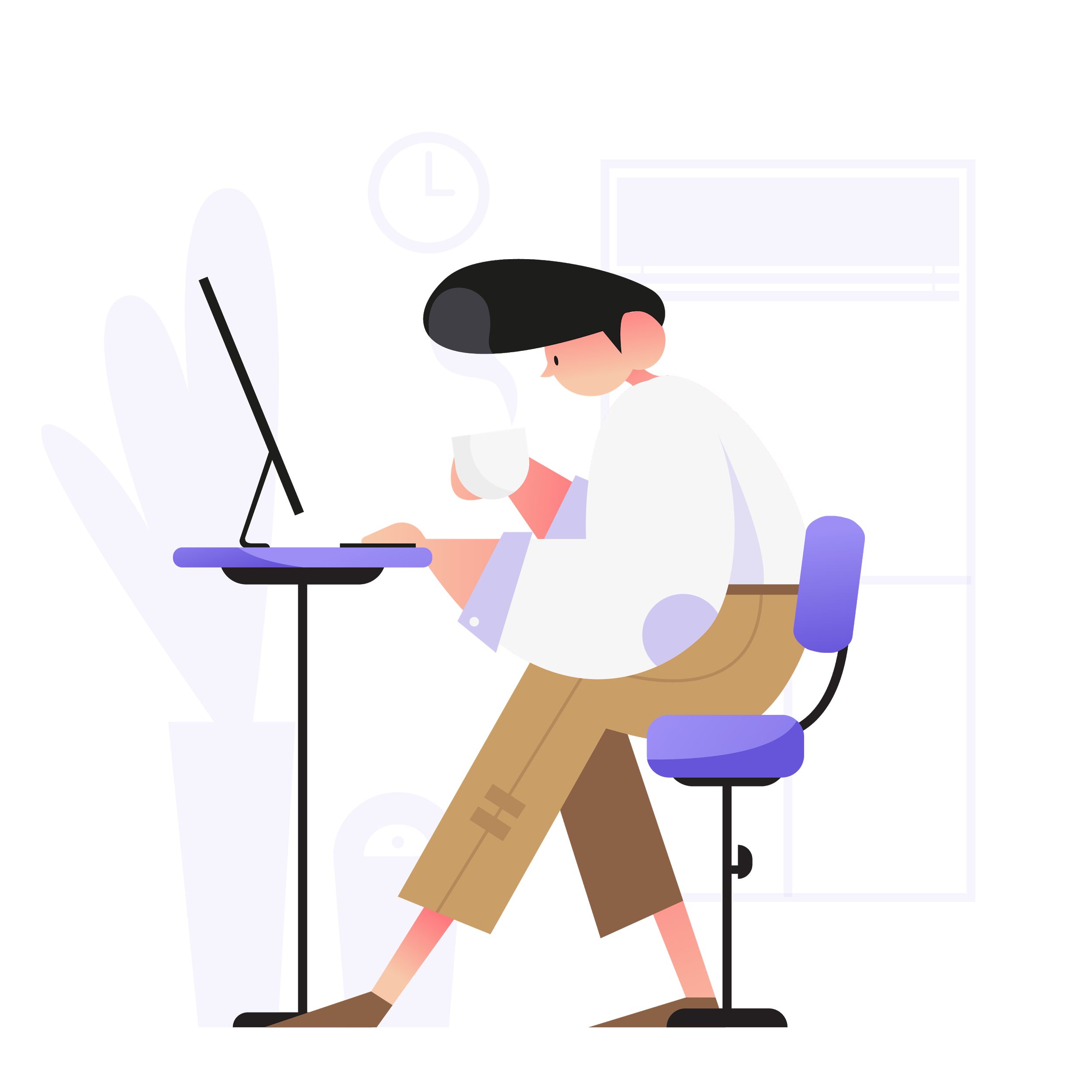
Алгоритмы и структуры данных на С#

Алгоритмы сортировки

Netcore 3.1



# На этом уроке

1. Сортировка Хоара (Quick sort).
2. Пирамидальная сортировка (Heap sort).
3. Улучшенная сортировка вставками.
4. Сортировка Шелла.
5. Сортировка слиянием (Merge sort).
6. Сортировка подсчётом (Counting sort).
7. Блочная сортировка (Bucket sort).
8. External sort.

