16.12.2024

Раздельная компиляция. Работа с простыми числами. Вычислительная геометрия

Филиппов Михаил Витальевич

m.filippov@g.nsu.ru

89232283872

Императивное программирование, 2024-2025

Новосибирский государственный университет

*НАСТОЯЩАЯ НАУКА

Давайте познакомимся

Филиппов Михаил Витальевич

- Окончил магистратуру ФФ НГУ
- Окончил аспирантуру ИТ СО РАН
- Являюсь м.н.с. ИТ СО РАН
- 7+ лет опыт в программировании C/C++

План лекции

Раздельная компиляция	Работа с простыми числами	Вычислительная геометрия
50 минут	15 минут	25 минут

План лекции

Раздельная компиляция

50 минут

Работа с простыми числами

15 минут

Вычислительная геометрия

25 минут

Общие слова

Компиляция программ, состоящих из двух и более файлов исходного кода

Unix
Компиляция a.out: cc file1.c file2.c -> file1.o и file2.o

Linux
Компиляция a.out: gcc file1.c file2.c -> file1.o и file2.o

Компиляторы командной строки DOS
Аналогично, но .o -> .obj

Хорошим тоном в программировании считается размещение прототипов функций и определенных констант в заголовочном файле.

Компиляторы интегрированных сред разработки (integrated development environment — IDE) для операционных систем Windows и Macintosh являются сориентированными на проекты. Проект описывает ресурсы, используемые конкретной программой.

Использование заголовочных файлов

- Если поместить функцию main() в один файл, а определения собственных функций — в другой, то в первом файле по-прежнему будут нужны прототипы функций. Вместо того чтобы набирать их каждый раз, когда применяется файл с функциями, прототипы функций можно сохранить в заголовочном файле.
- Более разумное решение предполагает размещение директив #define в заголовочном файле с последующим использованием директивы #include в каждом файле исходного кода.
- При разделении на h и cpp файлы повышается читаемость кода и эффективность компиляции.

Демо – несколько файлов: hotel.h

```
/* hotel.h -- константы и объявления для программы hotel.c */
#define QUIT 5
#define HOTEL1 180.00
#define HOTEL2 225.00
#define HOTEL3 255.00
#define HOTEL4 355.00
#define DISCOUNT 0.95
#define STARS "**********************************"
// отображает список возможных вариантов
int menu(void);
// возвращает количество суток, на которое бронируется номер
int getnights(void);
// вычисляет стоимость в зависимости от расценок и количества
// забронированных суток и отображает результат
void showprice(double rate, int nights);
```

Демо – несколько файлов: hotel.c 1/2

```c
/* hotel.c – функции управления отелем */
#include <stdio.h>
#include "hotel.h"
int menu(void)
{
    int code, status;
    printf("\n%s%s\n", STARS, STARS);
    printf("Введите число, соответствующее выбранному отелю:\n");
    printf("1) Fairfield Arms 2) Hotel Olympic\n");
    printf("3) Chertworthy Plaza 4) The Stockton\n");
    printf("5) выход \n");
    printf("%s%s\n", STARS, STARS);
    while ((status = scanf("%d", &code)) != 1 || (code < 1 || code > 5))
    {
        if (status != 1)
            scanf("%*s"); // отбрасывание нецелочисленного ввода
        printf("Введите целое число от 1 до 5.\n");
    }
    return code;
}
```

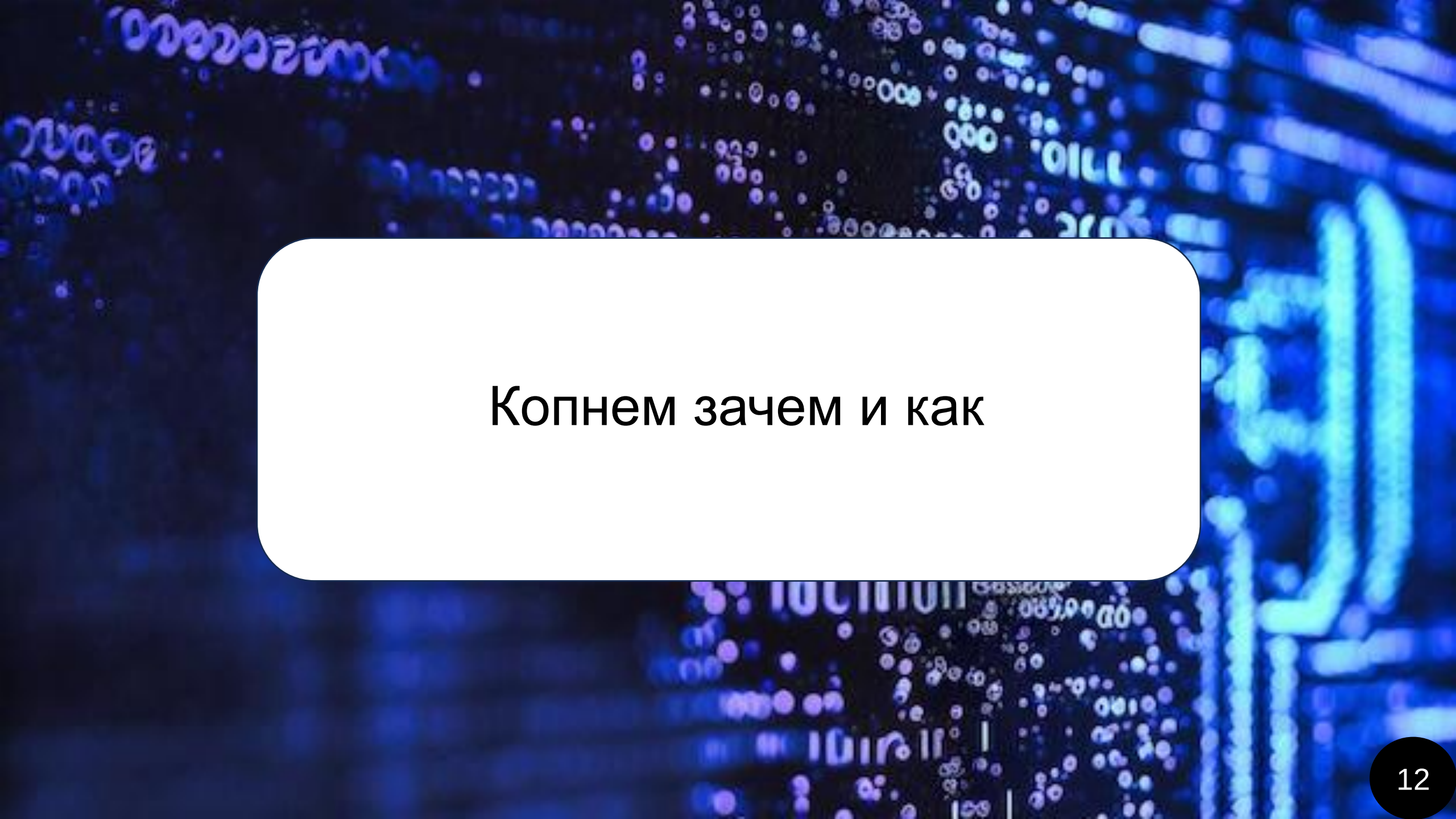
Демо – несколько файлов: hotel.c 2/2

```
int getnights(void)
{
    int nights;
    printf("На сколько суток вы бронируете номер? ");
    while (scanf("%d", &nights) != 1)
    {
        scanf("%*s"); // исключение нецелочисленного ввода
        printf("Введите целое число, такое как 2.\n");
    }
    return nights;
}
void showprice(double rate, int nights)
{
    int n;
    double total = 0.0;
    double factor = 1.0;
    for (n = 1; n <= nights; n++, factor *= DISCOUNT)
        total += rate * factor;
    printf("Общая стоимость составляет $%0.2f.\n", total);
}
```

Демо – несколько файлов: hotel.c 2/2

```c
#include <stdio.h>
#include "hotel.h" /* определяет константы, объявляет функции */
int main(void)
{
    int nights, code; double hotel_rate;
    while ((code = menu()) != QUIT)
    {
        switch (code)
        {
        case 1:
            hotel_rate = HOTEL1;
            break;
        …
        case 4:
            hotel_rate == HOTEL4;
            break;
        default:
            hotel_rate = 0.0;
            printf("Ошибка!\n");
            break;
        }
        nights = getnights();
        showprice(hotel_rate, nights);
    }
    return 0;
```

```
**********************************************************************
Введите число, соответствующее выбранному отелю:
l) Fairfield Arms 2) Hotel Olympic
3) Chertworthy Plaza 4) The Stockton
5) выход
**********************************************************************
1
На сколько суток вы бронируете номер? 5
Общая стоимость составляет $814.39.

**********************************************************************
Введите число, соответствующее выбранному отелю:
l) Fairfield Arms 2) Hotel Olympic
3) Chertworthy Plaza 4) The Stockton
5) выход
**********************************************************************
4
На сколько суток вы бронируете номер? 5
Общая стоимость составляет $1153.72.

**********************************************************************
Введите число, соответствующее выбранному отелю:
l) Fairfield Arms 2) Hotel Olympic
3) Chertworthy Plaza 4) The Stockton
5) выход
**********************************************************************
5
Благодарим за использование и желаем успехов.
```


Копнем зачем и как

Процесс компиляции

Компиляция файлов, написанных на языке C, обычно длится лишь несколько секунд, но за это время исходный код успевает пройти процесс обработки с участием четырех отдельных компонентов, каждый из которых имеет определенное назначение:

- препроцессор;
- компилятор;
- ассемблер;
- компоновщик.

Каждый из них принимает определенный ввод от предыдущего компонента и генерирует определенный вывод для следующего. Этот процесс продолжается, пока последний компонент не сгенерирует итоговый продукт.

Полный список доступных компиляторов можно найти в «Википедии»: https://en.wikipedia.org/wiki/List_of_compilers#C_compilers

Процесс компиляции

Платформа — сочетание операционной системы и оборудования (или архитектуры), самая важная часть которого — набор инструкций центрального процессора. Операционная система играет роль программного компонента платформы, а аппаратный компонент определяется архитектурой. Например, мы можем иметь дело с ОС Ubuntu на ARM-плате или с Microsoft Windows на компьютере с 64-битным процессором AMD.

Кросс-платформенное программное обеспечение может работать на разных платформах. Но при этом необходимо понимать, что кросс-платформенность отличается от переносимости. Кросс-платформенное ПО обычно предоставляет разные двоичные (итоговые объектные) файлы и установщики для каждой среды, тогда как переносимые программы везде используют одни и те же исполняемые и установочные файлы.

Некоторые компиляторы для C, такие как gcc и clang, являются кросс-платформенными — они умеют генерировать код для разных платформ.

Сборка проекта на языке C

Заголовочные и исходные файлы

Любой проект на C содержит исходный код (или кодовую базу) и другие документы, которые описывают разрабатываемое приложение и используемые стандарты. Код C обычно хранится в файлах двух типов:

- в заголовочных файлах, как правило, имеющих расширение .h;
- в исходных файлах с расширением.c.

Для краткости заголовочные файлы называются заголовками, а исходные файлы — исходниками.

