основы алгоритмизации и программирования

ВВЕДЕНИЕ В ЯЗЫК С

ПЛАН ЛЕКЦИИ

1. Введение
2. Структура программы
3. Директивы
4. Библиотеки
5. Типы данных
    - 5.1 Простые данные
    - 5.2 Составные данные
    - 5.3 Другие типы данных

ПЛАН ЛЕКЦИИ

6. Преобразование типов
7. Переменные и константы
8. Операции отношения. Логические операции
9. Приоритет операций
10. Условные операторы
11. Операторы цикла
12. Функции

ВВЕДЕНИЕ

Язык Си, созданный Денисом Ритчи в начале 70-х годов в Bell Laboratory американской корпорации AT&T, является одним из универсальных языков программирования. Язык Си считается языком системного программирования, хотя он удобен и для написания прикладных программ. Среди преимуществ языка Си следует отметить переносимость программ на компьютеры различной архитектуры и из одной операционной системы в другую, лаконичность записи алгоритмов, логическую стройность программ, а также возможность получить программный код, сравнимый по скорости выполнения с программами, написанными на языке ассемблера. Последнее связано с тем, что хотя Си является языком высокого уровня, имеющим полный набор конструкций структурного программирования, он также обладает набором низкоуровневых средств, обеспечивающих доступ к аппаратным средствам компьютера. С 1989 года язык Си регламентируется стандартом Американского института национальных стандартов ANSI C. В настоящее время, кроме стандарта ANSI C разработан международный стандарт ISO C (International Standard Organization C).

СТРУКТУРА ПРОГРАММЫ

```c
#include <stdio.h> //Подключение библиотеки ввода/вывода
#define C 23 //Объявление константы C = 23

int global = 10; //Объявление глобальной переменной

//Функция суммы двух целых чисел
int sum(int a, int b)
{
    return a + b;
}

//Точка входа в программу - главная функция
int main()
{
    int local = 5; //Локальная переменная
    const int b = 3; //Константа типа int
    //Вызов функции и присваивание её результата переменной local
    local = sum(global, b);
    printf("%d", local); //Вывод результата на экран: 13

    //Возврат целочисленного значения перед завершением функции
    return 0;
}
```

ДИРЕКТИВЫ

Директивы препроцессора представляют собой инструкции, записанные в тексте программы на СИ, и выполняемые до трансляции программы. Все директивы препроцессора начинаются со знака #. После директив препроцессора точка с запятой не ставятся.

Основные директивы препроцессора:

- #include широко используется для включения в программу так называемых заголовочных файлов, содержащих прототипы библиотечных функций, и поэтому большинство программ на СИ начинаются с этой директивы.
- #define используется, что бы создать константы.

БИБЛИОТЕКИ

Стандартная библиотека языка Си – это набор отдельных файлов, которые расширяют возможности языка Си.
Возможности языка Си, без функций стандартной библиотеки, очень ограничены. Не представляется возможным даже вывести значение переменной на экран. Но благодаря дополнительным модулям (стандартным заголовочным файлам) возможности языка могут быть существенно расширены. Библиотеки можно подключать с помощью директивы #include.
Вот некоторые, **наиболее практичные:**

- <stdio.h> //Осуществляет ввод и вывод на экран
- <math.h> //Для вычисления основных математических функций
- <string.h> //Позволяет работать с различными видами строк
- <time.h> //Для конвертации между различными форматами даты и времени
- <locale.h> //Используется для задач, связанных с локализацией

ТИПЫ ДАННЫХ

Тип данных определяет множество значений, набор операций, которые можно применять к таким значениям и способ реализации хранения значений и выполнения операций.

Различают следующие типы данных:

Простые данные:
целочисленные,
вещественные,
символьные
логические.

Составные данные:
массив,
строковый тип ,
структура.

Другие типы данных:
указатель.

ПРОСТЫЕ ДАННЫЕ

Целочисленные данные могут быть представлены в знаковой и беззнаковой форме. Основные типы и размеры целочисленных данных:

- short int (2 байта)
- int (4 байта)
- long int (8 байт)

Вещественный тип предназначен для представления действительных чисел. Различают три основных типа представления вещественных чисел в языке Си:

- float (4 байта)
- double (8 байт)
- long double (16 байт)

Символьный тип хранит код символа и используется для отображения символов в различных кодировках. Символьные данные задаются в кодах и по сути представляют собой целочисленные значения. Для хранения кодов символов в языке Си используется тип char (1 байт).

Логический тип применяется в логических операциях, используется при алгоритмических проверках условий и в циклах и имеет два значения: истина — true ложь — false. Для хранения используется тип bool (1 байт).

СОСТАВНЫЕ ДАННЫЕ

Массив — индексированный набор элементов одного типа.

Строковый тип — массив, хранящий строку символов.

Структура — набор различных элементов (полей записи), хранимый как единое целое и предусматривающий доступ к отдельным полям структуры.

ДРУГИЕ ТИПЫ ДАННЫХ

Указатель — хранит адрес в памяти компьютера, указывающий на какую-либо информацию, как правило — указатель на переменную.

ПРЕОБРАЗОВАНИЯ ТИПОВ

Приведением типа называется преобразование значения переменной одного типа в значение другого типа.

При явном приведении перед выражением следует указать в круглых скобках имя типа, к которому необходимо преобразовать исходное значение.

Пример явного приведения типа.

```
int x = 5;
double y = 15.3;
x = (int) y;
y = (double) x;
```

При неявном приведении преобразование происходит автоматически, по правилам, заложенным в языке Си.

Пример неявного приведения типа.

```
int x = 5;
double y = 15.3;
y = x; //здесь происходит неявное приведение типа к double
x = y; //здесь происходит неявное приведение типа к int
```

ПЕРЕМЕННЫЕ И КОНСТАНТЫ

Переменная — область памяти, в которую могут помещаться различные значения.
Любая переменная до ее использования в программе на языке Си должна быть объявлена, то есть для нее должны быть указаны тип и имя.
Объявление переменных в Си осуществляется в форме:

ТипПеременной ИмяПеременной; `// int i;`

При объявлении переменной ей может быть присвоено начальное значение в форме:

ТипПеременной ИмяПеременной=значение; `// int i = 10;`

Константа — это ограниченная последовательность символов алфавита языка, представляющая собой изображение неизменяемого объекта.
Константы бывают числовые, символьные и строковые. Числовые константы делятся на целочисленные и вещественные.
Объявление константы в Си осуществляется в форме:

const тип ИмяПеременной = НачальноеЗначение; `// const int i = 10;`

ОПЕРАЦИИ ОТНОШЕНИЯ И ЛОГИКИ

Операции отношения:

- a == b эквивалентно — проверка на равенство;
- a != b не равно — проверка на неравенство;
- a < b меньше;
- a > b больше;
- a <= b меньше или равно;
- a >= b больше или равно.

Операции отношения используются при организации условий и ветвлений.

Логические операции:

a **&&** b — И (конъюнкция) — требуется одновременное выполнение всех операций отношения;

a || b — ИЛИ (дизъюнкция) — требуется выполнение хотя бы одной операции отношения;

!a — НЕ (инверсия) — требуется невыполнение операции отношения.

Логические операции чаще всего используются в операциях проверки условия if и могут выполняться над любыми объектами.

ПРИОРИТЕТ ОПЕРАЦИЙ

1.	Унарные	! ++ -- + -	Логическое НЕ Инкрементирование Декрементирование Унарный плюс Унарный минус
2.	Арифметические	*, / +,-	Умножение, деление Сложение, вычитание
3.	Отношения	>, < >= <= ==, !=	Больше, меньше Больше или равно Меньше или равно Равно, не равно
4.	Логические	&& ||	Логическое И Логическое ИЛИ
5.	Условная	?:	Тернарное условие
6.	Присваивания	= +=, -= *=, /=, %=	Простое присваивание Присваивание через сумму и разность Присваивание через произведение, частое и остаток

УСЛОВНЫЕ ОПЕРАТОРЫ

Разветвляющимся называется такой алгоритм, в котором выбирается один из нескольких возможных вариантов вычислительного процесса.
Признаком разветвляющегося алгоритма является наличие операций проверки условия.
Чаще всего для проверки условия используется условный оператор if.

Условный оператор if

Условный оператор if может использоваться в форме ***полной*** или ***неполной*** развилки.

Полная развилка	Неполная развилка

if (Условие)
{
  БлокОпераций1;
}
	if (Условие)
	{
	  БлокОпераций1;
	}
	else
	{
	  БлокОпераций2;
	}

УСЛОВНЫЕ ОПЕРАТОРЫ

Оператор ветвления switch (оператор множественного выбора)
Оператор if позволяет осуществить выбор только между двумя вариантами. Для того, чтобы производить выбор одного из нескольких вариантов необходимо использовать вложенный оператор if. С этой же целью можно использовать оператор ветвления switch.

Общая форма записи:

```
switch (ЦелоеВыражение)
{
 case Константа1: БлокОпераций1;
   break;
 case Константа2: БлокОпераций2;
   break;
 . . .
 case Константаn: БлокОперацийn;
   break;
 default: БлокОперацийПоУмолчанию;
   break;
}
```

Константы в опциях case должны быть целого типа (могут быть символами).

ОПЕРАТОРЫ ЦИКЛА

Циклом называется блок кода, который для решения задачи требуется повторить несколько раз. В языке Си следующие виды циклов:

Цикл с предусловием	Цикл с постусловием	Параметрический цикл

while (Условие)
{
БлокОпераций;
}
	do {
	БлокОпераций;
	}
	while (Условие);
		for
		(Инициализация; Условие; Модификация)
		{
		БлокОпераций;
		}

Особенность этого цикла состоит в том, что вполне возможно, что тело цикла не будет выполнено ни разу если в момент первой проверки проверяемое условие окажется ложным.	Использовать цикл лучше в тех случаях, когда должна быть выполнена хотя бы одна итерация, либо когда инициализация объектов, участвующих в проверке условия, происходит внутри тела цикла.	Для организации такого цикла необходимо осуществить три операции: Инициализация - присваивание параметру цикла начального значения; Условие - проверка условия повторения цикла (сравнение величины параметра с некоторым граничным значением); Модификация - изменение значения параметра для следующего прохождения тела цикла.

ФУНКЦИИ

Функция — это подпрограмма, которая может содержаться в основной программе, а может быть создана отдельно (в библиотеке). Каждая функция выполняет в программе определенные действия.
Определение функции имеет следующий синтаксис:

```
ТипВозвращаемогоЗначения ИмяФункции(СписокФормальныхАргументов)
{
  ТелоФункции;
 ...
 return(ВозвращаемоеЗначение);
}
```

Пример: Функция сложения двух вещественных чисел.

```
float function(float x, float z)
{
  float y;
  y=x+z;
  return(y);
}
```

основы алгоритмизации и программирования

БЛОК-СХЕМЫ АЛГОРИТМОВ

ПЛАН ЛЕКЦИИ

Часть 1:

1. Понятие алгоритма

2. Определение «блок-схемы»

3. Элементы составления блок-схем

ПЛАН ЛЕКЦИИ

Часть 2:

4. Блок-схемы алгоритмов линейной структуры

5. Блок-схемы алгоритмов разветвлённой структуры

6. Блок схемы алгоритмов циклической структуры

7. Блок схемы алгоритмов с итерационным циклом

8. Элементы «модификация» и «решение» в циклических алгоритмах

ПОНЯТИЕ АЛГОРИТМА

Решение любой задачи на ЭВМ необходимо разбить на следующие этапы: разработка алгоритма решения задачи, составление программы решения задачи на алгоритмическом языке, ввод программы в ЭВМ, отладка программы (исправление ошибок), выполнение программы на ПК, анализ полученных результатов. Рассмотрим первый этап решения задачи – разработку алгоритма.

Алгоритм – конечная последовательность действий, однозначно определяющая процесс обработки исходных и промежуточных данных в ходе решения задачи. Разработка алгоритма решения задачи – это разбиение задачи на последовательно выполняемые этапы, причем результаты выполнения предыдущих этапов могут использоваться при выполнении последующих.

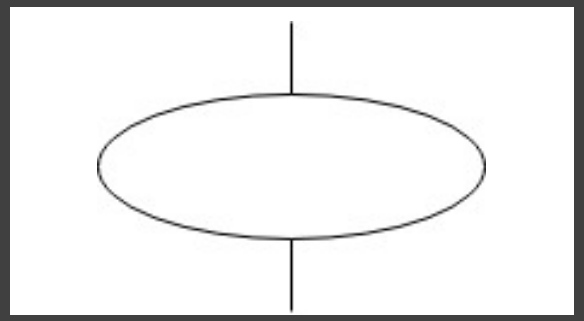
Существуют следующие формы представления алгоритмов:

- Словесная
- Графическая (блок-схема)
- Псевдокод
- Программная

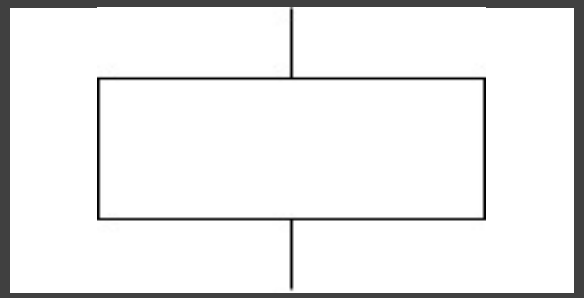
ОПРЕДЕЛЕНИЕ «БЛОК-СХЕМЫ»

Блок-схема — распространенный тип **схем** (графических моделей), описывающих **алгоритмы** или процессы, в которых отдельные шаги изображаются в виде **блоков** различной формы, соединенных между собой линиями, указывающими направление последовательности.
На территории Российской Федерации действует единая система программной документации (ЕСПД), частью которой является Государственный стандарт — **ГОСТ 19.701-90 «Схемы алгоритмов программ, данных и систем»** . Стандарт в частности регулирует способы построения схем и внешний вид их элементов.

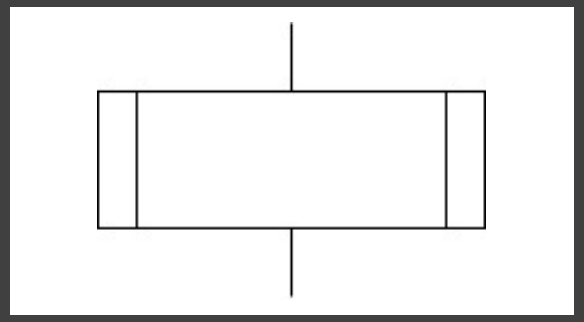
ЭЛЕМЕНТЫ БЛОК-СХЕМ



Элемент **«Пуск»** отвечает за начало - конец алгоритма, вход - выход в подпрограмму.

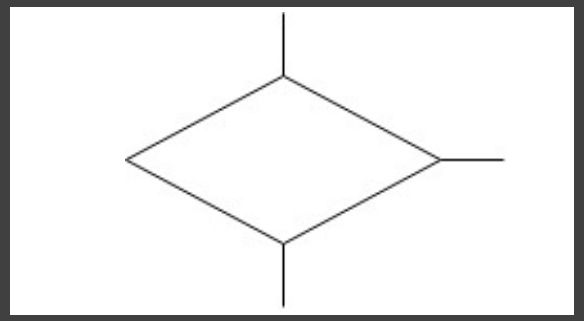


В элементе **«Процесс»** выполняются операции по вычислению или присваиванию.

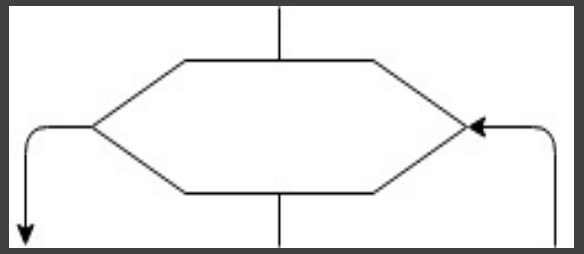


Элемент **«Предопределённый процесс»** отвечает за вычисление по подпрограмме (вызов внешних функций).

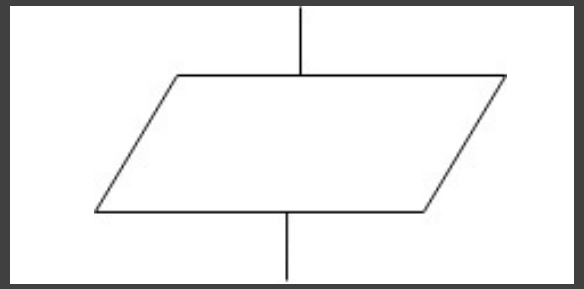
ЭЛЕМЕНТЫ БЛОК-СХЕМ



Элемент **«Решение»** отвечает за проверку условий. С его помощью реализуется ветвление алгоритма, а также итерационный цикл.

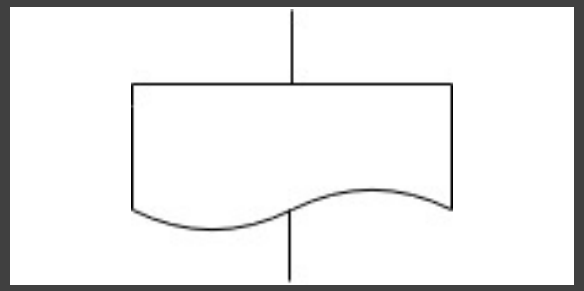


Элемент **«Модификация»** отвечает за начало параметрического цикла.



Элемент **«Клавиатура»** отвечает за вывод и ввод в общем виде.

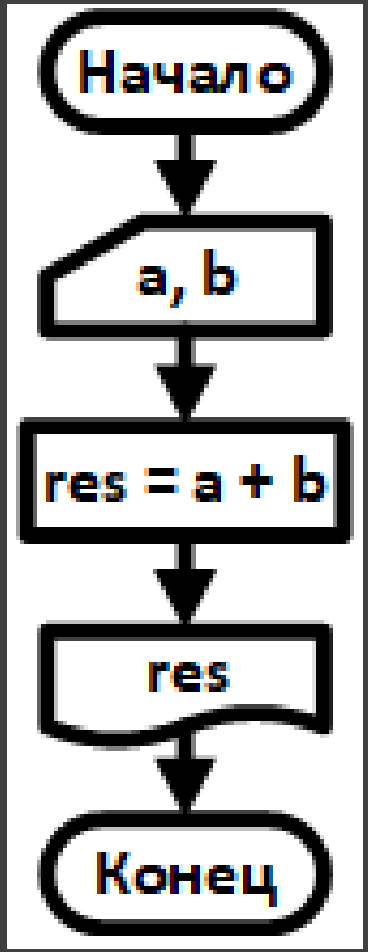
ЭЛЕМЕНТЫ БЛОК-СХЕМ



Элемент **«Документ»** отвечает за вывод результатов на печать.

ЛИНЕЙНАЯ СТРУКТУРА

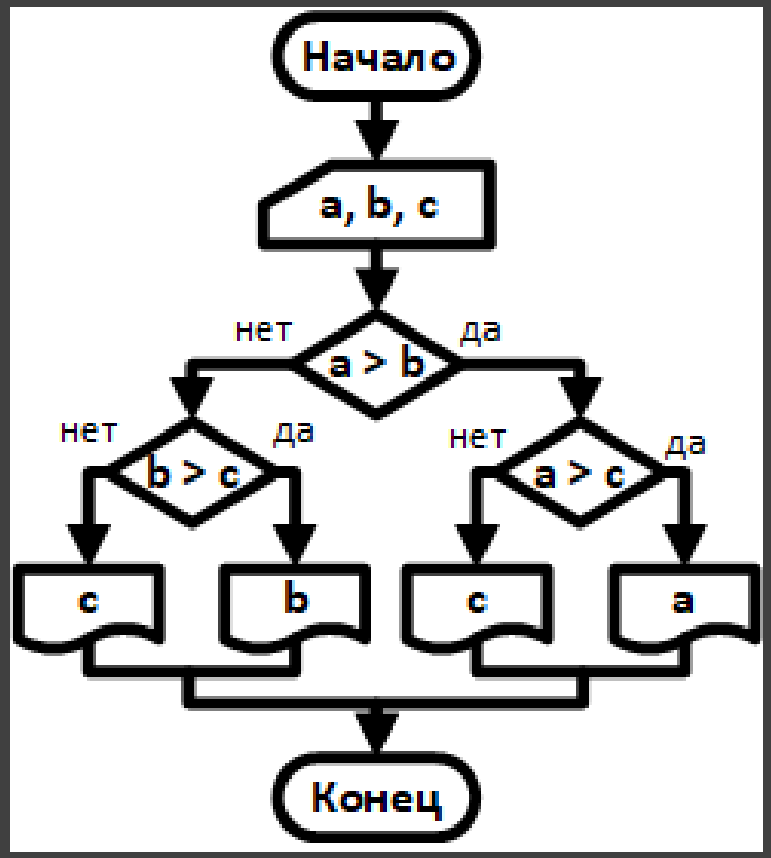
Линейные — алгоритмы, символы которых изображены в той последовательности, в которой должны быть выполнены. Алгоритм линейной структуры изображается линейной последовательностью связанных друг с другом блоков. Такой порядок выполнения действий называется естественным. Поэтому в схемах алгоритмов линейной структуры нет блока «Решение». Как правило, алгоритмы линейной структуры состоят из трех частей: ввод исходных данных, вычисления результатов по формулам, вывод значений результатов. Это самые простые алгоритмы.



