ЛЕКЦИЯ 12
REGEX И АВТОМАТЫ

АЛГОРИТМИЗАЦИЯ И
ПРОГРАММИРОВАНИЕ

ЛЕКТОР ФУРМАВНИН С.А.

АЛФАВИТЫ И СТРОКИ

Алфавит – это множество символов, например $\{a, b, c\}$

Строка – это последовательность символов, например $w = \{a, a, c, d\}$

- Для краткости можно записывать $w = aacb$. Пустая строка обозначается Λ
- Конкатенация строк: $w = aacb, z = ba, wz = aacbba, zw = baaacb$
- Степень: $w^3 = www$, $w^0 = \Lambda$

Языком над данным алфавитом называется множество строк

- Язык L_{empty} =пустое множество
- Язык L_{free} =все возможные строки алфавита (группа конкатенации)
- Язык $L_{xb} = \{b, ab, bb, aab, abb, bab, \dots\} = (a|b) * b$
- Язык $L_{xby} = \{b, ab, bb, abb, abc, bbb, bbc, \dots\} = (a|b) * b(b|c) *$

ЗАДАЧИ ДЛЯ ФОРМАЛЬНЫХ ЯЗЫКОВ

- Принадлежность: имея язык L и строчку w, определить принадлежит ли она языку
- Порождение: имея язык L, порождать все его строки последовательно
- Эквивалентность: имея язык L_1 и язык L_2, определить принадлежат ли им одинаковые элементы
- Отрицание: имея язык L_1, описать язык L_2, такой, что он содержит все строки, не принадлежащие L_1
- Для некоторых особенно простых языков задача принадлежности решается посредством конечных автоматов.

КОНЕЧНЫЕ АВТОМАТЫ

Конечный автомат – это $\{Q, S, q_0, F, \delta\}$, т.е. совокупность состояний (выделены начальное и принимающие), входного алфавита и функции переходов

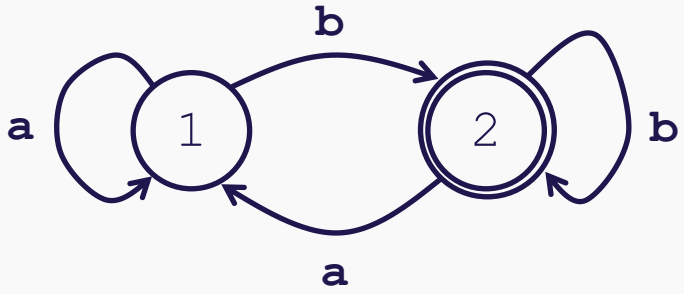
Для примера рассмотрим автомат, принимающий L_{xb}

$Q = \{1,2\}, S = \{a, b\}, q_0 = 1, F = \{2\}$

$\delta = \{\{\{1, a\}, 1\}, \{\{1, b\}, 2\}, \{\{2, a\}, 1\}, \{\{2, b\}, 2\}\}$

Исполняя шаг за шагом правильную входную строку, мы закончим в принимающем состоянии

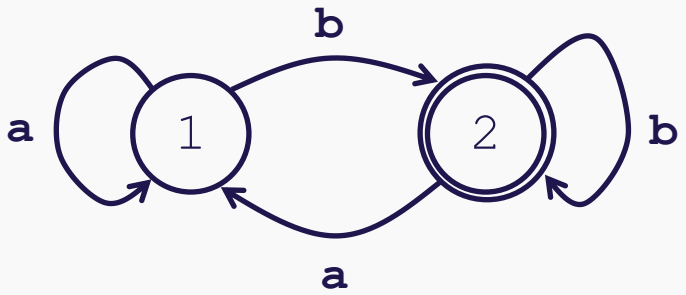
```
Input:   a a a b b a b a b
State: 1 1 1 1 2 2 1 2 1 2
```



АВТОМАТ НА ЯЗЫКЕ С

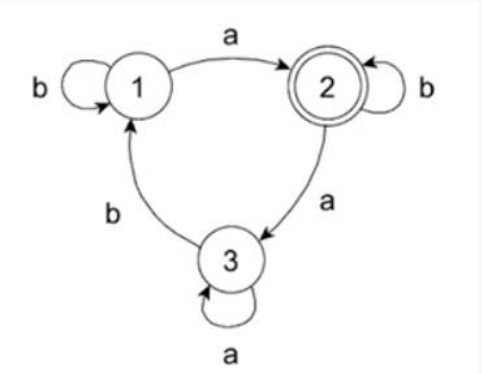
Простой автомат моделируется простой программой

```
char nextsym();
state = 1;
for (;;) {
    char c = nextsym();
    switch c: {
        case ‘a’: state = 1; break;
        case ‘b’: state = 2; break;
        default: return (state == b);
    }
}
```



