Министерство образования и науки Российской Федерации федеральное государственное
автономное образовательное учреждение высшего образования
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Факультет «Программной инженерии и компьютерной техники.»

Алгоритмы и структуры данных

Лабораторная работа №3
Дополнительные задания.

Выполнил
Григорьев Давид Владимирович
Группа: P3215
Преподаватели
Косяков Михаил Сергеевич
Тараканов Денис Сергеевич

Санкт-Петербург 2025

Содержание

1 Реализации стека

1.1 Динамическим массивом

Реализация стека на основе динамического массива (например, `std::vector`). Основные операции стека (`push`, `pop`, `top`) соответствуют добавлению/удалению элементов в конце массива.

Преимущества:

- Эффективность: операции `push`/`pop` выполняются за $O(1)$ (амортизированно) благодаря динамическому расширению массива.
- Простота реализации: использование `std::vector` упрощает управление памятью.

Недостатки:

- Перераспределение памяти: при достижении емкости массива происходит выделение новой памяти и копирование элементов. Это довольно дорого, учитывая то, что в стэке нам важен обычно только самый верхний элемент.

Когда использовать: Нужно часто обращаться не только к самому верхнему элементу, и при этом пользоваться локальностью памяти (к примеру, проводить много поисков по стеку).

Пример кода:

```
#include <vector>
template <typename T>
class Stack {
private:
    std::vector<T> data;
public:
    void push(const T& value) { data.push_back(value); }
    void pop() { data.pop_back(); }
    T& top() { return data.back(); }
    bool empty() const { return data.empty(); }
};
```

1.2 Статическим массивом

Реализация на статическом массиве предполагает фиксированный размер стека, определённый на этапе компиляции.

Преимущества:

- Предсказуемая производительность: отсутствуют накладные расходы на динамическое управление памятью.
- Те же, что и с динамическим массивом

Недостатки:

- Ограниченный размер: риск переполнения стека.
- Возможно будет неиспользованная память, если выделять с запасом.

Когда использовать: Когда самостоятельно управляешь памятью выполнения программы, к примеру, в embedded системах, где переполнение стека может означать только ошибку.

Пример кода:

```
#define MAX_SIZE 100
template <typename T>
class Stack {
private:
    T data[MAX_SIZE];
    int topIndex;
public:
    Stack() : topIndex(-1) {}
    void push(const T& value) {
        if (topIndex < MAX_SIZE - 1) data[++topIndex] = value;
    }
    void pop() { if (!empty()) --topIndex; }
    T& top() { return data[topIndex]; }
    bool empty() const { return topIndex == -1; }
};
```

1.3 Связным списком

Реализация через односвязный список позволяет динамически выделять память под каждый новый элемент стека.

Преимущества:

- Неограниченный размер: стек расширяется до тех пор, пока есть свободная память.
- Эффективность вставки/удаления: операции выполняются за $O(1)$.

Недостатки:

- Дополнительный расход памяти: каждый элемент требует хранения указателя на следующий узел.
- Полное отсутствие локальности памяти.

Когда использовать: Возможно в многопоточных приложениях, ведь не надо копировать никакие данные при вставке.

Пример кода:

```
template <typename T>
struct Node {
    T data;
    Node* next;
};

template <typename T>
class Stack {
private:
    Node<T>* topNode;
public:
    Stack() : topNode(nullptr) {}
    void push(const T& value) {
        Node<T>* newNode = new Node<T>{value, topNode};
        topNode = newNode;
    }
    void pop() {
        if (topNode) {
            Node<T>* temp = topNode;
            topNode = topNode->next;
```

```
            delete temp;
        }
    }
    T& top() { return topNode->data; }
    bool empty() const { return topNode == nullptr; }
};
```

1.4 Через указатель на массив указателей (T**)

Реализация через блочную структуру, где данные хранятся в блоках фиксированного размера, а индекс элемента вычисляется через деление на размер блока. Используется массив указателей (`T**`) для ссылки на блоки. Такая реализация похожа на настоящую реализацию `std::deque`.