Сборка проекта на языке C

- Заголовочный файл обычно содержит перечисления, макросы и определения типов, а также объявления функций, глобальных переменных и структур. В языке C объявление и определение некоторых элементов программирования, таких как функции, переменные и структуры, могут находиться в разных файлах.
- Объявления принято хранить в заголовочных файлах, а соответствующие определения — в исходных. Это особенно относится к функциям.
- Объявления функций настоятельно рекомендуется размещать в заголовках, а их определения — в соответствующих исходниках. И хотя это не является обязательным требованием, такой подход к проектированию позволит вам хранить определения функций вне заголовочных файлов.
- Определения структур и объявления можно хранить и в разных файлах, но это делается в особых случаях.
- К заголовочным файлам можно подключать только другие заголовки, но не исходники. К исходным файлам можно подключать только заголовки. Подключение одних заголовочных файлов к другим считается дурным тоном. Если вы так делаете, то это обычно говорит о серьезной проблеме в архитектуре вашего проекта.

Сборка проекта на языке С

объявление функции

```
double average(int *, int);
```

определение функции

```
double average(int *array, int length)
{
    if (length <= 0)
        return 0;
    double sum = 0.0;
    for (int i = 0; i < length; i++)
        sum += array[i];
    return sum / length;
}
```

Правило 1: компилируются только исходные файлы. Заголовки не должны содержать ничего, кроме объявлений.
Правило 2: каждый исходный файл компилируется по отдельности.

!!!!!!Объявление функции хранится в заголовочном файле, а определение (или тело) — в исходном. Нарушать данное правило можно лишь в редких случаях. Кроме того, чтобы иметь доступ к объявлению, исходник должен подключить заголовочный файл.

Пример 2

main.c

```c
#include <stdio.h>
#include "average.h"
int main(int argc, char **argv)
{
    // объявление массива
    int array[5];
    // заполнение массива
    array[0] = 10;
    array[1] = 3;
    array[2] = 5;
    array[3] = -8;
    array[4] = 9;
    // вычисление среднего значения с
помощью функции avg
    double average = avg(array, 5,
NORMAL);
    printf("The average: %f\n",
average);
    average = avg(array, 5, SQUARED);
    printf("The squared average: %f\n",
average);
    return 0;
}
```

average.h

```c
#ifndef AVERAGE_H
#define AVERAGE_H
typedef enum
{
    NONE,
    NORMAL,
    SQUARED
} average_type_t;
// объявление функции
double avg(int *, int, average_type_t);
#endif
```

average.c

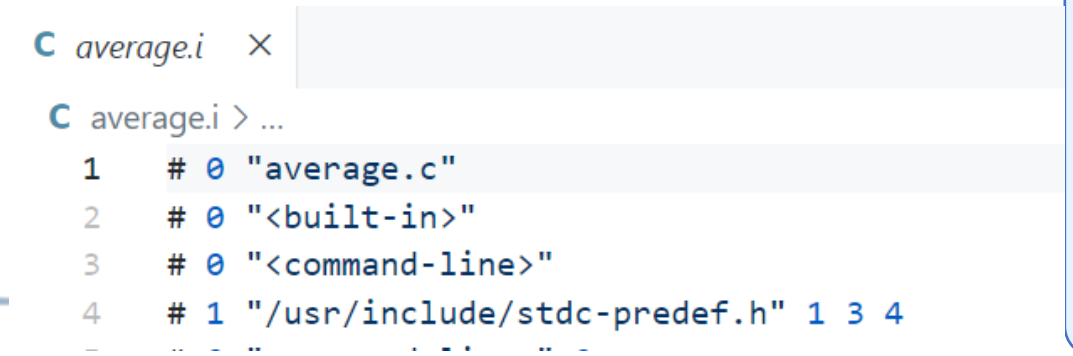
```c
#include "average.h"
double avg(int *array, int length,
average_type_t type)
{
    if (length <= 0 || type == NONE)
        return 0;
    double sum = 0.0;
    for (int i = 0; i < length; i++)
    {
        if (type == NORMAL)
            sum += array[i];
        else if (type == SQUARED)
            sum += array[i] * array[i];
    }
    return sum / length;
}
```

Этап 1: предобработка

Это первый этап компиляции. К исходному файлу подключается ряд заголовков. Но перед компиляцией препроцессор собирает их содержимое в единый блок кода. Иными словами, после предобработки мы получаем один фрагмент кода на языке C, сформированный путем копирования заголовочных файлов в исходные.

На данном этапе должны быть выполнены и другие директивы препроцессора. Код, прошедший такую обработку, называется единицей трансляции (или единицей компиляции). **Единица трансляции** — это отдельный логический блок кода на C, сгенерированный препроцессором и готовый к компиляции.

```
$ gcc -E average.c > average.i
```



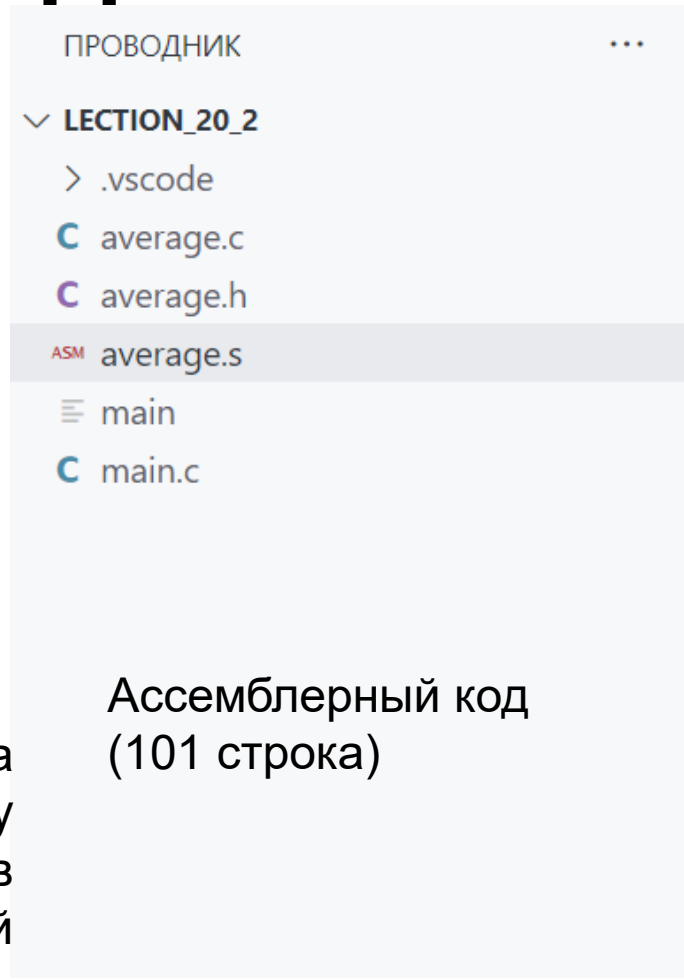

```
lection_20_2> gcc -E average.c
# 0 "average.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "average.c"
# 1 "average.h" 1

typedef enum
{
    NONE,
    NORMAL,
    SQUARED
} average_type_t;

double avg(int *, int, average_type_t);
# 2 "average.c" 2
double avg(int *array, int length,
average_type_t type)
{
    if (length <= 0 || type == NONE)
        return 0;
    double sum = 0.0;
    for (int i = 0; i < length; i++)
    {
        if (type == NORMAL)
            sum += array[i];
        else if (type == SQUARED)
            sum += array[i] * array[i];
    }
    return sum / length;
}
```

Этап 2: компиляция в ассемблерный код

```
$ gcc -S average.c
$ cat average.s
```

После получения единицы трансляции можно переходить ко второму этапу — компиляции. На вход подается единица компиляции, полученная на предыдущем этапе, а на выходе получается соответствующий ассемблерный код. Он все еще может быть прочитан человеком, но уже зависит от аппаратной архитектуры и приближен к оборудованию. Для превращения в машинные инструкции его нужно продолжать обрабатывать.

В рамках данного этапа компилятор анализирует единицу трансляции и превращает ее в ассемблерный код, рассчитанный на ***целевую архитектуру***.



Ассемблерный код (101 строка)

average.s

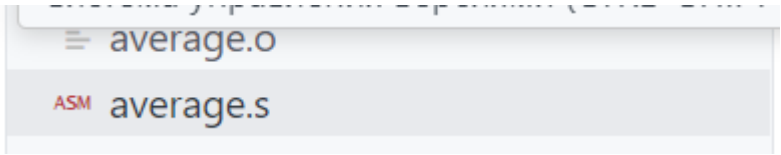
```
1       .file   "average.c"
2       .text
3       .globl  avg
4       .type   avg, @function
5   avg:
6   .LFB0:
7       .cfi_startproc
8       endbr64
9       pushq   %rbp
10      .cfi_def_cfa_offset 16
11      .cfi_offset 6, -16
12      movq    %rsp, %rbp
13      .cfi_def_cfa_register 6
14      movq    %rdi, -24(%rbp)
15      movl    %esi, -28(%rbp)
16      movl    %edx, -32(%rbp)
```

Этап 3: компиляция в машинные инструкции

Цель — сгенерировать инструкции машинного уровня (или машинный код) на основе ассемблерного кода, созданного компилятором на предыдущем этапе. У каждой архитектуры есть свой ассемблер, который может преобразовать собственный ассемблерный код в машинные инструкции.
Файл с машинными инструкциями, который мы сгенерируем в этом подразделе, называется ***объектным файлом***. Переносимые объектные файлы иногда называют промежуточными.
Цель этапа — сгенерировать переносимый объектный файл из ассемблерного кода, созданного компилятором. Любой другой созданный нами продукт будет основан на объектных файлах, сгенерированных ассемблером на данном этапе.

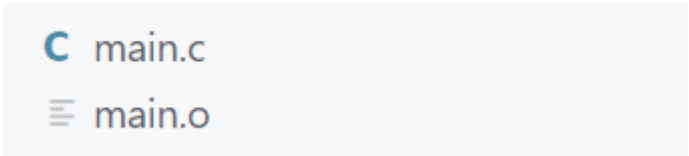
Создание объектного файла из ассемблерного кода

```
$ as average.s -o average.o
```



Компиляция одного из исходников с последующей генерацией переносимого объектного файла

```
$ gcc -c main.c
```



Двоичный файл и объектный файл содержат машинные инструкции и являются синонимами. Как можно видеть в приведенной выше команде, имя выходного объектного файла указывается с помощью параметра -o. Переносимые объектные файлы обычно имеют расширение .o (или .obj в Microsoft Windows). Содержимое объектного файла, будь то .o или .obj, нельзя представить в текстовом виде.

Этап 4: компоновка

На данном этапе необходимо объединить два объектных файла, чтобы создать еще один объектный файл, на сей раз исполняемый. Это делается путем компоновки.

Поддержка новых архитектур

Можно собрать для новой архитектуры, если выполнены два условия.

1. Известна версия ассемблера.
2. Доступна утилита (или программа) от соответствующего производителя, позволяющая скомпилировать ассемблерный код в машинные инструкции.

Двумя инструментами, без которых нельзя обойтись при работе с новой архитектурой, являются:

- компилятор языка C;
- компоновщик.

Попытка скомпоновать объектные файлы путем ручного использования утилиты ld

```
$ ld average.o main.o
ld: warning: cannot find entry symbol
_start; defaulting to 0000000000401000
ld: main.o: in function `main':
main.c:(.text+0x7d): undefined
reference to `printf'
ld: main.c:(.text+0xb9): undefined
reference to `printf'
ld: main.c:(.text+0xd2): undefined
reference to `__stack_chk_fail'
```

Использование gcc для компоновки объектных файлов

```
$ gcc average.o main.o
$ ./a.out
The average: 3.800000
The squared average: 55.800000
```

Подробное описание этапа

Для работы платформам, аналогичным Unix-подобным операционным системам, нужны ранее упомянутые инструменты, такие как ассемблер и компоновщик. Компоновщик по умолчанию в Unix-подобных системах — ld.

