ВРЕМЯ И ПАМЯТЬ

Асимптотическая сложность алгоритмов. Время и память как ресурсы.
Плюс нечто о битовых операциях.

К. Владимиров, Syntacore, 2023
mail-to: konstantin.vladimirov@gmail.com

СЕМИНАР 2.1

Простые числа и структуры данных

Warmup: простые числа

- Определение.

$unit\ x \iff \exists u, ux = 1$

$prime\ p \iff \nexists x, x \neq u \cap x \neq pu \cap x \backslash \mathrm{p}$

- Мнемоническое правило "делится только на единицу и на само себя".
- Но строго говоря в целых числах units это 1 и -1.
- Значит реально ещё на -1 и на минус само себя.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Алгоритм P

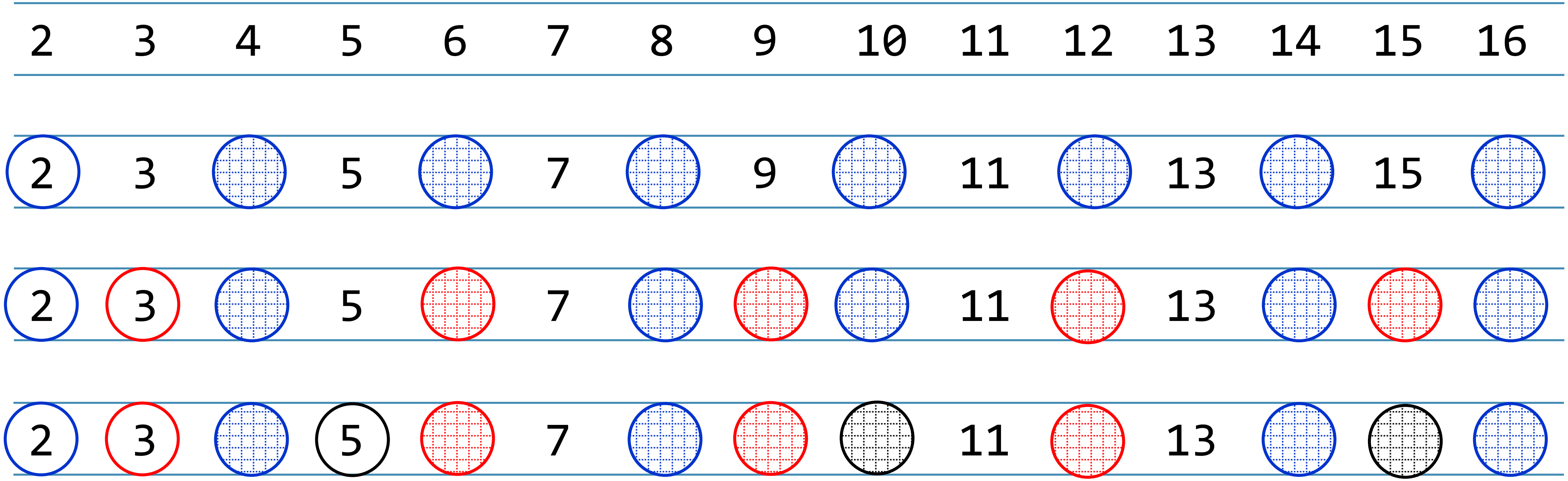
- Простейший способ определить, является ли число простым.

```
int is_prime(unsigned n) {
  if (n < 2) return 0;

  for (int j = 2; j * j <= n; ++j)
    if ((n % j) == 0)
      return 0;

  return 1;
}
```

Problem PN – N-ное простое число

- Первым простым числом является 2, шестым является 13.
- Используйте алгоритм P чтобы вычислить N-е простое число.

Простые числа: решето Эратосфена



- Вычеркиваются все числа кратные каждому простому. Следующее простое это ближайшее невычеркнутое.

Минимум о динамической памяти

- Динамическую память можно выделять и освобождать по необходимости.

```
int *p = malloc(10 * sizeof(int)); // не инициализирована
int *q = calloc(10, sizeof(int)); // обнулена
```

- Здесь мы выделили массив из десяти целых чисел.

```
p[0] = 1; p[9] = 15; // ok, но p[10] будет ошибкой
```

- Теперь нам необходимо освободить память.

```
free(p); free(q);
```

- В этой точке p указывает в никуда, под ним нет памяти.

Problem SE – использование решета

- На входе N.
- На выходе вам предлагается напечатать решето Эратосфена от 2 до N.
- Печатайте 0 если число простое и 1 если составное.
- Пример: для N = 11.

$0\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ 0$

- Используйте динамическую память, не используйте VLA.

Минимум о структурах

- Структура задаётся ключевым словом struct и содержит поля разных типов

```
struct S { int x; int y; char z; };
```

- Объект структуры можно инициализировать при первом определении

```
struct S t = {1, 2, 'a'};
```

- Доступ к полям структуры делается через точку

```
t.x = t.y + 1;
assert(t.x == 3);
```

- Возможен указатель на структуру, тогда обращаемся через стрелку.

```
struct S *pt = &t; assert(pt->x == 3);
```

Problem TS – площадь треугольника

- Вам задан треугольник с целыми координатами. Его удвоенная площадь это целое число.
- Структура для треугольника.

```
struct point_t { int x, y; };

struct triangle_t { struct point_t pts[3]; };
```

Ваша задача: найти его площадь. Это очень простая задача.

```
int double_square(struct triangle_t tr);
```

Проектирование решета

- Решето это объединение его размера с указателем на память.

```
struct sieve_t {
  unsigned size;
  unsigned char *sieve;
};
```

- Использование

```
struct sieve_t s = init_sieve(100); // для чисел от 0 до 100

assert(s.sieve != NULL && s.size > 0);

int is63 = is_prime(s, 63); // проверяем простое ли число 63
```

Освобождение памяти

- После использования решета следует освободить выделенную через `calloc` память

```
void free_sieve(struct sieve_t *s) {
  free(s->sieve);
  s->sieve = 0;
  s->size = 0;
}
```

- Для надёжности мы занулили указатель
- Все ли понимают почему сюда решето пришло по указателю?

