# Краткие ответы на экзамен по Основам Алгоритмизации и Программированию

Первый Вопрос:

## Первая группа:

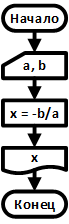
### 1) Основные этапы решения задач на ЭВМ

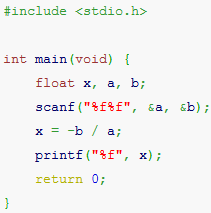
1. **Формулировка задачи(математическая)** - это описание задачи с помощью математических формул или их последовательности;
2. **Выбор метода вычисления**:
   1. **Точные методы** - обеспечивает точные вычисления искомых значений и представляет собой последовательность арифметических и логических операций;
   2. **Численные методы** - методы приближённого решения математических задач, сводящиеся к выполнению конечного числа элементарных операций над числами;
3. **Разработка схемы алгоритма**;  
   **Алгоритм** - это конечная последовательность действий, однозначно определяющая процесс обработки

**Машинный алгоритм** - 1) это алгоритм, который может выполнить цифровая машина;   
2) однозначно определенный процесс обработки исходных и промежуточных данных в ходе решения задач.

1. **Составление программы на языке программирования** - процесс написания программы.

**Пример**:  
Задача: Найти корни уравнения: ax + b = 0, где a, b - значения, заданные пользователем.

1. Выразим x: x = - b/a
2. Метод вычисления: Точный метод, так как у нас есть готовая формула
3. Блок-схема  
   
4. Программа на Си:



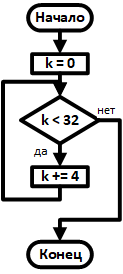
### 2) Способы и средства описания алгоритма

**Алгоритм** - это конечная последовательность действий, однозначно определяющая процесс обработки

**Машинный алгоритм** - 1) это алгоритм, который может выполнить цифровая машина;   
2) однозначно определенный процесс обработки исходных и промежуточных данных в ходе решения задач.

* **Словесное представление (план, пункты)**

1. k = 0
2. если k < 32, то **п. 3**, иначе **п. 5**
3. k += 4
4. **п. 2**
5. конец алгоритма

* **Блок-схемы**
* **Псевдокод** - основные конструкции ЯП без ненужных для понимания алгоритма деталей;

нач  
k = 0

пока k < 32 нц  
k += 4  
кц  
кон алг

### 3) Определение вычислительной сложности алгоритма

Сложность алгоритмов обычно оценивают по времени выполнения или по используемой памяти. Алгоритм имеет сложность **O(f(n))**, если при увеличении размера входных данных **n**, время выполнения алгоритма возрастает с той же скоростью, что и функция **f(n)**. Оценивая порядок сложности алгоритма, необходимо использовать только ту часть, которая возрастает быстрее всего.

Примеры некоторых видов сложности по времени:

**O(n) — линейная сложность**

Пример: алгоритм поиска наибольшего элемента в не отсортированном массиве. Нам придётся пройтись по всем n элементам массива, чтобы понять, какой из них максимальный.

**O(log n) — логарифмическая сложность**

Пример: вводится число *a*, являющееся степенью двойки (*= a*). Нужно найти эту степень, путём деления этого числа на два и подсчёта количества этих делений.

**O(n^2) — квадратичная сложность**

Пример: алгоритм сортировки пузырьком. Здесь мы не учитываем остальные операции, а только два вложенных цикла, так как только они будут иметь смысл при увеличении кол-ва данных

**O(2^n) — экспоненциальная сложность**

Пример: вывести на экран все двоичные числа длин N

**O(1) - не зависящая от размера данных сложность**Пример: для определения значения третьего элемента массива не нужно ни запоминать элементы, ни проходить по ним сколько-то раз.

**Пример:** Пусть 1 мкс - время выполнения алгоритма со сложностью O(n), при n = 1:

|  | **n = 1** | **n = 2** | **n = 4** | **n = 8** | **n = 16** |
| --- | --- | --- | --- | --- | --- |
| **O(log n)** | 0 (б/м) | 1 | 2 | 3 | 4 |
| **O(n)** | 1 | 2 | 4 | 8 | 16 |
| **O(n^2)** | 1 | 4 | 16 | 64 | 256 |
| **O(2^n)** | 2 | 4 | 16 | 256 | 65536 |

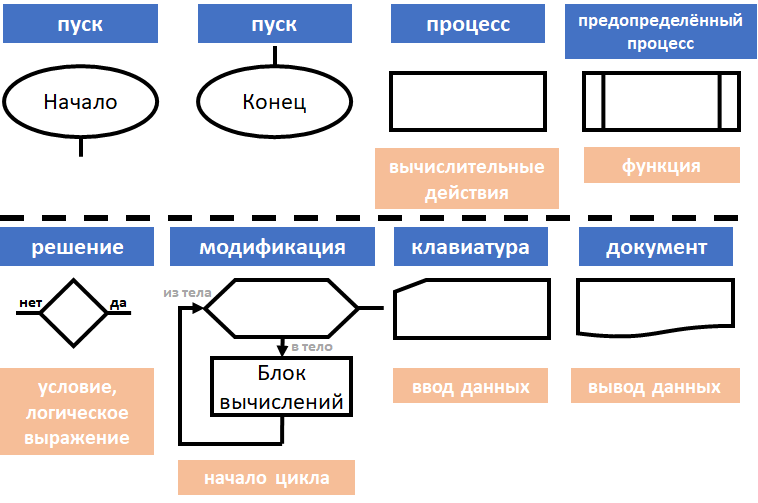
### 4) Блок-схемы алгоритмов линейных структур