Блок-схема, описывает алгоритм сложения двух введённых чисел и их вывод.

РАЗВЕТВЛЁННАЯ СТРУКТУРА

Разветвлённые алгоритмы — алгоритмы, в которых выполнение той или иной последовательности действий происходит в зависимости от результатов проверки какого-либо условия.



Блок-схема, описывающая алгоритм нахождения и вывода максимального из трёх введённых чисел.

ЦИКЛИЧЕСКАЯ СТРУКТУРА

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

Основные действия цикла:

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

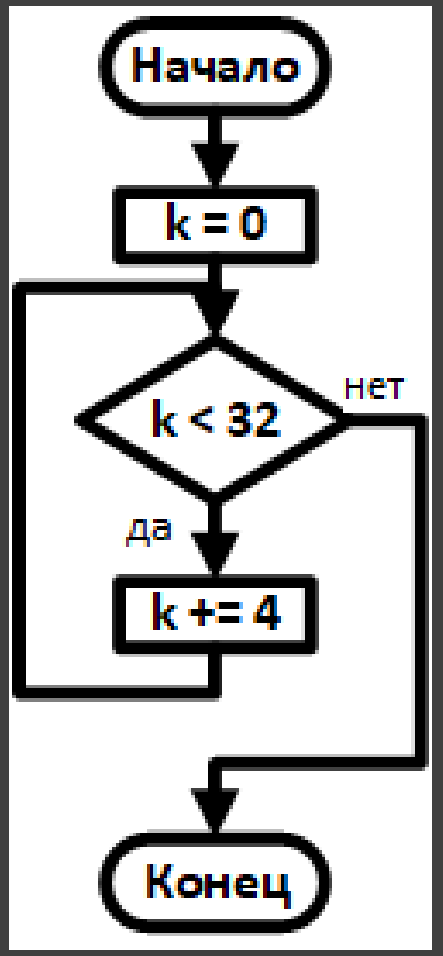
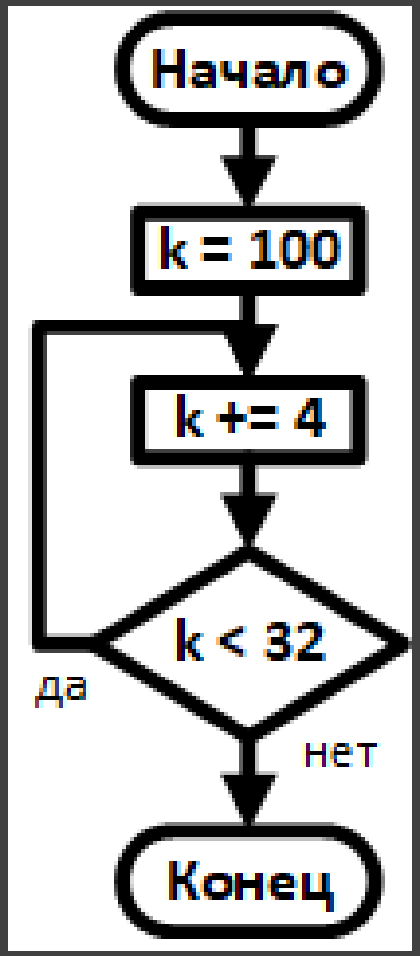
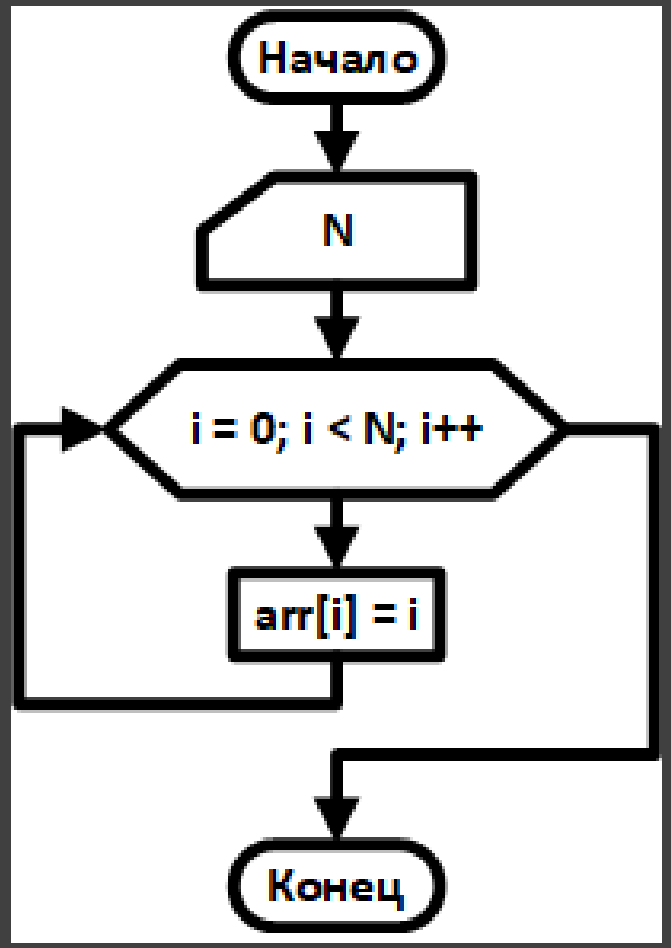
Параметр цикла — это переменная которая изменяется при повторении цикла

Существует три формы циклов:

- Цикл с предусловием. Условие проверяется до тела цикла. Есть вероятность, что тело цикла не выполнится ни разу.
- Цикл с постусловием. Условие проверяется после тела цикла. Хотя бы один раз тело цикла обязательно выполнится.

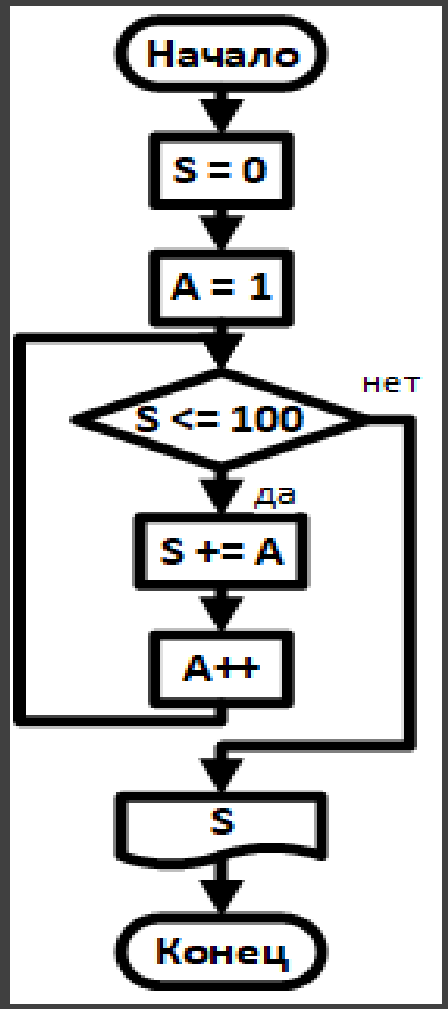
Параметрический цикл. Это цикл с заранее известным количеством итераций.

ЦИКЛИЧЕСКАЯ СТРУКТУРА

Цикл с предусловием	Цикл с постусловием	Параметрический цикл
		

ИТЕРАЦИОННЫЙ ЦИКЛ

Особенностью **итерационного цикла** является то, что число повторений операторов тела цикла заранее неизвестно. Для его организации используется цикл типа while. Выход из итерационного цикла осуществляется в случае невыполнения данного условия. На каждом шаге вычислений происходит последовательное приближение к искомому результату и проверка условия достижения последнего.



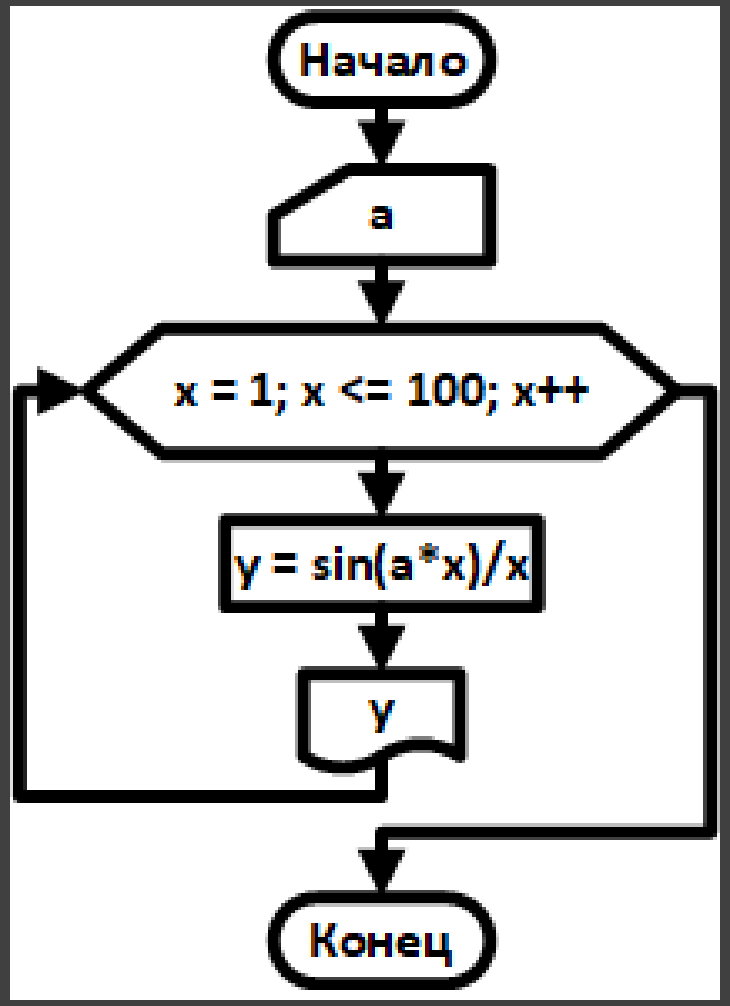
Алгоритм последовательно считает сумму всех чисел подряд от 1, до тех пор, пока сумма не превысит 100 и выводит её.

«МОДИФИКАЦИЯ» И «РЕШЕНИЕ»

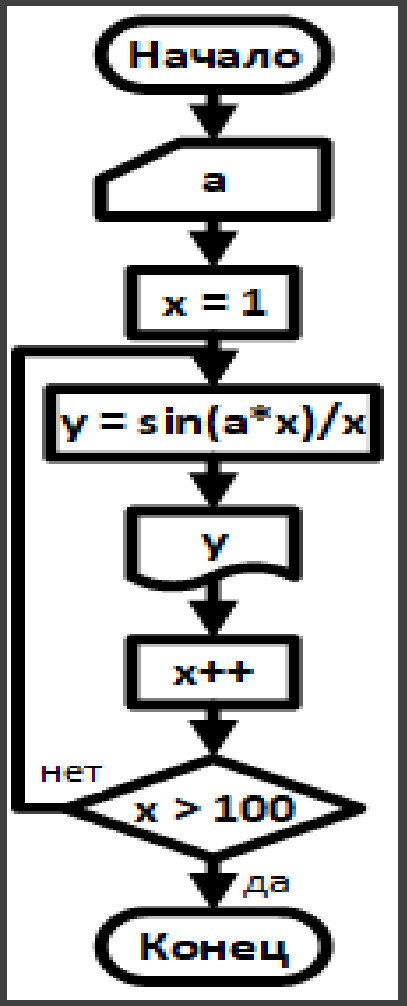
Алгоритм циклической структуры можно организовать с помощью элементов блок-схем: «модификация» и «решение».

В примерах программа последовательно производит расчёт по формуле: y = sin(ax)/x и выводит результат на экран. Количество результатов определяет цикл.

В элементе «модификация» задано начальное и граничное значение для x: x = 1; x <= 100. Следовательно, пока выполняется условие тело цикла будет выполняться.
Аналогично в блок-схеме с «решением». Только там начальное значение задано в «процессе», а условие проверяется после тела цикла.



Блок-схема с «модификацией».



Блок-схема с «решением».

основы алгоритмизации и программирования

СОРТИРОВКА «ПУЗЫРЁК»

ПЛАН ЛЕКЦИИ

Часть 1:

1. Понятие алгоритма
2. Идея алгоритма
3. Визуализация
4. Словесное представление

ПЛАН ЛЕКЦИИ

Часть 2:

5. Реализация алгоритма через параметрический цикл
   - 5.1 Блок-схема
   - 5.2 Подробный разбор элементов блок-схемы
   - 5.3 Программная реализация на языке С

Часть 3:

6. Реализация алгоритма через цикл с предусловием
   - 6.1 Блок-схема
   - 6.2 Подробный разбор элементов блок-схемы
   - 6.3 Программная реализация на языке С

ПОНЯТИЕ АЛГОРИТМА

Сортировка «Пузырьком» (англ Bubble Sort) – один из наиболее известных и простых алгоритмов сортировки. Он крайне лёгок в понимании, однако эффективен лишь при малых размерах массива.
Средняя сложность алгоритма – квадратичная, или же $O(n^2)$.
Алгоритм состоит в повторяющихся проходах по сортируемому массиву. На каждой итерации последовательно сравниваются соседние элементы, и, если порядок в паре неверный, то элементы меняют местами. За каждый проход по массиву как минимум один элемент встает на свое место: меньший по значению элемент (более «лёгкий») продвигается к началу массива, или же «всплывает, как пузырёк», (отсюда и название).

ИДЕЯ АЛГОРИТМА

Алгоритм основан на повторяющихся проходах по сортируемому массиву. За каждый проход последовательно сравниваются соседние элементы. Если порядок в паре неверный, то происходит обмен значений элементов. При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива («всплывает» до нужной позиции, как пузырёк в воде — отсюда и название алгоритма).

Проходы по массиву повторяются n-1 (где n – размер массива) раз или до тех пор, пока не будет перестановок, т.е. когда массив окажется отсортированным.

выбираем элемент

сравниваем его с соседним

если он больше, то переставляем элементы местами

СЛОВЕСНОЕ ПРЕДСТАВЛЕНИЕ

array – массив, n – длина массива

1 параметр внешнего цикла i = 0

2 если i < n – 1, то п.3, иначе п.9

3 параметр внутреннего цикла j = 0

4 если j < n – 1 – i, то п.5, иначе п.8

5 если array[j] > array[j + 1], то п.6, иначе п.7

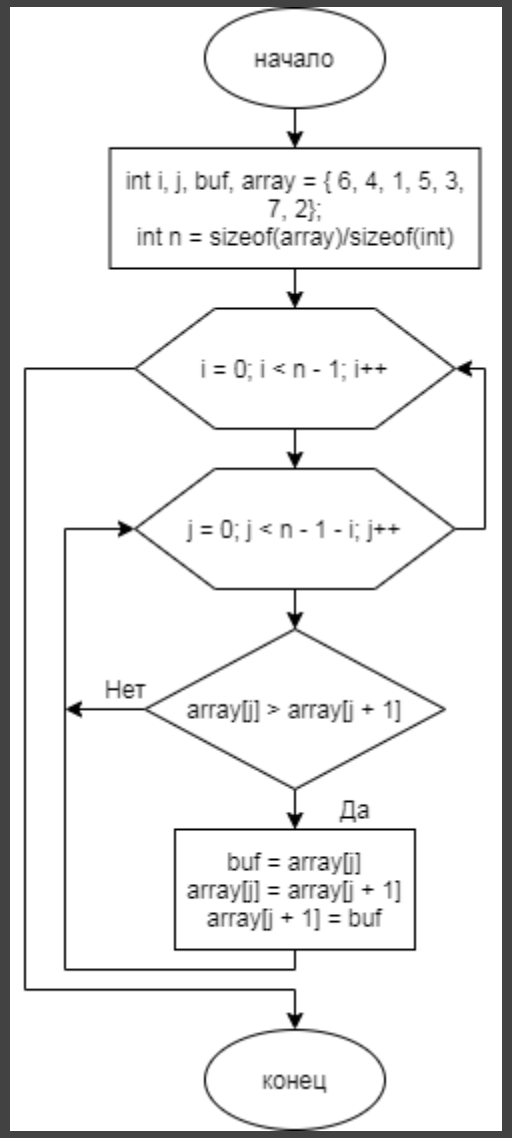
6 обмен значениями array[j] и array[j + 1]

7 j++, п.4

8 i++, п.2

9 конец алгоритма

БЛОК-СХЕМА

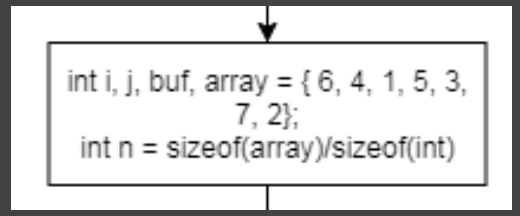


Блок-схема алгоритма с использованием элемента «модификация»

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

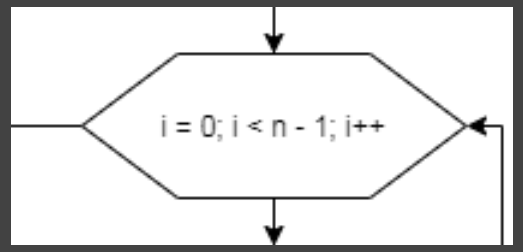
1 Элемент «пуск» используется для обозначения начала алгоритма.

2 Через следующий элемент, «процесс», реализовано объявление и инициализация переменных, используемых далее в программе.




```
int i, j, buf, array = { 6, 4, 1, 5, 3, 7, 2 };
int n = sizeof(array) / sizeof(int);
```

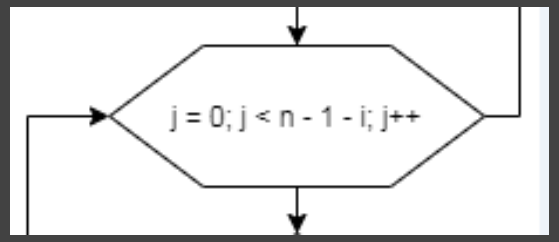
3 Третий элемент, «модификация», указывает на начало внешнего цикла. В коде эта часть реализована с помощью оператора for.




```
for(int i = 0; i < n - 1; i++){
    …
}
```

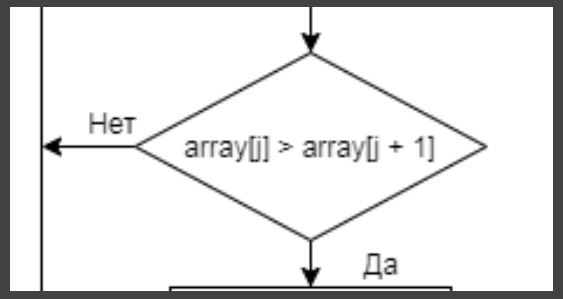
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

4 Следующий элемент «модификация» отвечает за начало внутреннего цикла.




```
for(j = 0; j < n - 1 - i; j++){
    …
}
```

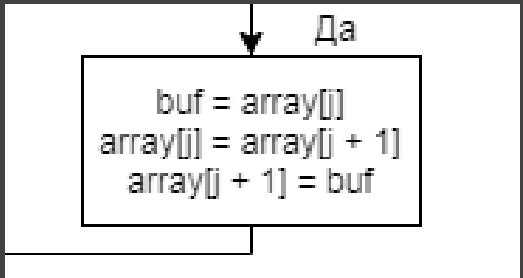
5 Далее происходит проверка условия array[j] > array[j + 1]. В блок-схеме это реализовано с помощью элемента «решение», в коде – с использованием оператора if. От элемента «решение» отходит две ветви – «да», указывающая на действия, выполняемые при верности условия, и «нет», направляющая на следующую итерацию внутреннего цикла.




```
if(array[j] > array[j + 1]){
    …
}
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

6 Далее идет элемент «процесс», через который реализована перестановка элементов array[j] и array[j + 1], в случае выполнения условия предыдущего пункта.