Оглавление

[На этом уроке](#_2et92p0)

[Введение](#_o5si5otb93r2)

[Сортировки](#_fxuyf22lbgsq)

[Сортировка Хоара (Quick sort)](#_f203ow6got7z)

[Реализация](#_1epketlq6q8q)

[Пирамидальная сортировка (Heap sort)](#_3znysh7)

[Улучшенная сортировка вставками](#_zbh4seamix9u)

[Сортировка Шелла](#_fz10u4gikrvy)

[Алгоритм](#_4aih8hvg6fgy)

[Выбор длин промежутков](#_dqod1p7m1bpp)

[Сортировка слиянием (Merge sort)](#_u2zedvsclkru)

[Сортировка подсчётом (Counting sort)](#_tyjcwt)

[Блочная сортировка (Bucket sort)](#_1t3h5sf)

[External sort](#_vdzvgsas1gyl)

[Приложение](#_ixj552dbyctx)

[Реализация быстрой сортировки на C#](#_3rdcrjn)

[Реализация пирамидальной сортировки](#_26in1rg)

[Заключение](#_wbuqhaaey5ps)

[Практическое задание](#_41bsjxwl630u)

[1. Bucketsort](#_c9dtpfeb0upc)

[2. External sort](#_xe6giytzz727)

[Дополнительные материалы](#_2jxsxqh)

[Используемая литература](#_7c9jrnh1eqkx)

# Введение

Алгоритмы сортировки позволяют нам упорядочить элементы для дальнейшей работы с ними. Знание, что данные отсортированы, даёт нам возможность искать элементы быстрее (двоичная сортировка), получить N самых больших или малых элементов или просто отобразить данные в порядке возрастания или убывания в UI. Но у каждого алгоритма сортировки свои особенности, которые позволяют ему быть лучше в некоторых ситуациях, чем остальные. В рамках урока мы рассмотрим наиболее часто встречающиеся алгоритмы сортировки.

# Сортировки

## Сортировка Хоара (Quick sort)

Quick sort — существенно улучшенный вариант алгоритма сортировки с помощью прямого обмена (его варианты известны как [«Пузырьковая сортировка»](https://ru.wikipedia.org/wiki/%D0%9F%D1%83%D0%B7%D1%8B%D1%80%D1%8C%D0%BA%D0%BE%D0%B2%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0) и [«Шейкерная сортировка»](https://ru.wikipedia.org/wiki/%D0%A8%D0%B5%D0%B9%D0%BA%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0)), известного в том числе своей низкой эффективностью. Принципиальное отличие состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии, и после каждого прохода элементы делятся на две независимые группы. Любопытный факт: улучшение самого неэффективного прямого метода сортировки дало в результате один из наиболее эффективных улучшенных методов.

Алгоритм состоит в следующем:

1. Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность (см. ниже).
2. Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующие друг за другом: «меньше опорного», «равные» и «бóльше».
3. Для отрезков «меньших» и «бóльших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

На практике массив обычно делят не на три, а на две части: например, «меньше опорного» и «равные и больше»; такой подход в общем случае эффективнее, так как упрощает алгоритм разделения.

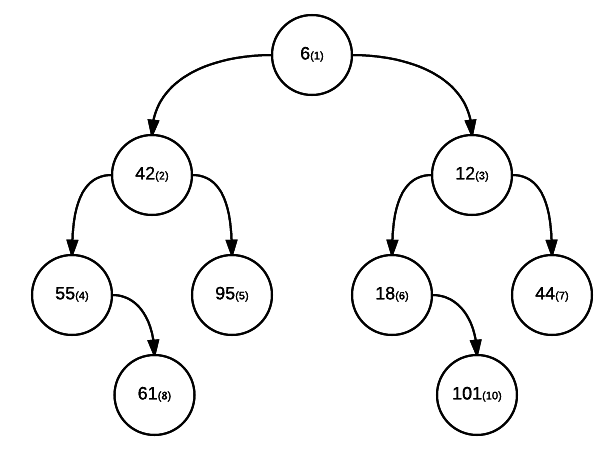
### Реализация

Эта схема использует два индекса (один в начале массива, другой в конце), которые приближаются друг к другу, пока не найдётся пара элементов, где один больше опорного и расположен перед ним, а второй меньше и расположен после. Эти элементы меняются местами. Обмен происходит до тех пор, пока индексы не пересекутся. Алгоритм возвращает последний индекс. Такая схема также эффективна в O(n2), когда входной массив уже отсортирован. Сортировка с её использованием нестабильна — это означает, что алгоритм может также переставить местами уже отсортированные данные, которые не требуют перестановки.