**Линейные** - алгоритмы, символы которых изображены в той последовательности, в которой должны быть выполнены. Например, данная блок-схема описывает алгоритм сложения двух введённых чисел и их вывод:

### 5) Блок-схемы алгоритмов разветвленных структур

**Разветвлённые** - алгоритмы, в которых выполнение той или иной последовательности действий происходит в зависимости от результатов проверки какого-либо условия.   
Например, данная блок-схема описывает алгоритм нахождения и вывода максимального из трёх введённых чисел:

### 6) Элементы составления блок-схем алгоритмов



### 7) Блок-схемы алгоритмов циклических структур

**Циклические** – это алгоритмы, в которых предусмотрено неоднократное выполнение одной и той же последовательности действий.

Основные действия цикла:

1. Начальная инициализация параметров цикла;
2. Изменение значений параметров в конце тела цикла;
3. Проверка условий выполнения (окончания) цикла.

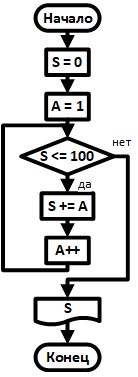
Параметр цикла -это переменная которая изменяется при повторении цикла

Существует три формы циклов:

| **Цикл с предусловием** | **Цикл с постусловием**  (в любом случае хотя бы раз будут выполнены действия внутри тела цикла) | **Цикл с параметром** |
| --- | --- | --- |
|  |  |  |

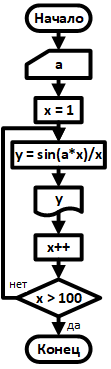
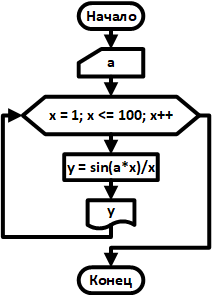
### 8) Блок-схемы алгоритмов с итерационным циклом

Особенность итерационного цикла является то, что число повторений операций тела цикла заранее неизвестно. Выход из итерационного цикла осуществляется в случае невыполнения данного условия. Например, данный алгоритм последовательно считает сумму всех чисел подряд от 1, до тех пор, пока сумма не превысит 100 и выводит её:



### 9) Использование элементов “модификация” и “решение” в циклических алгоритмах

y = sin(ax)/x, = 1, 2, 3, … 100



## Вторая группа (язык программирования C):

### 1) Структура программы, директивы, переменные и константы



### 2) Типы данных, преобразования их

**Целочисленные типы**

Могут быть знаковыми и беззнаковыми(в зависимости от этого меняется их диапазон)

1. ***char*** (1 байт)
2. ***int*** (4 байта)
3. ***short int*** (2 байта)
4. ***long int*** (8 байт)

**Типы с плавающей точкой**

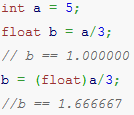
1. ***float*** (4 байта)
2. **double** (8 байт)
3. ***long double*** (16 байт)

**Преобразовывать типы** друг к другу можно как **явно**, так и **неявно**

**Неявным преобразованием** являются выражения в которых используются разные типы(при этом происходит расширение меньшего типа до размеров большего),



**Явным преобразованием** является выражение в котором указано, к какому типу привести то или иное значение. Для того чтобы привести один тип к другому явно, нужно в круглых скобках указать тип, к которому нужно привести значение, а после этого само значение.



### 3) Операция отношения, логические операции, приоритет операций

В языке Си выделяют следующие операторы:

**Операции отношений**

1. равенство a == b
2. неравенство a != b
3. больше a > b
4. меньше a < b
5. больше или равно a >= b
6. меньше или равно a <= b

Результатом этих операций является 1, если результат выполнения операции — истина, и 0, если — ложь

**Логические операции**

1. конъюнкция, “И” a && b
2. дизъюнкция, “ИЛИ” a || b
3. инверсия, “НЕ” !a

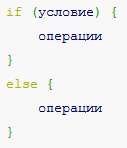
**Приоритет операций:**

1. Унарные(!, ++, --, +, -);
2. Арифметические(\*, /, %, +, -) - по правилам математики;
3. Отношения(>, <, <=, >=, ==, !=);
4. Логические(&&, ||);
5. Условная(?:);
6. Присваивания(=, +=, -=, \*=, /=, %=).