Проверка с помощью решета

- Поскольку решето содержит `1` для составных и `0` для простых, на проверке, надо инвертировать логику

```
unsigned is_prime(struct sieve_t s, unsigned n) {
  assert(n < s.size);
  return (s.sieve[n] == 1) ? 0 : 1;
}
```

- Использован тернарный оператор `a ? b : c`
- Означает "если a, то b, иначе c". Более короткая форма, когда не хочется писать `if`.

Сводим всё вместе

- Выделить решето на `100` элементов

```
struct sieve_t s = init_sieve(100);
```

- Проверить число 97 с помощью решета

```
assert(is_prime(s, 97) == 1);
```

- Освободить решето

```
free_sieve(&s);
```

Problem PS – снова N-е простое

- Чтобы вычислить N-е простое число для `N > 20`, с помощью решета, нужно построить решето до числа $N(\log N + \log\log N)$

```
unsigned long long sieve_bound(unsigned num) {
  assert(num > 20);
  double dnum = num;
  double dres = dnum * (log(dnum) + log(log(dnum)));
  return (unsigned long long) round(dres);
}
```

- Сравните результаты с результатами задачи PN
- Замерьте до какого числа вы сможете дойти за 60 секунд

Problem GF* – генерирующие формулы

- Эйлер открыл потрясающую формулу $n^2 + n + 41$.
- Она генерирует для $0 \leq n \leq 39$, `40` последовательных простых чисел.
- Ещё более потрясающая формула $36n^2 - 810n + 2753$ генерирует 45 последовательных простых (http://oeis.org/A050268)
- Используйте компьютер, чтобы рассмотреть все формулы, вида $n^2 + an + b, |a| < 1000, |b| < 1000$
- Какую самую длинную последовательность простых чисел вы сможете сгенерировать?
- Оптимизируйте алгоритм, отсекая заведомо плохие b (для `n=0`, число сразу должно быть простым).

Problem CC* – циркулярные простые

- Число 197 называется циркулярным простым, поскольку простыми являются все циклические перестановки его разрядов: 197 $\rightarrow$ 971 $\rightarrow$ 719
- Необходимо для заданного числа N определить ближайшее к нему циркулярное простое. Например для числа `200` ближайшим циркулярным простым будет 197
- Подумайте можно ли легко понять какого размера решето вам нужно?
- В зависимости от математических свойств циркулярных простых чисел (если они встречаются часто) проверка алгоритмом P может быть эффективнее решета. Ожидаете ли вы найти ближайшее циркулярное простое к `200000` достаточно близко, чтобы решето не окупалось?
- Подумайте о решении, которое будет комбинировать алгоритм P и решето

*по мотивам project Euler, 35

Снова алгоритм P

- Как оценить время работы этой функции?

```
int is_prime(unsigned n) {
  if (n < 2) return 0;

  for (int j = 2; j * j <= n; ++j)
    if ((n % j) == 0)
      return 0;

  return 1;
}
```

Снова алгоритм P

- Самый очевидный способ – подсчитать

```
int is_prime(unsigned n) {            // t1
  if (n < 2) return 0;                // t2

  for (int j = 2; j * j <= n; ++j)  // sqrt(n) * t3
    if ((n % j) == 0)                 // sqrt(n) * t4
      return 0;

  return 1;
}
```

Обсуждение

- Итак, время исполнения алгоритма P составляет $k_1 + k_2\sqrt{n}$
- Здесь n это проверяемое нами на простоту число.
- От чего зависят значения k_1 и k_2?

Обсуждение

- Итак, время исполнения алгоритма F составляет $k_1 + k_2 n$
- Здесь n это проверяемое нами на простоту число.
- От чего зависят значения k_1 и k_2?
  - Быстродействие компьютера (laptop vs supercomputer)
  - Архитектура микропроцессора, в частности насколько дорогой branch и какие там внутри детали реализации подсистем памяти и арифметики/логики
  - Качество компилятора и линкера (насколько оптимизирован код)
- На практике мы часто не знаем и знать не можем большую часть этих параметров.
- Но мы всегда знаем наш **главный параметр** n.

Снова наивный подход

- Для примера оценим выполнение чисел Фибоначчи при наивном подходе

```
unsigned long long fib (unsigned n) {
  if (n == 0) return 0ull;                    // t_a ϕ^n
  if (n <= 2) return 1ull;                    // t_a ϕ^n
  return fib(n - 1) + fib(n - 2);             // (t_b + t_c) ϕ^n
}
```

- Имеем ровно $\frac{1}{\sqrt{5}}\phi^n$ вызовов функции и общее время $k_3\phi^n$
- Первая мысль при сравнении $k_1 + k_2 n$ против $k_3\phi^n$ это: а есть ли вообще разница какие значения имеют k_1, k_2, k_3?
- При достаточно большом n, всегда $k_1 + k_2 n < k_3\phi^n$

O-нотация

- Базовая интуиция, что при достаточно большом n, выполняется

$$k_6 < k_4 + k_5 \log(n) < k_1 + k_2 n < k_3 1.61^n$$

- Получает своё развитие в O-нотации

$$f(n) = O\big(g(n)\big) \leftrightarrow \exists k, M \mid \forall n > k, M \cdot g(n) \geq |f(n)|$$

