СТРОКИ И АВТОМАТЫ

Работа с текстовой информацией в языке C.

К. Владимиров, Yadro, 2024
mail-to: konstantin.vladimirov@gmail.com

СЕМИНАР 5.1

Символы и строки.

Значения символов

- То, что вы привыкли считать символами, на самом деле кодируется числами

```
int c = 'A';
printf("%d\n", c); // что на экране?
```

- Небольшие цифры это коды для символов

```
char x = 65;
printf("%c\n", x); // а сейчас?
```

- В Unix формат символов это UTF-8.
- Его первые 127 символов совпадают с ASCII

Специальные символы

Code	Name	Escape	Abbr	Keys
0	Null		NUL	CTRL+@
1	Start of head		SOH	CTRL+A
2	Start of text		STX	CTRL+B
3	End of text		ETX	CTRL+C
4	End of xmit		EOT	CTRL+D
5	Enquiry		ENQ	CTRL+E
6	Acknowledge		ACK	CTRL+F
7	Bell	\a	BEL	CTRL+G
8	Backspace	\b	BS	CTRL+H
9	Horz tab	\t	TAB	CTRL+I
10	Line feed	\n	LF	CTRL+J
11	Vert tab	\v	VT	CTRL+K
12	Form feed	\f	FF	CTRL+L
13	Carriage feed	\r	CR	CTRL+M
14	Shift out		SO	CTRL+N
15	Shift in		SI	CTRL+O
16	Data line escape		DLE	CTRL+P
17	Device ctrl 1		DC1	CTRL+Q
18	Device ctrl 2		DC2	CTRL+R
19	Device ctrl 3		DC3	CTRL+S
20	Device ctrl 4		DC4	CTRL+T
21	Neg acknowledge		NAK	CTRL+U
22	Sync idle		SYN	CTRL+V
23	End xmit block		ETB	CTRL+W
24	Cancel		CAN	CTRL+X
25	End of medium		EM	CTRL+Y
26	End of file		EOF	CTRL+Z
27	Escape		ESC	CTRL+[
28	File separator		FS	CTRL+\
29	Group separator		GS	CTRL+]
30	Record separator		RS	CTRL+^
31	Unit separator		US	CTRL+_

https://en.wikipedia.org/wiki/C0_and_C1_control_codes

Таблица ASCII

0		(nul)	16	►	(dle)	32	sp	48	0	64	@	80	P	96	`	112	p
1	☺	(soh)	17	◄	(dc1)	33	!	49	1	65	A	81	Q	97	a	113	q
2	☻	(stx)	18	↕	(dc2)	34	"	50	2	66	B	82	R	98	b	114	r
3	♥	(etx)	19	‼	(dc3)	35	#	51	3	67	C	83	S	99	c	115	s
4	♦	(eot)	20	¶	(dc4)	36	$	52	4	68	D	84	T	100	d	116	t
5	♣	(enq)	21	§	(nak)	37	%	53	5	69	E	85	U	101	e	117	u
6	♠	(ack)	22	▬	(syn)	38	&	54	6	70	F	86	V	102	f	118	v
7	•	(bel)	23	↨	(etb)	39	'	55	7	71	G	87	W	103	g	119	w
8	◘	(bs)	24	↑	(can)	40	(	56	8	72	H	88	X	104	h	120	x
9		(tab)	25	↓	(em)	41	)	57	9	73	I	89	Y	105	i	121	y
10		(lf)	26		(eof)	42	*	58	:	74	J	90	Z	106	j	122	z
11	♂	(vt)	27	←	(esc)	43	+	59	;	75	K	91	[	107	k	123	{
12	♀	(ff)	28	∟	(fs)	44	,	60	<	76	L	92	\	108	l	124	|
13		(cr)	29	↔	(gs)	45	-	61	=	77	M	93	]	109	m	125	}
14	♫	(so)	30	▲	(rs)	46	.	62	>	78	N	94	^	110	n	126	~
15	☼	(si)	31	▼	(us)	47	/	63	?	79	O	95	_	111	o	127	⌂

Категории символов

- Специальный заголовочный файл <ctype.h> содержит прототипы функций, категоризирующих символы.
- Например is_alpha(c) проверяет является ли c одним из алфавитных символов (abcdef....), а is_digit(c) проверяет на цифру. Их объединяет is_alphanum(c)
- Иногда в программах которые анализируют символьную информацию важно знать что на вход пришло "что-то вроде буквы" или "что-то вроде пробела"
- На следующей картинке сведены основные определители из ctype

Категории символов

ASCII values			characters	iscntrl iswcntrl	isprint iswprint	isspace iswspace	isblank iswblank	isgraph iswgraph	ispunct iswpunct	isalnum iswalnum	isalpha iswalpha	isupper iswupper	islower iswlower	isdigit iswdigit	isxdigit iswxdigit
decimal	hexadecimal	octal													
0–8	\x0–\x8	\0–\10	control codes (NUL, etc.)	≠0	0	0	0	0	0	0	0	0	0	0	0
9	\x9	\11	tab (\t)	≠0	0	≠0	≠0	0	0	0	0	0	0	0	0
10–13	\xA–\xD	\12–\15	whitespaces (\n, \v, \f, \r)	≠0	0	≠0	0	0	0	0	0	0	0	0	0
14–31	\xE–\x1F	\16–\37	control codes	≠0	0	0	0	0	0	0	0	0	0	0	0
32	\x20	\40	space	0	≠0	≠0	≠0	0	0	0	0	0	0	0	0
33–47	\x21–\x2F	\41–\57	!"#$%&'()*+,-./	0	≠0	0	0	≠0	≠0	0	0	0	0	0	0
48–57	\x30–\x39	\60–\71	0123456789	0	≠0	0	0	≠0	0	≠0	0	0	0	≠0	≠0
58–64	\x3A–\x40	\72–\100	:;<=>?@	0	≠0	0	0	≠0	≠0	0	0	0	0	0	0
65–70	\x41–\x46	\101–\106	ABCDEF	0	≠0	0	0	≠0	0	≠0	≠0	≠0	0	0	≠0
71–90	\x47–\x5A	\107–\132	GHIJKLMNOP QRSTUVWXYZ	0	≠0	0	0	≠0	0	≠0	≠0	≠0	0	0	0
91–96	\x5B–\x60	\133–\140	[\]^_`	0	≠0	0	0	≠0	≠0	0	0	0	0	0	0
97–102	\x61–\x66	\141–\146	abcdef	0	≠0	0	0	≠0	0	≠0	≠0	0	≠0	0	≠0
103–122	\x67–\x7A	\147–\172	ghijklmnop qrstuvwxyz	0	≠0	0	0	≠0	0	≠0	≠0	0	≠0	0	0
123–126	\x7B–\x7E	\172–\176	{|}~	0	≠0	0	0	≠0	≠0	0	0	0	0	0	0
127	\x7F	\177	backspace character (DEL)	≠0	0	0	0	0	0	0	0	0	0	0	0

Источник: https://en.cppreference.com/w/c/string/byte/isspace

Disclaimer: мы не нырнём глубоко

- Далее мы работаем только с ASCII строками то есть со строками, состоящими из однобайтных символов UTF8.
- Учёт различных кодировок, русских букв, китайских иероглифов и всего такого просто не входит в программу первого курса. Могу предложить собраться дополнительно.

Строковые литералы

- Следующая строчка может показаться смутно знакомой

```
printf("%s\n", "Hello, world!");
```

- Символы `"Hello, world"` в кавычках это строковый литерал

H	e	l	l	o	,		w	o	r	l	d	!	\0

- Строковый литерал это строка известная во время компиляции
- Любая строка в языке C оканчивается завершающим нулевым символом
- Нулевой символ `'\0'` это не символ `'0'`. Это спецсимвол, имеющий код `0`.

