ВРЕМЯ И ПАМЯТЬ

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

К. Владимиров, Intel, 2021
mail-to: konstantin.vladimirov@gmail.com

Снова алгоритм F

- Как оценить время работы этой функции?

```
unsigned long long fib (unsigned n) {
  unsigned long long first = 0ull;
  unsigned long long second = 1ull;
  int idx;
  if (n == 0) return 0ull;
  for (idx = 2; idx <= n; ++idx) {
    unsigned long long tmp = second;
    second = second + first;
    first = tmp;
  }
  return second;
}
```

Снова алгоритм F

- Самый очевидный способ – подсчитать

```
unsigned long long fib (unsigned n) {
  unsigned long long first = 0ull;        // ta + te
  unsigned long long second = 1ull;       // ta + te
  int idx;                                // ta
  if (n == 0) return 0ull;                // tb
  for (idx = 2; idx <= n; ++idx) {        // ta + n * (tc + tp) + tb
    unsigned long long tmp = second;      // n * (ta + te)
    second = second + first;              // n * (tp + te)
    first = tmp;                          // n * te
  }                                       // n * tb
  return second;                          // tb
}                                         // Итого: k1 + n * k2
```

Обсуждение

- Итак, время исполнения алгоритма F составляет $k_1 + k_2 n$
- Здесь n это номер вычисленного числа Фибоначчи
- От чего зависят значения k_1 и k_2?

Обсуждение

- Итак, время исполнения алгоритма F составляет $k_1 + k_2 n$
- Здесь n это номер вычисленного числа Фибоначчи
- 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)

Обсуждение

- Итак, время исполнения алгоритма F составляет $k_1 + k_2 n$
- Здесь n это номер вычисленного числа Фибоначчи
- 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 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 $k_1 + k_2 n < k_3\phi^n$