О-нотация

- Базовая интуиция, что при достаточно большом n, выполняется

$$k_6 < k_4 + k_5 \log(n) < k_1 + k_2 n < k_3 1.61^n$$

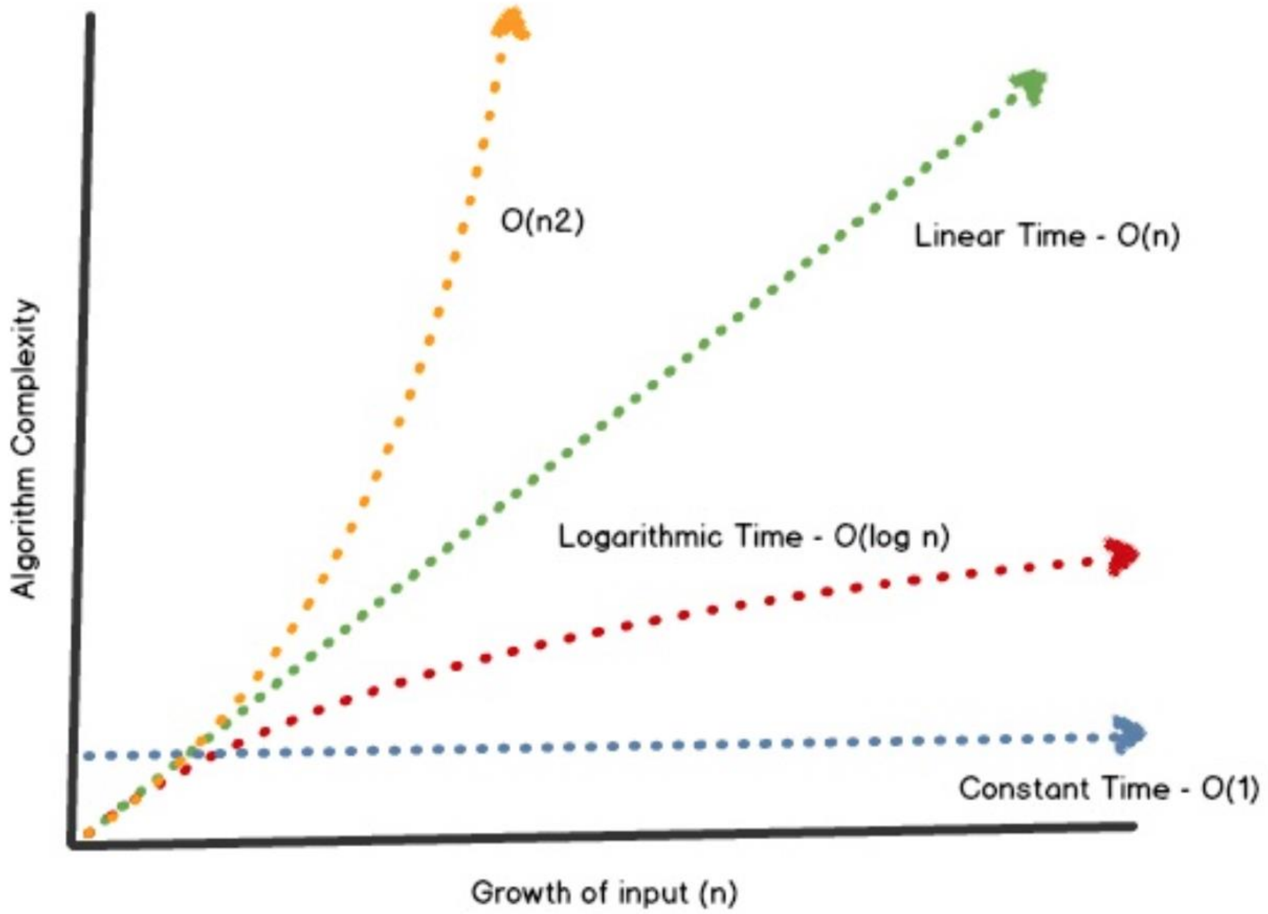
- Получает своё развитие в О-нотации

$$f(n) = O\big(g(n)\big) \leftrightarrow \exists k, M \mid \forall n > k, M \cdot g(n) \geq |f(n)|$$

- Например $2x^3 + 14x^2 + 87x + 3 = O(x^3)$
- О-нотация не слишком строгая. Например то же выражение это также $O(x^4)$
- Говорят, что О-нотация отражает асимптотику зависимости ресурса (например времени работы алгоритма) от главного параметра в задаче (например номера числа Фибоначчи)

O-нотация

	n	$nlog(n)$	n^2	2^n
10	1	1	1	1
50	1	1	1	13д
10^6	1	1	15м	∞
10^{10}	10с	2м	3г	∞
10^{14}	2ч	28ч	∞	∞



Алгоритм P, первое приближение

- Простейший способ определить, является ли число простым

```
int is_prime(unsigned n) {

  if (n < 2) return 0;

  for (int j = 2; j * j <= n; ++j)
    if ((n % j) == 0)
      return 0;

  return 1;
}
```

- Асимптотическая сложность $O(\sqrt{n})$. Кажется, этот алгоритм можно улучшить

Алгоритм P, второе приближение

- Используем тот факт, что чётные всегда не простые кроме 2

```
int is_prime(unsigned n) {
  if (n == 2) return 1;
  if ((n < 2) || ((n % 2) == 0)) return 0;

  for (int j = 3; j * j <= n; j += 2)
    if ((n % j) == 0)
      return 0;

  return 1;
}
```

- Скорость превосходит первое приближение вдвое. Асимптотика?

isprime-improved.c

Алгоритм P

- Используем тот факт, что простые всегда имеют вид $6k \pm 1$

```
int is_prime(unsigned n) {
  if ((n == 2) || (n == 3)) return 1;
  if ((n < 2) || ((n % 2) == 0) || ((n % 3) == 0)) return 0;

  for (int j = 5; j * j <= n; j += 6)
    if (((n % j) == 0) || ((n % (j + 2)) == 0))
      return 0;

  return 1;
}
```

- Скорость превосходит первое приближение втрое. Асимптотика?

isprime-advanced.c

Суть асимптотики

- Асимптотическая сложность не измеряет время выполнения задачи.
- Она измеряет то, как изменяется время выполнения при изменении входных данных

Обсуждение

- Пока что речь шла только об одном ресурсе – времени
- Но бывают и другие ресурсы. Ваши предположения?