Изменяемые строки

- Самый простой способ создать изменяемую строку это объявить массив символов

```
char hello[20] = "Hello, world!";
```

- Символы после завершающего нуля (он тут 14-й) содержат мусор
- Далее содержимое этой строки можно поменять

```
hello[1] = 'a';
printf("%s\n", hello); // → Hallo, world!
```

- Изменяемая строка это любой массив символов, завершающийся нулём
- Символы после завершающего нуля ничего не значат

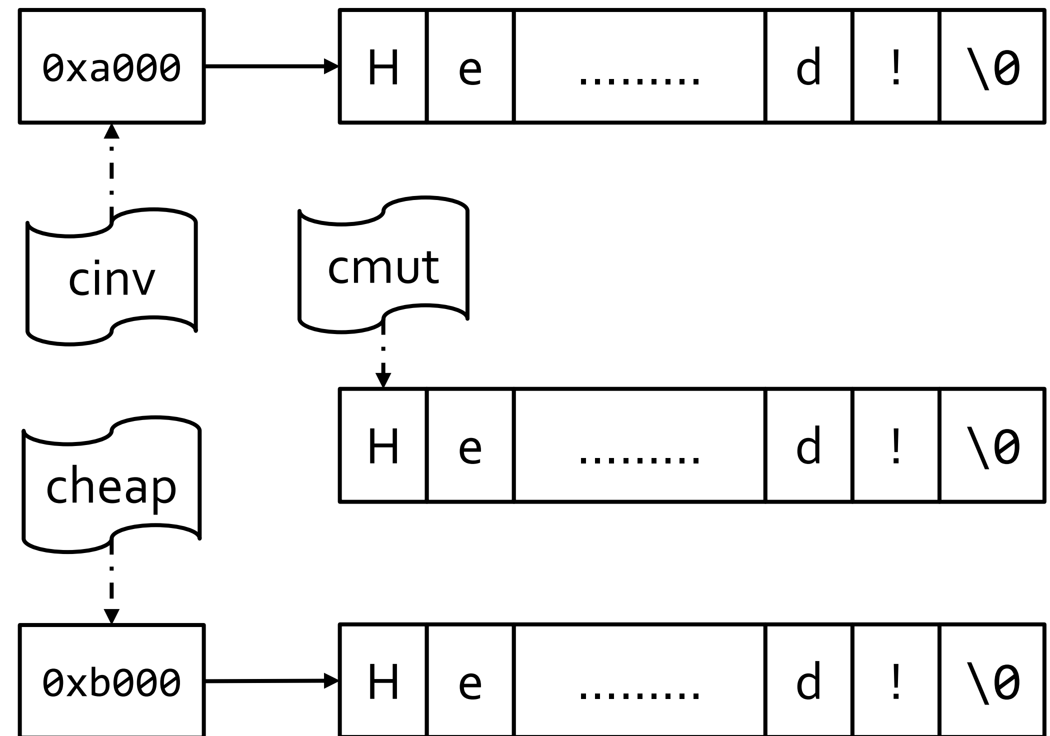
C-строки и разные виды памяти

- Обычно используют указатель на первый элемент

```
char const * cinv = "Hello, world";
char cmut[] = "Hello, world";
char *cheap = (char *) malloc(50);
strcpy (cheap, cinv);
```

- Что вы думаете про следующие строки?

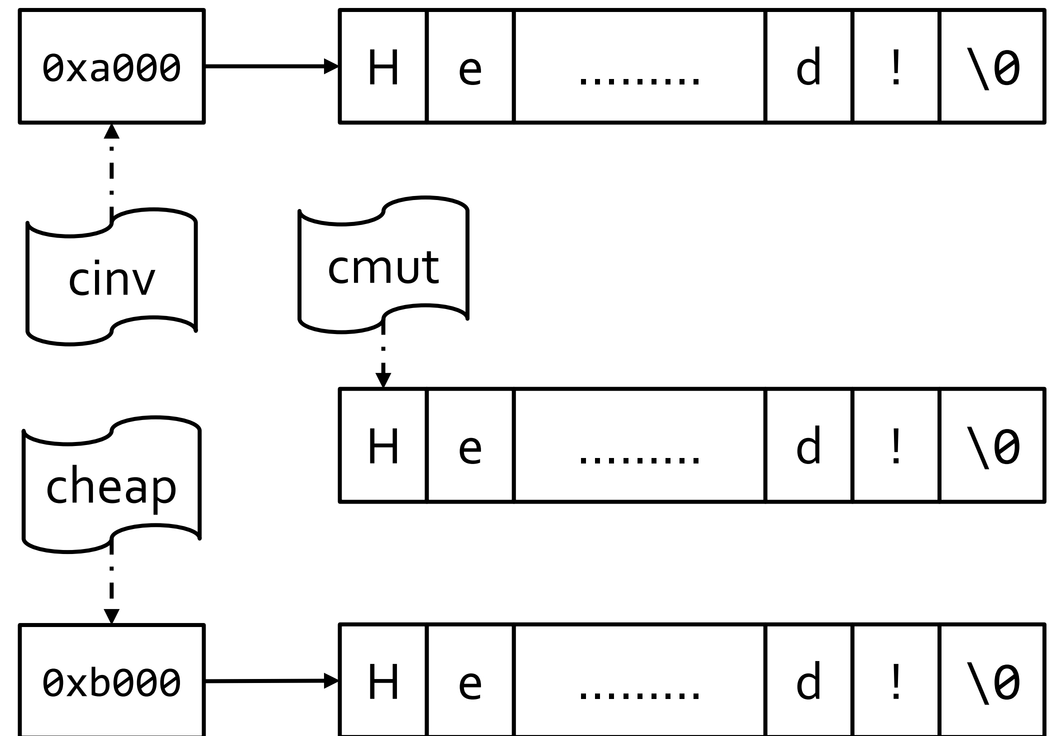
```
cheap = cinv; // ???
cinv = 0;     // ???
cmut = cheap; // ???
```



C-строки и разные виды памяти

- Обычно используют указатель на первый элемент

```
char const * cinv = "Hello, world";
char cmut[] = "Hello, world";
char *cheap = (char *) malloc(50);
strcpy (cheap, cinv);
cheap = cinv; // ошибка компиляции
              + утечка памяти
cinv = 0; // ok
cmut = cheap; // ошибка компиляции
```



Problem SCN – суммирование в строке

- Ваша задача написать программу которая суммирует строчку, воспринимая символы в ней как коды этих символов

H	e	l	l	o	,		w	o	r	l	d	!	\0
#72	#101	#108	#108	#111	#44	#32	#119	#111	#114	#108	#100	#33	#0

- Для `"Hello, world!"` ответом будет `1161`

Копирование строк

- Одна строка может быть скопирована в другую

```
char duplicate[20];

const char *src = "Hello, world!";
char *dst = duplicate;

while (*src != '\0') {
  *dst = *src;
  dst += 1;
  src += 1;
}

*dst = '\0';
```

Копирование строк

- Одна строка может быть скопирована в другую

```
char duplicate[20];

const char *src = "Hello, world!";
char *dst = duplicate;

while ((*dst++ = *src++) != '\0') {} // немного джигитовки
```

Копирование строк

- Одна строка может быть скопирована в другую

```
#include <string.h>

const char *src = "Hello, world!";
char duplicate[20];

strcpy(duplicate, src); // библиотечная функция
```

- Вашим выбором по умолчанию должно быть именно использование библиотечных функций, так как они почти всегда корректны и куда лучше оптимизированы.
- В языке C довольно много библиотечных функций для работы со строками.