Ошибка, нам нужно предоставить компоновщику какие-то другие объектные файлы (не обязательно переносимые), содержащие определения функций printf и __stack_chk_fail.

Более простой способ создания итогового исполняемого файла. Сборка проекта состоит из компиляции исходников с последующей их компоновкой и, возможно, добавлением других библиотек. В результате получаются итоговые продукты.

Препроцессор

Как работает препроцессор: копирует содержимое другого файла или развертывает макрос, заменяя текст.

Препроцессор ничего не знает о языке C; прежде чем выполнять какие-либо дальнейшие действия, он должен разобрать входящий файл с помощью синтаксического анализатора. То есть использует анализатор для поиска директив в исходном коде.

Синтаксический анализатор — это, как правило, программа, которая разбирает входные данные и извлекает из них определенные элементы для дальнейшего анализа и обработки. Чтобы разбить входные данные на более мелкие и полезные элементы, анализатор должен понимать их структуру.

В большинстве Unix-подобных операционных систем есть утилита под названием cpp (расшифровывается как C Pre-Processor, а не как C Plus Plus!). Она входит в пакет разработки для языка C, который поставляется вместе с любой разновидностью Unix. Утилита позволяет предварительно обрабатывать файлы на C, что, собственно, и делают в фоновом режиме такие компиляторы, как gcc.

```
$ cpp average.c
# 0 "average.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "average.c"
# 1 "average.h" 1

typedef enum
{
    NONE,
    NORMAL,
    SQUARED
} average_type_t;

double avg(int *, int, average_type_t);
# 2 "average.c" 2
double avg(int *array, int length,
average_type_t type)
{
    if (length <= 0 || type == NONE)
        return 0;
    double sum = 0.0;
    for (int i = 0; i < length; i++)
    {
        if (type == NORMAL)
            sum += array[i];
        else if (type == SQUARED)
            sum += array[i] * array[i];
    }
    return sum / length;
}
```

Компилятор

Компилятор принимает на вход единицу трансляции, подготовленную препроцессором, и генерирует соответствующие инструкции ассемблера. После компиляции множественных исходников на языке C начинается преобразование сгенерированного ассемблерного кода в переносимые объектные файлы, которые затем объединяются (возможно, с помощью других объектных файлов) в библиотеку или исполняемую программу; при этом применяются такие инструменты, как ассемблер и компоновщик, входящие в состав платформы.

Одна из сложностей компиляции кода на C состоит в получении корректных ассемблерных инструкций, совместимых с целевой архитектурой. Утилита gcc позволяет компилировать один и тот же исходник для разных архитектур (пример. ARM, Intel x86, AMD и др). У каждой архитектуры есть свой набор инструкций процессора, ***полная ответственность за генерацию корректного ассемблерного кода для конкретной архитектуры лежит на компиляторе gcc*** (или другом).

Чтобы преодолеть указанные трудности, gcc (или любой другой компилятор) разделяет данную задачу на два этапа. Вначале он анализирует единицу трансляции и переводит ее в переносимую структуру данных под названием **«дерево абстрактного синтаксиса» (abstract syntax tree, AST);** данная структура не имеет прямого отношения к языку C. Затем на ее основе генерируется подходящий **ассемблерный код** для целевой архитектуры. Все, что относится к определенной архитектуре, происходит на втором этапе. За первый этап отвечает подкомпонент интерфейс компилятора (compiler frontend), а за второй — кодогенератор (compiler backend).

Компиляция и компоновка

AST

```
int main() {
    int var1 = 1;
    double var2 = 2.5;
    int var3 = var1 + var2;
    return 0;
}
```

В качестве абстрактной структуры данных, которая передается между интерфейсом компилятора и кодогенератором, проект LLVM Compiler Infrastructure Project использует промежуточное представление (intermediate representation, LLVM IR).

-FunctionDecl представляет функцию main. Выше находится метаинформация о единице трансляции, которая передается компилятору.

Вслед за FunctionDecl идут элементы (или узлы) дерева для объявлений, бинарных операций, инструкции return и даже для выражений неявного приведения типов.

Преимущество генерации AST для исходного кода состоит в том, что вы можете переставить местами инструкции, удалить неиспользуемые ветки и заменить некоторые блоки в целях повышения производительности, не меняя при этом логику программы.

```
erPoint/lection_20_3$ clang -Xclang -ast-dump -fsyntax-only main.c
TranslationUnitDecl 0x9638e8 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x964110 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128
| `-BuiltinType 0x963eb0 '__int128'
|-TypedefDecl 0x964180 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigne
| `-BuiltinType 0x963ed0 'unsigned __int128'
|-TypedefDecl 0x964488 <<invalid sloc>> <invalid sloc> implicit __NSConstantString '
stantString_tag'
| `-RecordType 0x964260 'struct __NSConstantString_tag'
|   `-Record 0x9641d8 '__NSConstantString_tag'
|-TypedefDecl 0x964520 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list
| `-PointerType 0x9644e0 'char *'
|   `-BuiltinType 0x963990 'char'
|-TypedefDecl 0x964818 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list 's
t_tag[1]'
| `-ConstantArrayType 0x9647c0 'struct __va_list_tag[1]' 1
|   `-RecordType 0x964600 'struct __va_list_tag'
`-CompoundStmt 0x9ba8b0 <line:2:1, line:7:1>
  |-DeclStmt 0x9ba658 <line:3:5, col:17>
  | `-VarDecl 0x9ba5d0 <col:5, col:16> col:9 used var1 'int' cinit
  |   `-IntegerLiteral 0x9ba638 <col:16> 'int' 1
  |-DeclStmt 0x9ba710 <line:4:5, col:22>
  | `-VarDecl 0x9ba688 <col:5, col:19> col:12 used var2 'double' cinit
  |   `-FloatingLiteral 0x9ba6f0 <col:19> 'double' 2.500000e+00
  |-DeclStmt 0x9ba868 <line:5:5, col:27>
  | `-VarDecl 0x9ba740 <col:5, col:23> col:9 var3 'int' cinit
  |   `-ImplicitCastExpr 0x9ba850 <col:16, col:23> 'int' <FloatingToIntegral>
  |     `-BinaryOperator 0x9ba830 <col:16, col:23> 'double' '+'
  |       |-ImplicitCastExpr 0x9ba818 <col:16> 'double' <IntegralToFloating>
  |       | `-ImplicitCastExpr 0x9ba7e8 <col:16> 'int' <LValueToRValue>
  |-DeclStmt 0x9ba658 <line:3:5, col:17>
  | `-VarDecl 0x9ba5d0 <col:5, col:16> col:9 used var1 'int' cinit
  |   `-IntegerLiteral 0x9ba638 <col:16> 'int' 1
  |-DeclStmt 0x9ba710 <line:4:5, col:22>
  | `-VarDecl 0x9ba688 <col:5, col:19> col:12 used var2 'double' cinit
  |   `-FloatingLiteral 0x9ba6f0 <col:19> 'double' 2.500000e+00
  |-DeclStmt 0x9ba868 <line:5:5, col:27>
  | `-VarDecl 0x9ba740 <col:5, col:23> col:9 var3 'int' cinit
  |   `-ImplicitCastExpr 0x9ba850 <col:16, col:23> 'int' <FloatingToIntegral>
  |     `-BinaryOperator 0x9ba830 <col:16, col:23> 'double' '+'
  |       |-ImplicitCastExpr 0x9ba818 <col:16> 'double' <IntegralToFloating>
  |       | `-ImplicitCastExpr 0x9ba7e8 <col:16> 'int' <LValueToRValue>
  |       |   `-DeclRefExpr 0x9ba7a8 <col:16> 'int' lvalue Var 0x9ba5d0 'var1' 'int'
  |       `-ImplicitCastExpr 0x9ba800 <col:23> 'double' <LValueToRValue>
  |         `-DeclRefExpr 0x9ba7c8 <col:23> 'double' lvalue Var 0x9ba688 'var2' 'double'
  `-ReturnStmt 0x9ba8a0 <line:6:5, col:12>
    `-IntegerLiteral 0x9ba880 <col:12> 'int' 0
```

Ассемблер

Для создания объектных файлов с подходящими машинными инструкциями платформа должна предоставлять ассемблер. В Unix-подобных операционных системах ассемблер можно использовать с помощью специальной утилиты.

Если установить на том же компьютере другую Unix-подобную операционную систему, то версия ассемблера может измениться. Это очень важно, поскольку вы можете получить другие объектные файлы, хотя аппаратное обеспечение и соответствующие машинные инструкции останутся прежними!

Объектные файлы, сгенерированные в Linux для 64-битной архитектуры AMD, могут отличаться от результатов компиляции той же программы в другой ОС, такой как FreeBSD или macOS, и на том же оборудовании. Это значит, что несмотря на одинаковые машинные инструкции, объектные файлы не могут совпадать ввиду различных форматов, используемых в разных системах.

Иными словами, каждая ОС поддерживает свой двоичный формат (формат объектных файлов) для хранения инструкций машинного уровня. Поэтому содержимое объектного файла определяется двумя факторами: архитектурой (аппаратным обеспечением) и операционной системой. Эту комбинацию мы обычно называем платформой.

В Linux используется формат ELF (Executable and Linking Format — формат исполняемых и компонуемых файлов). Он применяется для всех исполняемых/объектных файлов и разделяемых библиотек. Проще говоря, в Linux ассемблер создает ELF-файлы.

Компоновщик

Сборка проекта на языке C начинается с компиляции всех его исходников в соответствующие переносимые объектные файлы. Данный этап необходим для создания конечных продуктов, но одного его недостаточно. Остается сделать еще один шаг. Из проекта на C/C++ можно получить следующие продукты:

- ряд исполняемых файлов, которые в большинстве Unix-подобных систем имеют расширение .out. В Microsoft Windows, как правило, используется расширение .exe;
- ряд статических библиотек с расширением .a (в большинстве Unix-подобных систем) или .lib (в Microsoft Windows);
- ряд динамических библиотек или разделяемых объектных файлов. В Unix-подобных операционных системах они обычно имеют расширение .so, а в macOS и Microsoft Windows — .dylib и .dll соответственно.

Переносимые объектные файлы не входят в число этих продуктов. Они представляют собой промежуточные ресурсы, которые используются на этапе компоновки для получения перечисленных выше продуктов; далее они не нужны. Вся ответственность за создание конечных продуктов из переносимых объектных файлов ложится на компоновщик.

Последнее, но важное замечание относится к терминам, которые мы здесь используем: все три продукта называются объектными файлами. Поэтому объектные файлы, сгенерированные ассемблером в качестве промежуточных ресурсов, лучше называть переносимыми.

Принцип работы компоновщика

Компоновщик объединяет все переносимые объектные файлы вдобавок к указанным статическим библиотекам, чтобы получить готовую программу.

Объектный файл содержит инструкции машинного уровня, эквивалентные единице трансляции. Однако они хранятся не в произвольном порядке, а сгруппированы в так называемые символы.