Другие ресурсы

- Пока что речь шла только об одном ресурсе – времени
- Но бывают и другие ресурсы:
  - Память
  - Объём пересылаемых по сети данных
  - Сложность разработки в человеко-часах
  - Стоимость лицензий для подключаемых сторонних библиотек
  - Энергопотребление компьютера
- Память выделена потому что это второй по важности ресурс после времени
- Разумеется память это ресурс только в языках с явным управлением памятью. К счастью язык C из этих

Обсуждение

- Вам надо часто искать N-е простое для $0 < N < M$.
- Оцените асимптотику решета Эратосфена по памяти.

Упражнения с асимптотикой

- Оцените асимптотику следующих выражений

$$n + \log(n) + \sin(n)$$

$$5^{\log_2 n} + n^2\sqrt{n}$$

$$n^{100} + 1.1^n$$

- Расположите выражения по возрастанию порядка роста

3^n	$n \log_2 n$	$\log_4 n$	n	$2^{\log_5 n}$	n^2	$\sqrt{n}$	2^{2n}

Problem LM – наименьшее кратное

- Число 2520 является наименьшим числом, которое делится без остатка на числа от 2 до 10
- Задача состоит в том, чтобы найти наименьшее число, которое делится без остатка на числа от 2 до N
- Вам предлагают наивный алгоритм: идти от числа N вверх и каждое встретившееся число проверять для каждого из N чисел. (см. lmnaive.c)
- **Оцените асимптотику наивного алгоритма**
- Подумайте, можно ли использовать алгоритм E для лучшего решения этой задачи?
- Математический инсайт: $lcm(a,b) = \frac{ab}{\gcd(a,b)}$ и $lcm(a,b,c) = lcm(lcm(a,b),c)$

СЕМИНАР 2.2

Побитовая арифметика

Мотивация

- Наше решето с хранением признака в каждом байте кажется экономным.

0	0	1	0	1	....

- Но давайте посмотрим правильный масштаб?

00000000	00000000	00000001	00000000	00000001	....

Побитовое представление

- Минимальной адресуемой единицей в языке C является байт.

```
int x = 0x10176F4A;
```

00010000	00010111	01101111	01001010

- Но что если есть необходимость работать с отдельными битами внутри байта?
  - Установить бит под номером n в числе x в значение `0` или `1`.
  - Считать значение бита под номером n в числе x.
  - Инвертировать бит под номером n в числе x.
- Для этого проще всего использовать побитовую арифметику.

Всевозможные битовые операции

x	y	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

- Всего возможно 16 битовых операций над двумя аргументами.
- Найдите среди приведённых таблиц истинности знакомые. Например какой номер имеет логическое "и" (конъюнкция)? Какой номер имеет "или" (дизъюнцкия)? Как вы истрактуете остальные операции?

Основные битовые операции в языке

- В языке C представлено всего четыре базовые битовые операции
- Также есть сдвиги (правый и левый)

$$(x \ll n) == x * 2^n$$
$$(x \gg n) == x / 2^n$$

- К сожалению нет тернарной медианы
- Но `<x, y, z>` всегда можно собрать из примитивных операций

<table>
<tr><th>Операция</th><th>Название</th><th>Таблица истинности</th></tr>
<tr><td>x & y</td><td>коньюнкция</td><td>0 0 0 1</td></tr>
<tr><td>x | y</td><td>дизъюнкция</td><td>0 1 1 1</td></tr>
<tr><td>~x</td><td>отрицание</td><td>1 0 * *</td></tr>
<tr><td>x ^ y</td><td>исключающее или</td><td>0 1 1 0</td></tr>
<tr><td>~x | y</td><td>импликация</td><td>1 1 0 1</td></tr>
<tr><td>~(x & y)</td><td>штрих Шеффера</td><td>1 1 1 0</td></tr>
<tr><td>~(x | y)</td><td>стрелка Пирса</td><td>1 0 0 0</td></tr>
</table>

```
median(x, y, z) == (x | y) & (y | z) & (x | z)
```

Тренируемся в битовой арифметике

- Подсчитайте очень быстро в уме (не используйте компьютеры)

```
0xAC & 0x28 = ?

0xE2 | (~0xEF) = ?

075 & 063 = ?

0b10100 ^ 0xFF = ?

66 ^ 18 = ?

0x23 >> 2 = ?

0x23 << 4 = ?
```

Операции в битовой арифметике

- Установить бит под номером n в числе x в значение `1`

```
x = x | (1u << n);
```

- Установить бит под номером n в числе x в значение `0`

```
x = x & ~(1u << n);
```

- Считать значение бита под номером n в числе x

```
val = (x >> n) & 1u;
```

- Инвертировать бит под номером n в числе x

```
x = x ^ (1u << n);
```

Длинные и короткие операции

- Важно не путать длинные (логические) с короткими (побитовыми) операциями

```
unsigned char c = 0x78;
if (c && 1) { printf("long"); } // логическое "и"
if (c & 1) { printf("short"); } // побитовое "и"
```

- Точно также у нас есть длинное (логическое) или: `a || b`
- Точно также у нас есть логическое не: `!c == false`, но `~c == 0x87`
- Какая логическая операция соответствует побитовому исключающему или?

Problem BP -- старший и младший бит

- Вам предлагается найти позицию старшего установленного бита в числе. Также найдите позицию младшего установленного бита.
- Пример.
- Вход: 269971274
- Выход: 28 2

Problem BI -- индекс бита

- Вам предлагается инвертировать определённый по счёту бит в массиве.