API для работы с C-строками

Библиотечная функция	Что она делает
`strcpy(dest, src)`	Копирует `src` в `dest`
`strcat(dest, src)`	Дописывает `src` в конец `dest`
`strlen(src)`	Вычисляет длину `src`
`strcmp(src1, src2)`	Сравнивает `src1` и `src2`. Возвращает `0`, `1` или `-1`
`strchr(src, c)`	Ищет символ в строке
`strstr(haystack, needle)`	Ищет подстроку в строке
`strspn(src1, src2)`	Ищет максимальный префикс `src1` состоящий только из символов `src2`
`strcspn(src1, src2)`	Тоже только не из символов `src2`
`strpbrk(src1, src2)`	Ищет первое вхождение в `src1` любого символа из `src2`

Обсуждение

- Одна забавная странность в сигнатурах стандартной библиотеки заключается в том, что возвращаемый тип почти всегда `char*`

```
char *strstr(const char *s1, const char *s2);
```

- Казалось бы, если мы принимаем оба аргумента `const char*` почему бы не возвращать `const char*`
- У кого есть идеи почему так сделано?

Problem SR – переворот подстрок

- Напишите программу, которая берёт одно слово, а потом некоторый текст и переворачивает в этом тексте все вхождения этого слова
- Например для слова `"world"` и текста `"Hello, world!"` результатом должно быть `"Hello, dlrow!"`
- Вы можете предполагать что текст состоит из слов, разделённых пробелами и пунктуацией, а слово состоит из алфавитных символов
- Вы также можете использовать любые функции стандартной библиотеки, включая `strstr` для поиска подстроки

Обсуждение

- Ещё одной проблемой strcat (кроме обсуждённых ранее проблем с асимптотикой) является возможное переполнение буфера
- Допустим у нас строки живут только в динамической памяти
- Можем ли мы тогда написать strcat которая увеличивает буфер, если его не хватает? Как она могла бы работать?

Реаллокации

- Чтобы немного расширить буфер в динамической памяти, делать `malloc` на новый размер и `free` на старом месте может быть накладно
- Здесь на помощь приходит `realloc`

```
void * realloc(void * ptr, size_t new_size);
```

- Например

```
int * arr = (int *) malloc(100);
arr = (int *) realloc(arr, 1000);
```

- При нехватке памяти, `realloc` возвращает `NULL`
- Внимание: `realloc` не работает на стеке и в глобальной памяти

Problem SA – concat + realloc

- Ваша задача написать функцию

```
char *strcat_r(char *dest, int bufsz, const char *src);
```

- При нехватке места в старом буфере эта функция должна реаллоцировать память и возвращать новый указатель
- Будьте очень внимательны с нулевым символом

Problem SP – замена в строке

- Реализуйте функцию, осуществляющую замену всех подстрок в строке
- Функция должна вернуть новую строку, аллоцированную в куче

```
char * replace(char *str, char const *from, char const *to);
```

- Например

```
char *s = (char *) malloc(41);
strcpy(s, "Hello, %username, how are you, %username");

char *r = replace(s, "%username", "Eric, the Blood Axe");
```

- Обратите внимание, что строка from может быть и длиннее и короче чем to

Problem SU – поиск CI подстроки

- Напишите простую функцию `strstrci`, которая ищет подстроку в строке, независимо от регистра символов (`ci` означает `case insensitive`)

```
char * strstrci(char const * needle, char const * haystack);
```

- Ниже приведён пример применения

```
char const *needle = "Ab", *src = "abracadaBra";
char *pos1, *pos2, *pos3;

pos1 = strstrci(src, needle);       assert(pos1 != NULL);
pos2 = strstrci(pos1 + 2, needle);  assert(pos2 != NULL);
pos3 = strstrci(pos2 + 2, needle);  assert(pos3 == NULL);
```

- Первая найденная позиция `"abracadabra"`, вторая `"abracadaBra"`
- Оцените асимптотику наивного алгоритма

Задача ассемблирования

- Простейший ассемблерный код состоит из мнемоник, которые работают с регистрами, константами и т.д.

```
add r0, r1, r2

or r0, r0, 14

sub r1, r0, 45
```

- Представим что нам надо считать массив мнемоник и их операндов.
- Эта задача является задачей лексического анализа.

Пример: лексический анализ

- Нужно ввести выражение, содержащее мнемонику и операнды

```
add r0, r1, r2
```

- Казалось бы это несложно, но есть проблемы
- Пробелы могут быть введены лишние или не введены вообще

```
add r0,r1,r2
```

- Нужна диагностика для ошибок

```
addf r0,r1,
```

- Как представить считанное выражение?

Задача лексического анализа

```
"   add r0, r1,    r2"
```

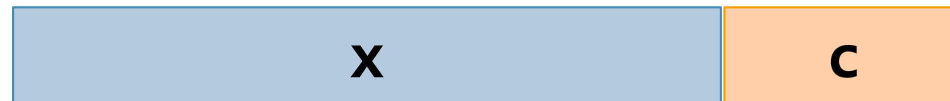
add	r0	r1	r2

- Как вы хотели бы хранить сведения о лексеме в памяти компьютера?
- Тип лексемы: оператор, скобка, число.
- Вместе с оператором надо хранить тип операции, со скобкой тип скобки с числом его значение.

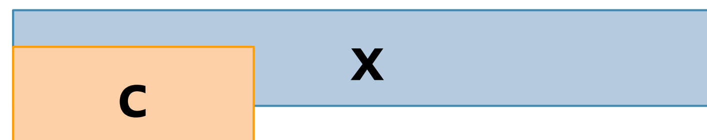
Объединения

- Решением проблемы являются объединения.
- Объединение очень похоже на структуру, только все поля в нём лежат по одному адресу.

```
struct S {
  int x;
  char c;
};
```



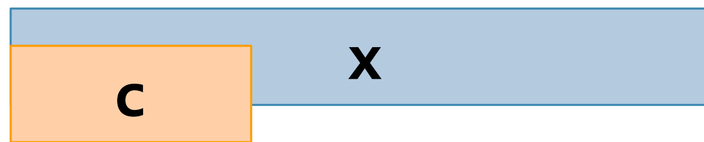

```
union U {
  int x;
  char c;
};
```



Объединения

- Брать из объединения лучше именно то, что туда положил.

```
union IntChar {
  int x;
  char c;
};

union IntChar ic;

ic.c = 'a';
int y = ic.x; // type punning
```

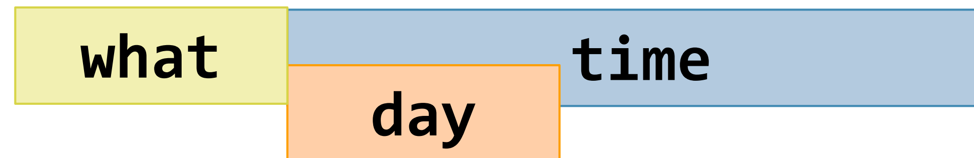


- И здесь удобно использовать перечисления.

Объединения

- Здесь показана структура с объединением и перечислением.

```
enum DTS { DT_DAY = 0, DT_TIME };

struct DT {
  enum DTS what;
  union DayOrTime {
    int day;
    time_t time;
  } u;
}

struct DT d1 = { .what = DT_DAY, .u.day = 42 }; // C style
struct DT d2 = { DT_DAY, 42 }; // C++ compatible
```



Идея: массив лексем

- Лексема это операция, регистр или число.

```
enum lexem_kind_t { OP, REG, NUM };
enum operation_t { ADD, SUB, MUL, DIV };

struct lexem_t {
  enum lexem_kind_t kind;
  union {
    enum operation_t op;
    int reg;
    int num;
  } lex;
};
```

```
"   add r0, r1,    r2"
```

add	r0	r1	r2

СЕМИНАР 5.2

Конечные автоматы и регулярные выражения.