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

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

О-нотация

- Базовая интуиция, что при достаточно большом n, выполняется

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

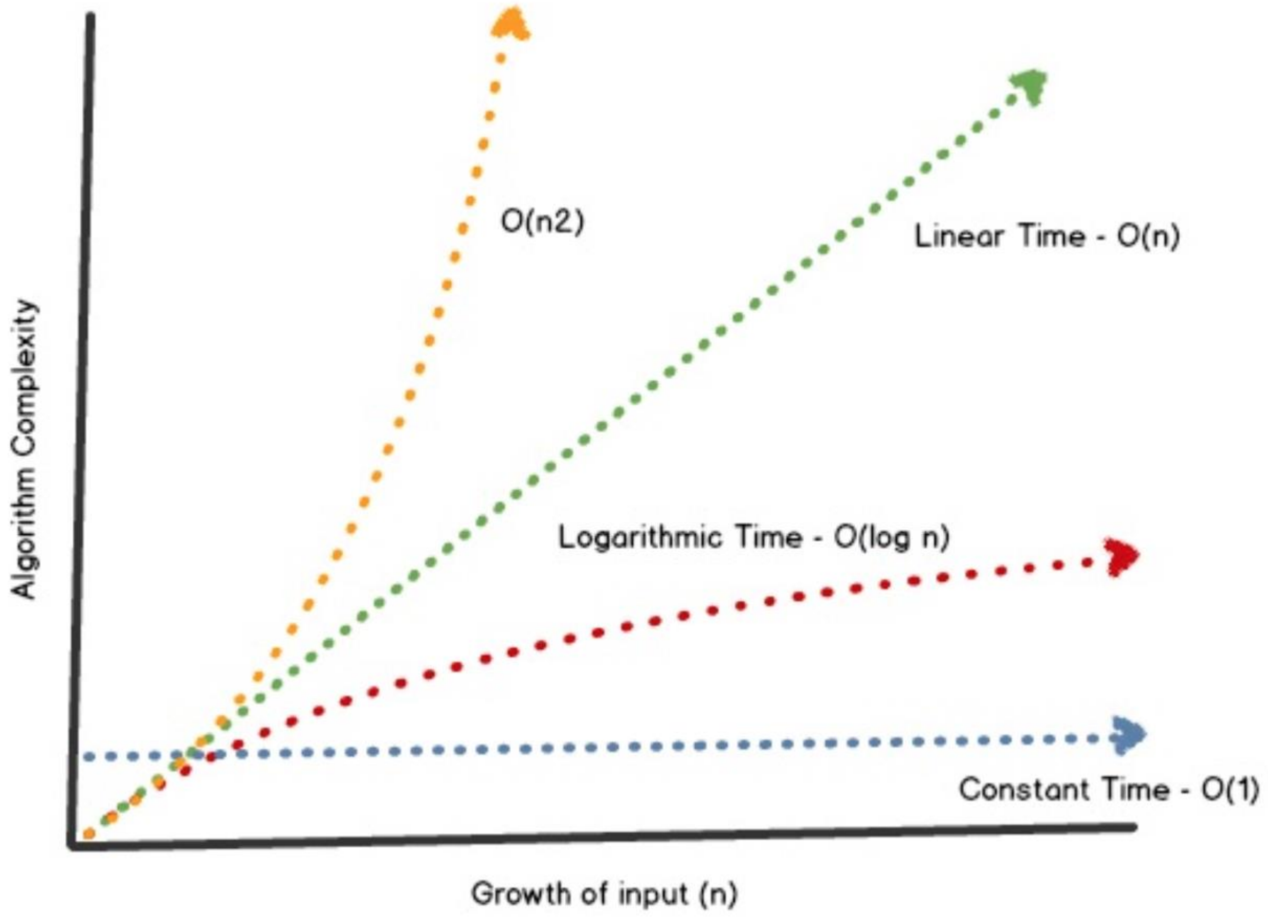
- Получает своё развитие в О-нотации

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

- 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	∞	∞



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 $lcm(a, b, c) = lcm(lcm(a, b), c)$

Warmup: простые числа

- Определение

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

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

- 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, 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;
}
```

- Оцените асимптотику этого алгоритма по времени

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

Обсуждение

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

Problem PN – N-ное простое число

- Первым простым числом является 2, шестым является 13
- Используйте алгоритм P чтобы вычислить N-е простое число
- Вычислите для начала N = 1000, потом увеличивайте.
- До какого числа вы сможете досчитать в пределах 60 секунд (используйте ulimit)?
- Сможете ли вы использовать для оптимизации идеи, использованные для оптимизации алгоритма P?
- Оцените асимптотику вашего решения

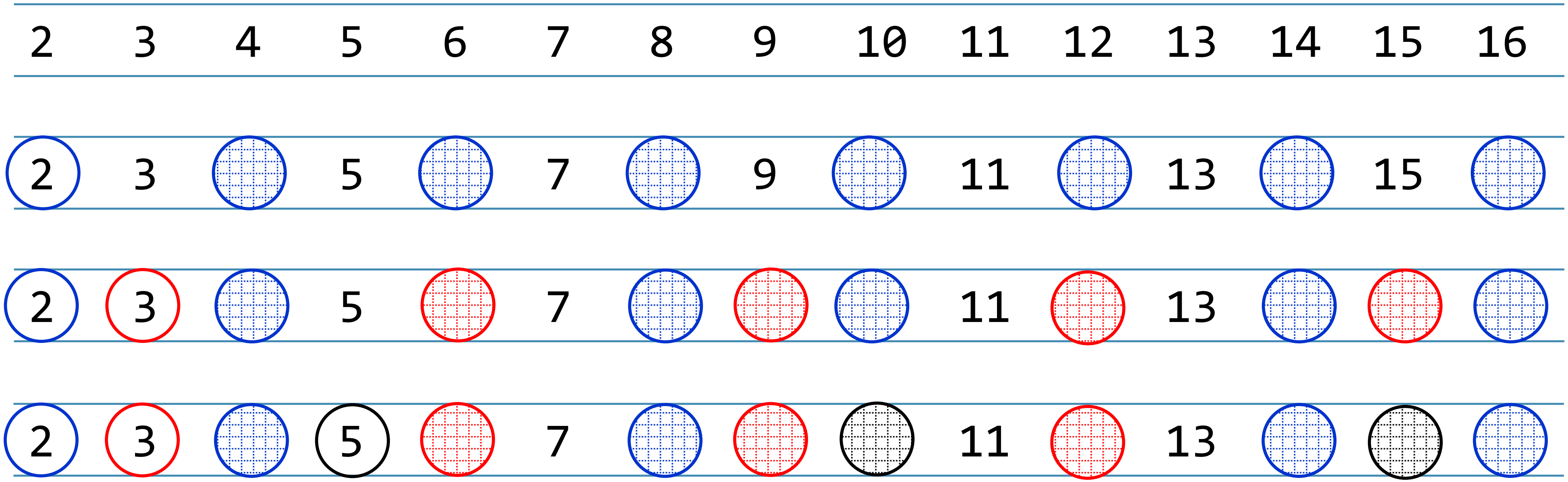
Обсуждение

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

Обсуждение

- 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 С из этих

Простые числа: решето Эратосфена



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

Проектирование решета: структуры

- Решето это объединение его размера с указателем на память. Объединения типов называются структурами.