```
buf = array[j];
array[j] = array[j + 1];
array[j + 1] = buf;
```

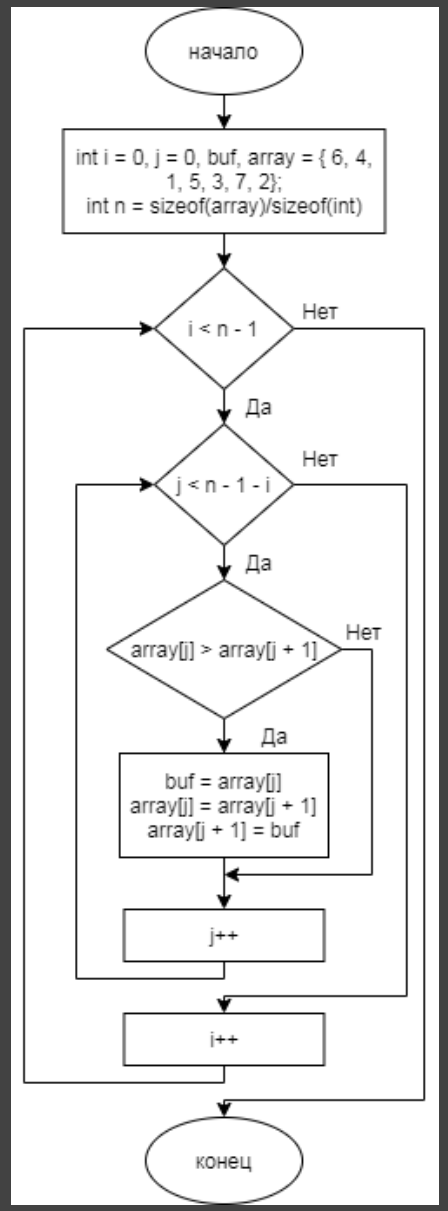
7 Происходит переход на следующую итерацию внутреннего цикла, по завершении которого – переход на следующую итерацию внешнего цикла. После окончания внешнего цикла происходит завершение алгоритма (в блок-схеме это реализовано через элемент «пуск»).

КОД С ПОМОЩЬЮ FOR

Программная реализация «Пузырька» через параметрический цикл

```
int main(){
    int i, j, buf, arr[] = { 6, 4, 1, 5, 3,
7, 2};
    int n = sizeof(arr) / sizeof(int);
    for(i = 0; i < n - 1; i++){
        for(j = 0; j < n - 1 - i; j++){
            if(arr[j] > arr[j + 1]){
                buf = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = buf;
            }
        }
    }
}
```

БЛОК-СХЕМА

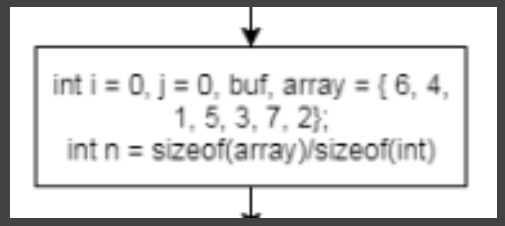


Блок-схема алгоритма с использованием элемента «решение»

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

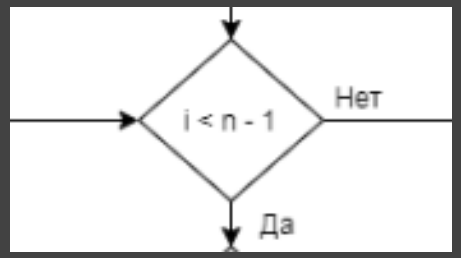
1 Элемент «пуск» используется для обозначения начала алгоритма.

2 Через следующий элемент, «процесс», реализовано объявление и инициализация переменных, используемых далее в программе.




```
int i = 0, j = 0, buf, array = { 6, 4, 1, 5, 3, 7, 2 };
int n = sizeof(array) / sizeof(int);
```

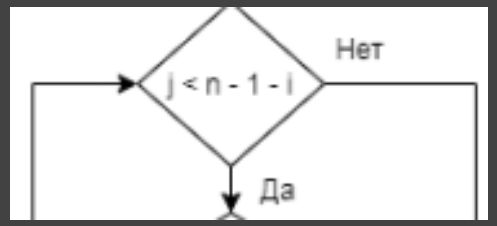
3 Начало внешнего цикла с предусловием в блок-схеме реализовано с помощью элемента «решение», в коде – с использованием оператора while. От «решения» отходят две ветви: «да», по которой идем в случае верности условия, и «нет», указывающая на действия, выполняемые при ложности условия.




```
while(i < n - 1){
    …
}
```

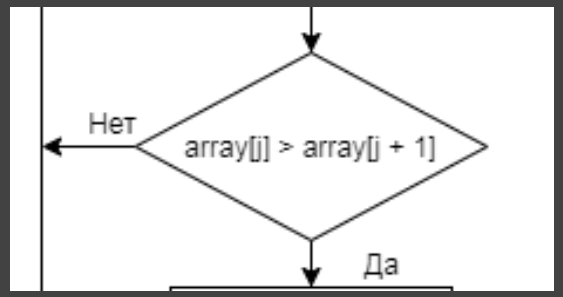
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

4 Через следующий элемент, «решение», реализовано начало внутреннего цикла. От элемента отходят две ветви: «да», по которой идем в случае верности условия, и «нет», указывающая на действия, выполняемые при ложности условия. В коде это реализовано при помощи оператора while.




```
while(j < n – 1- i){
    …
}
```

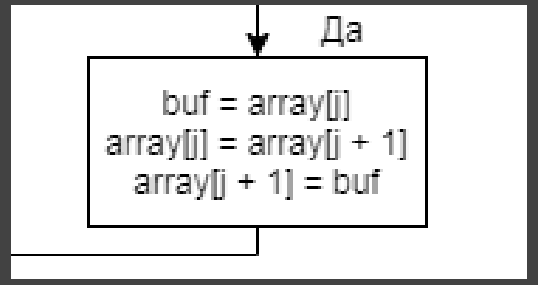
5 Далее происходит проверка условия array[j] > array[j + 1]. В блок-схеме это реализовано с помощью элемента «решение», в коде – с использованием оператора if. От элемента «решение» отходит две ветви – «да», указывающая на действия, выполняемые при верности условия, и «нет», направляющая на следующую итерацию внутреннего цикла.




```
if(array[j] > array[j + 1]){
    …
}
```

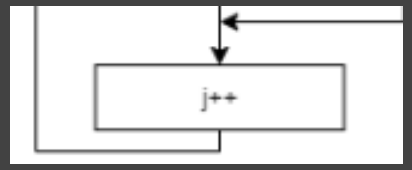
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

6 Далее идет элемент «процесс», через который реализована перестановка элементов array[j] и array[j + 1], в случае выполнения условия предыдущего пункта.



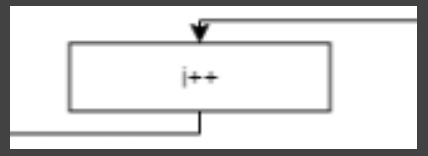

```
buf = array[j];
array[j] = array[j + 1];
array[j + 1] = buf;
```

7 Следующий элемент, «процесс», отвечает за увеличение на единицу переменной j.




```
j++;
```

8 Следующий элемент, «процесс», отвечает за увеличение на единицу переменной i.




```
i++;
```

9 Окончание алгоритма реализовано через элемент «пуск».

КОД С ПОМОЩЬЮ WHILE

Программная реализация «Пузырька» через цикл с предусловием

```
int main(){
    int i = 0, j = 0, buf, arr[] =
{ 6, 4, 1, 5, 3, 7, 2};
    int n = sizeof(arr) /
sizeof(int);
    while(i < n -1){
        while(j < n - 1 - i){
            if(arr[j] > arr[j +
1]){
                buf = arr[j];
                arr[j] = arr[j +
1];
                arr[j + 1] = buf;
            }
            j++;
        }
        i++;
    }
}
```

основы алгоритмизации и программирования

СОРТИРОВКА «РАСЧЁСКА»

ПЛАН ЛЕКЦИИ

Часть 1:

1. Понятие алгоритма

2. Идея алгоритма

3. Визуализация

4. Словесное представление алгоритма

ПЛАН ЛЕКЦИИ

Часть 2:

5. Реализация алгоритма через параметрический цикл:

5.1 Блок-схема

5.2 Подробный разбор элементов блок-схемы

5.3 Программная реализация на языке программирования С

ПЛАН ЛЕКЦИИ

Часть 2:

6. Реализация алгоритма через цикл с предусловием:

6.1 Блок-схема

6.2 Подробный разбор элементов блок-схемы

6.3 Программная реализация на языке программирования C

ПОНЯТИЕ АЛГОРИТМА

Сортировка расчёской (англ. *comb sort*) — это довольно упрощённый алгоритм сортировки, изначально спроектированный в 1980 г.
Сортировка расчёской улучшает сортировку пузырьком, и конкурирует с алгоритмами, подобными быстрой сортировке. Основная идея — устранить *черепах*, или маленькие значения в конце списка, которые крайне замедляют сортировку пузырьком.
В сортировке пузырьком, когда сравниваются два элемента, промежуток (расстояние друг от друга) равен 1. Основная идея сортировки расчёской в том, что этот промежуток может быть гораздо больше, чем единица.

ИДЕЯ АЛГОРИТМА

Алгоритм является модификацией «пузырька». Отличие алгоритмов состоит в том, что сравниваются не соседние элементы, а отстоящие друг от друга на определённую величину, или шаг (назовём его step). Алгоритм реализован с помощью двух циклов. Окончание внешнего цикла (и алгоритма) происходит тогда, когда step станет меньше 1. На первой итерации расстояние (step) максимально возможное (размер массива – 1), а на последующих итерациях оно изменяется по формуле step /= k (дробная часть отбрасывается). k – это фактор уменьшения, константа, равная 1.2473309 (при написании программы можно использовать примерное значение, равное 1.247). Во внутреннем цикле движение происходит от начала к концу, перемещаясь на step. Если значение текущего элемента больше, чем значение элемента через step шагов от текущего, то сравниваемые элементы меняются местами. Условием продолжения цикла является условие i < n – step (где i – номер текущего элемента).

Сравниваются элементы, отстоящие друг от друга на определённый шаг

Если текущий элемент больше следующего,
то они меняются местами

Массив отсортирован!

СЛОВЕСНОЕ ПРЕДСТАВЛЕНИЕ

array – массив, n – длина массива, k – фактор уменьшения, равный 1.247, step – шаг

1 расчет шага (step = n – 1)

2 если step >= 1, то п.3, иначе п.10

3 параметр внутреннего цикла i = 0

4 если step >= 1, то п.3, иначе переход к п.10

5 если i < n – step, то п.6, иначе п.9

6 если array[i] > array[i + step], то п.7, иначе п.8

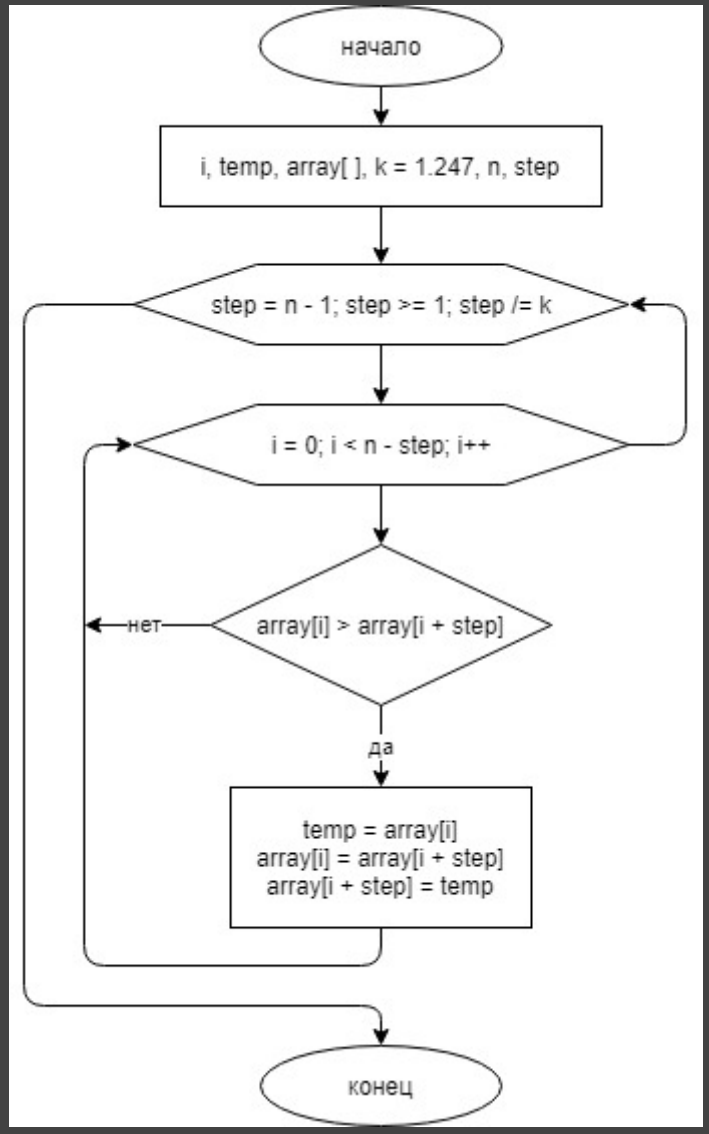
7 перестановка array[i] и array[i + step]

8 i++, п.4

9 step/=k, п.2

10 конец алгоритма

БЛОК-СХЕМА

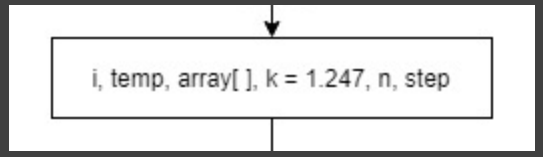


Блок-схема алгоритма с использованием элемента «модификация»

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

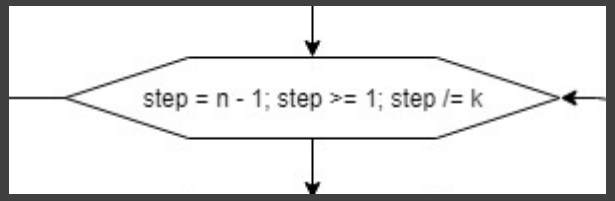
1 Первый элемент «пуск» мы используем для обозначения начала работы алгоритма.

2 Второй элемент «процесс» отвечает за объявление и инициализацию переменных, используемых в дальнейшем при работе алгоритма.




```
int i, temp, step;
int array[] = { 1, 4, 5, 0, 3, 2 };
int n = sizeof(array) / sizeof(int);
float k = 1.247;
```

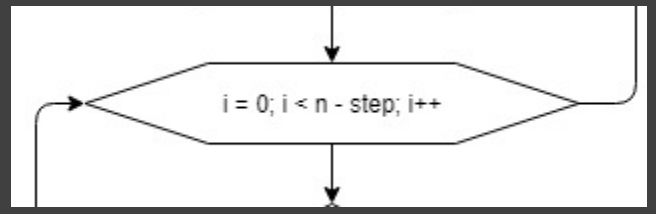
3 Третий элемент блок-схемы «модификация» отвечает за начало внешнего цикла. В коде «модификация» реализуется через параметрический цикл for.




```
for (step = n - 1; step >= 1; step /= k) {
    …
}
```

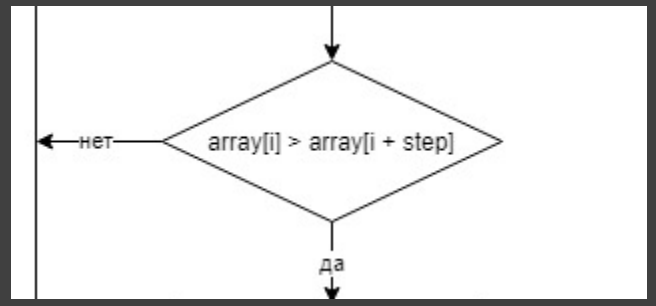
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

4 Четвёртый элемент, также «модификация», отвечает за начало внутреннего цикла.




```
for (i = 0; i < n - step; i++) {
        …
    }
```

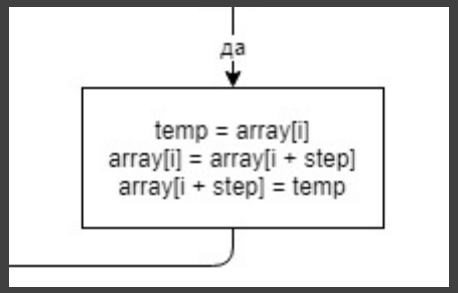
5 Элемент блок-схемы «решение» отвечает за реализацию ветвления алгоритма. Значения двух элементов массива сравниваются, и в зависимости от результата выполняются соответствующие действия.
В коде «решение» реализовано через неполный условный оператор if.




```
if (array[i] > array[i + step]) {
        …
    }
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

6 Элемент блок-схемы «процесс» отвечает за обмен значений элементов массива в случае истинности предшествующего условия.




```
temp = array[i];
array[i] = array[i + step];
array[i + step] = temp;
```

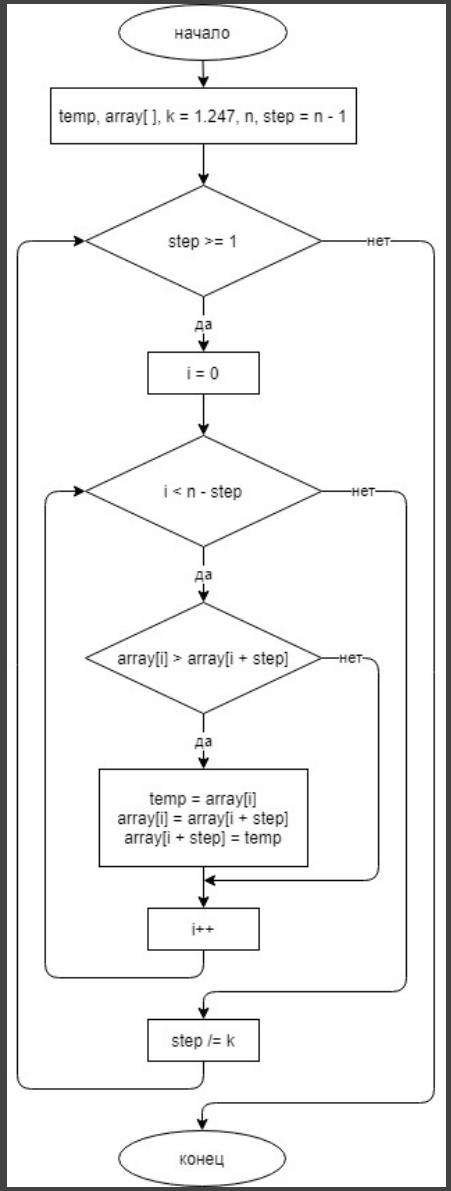
7 Последний элемент «пуск» завершает алгоритм.

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

```
int main() {
    int i, temp, step;
    int array[] = { 1, 4, 5, 0, 3, 2 };
    int n = sizeof(array) / sizeof(int);
    float k = 1.247;
    for (step = n - 1; step >= 1; step /= k) {
        for (i = 0; i < n - step; i++) {
            if (array[i] > array[i + step]) {
                temp = array[i];
                array[i] = array[i + step];
                array[i + step] = temp;
            }
        }
   }
   return 0;
}
```

Программная реализация «Расчёски» через параметрический цикл

БЛОК-СХЕМА

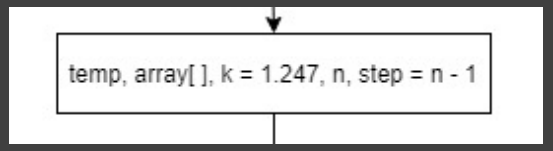


Блок-схема алгоритма с использованием элемента «решение»

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

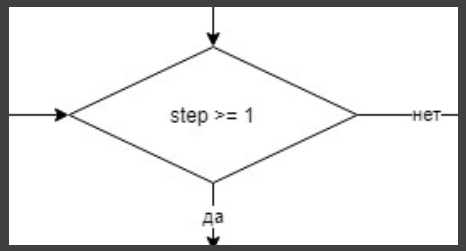
1 Первый элемент «пуск» мы используем для обозначения начала работы алгоритма.