Алфавиты и строки

- Алфавит это множество символов, например $\{a, b, c\}$
- Строка это последовательность символов, например $w = \{a, a, c, b\}$
  - 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 строк
  - Язык L_{free} = все возможные строки алфавита (группа по 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

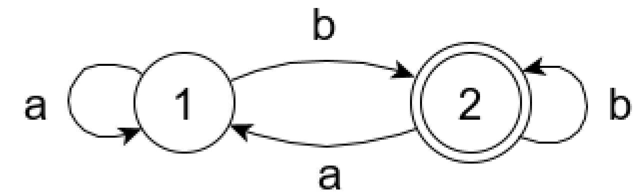
- Принадлежность: имея язык L и строчку w, определить принадлежит ли она языку.
- Порождение: имея язык L, порождать все его строки последовательно.
- Эквивалентность: имея язык L_1 и язык L_2, определить принадлежат ли им одинаковые элементы.
- Отрицание: имея язык L_1, описать язык L_2, такой, что 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}

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

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



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

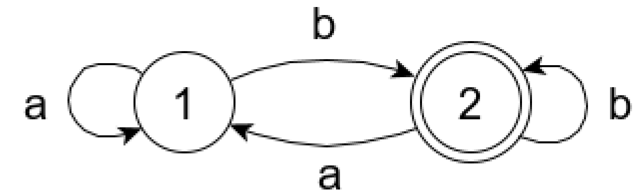
- Исполняя шаг за шагом правильную входную строку, мы закончим в принимающем состоянии.

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

Автоматы на языке C

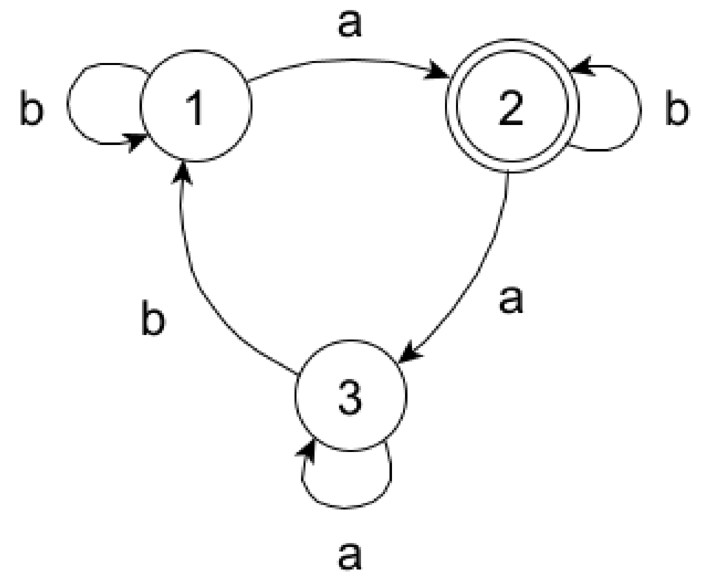
- Простой автомат моделируется простой программой.

```
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': {
    switch(state) {
      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.

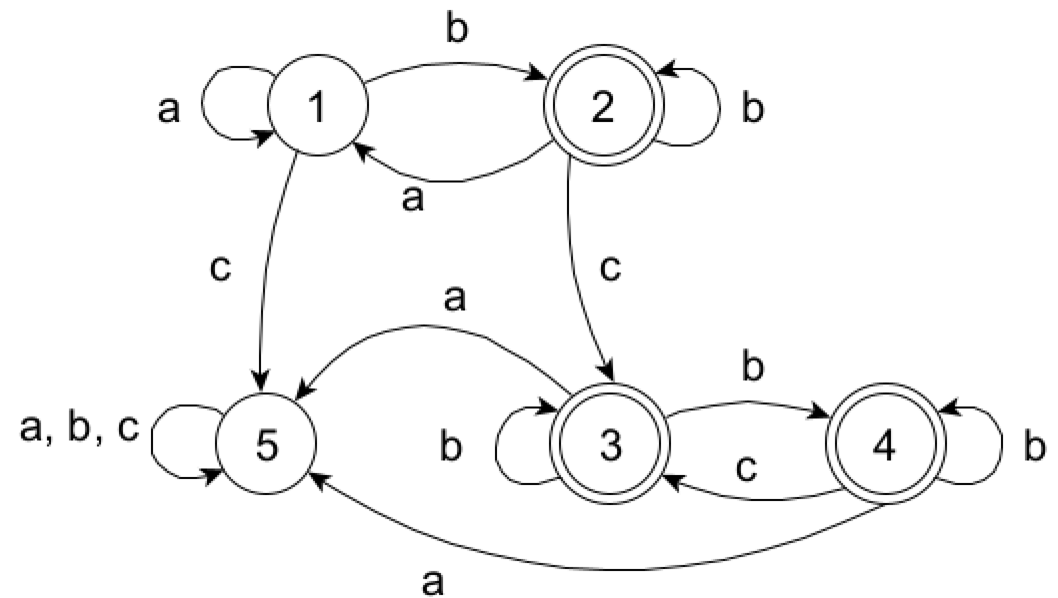
```
void copy(int *to, int *from, int count) {
  n = (count + 3) / 4;
  switch (count % 4) {
    case 0: do { *to++ = *from++;
    case 3:      *to++ = *from++;
    case 2:      *to++ = *from++;
    case 1:      *to++ = *from++;
            } while (--n > 0);
  }
}
```

Упражнение с автоматами

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

Problem RXA

- Напишите на языке C программу, которая симулирует следующий автомат.
- По странному совпадению это как раз автомат для языка L_{xby}
- Далее мы узнаем для каких вообще языков бывают возможны автоматы.



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

Регулярные выражения

- Любой алфавитный символ означает язык из этого символа: a это $\{a\}$
- Конкатенация $L_x L_y = \{wz | w \in L_x \land z \in L_y\}$
- Дизъюнкция $(L_x + L_y) \ = \{w | w \in L_x \lor w \in L_y\}$
- Замыкание $(L_x)^* = \{\{\Lambda\}, L_x, L_x L_x, L_x L_x L_x, \dots\}$
- 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, c, z

[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, ...
```


BRE syntax

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: `0`, если строка сматчилась, иначе REG_NOMATCH, возможны также ошибки.

https://godbolt.org/z/Pd4GTrvbz

Обработка ошибок

- Ошибки обрабатываются очень похоже на perror для файлов.

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

```
if (res != 0 && res != REG_NOMATCH) {
  regerror(res, &regex, buf, sizeof(buf));
  fprintf(stderr, "Regex match failed: %s\n", buf);
}
```

- В конце мы освобождаем откомпилированную регулярку.

```
regfree(&regex);
```


https://godbolt.org/z/PE5fezfE1

Problem RXB – 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

```
vasya1976@mail.ru
Tamara.Ivanovna@gmail.com
3apa3a@vasyan.org
```

- Ваша задача отличить правильные адреса от неправильных. Для правильных выводите на stdout число `1`, для неправильных `0`.