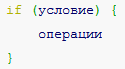
### 4) Циклы, ветвления

**Оператором ветвления** в языке Си является конструкция if-else и switch

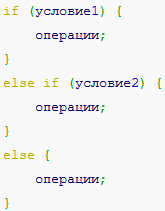
*Полная форма* **if-else**



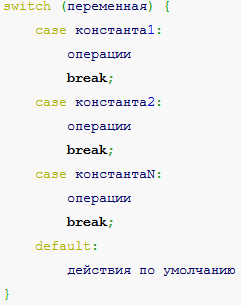
*Неполная форма*(без else)



Оператор if может быть вложенным



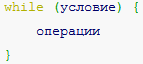
Оператор ветвления **switch**, выполняющий действия, основываясь на сравнении значения со списком констант



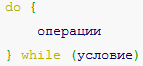
Если значение переменной не соответствует ни одному case, то выполняется default ветвь. Она может отсутствовать, тогда вообще ничего не выполняется. Оператор **break** позволяет выйти из блока.

В языке Си есть 3 вида циклов: **цикл с предусловием**, **постусловием** и **параметрический**

В **цикле с предусловием** сначала проверяется условие, а затем в зависимости от результата выполняется, либо не выполняется тело цикла.

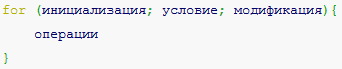


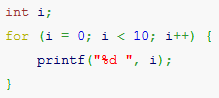
В **цикле с постусловием** сначала выполняется тело, а затем проверяется условие.



**Параметрический цикл** сначала проверяет условие, а затем на основании результата выполняется или нет тело цикла. Инициализация счетчика и его изменение задаются в объявлении самого цикла.

В записи цикла for можно опустить одно или несколько выражений, но нельзя опускать точку с запятой, разделяющие три составляющие цикла. Для разделения нескольких выражений может использоваться запятая.



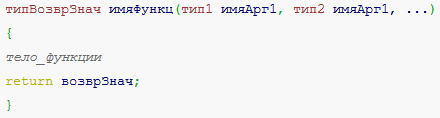
Пример:  
****

В теле любого цикла можно использовать **break** (позволяет выйти из цикла) и **continue** (позволяет сразу перейти на следующую итерацию).

### 5) Определение функции

**Функция** - это своего рода подпрограмма, которая может содержаться в основной программе, а может быть создана отдельно (в библиотеке). Функцию, либо её прототип нужно определять обязательно до её вызова. Если объявлен прототип функции, то функцию можно расположить и ниже места её вызова. При объявлении функции указываются формальные параметры, которые потом используются внутри самой функции. При вызове функции мы используем фактические параметры.

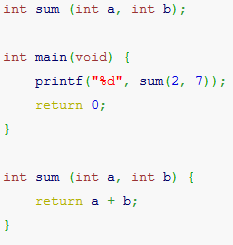
Пример определения функции:



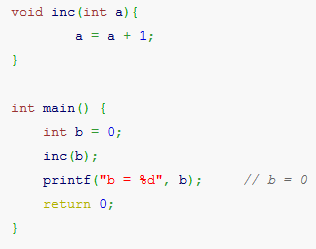
Функция может вызвать сама себя, тогда она называется **рекурсивной**.

Если функция не принимает аргументов, то скобки оставляют пустыми. Можно также написать слово void. Функция может и не возвращать ничего, тогда вместо возвращаемого типа написать void(тогда чтобы выйти из функции нужно использовать просто return;).

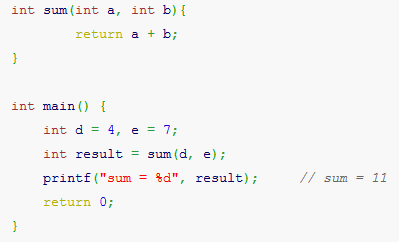
*Пример использования функции:*



### 6) Передача аргумента в функцию по значению и возврат значений При объявлении функции указываются формальные параметры, которые потом используются внутри самой функции. При вызове функции мы используем фактические параметры.

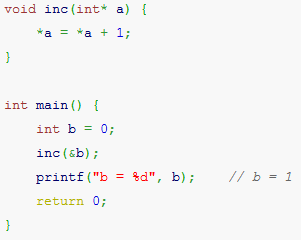


В примере выше, мы **передаём** только значение переменной b, поэтому все изменения аргумента, происходящие в функции, не будут отражаться на переменной b.  
Ключевое слово **void** в стандарте языка Си используется для указания того, что функция не возвращает значение.

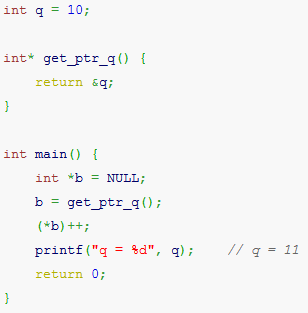


В данном примере, функция **возвращает** нам значение - сумму двух переменных, которых мы ей передали.

### 7) Передача аргументов в функцию и возврат значений через указатель



В примере выше, мы **передаём** не значение переменной b, а её адрес, поэтому все изменения аргумента, происходящие в функции, будут отражаться на переменной b.