00000011	00000011	00000011	00000100
31 24	23 16	15 8	7 0

- Пример.
- Вход: 4 4 3 3 3 17
- Выход: 4 3 1 3

Problem PE – побитовое решето

- Сейчас решето Эратосфена, которое строит алгоритм S, хранит `unsigned char` (то есть 8 бит) на каждый признак простоты числа. Это немыслимый расход памяти
- Вам предлагается оптимизировать построение решета таким образом, чтобы признак того является ли число простым хранился в каждом бите решета
- Это позволит сократить расход памяти в 8 раз
- Разумеется это несколько усложнит функции `init_sieve` и `is_prime`
- Подумайте о тестировании вашего решета

Домашняя работа HWE

- Результаты, полученные в Problem PE, могут быть улучшены далее: память можно сократить ещё в два раза, если хранить в каждом бите только признаки простоты для нечётных чисел
- На самом деле, память можно сократить ещё в полтора раза, если хранить два массива: первый для всех $(6k - 1)$-ых а второй для всех $(6k + 1)$-ых битов

```
struct sieve_t {
  unsigned size;
  unsigned char *plus1; // for 7, 13, 19, ....
  unsigned char *minus1; // for 5, 11, 17, ....
};
```

- Реализуйте такое решето. Поможет ли оно вам найти миллиардное простое?

Постановка задачи: popcount

- Допустим у вас есть задача подсчитать количество установленных бит в числе. Её решение циклом довольно просто.

```
count = 0;
for (int i = 0; i < sizeof(x) * CHAR_BIT; ++i)
  count += (x >> i) & 1;
```

- Можно ли это решение улучшить?

Р-адические числа

$\dots 0000010 = 2$

$\dots 0000001 = 1$

$\dots 0000000 = 0$

Чему равен -1?

P-адические числа

$\ldots 0000010 = 2$

$\ldots 0000001 = 1$

$\ldots 0000000 = 0$

$\ldots 1111111 = -1$

$\ldots 1111110 = -2$

$\ldots 1111101 = -3$

$\ldots 1111100 = -4$

- Чему равно $x + \bar{x}$?

P-адические соотношения

$\ldots 0000010 = 2$

$\ldots 0000001 = 1$

$\ldots 0000000 = 0$

$\ldots 1111111 = -1$

$\ldots 1111110 = -2$

$\ldots 1111101 = -3$

$\ldots 1111100 = -4$

$x + \bar{x} = -1$

$(x - 1) + \overline{x - 1} = -1$

$\bar{x} + 1 = \overline{x - 1}$

$-x = \bar{x} + 1 = \overline{x - 1}$

$\boxed{-x = \overline{x - 1}}$

- Чему равно $x \mathbin{\&} (x - 1)$?

Почти всегда истинные рассуждения

- Рассмотрим

$x = (\alpha 01^a 10^b)$

- Тогда

$\bar{x} = (\bar{\alpha} 10^a 01^b)$

$x - 1 = \left(\alpha 01^a 01^b\right)$

$-x = (\bar{\alpha} 10^a 10^b)$

$x + \bar{x} = -1$

$(x - 1) + \overline{x - 1} = -1$

$\bar{x} + 1 = \overline{x - 1}$

$-x = \bar{x} + 1 = \overline{x - 1}$

$-x = \overline{x - 1}$

$\boxed{x \mathbin{\&} (x - 1) = (\alpha 01^a 00^b)}$

$x \mathbin{\&} (x - 1)$ это x без LSB

popcount с хитрым трюком

- Допустим у вас есть задача подсчитать количество установленных бит в числе. Её решение циклом довольно просто.

```
count = 0;
for (int i = 0; i < sizeof(x) * CHAR_BIT; ++i)
  count += (x >> i) & 1;
```

- Теперь мы можем написать так:

```
count = 0;
for (int i = x; i > 0; i = i & (i - 1))
  count += 1;
```

- Это почти волшебство. Мы идём только по тем числам по которым должны.

Новые трюки на ваш вкус

$x = (\alpha 01^a 10^b)$

$\bar{x} = (\bar{\alpha} 10^a 01^b)$

$x - 1 = \left(\alpha 01^a 01^b\right)$

$\bar{x} = (\bar{\alpha} 10^a 01^b)$

$-x = (\bar{\alpha} 10^a 10^b)$

Удаление LSB

$x \,\&\, (x - 1) = (\alpha 01^a 00^b)$

Вычленение LSB

$x \,\&\, (-x) = (000^a 10^b)$

Распространение LSB

$x \wedge (x - 1) = (000^a 11^b)$

Перебор подмножеств

- Допустим мы хотим распечатать все подмножества множества $\{0, 3, 5, 6\}$.

$\{0\}, \{3,0\}, \{5,0\}, \{6,0\}, \{5,3\}, \{6,3\}, \{6,5\}, \{5,3,0\}, \{6,3,0\}, \{6,5,0\}, \{6,5,3\}, \{6,5,3,0\}$

- Как научить этому компьютер если максимальное $N < 32$?
- Рассмотрим $M = (1101001)_2$. Здесь выставлены биты с номерами 6,5,3,0.
- Тогда от $x = LSB$ шаг $x = (x - M)\ \&\ M$ даёт нам следующее подмножество.

$x = 1, x = (1\ - 69)\ \&\ 69 = 9, x = (9\ - 69)\ \&\ 69 = 33$ и т. д.