На самом же деле в объектном файле хранится много элементов, но символы — компонент, который объясняет принцип работы компоновщика и то, как из нескольких объектных файлов получается один.

example.c

```c
int average(int a, int b)
{
    return (a + b) / 2;
}
int sum(int *numbers, int count)
{
    int sum = 0;
    for (int i = 0; i < count; i++)
    {
        sum += numbers[i];
    }
    return sum;
}
```

Принцип работы компоновщика

Компиляция исходного файла

```
$ gcc -c example.c -o example.o
```

Использование утилиты nm для просмотра символов, определенных в переносимом объектном файле

```
$ nm example.o
0000000000000000 T average
0000000000000021 T sum
```

Применение утилиты readelf для просмотра таблицы символов в переносимом объектном файле

```
$ readelf -s example.o

Symbol table '.symtab' contains 5 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS example.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000000    33 FUNC    GLOBAL DEFAULT    1 average
     4: 0000000000000021    73 FUNC    GLOBAL DEFAULT    1 sum
```

Принцип работы компоновщика

Применение утилиты objdump для просмотра инструкций символов, определенных в переносимом объектном файле

```
$ objdump -d example.o

example.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <average>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   89 7d fc                mov    %edi,-0x4(%rbp)
   b:   89 75 f8                mov    %esi,-0x8(%rbp)
   e:   8b 55 fc                mov    -0x4(%rbp),%edx
  ...
```

Принцип работы компоновщика

decls.h

```c
#ifndef DECLS_H
#define DECLS_H
int add(int, int);
int multiply(int, int);
#endif
```

add.c

```c
#include "decls.h"
int add(int a, int b) {
    return a + b;
}
```

multiply.c

```c
#include "decls.h"
int multiply(int a, int b) {
    return a * b;
}
```

main.c

```c
#include "decls.h"
int main(int argc, char **argv)
{
    int x = add(4, 5);
    int y = multiply(9, x);
    return 0;
}
```

Пример состоит из четырех файлов: трех исходников и одного заголовка. В заголовочном файле объявлено две функции, определения которых находятся в разных исходных файлах. Третий исходник содержит функцию main.
Функции, представленные в примере, крайне просты, и после компиляции каждая из них будет состоять всего из нескольких машинных инструкций.

Принцип работы компоновщика

```
$ gcc -c add.c -o add.o
$ gcc -c multiply.c -o multiply.o
$ gcc -c main.c -o main.o
```

Вывод символов, определенных в файле add.o

```
$ nm add.o
0000000000000000 T add
```

Вывод символов, определенных в файле multiply.o

```
$ nm multiply.o
0000000000000000 T multiply
```

Вывод символов, определенных в файле main.o

```
$ nm main.o
                 U add
0000000000000000 T main
                 U multiply
```

U (unresolved — «неразрешенная»)

Принцип работы компоновщика

Подытожим: мы получили три объектных файла, в одном из которых есть неразрешенные символы. Таким образом, перед нами стоит вполне понятная задача: предоставить компоновщику символы, которые можно найти в других объектных файлах. Получив все необходимые символы, компоновщик сможет продолжить их объединение в итоговый рабочий исполняемый файл. Если компоновщику не удастся найти определения неразрешенных символов, то он прервет работу и проинформирует нас об ошибке компоновки.

```
$ gcc add.o multiply.o main.o
```