В данном примере, функция **возвращает** нам адрес глобальной переменной, изменяя значение переменной по этому адресу, мы напрямую воздействуем на переменную q. Ф*ункция не может вернуть указатель на локальную переменну*ю, её нужно объявить как статическую.

### 8) Аргументы по умолчанию

**В Си аргументов по умолчанию НЕТ!!!**

Но если писать для C++, то пример ниже

int sum(int a, int b = 10) { // int b = 10 - это и есть аргумент по умолчанию

return a + b;

}  
Все аргументы по умолчанию в определении функции должны располагаться справа.

При вызове функции:

sum(5); // результат будет 15

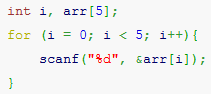
sum(6, 2); // результат будет 8

### 9) Объявление и инициализация одномерного и двумерного массива

**Одномерный массив**

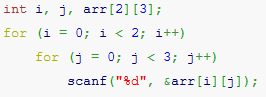
При инициализации массива сразу при объявлении можно не указывать его размер: компилятор его сам подставит. Но если количество инициализирующих значений, указанных в фигурных скобках, меньше, чем количество элементов массива, указанное в квадратных скобках, то все оставшиеся элементы в массиве будут равны нулю.





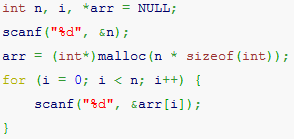
**Двумерный массив**





### Динамические массивы

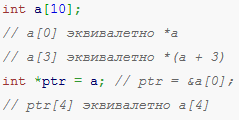
Для работы нужна библиотека stdlib.h

Функция *malloc(size)* выделяет блок памяти, размером *size* байт, и возвращает указатель на начало блока. Содержание выделенного блока памяти не инициализируется, оно остается с неопределенными значениями.  


### 10) Работа с элементами массива через указатель

**Одномерный массив**

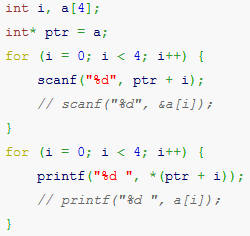
Имя массива - это указатель на первый элемент этого массива.



**Многомерный массив**

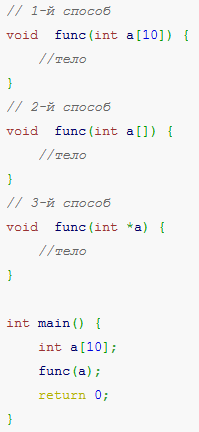
По своему строению многомерный массив является обыкновенным, "одномерным", массивом. Все элементы расположены друг за другом. Поэтому работа с ним через указатели не сильно отличается от работы с одномерным массивом.

**Пример работы с элементами массива через указатель**:



### 11) Передача массивов в функцию, возвращение массива из функции

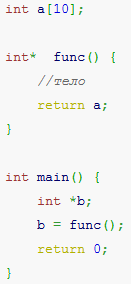
Когда массив используется в качестве аргумента функции, **передается только указатель на первый элемент массива**, а не копия всего массива. Имеется три способа объявления параметра, предназначенного для получения указателя на массив.



Размер массива в функцию автоматически не передается, поэтому если размер массива заранее (на этапе компиляции) не оговорен, то нужно передать параметр, который содержит количество элементов в массиве.

**Вернуть** из функции, как и предать, можно только указатель на первый элемент массива.

Также ***функция не может вернуть указатель на локальную переменную***, поэтому либо её нужно объявить как статическую, либо возвратить глобальную переменную.

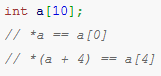
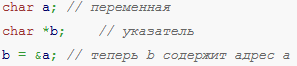


### 12) Определение указателей и их особенности в зависимости от типа данных

**Указатель** – это переменная, которая хранит адрес области памяти. Указатель, как и переменная, имеет тип. Синтаксис объявления указателей:

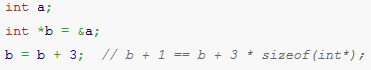
тип \*ИмяОбъекта;

Основной пример указателя - массив.



**Для справки:** стоит заметить, что **int\* b**,  **int \* b** и **int \*b** эквивалентны.

Указатели поддерживают **арифметические операции**:



Сложение (вычитание) у указателей немного иное: прибавляя(вычитая) к указателю число, мы сдвигаем его на *число \* размер типа* байт

**Операторы** для работы с указателями:

& — оператор взятия адреса переменной

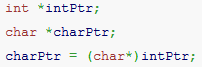
\* — оператор разыменования - возвращает переменную по адресу указателя



**Размер указателя** не зависит от его типа, а только от разрядности системы:

на 32-битной - 4 байта

на 64-битной - 8 байт

Указатель одного типа можно **привести** к указателю другого:  


Принято при определении указателя, если он не инициализируется конкретным значением, делать его равным **NULL**:  


На языке C есть возможность создать указатель на неопределенный тип(универсальный указатель) - **void**.Ему может быть присвоен адрес переменной любого типа. Перед разыменованием, его нужно привести к одному из типов.  