БОЛЕЕ СЛОЖНЫЙ АВТОМАТ

```
char c = nextsym();
switch c: {
    case ‘a’: {
      case 1; state = 2; break;
      case 2: state = 3; break;
      case 3: state = 3; break;
    }
    break;
}
case ‘b’: {
    // …
```



$$L_{abb} = b^*ab^*(aa^*bb^*ab^*)^*$$

ОСОБЕННОСТИ SWITCH: DUFF DEVICE

Экзотическая, но не слишком пугающая идея

```
void copy(int *to, int *from, int count) {
  n = (count + 3) / 4;
  switch (count % 4) {
    case 0: do { *to++ = *from++;
    case 3:      *to++ = *from++;
    case 2:      *to++ = *from++;
    case 1:      *to++ = *from++;
            } while (--n > 0);
  }
}
```

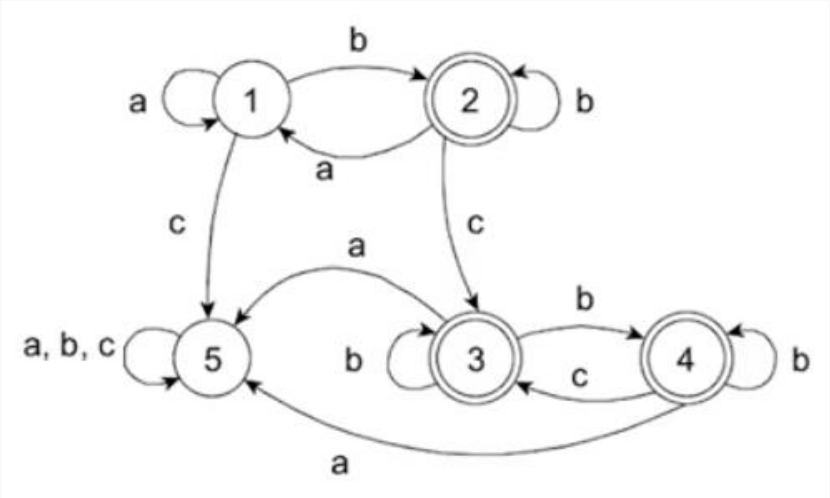
УПРАЖНЕНИЕ С АВТОМАТАМИ

Постройте автомат, который принимает язык $L_{xby} = (a|b)^* b (b|c)^*$

УПРАЖНЕНИЕ С АВТОМАТАМИ

Постройте автомат, который принимает язык $L_{xby} = (a|b)^* b (b|c)^*$
Напишите на языке C++ класс, который симулирует этот автомат.

Далее мы узнаем для каких языков вообще бывают возможны автоматы



РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

- Любой алфавитный символ означает язык из этого символа: a это $\{a\}$
- Конкатенация $L_x L_y = \{wz | w \in L_x \wedge z \in L_y\}$
- Дизъюнкция $(L_x + L_y) = \{w | w \in L_x \vee w \in L_y\}$
- Замыкание $(L_x)^* = \{\{\Lambda\}, L_x, L_x L_x, L_x L_x L_x, \ldots\}$
- Язык L_1 теперь можно описать как $a^* b^*$
- Упражнение: назовите любую строчку, принадлежащую языку $(c(a+b)^* ab)^* ca$
- Упражнение: принадлежит ли ему строчка $caabbca$? Как вы это установили?

POSIX BRE

```
a.c : abc, aac, acc
[azc] : a, z, c
[a-z] : a, b, c, …, z
[^a-c] : d, e, f, …
^abc, bcd$ : abcd
ab*c : abc, abbc, abbbc,
…
a\{3\} : aaa
a\{3,\} : aaa, aaaa, …
```

```
\(ab\)* : ab, abab, …
[[:upper:]] = [A-Z]
[[:lower:]] = [a-z]
[[:alpha:]] = [A-Za-z]
[[:digit:]] = [0-9]
[[:xdigit:]] =[0-9a-fA-F]
[[:alnum:]] = [A-Za-z0-9]
[[:word:]] = [A-Za-z0-9_]
[[:space:]] =[\t\n\r\f\v]

a[[:digit:]]b = a1b, …
[^[ABZ[:lower:]]: C, D, …
```