2 Второй элемент «процесс» отвечает за объявление и инициализацию переменных, используемых в дальнейшем при работе алгоритма.




```
int i, temp;
int array[] = { 1, 4, 5, 0, 3, 2 };
int n = sizeof(array) / sizeof(int);
float k = 1.247;
int step = n - 1;
```

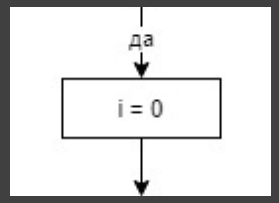
3 Элемент блок-схемы «решение» отвечает за начало внешнего цикла. Проверяется истинность условия step >= 1. Если она подтверждается, то тело цикла выполняется. В коде данный элемент реализован через цикл с предусловием while.




```
while (step >= 1) {
  …
}
```

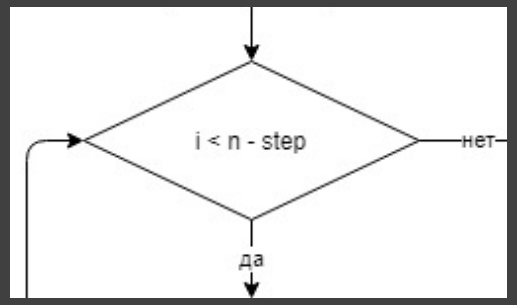
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

4 В следующем элементе блок-схемы «процесс» переменной i присваивается начальное значение.




```
i = 0;
```

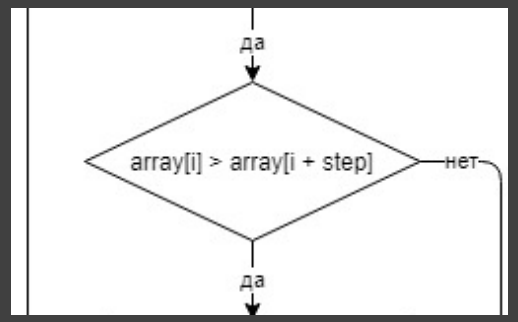
5 Ещё один элемент блок-схемы «решение» отвечает за начало внутреннего цикла. Так же как и в случае с внешним циклом элемент реализован через цикл с предусловием while. Проверяется истинность условия i < n – step, и в зависимости от результата проверки выполняется один из возможных вариантов вычислительного процесса.




```
while (i < n - step) {
  …
}
```

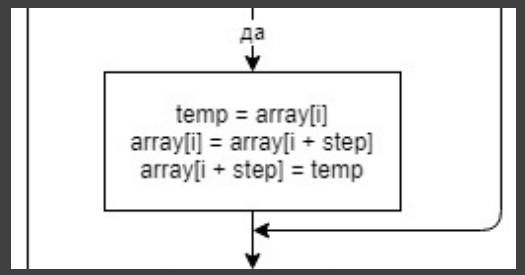
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

6 Очередной элемент блок-схемы «решение» отвечает за реализацию ветвления алгоритма. Значения двух элементов массива сравниваются, и в зависимости от результата выполняются соответствующие действия.
В коде реализовано через неполный условный оператор if.




```
if (array[i] > array[i + step])
{
   …
}
```

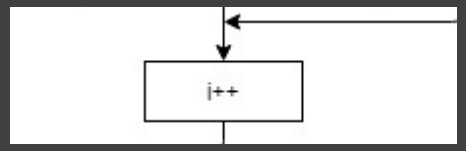
7 Элемент блок-схемы «процесс» отвечает за обмен значений элементов массива в случае истинности предшествующего условия.




```
temp = array[i];
array[i] = array[i + step];
array[i + step] = temp;
```

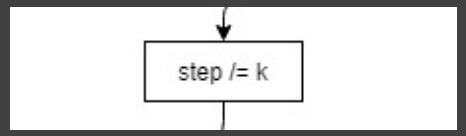
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

8 Вне зависимости от результата проверки условия array[i] > array[i + step] будет произведено инкрементирование переменной i (увеличение значения параметра внутреннего цикла на 1). «Процесс» – элемент блок-схемы отвечающий за это действие.




```
i++;
```

9 Следующий элемент блок-схемы «процесс» выполняет действие в отношении параметра внешнего цикла, изменяя его значение.




```
step /= k;
```

10 Последний элемент блок-схемы «пуск» завершает алгоритм.

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

```
int main() {
    int i, temp;
    int array[] = { 1, 4, 5, 0, 3, 2 };
    int n = sizeof(array) / sizeof(int);
    float k = 1.247;
    int step = n - 1;
    while (step >= 1) {
          i = 0;
          while (i < n - step) {
              if (array[i] > array[i + step]) {
                  temp = array[i];
                  array[i] = array[i + step];
                  array[i + step] = temp;
              }
              i++;
        }
        step /= k;
    }
    return 0;
}
```

Программная реализация «Расчёски» через цикл с предусловием

основы алгоритмизации и программирования

СОРТИРОВКА «ВСТАВКАМИ»

ПЛАН ЛЕКЦИИ

Часть 1:

1. Понятие алгоритма

2. Идея алгоритма

3. Визуализация

4. Словесное представление алгоритма

ПЛАН ЛЕКЦИИ

Часть 2:

5. Реализация алгоритма через цикл с предусловием:

5.1 Блок-схема

5.2 Подробный разбор элементов блок-схемы

5.3 Программная реализация на языке программирования C

ПЛАН ЛЕКЦИИ

Часть 2:

6. Реализация алгоритма через параметрический цикл:

6.1 Блок-схема

6.2 Подробный разбор элементов блок-схемы

6.3 Программная реализация на языке программирования C

ПОНЯТИЕ АЛГОРИТМА

Сортировка вставками (англ. *Insertion sort*) — алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов. Вычислительная сложность — $O(n^2)$.

ИДЕЯ АЛГОРИТМА

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.

Массив отсортирован!

СЛОВЕСНОЕ ПРЕДСТАВЛЕНИЕ

n – длина массива

1 Номер анализ. эл-та равен единице

2 Если номер анализ. эл-та < n, то п. 3, иначе п. 9

3 Запоминаем значение анализ. эл-та

4 Номер текущего эл-та равен номеру анализ. эл-та

5 Если номер текущего эл-та > 0 и значение анализ. эл-та < значения эл-та, предшествующего текущему, то п. 6, иначе п. 7

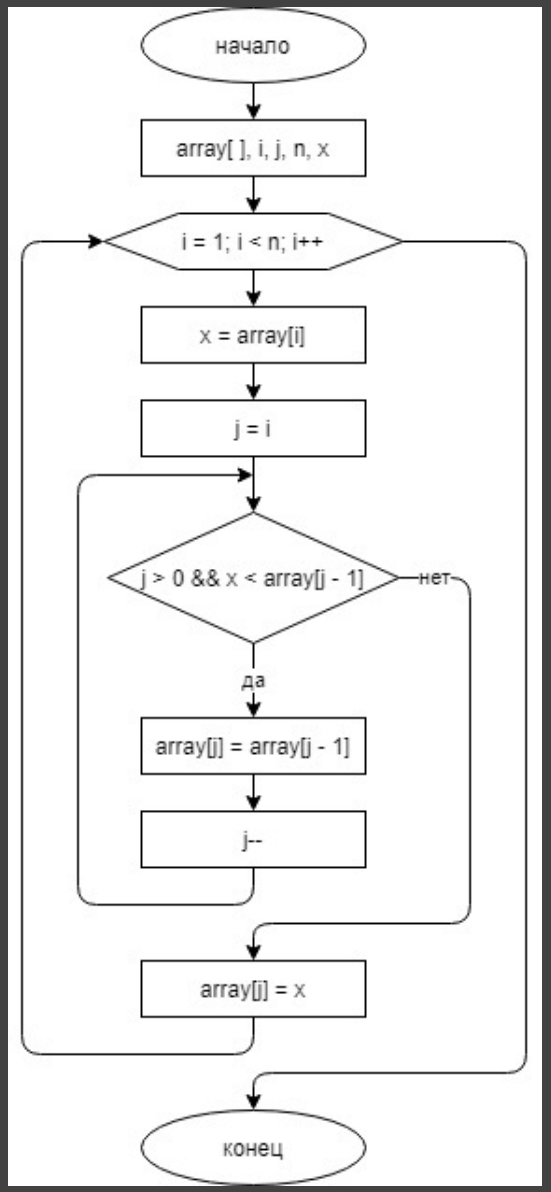
6 Значение текущего эл-та равно значению элемента с номером (н.т.э - 1), п. 5

7 В текущий элемент записать значение анализ. эл-та

8 Уменьшить на 1 номер анализ. эл-та, п. 2

9 Конец алгоритма

БЛОК-СХЕМА

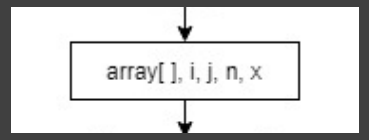


Блок-схема алгоритма с использованием элемента «решение»

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

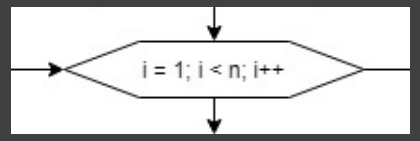
1 Первый элемент «пуск» мы используем для обозначения начала работы алгоритма.

2 Второй элемент «процесс» отвечает за объявление и инициализацию переменных, используемых в дальнейшем при работе алгоритма.




```
int i, j, x;
int array[ ] = { 1, 4, 6, 7, 8, 3};
int n = sizeof(array) / sizeof(int);
```

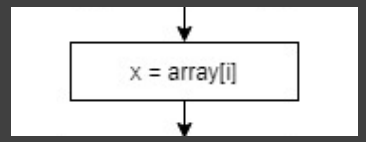
3 Третий элемент блок-схемы «модификация» отвечает за начало цикла. Начальное значение номера анализируемого элемента равно единице. В коде «модификация» реализуется через параметрический цикл for.




```
for (i = 1; i < n; i++) {
    …
}
```

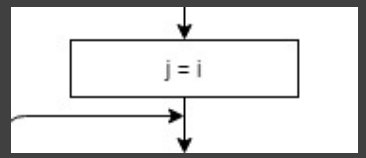
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

4 В четвёртом элементе, «процессе», запоминаем значение анализируемого элемента.



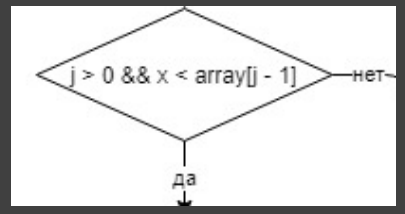

```
x = array[i];
```

5 Пятый элемент «процесс». Номер текущего элемента равен номеру анализируемого элемента.




```
j = i;
```

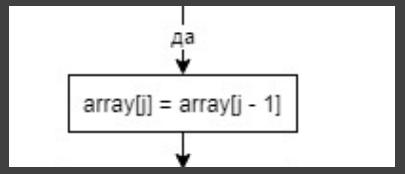
6 Шестой элемент блок-схемы «решение» отвечает за реализацию ветвления алгоритма. Проверяется одновременное выполнение двух условий, и в зависимости от результата выполняются соответствующие действия.
В коде «решение» реализовано через неполный условный оператор if.




```
while (j > 0 && x < array[j - 1] {
        …
}
```

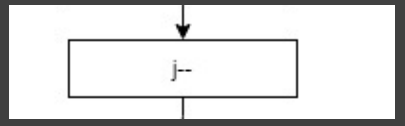
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

7 В случае истинности предыдущего условия вычислительный процесс переходит к элементу «процесс». В нём значение текущего элемента становится равно значению предшествующего элемента.




```
array[j] = array[j - 1];
```

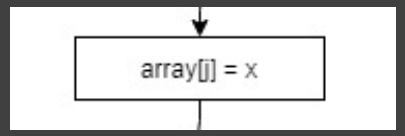
8 Ещё один элемент «процесс». В нём происходит уменьшение номера текущего элемента на 1. После вычислительный процесс переходит к «решению» и, соответственно, к проверке условий.




```
j--;
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

9 В случае невыполнения условий в «процессе» значению текущего элемента присваивается запомненное ранее значение анализируемого элемента.




```
array[j] = x;
```

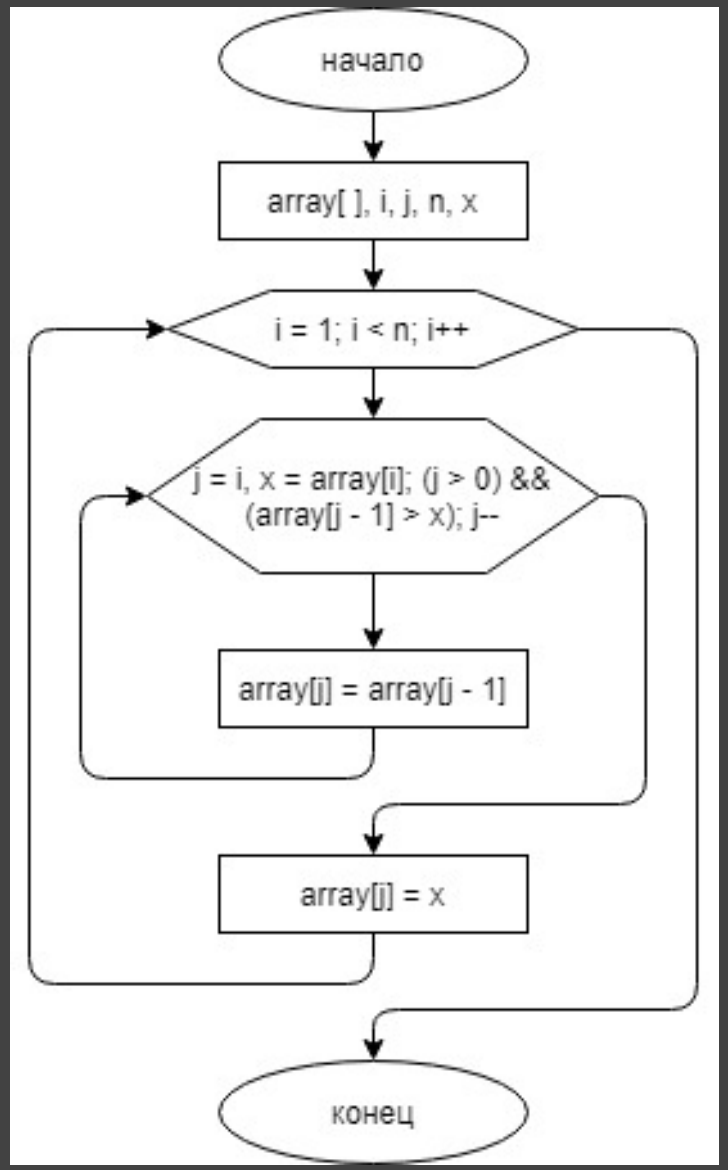
10 Последний элемент блок-схемы «пуск» завершает алгоритм.

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

Программная реализация сортировки
вставками через цикл с предусловием

```
int main()
{
    int i, j, x;
    int array[ ] = { 1, 4, 6, 7, 8, 3};
    int n = sizeof(array) / sizeof(int);
    for (i = 1; i < n; i++) {
        x = array[i];
        j = i;
        while (j > 0 && x < array[j - 1]) {
            array[j] = array[j - 1];
            j--;
        }
        array[j] = x;
    }
    return 0;
}
```

БЛОК-СХЕМА

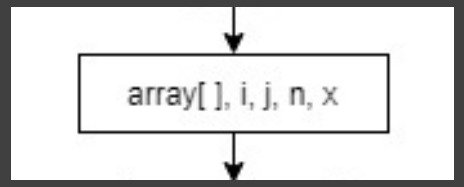


Блок-схема алгоритма с использованием элемента «модификация»

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

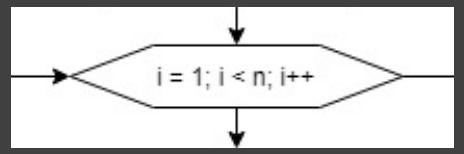
1 Первый элемент «пуск» мы используем для обозначения начала работы алгоритма.

2 Второй элемент «процесс» отвечает за объявление и инициализацию переменных, используемых в дальнейшем при работе алгоритма.




```
int i, j, x;
int array[ ] = { 1, 4, 6, 7, 8, 3};
int n = sizeof(array) / sizeof(int);
```

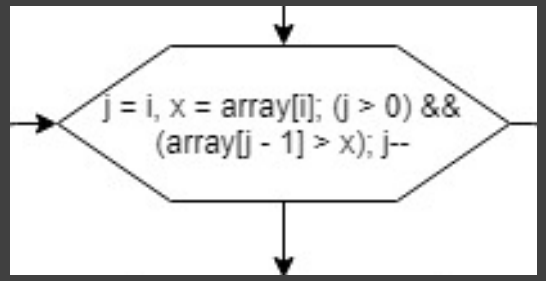
3 Третий элемент блок-схемы «модификация» отвечает за начало внешнего цикла. Начальное значение номера анализируемого элемента равно единице. В коде «модификация» реализуется через параметрический цикл for.




```
for (i = 1; i < n; i++) {
   …
}
```

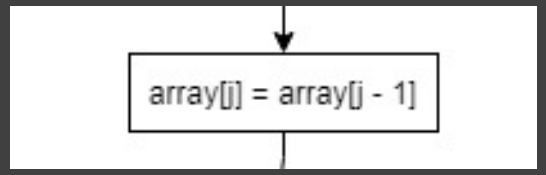
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

4 Четвёртый элемент, также «модификация», отвечает за начало внутреннего цикла. В нём значению номера текущего элемента присваивается значение номера анализируемого элемента, запоминается значение анализ. эл-та с помощью вспомогательной переменной x, а также производится проверка одновременного выполнения двух условий.




```
for(j = i, x = array[i]; (j > 0) && (array[j -
1] > x); j--) {
   …
}
```

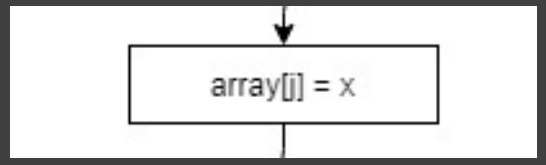
5 Пятый элемент «процесс». В нём значение текущего элемента становится равно значению предшествующего элемента.




```
array[j] = array[j - 1];
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

6 После выхода из внутреннего цикла в «процессе» значению текущего элемента присваивается запомненное ранее значение анализируемого элемента.




```
array[j] = x;
```

7 Последний элемент блок-схемы «пуск» завершает алгоритм.

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

Программная реализация сортировки
вставками через параметрические циклы

```
int main()
{
    int i, j , x;
    int array[] = { 1, 4, 6, 7, 8, 3 };
    int n = sizeof(array) / sizeof(int);
    for (i = 1; i < n; i++) {
        for (j = i, x = array[i]; (j > 0) && (array[j - 1] > x); j--) {
            array[j] = array[j - 1];
        }
        array[j] = x;
    }
    return 0;
}
```

основы алгоритмизации и программирования

СОРТИРОВКА ШЕЛЛА

ПЛАН ЛЕКЦИИ

Часть 1:

1. Понятие алгоритма

2. Идея алгоритма

3. Визуализация

4. Словесное представление алгоритма

ПЛАН ЛЕКЦИИ

Часть 2:

5. Реализация алгоритма через параметрический цикл:

5.1 Блок-схема

5.2 Подробный разбор элементов блок-схемы

5.3 Программная реализация на языке программирования С

ПЛАН ЛЕКЦИИ

Часть 2:

6. Реализация алгоритма через цикл с предусловием:

6.1 Блок-схема

6.2 Подробный разбор элементов блок-схемы

6.3 Программная реализация на языке программирования C

ПОНЯТИЕ АЛГОРИТМА

Сортировка Шелла (англ. *Shell sort*) — алгоритм сортировки, являющийся усовершенствованным вариантом сортировки вставками. Идея метода Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга. Иными словами — это сортировка вставками с предварительными «грубыми» проходами. Аналогичный метод усовершенствования пузырьковой сортировки называется сортировка расчёской.

