Лекция №5

Алгоритмы

Продвинутый курс Си

План курса

- Вводный урок
- Структуры. Динамические типы
- Библиотеки языка С
- Оптимизация кода
- Алгоритмы
- Компиляция и компиляторы
- Динамические структуры данных
- Курсовая работа

Маршрут

Алгоритмы

- Разберём, как принято оценивать сложность алгоритма
- Как побитовые операции используются в алгоритмах шифрования
- Изучим некоторые алгоритмы и оценим
  их производительность

EASY EMBEDDED

Оценка сложности алгоритмов

Оценка сложности алгоритмов

Существует два ресурса, которые необходимо оценить при составлении алгоритма:

- Сложность по памяти
- Сложность по процессорному времени

Оба параметра зависят от размера входных данных, при этом точное время обычно никого не интересует: оно зависит от процессора, размера и типа данных, языка программирования и других параметров.

Важна лишь асимптотическая сложность, т. е. сложность при стремлении размера входных данных к бесконечности.

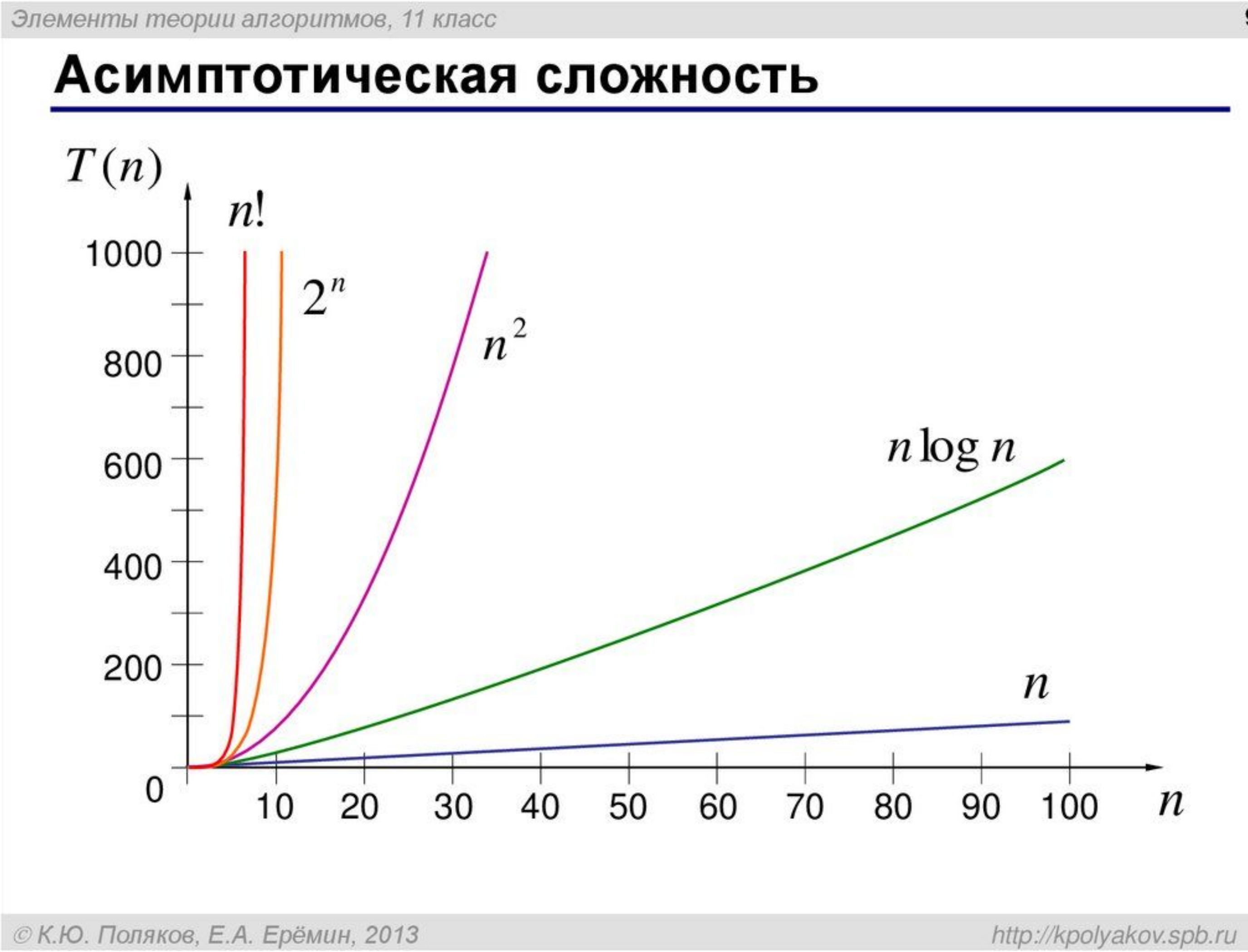
Оценка сложности алгоритмов

Для формализации оценки сложности алгоритма используется О-нотация, причём для простоты понимания и наглядности в неё обычно включают только существенно значимый параметр.

Например, есть алгоритм, который выполняется за $10n^5 + 2n$ итераций.

Как видно из выражения, на скорость гораздо больше будет влиять $10n^5$, чем $2n$.

Поэтому сложность такого алгоритма будет $O(n^5)$, т. е. $2n$ и коэффициент при n^5 будут отброшены.