Матчим куски регулярных выражений

- Мы можем не только отвечать на вопрос "есть или нет" но и вернуть то что конкретно попало под паттерн.

```
regmatch_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.

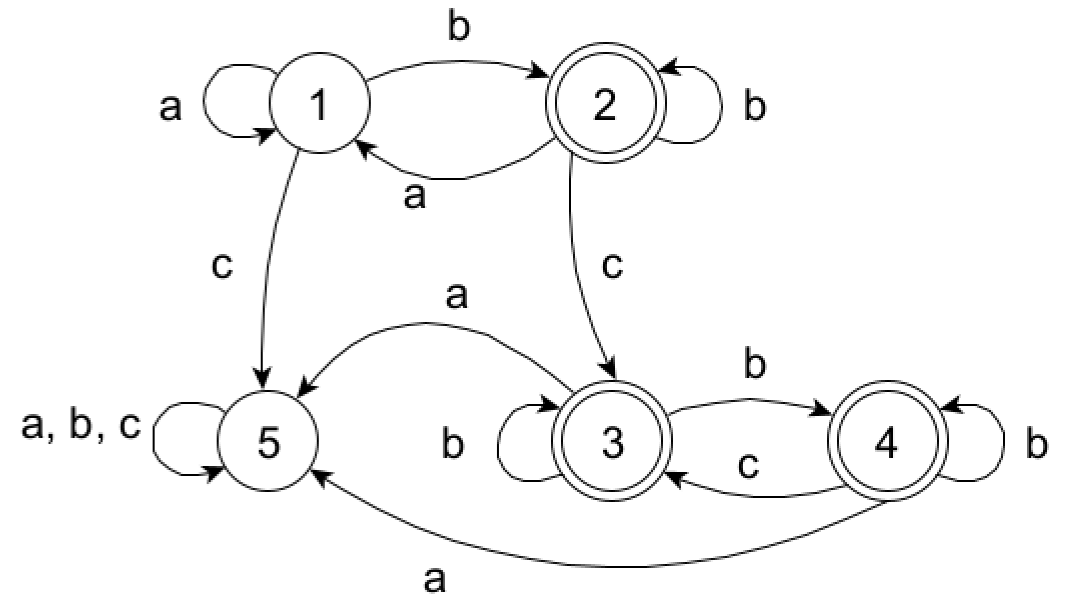
https://godbolt.org/z/8cb8hfr5a

Problem RXM – примерный поиск

- Продолжая Problem SR 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.



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

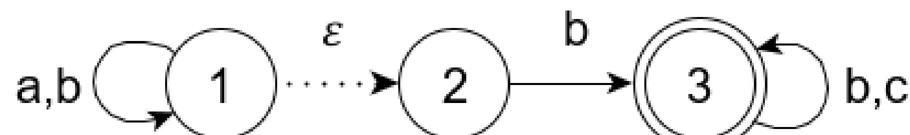
Эпсилон-переходы

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

aabbcb

aabbbabcc

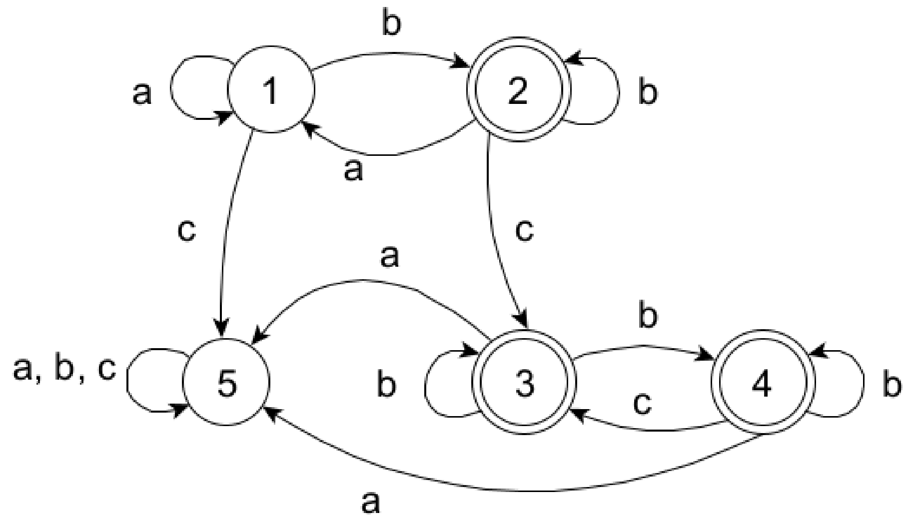
- Что если мы разрешим спонтанные (эпислон) переходы?
- Будем считать, что автомат принимает строку, если хотя бы одна траектория ведет в принимающее состояние
- Такие автоматы будем называть недетерминированными NFA в противоположность DFA



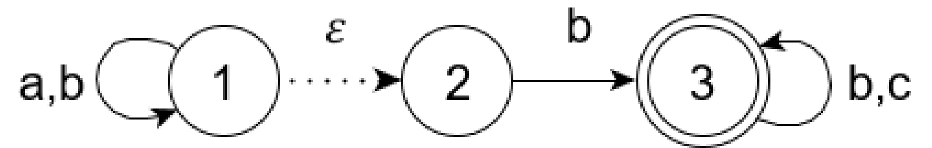
$$L_{xby} = (a|b)^* b (b|c)^*$$

Обсуждение

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



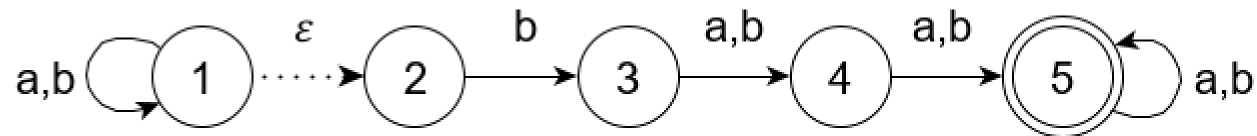
s a a b b b a b c c
1 3 2 3 4 4 2 3 5 5



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



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

[Статья Майкла Рабина и Даны Скотта]

Problem RXP – упражнения 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 найдётся такая декомпозиция $w = xyz$, что все слова xy^nz также принадлежат этому языку.
- Поэтому язык a^nb^m регулярный: вместе с $a^{n-1}ab^m$ ему принадлежат все $a^{n-1}a^kb^m$.
- Но это значит, что язык a^nb^n **не регулярный**.
- Также не регулярен язык всевозможных регулярных выражений.

Pumping lemma

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@ABCDEABCDEABCDEABCDZABCDE

0000001234012340123401234512340

- Обратим внимание на забавный факт: мы легко нашли вхождение просто перебирая значения префиксной функции.

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 ещё раз.
- Пример: abacab

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

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

$S[6] = S[k+1] = S[2] \Rightarrow f[6] = k = 2$;

HWK – алгоритм КМП (optional)

- 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 идёт 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`

HWK – алгоритм КМП (optional)

- 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 идёт 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`

HWK – алгоритм КМП (optional)

- 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 идёт поиск

						1														
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`

HWK – алгоритм КМП (optional)

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

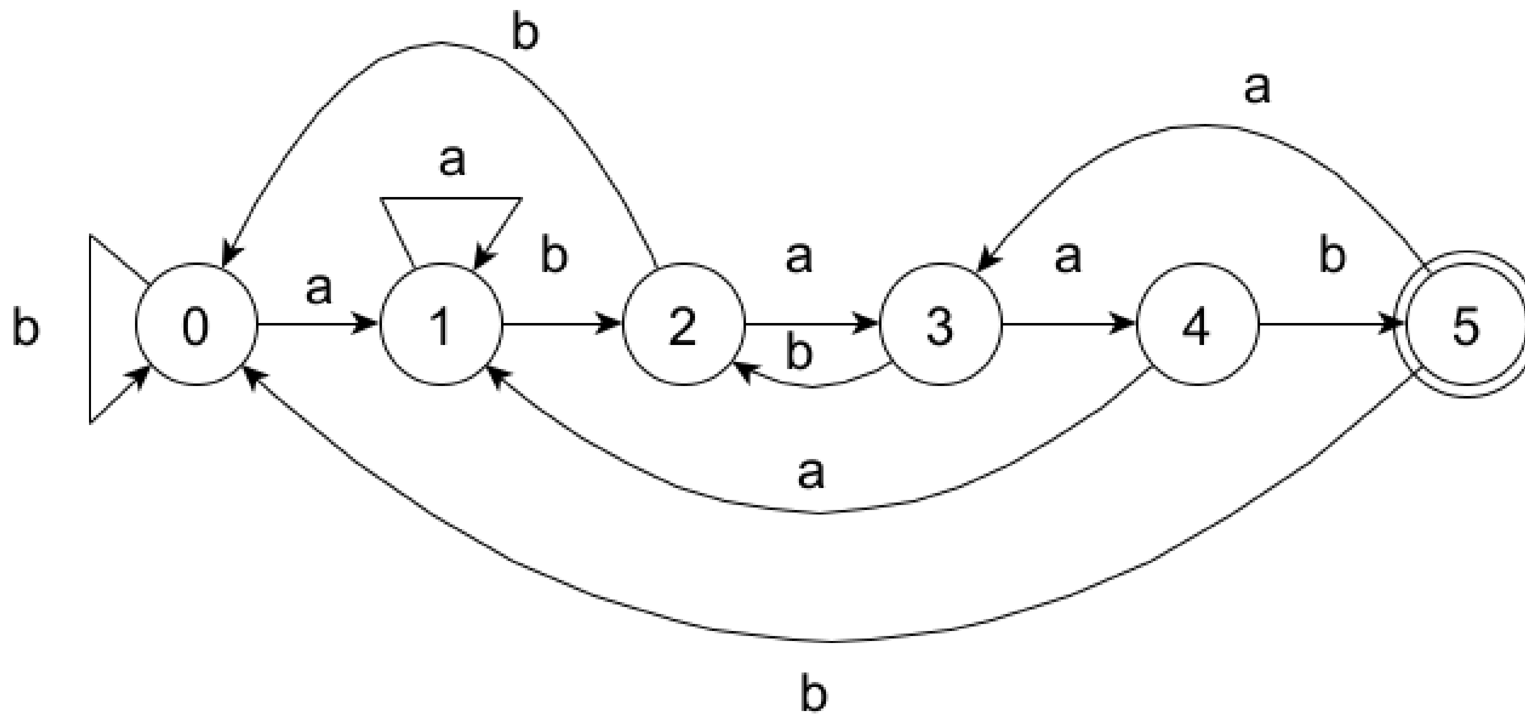
W[i]	A	B	A	C	A	B
S[0:i]	A	AB	ABA	ABAC	ABACA	ABACAB
f[i]	0	0	1	0	1	2

- Далее с использованием этой табличной функции идёт 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 abaab.

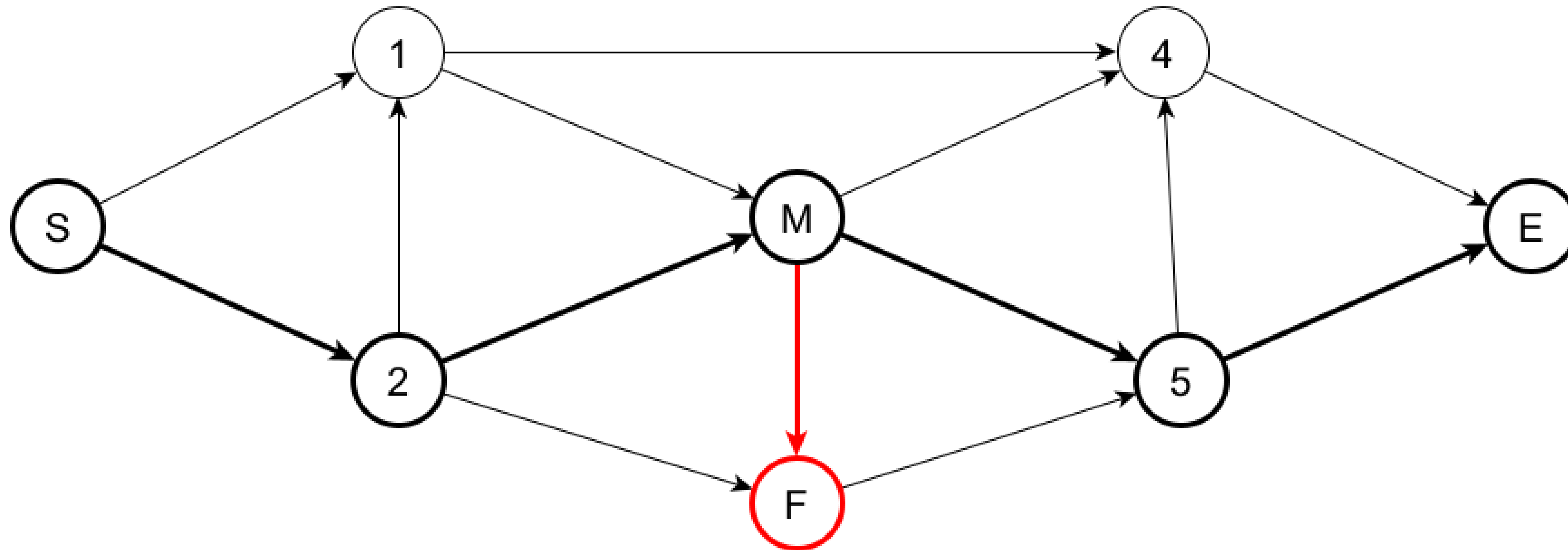
Обсуждение