ИДЕЯ АЛГОРИТМА

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.

Сравнивая

6	4	1	5	3	7	2

Массив отсортирован!

СЛОВЕСНОЕ ПРЕДСТАВЛЕНИЕ

n – длина массива, d – шаг

1 Рассчитываем начальное значение шага: d = n / 2

2 Если d > 0, то п.3, иначе п. 14

3 Номер анализ. эл-та = d

4 Если номер анализ. элемента < n, то п.5, иначе п.13

5 Запоминаем значение анализ. элемента

6 Номер текущего элемента = номеру анализ. элемента

7 Если номер текущего элемента >= d, то п.8, иначе п.11

8 Если значение текущего элемента < значение элемента с номером (текущего элемента – d), то п.9, иначе п.11

9 Значение т.э. = значение эл-та с номером (т.э – d)

СЛОВЕСНОЕ ПРЕДСТАВЛЕНИЕ

n – длина массива, d – шаг

10 Номер т.э = номер (т.э – d), п. 7

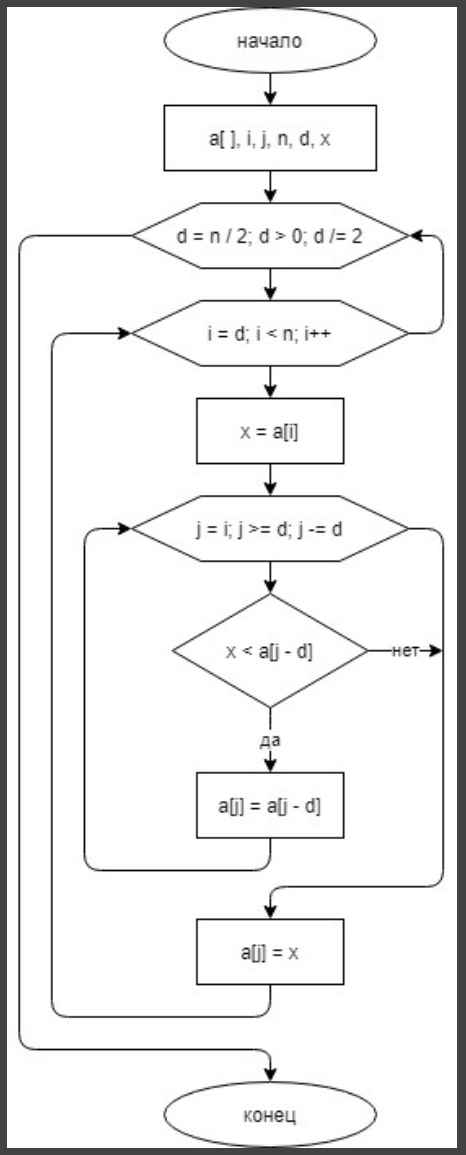
11 Значение т.э = значение анализ. элемента

12 i++, п. 4

13 d /= 2, п. 2

14 Конец алгоритма

БЛОК-СХЕМА

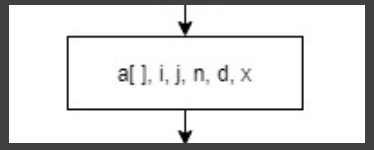


Блок-схема алгоритма с использованием элемента «модификация»

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

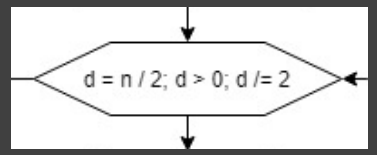
1 Первый элемент «пуск» мы используем для обозначения начала работы алгоритма.

2 Второй элемент «процесс» отвечает за объявление и инициализацию переменных, используемых в дальнейшем при работе алгоритма.




```
int i, x, d, j;
int a[ ] = { 1, 4, 6, 7, 8, 3};
int n = sizeof(a) / sizeof(int);
```

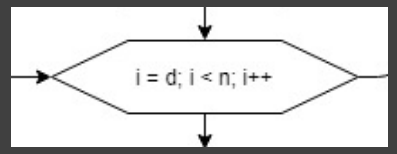
3 Третий элемент блок-схемы «модификация» отвечает за начало цикла. Шагу присваивается начальное значение. В коде «модификация» реализуется через параметрический цикл for.




```
for (d = n / 2; d > 0; d /= 2) {
    …
}
```

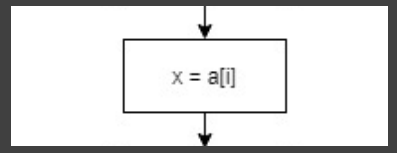
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

4 Четвёртый элемент, также «модификация», отвечает за начало цикла.



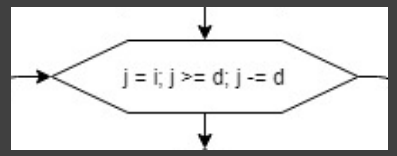

```
for (i = d; i < n; i++) {
      …
   }
```

5 Пятый элемент «процесс». В нем мы запоминаем значение анализируемого элемента, используя для этого вспомогательную переменную x.




```
x = a[i];
```

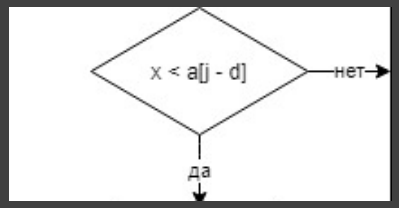
6 Шестой элемент очередная «модификация. Номеру текущего элемента присваивается значение номера анализируемого элемента.




```
for (j = i; j >= d; j -= d) {
      …
   }
```

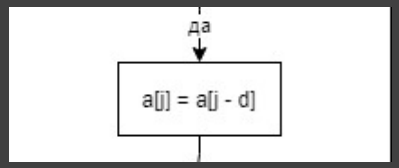
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

7 Элемент блок-схемы «решение» отвечает за реализацию ветвления алгоритма. Значения двух элементов массива сравниваются, и в зависимости от результата выполняются соответствующие действия.
В коде «решение» реализовано через неполный условный оператор if.




```
if (x < a[j - d]){
   …
}
```

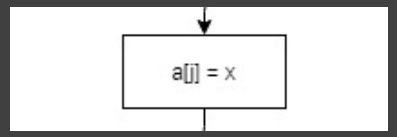
8 Элемент блок-схемы «процесс» отвечает за обмен значений элементов массива в случае истинности предшествующего условия.




```
a[j] = a[j - d];
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

9 В случае невыполнения предшествующего условия (или при завершении цикла) значение текущего элемента становится равно запомненному значению анализируемого элемента.




```
a[j] = x;
```

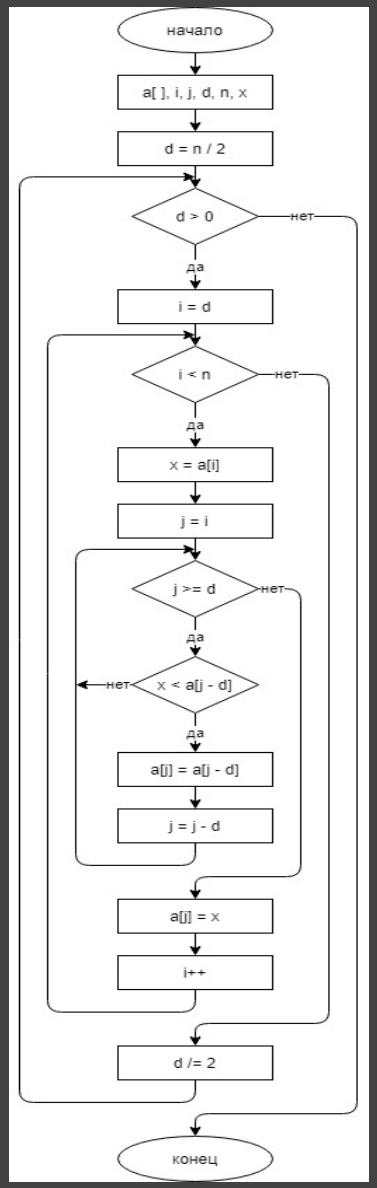
10 Последний элемент блок-схемы «пуск» завершает алгоритм.

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

```
int main()
{
    int i, x, d, j;
    int a[ ] = { 1, 4, 6, 7, 8, 3};
    int n = sizeof(a) / sizeof(int);
    for (d = n / 2; d > 0; d /= 2) {
        for (i = d; i < n; i++) {
            x = a[i];
            for (j = i; j >= d; j -= d) {
                if (x < a[j - d])
                    a[j] = a[j - d];
                else break;
            }
            a[j] = x;
        }
    }
    return 0;
}
```

Программная реализация сортировки Шелла через параметрический цикл

БЛОК-СХЕМА



Блок-схема алгоритма с использованием элемента «решение»

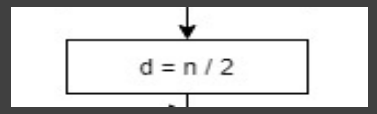
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

1 Первый элемент «пуск» мы используем для обозначения начала работы алгоритма.

2 Второй элемент «процесс» отвечает за объявление и инициализацию переменных, используемых в дальнейшем при работе алгоритма.


```
int i, x, d, j;
int a[ ] = { 1, 4, 6, 7, 8, 3};
int n = sizeof(a) / sizeof(int);
```

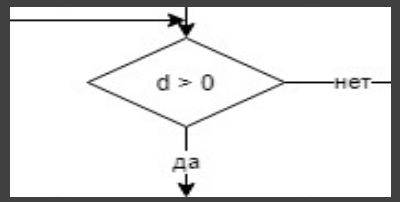
3 Третий элемент блок-схемы, также «процесс». В нём переменной шага присваивается начальное значение.




```
d = n / 2;
```

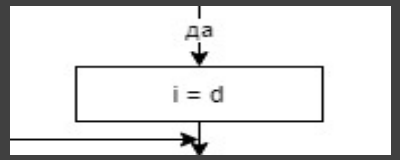
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

4 Четвёртый элемент «решение» отвечает за начало цикла. Элемент реализован через цикл с предусловием while. Проверяется истинность условия d > 0, и в зависимости от результата проверки выполняется один из возможных вариантов вычислительного процесса.




```
while (d > 0) {
   …
}
```

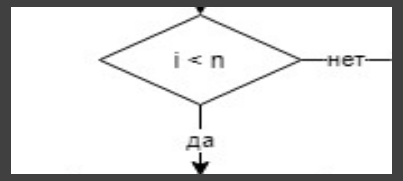
5 В случае истинности предшествующего условия в элементе «процесс» номер анализируемого элемента приравнивается значению шага.




```
i = d;
```

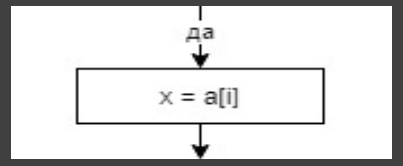
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

6 Шестой элемент «решение» отвечает за начало цикла. Проверяется истинность условия i < n, и в зависимости от результата проверки выполняется один из возможных вариантов вычислительного процесса.




```
while (i < n) {
   …
}
```

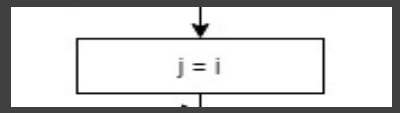
7 В случае истинности предыдущего условия вычислительный процесс переходит к элементу «процесс». В нём мы запоминаем значение анализируемого элемента, используя для этого вспомогательную переменную x.




```
x = a[i];
```

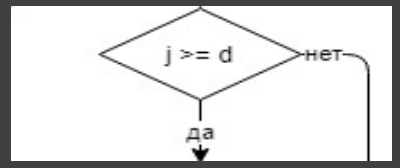
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

8 Следующий элемент «процесс». В нём номеру текущего элемента присваивается значение номера анализируемого элемента.



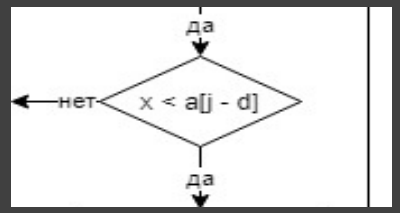

```
j = i;
```

9 Девятый элемент «решение» отвечает за начало цикла. Проверяется истинность условия j >= d.




```
while (j >= d) {
  …
}
```

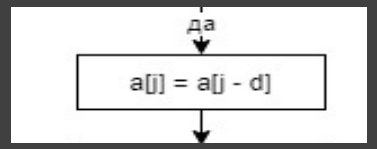
10 Десятый элемент блок-схемы «решение» отвечает за реализацию ветвления алгоритма. Значения двух элементов массива сравниваются, и в зависимости от результата выполняются соответствующие действия.
В коде «решение» реализовано через неполный условный оператор if.




```
if (x < a[j – d]) {
  …
}
```

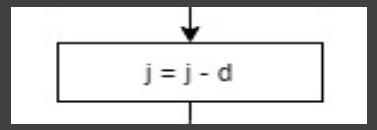
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

11 В случае истинности предыдущего условия вычислительный процесс переходит к элементу «процесс». В нём происходит обмен значений элементов.



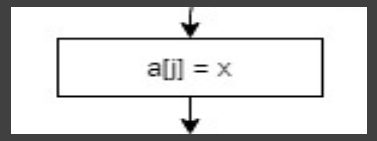

```
a[j] = a[j - d];
```

12 В следующем элементе, также «процессе». Номеру текущего элемента присваивается новое значение.




```
j = j - d;
```

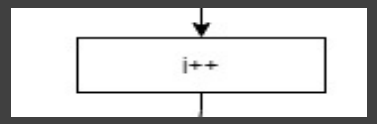
13 В случае невыполнения условия j >= d вычислительный процесс переходит в этому элементу «процесс». В нём значение текущего элемента становится равно запомненному значению анализируемого элемента.




```
a[j] = x;
```

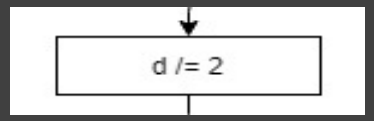
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

14 В следующем элементе выполняется увеличение номера анализируемого элемента на 1 для последующего прохождения тела цикла.




```
i++;
```

15 В случае невыполнения условия i < n, вычислительный процесс переходит в этому элементу «процесс». В нём вычисляется новое значение шага.




```
d /= 2;
```

16 В случае невыполнения условия d > 0, происходит завершение алгоритма, что и отражает последний элемент блок-схемы «пуск».

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

```
int main()
{
    int d, i, x, j;
    int a[] = { 1, 4, 6, 7, 8, 3 };
    int n = sizeof(a) / sizeof(int);
    d = n / 2;
    while (d > 0) {
        i = d;
         while (i < n) {
                x = a[i];
                j = i;
                while (j >= d) {
                        if (x < a[j - d]) {
                                a[j] = a[j - d];
                                j = j - d;
                        }
                        else break;
                }
                a[j] = x;
                i++;
        }
        d = d / 2;
    }
    return 0;
}
```

Программная реализация сортировки Шелла через цикл с предусловием

основы алгоритмизации и программирования

СОРТИРОВКА «ВЫБОРОМ»

ПЛАН ЛЕКЦИИ

Часть 1:

1. Понятие алгоритма
2. Идея алгоритма
3. Визуализация
4. Словесное представление алгоритма

Часть 2:

5. Блок-схема
6. Подробный разбор элементов блок-схемы
7. Программная реализация на языках программирования C, C++; C#; Java

ПОНЯТИЕ АЛГОРИТМА

Сортировка выбором (*Selection sort*) — алгоритм сортировки. Может быть как устойчивый, так и неустойчивый. На массиве из n элементов имеет время выполнения в худшем, среднем и лучшем случае $\Theta(n^2)$, предполагая что сравнения делаются за постоянное время

Это возможно, самый простой в реализации алгоритм сортировки. Как и в большинстве других подобных алгоритмов, в его основе лежит операция сравнения. Сравнивая каждый элемент с каждым, и в случае необходимости производя обмен, метод приводит последовательность к необходимому упорядоченному виду.

ИДЕЯ АЛГОРИТМА

Идея алгоритма очень проста. Пусть имеется массив A размером N, тогда сортировка выбором сводится к следующему:

• берем первый элемент последовательности A[i], здесь i – номер элемента, для первого i равен 1;

• находим минимальный (максимальный) элемент последовательности и запоминаем его номер;

• если номер первого элемента и номер найденного элемента не совпадают, тогда два этих элемента обмениваются значениями, иначе никаких манипуляций не происходит;

• увеличиваем i на 1 и продолжаем сортировку оставшейся части массива.

С каждым последующим шагом размер подмассива, с которым работает алгоритм, уменьшается.

Находим первый элемент подмассива

Находим первый элемент подмассива

Меняем местами первый элемент в подмассиве
и найденный минимум

Сравниваем каждый следующий элемент с
сохраненным (т.е. с последним наименьшим в
подмассиве, и запоминаем наименьший)

СЛОВЕСНОЕ ПРЕДСТАВЛЕНИЕ

array – массив, N- длина массива, i,j- индексы массивов, min – индекс локального минимума

1 Сортировка начинается с первого элемента i=0

2 Если i < N - 1, то **п. 3**, иначе к пункту **12**

3 min = i, j = i + 1

4 Если j < N, то к пункту , иначе к пункту **9**

5 Ищем локальный минимум. Если array[j] < array[min], то к пункту **6**, иначе к пункту **7**

6 Запоминаем новый индекс (min = j)

7 j++

8 К пункту 4.

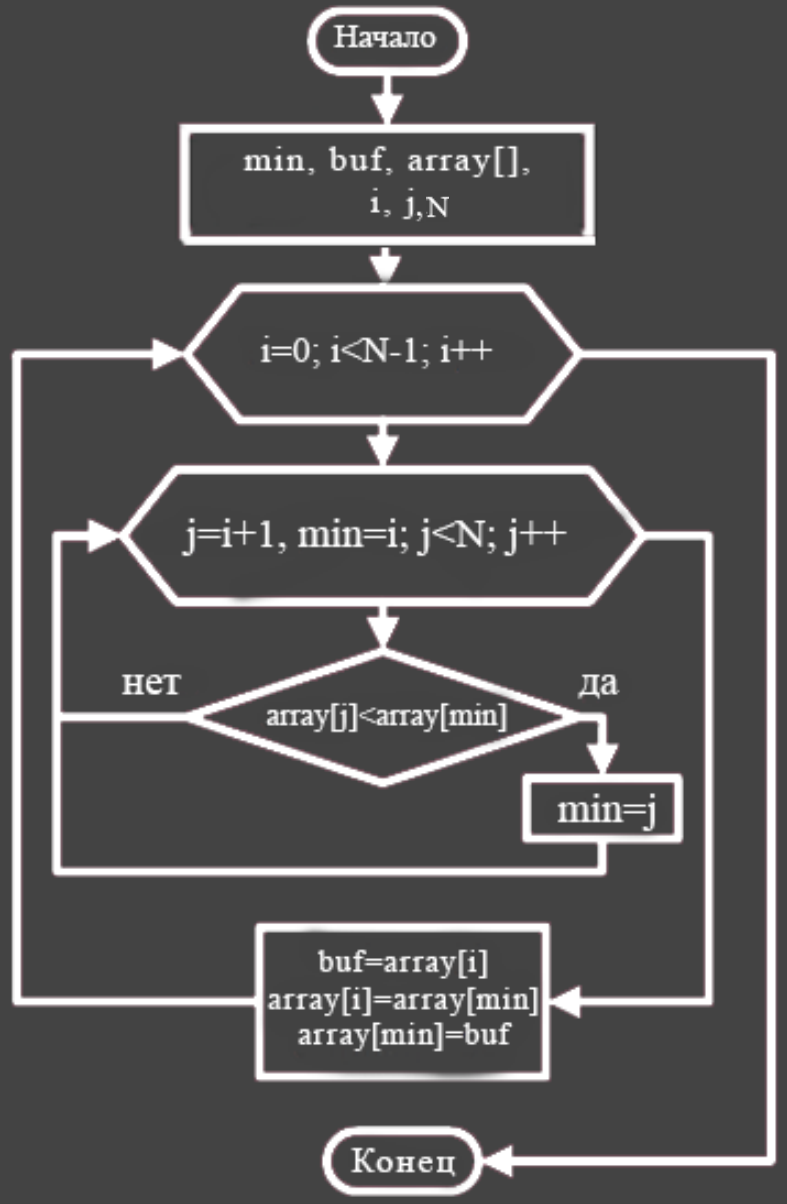
9 Обмен значениями array[i] и array[min].

10 i++

11 К пункту **2**

12 Конец алгоритма

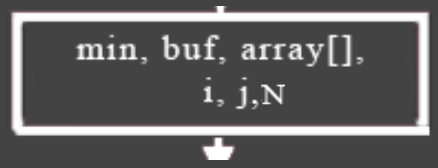
БЛОК-СХЕМА


Начало
min, buf, array[],
i, j,N
i=0; i<N-1; i++
j=i+1, min=i; j<N; j++
нет
array[j]<array[min]
да
min=j
buf=array[i]
array[i]=array[min]
array[min]=buf
Конец

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

1 Первый элемент («пуск») мы используем для обозначения начала работы алгоритма.

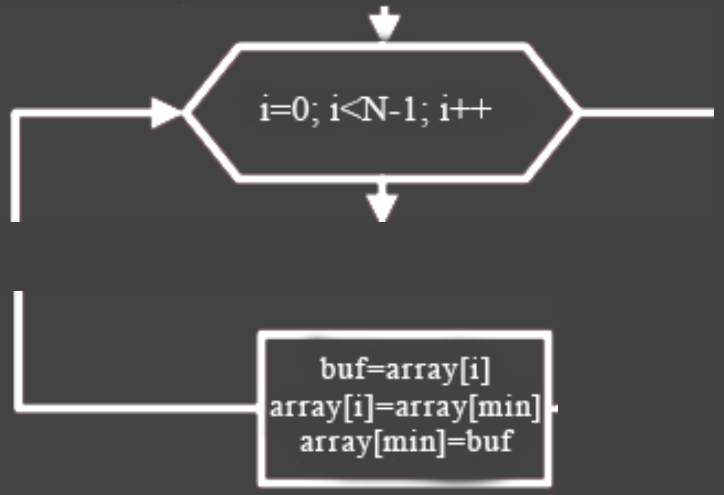
2 Во втором элементе, называемом «процесс», мы задаем переменные, которые в дальнейшем будем применять для работы алгоритма.