|  |
| --- |
| Quicksort(Data: values[], Integer: start, Integer: end)  <Выбираем элемент из массива. Называем его разделяющим элементом.>    <Переносим элементы, которые меньше разделяющего, в начало массива.>  <Переносим элементы, которые больше разделяющего или равны ему,   в конец массива.>  <Пусть серединой будет индекс, где помещен разделяющий элемент.>    *// Рекурсивно сортируем две части массива.*  Quicksort(values, start, middle - 1)  Quicksort(values, middle + 1, end) End Quicksort |

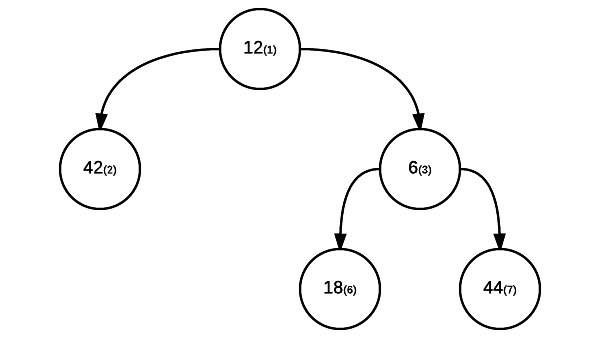
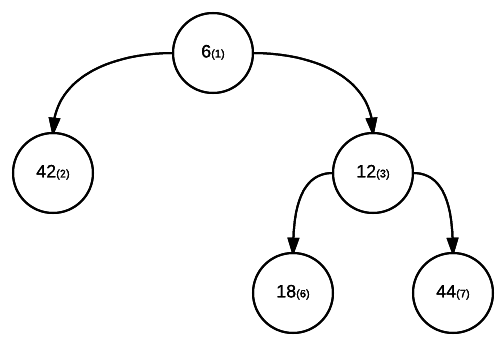
## Пирамидальная сортировка (Heap sort)

Сначала проведём необходимую подготовительную работу. Рассмотрим ряд чисел, расположенных следующим образом:

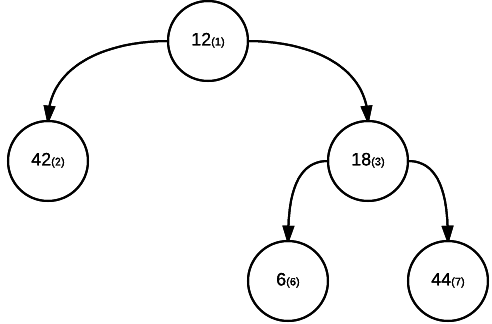


Если пронумеровать числа ai так, как это сделано на рисунке (номер указан в скобках), то нетрудно увидеть, что ai <= a2i и ai <= a2i+1 для всякого i = 1, 2, ..., n / 2, где n — количество чисел. В дальнейшем числа a2i и a2i+1 мы будем называть потомками числа ai.

Последовательность чисел, подчиняющихся указанному условию, мы будем называть пирамидой. Очевидно, что на вершине пирамиды находится минимальный элемент. Тогда можно взять число с вершины и разместить его на последнем месте в пирамиде, а число, находившееся на этом месте, расположить так, чтобы числа без этого первого минимума также составляли пирамиду (чтобы и для них соблюдалось указанное условие).