Посмотрим, что произойдет, если компоновщику не удастся найти подходящие определения. Укажем для этого только два промежуточных объектных файла: main.o и add.o

```
$ gcc add.o main.o
/usr/bin/ld: main.o: in function `main':
main.c:(.text+0x30): undefined reference to `multiply'
collect2: error: ld returned 1 exit status
```

```
$ gcc add.o multiply.o
/usr/bin/ld: /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-
gnu/Scrt1.o: in function `_start':
(.text+0x1b): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

План лекции

Раздельная компиляция	Работа с простыми числами	Вычислительная геометрия
50 минут	15 минут	25 минут

Работа с простыми числами

В 1640 году Ферма предположил, что формула

$$F_n = 2^{2^n} + 1$$

всегда дает простые числа, и числа этого вида получили название "числа Ферма". Предположение Ферма справедливо для n от 0 до 4, но в 1732 году Эйлер обнаружил, что

$$F_5 = 2^{2^5} + 1 = 641 \cdot 6\,700\,417.$$

(Мы уже имели дело с этими множителями, когда рассматривали вопросы деления на константу на 32-битовом компьютере.) Затем в 1880 году Ландри показал, что

$$F_6 = 2^{2^6} + 1 = 274\,177 \cdot 67\,280\,421\,310\,721.$$

В настоящее время известно, что F - составные числа для многих больших значений n, в частности для и от 7 до 16 включительно. В настоящее время неизвестно ни одно значение n > 4, для которого число Ферма было бы простым. Увы, гипотеза оказалась слишком поспешной…

Почему же Ферма использовал двойное возведение в степень? Он знал, что если ты имеет нечетный множитель, отличный от 1, то 2^m +1 -- составное число. Если m = ab, где b нечетно и не равно 1, то

$$2^{ab} + 1 = (2^a + 1)(2^{a(b-1)} - 2^{a(b-2)} + 2^{a(b-3)} - \cdots + 1).$$

Зная это, Ферма заинтересовался числами 2^m +1, такими, где т не имеет ни одного нечетного множителя, отличного от 1, т.е. m = 2^n. Он испробовал несколько значений n и убедился, что они дают простые числа 2^n + 1.

Работа с простыми числами

В 1772 году Эйлером был открыт удивительный полином

$$f(n) = n^2 + n + 41,$$

обладающий тем свойством, что он дает простые числа для всех и от 0 до 39.

Этот результат может быть расширен. Так как

$$f(-n) = n^2 - n + 41 = f(n-1),$$

f(-n) дает простые числа для каждого n из диапазона от 1 до 40, т.е. f(n) дает простые числа для всех и от -1 до -40. Таким образом, полином

$$f(n-40) = (n-40)^2 + (n-40) + 41 = n^2 - 79n + 1601$$

дает простые числа для всех n от 0 до 79. (Однако эстетическая привлекательность формулы при этом теряется, так как она дает повторяющиеся и немонотонные результаты: для n = 0,1, …,79 формула дает числа 1601, 1523, 1447, …, 43, 41, 41, 43, …, 1447, 1523, 1601.)

Несмотря на всю привлекательность полинома, открытого Эйлером, известно, что не существует полиномиальной формулы f(n), которая давала бы простое число для каждого n (не считая тривиального случая константного полинома типа f(n) = 5). Более того, справедлива следующая теорема.

Работа с простыми числами

Теорема. Если $f(n) = P(n, 2^n, 3^n, \dots, k^n)$ представляет собой полином с целочисленными коэффициентами от своих аргументов и f(n)при n→∞, то f(n) является составным значением для бесконечного множества значений n.

Следовательно, формула типа $n^2 \cdot 2^n + 2n^3 + 2n + 5$ должна давать бесконечное количество составных чисел. С другой стороны, теорема ничего не говорит о формулах, содержащих члены наподобие 2^{2^n}, n^n или n!.

Работа с простыми числами

Формула для n-го целого простого числа может быть получена с помощью функции получения максимального целого, не превосходящего данное число ("пол"), и магического значения
a = 0.2030050007000110000 13…
Число a в десятичной системе счисления представляет собой первое простое число, записанное в первой позиции после десятичной точки, второе простое число, записанное в следующих двух позициях, третье, записанное в следующих трех позициях, и т.д. При этом для очередного простого числа всегда найдется достаточно свободного места, так как $p_n < 10^n$. Не доказывая этого, просто заметим: поскольку известно, что между n и 2n (n≥2) всегда есть простое число, значит, между и n и 10n оно есть наверняка, а отсюда следует, что $p_n < 10^n$. Формула для n-го простого числа выглядит следующим образом:

$$p_n = \left[10^{\frac{n^2+n}{2}} a\right] - 10^n \left[10^{\frac{n^2-n}{2}} a\right]$$

где было использовано соотношение

$$\sum_{i=1}^{n} n = \frac{n^2+n}{2}.$$

Например:

$$p_3 = [10^6 a] - 10^3 [10^3 a] = 203005 - 203000 = 5.$$

Это довольно дешевый трюк, который требует знания окончательного результата для корректного определения a.

Работа с простыми числами

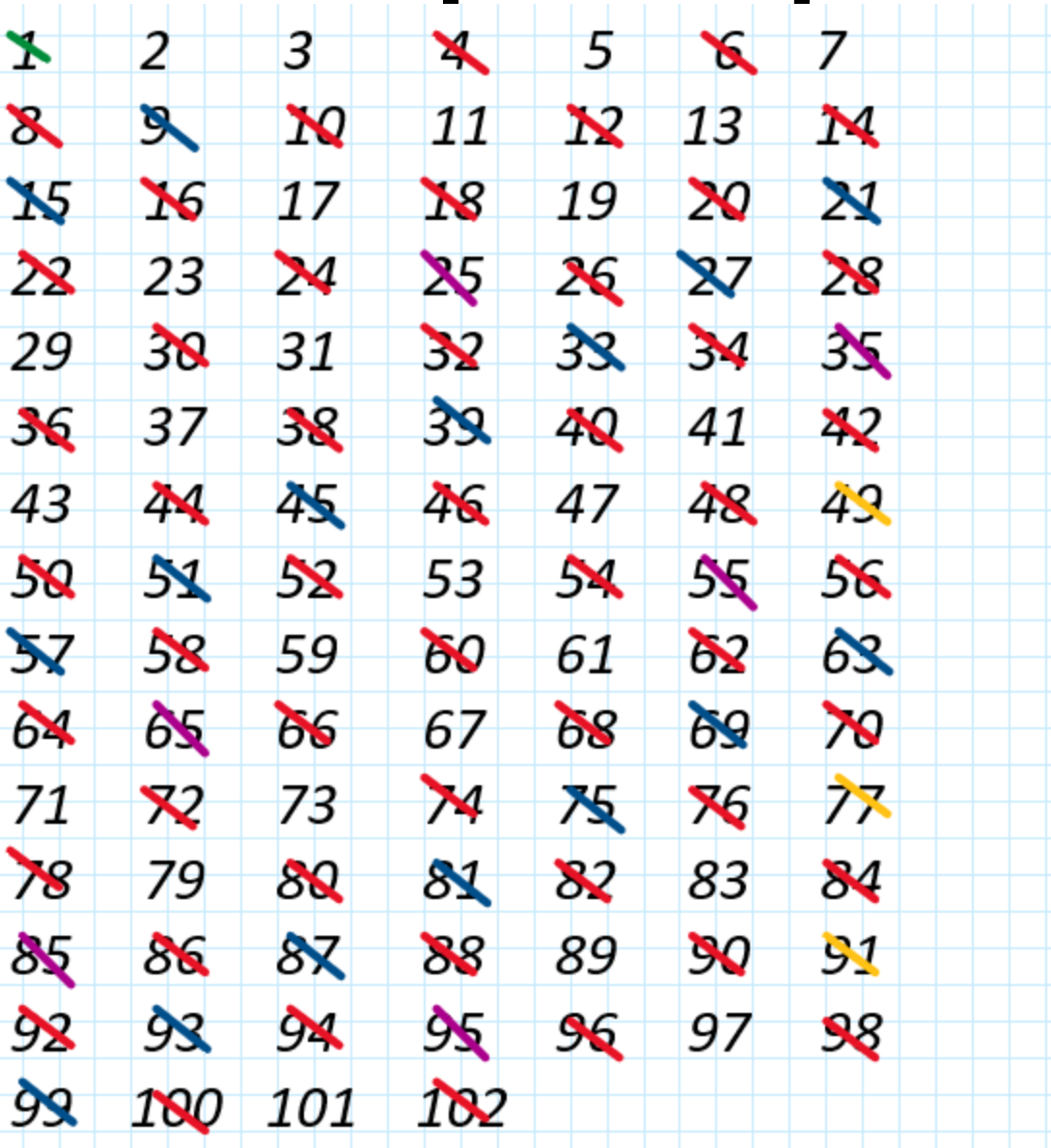
Кристофер.П. Вилланс (C.P. Willans) предложил несколько формул для вычисления простых чисел:

$$p_n = 1 + \sum_{m=2}^{2^n} \left[\sqrt[m]{n} \left(\sum_{x=1}^{m} \left[\cos^2 \pi \frac{(x-1)! + 1}{x} \right] \right)^{-\frac{1}{n}} \right]$$

$$\pi(x) = \sum_{k=2}^{x} \left[\cos^2 \left(\pi \frac{(x-1)! + 1}{x} \right) \right],$$

где π(x)— количество простых чисел, не превышающих x. Формулы Уилланса математически корректны и всегда работают для вычисления простых чисел, но они крайне неэффективны на практике из-за огромного числа операций.

Решето Эратосфена (Сегментированное)



Временная сложность: O(nloglogn), что делает алгоритм одним из самых быстрых для поиска простых чисел в диапазоне.

Пространственная сложность: O(n) или (O(√n) – сегментированное)

Пол Притчард (Paul Pritchard) разработал несколько алгоритмов для поиска простых чисел, включая так называемое "решето Притчарда" (Sieve of Pritchard) или "динамическое колесное решето". Я опишу его наиболее известный алгоритм — колесное решето — и его сложность.

Временная сложность: O(n/loglogn). Это сублинейная асимптотика, что делает его быстрее, чем O(nloglogn) решета Эратосфена, за счёт минимизации операций удаления составных чисел.

Пространственная сложность: O(n/loglogn) бит в базовой версии, но Притчард предложил вариант с O(n/loglogn) бит, если хранить только колесо и небольшие буферы.

План лекции

Раздельная компиляция	Работа с простыми числами	Вычислительная геометрия
50 минут	15 минут	25 минут

Введение

Вычислительная геометрия - это раздел информатики, изучающий алгоритмы, предназначенные для решения геометрических задач. В современных инженерных и математических расчетах вычислительная геометрия, в числе других областей знаний, применяется в машинной графике, в робототехнике, при разработке СБИС, при автоматизированном проектировании, в металлургии, в статистике…

Роль входных данных в задачах вычислительной геометрии обычно играет описание множества таких геометрических объектов, как точки, отрезки или вершины многоугольника в порядке обхода против часовой стрелки. На выходе часто дается ответ на такие запросы об этих объектах, как наличие пересекающихся линий или параметры новых геометрических объектов, например выпуклой оболочки множества точек (это минимальный выпуклый многоугольник, содержащий данное множество).

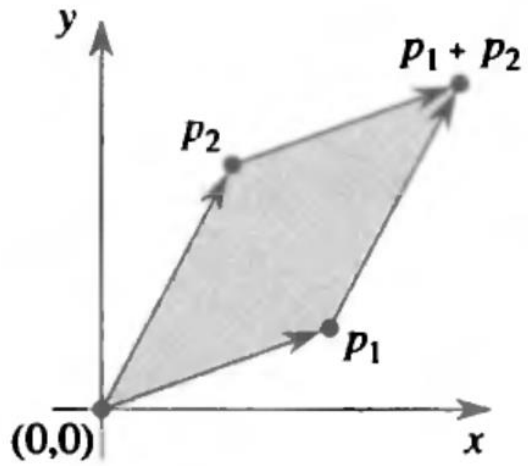
Свойства отрезков

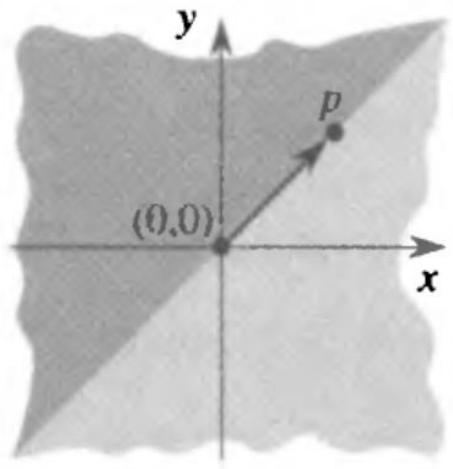
Выпуклой комбинацией (convex combination) двух различных точек $p_1 = (x_1,y_1)$ и $p_2 = (x_2,y_2)$ называется любая точка $p_3 = (x_3, y_3)$, такая, что для некоторого значения a, принадлежащего интервалу $0 < \alpha < 1$, выполняются равенства $x_3 = x_1 + (1 - \alpha)x_2$ и $y_3 = \alpha y_1 + (1 - a)y_2$ (пишут также $p_3 = \alpha p_1 + (1 - \alpha)p_2$). Интуитивно понятно, что в роли p_3 может выступать любая точка, которая принадлежит прямой, соединяющей точки p_1 и p_2, и находится между этими точками. Если заданы две различные точки, p_1 и p_2, то отрезком (line segment) $\overline{p_1 p_2}$ называется множество выпуклых комбинаций p_1 и p_2. Точки p_1 и p_2 называются конечными точками (endpoints) отрезка $\overline{p_1 p_2}$. Иногда играет роль порядок следования точек p_1 и p_2, и тогда говорят о направленном отрезке (directed segment) $\overrightarrow{p_1 p_2}$. Если точка p_1 совпадает с началом координат (origin), т.е. имеет координаты (0,0), то направленный отрезок $\overrightarrow{p_1 p_2}$ можно рассматривать как вектор (vector) p_2.

Векторное произведение

Вычисление векторного произведения составляет основу методов работы с отрезками. Рассмотрим векторы p_1 и p_2. Векторное произведение (cross product) $p_1 \times p_2$ можно интерпретировать как знаковое значение площади параллелограмма, образованного точками (0,0), p_1, p_2 и $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$. Эквивалентное, но более полезное определение векторного произведения - определитель матрицы:

$$p_1 \times p_2 = det\begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1 = -p_2 \times p_1.$$





Векторное произведение

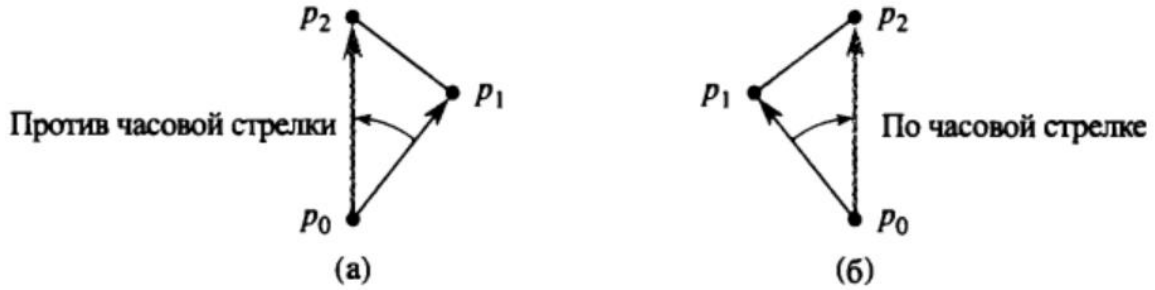
Чтобы определить, находится ли направленный отрезок $\overrightarrow{p_0p_1}$ по часовой стрелке от направленного отрезка $\overrightarrow{p_0p_2}$ или против относительно их общей точки p_0, достаточно использовать ее как начало координат. Сначала обозначим величину $p_1 - p_0$ как вектор $p_1' = (x_1', y_1')$, где $x_1' = x_1 - x_0$ и $y_1' = y_1 - y_0$, а затем введем аналогичные обозначения для величины $p_2 - p_0$. После этого вычислим векторное произведение

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(x_2 - x_0) - (x_2 - x_0)(y_1 - y_0).$$

Если это векторное произведение положительно, $\overrightarrow{p_0p_1}$ находится по часовой стрелке от $\overrightarrow{p_0p_2}$; если отрицательно - против часовой стрелки.

Поворот последовательных отрезков

Следующий вопрос заключается в том, куда сворачивают два последовательных отрезка, $\overline{p_0p_1}$ и $\overline{p_1p_2}$, в точке p_1 - влево или вправо. Можно привести эквивалентную формулировку этого вопроса - определить знак угла $\angle\overrightarrow{p_1p_2}$,. Векторное произведение позволяет ответить на этот вопрос, не вычисляя величину угла.



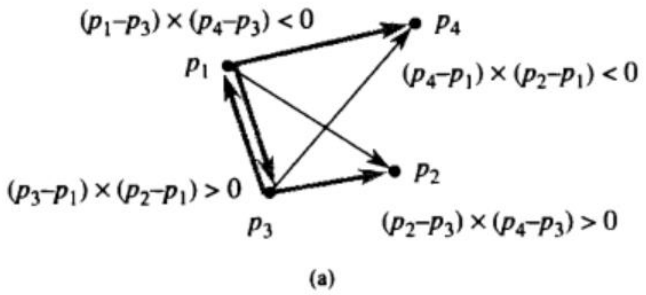
Мы просто проверяем, находится ли направленный отрезок $\overrightarrow{p_0p_2}$ по часовой стрелке или против часовой стрелки относительно направленного отрезка $\overrightarrow{p_0p_1}$. Для этого вычисляется векторное произведение $(p_2 – p_0) \times (p_1 – p_0)$.

Если эта величина отрицательна, то направленный отрезок $\overrightarrow{p_0p_2}$ находится против часовой стрелки по отношению к направленному отрезку $\overrightarrow{p_0p_1}$, так что в точке p_1 мы делаем поворот влево. Положительное значение векторного произведения указывает на ориентацию по часовой стрелке и поворот вправо. Нулевое векторное произведение означает, что точки p_0, p_1 и p_2 коллинеарны.

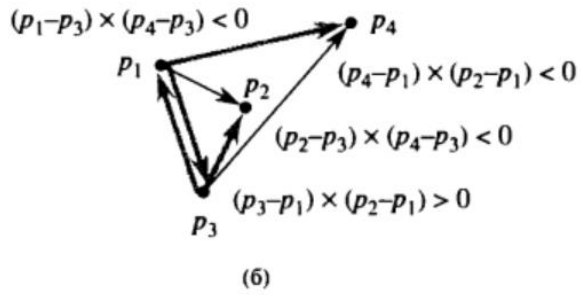
Пересекаются ли два отрезка?

Чтобы определить, пересекаются ли два отрезка, следует проверить, пресекает ли каждый из них прямую, содержащую другой отрезок. Отрезок $\overline{p_1 p_2}$ пересекает (straddles) прямую, если конечные точки отрезка p_1 и p_2 лежат в разных полуплоскостях, на которые прямая разбивает плоскость. В граничном случае точка p_1 или точка p_2 (или обе эти точки) лежит непосредственно на прямой. Два отрезка пересекаются тогда и только тогда, когда выполняется одно из сформулированных ниже условий (или оба эти условия одновременно).

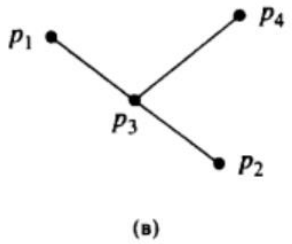
1. Каждый отрезок пересекает прямую, на которой лежит другой отрезок.
2. Конечная точка одного из отрезков лежит на другом отрезке (граничный случай).



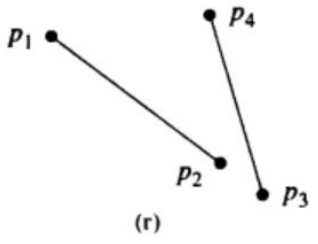
(а)



(б)



(в)



(г)

Пересекаются ли два отрезка?