```
int N, min, buf, i, j, arr[] = { 5, 7, 8, 4, 9 };
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

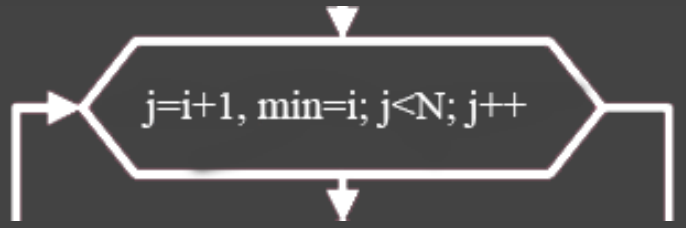
3 Третьим по очереди мы применяем элемент «цикл» для проведения нескольких одинаковых действий и проверки условия: от первого элемента массива до последнего проводится проверка внутри цикла. От «цикла» отходит две ветви: по первой мы идем в случае, если результат проверки условия положительный (заход в цикл и «проход» по всему массиву и процесс сортировки), по второй – если отрицательный (элементы массива закончились).




```
for (i=0; i<N-1; i++) {
    …
    buf=arr[i];
    arr[i]=arr[min];
    arr[min]=buf;
}
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

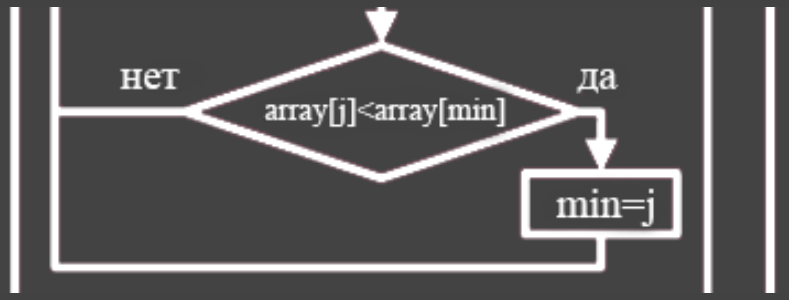
4 Следующим мы используем такой элемент блок-схемы как «цикл» для того, чтобы произвести несколько раз одно и то же действие с выполнением некоторого условия.




```
for (j=i+1, min=i; j<N; j++){
    …
}
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

5 Далее мы еще раз используем элемент «условие» для проверки условия и нахождения локального минимума. От «условия» отходит две ветви: по первой мы идем в случае, если результат проверки условия положительный(запоминаем номер элемента локального минимума), по второй – если отрицательный .




```
if (arr[j]<arr[min])
    min=j;
```

6 Последний элемент («пуск») мы применяем для обозначения конца работы алгоритма.

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ С

```
int main() {
  int i, j, min, buf, N = 7;
  int arr[] = {6, 4, 1, 5, 3, 7, 2};

  for (i = 0; i < N - 1; i++) {
       for (j = i + 1, min = i; j < N; j++){
           if (arr[j] < arr[min])
               min = j;
       }
       buf = arr[i];
       arr[i] = arr[min];
       arr[min] = buf;
   }
   return 0;
}
```

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ C++

```
#include <iostream>
using namespace std;

int main()
{
    int N = 7, j, i, buf, min;
    arr = new int[6, 4, 1, 5, 3, 7, 2];

    for (i = 0; i < N - 1; i++) {
        for (j = i + 1, min = i; j < N;j++) {
            if (arr[j]< arr[min])
            min = j;
        }
        buf = arr[i];
        arr[i] = arr[min];
        arr[min] = buf;
    }
    return 0;
}
```

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ C#

```
static void Main(string[] args)
{
     int[] a = {6, 4, 1, 5, 3, 7, 2};
     int i, j, min, temp;
     for(i = 0; i < a.Length - 1; i++)
     {
         min = i; //устанавливаем начальное
значение минимального индекса

 //находим минимальный индекс элемента
         for (j = i + 1; j < a.Length; j++)
         {
             if (a[j] < a[min])
                 min = j;
         }
//продолжение кода
//меняем значения местами
                 temp = a[i];
                 a[i] = a[min];
                 a[min] = temp;
                }

                Console.ReadKey();
    }
  }
}
```

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ JAVA

```
public static void sort(int[] arr) {
        for (int min = 0; min < arr.length - 1; min++) {
            int i = min;
            for (int j = min + 1; j < arr.length; j++) {
                if (arr[j] < arr[i]) {
                i = j;
                }
            }
            int buf = arr[min];
            arr[min] = arr[i];
            arr[i] = buf;
        }
 }
```

основы алгоритмизации и программирования

СОРТИРОВКА «ГНОМЬЯ»

ПЛАН ЛЕКЦИИ

Часть 1:

1. Понятие алгоритма
2. Идея алгоритма
3. Визуализация
4. Словесное представление алгоритма

Часть 2:

5. Блок-схема
6. Подробный разбор элементов блок-схемы
7. Программная реализация на языках программирования C, C++; C#; Java

ПОНЯТИЕ АЛГОРИТМА

Гномья сортировка (англ. *Gnome sort*) — алгоритм сортировки, похожий на сортировку вставками, но в отличие от последней перед вставкой на нужное место происходит серия обменов, как в сортировке пузырьком. Название происходит от предполагаемого поведения садовых гномов при сортировке линии садовых горшков.

Алгоритм концептуально простой, не требует вложенных циклов. Время работы $O(n^2)$. На практике алгоритм может работать так же быстро, как и сортировка вставками.

ИДЕЯ АЛГОРИТМА

Идея алгоритма очень проста. Пусть имеется массив A размером N, тогда сортировка выбором сводится к следующему:

• 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 ПРЕДСТАВЛЕНИЕ

arr – массив, N- длина массива, i,j- индексы массивов, min – индекс локального минимума

1 Сортировка начинается со второго и третьего элементов i=1, j=2;

2 Если i < N, то к пункту **3**, иначе к пункту **9**

3 если arr[i - 1] > arr[i], то к пункту **4**, иначе к пункту **7**

4 Меняем местами значения arr[i] и arr[i - 1]

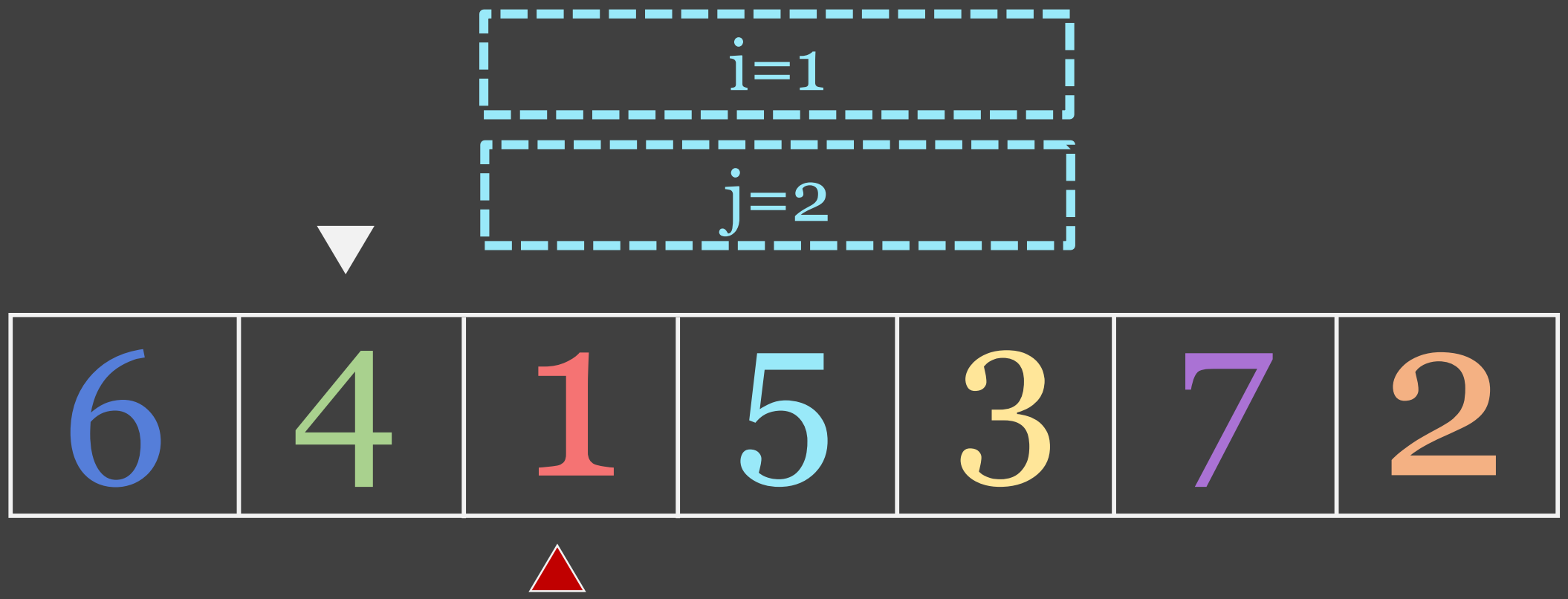
5 Шагаем на один элемент назад i--

6 Если i > 0, то к пункту **2**(используя оператор continue), иначе к пункту **7**

7 i = j++

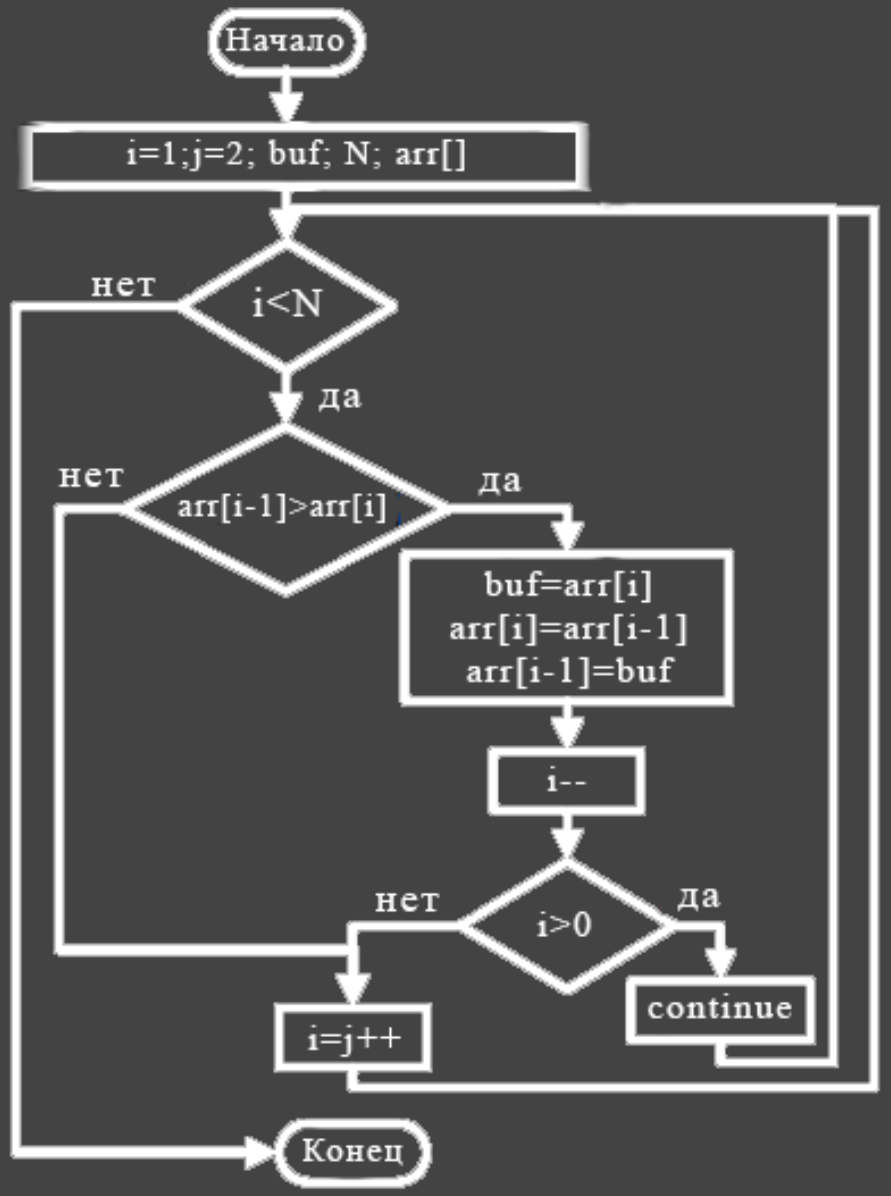
8 К пункту 2.

9 Конец алгоритма



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.

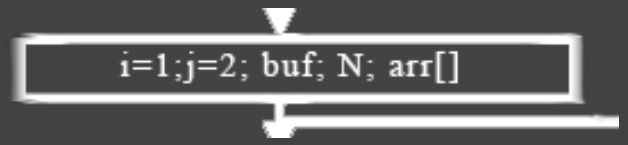
БЛОК-СХЕМА


Начало
i=1;j=2; buf; N; arr[]
нет
i<N
да
нет
arr[i-1]>arr[i]
да
buf=arr[i]
arr[i]=arr[i-1]
arr[i-1]=buf
i--
нет
i>0
да
continue
i=j++
Конец

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

1 Первый элемент («пуск») мы используем для обозначения начала работы алгоритма.

2 Во втором элементе, называемом «процесс», мы задаем переменные, которые в дальнейшем будем применять для работы алгоритма.




```
Int N, min, buf, i, j, arr[] = { 6, 4, 1, 5, 3, 7, 2 };
```

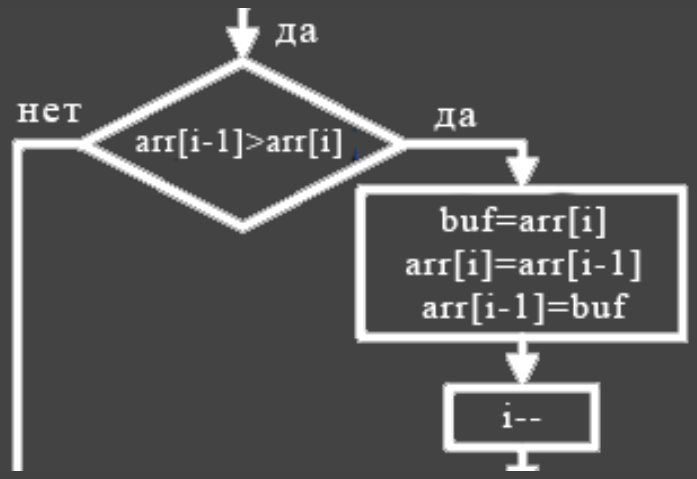
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

3 Третьим по очереди мы применяем элемент «цикл» для проведения нескольких одинаковых действий и проверки условия: от первого элемента массива до последнего проводится проверка внутри цикла. От «цикла» отходит две ветви: по первой мы идем в случае, если результат проверки условия положительный (заход в цикл и «проход» по всему массиву и процесс сортировки), по второй – если отрицательный (элементы массива закончились).


```
While (i<N) {
…
}
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

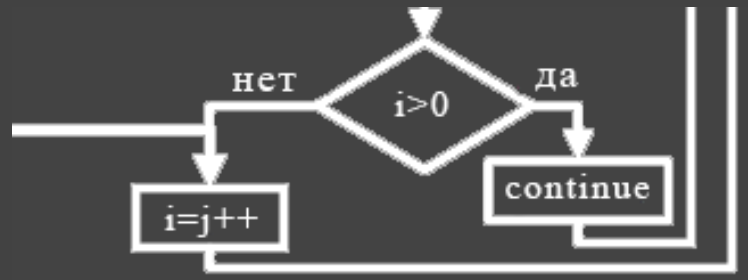
4 Следующим мы используем такой элемент блок-схемы как «цикл» для того, чтобы произвести несколько раз одно и то же действие с выполнением некоторого условия.




