j-Gap)}
        Temp := aList.List^[j];
        aList.List^[j] := aList.List^[i];
        aList.List^[i] := Temp;
        {была выполнена перестановка, следовательно, сортировка не завершена}
        Done := false;
      end;

  until Done and (Gap = 1);
end;
```

3.1 Сортировка деревом

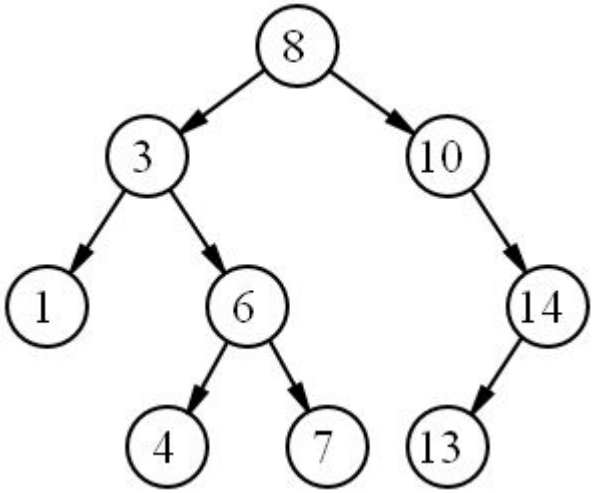
Сортировка с помощью двоичного дерева (сортировка двоичным деревом, сортировка деревом, древесная сортировка, сортировка с помощью бинарного дерева, англ. tree sort) – универсальный **алгоритм** сортировки, заключающийся

1 - в **построении двоичного дерева поиска** по ключам массива (списка),

2 - **с последующей сборкой результирующего массива путём симметричного обхода узлов построенного дерева в необходимом порядке следования ключей**.

Данная сортировка является оптимальной при получении данных путём непосредственного чтения из потока (например, из файла или консоли).

Относится к алгоритмам сортировки вставками.



3.2 Сортировка деревом

Эффективность

Процедура **добавления объекта в бинарное дерево** имеет среднюю алгоритмическую **сложность порядка O(log(n)).**

Соответственно, **для n объектов** сложность будет составлять **O(n log(n))**, что относит сортировку с помощью двоичного дерева к группе «**быстрых сортировок**».

Однако, **сложность добавления объекта в разбалансированное дерево** может достигать **O(n)**, что может привести к **общей сложности порядка $O(n^2)$**.

При физическом развёртывании древовидной структуры в памяти **требуется не менее чем 4n ячеек дополнительной памяти** (каждый узел должен содержать ссылки на элемент исходного массива, на родительский элемент, на левый и правый лист).

Сравнение алгоритмов сортировки

Алгоритмы неустойчивой сортировки

- **Сортировка выбором** (Selection sort) — Сложность алгоритма: O(n^2); поиск наименьшего или наибольшего элемента и помещения его в начало или конец упорядоченного списка
- **Сортировка Шелла** (Shell sort) — Сложность алгоритма: O(n log^2 n); попытка улучшить сортировку вставками
- **Сортировка расчёской** (Comb sort) — Сложность алгоритма: O(n log n)
- **Пирамидальная сортировка** (Сортировка кучи, Heapsort) — Сложность алгоритма: O(n log n); превращаем список в кучу, берём наибольший элемент и добавляем его в конец списка
- **Быстрая сортировка** (Quicksort), в варианте с минимальными затратами памяти — Сложность алгоритма: O(n log n) — среднее время, O(n^2) — худший случай; широко известен как быстрейший из известных для упорядочения больших случайных списков; с разбиением исходного набора данных на две половины так, что любой элемент первой половины упорядочен относительно любого элемента второй половины; затем алгоритм применяется рекурсивно к каждой половине. При использовании O(n) дополнительной памяти, можно сделать сортировку устойчивой.
- **Поразрядная (распределяющая) сортировка** — Сложность алгоритма: O(n·T); требуется O(T) дополнительной памяти.

Сравнение алгоритмов сортировки

Алгоритмы устойчивой сортировки

- **Сортировка пузырьком** (англ. Bubble sort) — сложность алгоритма: O(n^2); для каждой пары индексов производится обмен, если элементы расположены не по порядку.
- **Сортировка перемешиванием** (Шейкерная, Cocktail sort, bidirectional bubble sort) — Сложность алгоритма: O(n^2)
- **Сортировка вставками** (Insertion sort) — Сложность алгоритма: O(n^2); определяем где текущий элемент должен находиться в упорядоченном списке и вставляем его туда
- **Блочная сортировка** (Корзинная сортировка, Bucket sort) — Сложность алгоритма: O(n); требуется O(k) дополнительной памяти и знание о природе сортируемых данных, выходящее за рамки функций "переставить" и "сравнить".