O(1) сложность

Такой алгоритм не зависит от данных. Например, вернуть элемент массива, зная его индекс.

```
int arr[n];
a[0] = 1;
```

O(n) сложность

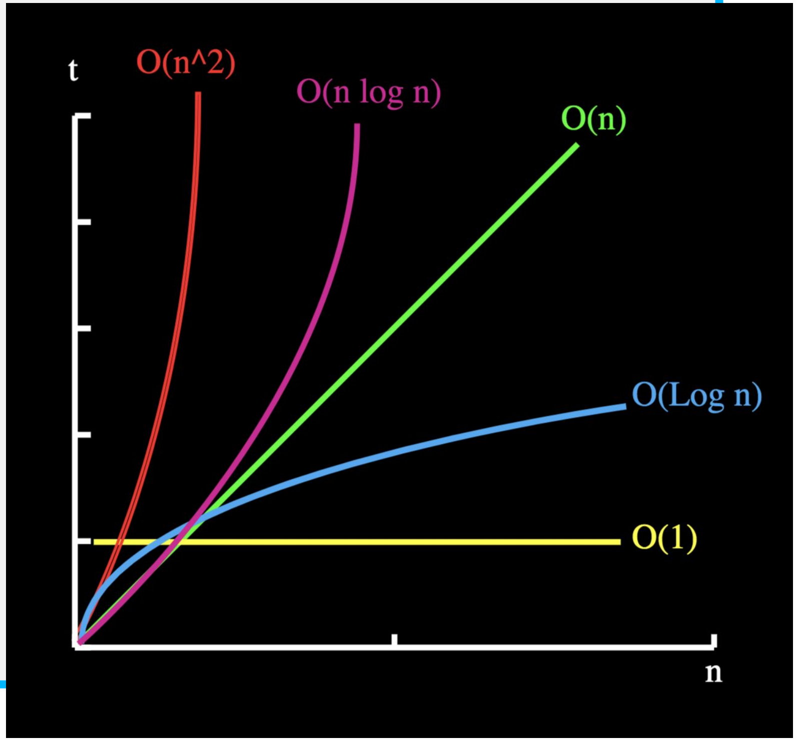
В таком алгоритме необходимо перебрать все n элементов в самом худшем случае. Например, алгоритм поиска максимума в массиве из n элементов:

```
int arr[n];
size_t imax=0;
for(size_t i=1; i<n; i++) {
    if( a[i]>a[imax] )
        imax=i;
}
```

O(log$_2$ n) сложность

Такую сложность имеет, например, алгоритм бинарного поиска. На каждом шаге количество входных данных уменьшается в два раза.

```
int arr[n] = {10,20,30,40,50}; // отсортированный массив
int findme = 20;  // то что будем искать в массиве
size_t ifind=0, start=0, end=n-1, middle;
while(1) {
    middle = (start+end)/2;
    if( arr[middle] == findme ) {
        ifind=middle;
        break;
    } else if( arr[middle]<findme )
        start = middle;
    else
        end = middle;
}
```

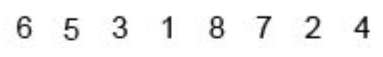


O(n log n) сложность

Подобную сложность имеют такие хорошие алгоритмы сортировки, как: сортировка слиянием, быстрая сортировка. Простой пример:

```
for(int i = 0; i < n; i++) //этот цикл выполнится n раз - O(n)
{
    for(int j = n; j > 0; j/=2) //этот цикл выполнится O(log n) раз
    {
        ...
    }
}
```

[Сортировка слиянием — Википедия]

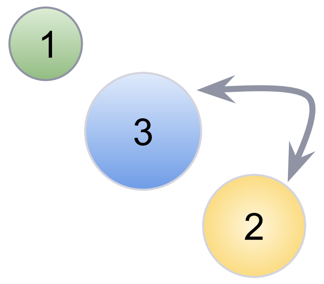


$O(n^2)$ сложность

В таких алгоритмах обычно два вложенных цикла, каждый из которых выполняет n раз. Пример такого алгоритма — более медленные сортировка пузырьком или сортировка вставками.

Еще один простой пример — поиск совпадающих элементов в массиве.

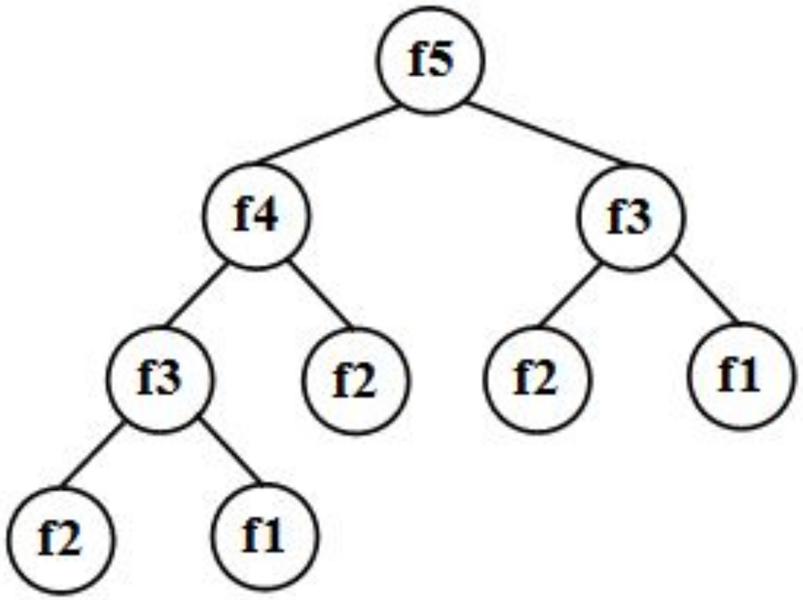
```
int arr[n] = {10,20,30,40,50,60,70,80,99,99};
_Bool hasduplicate = false;
for(size_t i=0; i<n ; i++)
    for(size_t j=0; j<n ; j++)
        if(i!=j && arr[i]==arr[j]) {
            hasduplicate = true;
        }
```



$O(2^n)$ сложность

Такую сложность имеет рекурсивная реализация для вычисления чисел Фибоначчи. Производительность функции удваивается для каждого элемента. Для каждого числа происходит два вызова себя, пока число не станет равно единице.

```
int fibonacci(int number) {
    if (number <= 1) {
        return number;
    } else {
        return fibonacci(number - 1) +
fibonacci(number - 2);
    }
}
```



XOR шифрование

XOR шифрование

В криптографии простой шифр XOR — это алгоритм шифрования, который работает в соответствии с принципами операции:

- A ^ 0 = A
- A ^ A = 0
- A ^ B = B ^ A
- (A ^ B) ^ C = A ^ (B ^ C)

Пример XOR шифрования

```c
#include<stdio.h>
#include<string.h>

void encryptDecrypt(char inpString[], char key[]) {
    size_t len = strlen(inpString);
    size_t key_len = strlen(key);
    for (size_t i = 0; i < len; i++) {
        inpString[i] = inpString[i] ^ key[i%key_len];
    }
}
```