Можно сделать вывод, что тип указателю нужен в основном только для разыменования и сложения(вычитания).

# Второй и Третий Вопросы

### 1)Сортировка Пузырьком

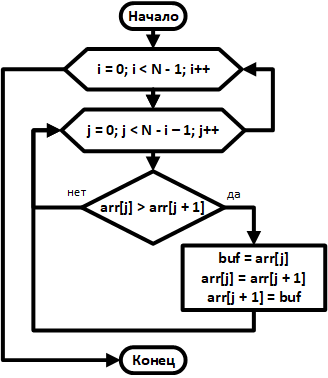
Алгоритм состоит в повторяющихся проходах по сортируемому массиву. На каждой итерации последовательно сравниваются соседние элементы, и, если порядок в паре неверный, то элементы меняются местами. За каждый проход по массиву как минимум один элемент встает на свое место(всплывает как пузырёк)

**Словесное описание:**

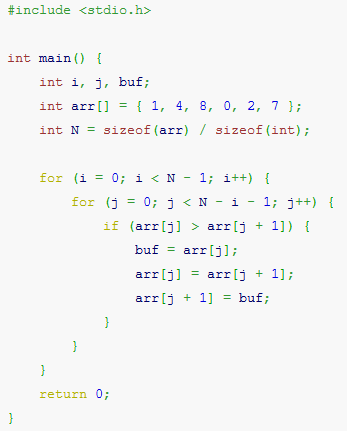
arr - массив, N - длина массива

1. i = 0
2. если i < N - 1, то **п. 3**, иначе **п. 11**
3. j = 0
4. если j < N - i - 1, то **п. 5**, иначе **п. 9**
5. если arr[j] > arr[j + 1], то **п. 6**, иначе **п. 7**
6. обмен значениями arr[j] и arr[j + 1]
7. j++
8. **п. 4**
9. i++
10. **п. 2**
11. конец алгоритма

**Блок-схема:**

****

**Программа на Си:**

****

### 2)Сортировка Расчёской (продв. сорт. Пузырьком)

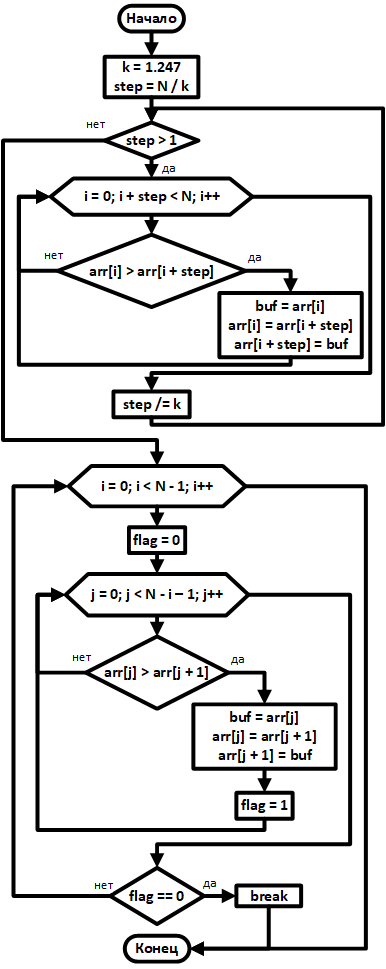
Идея сортировки состоит в том, чтобы первоначально брать достаточно большое расстояние между сравниваемыми элементами(чтобы устранить маленькие значения в конце списка, замедляющие сортировку пузырьком) и постепенно сужать это расстояние вплоть до минимального. Разрыв между сравниваемыми элементами лучше брать с учётом специальной величины - фактором уменьшения, оптимальное значение которой равно примерно 1,247. Сначала расстояние между элементами равно размеру массива, разделённого на фактор уменьшения (дробная часть отбрасывается). Затем сортировка повторяется при шаге, делённом на фактор уменьшения, и в конце процесс сортировки завершается при шаге, равном 1. После этого идёт модифицированная сортировка пузырьком

**Словесное описание:**

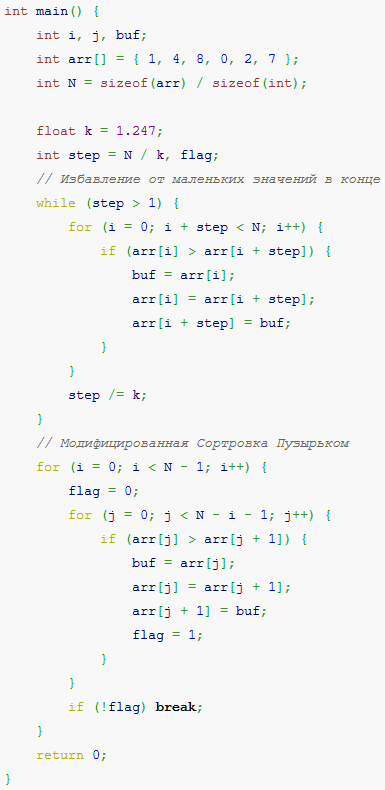
arr - массив, N - длина массива, flag - переменная для фиксации факта перемещения элемента за внутренний цикл, k - фактор уменьшения, step - шаг