Преимущества:

- Эффективное расширение: блоки выделяются по мере необходимости без полного перераспределения.

- Локализация данных: элементы одного блока находятся рядом в памяти, что улучшает работу кэша.

Недостатки:

- Сложность реализации: требуется управление блоками и вычисление индексов.

- Низкая производительность для стека: операции происходят только с одним концом, что делает блочную структуру избыточной.

Когда использовать: Если стек является частью более сложной структуры данных (например, универсального контейнера для стека и очереди).

Пример кода:

```
template <typename T>
class Stack {
private:
    T** blocks;       // Массив указателей на блоки
    int blockSize;    // Размер одного блока
    int blockCount;   // Количество блоков
    int size;         // Текущий размер стека

    int blockIndex(int index) { return index / blockSize; }
    int elementIndex(int index) { return index % blockSize; }

public:
    Stack(int _blockSize = 1024) : blockSize(_blockSize), blockCount(1), size(0) {
        blocks = new T*[blockCount];
        blocks[0] = new T[blockSize];
    }

    ~Stack() {
        for (int i = 0; i < blockCount; ++i) delete[] blocks[i];
        delete[] blocks;
    }

    void push(const T& value) {
        if (size >= blockCount * blockSize) {
            T** newBlocks = new T*[blockCount + 1];
            for (int i = 0; i < blockCount; ++i) newBlocks[i] = blocks[i];
            newBlocks[blockCount] = new T[blockSize];
            delete[] blocks;
            blocks = newBlocks;
            blockCount++;
        }
```

```
        blocks[blockIndex(size)][elementIndex(size)] = value;
        size++;
    }

    void pop() {
        if (size > 0) size--;
    }

    T& top() {
        return blocks[blockIndex(size - 1)][elementIndex(size - 1)];
    }

    bool empty() const { return size == 0; }
};
```

1.5 Сравнение реализаций

Реализация	push/pop	Память	Гибкость	Рекомендуется для
Динамический массив	$O(1)$ аморт.	Автоматическое	Среднее	Общих случаев, когда важна скорость
Статический массив	$O(1)$	Фиксированное	Низкое	Систем с ограничениями по памяти
Связный список	$O(1)$	Динамическое	Высокое	Многопоточных приложений
Через T** (блочная)	$O(1)$	Блочное	Среднее	Универсальных контейнеров

1.6 Стандартная реализация в C++

В C++ стек представлен шаблонным классом `std::stack`, который по умолчанию использует `std::deque`. Однако его можно настроить на использование `std::vector` или `std::list`:

```
std::stack<int> s; // Использует deque
std::stack<int, std::vector<int>> s_vec; // Использует vector
std::stack<int, std::list<int>> s_list; // Использует list
```

2 Дерево отрезков

2.1 Описание

Дерево отрезков — это структура данных, которая позволяет эффективно обрабатывать запросы на интервалах и выполнять операции обновления элементов. Основное требование к операциям: они должны быть *ассоциативными*, то есть $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.

Примеры поддерживаемых операций

- **Минимум/максимум** на отрезке ($\min(a, b)$, $\max(a, b)$).
- **Наибольший общий делитель (НОД)** ($\gcd(a, b)$).
- **Произведение матриц** ($A \cdot B$, ассоциативно, но не коммутативно).
- **Композиция перестановок**.
- **Сумма** ($a + b$).
- **Побитовые операции**: & (И), | (ИЛИ), $\wedge$ (Исключающее ИЛИ).

2.2 Структура дерева

Дерево отрезков строится на массиве $A[1..n]$. Каждый лист соответствует элементу массива, а внутренние узлы хранят результат операции $\oplus$ над своими дочерними узлами.

Сложность операций

- Запрос на интервале: $O(\log n)$.
- Обновление элемента: $O(\log n)$.

2.3 Спуск в дереве

Спуск (traversal) — это процесс перемещения от корня к листьям для выполнения запроса или обновления. При этом учитываются отложенные операции через `push`.