Пример XOR шифрования

```c
int main() {
    char sampleString[] = "Hello world";
    printf("Encrypted String: ");
    encryptDecrypt(sampleString, "PASSWORD");
    printf("%s\n", sampleString);

    printf("Decrypted String: ");
    encryptDecrypt(sampleString, "PASSWORD");
    printf("%s\n", sampleString);
    return 0;
}
```

```
Encrypted String: $??8o%+"-7
Decrypted String: Hello world


HASSWORD
```

Пример XOR шифрования

Что плохо в этом примере? Если символ в слове key совпадает с символом в кодируемом слове, то в строку будет занесено число 0 — признак конца строки.

```
...
    encryptDecrypt(sampleString, "HASSWORD");
...
Как исправить?
struct String {
   int   len;
   char* str;
};

void encryptDecrypt(struct String* inpString,struct String* key) {
    for (size_t i = 0; i < inpString->len; i++)
        inpString->str[i] = inpString->str[i] ^ key->str[i%key->len];
}
```

Encrypted String:
Decrypted String:

Бинарное возведение в степень

Бинарное возведение в степень

Двоичное возведение в степень — это приём, позволяющий возводить любое число в n-ую степень за O(log n) умножений (вместо n умножений при обычном подходе).

Более того, описываемый здесь приём применим к любой ассоциативной операции, а не только к умножению чисел. Операция называется ассоциативной, если для любых a, b, c выполняется:

$$a * (b * c) = (a * b) * c$$

Алгоритм основан на том, что для любого числа a и чётного числа n выполнимо очевидное тождество:

$$a^n = (a^{n/2})^2 = a^{n/2} * a^{n/2}$$

Если степень n нечётна, то перейдём к степени n-1, которая будет уже чётной:

$$a^n = a^{n-1} * a$$

Например: $5^7 = 5*5*5*5*5*5*5 = 5^6*5 = 5^2*5^2*5^2*5$ — 3 умножения

Пример бинарного возведения в степень

Получаем рекуррентную формулу: если степень чётная, то переходим к n/2, а иначе — к n-1. Данный алгоритм будет работать за O(log n), вместо n итераций.

```
int binpow (int n, int pow) {
    if (pow == 0)
        return 1;
    if ( (pow & 1) == 1) // нечетная степень
        return binpow (n, pow-1) * n;
    else { // четная степень
        int b = binpow (n, pow>>1);
        return b * b;
    }
}
```

EASY EMBEDDED

Бинарное возведение в степень и Фибоначчи

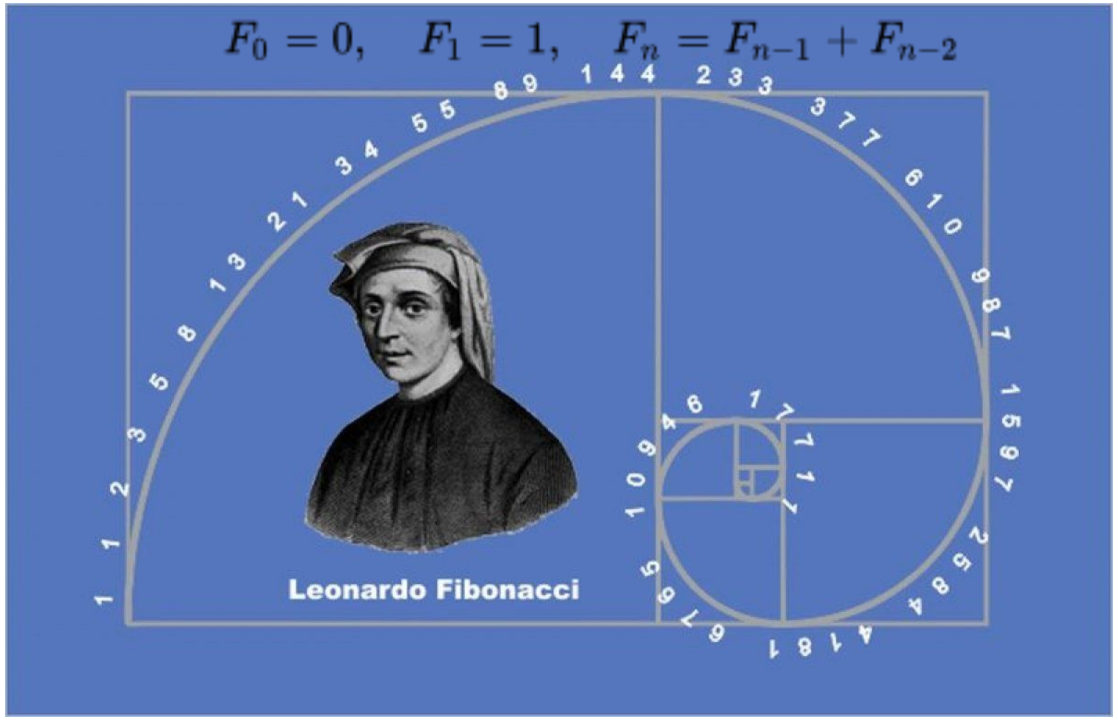
Бинарное возведение в степень и Фибоначчи

Числа Фибоначчи можно получать возведением в степень матрицы:

```
|0 1|n        |F(n-1) F(n)   |
|1 1|      =  |F(n)   F(n+1)|
```

Умножив матрицу из чисел Фибоначчи на матрицу с тремя единицами, делаем один шаг в последовательности Фибоначчи.

Далее приведен пример быстрого поиска чисел Фибоначчи.