```c
#include <stdio.h>
#include <stdbool.h>
// Структура для точки
typedef struct {
    int x, y;
} Point;
// Функция направления (векторное произведение)
int Direction(Point pi, Point pj, Point pk) {
    return ((pk.x - pi.x) * (pj.y - pi.y)) - ((pk.y - pi.y) * (pj.x - pi.x));
}
// Проверка, лежит ли точка pk на отрезке pi-pj
bool OnSegment(Point pi, Point pj, Point pk)
{
    int min_x = pi.x < pj.x ? pi.x : pj.x;
    int max_x = pi.x > pj.x ? pi.x : pj.x;
    int min_y = pi.y < pj.y ? pi.y : pj.y;
    int max_y = pi.y > pj.y ? pj.y : pi.y;
    return (min_x <= pk.x && pk.x <= max_x) && (min_y <= pk.y && pk.y <= max_y);
}
```

Пересекаются ли два отрезка?

```
// Проверка пересечения отрезков p1-p2 и p3-p4
bool SegmentsIntersect(Point p1, Point p2, Point p3, Point p4) {
    int d1 = Direction(p3, p4, p1);
    int d2 = Direction(p3, p4, p2);
    int d3 = Direction(p1, p2, p3);
    int d4 = Direction(p1, p2, p4);
    // Основной случай: отрезки пересекаются, если направления противоположны
    if (((d1 > 0 && d2 < 0)||(d1 < 0 && d2 > 0))&&((d3 > 0 && d4 < 0)||(d3 < 0 && d4 > 0)))
        return true;
    // Проверка особых случаев (точка на отрезке)
    else if (d1 == 0 && OnSegment(p3, p4, p1))
        return true;
    else if (d2 == 0 && OnSegment(p3, p4, p2))
        return true;
    else if (d3 == 0 && OnSegment(p1, p2, p3))
        return true;
    else if (d4 == 0 && OnSegment(p1, p2, p4))
        return true;
    else
        return false;
}
```

Определение наличия пересекающихся отрезков

В этом алгоритме используется метод, известный под названием "выметание", который часто встречается в алгоритмах вычислительной геометрии. С помощью данного алгоритма или его вариации можно решать и другие задачи вычислительной геометрии. Время работы этого алгоритма равно O(nlgn), где n - количество заданных отрезков. В нем лишь определяется, существуют пересечения или нет, но не выводятся данные обо всех этих пересечениях.

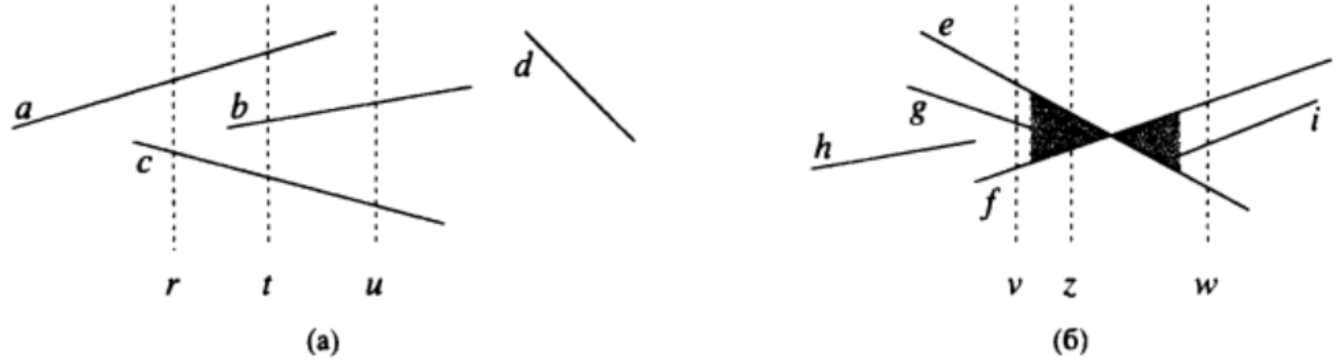
В методе выметания (sweeping) по заданному множеству геометрических объектов проводится воображаемая вертикальная выметающая прямая (sweep line), которая обычно движется слева направо. Измерение, вдоль которого двигается выметающая прямая (в данном случае это измерение x), трактуется как время. Выметание предоставляет способ упорядочения геометрических объектов, обычно путем размещения их параметров в динамической структуре данных, что позволяет воспользоваться взаимоотношениями между этими объектами.

Чтобы описать алгоритм и доказать его способность корректно определить, пересекаются ли какие-либо из n отрезков, сделаем два упрощающих предположения.

Упорядочение отрезков

Поскольку предполагается, что вертикальные отрезки отсутствуют, каждый входной отрезок пересекает данную вертикальную выметающую прямую в одной точке. Таким образом, отрезки, пересекающие вертикальную выметающую прямую, можно упорядочить по координате y точки пересечения. Чтобы быть более точными, рассмотрим два отрезка, s_1 и s_2. Говорят, что они сравнимы (comparable) в координате x, если вертикальная выметающая прямая с координатой x пересекает оба этих отрезка. Говорят также, что отрезок s_1 расположен над (above) отрезком s_2 в x (записывается $s_1 \succcurlyeq_x s_2$), если отрезки s_1 и s_2 сравнимы в координате x и точка пересечения отрезка s_1 с выметающей прямой в координате x находится выше, чем точка пересечения отрезка s_2 с этой выметающей прямой, или если s_1 и s_2 пересекаются на выметающей прямой.

Пример, выполняются соотношения a $\succcurlyeq_r$ c, a $\succcurlyeq_t$ b, b $\succcurlyeq_t$ c, a $\succcurlyeq_t$ c и b $\succcurlyeq_u$ c. Отрезок d не сравним ни с каким другим отрезком.



Перемещение выметающей прямой

В выметающих алгоритмах обычно обрабатываются два набора данных.

1. ***Состояние относительно выметающей прямой*** (sweep-line status) описывает соотношения между объектами, пересекаемыми выметающей прямой.
2. ***Таблица, или расписание, точек-событий*** (event-point schedule) – это последовательность координат x, упорядоченных слева направо, в которой определяются точки останова выметающей прямой. Каждая такая точка останова называется точкой-событием (event point). По мере перемещения выметающей прямой слева направо, когда она достигает x-координаты точки-события, выметание приостанавливается, выполняется обработка точки-события, после чего выметание продолжается. Изменение состояния относительно выметающей прямой происходит только в точках-событиях.

Состояние относительно выметающей прямой является полностью упорядоченным множеством T, в котором необходимо реализовать такие операции.

Insert(T, s): вставка отрезка s в T.

Delete(T, s): удаление отрезка s из T.

Above(T, s): возврат отрезка, находящегося непосредственно над отрезком s в T.

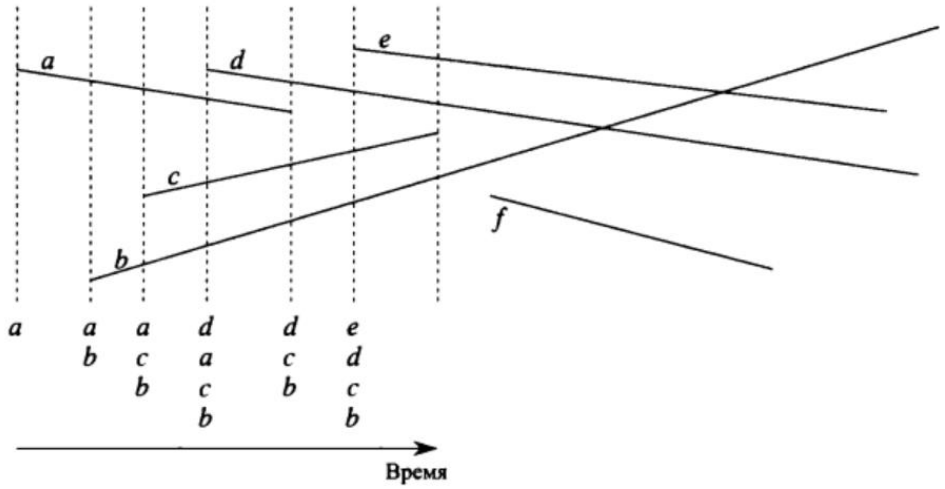
Below(T, s): возврат отрезка, находящегося непосредственно под отрезком s в T.

Возможна ситуация, когда отрезки s_1 и s_2 находятся взаимно один над другим в упорядочении T; такая ситуация может возникнуть, если s_1 и s_2 пересекаются на выметающей прямой, для которой задается упорядочение T. В этом случае эти два отрезка могут находиться в T в любом порядке.

Если всего имеется n входных отрезков, каждую из перечисленных выше операций с помощью красно-черных деревьев можно выполнить за время O(lgn).

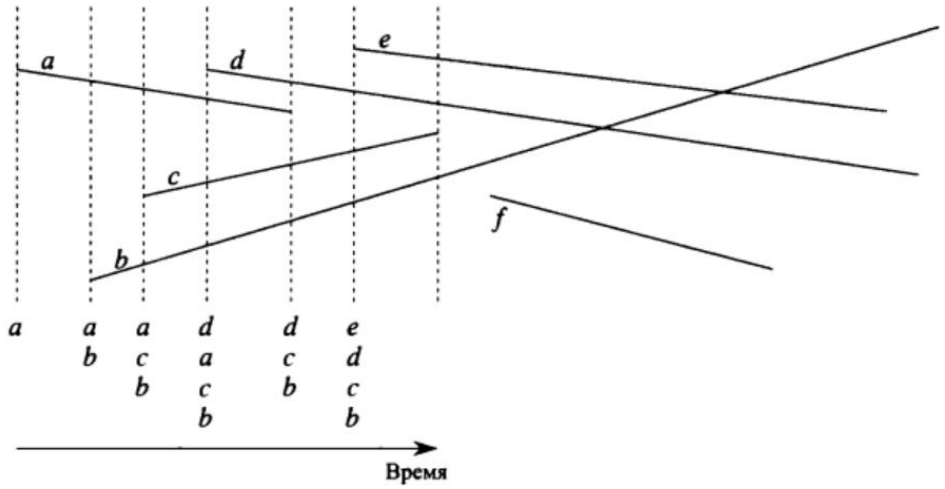
Перемещение выметающей прямой

Any-Segments-Intersect(S)
1 T = Ø
2 Сортировка конечных точек отрезков S слева направо с разрешением совпадений путем помещения левых конечных точек перед правыми и путем помещения точек с меньшими координатами y перед теми, координата y которых больше
3 for каждой точки p в отсортированном списке конечных точек
4 if p является левой конечной точкой отрезка s
5 Insert(T, s)