1. k = 1.247, step = N / k
2. если step > 1, то **п. 3**, иначе **п. 11**
3. i = 0
4. если i + step < N, то **п. 5**, иначе **п. 9**
5. если arr[i] > arr[i + step], то **п. 6**, иначе **п. 7**
6. обмен значениями arr[i] и arr[i + step]
7. i++
8. **п. 4**
9. step /= k
10. **п. 2**
11. i = 0
12. если i < N - 1, то **п. 13**, иначе **п. 23**
13. flag = 0, j = 0
14. если j < N - i - 1, то **п. 15**, иначе **п. 20**
15. если arr[j] > arr[j + 1], то **п. 16**, иначе **п. 18**
16. обмен значениями arr[j] и arr[j + 1]
17. flag = 1
18. j++
19. **п. 14**
20. если flag == 0, то **п. 23**(оператор break), иначе **п. 21**
21. i++
22. **п. 12**
23. конец алгоритма

**Блок-схема:**

****

**Программа на Си:**

****

### 3)Сортировка Вставками

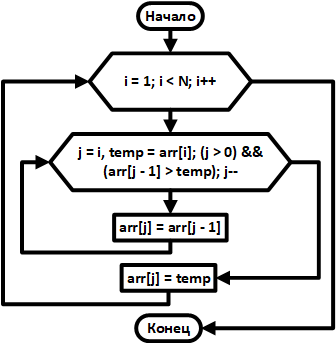
Сортируемый массив можно разделить на две части — отсортированная часть и неотсортированная. В начале сортировки первый элемент массива считается отсортированным, все остальные — не отсортированные. Начиная со второго элемента массива и заканчивая последним, алгоритм вставляет неотсортированный элемент массива в нужную позицию в отсортированной части массива.

**Словесное описание:**

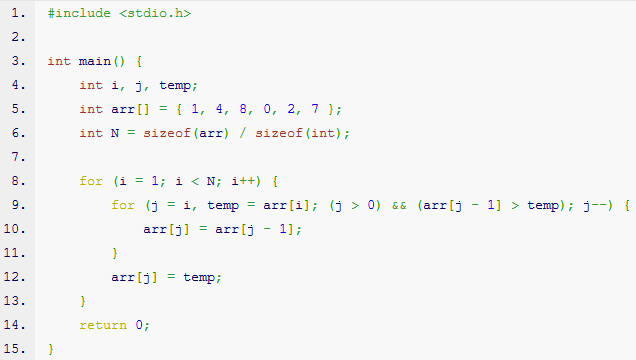
arr - массив, N - длина массива

1. i = 1
2. если i < N, то **п. 3**, иначе **п. 11**
3. j = i, temp = arr[i]
4. если (j > 0) && (arr[j - 1] > temp), то **п. 5**, иначе, **п. 8**
5. arr[j] = arr[j - 1]
6. j--
7. **п. 4**
8. arr[j] = temp
9. i++
10. **п. 2**
11. конец алгоритма

**Блок-схема:**

****

**Программа на Си:**

****

### 4)Сортировка Шелла (продв. сорт. Вставками)

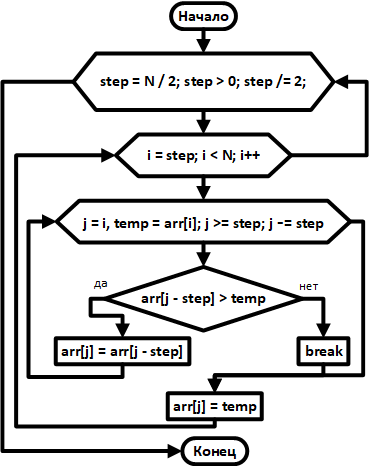
Алгоритм сортирует элементы отстоящие друг от друга на некотором расстоянии. Затем сортировка повторяется при меньших значениях шага, и в конце процесс сортировки Шелла завершается при шаге, равном 1 (а именно обычной сортировкой вставками). Шелл предложил такую последовательность размера шага: N/2, N/4, N/8 …, где N – количество элементов в сортируемом массиве.