Бинарное возведение в степень и Фибоначчи

Умножив матрицу из чисел Фибоначчи на матрицу с тремя единицами, делаем один шаг в последовательности Фибоначчи. Ниже приведен пример быстрого поиска чисел Фибоначчи.

```
/* Функция умножает матрицу a на b и результат заносит в a */
void mulMatr(int a[][SIZE], int b[][SIZE])
{
    int res[SIZE][SIZE] = {{0}};

    for (int i = 0; i < SIZE; i++)
        for (int j = 0; j < SIZE; j++)
            for (int k = 0; k < SIZE; k++) {
                res[i][j] += a[i][k] * b[k][j];
            }
    for (int i = 0; i < SIZE; i++)
        for (int j = 0; j < SIZE; j++)
            a[i][j] = res[i][j];
}
```

Бинарное возведение в степень и Фибоначчи

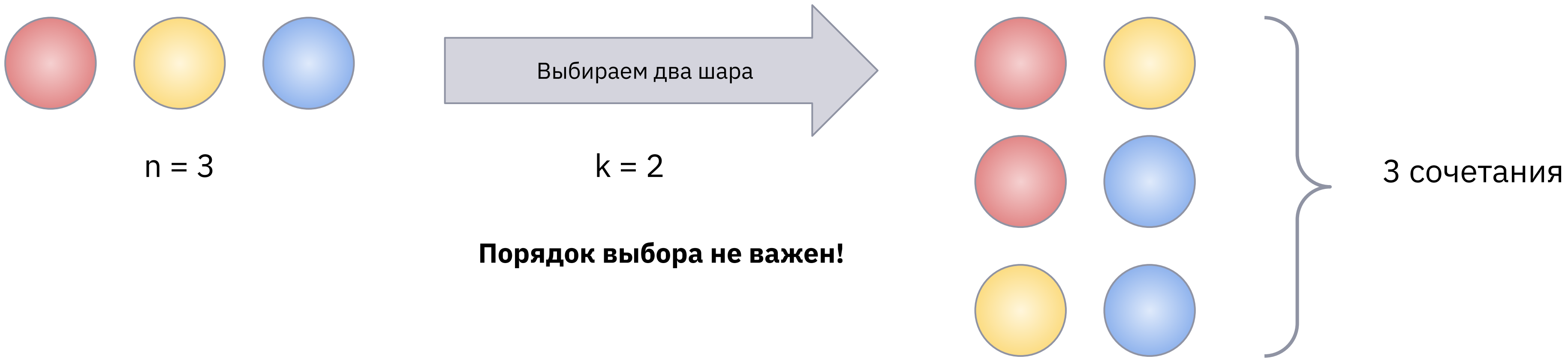
Умножив матрицу из чисел Фибоначчи на матрицу с тремя единицами, делаем один шаг в последовательности Фибоначчи. Ниже приведен пример быстрого поиска чисел Фибоначчи.

```
void printMatr(int a[][SIZE]) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++)
{
            printf("%2d",a[i][j]);
        }
        printf("\n");
    }
}
int fibMatr(int pow) {
    int t[2][2] = {{1,0},{0,1}};
    int p[2][2] = {{0,1},{1,1}};
```

```
    while(pow) {
        if( pow%2 )
            mulMatr(t,p);
        mulMatr(p,p);
        pow >>= 1;
    }
    return t[1][0];
}
int main() {
    fibMatr(10); // 55
    return 0;
}
```

Биномиальные коэффициенты

Число сочетаний без повторений



Порядок выбора не важен!

$$C_n^k = \frac{n!}{k! \cdot (n-k)!}$$

$$C_3^2 = \frac{3!}{2! \cdot (3-2)!} = \frac{6}{2} = 3$$

Источник: https://pismoref.ru/wp-content/uploads/e/b/0/eb0393644a8ae0444ffebf58351bc0c8.jpg

Биномиальные коэффициенты

C_n^k — это количество способов выбрать набор из k предметов из n различных предметов без учёта порядка расположения этих элементов.

Например, есть три объекта {1,2,3} (n=3), составляем сочетания по 2 (k=2) объекта в каждом.

Тогда выборки {1,2} и {2,1} — это одно и то же сочетание (так как комбинации отличаются лишь порядком).

А всего различных сочетаний из 3 объектов по 2 будет три: {1,2}, {1,3}, {2,3}.

Например, число байт, в которых ровно 3 единицы — это число C_n^k (число способов выбрать три бита, в которых будут стоять единицы, из восьми бит байта). Формула для вычисления выглядит так:

$$C_n^k = \frac{n!}{k!(n-k)!}$$

Биномиальные коэффициенты

Если записать данный алгоритм в явном виде, то такая функция будет работать долго, за счёт повторного вычисления факториалов. Также она может вызвать переполнение, даже если ответ помещается в какой-нибудь тип данных, вычисление промежуточных факториалов может не поместится в данный тип и привести к ошибке.

```
int cnk(int n, int k) {
    int res = 1;
    for (int i=n-k+1; i<=n; ++i)
        res *= i;
    for (int i=2; i<=k; ++i)
        res /= i;
    return res;
}
```

```
Input: 3 2
output: 3

Input: 30 10
output: -108
```

Реализация с использованием double

Рассмотрим реализацию с использованием вспомогательной вещественной переменной типа double.

1. Можно сократить n! на k! и вычислять только произведение: от i=n-k+1 до i<=n.

2. Можно заменить дробь на произведение k дробей, каждая из которых является вещественной: (n-k+i) / i

$$C_n^k = \frac{n!}{k!(n-k)!}$$

Получим такую реализацию:

```
int cnk2(int n, int k) {
    double res = 1;
    for (int i=1; i<=k; ++i)
        res = res * (n-k+i) / i;
    return (int) (res + 0.01);
}
```

```
Input: 3 2
output: 3

Input: 30 10
output: 30045015
```

Биномиальные коэффициенты

Также из первой формулы для вычисления можно вывести рекуррентную формулу вида:

$$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$$

С использованием рекуррентного соотношения можно построить треугольник Паскаля, и уже из него брать результат.

Преимущество этого метода в том, что промежуточные результаты никогда не превосходят ответа, и для вычисления каждого нового элемента таблицы надо всего лишь одно сложение.