Алгоритм спуска

1. Если текущий интервал полностью лежит в запросе, вернуть сохранённое значение.
2. Вызвать `push` для текущего узла.
3. Рекурсивно обработать левого и правого потомка.
4. Объединить результаты подынтервалов ($\oplus$).

2.4 Дополнительные операции

1. Поиск количества элементов > x

Для этого дерево должно хранить отсортированные подынтервалы (например, с использованием *дерева отрезков с хранением множеств*).

2. Максимальная подпоследовательность

Реализуется через хранение в узле:

- максимальной суммы на отрезке,
- максимальной суммы с начала,
- максимальной суммы с конца,
- общей суммы.

3 Применение дерева отрезков для задачи Иосифа Флавия

3.1 Постановка задачи

Задача Иосифа Флавия требует определить порядок устранения солдат, расположенных по кругу, через фиксированный шаг k. Классическое решение имеет сложность $O(nk)$, но использование дерева отрезков позволяет снизить сложность до $O(n \log n)$.

3.2 Адаптация структуры данных

- Каждый лист дерева соответствует живому солдату (значение 1)
- Внутренние узлы хранят сумму своих потомков (количество живых солдат в интервале)
- Размер дерева расширяется до ближайшей степени двойки для упрощения индексации

3.3 Ключевые операции

- **Построение дерева**:
  - Листья инициализируются единицами
  - Внутренние узлы вычисляются как сумма дочерних
  - Сложность: $O(n)$
- **Устранение солдата**:
  - Обновление значений от листа к корню (уменьшение суммы)
  - Сложность: $O(\log n)$
- **Поиск k-го солдата**:
  - Рекурсивный спуск с учётом количества живых в поддеревьях
  - Автоматическая обработка круговой структуры через перерасчёт позиций
  - Сложность: $O(\log n)$

3.4 Алгоритм поиска позиции

1. Начинаем с предыдущей позиции устранения
2. Вычисляем относительный номер цели в текущем кольце
3. Спускаемся по дереву:
   - Если в левом поддереве достаточно элементов — идём влево
   - Иначе вычитаем количество левых элементов и идём вправо
4. При достижении листа возвращаем его позицию

Пример спуска в дереве отрезков

$n = 5$, $k = 2$. Дерево отрезков строится на массиве из 5 элементов, расширенном до ближайшей степени двойки $2^3 = 8$. Листья дерева (позиции 4–8) соответствуют солдатам 1–5 (значения 1), остальные листья (если есть) инициализируются нулями. Внутренние узлы хранят сумму своих потомков. Переменная `current_position_` играет ключевую роль в отслеживании текущей позиции для следующего удаления .

Шаг 1: Удаление второго солдата При первом вызове `EliminateNext()` текущая позиция равна $k = 2$ (инициализирована в конструкторе). Алгоритм обновляет дерево, уменьшая значение листа, соответствующего позиции 2, на 1 (теперь он равен 0). После этого `current_position_` обновляется как $node - last\_internal\_node\_ = 9 - 7 = 2$ (индекс листа 9 соответствует позиции 2) . Обновлённая структура дерева:

```
Корень (1): 4
 Левое поддерево (2): 1 (сумма листьев 8 и 9: 1 + 0)
 Правое поддерево (3): 3 (сумма листьев 10-12: 1 + 1 + 1)
```

Шаг 2: Спуск для поиска следующего солдата Теперь необходимо найти следующего солдата через шаг $k = 2$. Алгоритм начинает спуск с корня, учитывая текущую позицию $current\_position\_ = 2$ (обновлена на шаге 1). Поиск ведётся с использованием шага $k = 2$:

1. **Корень (1)**: значение 4 (количество живых солдат). - Левое поддерево (2) имеет значение 1. - Поскольку $k = 2 > 1$, переходим в правое поддерево и обновляем $k = 2 - 1 = 1$.