6 if (Above(T, s) существует и пересекает s) или (Below(T, s) существует и пересекает s)
7 return TRUE
8 if p является правой конечной точкой отрезка s
9 if (и Above(T, s), и Below(T, s) существуют) и (Above(T,s) пересекает Below(T,s))
10 return TRUE
11 Delete(T,s)
12 return FALSE



Перемещение выметающей прямой

Any-Segments-Intersect(S)
1 T = Ø
2 Сортировка конечных точек отрезков S слева направо с разрешением совпадений путем помещения левых конечных точек перед правыми и путем помещения точек с меньшими координатами y перед теми, координата y которых больше
3 for каждой точки p в отсортированном списке конечных точек
4 if p является левой конечной точкой отрезка s
5 Insert(T, s)
6 if (Above(T, s) существует и пересекает s) или (Below(T, s) существует и пересекает s)
7 return TRUE
8 if p является правой конечной точкой отрезка s
9 if (и Above(T, s), и Below(T, s) существуют) и (Above(T,s) пересекает Below(T,s))
10 return TRUE
11 Delete(T,s)
12 return FALSE

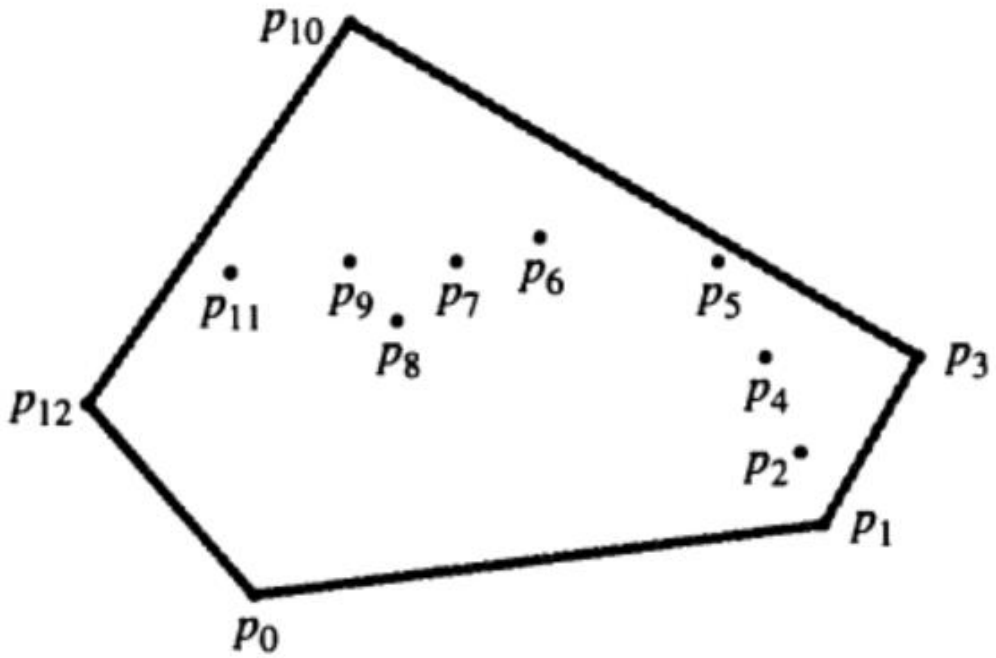


Время работы

Если множество S содержит n отрезков, то процедура Any-Segments-Intersect выполняется за время O(nlogn). Выполнение строки 1 занимает время O(1). Строка 2 с помощью сортировки слиянием или пирамидальной сортировки выполняется за время O(nlogn). Поскольку всего имеется 2n точек-событий, а цикл for в строках 3-11 выполняет не более одной итерации на точку, он выпалняется не более 2n раз. На каждую итерацию требуется время O(logn), поскольку выполнение каждой операции в красно-черном дереве занимает время O(logn), и каждая проверка пересечения выполняется за время O(1). Таким образом, полное время работы равно O(nlgn).

Поиск выпуклой оболочки

Выпуклой оболочкой (convex hull) множества точек Q (обозначается CH(Q)) называется наименьший выпуклый многоугольник P, такой, что каждая точка из Q находится либо на границе многоугольника P, либо в его внутренней области. Неявно предполагается, что все точки множества Q различны и что в нем имеется как минимум три не коллинеарные точки. Интуитивно можно представлять каждую точку множества Q в виде торчащего из доски гвоздя; тогда выпуклая оболочка будет иметь форму, полученную в результате наматывания на гвозди тугой резиновой нити.



Поиск выпуклой оболочки

- В **инкрементном методе** (incremental method) точки сначала сортируются слева направо, в результате чего получается последовательность (p_1, p_2, p_n). На i-м этапе выпуклая оболочка i - 1 крайних слева точек CH({p_1,p_2, p_{i-1}}) обновляется в соответствии с положением i-й слева точки, формируя, таким образом, оболочку CH({p_1,p_2,p_i}).
- В **методе декомпозиции** (devide-and-conquer method) множество n точек за время Θ(n) разбивается на два подмножества, в одном из которых содержится [n/2] крайних слева точек, а во втором - [n/2] крайних справа. Затем рекурсивно вычисляются выпуклые оболочки этих подмножеств, которые впоследствии объединяются с помощью одного остроумного метода за время O(n). Время работы этого метода описывается знакомым рекуррентным соотношением T(n) = 2T(n/2) + O(n), решение которого дает время работы O(nlgn).
- **Метод отсечения и поиска** (prune-and-search method) - время работы в наихудшем случае ведет себя линейно. В нем строится верхняя часть (или "верхняя цепь") выпуклой оболочки путем многократного отбрасывания фиксированной части оставшихся точек до тех пор, пока не останется только верхняя цепь выпуклой оболочки. Затем то же самое выполняется с нижней цепью. В асимптотическом пределе этот метод самый быстрый: если выпуклая оболочка содержит h вершин, время его работы равно O(nlgh).

Поиск выпуклой оболочки

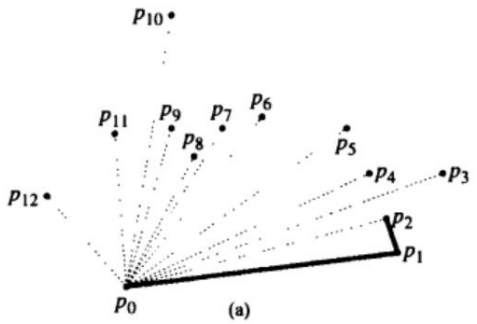
- В **инкрементном методе** (incremental method) точки сначала сортируются слева направо, в результате чего получается последовательность (p_1, p_2, p_n). На i-м этапе выпуклая оболочка i - 1 крайних слева точек CH({p_1,p_2, p_{i-1}}) обновляется в соответствии с положением i-й слева точки, формируя, таким образом, оболочку CH({p_1,p_2,p_i}).
- В **методе декомпозиции** (devide-and-conquer method) множество n точек за время Θ(n) разбивается на два подмножества, в одном из которых содержится [n/2] крайних слева точек, а во втором - [n/2] крайних справа. Затем рекурсивно вычисляются выпуклые оболочки этих подмножеств, которые впоследствии объединяются с помощью одного остроумного метода за время O(n). Время работы этого метода описывается знакомым рекуррентным соотношением T(n) = 2T(n/2) + O(n), решение которого дает время работы O(nlgn).
- **Метод отсечения и поиска** (prune-and-search method) - время работы в наихудшем случае ведет себя линейно. В нем строится верхняя часть (или "верхняя цепь") выпуклой оболочки путем многократного отбрасывания фиксированной части оставшихся точек до тех пор, пока не останется только верхняя цепь выпуклой оболочки. Затем то же самое выполняется с нижней цепью. В асимптотическом пределе этот метод самый быстрый: если выпуклая оболочка содержит h вершин, время его работы равно O(nlgh).

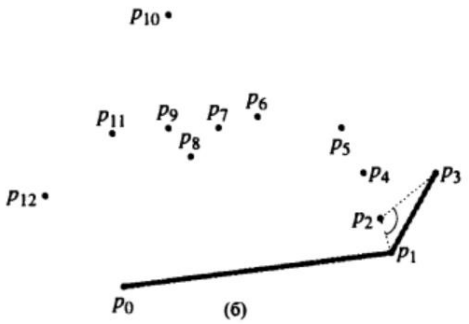
Сканирование по Грэхему

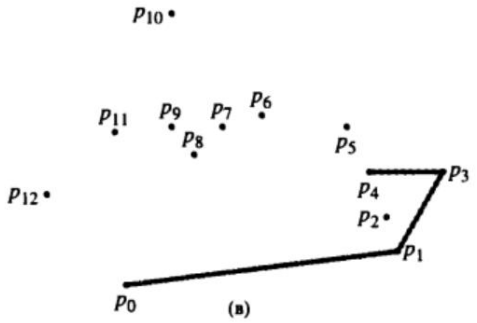
Graham-Scan(Q)
1 Пусть p0 - точка Q с минимальной координатой y, или крайняя слева из таких точек при наличии совпадений
2 Пусть (p1, p2, pm) - остальные точки Q, отсортированные в порядке возрастания полярного угла, измеряемого против часовой стрелки относительно p0 (если полярные углы нескольких точек совпадают, то из множества удаляются все эти точки, кроме одной, самой дальней от точки p0)
3 if m < 2
4 return "выпуклая оболочка пуста"
5 else пусть S - пустой стек
6 Push(p0, S)
7 Push(p1, S)
8 Push(p2, S)
9 for i = 3 to m
10 while угол, образованный точками Next-To-Top(S),
Top(S) и p, не образует поворот влево
11 Pop(S)
12 Push(p1, S)
13 return S

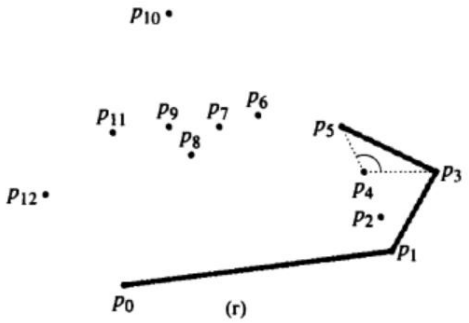