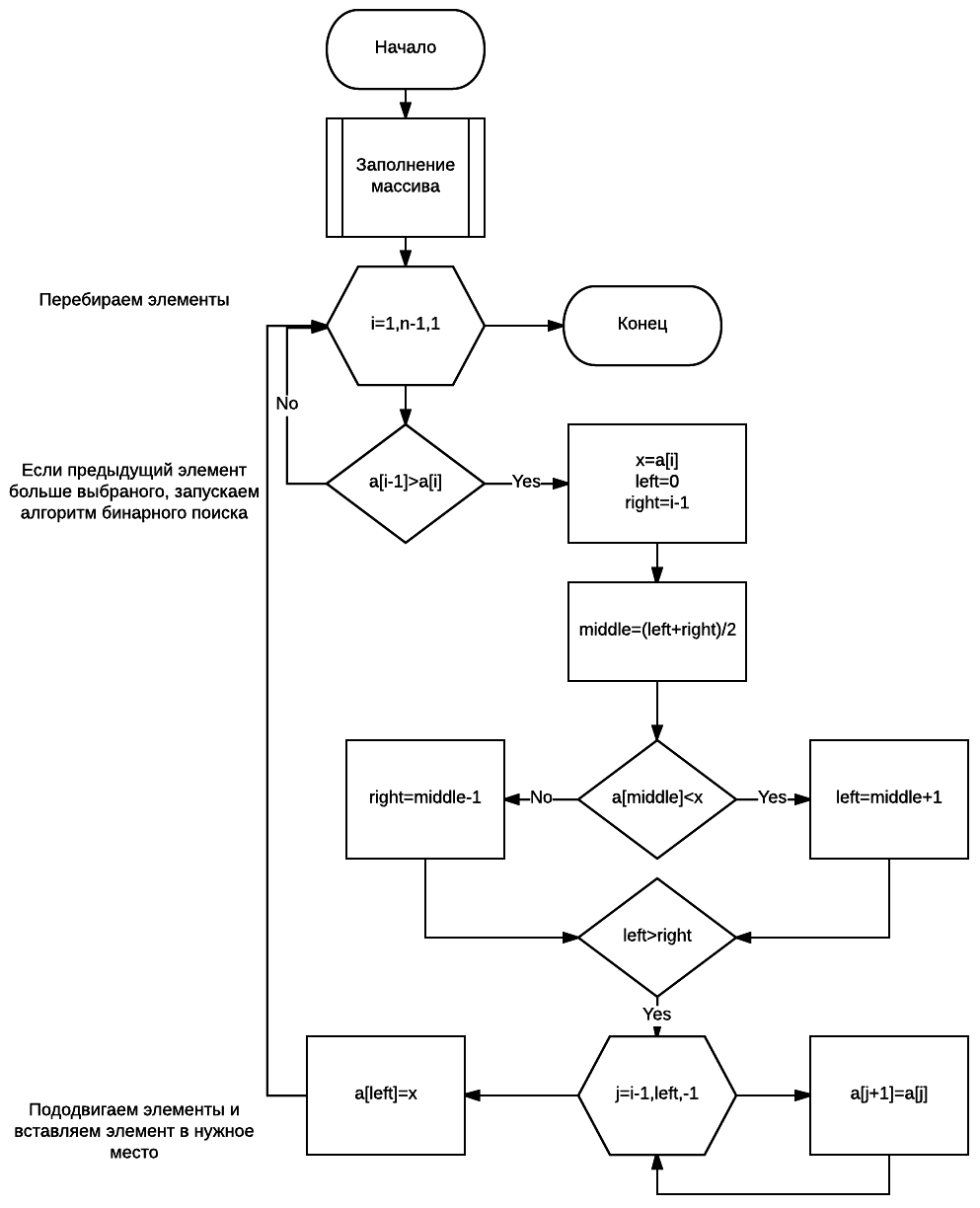
В результате переразмещения всех элементов на вершине пирамиды опять окажется минимальное из всех чисел (без учёта первого минимума). После этого новый минимум с вершины располагается на предпоследнем месте, а находившееся там число перемещается в другое (соответствующее ему) место в новой пирамиде. При этом на вершине вновь окажется очередной минимальный элемент. В результате многократного повторения указанных действий вся последовательность чисел, взятых из пирамиды, будет упорядочена.

Сделать так, чтобы на новой итерации снова получилась пирамида, можно, поместив требуемое число сначала в вершину пирамиды, а затем, сравнивая это число с его потомками, опускать его по пути, на котором находятся меньшие по сравнению с ним числа (которые одновременно поднимаются вверх). Процесс прекращается, когда встретятся «потомки», большие или равные размещаемому числу.