2. **Правое поддерево корня (3)**: значение 3. - Левое поддерево узла 3 (узел 6) имеет значение 1. - Поскольку $k = 1 \leq 1$, переходим в левое поддерево.

3. **Лист** 10: соответствует солдату 3. - Обновляем дерево, уменьшая значение этого листа на 1 (теперь он равен 0). - `current_position_` обновляется как $10 - 7 = 3$.

Результат: второй удалённый солдат — позиция 3. Новая текущая позиция: 3.
Шаг 3: Продолжение спуска Оставшиеся солдаты: $[1, 4, 5]$. Следующий шаг $k = 2$:

1. **Корень (1)**: значение 3. - Левое поддерево (2) имеет значение 1 (солдат 1). - Поскольку $k = 2 > 1$, переходим в правое поддерево и обновляем $k = 2 - 1 = 1$.

2. **Правое поддерево корня (3)**: значение 2 (солдаты 4, 5). - Левое поддерево узла 3 (узел 6) имеет значение 0. - Переходим в правое поддерево и проверяем узел 7 (значение 1). - Поскольку $k = 1 \leq 1$, находим лист 11 (солдат 4). - `current_position_` обновляется как $11 - 7 = 4$.

Результат: третий удалённый солдат — позиция 4. Новая текущая позиция: 4.

3.5 Полный исходный код

```
#include <cstddef>
#include <cstdio>
#include <iostream>
#include <vector>

class JosephusTree {
  std::vector<int> segment_tree_;
  int total_soldiers_;
  int last_internal_node_;
  int elimination_step_;
  int current_position_;
  int remaining_count_;

  static unsigned NextPowerOfTwo(unsigned n) {
```

```
    n--;
    n |= n >> 1;
    n |= n >> 2;
    n |= n >> 4;
    n |= n >> 8;
    n |= n >> 16;
    return n + 1;
  }

  static constexpr int Parent(int node) { return node / 2; }
  static constexpr int Left(int node) { return node * 2; }
  static constexpr int Right(int node) { return (node * 2) + 1; }

  bool Valid(int node) const {
    return node <= last_internal_node_ + total_soldiers_ && node > 0;
  }

public:
  JosephusTree(int n, int k)
      : total_soldiers_(n), elimination_step_(k), current_position_(k),
        remaining_count_(n) {
    unsigned req_size = NextPowerOfTwo(n * 2);
    segment_tree_.resize(req_size);

    unsigned m = (n * 2) - 1;
    int height = 31 - __builtin_clz(m);
    last_internal_node_ = (1 << height) - 1;

    for (int i = last_internal_node_ + 1; i <= last_internal_node_ + n; ++i) {
      segment_tree_[static_cast<size_t>(i)] = 1;
    }

    for (int i = last_internal_node_; i >= 1; --i) {
      segment_tree_[static_cast<size_t>(i)] =
          (Valid(Left(i)) ? segment_tree_[static_cast<size_t>(Left(i))] : 0) +
          (Valid(Right(i)) ? segment_tree_[static_cast<size_t>(Right(i))] : 0);
    }
  }

  int Remaining() const { return remaining_count_; }

  int EliminateNext() {
    if (remaining_count_-- == total_soldiers_) {
      return current_position_;
    }

    size_t node = last_internal_node_ + current_position_;
    while (node > 0) {
      segment_tree_[node]--;
      node = Parent(node);
    }

    node = last_internal_node_ + current_position_;
    int k = elimination_step_;
    enum class Dir { FromLeft, FromRight, FromAbove };
```

```
    Dir dir = Dir::FromRight;

    while (k > 0) {
      const size_t l = static_cast<size_t>(Left(node));
      const size_t r = static_cast<size_t>(Right(node));

      if (dir == Dir::FromAbove) {
        if (Valid(l) && k > segment_tree_[l]) {
          k -= segment_tree_[l];
          node = r;
        } else if (!Valid(l) && k == segment_tree_[node]) {
          k--;
        } else {
          node = l;
        }
      } else if (dir == Dir::FromRight) {
        dir = (node == Right(Parent(node))) ? Dir::FromRight : Dir::FromLeft;
        node = Parent(node);
      } else {
        if (Valid(r) && k > segment_tree_[r]) {
          k -= segment_tree_[r];
          dir = (node == Right(Parent(node))) ? Dir::FromRight : Dir::FromLeft;
          node = Parent(node);
        } else {
          node = r;
          dir = Dir::FromAbove;
        }
      }

      if (!Valid(node)) {
        node = last_internal_node_ + 1;
        k -= segment_tree_[node];
        dir = Dir::FromRight;
      }
    }

    current_position_ = node - last_internal_node_;
    return current_position_;
  }
};

int main() {
  int N = 0;
  int K = 0;
  std::cin >> N >> K;

  JosephusTree jt(N, K);
  while (jt.Remaining()) {
    int n = jt.EliminateNext();
    char buf[16];
    int p = 0;
    do buf[p++] = n % 10 + '0';
    while (n /= 10);
    while (p--) putchar(buf[p]);
    putchar(' ');
```

```
  }
}
```