```
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
```

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);
```

Алгоритм S: построение решета

```
struct sieve_t init_sieve (unsigned n) {
  unsigned char *sieve = calloc(n, sizeof(unsigned char));
  struct sieve_t res = { n, sieve };
  assert ((n > 1) && (sieve != NULL));

  unsigned r = (unsigned) sqrt (n) + 1;
  res.sieve[0] = res.sieve[1] = 1;

  // напишите здесь код, который для каждого i устанавливает
  // res.sieve[i] = 1, если число составное

  return res;
}
```


sieve.c

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;
}
```

- 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).

Работа с битами

- Минимальной адресуемой единицей в языке C является байт
- 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
- Можно использовать для этого обычную арифметику

```
x = (y / pow(2, n)) % 2; // x = nth bit value in y
```

- Но лучше и проще использовать специальную битовую арифметику

```
x = (y >> n) & 1; // x = nth bit value in y
```

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><code>x & y</code></td><td>коньюнкция</td><td>0 0 0 1</td></tr>
<tr><td><code>x | y</code></td><td>дизъюнкция</td><td>0 1 1 1</td></tr>
<tr><td><code>~x</code></td><td>отрицание</td><td>1 0</td></tr>
<tr><td><code>x ^ y</code></td><td>исключающее или</td><td>1 0 0 1</td></tr>
<tr><td><code>~x | y</code></td><td>импликация</td><td>1 1 0 1</td></tr>
<tr><td><code>~(x & y)</code></td><td>штрих Шеффера</td><td>1 1 1 0</td></tr>
<tr><td><code>~(x | y)</code></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 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?

Problem CC* – циркулярные простые

- Число 197 называется циркулярным простым, поскольку простыми являются все циклические перестановки его разрядов: 197 $\to$ 971 $\to$ 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

Тест Ферма

- 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.
- 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 или ищут перебором

Псевдослучайные числа

- По настоящему случайные числа довольно сложны. Вместо них используются **псевдослучайные**, то есть выглядящие как случайные, но сгенерированные детерминированно
- 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`
- Это не лучший, но распространённый способ
- Чтобы не получать от запуска к запуску одинаковые числа, можно 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

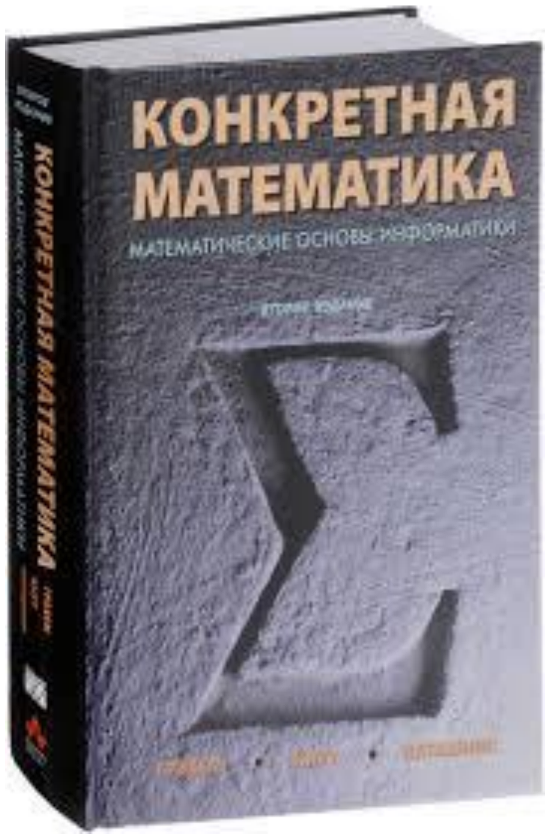
- Некоторые числа Фибоначчи, например 5 и 13 являются также простыми числами. Разумеется, список простых чисел Фибоначчи не слишком интересен, его легко нагуглить
- К счастью, в мире много других интересных последовательностей, похожих на числа Фибоначчи, например такая: $F_n = kF_{n-1} + nF_{n-2}$
- 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 $0 < k, n < 256$

Дальнейшие направления

- Работа с арифметическими объектами, такими как простые числа, может быть увлекательной и небезынтересной. Увы, не всё можно поместить в программу семинара
  - Улучшением теста Ферма является тест Миллера-Рабина, который можно даже сделать детерминированным
  - Потрясающе красивые задачи лежат в области факторизации чисел и дискретных логарифмов
  - Простые числа используются в шифровании, в том числе алгоритм RSA очень полезно знать каждому
  - Хорошие случайные числа для криптографии мистически связаны с гигантскими простыми числами Мерсенна
- Высокие уровни HWE дают возможность потренироваться во всём этом

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.

БИТОВЫЕ ОПЕРАЦИИ*

Красота побитовой и смешанной арифметики

Смешанная битовая арифметика

- Представим число из бесконечного количества бит

$10 = \cdots 01010$, $1 = \cdots 00001$, $0 = \cdots 00000$

- По правилам вычитания с переносом $-1 = \cdots 1111$
- Теперь $x + \bar{x} = -1$ откуда $-x = \bar{x} + 1 = \overline{x - 1}$ (докажите это)
- Пусть $x = \alpha 01^a 10^b$ тогда $\bar{x} = \bar{\alpha} 10^a 01^b$, $x - 1 = \alpha 01^a 01^b$, $-x = \bar{\alpha} 01^a 10^b$
- Чему тогда равен $x \mathbin{\&} (x - 1)$? Как оно соотносится с числом x?

Две функции popcount

- Первый вариант: наивный проход с маской