НЕМНОГО ПРО GREP

Утилита grep позволяет играть с регулярными выражениями в консоли и в текстовых файлах

```
$ cat test.txt

aaababbbcbcbc
aaababbbcbabc
aaaaaaabccccc
aaaacccc

$ grep -Ex “[ab]*b[bc]*” test.txt

aaababbbcbcbc
aaaaaaabccccc
```

РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ В C

Хедер не стандартный, но поддержан в POSIX-совместимых системах

```
#include <regex.h>
```

Отдельно компилируем выражение (в конце надо освободить)

```
regex_t regex;
res = regcomp(&regex, regex_source, flags);
```

Отдельно матчим строку

```
res = regexec(&regex, string_to_match, 0, NULL, 0);
```

Код возврата ноль, если строка не сматчилась, иначе REG_NOMATCH, возможны также ошибки

РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ В C++

Хедер не стандартный, но поддержан в POSIX-совместимых системах

```
#include <regex>
```

Отдельно компилируем выражение

```
std::regex regex_pattern(pattern);
```

Отдельно матчим строку

```
res = std::regex_search(str, regex_pattern);
```

`std::regex_search` возвращает `true`, если строка сматчилась и `false` – если не сматчилась

УПРАЖНЕНИЕ. ПРОСТЫЕ РЕГУЛЯРКИ

Адрес электронной почты имеет вид something@domain.extension
Где нечто может включать разделение точками и цифры, а домен и расширение – буквенные.
Примеры правильных адресов:

vasya2001@mail.ru
Tamara.Ivanovna@rambler.ru
3apa3a@vasyan.org

Ваша задача – отличить правильные адреса от неправильных. Для правильных адресов выводите на stdout число 1, для неправильных – 0. Напишите функцию, которая реализует проверку правильноести адреса электронной почты

МАТЧИМ КУСКИ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

Мы можем не только отвечать на вопрос «есть или нет» но и вернуть то, что конкретно попало под паттерн

```
regmatches_t matches[MAX_MATCHES];

res = regexec(&regex, sz1, MAX_MATCHES, matches, 0);
// теперь у нас есть
// regex.re_nsub, matches[i].rm_so, matches[i].rm_eo
```

Это позволяет решать задачи приблизительного поиска

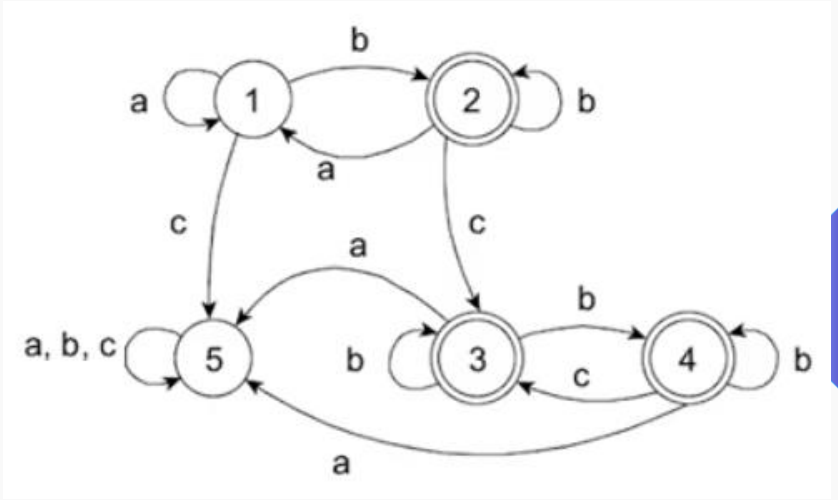
УПРАЖНЕНИЕ. ПРИМЕРНЫЙ ПОИСК

Продолжая задачу с домашней работы, переверните все подстроки по заданному регулярному выражению