## Улучшенная сортировка вставками

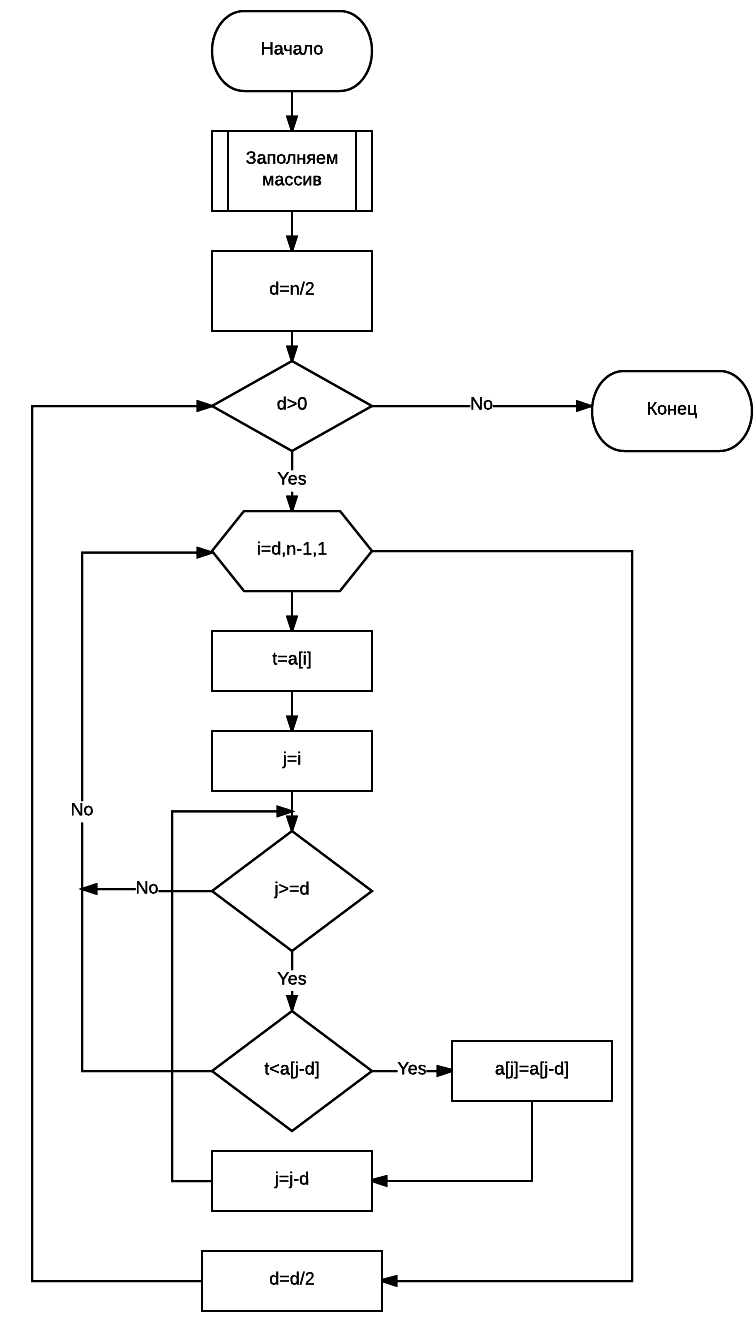
Улучшенная сортировка вставками — это та же сортировка вставками, совмещённая с бинарным поиском. В отличие от простой сортировки вставками, здесь место для вставки элемента ищется с помощью метода половинного деления. Сложность сортировки — O(N\*log(N)).



## Сортировка Шелла

Метод заключается в сравнении разделённых на группы элементов последовательностей, находящихся друг от друга на некотором расстоянии. Изначально это расстояние равно d или N / 2, где N — общее число элементов. На первом шаге каждая группа включает в себя два элемента, расположенных друг от друга на расстоянии N / 2: они сравниваются между собой и, в случае необходимости, меняются местами. На последующих шагах также происходят проверка и обмен, но расстояние d сокращается на d / 2, и количество групп, соответственно, сокращается. Постепенно расстояние между элементами уменьшается, и на d = 1 проход по массиву происходит в последний раз. Сложность: O(N\*lg(N))