Алгоритм BSUB

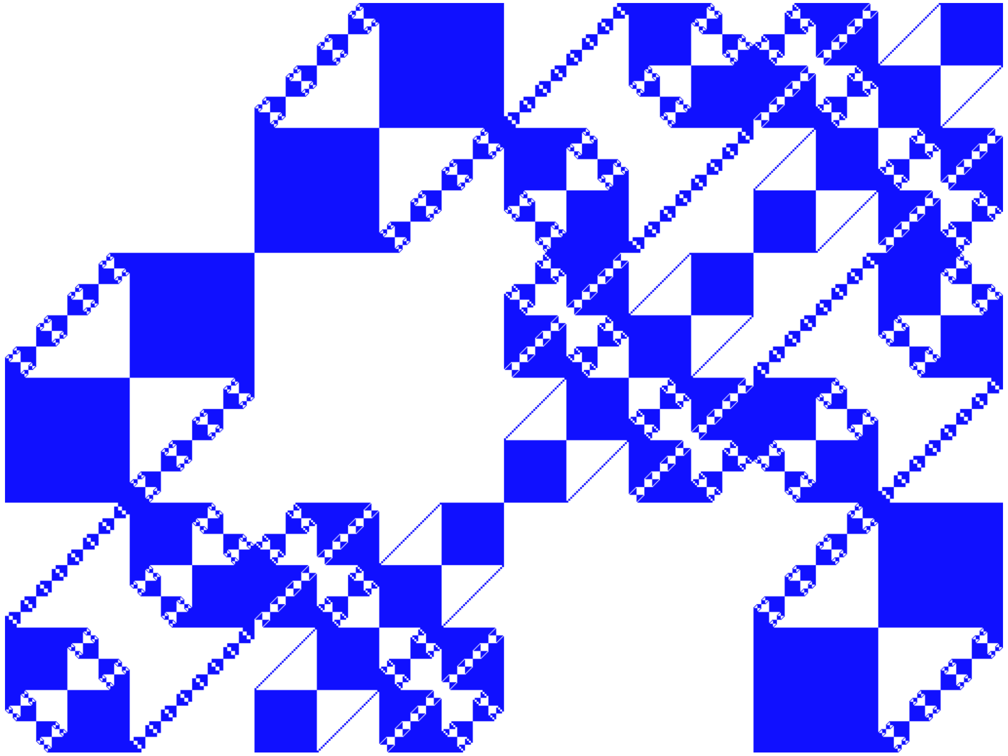
- Перебор всех подмножеств битового множества $x \to mask \ \& \ (x - mask)$

```
void all_proper_subsets(unsigned mask) {
  unsigned x = 1;
  while (x != mask) {
    visit(x);
    x = (x - mask) & mask;
  }
}
```

- Писать настоящий перебор подмножеств конечно куда сложнее

Ковры Кнута

- Булевы функции интересно отображать коврами ($TAOCP$, 7.1.3)
- Ковёр это функция от двух переменных, принимающая значения нуля и единицы
- Закрашивая белым и синим и смешивая арифметику, можно получить интересные паттерны
- Справа $\left((x \oplus y)^2 \gg 17\right) \,\&\, 1$



funcbitmap.c

СЕМИНАР 2.3

Случайность и время.

Обсуждение

- Когда начинается наше время?

Наше время

- 0 секунд было 1 января 1970 года (Unix time).
- Диапазон от $-(2^{31} - 1)$ до $(2^{31} - 1)$ сек.
- То есть от 13 декабря 1901 года до 19 января 2038 года.
- Но **почему** это так?

Наше время

- 0 секунд было 1 января 1970 года (Unix time).
- Диапазон от $-(2^{31}-1)$ до $(2^{31}-1)$ сек.
- То есть от 13 декабря 1901 года до 19 января 2038 года.
- Но **почему** это так?
- Дело в том, что в 1970м году, в машине `PDP-11` для хранения времени был выбран тип, размером 4 байта.
- С тех пор человечество целиком так и не перешло на 8-байтное время. Но мы в процессе.
- Вопрос к математически настроенной аудитории: надолго ли хватит 8 байт?

Работа со временем

- Основная функция:

```
time_t time(time_t *arg); // <time.h>
```

- Это интегральное значение, часто обозначающее количество секунд с начала эпохи. Возвращает результат и его же пишет в аргумент если не NULL.
- Дополнительные функции:

```
struct tm *gmtime(const time_t *timer); // epoch to UTC
char* asctime(const struct tm *time_ptr); // UTC to string
```

- Структура tm содержит поля для секунд, минут, часов и т. д.

https://godbolt.org/z/Ghrcs9GxY

Что если нужны точные замеры?

- Секунды с начала эпохи это так себе гранулярность. Иногда нам нужно мерить микро и даже нано секунды.

```
struct timespec; // содержит tv_sec и tv_nsec
```

```
int timespec_get(struct timespec *ts, int base);
```

- Только начиная с C11. Это не совсем совместимо с ранним версиями C. В ранних версиях можно использовать clock_gettime из POSIX.

```
struct timespec ts; struct tm *ptm;
```

```
timespec_get(&ts, TIME_UTC);
ptm = gmtime(&ts.tv_sec); // наносекунды в ts.tv_nsec
```


https://godbolt.org/z/96bnsGrfE

Расстояние между timestamps

```
const int MICROSEC_AS_NSEC = 1000, SEC_AS_MICROSEC = 1000000,
          SEC_AS_NSEC = 1000000000;

// возвращает double: нечто вроде 5.2071 секунд
double diff(struct timespec start, struct timespec end) {
  struct timespec temp;
  if (end.tv_nsec - start.tv_nsec < 0) {
    temp.tv_sec = end.tv_sec - start.tv_sec - 1;
    temp.tv_nsec = SEC_AS_NSEC + end.tv_nsec - start.tv_nsec;
  } else {
    temp.tv_sec = end.tv_sec - start.tv_sec;
    temp.tv_nsec = end.tv_nsec - start.tv_nsec;
  }
  double msec = temp.tv_sec * SEC_AS_MICROSEC + temp.tv_nsec / MICROSEC_AS_NSEC;
  return msec / SEC_AS_MICROSEC;
}
```

Обсуждение

- Чтобы протестировать большое число на простоту, построение решета бывает затруднительно. Например чтобы проверить $2^{50} + 7$, решето должно занимать много терабайт даже после всех оптимизаций.
- Наивный алгоритм работает за $O(\sqrt{N})$ но если M это количество бит в числе, то сложность уже $O(2^{M/2})$.
- Иногда нам может помочь рандомизированный алгоритм.