Код пример 20.5

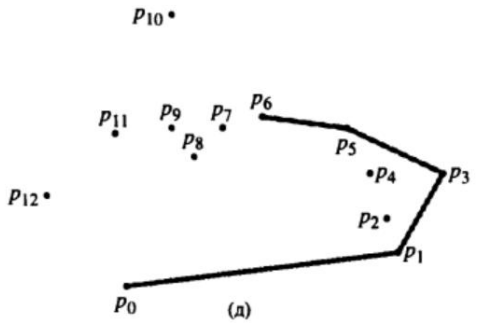
Сканирование по Грэхему

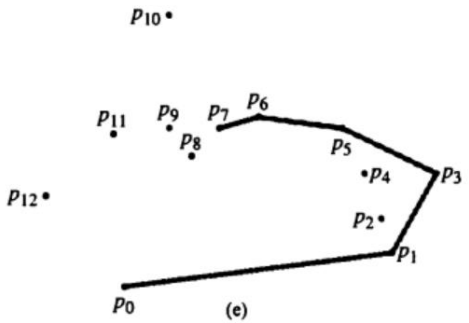


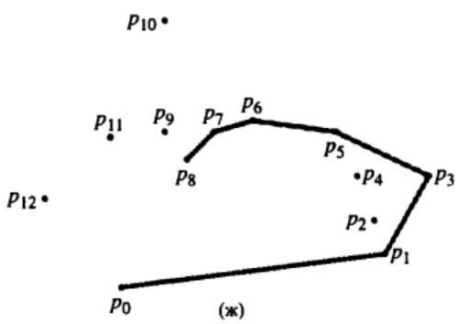


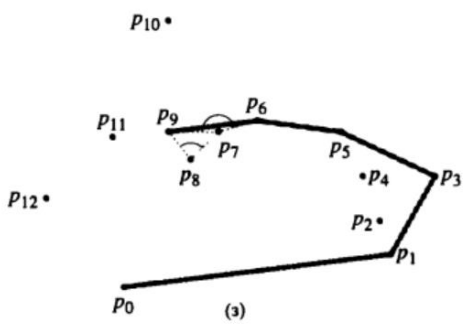


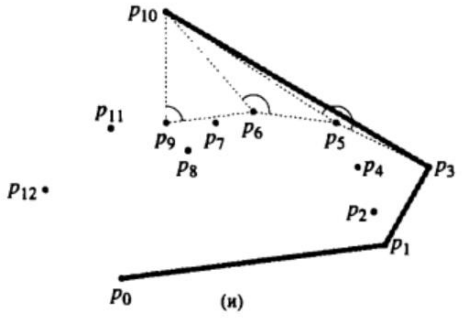


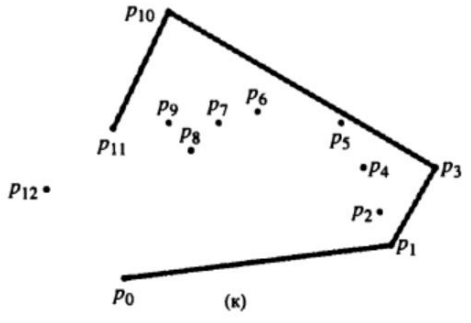


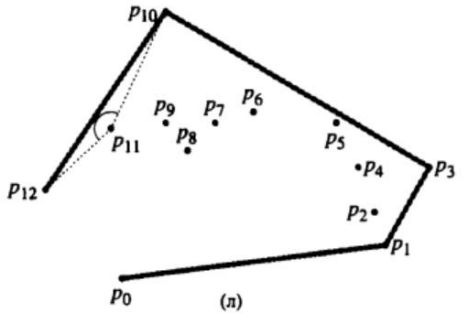


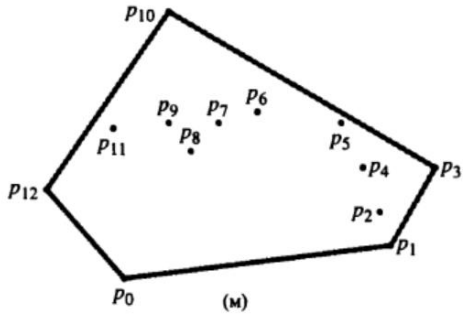






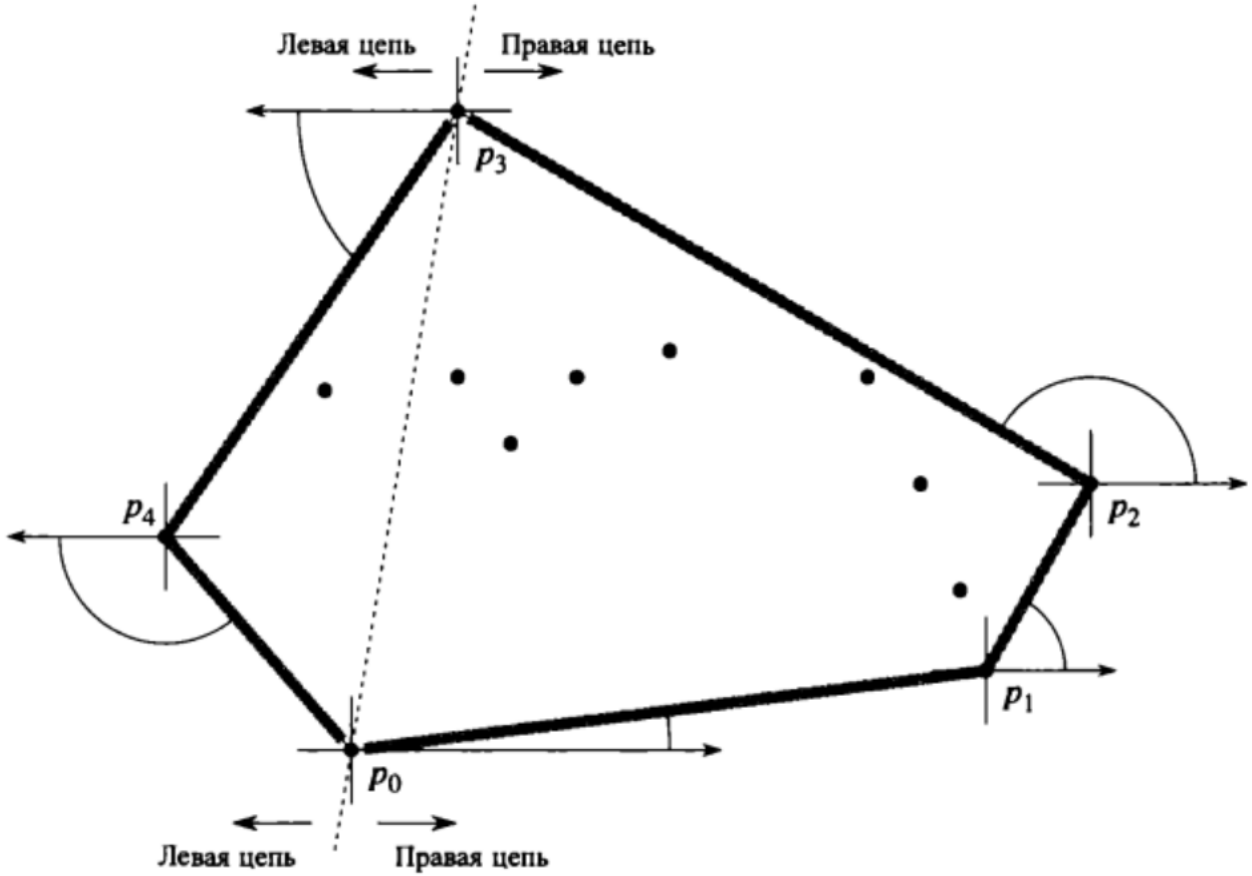






Обход по Джарвису

При построении выпуклой оболочки множества точек Q путем обхода по Джарвису (Jarvis's march) используется метод, известный как упаковка пакета (package wrapping) или упаковка подарка (gift wrapping). Время выполнения алгоритма равно O(nh), где h - количество вершин CH(Q). В том случае, когда h равно O(lgn), обход по Джарвису работает быстрее, чем сканирование по Грэхему.



Обход по Джарвису. В качестве первой вершины выбирается наинизшая точка p_0. Следующая вершина, p_1, имеет наименьший полярный угол по отношению к p_0 среди всех точек.

Поиск пары ближайших точек

Теперь рассмотрим задачу о поиске в множестве Q, состоящем из n ≥ 2 точек, пары точек, ближайших одна к другой. Термин "ближайший" относится к обычному евклидову расстоянию: расстояние между точками $p_1 = (x_1, y_1)$ и $p_2 = (x_2, y_2)$ равно $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Две точки множества Q могут совпадать; в этом случае расстояние между ними равно нулю. Эта задача находит применение, например, в системах управления движением транспорта. В системе управления воздушным или морским транспортом может понадобиться узнать, какие из двух транспортных средств находятся друг к другу ближе всего, чтобы предотвратить возможное столкновение между ними.

В алгоритме, работающем "в лоб", просто перебираются все $\binom{n}{2} = \Theta(n^2)$ пар точек. В данном разделе описывается решение этой задачи с помощью алгоритма разбиения ("разделяй и властвуй"). Время решения этим методом определяется знакомым рекуррентным соотношением $T(n) = 2T(n/2) + O(n)$. Таким образом, время работы этого алгоритма равно $O(n \lg n)$.

Алгоритм декомпозиции

В каждом рекурсивном вызове описываемого алгоритма в качестве входных данных используются подмножество $P \subseteq Q$ и массивы X и Y, в каждом из которых содержатся все точки входного подмножества P. Точки в массиве X отсортированы в порядке возрастания их координаты x. Аналогично массив Y отсортирован в порядке возрастания координаты y. Заметим, что достигнуть границы O(nlgn) не удастся, если сортировка будет выполняться в каждом рекурсивном вызове; если бы мы поступали таким образом, то время работы такого метода определялось бы рекуррентным соотношением $T(n) = 2T(n/2) + O(nlgn)$, решение которого равно $T(n) = O(nlg^2n)$. Немного позже станет понятно, как с помощью "предварительной сортировки" поддерживать свойство отсортированности, не прибегая к процедуре сортировки в каждом рекурсивном вызове.

В рекурсивном вызове со входными данными P, X и Y сначала проверяется, выполняется ли условие $|P| \leq 3$. Если оно справедливо, то в вызове просто применяется упомянутый выше метод решения "в лоб": сравниваются между собой все (1) пар точек и возвращается пара точек, расположенных друг к другу ближе других. Если же $|P| > 3$, то выполняется описанный ниже рекурсивный вызов в соответствии с парадигмой "разделяй и властвуй".

Алгоритм декомпозиции

Комбинирование. Ближайшая пара либо находится друг от друга на расстоянии δ, найденном в одном из рекурсивных вызовов, либо образована точками, одна из которых принадлежит множеству P_L, а вторая - множеству P_R. В алгоритме определяется, существует ли пара таких точек, расстояние между которыми меньше δ. Заметим, что если существует такая "пограничная" пара точек, расстояние между которыми меньше δ, то обе они не могут находиться от прямой l дальше, чем на расстоянии δ. Таким образом, как видно из рис. 3.11, (а), обе эти точки должны лежать в пределах вертикальной полосы шириной 2δ, в центре которой находится прямая l. Для поиска такой пары (если она существует) в алгоритме выполняются описанные ниже действия.

1. Создается массив Y' путем удаления из массива Y всех точек, не попадающих в полосу шириной 2δ. Как и в массиве Y, в массиве Y' точки отсортированы в порядке возрастания координаты y.
2. Для каждой точки p из массива Y' алгоритм пытается найти в этом же массиве точки, которые находятся в δ-окрестности точки p. Как мы вскоре увидим, достаточно рассмотреть лишь 7 точек, расположенных в массиве Y' после точки p. В алгоритме вычисляется расстояние от точки p до каждой из этих семи точек, и отслеживается наименьшее расстояние между точками δ', вычисленное по всем парам точек в массиве Y'.
3. Если δ' < δ, то в вертикальной полосе действительно содержится пара точек, которые находятся одна к другой ближе, чем те, которые были найдены в результате рекурсивных вызовов. В таком случае возвращаются эта пара точек и соответствующее ей расстояние δ'. В противном случае возвращаются пара ближайших точек и расстояние между ними δ, найденные в результате рекурсивных вызовов.

Реализация и время работы алгоритма

время работы представленного алгоритма описывалось рекуррентным соотношением T(n) = 2T(n/2) + O(n), где T(n) - время работы алгоритма для n точек.

```
void SplitArray(Point *Y, int y_size, int x_median, Point **YL,
int *yl_size, Point **YR, int *yr_size) {
    *YL = (Point *)malloc(y_size * sizeof(Point));
    *YR = (Point *)malloc(y_size * sizeof(Point));
    *yl_size = 0;
    *yr_size = 0;
    for (int i = 0; i < y_size; i++) {
        if (is_in_PL(Y[i], x_median))
            (*YL)[(*yl_size)++] = Y[i];
        else
            (*YR)[(*yr_size)++] = Y[i];
    }
}
```

следует выполнить их предварительную сортировку (presorting), которая осуществляется один раз перед первым рекурсивным вызовом.