```
if (a[i-1]>a[i]){
        B=a[i];
        a[i]=a[i-1];
        a[i-1]=B;
        i--;
        ...
    }
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

5 Далее мы еще раз используем элемент «условие» для проверки условия того, не является ли элемент первым с конца. От «условия» отходит две ветви: по первой мы идем в случае, если результат проверки условия положительный, по второй – если отрицательный, при котором мы рассматриваем следующий элемент массива, приравнивая новый номер к заранее сохраненному элементу .




```
if (i>0) continue;
      }
      i=j++;
    }
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

6 Последний элемент («пуск») мы применяем для обозначения конца работы алгоритма.

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ С

```
int main() {
  int i=1, j=2, buf, N=7;
  int arr[]={6, 4, 1, 5, 3, 7, 2};

  while (i<N) {
       if (arr[i-1]>arr[i]) {
           buf=arr[i];
           arr[i]=arr[i-1];
           arr[i-1]=buf;
           i--;
           if (i>0) continue;

       }
          i=j++;
    }
   return 0;
}
```

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ C++

```
#include <iostream>
using namespace std;

int main()
 {
   int N, j=2, i=1, buf;
   arr = new int[6, 4, 1, 5, 3, 7, 2];
   N=7;

   while (i<N) {
        if (arr[i-1]>arr[i]) {
            buf=arr[i];
            arr[i]=arr[i-1];
            arr[i-1]=buf;
            i--;
            if (i>0) continue;

        }
           i=j++;
     }
    return 0;
}
```

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ C#

```
     static void Main(string[] args)
        {
 int[] arr = {6, 4, 1, 5, 3, 7, 2};

     int i, j, buf;
    while (i<arr.Length) {
        if (arr[i-1]>arr[i]) {
            buf=arr[i];
            arr[i]=arr[i-1];
            arr[i-1]=buf;
            i--;
            if (i>0) continue;

        }
           i=j++;
     }
   Console.ReadKey();
  }
}
```

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ JAVA

```
public static void sort(int[] arr) {
            while (i < l) {
               if (i > 0 && arr[i - 1] > arr[i]) {
                 [arr[i], arr[i - 1]] = [arr[i - 1], arr[i]];
                 i--;
                 }
              else {
                 i++;
                   }

          } return arr;
}
```

основы алгоритмизации и программирования

СОРТИРОВКА «БЫСТРАЯ»

ПЛАН ЛЕКЦИИ

Часть 1:

1. Понятие алгоритма
2. Идея алгоритма
3. Визуализация
4. Словесное представление алгоритма

Часть 2:

5. Блок-схема
6. Подробный разбор элементов блок-схемы
7. Программная реализация на языке программирования С

ПОНЯТИЕ АЛГОРИТМА

Быстрая сортировка (англ. *quick sort*, сортировка Хоара) — один из самых известных и широко используемых алгоритмов сортировки. Среднее время работы O(n log n), что является асимптотически оптимальным временем работы для алгоритма, основанного на сравнении. Хотя время работы алгоритма для массива из n элементов в худшем случае может составить Θ(n^2), на практике этот алгоритм является одним из самых быстрых. Для этого алгоритма применяется рекурсивный метод.

Рекурсией называется ситуация, когда программа вызывает сама себя непосредственно или косвенно (через другие функции).

ИДЕЯ АЛГОРИТМА

Выбираем из массива элемент, называемый опорным, и запоминаем его значение. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность.

Далее начинаем двигаться от начала массива по возрастающей, а потом от конца массива по убывающей. Цель: переместить в правую часть элементы больше опорного, а в левую – элементы меньше опорного. Если во время движения по возрастающей находится элемент со значением больше опорного, то мы выходим из цикла, прибавляем единицу к индексу элемента, на котором остановились, и переходим к циклу с движением по убывающей. В этом цикле мы остаемся до тех пор, пока не находится элемент со значением меньше опорного.

ИДЕЯ АЛГОРИТМА

Как только такой элемент найден, мы отнимаем единицу от его индекса, и меняем значение элемента со значением элемента, на котором мы остановились в предыдущем цикле. Делаем так до тех пор, пока индекс левого элемента (найденного в первом цикле) меньше либо равен индексу правого элемента (найденного во втором цикле). В итоге получаем два подмассива (от начала до индекса правого элемента и от индекса левого элемента до конца). С этими подмассивами мы рекурсивно проделываем все то же самое, что и с большим массивом до тех пор, пока все элементы окончательно не отсортируются.


6
4
1
5
3
7
2
BEGIN
END

СЛОВЕСНОЕ ПРЕДСТАВЛЕНИЕ

array – массив, piv – номер опорного элемента, b – индекс первого элемента массива, e – индекс последнего элемента массива

1. Номер опорного элемента равен (b+e)/2
2. Начало курсор. Если l <= r (где l = b, а r = e), то переходим к пункту 3, иначе к пункту 13
3. Если array[l] < piv, то переходим к пункту 4, иначе к пункту 6.
4. Прибавляем единицу к индексу левого элемента (l++).
5. Переходим к пункту 3.
6. Если array[r] > piv, то переходим к пункту 7, иначе к пункту 9.
7. Убавляем на единицу индекс правого элемента (r--)
8. Переходим к пункту 6.
9. Если l <= r, то переходим к пункту 10, иначе переходим к пункту 13.
10. Меняем значения элементов с индексами l и r местами (array[l] и array[r])

СЛОВЕСНОЕ ПРЕДСТАВЛЕНИЕ

array – массив, piv – номер опорного элемента, b – индекс первого элемента массива, e – индекс последнего элемента массива

11 Прибавляем единицу к индексу левого элемента и убавляем на единицу индекс правого элемента.

12 Переходим к пункту 2.

13 Если b < r (индекс первого элемента массива меньше индекса правого элемента массива), то переходим к пункту 14, иначе к пункту 15.

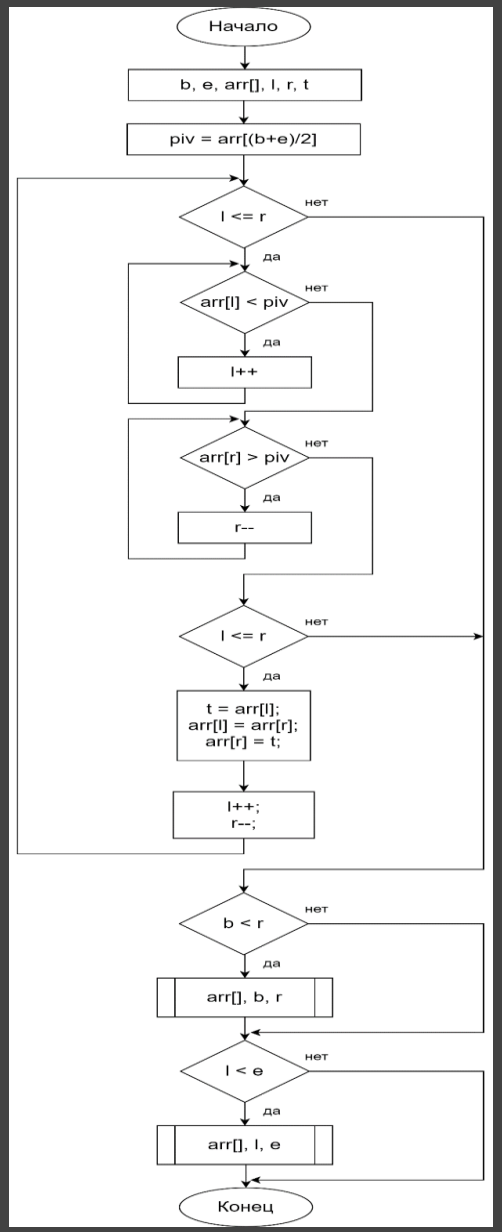
14 Вызов курсор: (array[], b, r).

15 Если l < e (индекс левого элемента меньше индекса последнего элемента массива), то переходим к пункту 16, иначе к пункту 17.

16 Вызов курсор: (arr[], l, e).

17 Конец алгоритма

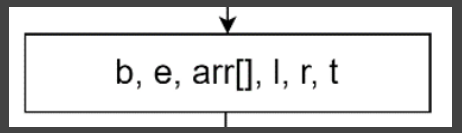
БЛОК-СХЕМА


Начало
b, e, arr[], l, r, t
piv = arr[(b+e)/2]
l <= r
нет
да
arr[l] < piv
нет
да
l++
arr[r] > piv
нет
да
r--
l <= r
нет
да
t = arr[l];
arr[l] = arr[r];
arr[r] = t;
l++;
r--;
b < r
нет
да
arr[], b, r
l < e
нет
да
arr[], l, e
Конец

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

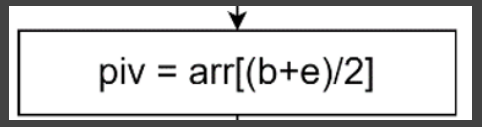
1 Первый элемент («пуск») мы используем для обозначения начала работы алгоритма.

2 Во втором элементе, называемом «процесс», мы задаем переменные, которые в дальнейшем будем применять для работы алгоритма.




```
int b, e, l, r, t, arr[] = { 5, 7, 8, 4, 9 };
```

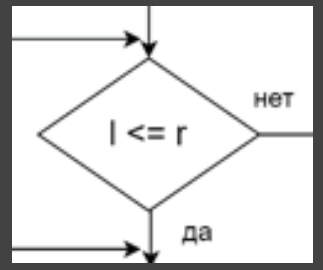
3 Третий элемент («процесс») служит нам для обозначения вычислительных действий, а именно для нахождения номера опорного элемента (piv).




```
int piv = arr[(b + e) / 2];
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

4 Следующий элемент («решение») используется нами для проверки условия: номер левого элемента меньше либо равен номеру правого элемента массива. От «решения» отходит две ветви: по первой мы идем в случае, если результат проверки условия положительный (элемент, к которому отходит эта ветвь будет рассмотрен нами в пункте **5**), по второй – если отрицательный (элемент, к которому отходит эта ветвь будет рассмотрен нами в пункте **12**).

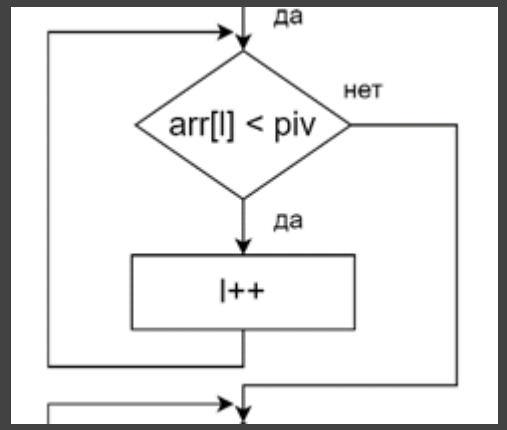



```
while (l <= r) //заголовок внешнего цикла
    {

    }
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

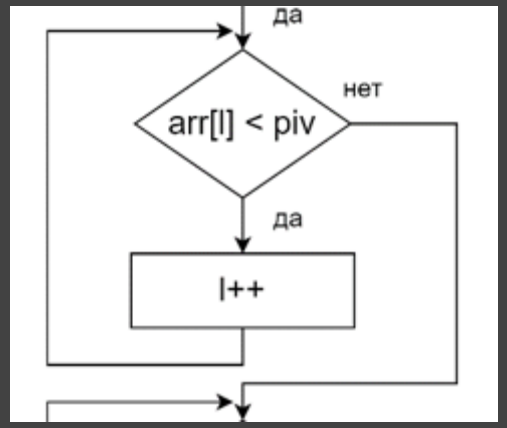
5 Пятым по очереди мы снова применяем элемент «решение» для проверки условия: значение элемента с номером левого элемента массива меньше номера опорного элемента. От «решения» отходит две ветви: по первой мы идем в случае, если результат проверки условия положительный (элемент, к которому отходит эта ветвь будет рассмотрен нами в пункте **6**), по второй – если отрицательный (элемент, к которому отходит эта ветвь будет рассмотрен нами в пункте **7**).




```
while (arr[l] < piv) // первый внутренний цикл
{
    l++; // тело первого внутреннего цикла
}
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

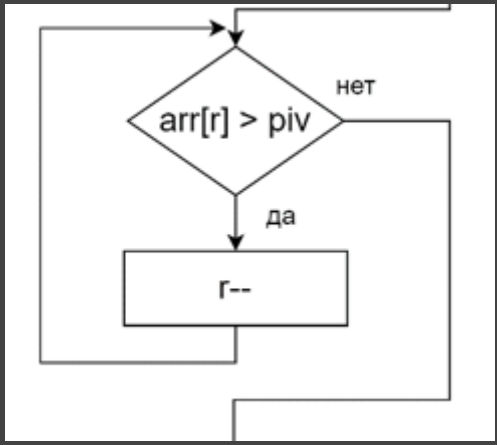
6 Следующим мы используем такой элемент блок-схемы как «процесс» для того, чтобы увеличить номер левого элемента на единицу. На «процессе» завершается один из внутренних циклов нашего алгоритма, поэтому от него отходит зацикливающая ветвь, которая подводит к предыдущему элементу «решение» для очередной проверки условия.




```
i++;
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

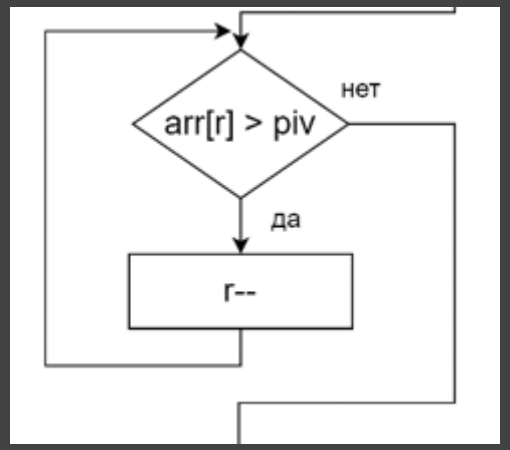
7 Далее мы еще раз используем элемент «решение» для проверки условия: значение элемента с номером правого элемента массива больше номера опорного элемента. От «решения» отходит две ветви: по первой мы идем в случае, если результат проверки условия положительный (элемент, к которому отходит эта ветвь будет рассмотрен нами в пункте **8**), по второй – если отрицательный (элемент, к которому отходит эта ветвь будет рассмотрен нами в пункте **9**).




```
while (arr[r] > piv) // второй внутренний цикл
{
    r--; // тело второго внутреннего цикла
}
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

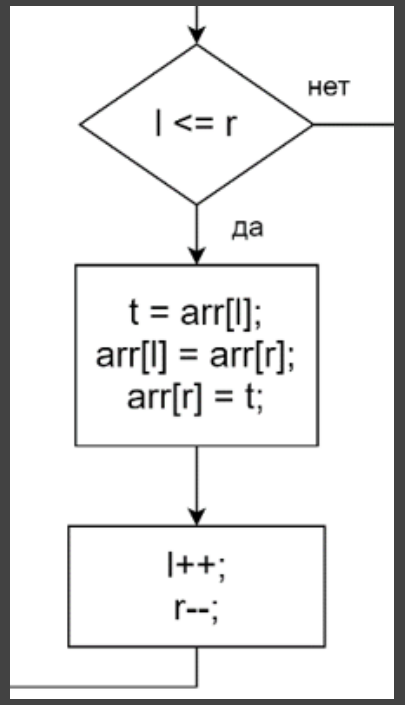
8 Следующим мы вновь применяем элемент - «процесс» для того, чтобы уменьшить номер правого элемента на единицу. На «процессе» завершается второй внутренний цикл алгоритма, поэтому от него отходит зацикливающая ветвь, которая подводит к предыдущему элементу «решение» для очередной проверки условия.




```
r--;
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

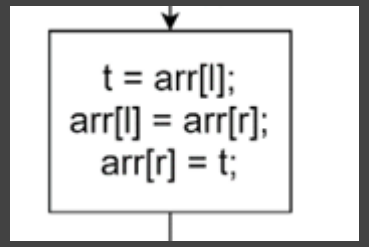
9 Девятым по очереди идет элемент «решение», необходимый нам для проверки следующего условия: номер левого элемента меньше либо равен номеру правого элемента массива. От «решения» отходит две ветви: по первой мы идем в случае, если результат проверки условия положительный (элемент, к которому отходит эта ветвь будет рассмотрен нами в пункте **10**), по второй – если отрицательный (элемент, к которому отходит эта ветвь будет рассмотрен нами в пункте **12**).




```
if (l <= r) // условие
{
    t = arr[l];   // выполняем перестановку
    arr[l] = arr[r];
    arr[r] = t;
    l++; // изменяем значения индексов элементов
    r--;
}// конец внешнего цикла
```

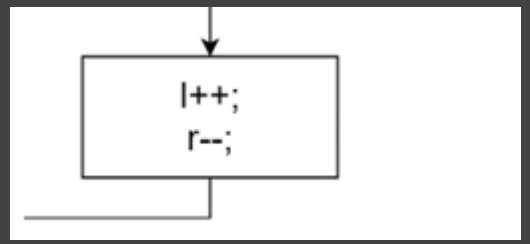
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

10 Далее мы используем «процесс» для осуществления перестановки элементов массива.




```
t = arr[l];
arr[l] = arr[r];
arr[r] = t;
```

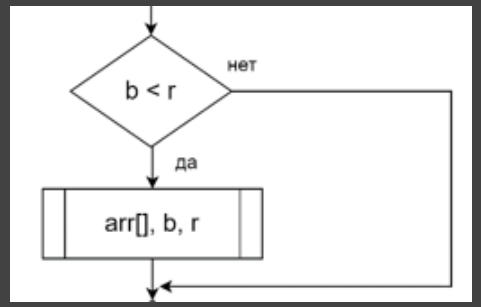
11 Следующим на очереди мы снова применяем элемент «процесс» для того, чтобы увеличить номер левого элемента на единицу и уменьшить номер правого элемента на единицу. На «процессе» завершается внешний цикл нашего алгоритма, поэтому от него отходит зацикливающая ветвь, которая подводит к элементу «решение» (который мы обсуждали в **4** пункте) для очередной проверки условия.




```
l++;
r--;
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

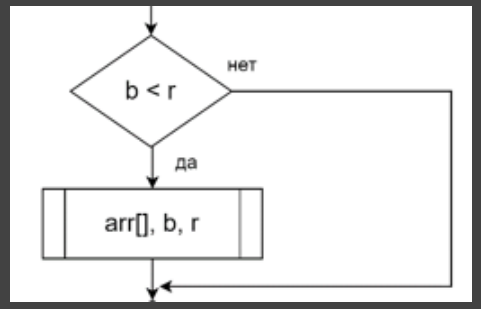
12 В представленном далее элементе – «решение» мы проверяем следующее условие: индекс первого элемента массива меньше индекса правого элемента массива. От «решения» отходит две ветви: по первой мы идем в случае, если результат проверки условия положительный (элемент, к которому отходит эта ветвь будет рассмотрен нами в пункте 13), по второй – если отрицательный (элемент, к которому отходит эта ветвь будет рассмотрен нами в пункте 14).




```
if (b < r) //условие
{
      qsort(arr, b, r);// вызываем функцию Быстрой сортировки
}
```

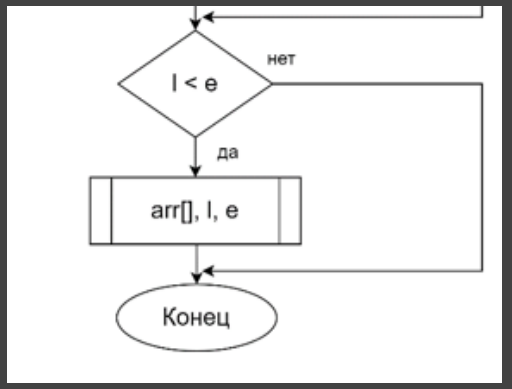
РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

13 Далее мы обращаемся к такому элементу как «предопределенный процесс» для вызова курсор (функция вызывает сама себя).




```
qsort(arr, b, r);
```

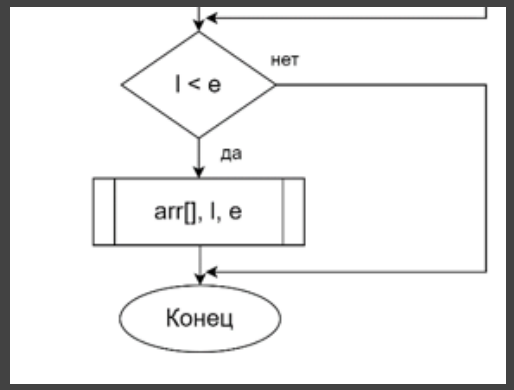
14 Следующим мы еще раз применяем элемент «решение» для проверки условия: индекс левого элемента массива меньше индекса последнего элемента массива. От «решения» отходит две ветви: по первой мы идем в случае, если результат проверки условия положительный (элемент, к которому отходит эта ветвь будет рассмотрен нами в пункте 15), по второй – если отрицательный (элемент, к которому отходит эта ветвь будет рассмотрен нами в пункте 16).




```
if (l<e) //условие
{
    qsort(arr, l, e);
}
```

РАЗБОР ЭЛЕМЕНТОВ БЛОК-СХЕМЫ

15 Следующим мы снова используем элемент «предопределенный процесс» для вызова курсор (функция вызывает сама себя), используя уже другие переменные




```
qsort(arr, l, e);
```

16 Последний элемент («пуск») мы применяем для обозначения конца работы алгоритма.

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

```
#include <iostream>
using namespace std;

void qsort(int* arr, int b, int e) {
    int piv = arr[(b + e) / 2];
    int l = b;
    int r = e;
    while (l <= r)
    {
        while (arr[l] < piv) l++;
        while (arr[r] > piv) r--;
        if (l <= r) // условие
        {
            int t = arr[l];
            arr[l] = arr[r];
            arr[r] = t;
            l++;
            r--;
        }
    }
}
```

```
//продолжение кода
int main()
{
    setlocale(LC_ALL, "Russian");
    int b, e, l, r, t, arr[] = { 5, 7,
8, 4, 9 };
    if (b < r) {
        qsort(arr, b, r);
    }
    if (l<e)
    {
        qsort(arr, l, e);
    }

}
```

основы алгоритмизации и программирования

«ВЫЧИСЛИТЕЛЬНАЯ СЛОЖНОСТЬ АЛГОРИТМОВ»

ПЛАН ЛЕКЦИИ

Часть 1:

1. Понятие алгоритма
2. Определение алгоритма
3. Понятие эффективности алгоритма
   - 3.1. sys-tem efficiency
   - 3.2. space efficiency
   - 3.3. computational efficiency

Часть 2:

4. Определение вычислительной сложности алгоритма
5. Примеры некоторых видов сложности по времени
   - 5.1. Линейная сложность
   - 5.2. Логарифмическая сложность
   - 5.3. Квадратичная сложность
   - 5.4. Экспоненциальная сложность
   - 5.5. Не зависящая от размера
6. Вычислительная неточность
7. Визуализация

ПОНЯТИЕ АЛГОРИТМА

Первоначально понятие алгоритма отождествлялось с понятием метода вычислений. С точки зрения современной практики алгоритм – программа, а критерием алгоритмичности вычислительного процесса является возможность его запрограммировать.
Именно благодаря этой реальности алгоритма, а также благодаря тому, что подход инженера к математическим методам всегда был конструктивным, понятие алгоритма в технике за короткий срок стал необычайно популярным.