Вход: `5 wo.*d 13 Hello, world!`
Выход: `Hello, dlrow!`

ОБСУЖДЕНИЕ

- Мы помним, что для довольно простых регулярных языков у нас получались довольно сложные автоматы
- Есть ли общий способ построить автомат по регулярному выражению?
- Судя по тому, что regcomp работает, как-то явно можно



$$L_{xby} = (a|b)^* b (b|c)^*$$

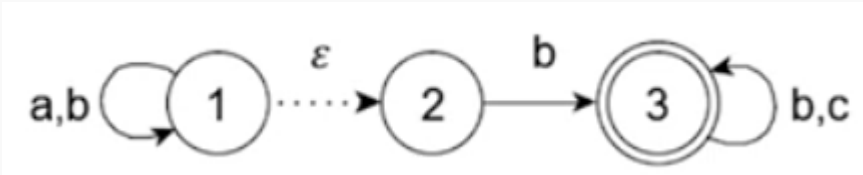
ЭПСИЛОН ПЕРЕХОДЫ

- Основная проблема сматчить выражение вроде $(a+b)^*b(b+c)^*$ - это недетерменизм в том, когда заканчивать матчинг первого замыкания

$aabbcb$

$aabbbabcc$

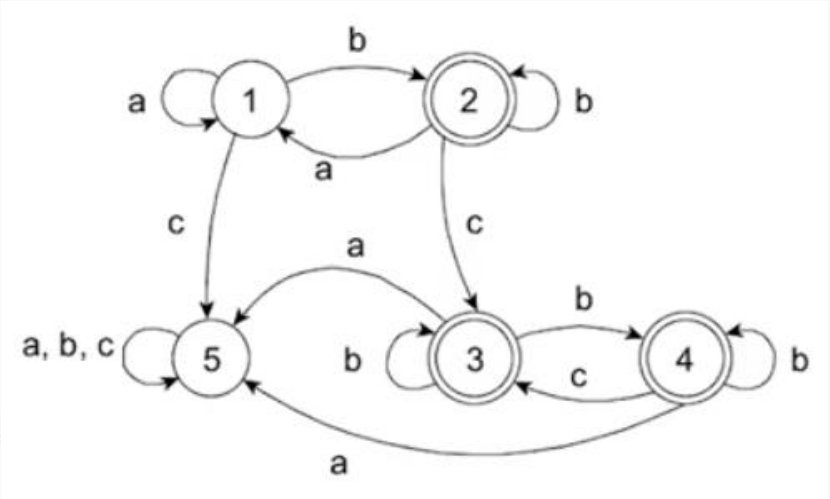
- Что если мы разрешим спонтанные (эпсилон) переходы?
- Будем считать, что автомат принимает строку, если хотя бы одна строка ведет в принимающее состояние
- Такие автоматы будем называть недетерминированными NFA в противоположность DFA



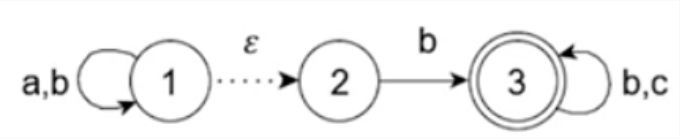
$$L_{xby} = (a|b)^*b(b|c)^*$$

ОБСУЖДЕНИЕ

- Какой автомат вам нравится больше? с точки зрения реального использования?



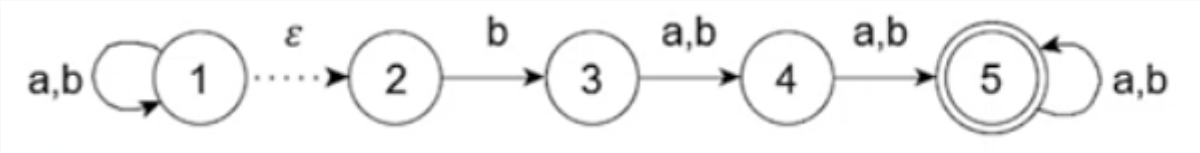

```
s a a b b b a b c c
1 3 2 3 4 4 2 3 5 5
```




```
s a a b b b a b c c
1 1 1 1 1 1 1 1 x
1 2 x
1 1 2 3 3 3 x
.....................
1 1 1 1 1 2 x
1 1 1 1 1 1 2 3 3 3
```