```
int popcount(unsigned x) {
  int n = 0, i;
  for (i = 0; i < sizeof(x) * CHAR_BIT; ++i) n += (x >> i) & 1;
  return n;
}
```

Второй вариант: тонкости битовой арифметики

```
int popcount(unsigned x) {
  int n = 0, i;
  while(x > 0) { x = x & (x - 1); n += 1; }
  return n;
}
```

Некоторые интересные соотношения

- $x = \alpha 01^a 10^b$, $\bar{x} = \bar{\alpha} 10^a 01^b$, $x - 1 = \alpha 01^a 01^b$, $-x = \bar{\alpha} 01^a 10^b$
- Стирание крайней справа единицы: $x \mathbin{\&} (x - 1)$
- Вычленение крайней справа единицы: $x \mathbin{\&} (-x)$
- Распространение крайней справа единицы вправо: x ^ $(x - 1)$
- Найдите свои?
- Например как перебрать все битовые подмножества числа $0\text{x}69 = 01101001$ т.е. множества $\{6,5,3,0\}$: $\{0\}$,…, $\{6\}$, $\{0,3\}$, …, $\{5,6\}$, …, $\{0,3,5\}$, …, $\{6,5,3\}$?

Алгоритм BSUB

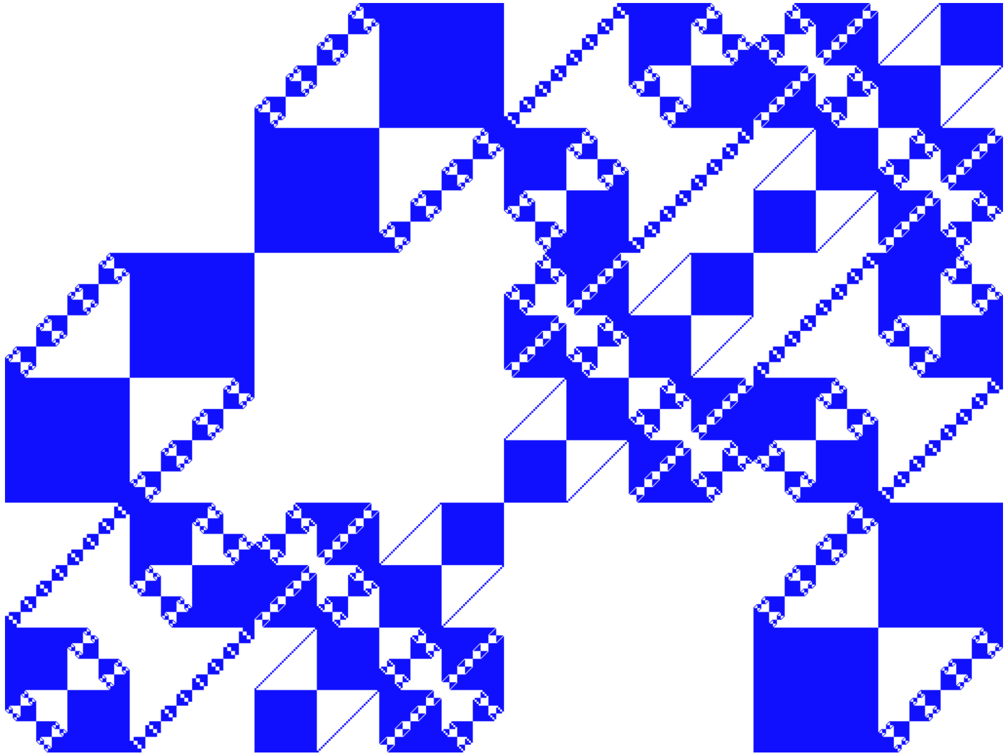
- Перебор всех подмножеств битового множества $x \to mask \ \& \ (x - mask)$