3.6 Сравнение с другими методами

Метод	Сложность	Примечания
Наивный	$O(n^2)$	Только для малых n
Рекурсивный	$O(n)$	Только для постоянного k
Дерево отрезков	$O(n \log n)$	Универсальный подход

4 Методы разрешения коллизий в хеш-мапе

4.1 Метод цепочек (Chaining)

Описание

Каждая ячейка хеш-таблицы содержит список (или дерево) для хранения элементов, попадающих в один индекс. При коллизии элемент добавляется в список.

Плюсы

- Простота реализации.
- Неограниченное количество элементов в одной ячейке.
- Устойчивость к переполнению таблицы.

Минусы

- Увеличение времени поиска при длинных списках.
- Неравномерное распределение элементов (группирование).

Случаи использования

- Когда частота коллизий высока, а размер данных непредсказуем.
- В реализациях, где важна простота вставки и удаления (например, Java `HashMap` с преобразованием списков в деревья при длине цепочки > 8).

4.2 Открытая адресация (Open Addressing)

Описание

Элементы хранятся непосредственно в массиве таблицы. При коллизии используется стратегия поиска свободной ячейки:

- Линейное пробирование.
- Квадратичное пробирование.
- Двойное хеширование.

Плюсы

- Эффективное использование кэша процессора (данные хранятся локально).
- Быстрый доступ при низкой загрузке таблицы.

Минусы

- Риск переполнения таблицы.
- Сложность удаления элементов (требуется маркировка "удалено").
- Группирование элементов при линейном пробировании.

Случаи использования

- Когда память ограничена, а частота коллизий низка.
- Для временных таблиц с фиксированным размером.

4.3 Улучшение хеш-функции

Описание

Использование сложных хеш-функций (например, криптографических), которые минимизируют коллизии.

Плюсы

- Снижение вероятности коллизий.
- Равномерное распределение данных.

Минусы

- Высокая вычислительная сложность функции.
- Не гарантирует отсутствия коллизий полностью.

Случаи использования

- В системах, где требуется высокая безопасность (например, блокчейн).
- При работе с чувствительными данными.

4.4 Динамическое расширение таблицы (Rehashing)

Описание

Увеличение размера таблицы и перехеширование элементов при достижении порога заполнения.

Плюсы

- Поддержание низкой вероятности коллизий.
- Гибкость в управлении ресурсами.

Минусы

- Высокие накладные расходы на перехеширование.
- Периодические задержки при расширении таблицы.

Случаи использования

- В динамических системах с переменным объемом данных.
- Для долгоживущих таблиц с непредсказуемым ростом.

4.5 Построение деревьев в Java (`HashMap`)

Описание

В Java `HashMap` при длине цепочки > 8 список преобразуется в красно-черное дерево для ускорения поиска (с O(n) до O(log n)).