Тест Ферма

- Математический инсайт: малая теорема Ферма

$a^{p-1} \equiv 1\ (p)$ если *p* простое и *a* не делится на *p*

- К сожалению для многих составных n, $\exists a$, $a^{n-1} \equiv 1\ (n)$ это Fermat liar

$38^{220} \equiv 1(221)$ но $24^{220} \equiv 81(221)$ значит 221 составное

- Такое a, что $a^{n-1} \neq 1\ (n)$ называется свидетелем (Fermat witness) непростоты, например 24 это witness для 221.
- Обычно свидетеля выбирают **случайно**.

Случайные числа

- Что делает последовательность предсказуемой?

$1, 2, 4, 8, \ldots$

Псевдослучайные числа

- По настоящему случайные числа довольно сложны. Вместо них используются **псевдослучайные**, то есть выглядящие как случайные, но сгенерированные детерминированно.

$77, 31, 18, 8, 60, 3, 86, 23, 79, 40, 10, 69, 92, 50, 55, 29, 9, 16, 96, 68, 39, 54, \ldots$

- Сможете ли угадать следующее число?
- Как бы вы сгенерировали такую последовательность?

https://godbolt.org/z/Gb4KnEvG4

Линейные конгруэнтности

- Основная идея для простых псевдослучайных чисел это линейная функция вида:

$x_{n+1} = (ax_n + b)\ \%\ m$

- В некоторых случаях функции получаются не слишком интересные:

$a = 33, b = 24, m = 31, x_0 = 1$ порождает: $26, 14, 21, 4, 1, 26, 14, 21, 4, 1, \dots$

- Давайте выберем функцию с периодом подлиннее?

https://godbolt.org/z/eKK1Esoez

rand и time

- В языке C псевдослучайные числа проще всего генерировать функцией `rand()`.
- Эта функция возвращает равномерное число от 0 до $RAND\_MAX$. Чтобы получить число от 0 до $N-1$, достаточо подсчитать `rand() % N`.
- Чтобы не получать от запуска к запуску одинаковые числа, можно однократно в начале программы задать seed через функцию `srand`.
- Обычным источником seed является текущее время в микросекундах с `1970` года: `srand(time(NULL))`.

Problem FT – тест Ферма

- Реализуйте тест Ферма.
- Для тестирования можно использовать решето и небольшие простые.
- Некоторые числа (они называются числами Кармайкла) не имеют свидетелей, а только лжецов и для них тест Ферма не работает. Сколько таких чисел вы нашли во время тестирования?

Problem PF* – Фибоподобные простые

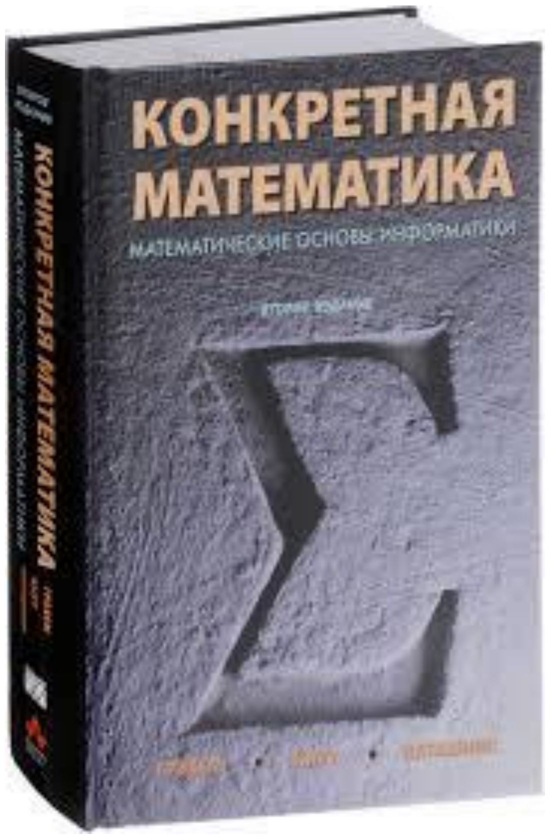
- Некоторые числа Фибоначчи, например 5 и 13 являются также простыми числами. Разумеется, список простых чисел Фибоначчи не слишком интересен, его легко нагуглить
- К счастью, в мире много других интересных последовательностей, похожих на числа Фибоначчи, например такая: $F_n = kF_{n-1} + nF_{n-2}$
- Ваша задача, получив на вход числа k и n вычислить самое большое простое число P, такое, что $P < 2^{64}$ и P входит в данную последовательность
- Например для $k = 1$ и $n = 1$ (т.е. для обычных чисел Фибоначчи) ответом является 99194853094755497
- Напишите программу, которая ищет ответ для любых $0 < k, n < 256$

Бенчмаркинг

- Опыт бенчмаркинга тестов на простоту.