**Словесное описание:**

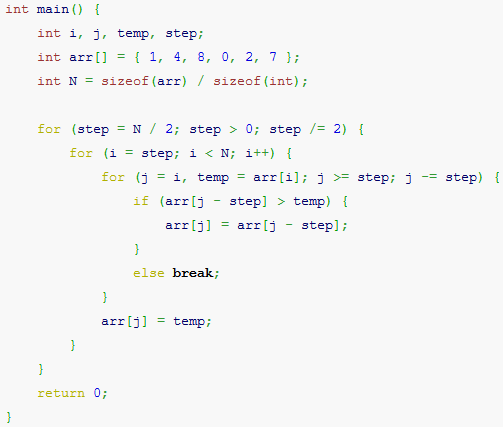
arr - массив, N - длина массива

1. step = N / 2
2. если step > 0, то **п. 3**, иначе **п. 17**
3. i = step
4. если i < N, то **п. 5**, иначе **п. 15**
5. j = i, temp = arr[i]
6. если j >= step, то **п. 7**, иначе, **п. 12**
7. если arr[j - step] > temp, то **п.8**, иначе **п.11**
8. arr[j] = arr[j - step]
9. j -= step
10. **п. 6**
11. **п. 12**(оператор break)
12. arr[j] = temp
13. i++
14. **п. 4**
15. step /= 2
16. **п. 2**
17. конец алгоритма

**Блок-схема:**

****

**Программа на Си:**



### 5)Сортировка Выбором

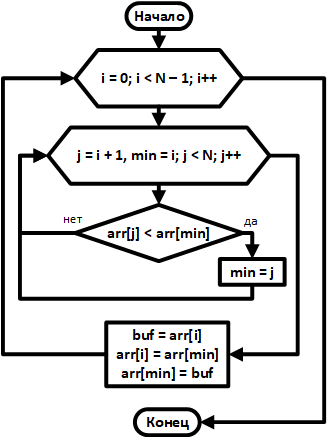
В неотсортированном подмассиве ищется локальный минимум. Найденный минимум меняется местами с первым элементом в подмассиве. Если в массиве остался неотсортированный подмассив, то действие повторяется.

**Словесное описание:**

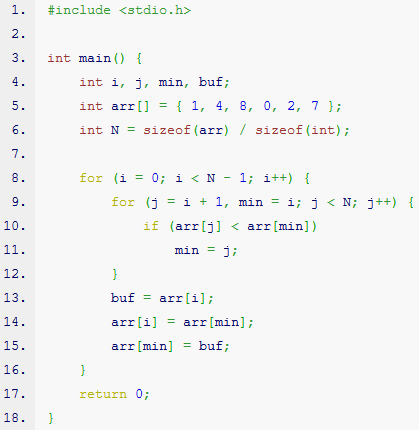
arr - массив, N - длина массива

1. i = 0
2. если i < N - 1, то **п. 3**, иначе **п. 12**
3. min = i, j = i + 1
4. если j < N, то **п. 5**, иначе **п. 9**
5. если arr[j] < arr[min], то **п. 6**, инача **п. 7**
6. min = j
7. j++
8. **п. 4**
9. обмен значениями arr[i] и arr[min]
10. i++
11. **п. 2**
12. конец алгоритма

**Блок-схема:**

****

**Программа на Си:**

****

### 6)Гномья сортировка (гибрид сорт. Пузырьком и Вставками)

Смотрим на текущий и предыдущий элемент массива: если они в правильном порядке, шагаем на один элемент вперед, иначе меняем их местами и шагаем на один элемент назад. Граничные условия: если нет предыдущего элемента, шагаем вперёд; если нет следующего элемента,стоп.

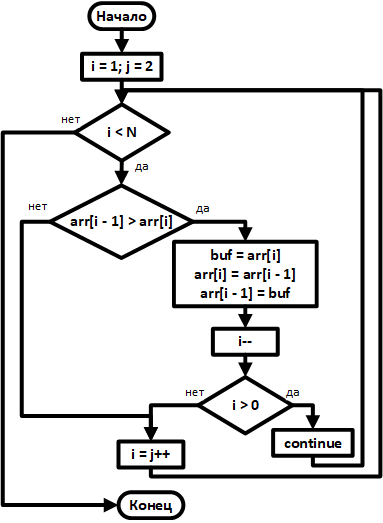
Это оптимизированная версия с использованием переменной j, чтобы разрешить прыжок вперёд туда, где он остановился до движения влево, избегая лишних итераций и сравнений:

**Словесное описание:**

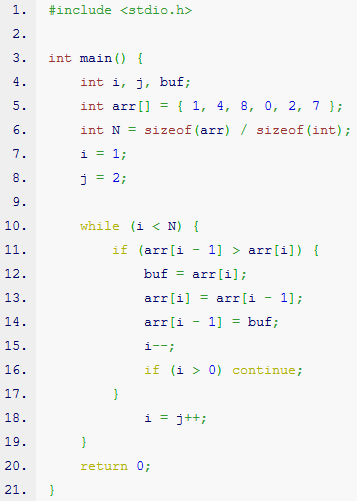
arr - массив, N - длина массива

1. i = 1, j = 2
2. если i < N, то **п. 3**, иначе **п. 9**
3. если arr[i - 1] > arr[i], то **п. 4**, иначе **п. 7**
4. обмен значениями arr[i] и arr[i - 1]
5. i--
6. если i > 0, то **п. 2**(оператор continue), иначе **п. 7**
7. i = j++
8. **п. 2**
9. конец алгоритма

**Блок-схема:**



**Программа на Си:**

****

### 7)Быстрая сортировка (сортировка Хоара)

Выбираем из массива опорный элемент - любой из элементов массива(от его выбора в некоторых случаях зависит эффективность). Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три части, следующие друг за другом: «меньшие опорного», «равные» и «большие». Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина больше единицы. Деление на три части происходит так: слева и справа запускаются два цикла, ищущие слева элемент, больший опорного, а справа - меньший и обменивает их до тех пор, пока итератор слева и итератор справа не пересекутся.