Плюсы

- Оптимизация производительности при высоких коллизиях.
- Баланс между скоростью и памятью.

Минусы

- Сложность реализации.
- Дополнительные затраты на поддержку структуры дерева.

Случаи использования

- В системах с высокой нагрузкой и частыми коллизиями.
- Для приложений, где критична скорость поиска (например, кэширование).

5 Оптимальность стратегии Менеджер-памяти 1

5.1 Введение

В данной задаче требуется реализовать менеджер памяти, который обрабатывает запросы на выделение и освобождение памяти. В текущей реализации используется **max-heap** и **worst-fit** подход: для каждого запроса выбирается самый большой доступный блок. В данном анализе рассматриваются преимущества и недостатки этой стратегии, а также возможные альтернативы.

5.2 Текущая стратегия: worst-fit

5.2.1 Описание

Worst-fit — это стратегия выделения памяти, при которой выбирается **самый большой доступный блок**, подходящий по размеру. В реализации используется **max-heap**, где каждый узел представляет свободный блок памяти, и сравнение производится по размеру блока.

5.2.2 Преимущества

- **Высокая производительность**: операции поиска, вставки и удаления выполняются за $O(\log n)$, что критично при большом количестве запросов ($M \leq 10^5$).
- **Простота реализации**: управление памятью через кучу и связные списки логически понятно и легко транслируется в код.
- **Снижение фрагментации**: при освобождении памяти автоматически объединяются смежные свободные блоки, что уменьшает количество мелких фрагментов.
- **Корректность**: реализация успешно прошла все тесты, что подтверждает её надёжность.

5.2.3 Недостатки

- **Фрагментация памяти**: разбиение крупных блоков на мелкие приводит к увеличению числа непригодных для использования фрагментов.
- **Неэффективное использование ресурсов**: worst-fit выбирает самый большой блок, даже если он значительно превышает размер запроса, что может привести к потере возможности обслужить будущие запросы.
- **Более высокое количество отклонённых запросов**: в сценариях с разнообразными размерами запросов worst-fit может привести к большему количеству отказов по сравнению с другими стратегиями.

5.3 Альтернативная стратегия: best-fit

5.3.1 Описание

Best-fit — это стратегия выделения памяти, при которой выбирается **наименьший подходящий блок**, то есть блок, размер которого не меньше запрашиваемого, но минимально возможен. Для реализации этой стратегии можно использовать **min-heap** или `std::multiset`, где блоки сортируются по размеру.

5.3.2 Преимущества

- **Минимизация фрагментации**: сохраняются крупные блоки для будущих запросов.
- **Больше успешных выделений**: в сценариях с разнообразными размерами запросов best-fit может удовлетворить больше запросов.
- **Устойчивость к "дыркам"**: мелкие запросы занимают мелкие блоки, оставляя крупные для больших запросов.

5.3.3 Недостатки

- **Увеличение времени работы**: поиск подходящего блока может занять $O(n)$ в худшем случае, если не использовать эффективные структуры данных.

5.4 Реализация best-fit в текущей задаче

Для реализации best-fit в текущей задаче можно использовать `std::multiset<Block*, Compare>`, где блоки хранятся в отсортированном виде по размеру.

5.4.1 Основные изменения

1. **Компаратор для сортировки блоков**:

```
struct BlockCompare {
    size_t requested_size = 0;
    bool operator()(const Block* a, const Block* b) const {
        return a->Size() < b->Size(); // Сортировка по размеру
    }
};
```

2. **Поиск наименьшего подходящего блока**:

```
auto it = free_blocks.lower_bound(requested_size);
if (it == free_blocks.end()) {
    // Запрос отклоняется
} else {
    // Выделяем блок
}
```

3. **Обновление методов** `Allocate` **и** `Free`:

   - В `Allocate`: выделяем блок, разбиваем его, если остаток > 0.
   - В `Free`: добавляем освобождённый блок в `free_blocks`, объединяем смежные блоки.