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

СЕМИНАР 5.3

Дискретное динамическое программирование

Принцип оптимальности Беллмана



- Если траектория от S до E оптимальна, то и траектория от S до M.
- Но не наоборот. Траектории от S до M и от M до F вместе не дают оптимальную траекторию.

Функциональное уравнение Беллмана

- «An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision» $[Bellman]$
- Изменение состояния:$x_t \to x_{t+1} = T(x_t, a_t)$ with pay-off $F(x_t, a_t)$
- Обозначим $V(x_0) = \max\limits_{a_0 \dots a_T} \sum_t F(x_t, a_t)$
- Тогда $V(x_t) = \max\limits_{a_t}\left\{F(x_t, a_t) + V\left(T(x_t, a_t)\right)\right\}$
- Простыми словам: любой кусок оптимальной траектории является оптимальной траекторией.

Задача размена монет

- Пусть есть монеты номиналами $c_1 \dots c_k$ и необходимо разменять сумму N взяв наименьшее количество монет.
- x_t это оставшаяся сумма для размена.
- $a_t \in \{c_1 \dots c_k\}$ это решение сколько мы берём на каждом шаге.

$$x_{t+1} = x_t - a_t$$

$$F(x_t, a_t) = 1$$

- Мы должны минимизировать количество шагов, то есть это задача на минимизацию, а не на максимизацию.
- Как записать функцию Беллмана?

Задача размена монет

- Пусть есть монеты номиналами $c_1 \ldots c_k$ и необходимо разменять сумму N взяв наименьшее количество монет.
- Запишем функциональное уравнение Беллмана.

$$V(t) = \min\big(V(t - c_1), \ldots, V(t - c_k)\big) + 1$$

- Иными словами: на каждом шаге мы берем такую монету, чтобы функция Беллмана была минимальной на прошлом шаге.
- Нас не волнует что тут min, т.к. $\max\big(f(t)\big) = -\min\big(-f(t)\big)$ и наоборот.

Размен монет

- Приложение к размену монет: пусть есть монеты номиналами c_1, c_2, c_3 и необходимо разменять сумму N
- Заведём таблицу $V(x_k)$ от 0 до N и протабулируем оптимальные размены. Пусть есть номиналы $1,3,4$ и надо разменять 10 используя минимальное количество монет

k	1	2	3	4	5	6	7	8	9	10

V	1	2								

Размен монет

- Приложение к размену монет: пусть есть монеты номиналами c_1, c_2, c_3 и необходимо разменять сумму N
- Заведём таблицу $V(x_k)$ от 0 до N и протабулируем оптимальные размены. Пусть есть номиналы $1{,}3{,}4$ и надо разменять 10 используя минимальное количество монет

k	1	2	3	4	5	6	7	8	9	10
V	1	2	1	1						

Размен монет

- Приложение к размену монет: пусть есть монеты номиналами c_1, c_2, c_3 и необходимо разменять сумму N
- Заведём таблицу $V(x_k)$ от 0 до N и протабулируем оптимальные размены. Пусть есть номиналы $1,3,4$ и надо разменять 10 используя минимальное количество монет

k	1	2	3	4	5	6	7	8	9	10

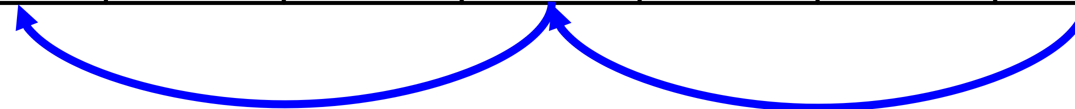
V	1	2	1	1	2					

Размен монет

- Приложение к размену монет: пусть есть монеты номиналами c_1, c_2, c_3 и необходимо разменять сумму N
- Заведём таблицу $V(x_k)$ от 0 до N и протабулируем оптимальные размены. Пусть есть номиналы $1,3,4$ и надо разменять 10 используя минимальное количество монет

k	1	2	3	4	5	6	7	8	9	10

V	1	2	1	1	2	2	2	2	3	3



Восходящее и нисходящее решение