- **Сортировка подсчётом** (Counting sort) — Сложность алгоритма: O(n+k); требуется O(n+k) дополнительной памяти.
- **Сортировка слиянием** (Merge sort) — Сложность алгоритма: O(n log n); требуется O(n) дополнительной памяти; выстраиваем первую и вторую половину списка отдельно, а затем — сливаем упорядоченные списки
- **Сортировка с помощью двоичного дерева** (англ. Tree sort) — Сложность алгоритма: O(n log n); требуется O(n) дополнительной памяти

Сравнение алгоритмов сортировки

Алгоритмы, не основанные **на сравнениях**

- **Блочная сортировка** (Корзинная сортировка, Bucket sort)
- **Лексикографическая или поразрядная сортировка** (Radix sort)
- **Сортировка подсчётом** (Counting sort)

Сравнение алгоритмов сортировки

Оценка сложности работы основных алгоритмов внутренней сортировки

Метод сортировки	Характеристики			
	T_{max}	T_{mid}	T_{min}	V_{max}
Подсчетом	$O(n*\log n)$		$O(n)$	$O(n)$
Вставками	$O(n^2)$		$O(n)$	$O(1)$
Шелла	$O(n^2)$	$O(n^{1,25})$	$O(n)$	$O(1)$
Выбором	$O(n^2)$			$O(1)$
Древесная	$O(n*\log n)$			$O(1)$
Пузырьковая	$O(n^2)$		$O(n)$	$O(1)$
Быстрая	$O(n^2)$	$O(n*\log n)$		$O(\log n)$
Слиянием	$O(n*\log n)$			$O(n)$
Распределением	$O(n)$			$O(n)$
Прочесыванием	$O(n^2)$			$O(1)$
Пирамидальная	$O(n*\log n)$			$O(n)$
Блочная	$O(n)$			$O(n)$

Сравнение алгоритмов сортировки

Правила выбора алгоритма сортировки, обеспечивающего максимальную производительность:

- если нужно **быстро реализовать** алгоритм сортировки, используйте **быструю сортировку**, а затем при необходимости поменяйте алгоритм;
- если **более 99 процентов списка уже отсортировано**, используйте **пузырьковую сортировку**;
- если **список очень мал** (100 или менее элементов), используйте **сортировку выбором**;
- если значения находятся **в связном списке**, используйте **блочную сортировку** на основе связного списка;
- если **элементы** в списке — **целые числа**, **разброс значений которых невелик** (до нескольких тысяч), используйте **сортировку подсчетом**;
- если **значения лежат в широком диапазоне и не являются целыми числами**, используйте **блочную сортировку** на основе массива;
- если вы **не можете тратить дополнительную память**, которая требуется для блочной сортировки, используйте **быструю сортировку**.

Сравнение алгоритмов сортировки

Преимущества и недостатки алгоритмов сортировки

Алгоритм	Преимущества	Недостатки
Сортировка вставкой	Очень прост Быстро сортирует небольшие списки	Очень медленно работает с большими списками
Сортировка вставкой на основе связанного списка	Прост Быстро сортирует небольшие списки Перемещает не данные, а указатели	Медленно работает с большими списками
Сортировка выбором	Очень прост Быстро сортирует небольшие списки	Медленно работает с большими списками
Пузырьковая сортировка	Быстро работает для почти отсортированных списков	Медленно работает во всех остальных случаях

Сравнение алгоритмов сортировки

Преимущества и недостатки алгоритмов сортировки (окончание)

Алгоритм	Преимущества	Недостатки
Быстрая сортировка	Быстро сортирует большие списки	Работает некорректно при большом количестве одинаковых значений
Сортировка слиянием	Быстро сортирует большие списки	Требует пространства под временные значения Работает медленнее, чем быстрая сортировка
Пирамидальная сортировка	Быстро сортирует большие списки Не требует пространства для временных значений	Работает медленнее, чем сортировка слиянием
Сортировка подсчетом	Очень быстро работает, если разброс входных значений невелик	Работает медленно, если диапазон значений > log(N) Требует дополнительной памяти Работает только с данными целого типа
Блочная сортировка	Очень быстро работает, если данные распределены равномерно Работает с данными, диапазон значений которых велик Работает с данными любого типа	Медленнее, чем сортировка подсчетом