### Алгоритм

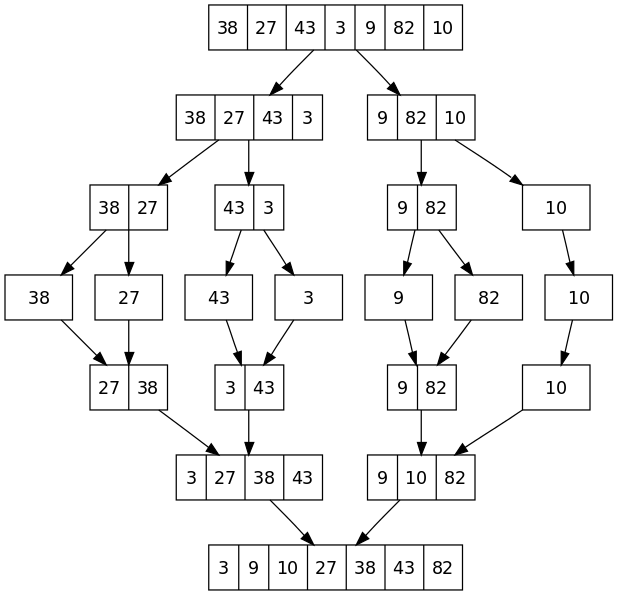


### Выбор длин промежутков

* Первоначально используемая Шеллом последовательность длин промежутков — d1 = N / 2, di = di-1 / 2, dk = 1. В худшем случае сложность алгоритма — O(N2).
* Предложенная Хиббардом последовательность: 2i – 1 <= N, i∈ N. Массив шагов заполняется перед сортировкой. В худшем случае сложность алгоритма — O(N3/2)
* Наиболее часто используемая последовательность шагов — di — изменяется по правилу: di+ 1 = (di– 1) / 2 (для массивов, содержащих более 500 элементов) и di+1 = (di – 1) / 3 (для массивов, содержащих менее 500 элементов). За d0 принимается число элементов массива. Метод заканчивает работу, когда di становится меньше 1.

## Сортировка слиянием (Merge sort)

Подобно быстрой сортировке, сортировка слиянием использует стратегию разделения массива. Если в первом случае выбирается разделитель, а оставшиеся элементы делятся на две группы — больше и меньше него, то во втором разделение происходит на две равные части, а затем алгоритм рекурсивно вызывает сам себя для их сортировки. Потом отсортированные половины сливаются в комбинированный отсортированный список.



Кроме сортируемых массивов, начального и конечного индексов алгоритм также берёт в качестве параметра рабочий массив, который использует для объединения отсортированных половин.

|  |
| --- |
| static void Merge(int[] array, int lowIndex, int middleIndex, int highIndex) {  var left = lowIndex;  var right = middleIndex + 1;  var tempArray = new int[highIndex - lowIndex + 1];  var index = 0;   while ((left <= middleIndex) && (right <= highIndex))  {  if (array[left] < array[right])  {  tempArray[index] = array[left];  left++;  }  else  {  tempArray[index] = array[right];  right++;  }   index++;  }   for (var i = left; i <= middleIndex; i++)  {  tempArray[index] = array[i];  index++;  }   for (var i = right; i <= highIndex; i++)  {  tempArray[index] = array[i];  index++;  }   for (var i = 0; i < tempArray.Length; i++)  {  array[lowIndex + i] = tempArray[i];  } }  static int[] MergeSort(int[] array, int lowIndex, int highIndex) {  if (lowIndex < highIndex)  {  if (highIndex - lowIndex == 1)  {  if (array[highIndex] < array[lowIndex])  {  var t = array[lowIndex];  array[lowIndex] = array[highIndex];  array[highIndex] = t;  }  }  else  {  var middleIndex = (lowIndex + highIndex) / 2;  MergeSort(array, lowIndex, middleIndex);  MergeSort(array, middleIndex + 1, highIndex);  Merge(array, lowIndex, middleIndex, highIndex);  }  }   return array; }  static int[] MergeSort(int[] array) {  return MergeSort(array, 0, array.Length - 1); } |

Код начинает работу с проверки общего количества элементов в части массива.

Если элемент один, значит, массив отсортирован, и алгоритм прекращает выполняться. Если элементов несколько, алгоритм рассчитывает индекс того, что находится в середине части, и рекурсивно вызывает сам себя для сортировки половин. Затем отсортированные половины объединяются. Они пересматриваются в цикле, при этом меньший элемент копируется в рабочий массив. Если одна половина пуста, алгоритм копирует остальные элементы из другой половины. На завершающем этапе объединённые элементы из рабочего массива переносятся обратно в исходный массив.

*Производительность* рассмотренного выше алгоритма составит O(N \* log(N)). Как и в случае пирамидальной сортировки, она не зависит от изначального расположения элементов, поэтому всегда одинакова. Нет здесь и худшего варианта, как в быстрой сортировке.

Сортировку слиянием тоже можно распараллелить. Когда алгоритм рекурсивно вызывает сам себя, он вправе передать один из таких вызовов другому процессору. Однако это требует некоторой координации: исходный вызов должен подождать, пока оба рекурсивных вызова закончатся, чтобы объединить их результаты. Быстрая сортировка, напротив, может приказать рекурсивным вызовам отсортировать определённую часть массива и не ждать их возвращения.