**Словесное описание:**

arr - массив, first – индекс первого элемента массива, last – индекс последнего элемента массива

**qsort(arr, first, last)**

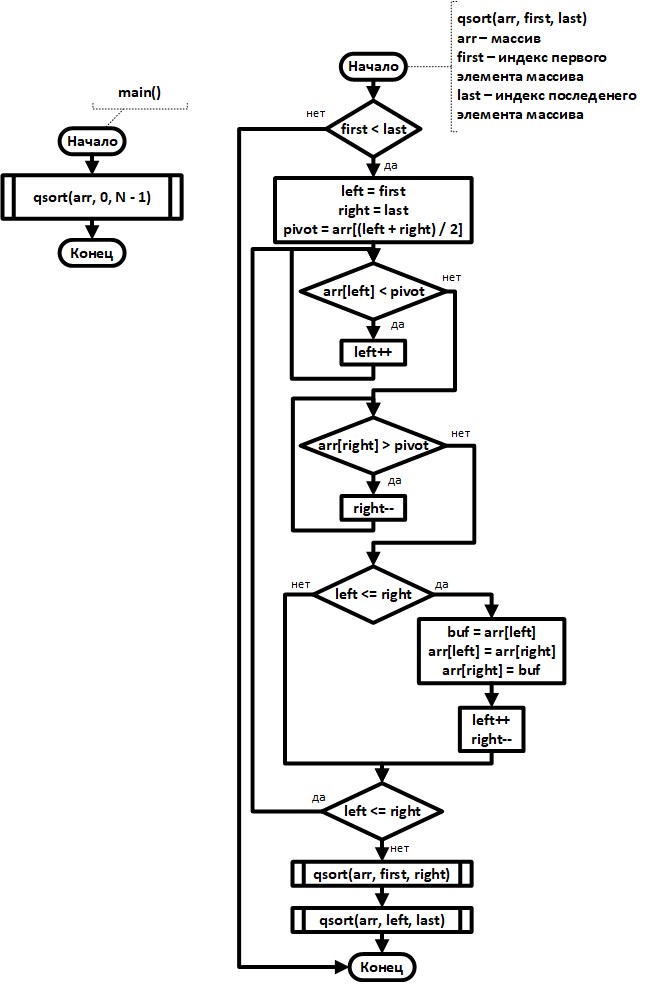
1. если first < last, то **п. 2**, иначе **п. 15**
2. left = first, right = last, pivot = arr[(left + right) / 2]
3. если arr[left] < pivot, то **п. 4**, иначе **п. 6**
4. left++
5. **п. 3**
6. если arr[right] > pivot, то **п. 7**, иначе **п. 9**
7. right--
8. **п. 6**
9. если left <= right, то **п. 10**, иначе **п. 12**
10. обмен значениями arr[left] и arr[right]
11. left++, right--
12. если left <= right, то **п. 3**, иначе **п. 13**
13. qsort(arr, first, right)
14. qsort(arr, left, last)
15. конец подпрограммы

arr - массив, N - длина массива

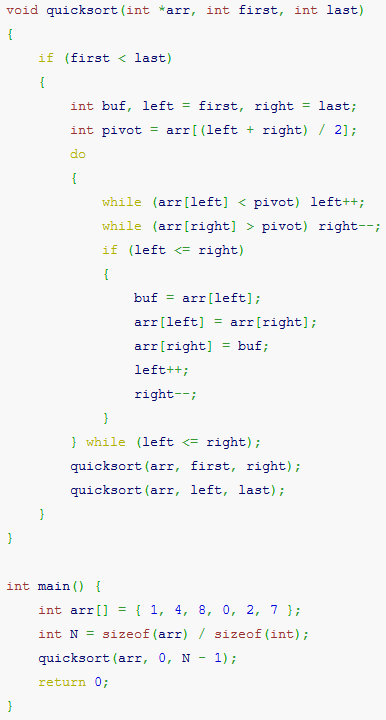
**main()**

1. qsort(arr, 0, N - 1)
2. конец алгоритма

**Блок-схема:**

****

**Программа на Си:**

****

## Ссылки на алгоритмы сортировки

1) <https://ideone.com/J28Cl1>

2) <https://ideone.com/IacthY>

3) [https://ideone.com/hLuaL7\](https://ideone.com/hLuaL7%5C)

4) <https://ideone.com/txCWOP>

5) <https://ideone.com/AZyHHX>

6) <https://ideone.com/3B5OVT>

7) <https://ideone.com/UCTF4B>