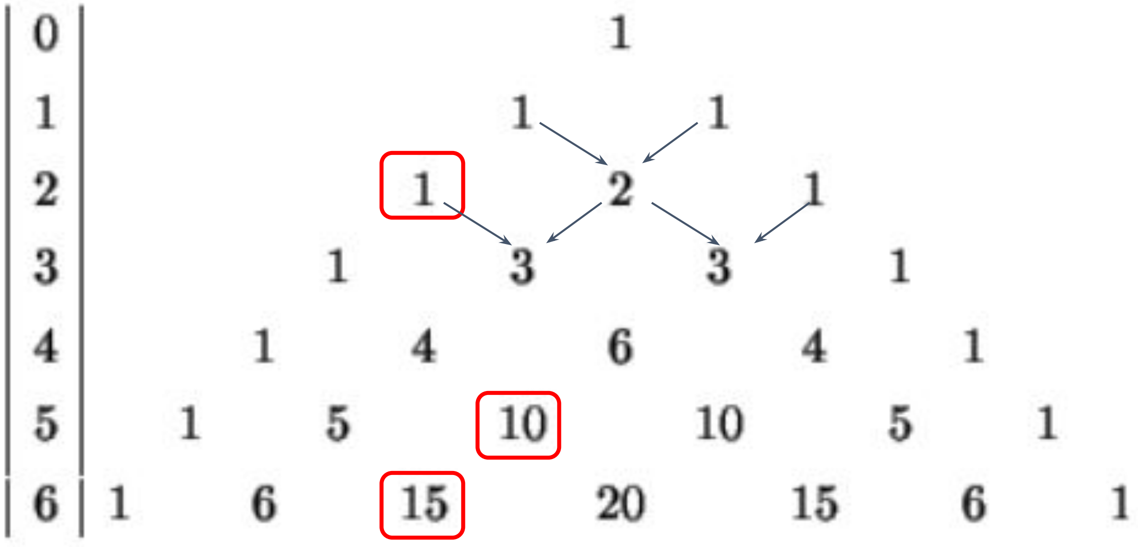
Недостатком является медленная работа для больших N и K.

Если таблица на самом деле не нужна, а нужно единственное значение, то для вычисления C_n^k понадобится строить треугольник для всех элементов.

Биномиальные коэффициенты

Это называется треугольником Паскаля, число номер k+1 в n-й строчке называется биномиальным коэффициентом C_n^k.

Например, $C_2^0=1$, $C_6^2=15$, $C_5^2=10$.



Вычисление n-ой строки треугольника Паскаля

```
#include <stdio.h>
#define N 1000
int main () {
    int c[N] = {0};
    int n, i, j;
    scanf ("%d", &n);

    c[0] = 1;
    for (j = 1; j <= n; j++)
        for (i = j; i >= 1; i--)
            c[i] = c[i - 1] + c[i];
    for (i = 0; i <= n; i++)
        printf ("%d ", c[i]);
    return 0;
}
```

```
5
1 5 10 10 5 1
```

Реализация функции вычисления биномиального коэффициента

```
int cnk3(int n, int k) {
    const int maxn = n;
    int C[maxn+1][maxn+1];
    for (int i=0; i<=maxn; ++i) {
        C[i][0] = C[i][i] = 1;
        for (int j=1; j<i; ++j)
            C[i][j] = C[i-1][j-1] +
C[i-1][j];
    }
    return C[n][k];
}
```

```
Input: 3 2
output: 3

Input: 30 10
output: 30045015
```

Табличный метод

В некоторых ситуациях выгодно заранее посчитать значения всех факториалов. Это нужно для того, чтобы впоследствии считать любой необходимый биномиальный коэффициент, производя лишь два деления.

```
int fact[]={1,1,2,6,24,120,720,5040,40320,
...};
int factorial(int n) {
    return fact[n];
}
int cnk4(int n, int k) {
    int res = factorial(n);
    res /= factorial(n-k);
    res /= factorial(k);
    return res;
}
```

```
Input: 3 2
output: 3

Input: 8 3
output: 56
```

EASY EMBEDDED

Обратная польская форма

Формат записи

Форма записи математических и логических выражений, в которой операнды расположены перед знаками операций. Рассмотрим пример такой записи:

1 2 + 4 × 3 +

В обычной инфиксной записи выражение выглядит так:

(1+2) * 4+3

Пример вычисления

Вычислить выражение 1 2 + 4 × 3 +

Шаг	Оставшаяся цепочка	Стек
1	1 2 + 4 × 3 +	1
2	2 + 4 × 3 +	1 2
3	+ 4 × 3 +	3
4	4 × 3 +	3 4
5	× 3 +	12
6	3 +	12 3
7	+	15

Обратная польская форма

Требуется вычислить его значение за O(n), где n — длина строки. Для вычисления выражений в обратной польской нотации удобно использовать стек. Порядок действий такой:

1. Обработка входного символа
2. Если на вход подано число, оно помещается на вершину стека.
3. Если на вход подан знак операции, то со стека снимаются два числа и над ними выполняется соответствующая операция. Результат выполненной операции кладётся обратно на вершину стека.
4. Если входной набор символов обработан не полностью, перейти к шагу 1.
5. После полной обработки входного набора символов, результат вычисления выражения лежит на вершине стека в качестве единственного числа.

Пример обратной польской записи

В этом примере используется функционал scanf для перевода строки из цифр в вещественное число. За работу с основным стеком отвечают функции push(number), pop().

```
#include <stdio.h>
#include "stack.c"
#define false 0
#define true 1

int main(void){
    int i=0, isMinus=false;
    int ret;
    datatype number; // double
описан в stack.c
    char c;
    printf("Input inverse string:
");
```

```
    while(
(ret=scanf("%lf",&number))!=EOF ) {
        if(ret==1) {
            push(number);
        } else {
            if(scanf("%c",&c)==EOF)
                break;
            operate(c);
        }
    }
    printf("answer = %lf\n",pop());
    return 0;
}
```

Стек реализован как обычный массив

```
#include "stack.h"
#define MAX_STACK_SIZE 255

datatype st[MAX_STACK_SIZE]; // массив -
стек
int pst=0; // заполненность  стека
void push(datatype v){ // используется для
вычислений
    st[pst++]=v;
}
datatype pop(){
    if(pst<=0) {
        fprintf(stderr, "Error. Stack
underflow");
        return 1;
    } else if(pst>MAX_STACK_SIZE) {
        fprintf(stderr, "Error. Stack
overflow");
        return 1;
    }
    return st[--pst];
}
int isEmpty(){ // определяет пустой ли стек
st
    return (pst<=0);
}
void operate(char c){ // вычисляем два
верхних значения на стеке st
    datatype arg1=pop(),arg2=pop();
    if (c=='+') push(arg1+arg2);
    else if (c=='-') push(arg1-arg2);
    else if (c=='*') push(arg1*arg2);
    else if (c=='/') push(arg2/arg1);
}
int isDigit(char c){ // проверяем является ли
символ цифрой
    return ((c>='0')&&(c<='9'));
}
```