```
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 $\left((x \oplus y)^2 \gg 17\right) \mathbin{\&} 1$



funcbitmap.c

Обмен битов

- Дано число `x` и необходимо поменять в нём местами `i`-й и `j`-й (например третий и седьмой) биты. Есть очевидный способ это сделать: считать и записать

$y = (x \gg i)\ \&\ 1$

$z = (x \gg j)\ \&\ 1$

$x = \left(x\ \&\ {\sim}2^i\ \&\ {\sim}2^j\right) \mid (y \ll j) \mid (z \ll i)$

- Гораздо менее очевидным способом является δ-обмен. Пусть $j > i$. Обозначим $\delta = j - i$. Тогда обмен битов в `x` выглядит так:

$y = (x \oplus (x \gg \delta))\ \&\ 2^j$

$x = x \oplus y \oplus (y \ll \delta)$

Дельта-обмены

- Пусть $j > i$. Обозначим $\delta = j - i$. Тогда обмен битов в x выглядит так:

$y = (x \oplus (x \gg \delta))\ \&\ 2^j$

$x = x \oplus y \oplus (y \ll \delta)$

- δ-обмен выглядит не слишком интуитивно, но он окупается, если заметить, что теперь вместо 2^j может стоять любая маска θ
- Вопрос к математически настроенной части аудитории: какие параметры δ и θ надо подобрать, чтобы в 64-разрядном числе обменять верхние 25 битов с нижними?
- Заметьте, что прямая схема с прошлого слайда так хорошо не обобщается на эту задачу: там придётся высчитывать руками маску и анти-маску

Problem BMT – транспонирование

- В каждом 64-битном числе можно сохранить в нижних битах матрицу размером $n \times n$ для $n \leq 8$ хранящую значения 0 и 1.
- Ваша задача по входному числу и размеру выдать выходное число с **транспонированной** битовой матрицей внутри

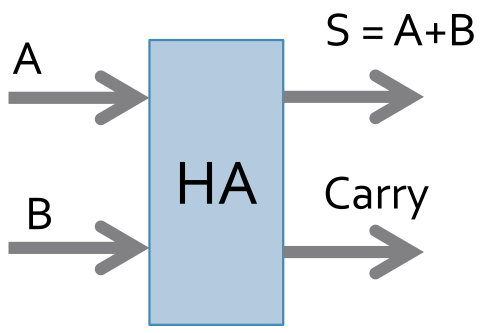
```
// sz > 1 && sz < 9
unsigned long long transpose(unsigned long long m, int sz);
```

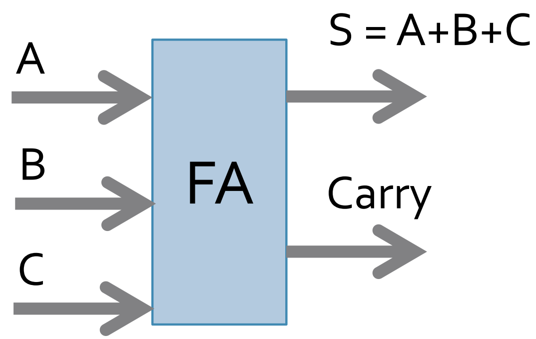
- Неиспользованные старшие биты считайте нулевыми
- Используйте дельта-обмены!