Понятие алгоритма, подобно понятиям множества и натурального числа, относится к числу столь фундаментальным понятий, что оно не может быть выражено через другие понятия.

ОПРЕДЕЛЕНИЕ АЛГОРИТМА

Определение 1.1. Алгоритм (алгорифм) – точное предписание, которое задает вычислительный процесс, начинающийся с произвольного исходного данного и направленный на получение полностью определенного этим исходным данным результата .

Определение 1.2. Алгоритм есть точное предписание, которое задает вычислительный процесс нахождения значений вычислимой функции по заданным значениям ее аргументов.

Определение 1.3. Алгоритм есть предписание, однозначно определяющее ход некоторых конструктивных процессов.

ЭФФЕКТИВНОСТЬ АЛГОРИТМА

sys-tem efficiency

Алгоритм, в конечном счете, выполняется в машинной системе со специфическим набором команд и периферийными устройствами. Для отдельной системы какой-либо алгоритм может быть разработан для полного использования преимуществ данного компьютера и поэтому достигает высокой степени эффективности. Критерий, называемый **системной эффективностью (sys-tem efficiency)**, сравнивает скорость выполнения двух или более алгоритмов, которые разработаны для выполнения одной и той же задачи. Выполняя эти алгоритмы на одном компьютере с одними и теми же наборами данных, мы можем определить относительное время, используя внутренние системные часы. Оценка времени становится мерой системной эффективности для каждого из алгоритмов.

ЭФФЕКТИВНОСТЬ АЛГОРИТМА

space efficiency

При работе с некоторыми алгоритмами могут стать проблемой ограничения памяти. Процесс может потребовать большого временного хранения, ограничивающего размер первоначального набора данных, или вызвать требующую времени дисковую подкачку. **Эффективность пространства (space efficiency)** — это мера относительного количества внутренней памяти, используемой каким-либо алгоритмом. Она может указать, какого типа компьютер способен выполнять этот алгоритм и полную системную эффективность алгоритма. Вследствие увеличения объема памяти в новых системах, анализ пространственной эффективности становится менее важным.

ЭФФЕКТИВНОСТЬ АЛГОРИТМА

computational efficiency

Третий критерий эффективности рассматривает внутреннюю структуру алгоритма, анализируя его разработку, включая количество тестов сравнения итераций и операторов присваивания, используемых алгоритмом. Эти типы измерений являются независимыми от какой-либо отдельной машинной системы. Критерий измеряет вычислительную сложность алгоритма относительно n, количества элементов данных в коллекции. Мы называем эти критерии **вычислительной эффективностью (computational efficiency)** алгоритма и разрабатываем нотацию **Big-O** для построения измерений, являющихся функциями n.

ВЫЧИСЛИТЕЛЬНАЯ СЛОЖНОСТЬ АЛГОРИТМА

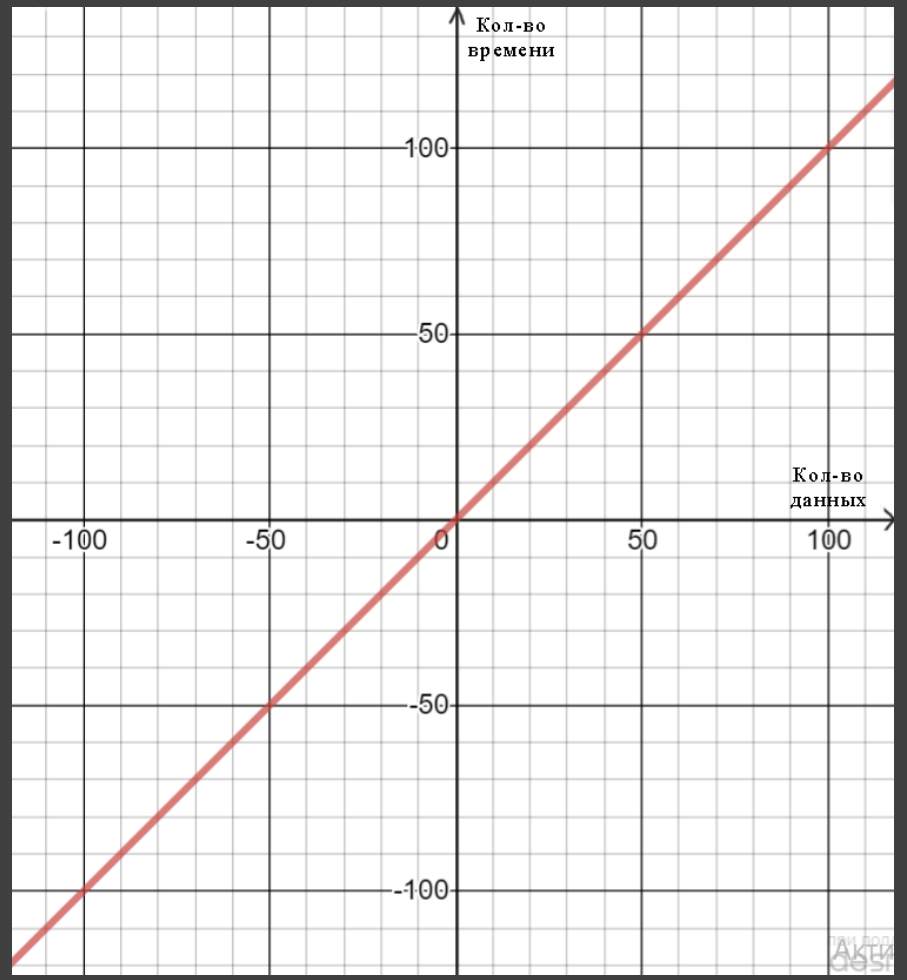
Сложность алгоритмов обычно оценивают по времени выполнения или по используемой памяти. Алгоритм имеет сложность **O(f(n))**, если при увеличении размера входных данных **n**, время выполнения алгоритма возрастает с той же скоростью, что и функция **f(n)**. Оценивая порядок сложности алгоритма, необходимо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

ЛИНЕЙНАЯ СЛОЖНОСТЬ

O(n) — линейная сложность

“Линейная” означает, что при увеличении объема данных время на их передачу вырастет примерно пропорционально. Если данных станет в 2 раза больше, и времени на их передачу понадобится в 2 раза больше. Если данных станет больше в 10 раз, и время передачи увеличится в 10 раз.

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

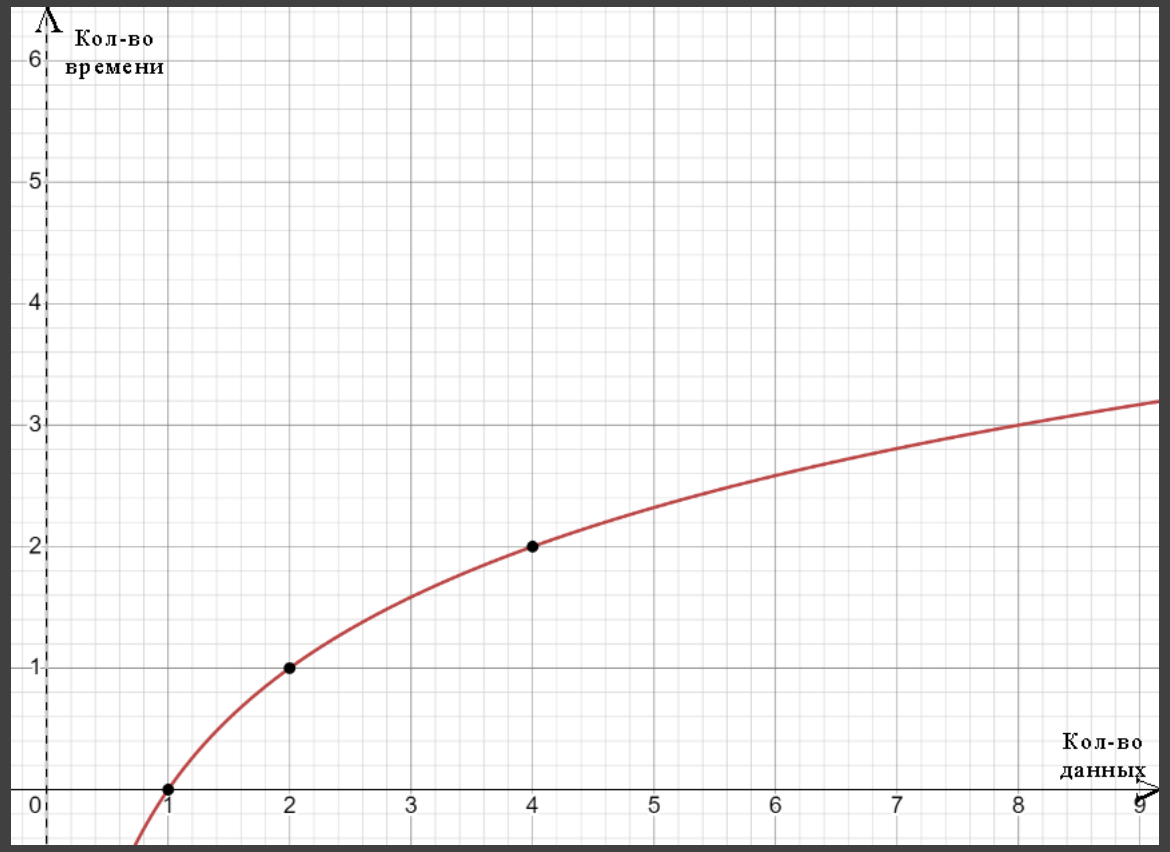


ЛОГАРИФМИЧЕСКАЯ СЛОЖНОСТЬ

O(log n) — логарифмическая сложность

“Логарифмическая” означает, что при увеличении объема данных время на их обработку вырастет логарифмически. Если данных станет в 4 раза больше, времени на их передачу понадобится в 2 раза больше. Если данных станет больше в 1024 раза, время передачи увеличится в 10 раз.

Пример: вводится число a, являющееся степенью двойки ($2^x = a$). Нужно найти эту степень, путём деления этого числа на два и подсчёта количества этих делений

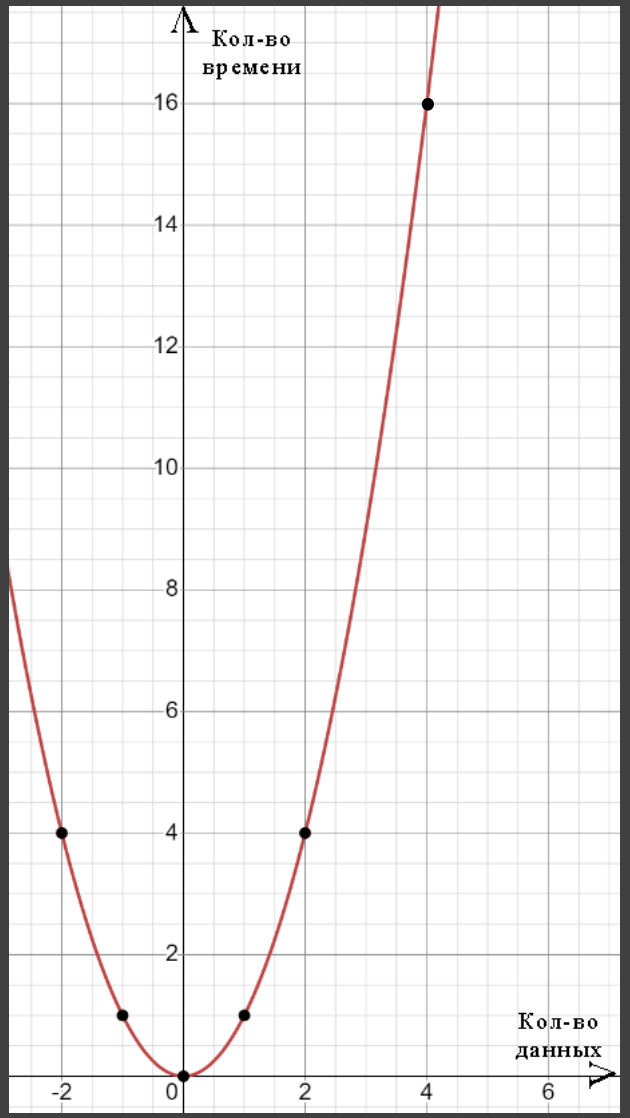


КВАДРАТИЧНАЯ СЛОЖНОСТЬ

O(n²) — квадратичная сложность

“Квадратичная” означает, что при увеличении объема данных время на их обработку вырастет в квадрате. Если данных станет в 2 раза больше, времени на их передачу понадобится в 4 раза больше. Если данных станет больше в 8 раз, время передачи увеличится в 64 раза.

Такую сложность имеет, например, алгоритм сортировки вставками. В канонической реализации он представляет из себя два вложенных цикла: один, чтобы проходить по всему массиву, а второй, чтобы находить место очередному элементу в уже отсортированной части. Таким образом, количество операций будет зависеть от размера массива как n * n, т.е. n².

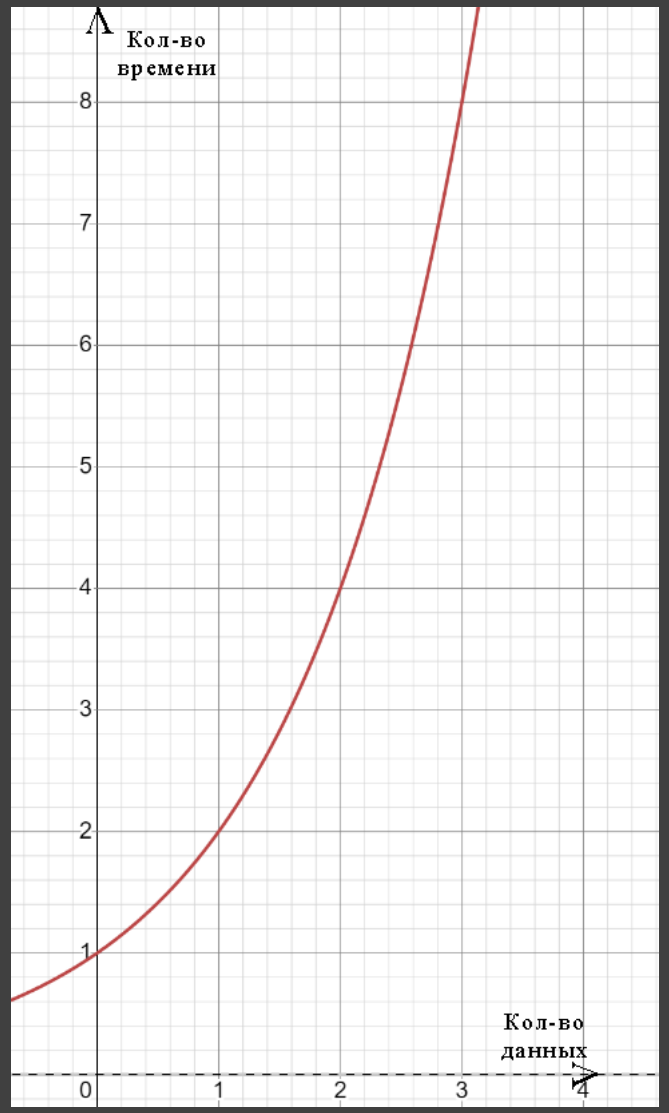


ЭКСПОНЕНЦИАЛЬНАЯ СЛОЖНОСТЬ

$O(2^n)$ — экспоненциальная сложность

“Экспоненциальная” означает, что если размерность задачи возрастает линейно, время ее решения возрастает экспоненциально.
Если данных станет в 2 раза больше, времени на их передачу понадобится в 4 раза больше. Если данных станет больше в 5 раза, время передачи увеличится в 32 раз.

Пример: вывести на экран все двоичные числа длин N

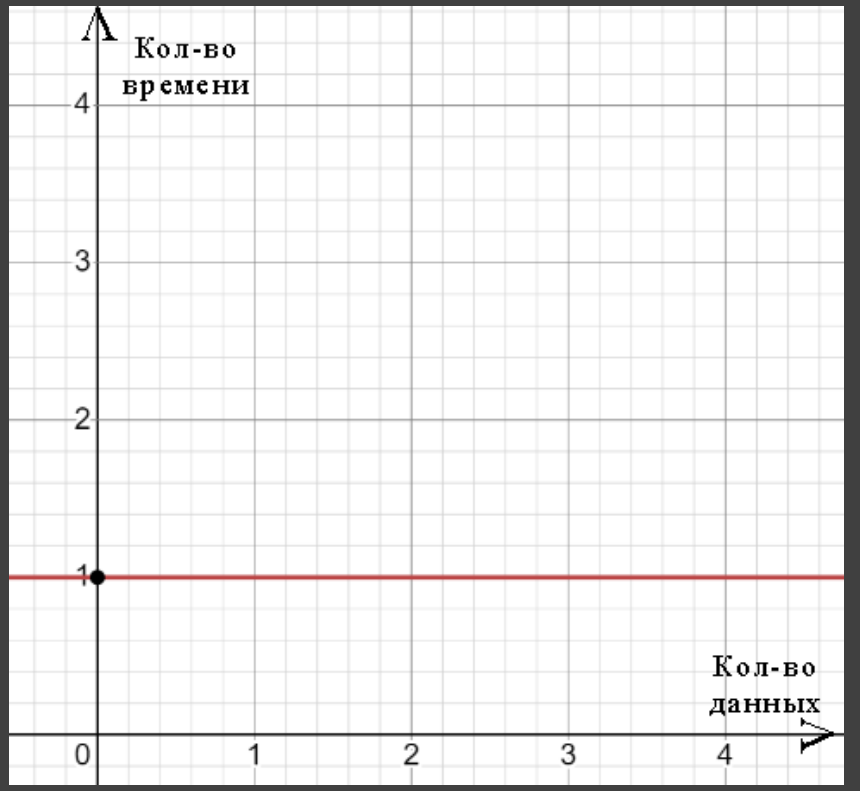


НЕ ЗАВИСЯЩАЯ ОТ РАЗМЕРА

O(1) - не зависящая от размера данных сложность

Также случается, что время работы алгоритма вообще не зависит от размера входных данных.

Например, для определения значения третьего элемента массива не нужно ни запоминать элементы, ни проходить по ним сколько-то раз. Всегда нужно просто дождаться в потоке входных данных третий элемент и это будет результатом, на вычисление которого для любого количества данных нужно одно и то же время.



ВЫЧИСЛИТЕЛЬНАЯ НЕТОЧНОСТЬ

Аналогично всему вышеперечисленному проводят оценку и по памяти, когда это важно. Однако алгоритмы могут использовать значительно больше памяти при увеличении размера входных данных, чем другие, но зато работать быстрее. И наоборот. Это помогает выбирать оптимальные пути решения задач исходя из текущих условий и требований.

ВИЗУАЛИЗАЦИЯ

Время выполнения алгоритма с определённой сложностью в зависимости от размера входных данных при скорости 10^6 операций в секунду:

	10	20	30	40	50	60
log(n)	$3{,}322*10^{-6}$ **сек**	$4{,}322*10^{-6}$ **сек**	$4{,}907*10^{-6}$ **сек**	$5{,}322*10^{-6}$ **сек**	$5{,}644*10^{-6}$ **сек**	$5{,}907*10^{-6}$ **сек**
n	0,00001 **сек**	0,00002 **сек**	0,00003 **сек**	0,00004 **сек**	0,00005 **сек**	0,00006 **сек**
n^2	0,0001 **сек**	0,0004 **сек**	0,0009 **сек**	0,0016 **сек**	0,0025 **сек**	0,0036 **сек**
2^n	0,001024 **сек**	1,05 **сек**	17,89 **мин**	12,72 **дн**	35,70 **лет**	365,59 **веков**