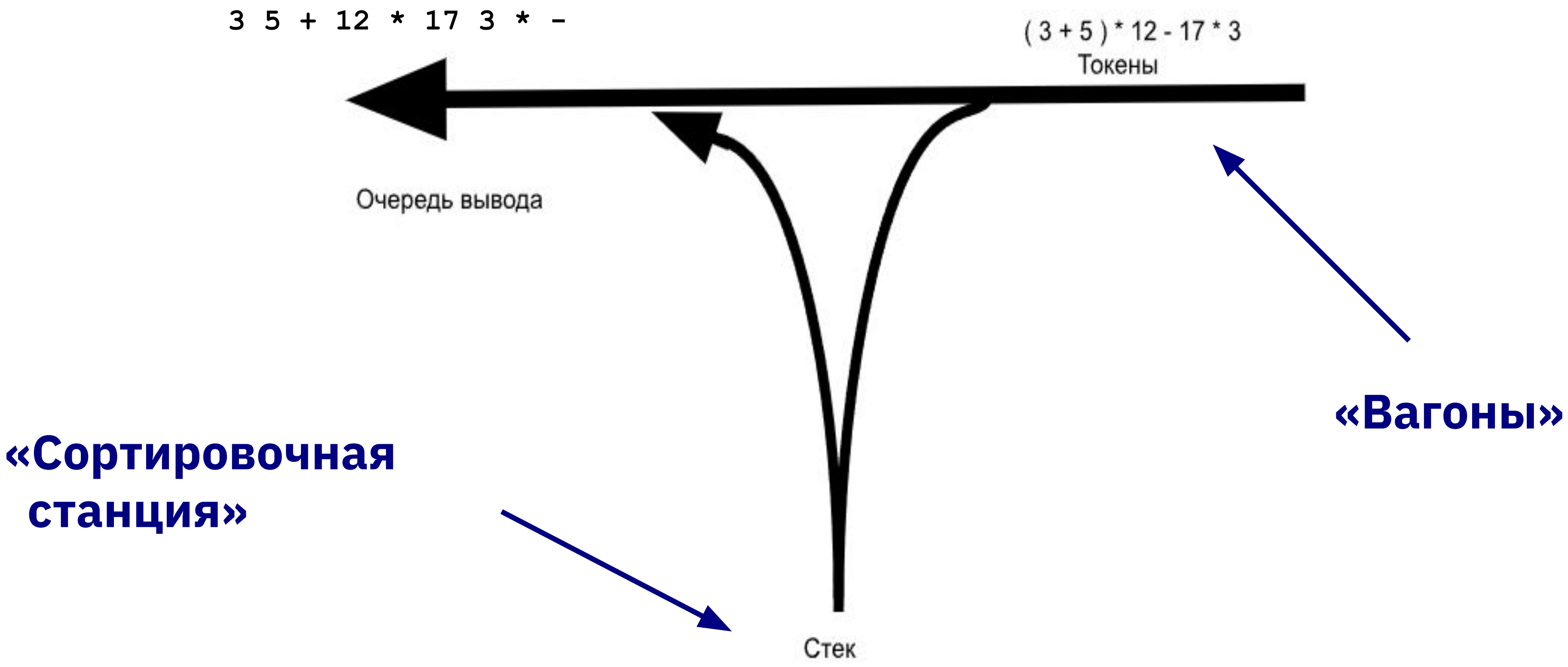
Инфиксная нотация

Инфиксная нотация

Алгоритм перевода из инфиксной записи в обратную польскую («сортировочная станция») был изобретён Э. Дейкстра.
Он использует свойство, по которому в обратной польской записи меняется только порядок и место операций в выражении, и отсортировывает операции в порядке выполнения действий.
Для реализации алгоритма нужны данные по приоритетам операций.

Алгоритм

В данном алгоритме под токеном подразумевается **число, скобка** или одна из операций (**+ - * /**).



Пример вычисления

Вычислить выражение (3+5)*12-17*3

Шаг	Оставшаяся цепочка	Стек	Вывод
1	(3+5)*12-17*3	(	
2	3+5)*12-17*3	(	3
3	+5)*12-17*3	(+	3
4	5)*12-17*3	(+	3 5
5	)*12-17*3		3 5 +
6	*12-17*3	*	3 5 +

Шаг	Цепочка	Стек	Вывод
7	12-17*3	*	3 5 + 12 *
8	-17*3	-	3 5 + 12 *
9	17*3	-	3 5 + 12 *17
10	*3	* -	3 5 + 12 *17
11	3	* -	3 5 + 12 *17 3
12			3 5 + 12 *17 3 * -

Обработка токенов

Пока не все токены обработаны:

1. Прочитать токен
2. Если токен — число, то добавить его в очередь вывода
3. Если токен — оператор op1, то:
   a. Пока присутствует на вершине стека токен оператор op2, чей приоритет выше или равен приоритету op1:
      i. Переложить op2 из стека в выходную очередь
   b. Положить op1 в стек
4. Если токен — открывающая скобка, то положить его в стек

Обработка токенов

5. Если токен — закрывающая скобка:
    a. Пока токен на вершине стека не открывающая скобка
        i. Переложить оператор из стека в выходную очередь.
        ii. **Внимание!** Если стек закончился до того, как был встречен токен-открывающая скобка, то в выражении пропущена скобка
    b. Выкинуть открывающую скобку из стека, но не добавлять в очередь вывода
6. Если больше не осталось токенов на входе:
    a. Пока есть токены операторы в стеке:
        i. Если токен оператор на вершине стека — открывающая скобка, то в выражении пропущена скобка
        ii. Переложить оператор из стека в выходную очередь
7. Конец

Пример

```
#include <stdio.h>
#include <stdlib.h>
#define BUFFER_SIZE 255
#define STACK_SIZE 255

char oper[STACK_SIZE] = {0}; // стек
для операций + - * / ( )
int oend=0; // заполненность стека
void push(char v) {
    oper[oend++] = v;
}
char pop() {
    if(oend<=0 || oend>=BUFFER_SIZE)
{
        fprintf(stderr,"Stack
overflow\n");
        exit(1);
    }
    return oper[--oend];
}
_Bool emptyStack() {
    return oend==0;
}
_Bool isOperator(char c) {
    return c=='+' || c=='-'
|| c=='*' || c=='/';
}
```