От булевой логики к арифметике

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

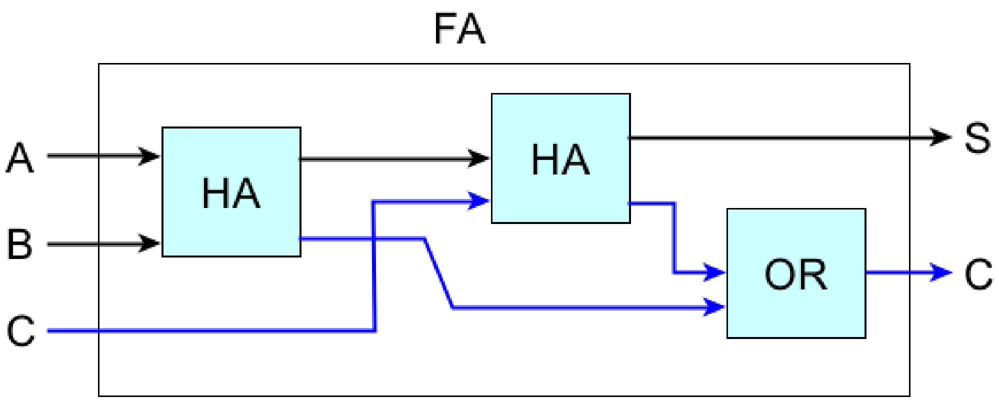
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1





- 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?

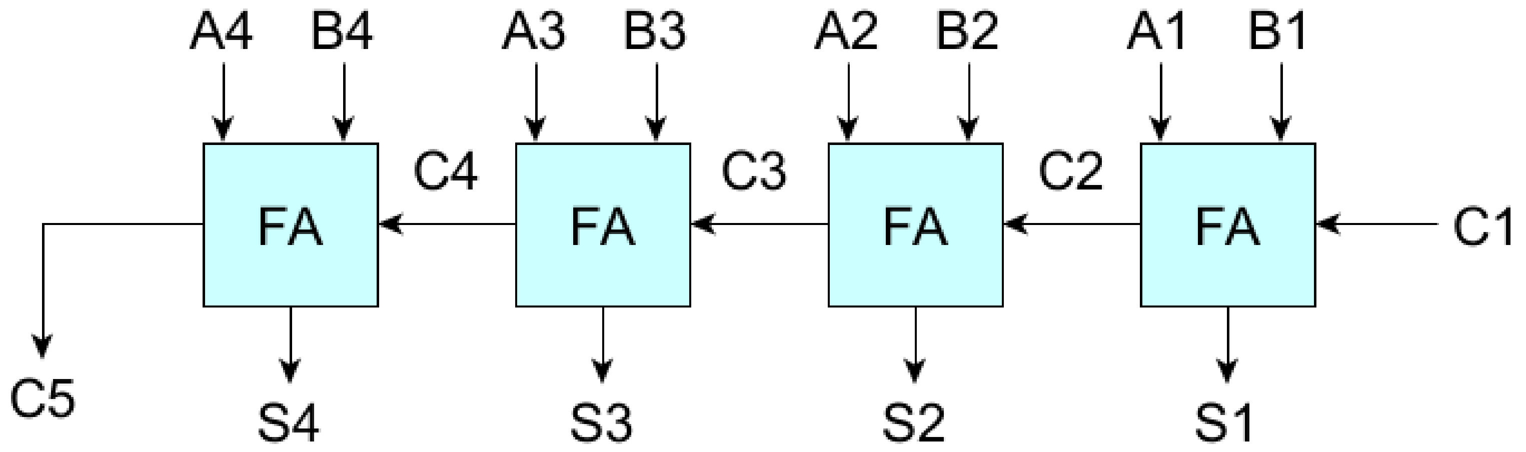
Полный сумматор



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

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

Схема сложения



- Какая сложность (по времени) сложения двух N-битных чисел?
- Как бы вы реализовали умножение?

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