```
int calc_change(int n, int m, int *changes, int *V) {
  int i;
  if (n == 0) return 0;
  if (n < 0) return INT_MAX;
  if (V[n] != INT_MAX) return V[n];
  for (i = 0; i < m; ++i) {
    int CT = calc_change(n - changes[i], m, changes, V);
    if (CT == INT_MAX) continue;
    V[n] = min(V[n], CT + 1);
  }
  return V[n];
}
```

$$V(t) = \min\big(V(t - c_1), \ldots, V(t - c_k)\big) + 1$$

Problem BP – задача о рюкзаке

- Вам необходимо написать программу, которая считывая со стандартного ввода:
  - Количество вещей
  - Общий вес входящий в рюкзак
  - Вес каждой вещи w_i
  - Стоимость каждой вещи v_i
- Выдавало бы наибольшую стоимость вещей, которые можно положить в рюкзак.
- Например для: $4\ 6\ 4\ 3\ 2\ 1\ 5\ 4\ 3\ 2$ ответом будет $2 + 3 + 4 = 9$

Функциональное уравнение Беллмана

$V(0, w) = 0$

$V(i, w) = V(i - 1, w)$ для $w < w_i$

$V(i, w) = \max(V(i - 1, w), V(i - 1, w - w_i) + v_i)$ для $w \geq w_i$

i	v	w
1	5	4
2	4	3
3	3	2
4	2	1

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3	4	5	7	8
4	0	2	3	5	6	7	9

Обсуждение

- Довольно сложно было догадаться в задаче о рюкзаке, что функция Беллмана будет двумерной.
- У нас там две степени свободы: каждую вещь можно взять или не взять (первый уровень) с учётом того как взяты остальные вещи (второй уровень).

Расстояние редактирования

- Как перейти от слова spoon к слову sponge?
- Что можно делать:
  - Вставлять букву в любое место (цена == 1)
  - Удалять любую букву (цена == 1)
  - Изменять любую букву (цена == 2)
- Нужно найти последовательность действий с минимальной ценой

```
spoon → sponn (+2) → spong (+2) → sponge (+1), cost = 5
spoon → spon (+1) → spong (+1) → sponge (+1), cost = 3
```

- Можем ли мы применить динамическое программирование?

Problem ED – расстояние редактирования

- Реализуйте программу, которая считывает со стандартного ввода
  - стоимость добавления
  - стоимость удаления
  - стоимость замены
  - длину первой строки
  - первую строку
  - длину второй строки
  - вторую строку
- И выводит на стандартный вывод минимальное расстояние редактирования

Функциональное уравнение

- Любая часть оптимальной траектории является оптимальной траекторией.

$$V(i,j) = min\begin{cases} V(i-1,j)+1 \\ V(i,j-1)+1 \\ V(i-1,j-1)+K(i,j) \end{cases}$$

$$K(i,j) = \begin{cases} 0, & S_1(i) = S_2(j) \\ 2, & S_1(i) \neq S_2(j) \end{cases}$$

- Задача теперь сводится к тому, чтобы найти $D(5,6)$.
- Потренируемся заполнять таблицу?

	#	S	P	O	N	G	E
#	0	1	2	3	4	5	6
S	1	$D(1,1)$					
P	2						
O	3						
O	4						
N	5						$D(5,6)$

Динамическое программирование

- Любая часть оптимальной траектории является оптимальной траекторией!

$$V(i,j) = min\begin{cases} V(i-1,j)+1 \\ V(i,j-1)+1 \\ V(i-1,j-1)+K(i,j) \end{cases}$$

$$K(i,j) = \begin{cases} 0, & S_1(i) = S_2(j) \\ 2, & S_1(i) \neq S_2(j) \end{cases}$$

- Задача теперь сводится к тому, чтобы найти $D(5,6)$
- Нашли.

	#	S	P	O	N	G	E
#	0	1	2	3	4	5	6
S	1	0	1	2	3	4	5
P	2	1	0	1	2	3	4
O	3	2	1	0	1	2	3
O	4	3	2	1	2	3	4
N	5	4	3	2	1	2	**3**

Восстановление решений

- Можем ли мы в задаче о расстоянии редактирования восстановить решение?
- Обратим внимание: может быть больше одного решения весом 3.
- Также следует заметить что у нас есть некоторый выбор: оптимизировать память или восстанавливать решение.

	#	S	P	O	N	G	E
#	0	1	2	3	4	5	6
S	1	0	1	2	3	4	5
P	2	1	0	1	2	3	4
O	3	2	1	0	1	2	3
O	4	3	2	1	2	3	4
N	5	4	3	2	1	2	3

https://en.wikipedia.org/wiki/Hirschberg%27s_algorithm

Вернёмся к пределам регулярности

- Мы хотели бы сматчить все правильные скобочные выражения и только их.
- Для этого невозможно написать регулярное выражение, но можно написать грамматику.

Грамматика

- По определению грамматика состоит из продукций $\alpha \to \beta$.

$A \to A + A \mid (A) \mid A.A \mid A^{*} \mid a \mid b \mid c \mid \varepsilon$

- Нетерминальные символы в общем случае могут стоять и слева и справа, терминальные только справа.
- Для языка регулярных выражений над $\{a, b, c\}$.
- Терминалы: a, b, c, $+$, (,), $*$
- Нетерминалы: пока только A.
- Обратим внимание: во всех продукциях языка регулярных выражений у нас слева всего один нетерминал.

Пример простой грамматики

- **Контекстно свободной** грамматикой называется такая, которую можно представить так, чтобы слева в каждой продукции был ровно один нетерминал.
- Любой регулярный язык тривиально является контекстно свободным
- Язык L_{parm} = $\{a^m b^n c a^m b^n\}$ не является контекстно свободным
- Грамматика для скобочных выражений.

$B \rightarrow (B)\ B$

- Возможные порождаемые строки – любые скобочные выражения.

`(()(())), ((()())()), ....`

Порождение

- Порождаем все скобочные выражения.

$B \rightarrow (B)\ B$

- Сделаем левое порождение (()(()))

$B \rightarrow (B)B \rightarrow ((B)B)B \rightarrow (()B)B \rightarrow (()(B)b)b \rightarrow (()((B)B)B)B$

- Чтобы не порождать, а распознавать выражения, приведём грамматику в нормальную форму.

Нормальная форма Хомского

- Разрешены только следующего вида продукции:

$A \to a, A \to BC, S \to \varepsilon$

- Основная идея в замене $A \to aB$ на $A \to XB, X \to a$.
- Пример: $B \to (B)\ B \mid \varepsilon$
- Как мы могли бы распознать строку (()(()))?
- Благодаря свойствам нормальной формы, можно привести её в нетерминалы.

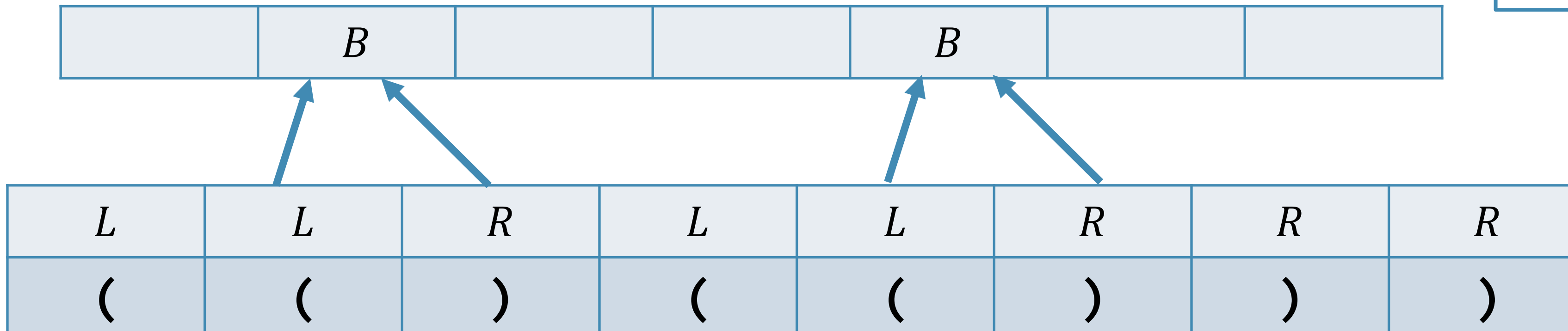
$$
\begin{aligned}
S &\to LZ_1 \mid LR \mid \varepsilon \\
B &\to LZ_1 \mid LR \\
Z_1 &\to BZ_2 \mid BR \\
Z_2 &\to RB \\
R &\to) \\
L &\to (
\end{aligned}
$$

L	L	R	L	L	R	R	R
(	(	)	(	(	)	)	)

Идея для алгоритма принадлежности

- Далее попробуем комбинировать подстроки длины 2, длины 3 и так далее.
- Пусть мы ищем возможный вывод $S \to LLRLLRRR$
- В этом случае мы должны рассматривать $X_i \to X_j X_k$
- У нас три параметра: начало диапазона, длина и продукция.

$$S \to LZ_1 \mid LR \mid \varepsilon$$
$$B \to LZ_1 \mid LR$$
$$Z_1 \to BZ_2 \mid BR$$
$$Z_2 \to RB$$
$$R \to)$$
$$L \to ($$



Принцип оптимальности

- Для алгоритма CYK принцип оптимальности примет следующий вид.

$V(1, i, j) = true$

$V(k, i, a) = \max\big(V(p, i, b)\ \wedge\ V(k-p, i+p, c)\big)$, где $X_a \to X_b X_c$

- Это сводится (снова) к заполнению двумерной таблицы. Но на этот раз мы должны рассматривать также диапазон индексов p от единицы до длины строки.
- Что сразу приводит нас к простому песвдокоду.

HWY – алгоритм Кока-Янгера-Касами

- Для строки с n символами и для k продукций, после начальной инициализации, алгоритм следующий:

```
for (l = 2; i < n; ++l)
  for (s = 1; s < n - l + 1; ++s)
    for (p = 1; p < l - 1; ++p)
      for each production Xa -> Xb Xc
        V[l, s, a] = max(V[l, s, a],
                         V[p, s, b] && V[l - p, s + p, c]);
```

- Реализуйте это на языке C.
- Если в итоге `V[n, 1, 1] = true` то слово принадлежит языку.

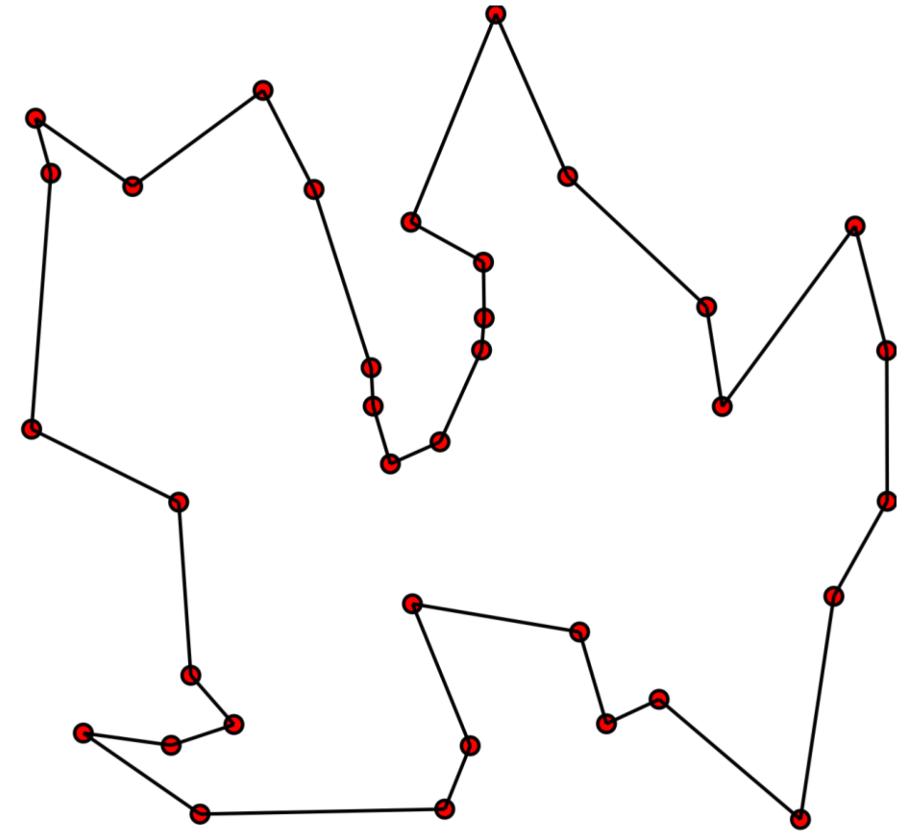
https://en.wikipedia.org/wiki/CYK_algorithm

Формирование инутиции

- Не все задачи можно эффективно решить динамическим программированием.
- Пример: traveling salesman problem.
- Можно ли написать функциональное уравнение Беллмана? В общем можно.

$$V(S,k) = \min(V(S - \{k\}, m) + d(m,k))$$

- Увы, решение при этом будет $O(n^2 2^n)$ по времени и по памяти.



https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm

Литература

- [C11] ISO/IEC – "Information technology – Programming languages – C", 2011
- $[K\&R]$ Brian W. Kernighan, Dennis Ritchie – The C programming language, 1988
- $[Bellman]$ Richard Bellman – Dynamic Programming, 1957
- $[Cormen]$ Thomas H. Cormen – Introduction to Algorithms, 2009
- $[TAOCP]$ Donald E. Knuth – The Art of Computer Programming, 2011
- $[SALG]$ Robert Sedgewick Algorithms, 4th edition, 2011