АЛГОРИТМ РАБИНА-СКОТТА

- Алгоритм перехода от NFA к DFA называется алгоритмом Рабина-Скотта или конструкцией подмножеств
- Увы, такой переход может привести к экспоненциальному росту числа состояний автомата
- Для примера, попробуйте применить powerset construction к следующему автомату, вы получите DFA о шестнадцати состояниях



$$L_{xbz} = (a|b)^* b(a|b)(a|b)(a|b)$$

УПРАЖНЕНИЕ В POWERSET

- На языке из символов 0 и 1, постройте DFA, распознающий множество всех строк, в которых всякая подстрока из пяти последовательных символов содержит хотя бы два 0.

```
0 1 0 0 1 0 0 1 1 0 – OK
0 1 1 0 1 1 0 0 0 0 – fail
0 1 1 0 1 0 1 1 0 1 – OK
1 0 0 1 0 1 0 0 1 0 – OK
1 0 0 1 1 1 1 0 1 0 – fail
```

ПРЕДЕЛЫ РЕГУЛЯРНОСТИ

Увы, не все языки являются регулярными

Лемма о накачке гласит, что для любого достаточно длинного слова w в регулярном языке найдется такая декомпозиция $w = xyz$, что все слова $xy^n z$ также принадлежат этому языку

Поэтому язык $a^n b^m$ регулярный: вместе с $a^{n-1}ab^m$ ему принадлежат все $a^{n-1}a^k b^m$

Но это значит, что язык $a^n b^n$ - **не регулярный**

Также не регулярен язык всевозможных регулярных выражений

ОБСУЖДЕНИЕ

Одной из интересных идей для эффективного поиска подстроки в строке является идея сделать из искомой подстроки автомат

FAILURE FUNCTION

Определим префикс-функцию, как длину максимального собственного префикса, совпадающего с максимальным собственным суффиксом

Строка: *ABCDEABCDEABCDEABCDZABCDE*

Подстрока: *ABCDZ*

Построим failure function

ABCDZ@*ABCDEABCDEABCDE**ABCDZ**ABCDE*

00000012340123401234012345123 40

ЭФФЕКТИВНОЕ ВЫЧИСЛЕНИЕ F[I]

Пусть для строки S, $f[i] = k$. Тогда:

- Если $S[i+1] = S[k+1]$, то $f[i+1] = k+1$ по определению
- Иначе если $k = 0$, то $f[i+1] = 0$
- Иначе устанавливаем $k = f[k]$ и пробуем еще раз
- Пример: $abacab$

$f[1] = k = 0; f[2] = k = 0; S[3] = S[k+1] = S[1] \Rightarrow f[3] = k = 1;$
$f[4] = f[k] = f[1] = 0; S[5] = S[k+1] \Rightarrow f[5] = k = 1;$
$S[6] = S[k+1] = S[2] \Rightarrow f[6] = k = 2;$

АЛГОРИТМ КМП

Что если мы вычислим failure function $f[i]$ только для искомой подстроки?

$W[i]$	A	B	A	C	A	B
$S[0:i]$	A	AB	ABA	ABAC	ABACA	ABACAB
$f[i]$	0	0	1	0	1	2

Далее с использованием этой табличной функции идет поиск

1	2	3																	
A	B	A	B	A	C	A	A	B	A	C	A	B	C	A	C	B	C	A	A	A
A	B	A	C	A	B														

Смотрим $f[2] == 1$, сдвигаем на $3 - f[2] == 2$

АЛГОРИТМ КМП

Что если мы вычислим failure function $f[i]$ только для искомой подстроки?

$W[i]$	A	B	A	C	A	B
$S[0:i]$	A	AB	ABA	ABAC	ABACA	ABACAB
$f[i]$	0	0	1	0	1	2

Далее с использованием этой табличной функции идет поиск

		1	2	3	4															
A	B	A	B	A	C	A	A	B	A	C	A	B	C	A	C	B	C	A	A	A
		A	B	A	C	A	B													

Смотрим $f[2] == 1$, сдвигаем на $3 - f[2] == 2$

АЛГОРИТМ КМП

Что если мы вычислим failure function $f[i]$ только для искомой подстроки?