Литература

- $[C11]$ ISO/IEC, "Information technology – Programming languages – C", ISO/IEC 9899:2011
- $[K\&R]$ Brian W. Kernighan, Dennis Ritchie – The C programming language, 1988
- $[KGP]$ Ronald L. Graham, Donald E. Knuth, Oren Patashnik – Concrete Mathematics: A Foundation for Computer Science, 1994
- $[TAOCP]$ Donald E. Knuth – The Art of Computer Programming, 2011
- Project Euler, problem 27: `https://projecteuler.net/problem=27`
- Prime formulas: `http://mathworld.wolfram.com/PrimeFormulas.html`



Секретные уровни

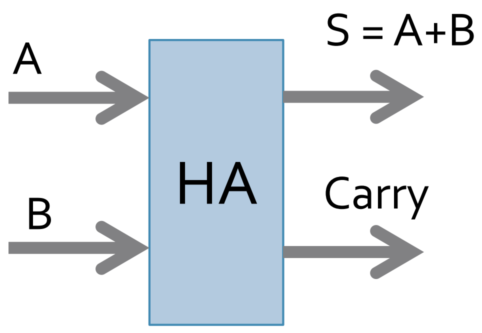
- Вы добрались до первого секретного уровня.
- Это как проблемы со звёздочкой, только это целые теоретические разделы со звёздочкой.
- Обычно они располагаются за списком литературы.
- Мы будем об этом говорить только если будет оставаться время или на допсеминарах.

ЦИФРОВАЯ ЛОГИКА

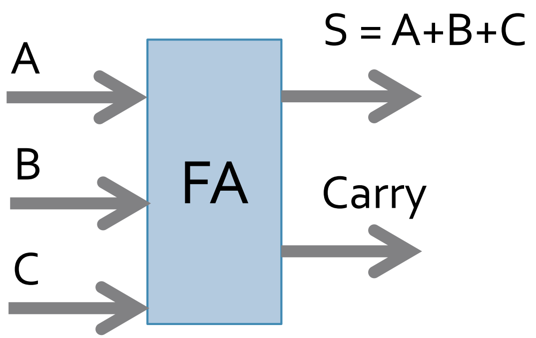
Основы работы цифровых машин

От булевой логики к арифметике

- Полусумматор (HA) это логическая схема, вычисляющая сумму и перенос для двух битов

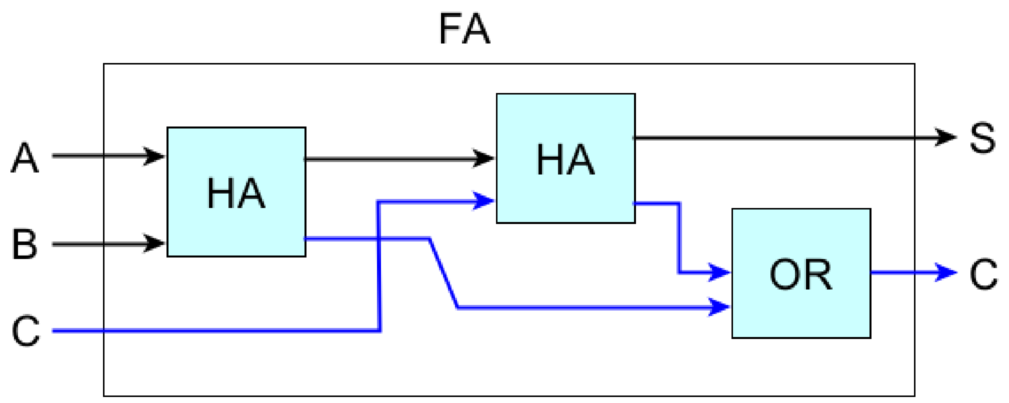


A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



- Очевидная реализация: S `= A xor B`, C `= A and B`
- Сумматор (FA): берет A, B, C, считает S и C. Как бы вы его реализовали?

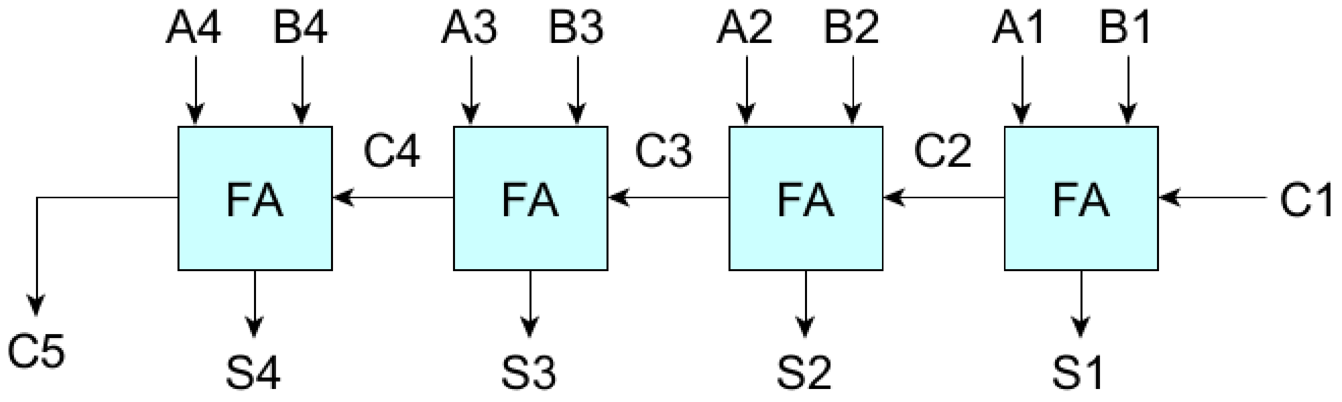
Полный сумматор



A	B	C	S	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- Теперь ясно почему полусумматор называется именно так.
- Как мы теперь построим сложение N-битных чисел?

Схема сложения



- Нравится ли вам эта схема?
- Какая сложность (по времени) сложения двух N-битных чисел?
- Как бы вы реализовали столь же наивное умножение?
- Для менее наивных подходов рекомендуется $[Mano]$

Дополнительная литература

- [$Savage$] John E. Savage – Models of Computation: Exploring the Power of Computing, 1998
- [$TAOCP$] Donald E. Knuth – The Art of Computer Programming (Vol 4a), 2011
- [$Mano$] M. Morris Mano – Logic and Computer Design Fundamentals, 5th edition, 2015
- Introduction to Digital Design and Integrated Circuits, https://inst.eecs.berkeley.edu/~eecs151/sp18, 2018