Пример

```
int priority(char c) {
    if(c=='+' || c=='-')
        return 1;
    if(c=='*' || c=='/')
        return 2;
    return 0;
}
int main(void){
    char c;
    int i=0, isMinus=false;
    int ret, pos=0;
    int number;
    char answer[BUFFER_SIZE]={0};
    printf("Input infix string: ");
```

Пример

```
while( (ret=scanf("%d",&number))!=EOF ) {
    int p=0;
    if(ret==1) {
        sprintf(answer+pos,"%d %n",number,&p);
        pos += p;
    } else {
        if(scanf("%c",&c)==EOF) break;
        if(isOperator(c)) {
            while( !emptyStack() ) {
                char top = pop();
                int p=0;
                if(priority(top)>=priority(c)) {
                    sprintf(answer+pos,"%c %n",top,&p);
                    pos += p;
                } else { // isOperator(top) == false
                    push(top);
                    break;
                }
            }
```

Пример

```
                push(c);
            } else if( c=='(' ) {
                push(c);
            } else if( c==')' ) {
                while(  (c=pop()) != '('  ) {
                    int p=0;
                    sprintf(answer+pos,"%c %n",c,&p);
                    pos += p;
                }
            }
        }
    }
    while( !emptyStack() ) {
        int p=0;
        sprintf(answer+pos,"%c %n", pop(), &p);
        pos += p;
    }
    printf("Answer: %s\n",answer);
    return 0;
}
```

Пример

```
Input infix string: ( 3 + 5 ) * 10 - 2 * 7
Answer: 3 5 + 10 * 2 7 * -
```

Хеш функция

Хеш функция

Хеш-функции – это функции, предназначенные для «сжатия» произвольного сообщения в некоторую комбинацию фиксированной длины, называемую сверткой.

Рассмотрим пример «хорошей» и «плохой» хеш функции. Необходимо сделать хеш функцию для телефонного номера.

«Плохой» вариант	«Хороший» вариант (возможно не самый)
Использовать первые три цифры.	Использовать последние три цифры.

Внимание! При неудачном алгоритме хэши разных данных могут совпадать

Хеш функции строк

Один из способов определить хэш-функцию от строки S:

$$h(S) = S[0] + S[1] * P^1 + S[2] * P^2 + S[3] * P^3 + \ldots + S[N] * P^N$$

P — некоторое простое число, которое примерно равно количеству символов во входном алфавите.

Для маленьких букв латинского алфавита возьмем ближайшее простое число **P = 31**

Пример

```c
#include <stdio.h>
#include <inttypes.h>

uint64_t getHash(char const *s) {
    const int p = 31;
    uint64_t hash = 0, p_pow = 1;
    while(*s) {
 /* отнимаем 'a' от кода буквы
 единицу прибавляем, чтобы у строки вида
'aaaaa' хэш был ненулевой */
        hash += (*s++ - 'a' + 1) * p_pow;
        p_pow *= p;
    }
    return hash;
}
```

```c
int main(void)
{
    char s[100] = {0};
    scanf("%s",s);
    printf("hash =
%llu\n", getHash(s));
    return 0;
}
```

```
Hello.
hash =
18446744072292316036
```

Быстрые алгоритмы для создания хешей

Алгоритм хеш-функции для строки, который использует побитовые операции:

- Каждый бит введённой строки должен влиять на результат полученного хеш.
- Один из простых способов сделать это — повернуть текущий результат на некоторое количество бит, а затем выполнить XOR текущего хэш-кода с текущим байтом.
- Поворот не должен быть кратен байту.

Не стоит использовать криптографические алгоритмы для создания хешей. Даже если они работают быстро, для создания хеш это всё же очень медленно.

Пример

Ещё один пример хеш-функции для строки использует побитовые операции. Важно то, что функция должна работать максимально быстро.

```
uint64_t getHash2(char const
*str)
{
    uint64_t hash = 5381;
    int32_t c;
    while (c = *str++)
        hash = ((hash<<5) +
hash) + c;
    return hash;
}
```

```
uint64_t rol(uint64_t n, size_t
shift) {
   return (n<<shift) | (n>>(64 -
shift));
}
uint64_t getHash3(char const *s){
    uint64_t result = 0x55555555;

    while (*s) {
        result ^= *s++;
        result = rol(result, 5);
    }
    return result;
}
```

Задачи

По пройденному материалу

1. Дано некоторое количество строк S[1..N], каждая длиной не более M символов. Допустим, требуется найти все повторяющиеся строки и разделить их на такие группы, чтобы в каждой из них были только одинаковые строки.
Обычной сортировкой строк мы бы получили алгоритм со сложностью O (N M log N), в то время как используя хэши, мы получим O (N M + N log N).
Алгоритм. Посчитаем хэш от каждой строки, и отсортируем строки по этому хэшу.

Пример

```
enum {TOTAL_STRINGS=10};
struct strings {
    char s[100];
    uint64_t hash;
} strs[TOTAL_STRINGS];
int cmphashes (const void *a, const void *b) {
   return ( (*(struct strings *)a).hash - (*(struct strings
*)b).hash );
}
```

Пример

```
int main(void)
{
    for(size_t i=0; i<TOTAL_STRINGS; i++) {
        scanf(" %[^\n]",strs[i].s);
        strs[i].hash = getHash(strs[i].s);
    }

    for(size_t i=0; i<TOTAL_STRINGS; i++)
        printf("%s: %llu\n",strs[i].s, strs[i].hash);
    qsort(strs, TOTAL_STRINGS, sizeof(struct strings), cmphashes);

    for(size_t i=0; i<TOTAL_STRINGS; i++)
        printf("%s: %llu\n",strs[i].s, strs[i].hash);

    return 0;
}
```

Входные данные

```
hello world
hello world
hello
world
hello
world
world
hello world
abcd
abcd
```

Скользящая ХЕШ функция

Скользящая ХЕШ функция

Используем в качестве примера шаблон текста «135» и текст «2135». Сначала вычисляем хеш шаблона 135 по следующему алгоритму:

$H(135) = 10^2*1 + 10^1*3 + 10^0*5 = 135$

Затем вычисляем хеш первых m = 3 символов текста, т. е. 213:

$H(213) = 10^2*2 + 10^1*1 + 10^0*3 = 213$

Совпадения не произошло. Теперь сдвигаем подстроку, отбросив первый символ предыдущего окна и добавив следующий. Получилось новая 135. Вычисляем для неё хеш:

$H(135) = 10^2*1 + 10^1*3 + 10^0*5 = 135$

Хеши совпадают, подстрока найдена.