Сортировка слиянием особенно полезна, когда данные не содержатся в памяти единовременно. Например, программе нужно отсортировать 1 млн записей о клиентах, каждая из которых занимает 1 Мбайт. Для комплексной загрузки данных ей понадобится около 1000 Тбайт памяти, что значительно превышает объём памяти подавляющего большинства компьютеров. Сортировка слиянием не требует такого количества ресурсов, алгоритму даже не нужно обращаться к элементам массива, пока не вернутся его рекурсивные вызовы. Он проходит через отсортированные половины линейным способом и объединяет их, что сокращает необходимость разбивать память компьютера на страницы. Быстрая сортировка, напротив, перемещая элементы в разные половины массива, перепрыгивает с одного места на другое, увеличивая разбивку на страницы и очень сильно замедляя алгоритм.

## Сортировка подсчётом (Counting sort)

Подобный метод удобно использовать, если сортировать приходится целые числа, лежащие в относительно небольшом диапазоне. Например, вам нужно упорядочить k целых чисел от 0 до 1000. Для этого мы должны установить количество элементов массива с определенным значением, а затем скопировать это значение по порядку нужное количество раз обратно в массив. Реализацию такого метода показывает следующий код:

|  |
| --- |
| static void SimpleCountingSort(int[] array, int count, int max) *// где k - длина массива А,*  *// а max - его максимальное значение* {  int[] countArray = new int[max];    for (int i = 0; i < 1000; i++)  countArray[array[i]]++;    var b = 0;  for (int j = 0; j < count; j++)  {  for (int i = 0; i < countArray[j] - 1; i++)  {  array[b++] = j;  }  } } |

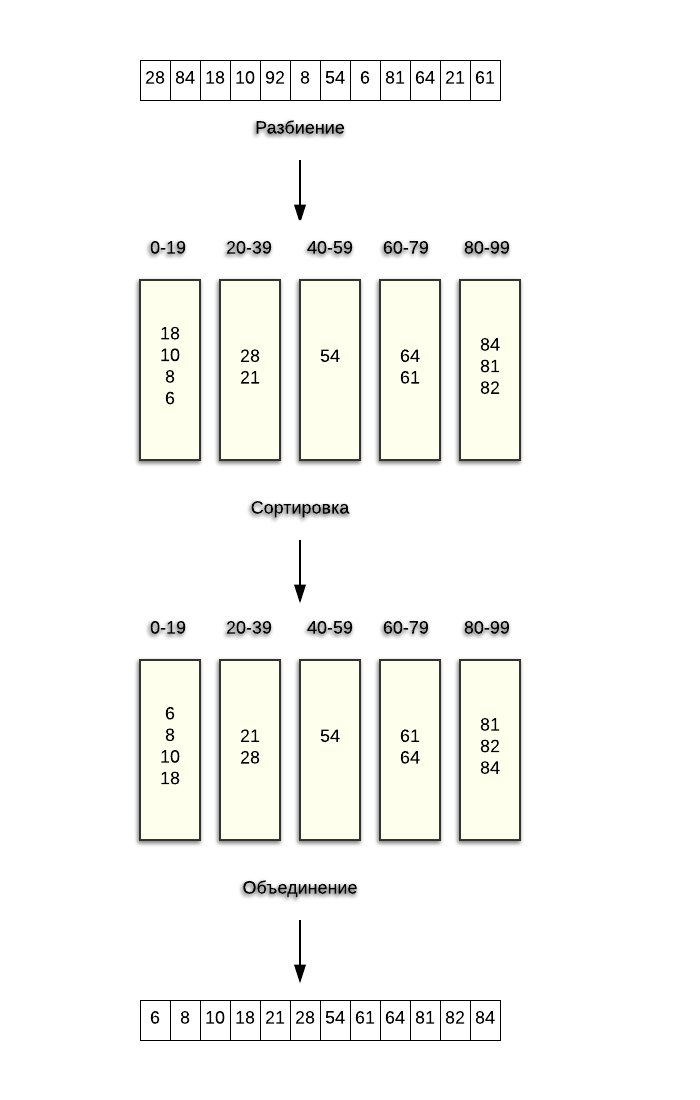
Вначале мы обнуляем массив. Далее подсчитываем количество вхождений каждого числа в массиве A – организуем частотный массив C. После этого заполняем массив A нужным количеством чисел, используя частотный массив C.