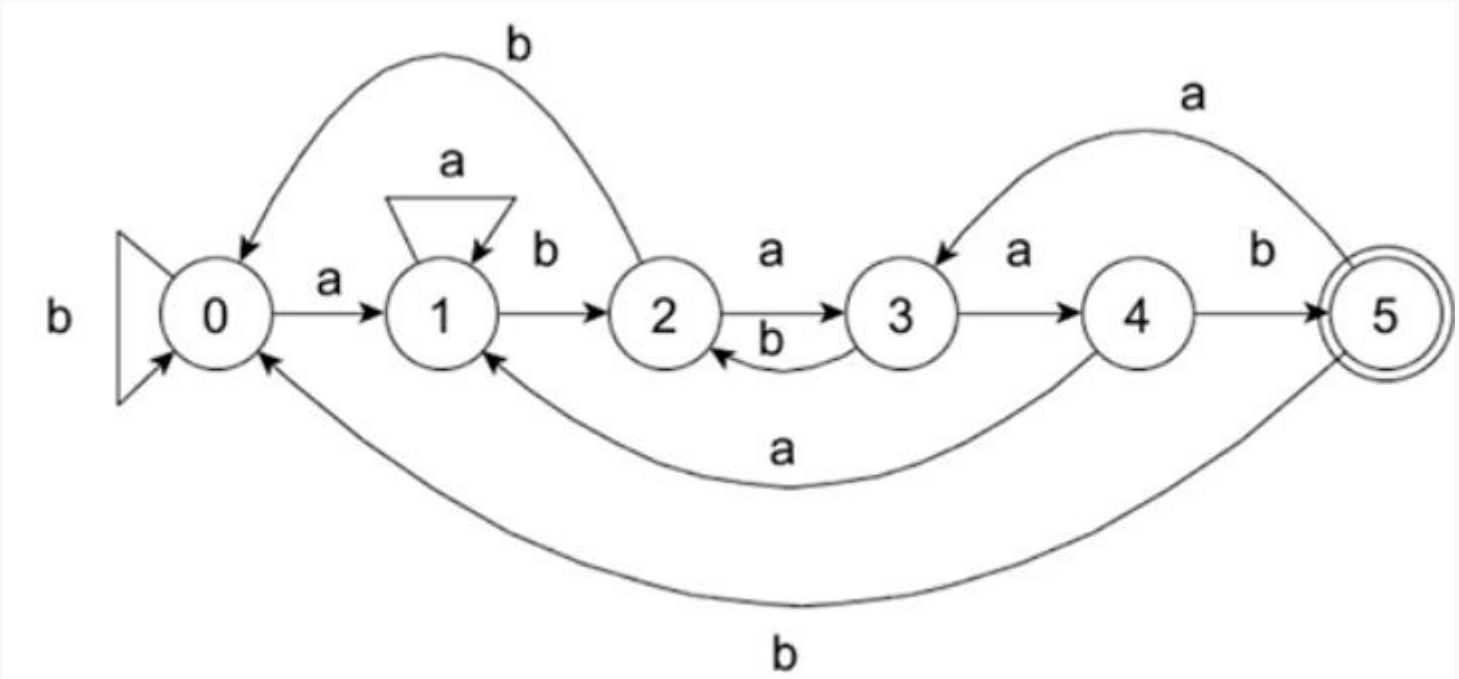
$W[i]$	A	B	A	C	A	B
$S[0:i]$	A	AB	ABA	ABAC	ABACA	ABACAB
$f[i]$	0	0	1	0	1	2

Далее с использованием этой табличной функции идет поиск

							M	A	T	C	H	!							
A	B	A	B	A	C	A	A	B	A	C	A	B	C	A	C	B	C	A	A	A
							A	B	A	C	A	B							

Смотрим $f[2] == 1$, сдвигаем на $3 - f[2] == 2$

FAILURE FUNCTION – ЭТО АВТОМАТ



Пример автомата для искомой строки abbab

ОБСУЖДЕНИЕ

Как вы думаете, можем ли мы поставить задачу более общо и искать не обязательно непрерывную подпоследовательность в последовательности?

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Бьерн Страуструп, Язык программирования C++/ ред. А. Боборыкин. – 4-е изд. - Москва: Издательство БИНОМ, 2023. – 1213 с.
2. Дональд Кнут Искусство программирования, том 1. Основные алгоритмы = The Art of Computer Programming, vol.1. Fundamental Algorithms. — 3-е изд. — М .: «Вильямс» , 2006. – 720 с.
3. Дональд Кнут Искусство программирования, том 2. Получисленные алгоритмы - The Art of Computer Programming, vol.2. Seminumerical Algorithms. — 3-е изд. — М .: «Вильямс» , 2007. — 832 с.
4. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.: МЦМНО, 1999. – 960 с.
5. Скотт Мейерс, Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ / ред. Д.А. Мовчан – 3-е изд. – Москва: ДМК Пресс, 2017. – 300 с.
6. Скотт Мейерс, Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов / ред. Д.А. Мовчан – 3-е изд. – Москва: ДМК Пресс, 2016. – 298 с.