Скользящая ХЕШ функция

Обратите внимание: переместив скользящую подстроку, нам пришлось вычислить весь хеш 213 и 135, что нежелательно, так как при этом мы вычислили хеш целых чисел, которые уже были в предыдущем действии.

Скользящая хеш-функция может запросто устранить эти дополнительные вычисления, убрав из подсчёта нового значения хеша первый символ предыдущего и добавив новый символ.

Мы можем избавиться от хеш-значения этого убранного из подсчёта символа, умножить полученное значение на основание, чтобы восстановить правильный порядок степени предыдущих нетронутых символов и добавить значение нового символа.

Скользящая ХЕШ функция

Так, мы можем вычислить хеш новой подстроки по формуле:

$H = (H_{previous} + C_{previous} * pow^{strlen-1})pow + C_{new}$

H_p - предыдущий хеш

$C_{prevoius}$ - предыдущий символ

C_{new} - новый символ

strlen - размер строк

pow - константа

Используя предыдущий пример сдвига от 213 к 135, мы можем получить новый хеш:

$H = (213-2 * 10^2)*10 + 5 = 135$

Пример

```c
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>
#include <string.h>

char s[100]="135", text[100]="21354562135";
uint64_t p=10;
uint64_t getHash(char *str, size_t len) {
    uint64_t hash=0;
    char c;
    for(size_t i=0; i<len; i++) {
        c=str[i];
        hash *= p;
        hash +=  (c-'0');
    }
    return hash;
}
```

Пример

```c
int main(void) {
    uint64_t ht[100]={0}, hs, p_pow=1;
    size_t lens=strlen(s), lent=strlen(text);
    hs = getHash(s,lens);
    printf("s hash = %lld\n",hs);
    for(size_t i=1; i<lens; i++)
        p_pow *= p;
    ht[0] = getHash(&text[0],lens);
    printf("%lld  ",ht[0]);
    for(size_t i=1; i<lent-lens+1; i++) {
        ht[i] = (ht[i-1]%p_pow)*p +
text[lens+i-1]-'0';
        printf("%lld ",ht[i]);
    }
    printf("\n");
    return 0;
}
```

```
s hash = 135
213  135 354
545 456 562
621 213 135
```

Z-функция

Z-функция

[Поиск подстроки и смежные вопросы]

Пусть дана строка s длины n.

Тогда Z-функция от этой строки — это массив длины n, i-ый элемент которого равен наибольшему числу символов, начиная с позиции i, совпадающих с первыми символами строки s.

Другими словами, z[i] — это наибольший общий префикс строки s и её i-го суффикса.

Примечание: первый элемент массива z[0] обычно считают неопределённым. Будем считать его равным нулю, это никак не влияет на работу алгоритма.

Пример

1. s[] = “aaaaa”

i	0	1	2	3	4
s[i]	a	a	a	a	a
z[i]	0	4	3	2	1

2. s[] = “aaabaab”

i	0	1	2	3	4	5	6
s[i]	a	a	a	b	a	a	b
z[i]	0	**2**	**1**	0	2	1	0

Пример

3. s[] = “abacaba”

i	0	1	2	3	4	5	6
s[i]	a	b	a	c	a	b	a
z[i]	0	0	1	0	3	0	1

4. s[] = “aaaabaa”

i	0	1	2	3	4	5	6
s[i]	a	a	a	a	b	a	a
z[i]	0	**3**	**2**	1	0	2	1

Пример

Тривиальная реализация со сложностью $O(n^2)$ будет выглядеть так:

```
void zFunction(char *s, int z[]) {
    int n =  strlen(s);
    for (int i=1; i<n; ++i)
        while ( i+z[i] < n && s[z[i]] == s[i+z[i]])
            ++z[i];
}
```

Пример

Более эффективная реализация со сложностью O(n):

```
void zFunction2 (char *s, int z[]) {
    int n = strlen(s);
    for (int i=1, l=0, r=0; i<n; ++i) {
        if (i <= r)
            z[i] = min (r-i+1, z[i-l]);
        while (i+z[i] < n && s[z[i]] == s[i+z[i]])
            ++z[i];
        if (i+z[i]-1 > r)
            l = i,  r = i+z[i]-1;
    }
}
```

Применение z-функции на практике

Рассмотрим задачу, в которой необходимо найти все вхождения слова **p** в текст **t**.

- Образуем строку **s = p + # + t**, т.е. к образцу припишем текст через символ-разделитель (данный символ не входит в текст и не входит в слово).
- Посчитаем для полученной строки Z-функцию.
- Для любого **i** в отрезке **[0; strlen(t)-1]**
  - Если **z[i + strlen(p) + 1]** равно **strlen(p)**
  - то слово **p** входит ли в текст **t**, начиная с позиции **i**.

В итоге сложность такого алгоритма будет **O(strlen(t) + strlen(p))**.

Пример

```c
char t[SIZE] = {0}, p[SIZE] = {0};
char s[SIZE+SIZE] = {0};
int z[SIZE+SIZE] = {0};
printf("Input text: ");
scanf("%s",t);
printf("Input word: ");
scanf("%s",p);
size_t tlen = strlen(t);
size_t plen = strlen(p);
sprintf(s,"%s#%s",p,t);
zFunction2(s,z);
for(size_t i=0; i<tlen; i++)
    if(z[i+plen] == plen)
        printf("find word in 
position %zu\n",i);
printf("\n");
```

```
Input text: AAAAB
Input word: AAAB
s = AAAB#AAAAB
find word in position 1

Input text: abcdbbbca
Input word: bc
s = bc#abcdbbbca
find word in position 2
find word in position 7
```