# Блочная сортировка (Bucket sort)

Алгоритм блочной или, как её ещё называют, корзинной сортировки делит элементы на блоки, а затем сортирует их с помощью рекурсивного вызова блочной сортировки или другого алгоритма и присоединяет содержимое блоков к исходному массиву. В следующем псевдокоде представлен высокоуровневый алгоритм, демонстрирующий этот метод:

|  |
| --- |
| Bucketsort(Data: values[])  <Создаём блоки.>  <Распределяем элементы по блокам.>  <Сортируем блоки.>  <Собираем элементы блоков в исходный массив.> End Bucketsort |

Блоками могут выступать стеки, связанные списки, очереди, массивы или любые другие структуры данных. Стадия объединения напоминает последнюю стадию в merge sort, где последние блоки массива объединяются с сортировкой. В базовом варианте блочной сортировки каждый блок должен содержать элементы, которые меньше элементов в следующем блоке. Но в некоторых ситуациях данным условием можно пренебречь, например, если заранее известно, что элементы массива имеют высокую повторяемость, то есть много одинаковых элементов.



## External sort

Extertnal sort — расширение блочной сортировки (но без особенности, что в каждом блоке элементы должны быть меньше, чем в следующем), для решения проблемы, когда массив не помещается в оперативную память и его хранят на внешней памяти, например просто на жестком диске. В этом алгоритме применяются следующие шаги:

1. Определение максимального размера блока.
2. Чтение данных из файла в блок.
3. Сортировка данных в блоке.
4. Сохранение содержимого блока в отдельный временный файл.
5. Если файл был прочитан не полностью, вернуться к пункту 2.
6. Постепенное чтение отсортированных блоков из ранее сохранённых файлов, и запись читаемых (в порядке сортировки) данных в финальный файл (стадия merge из mergesort).

# Приложение

## Реализация быстрой сортировки на C#

|  |
| --- |
| static void QuickSort(int[] array, int first, int last) {  int i = first, j = last, x = array[(first + last) / 2];   do  {  while (array[i] < x)  i++;  while (array[j] > x)  j--;   if(i <= j)  {  if (array[i] > array[j])  {  var tmp = array[i];  array[i] = array[j];  array[j] = tmp;  }   i++;  j--;  }  } while (i <= j);   if (i < last)  QuickSort(array, i, last);  if (first < j)  QuickSort(array, first, j); } |

## Реализация пирамидальной сортировки

|  |
| --- |
| static void HeapSort(int[] array, int n) //основной метод {  for (int i = n / 2 - 1; i >= 0; i--)  {  Heapify(array, n, i);  }   for (int i = n - 1; i >= 0; i--)  {  int temp = array[0];  array[0] = array[i];  array[i] = temp;  Heapify(array, i, 0);  } }  static void Heapify(int[] array, int n, int i) {  int largest = i;  int left = (i \* 2) + 1;  int right = (i \* 2) + 2;   if (left < n && array[left] > array[largest])  largest = left;   if (right < n && array[right] > array[largest])  largest = right;   if (largest != i)  {  int swap = array[i];  array[i] = array[largest];  array[largest] = swap;  Heapify(array, n, largest);  } } |

# 

# Заключение

В рамках урока мы рассмотрели алгоритмы сортировки, наиболее универсальный — quicksort, но в задачах, где высокая плотность одинаковых элементов, наиболее эффективна будет сортировка подсчётом. В задачах, где разделение массива будет более эффективно, например, в многопроцессорных системах, стоит рассмотреть mergesort, так как меньшее количество пересечений потока на одном и том же участке памяти. Но если речь идёт о сортировке больших массивов данных, возможно, вам подойдет external sort.

# Практическое задание

## 1. Bucketsort

Реализовать Bucketsort, проверить корректность работы.

## 2. External sort

Дописать реализацию Bucketsort до возможности сортировки больших массивов из файла (External sort).

# Дополнительные материалы

1. [Алгоритмы сортировки.](https://programm.top/c-sharp/algorithm/array-sort/)

# Используемая литература

1. Р. Стивенс. Алгоритмы. Теория и практическое применение. М.: Издательство «Э», 2016.
2. Н. Вирт. Алгоритмы и структуры данных. Новая версия для Оберона. М.: ДМК-пресс, 2010.
3. Д. Рихтер CLR via C# М.: Питер, 2019.