5.5 Заключение

- **Max-heap (worst-fit)**: подходит для быстрых выделений и крупных запросов. Прост в реализации и работает быстро.
- **Min-heap / best-fit**: требует более сложной логики, но может уменьшить количество отказов. Эффективно реализуется через `std::multiset`.

6 Первое дополнительное задание.

Пояснение к примененному алгоритму

Аналитическое решение задания D.

Временная сложность

- **Худший случай**: $O(d^2)$, где d – вместимость контейнера.
  - Цикл выполняется $O(d)$ раз (ограничение $d + 2$)
  - Поиск в истории: $O(d)$ на итерацию

Пространственная сложность

$O(d)$ для хранения истории состояний.

Практическая эффективность

- Для $d \leq 1000$: 10^6 операций
- Реальные сценарии редко достигают худшего случая

Код алгоритма

```cpp
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <vector>

int main() {
  size_t a_val = 0;
  size_t b_val = 0;
  size_t c_val = 0;
  size_t d_val = 0;
  size_t k_val = 0;
  std::cin >> a_val;
  std::cin >> b_val;
  std::cin >> c_val;
  std::cin >> d_val;
  std::cin >> k_val;

  std::vector<size_t> history;
  size_t current = a_val;
  history.push_back(current);

  for (size_t day = 1; day <= k_val; ++day) {
    size_t new_val = current * b_val;
    if (new_val < c_val) {
      std::cout << 0 << '\n';
      return 0;
    }
    size_t temp = new_val - c_val;
    if (temp <= 0) {
      std::cout << 0 << '\n';
```

```
      return 0;
    }
    temp = std::min(temp, d_val);

    if (temp == current) {
      std::cout << temp << '\n';
      return 0;
    }

    auto it = std::find(history.begin(), history.end(), temp);
    if (it != history.end()) {
      size_t pos = it - history.begin();
      size_t cycle_length = day - pos;
      size_t remaining_days = k_val - day;
      size_t final_pos = pos + (remaining_days % cycle_length);
      std::cout << history[final_pos] << '\n';
      return 0;
    }

    if (history.size() > d_val + 2) {
      break;
    }

    history.push_back(temp);
    current = temp;
  }

  std::cout << current << '\n';
  return 0;
}
```

7 Второе дополнительное задание

Вопрос 1: Амортизированная сложность push_back в векторе

Вектор — это динамический массив, который автоматически расширяется при добавлении элементов. Операция `push_back` имеет амортизированную сложность $O(1)$, потому что:

- При заполнении текущего буфера выделяется новый блок памяти (обычно в 2 раза больше предыдущего), и старые элементы копируются туда.
- Хотя копирование занимает $O(n)$ времени, это происходит редко. Среднее время на операцию остаётся константным благодаря геометрическому росту ёмкости.

Вопрос 2: Устройство std::deque и инвалидация указателей

`std::deque` реализован как последовательность фиксированных блоков памяти («карманов»), связанных через указатели. Это позволяет:

- Добавлять/удалять элементы с обоих концов за $O(1)$ времени.
- Указатели на элементы не инвалидируются при добавлении новых блоков, так как старые блоки остаются на месте. В отличие от `std::vector`, здесь нет перераспределения всей памяти при расширении.

Вопрос 3: Куча и удаление элементов

Куча (heap) — это почти полное бинарное дерево, где родитель больше (max-heap) или меньше (min-heap) своих потомков. **Способы удаления:** 1. *Удаление корня (наибольшего/наименьшего элемента)*: заменяем корень последним элементом, удаляем последний элемент, затем восстанавливаем свойство кучи «просеиванием вниз» ($O(\log n)$). 2. *Удаление произвольного элемента*: заменяем элемент на бесконечность (или противоположное значение), поднимаем его вверх до корня, затем удаляем корень.

Вопрос 4: Связный список и кольцевая реализация

Связный список — структура данных, где каждый узел хранит данные и указатель на следующий (и/или предыдущий) узел:

- *Односвязный список*: только указатель на следующий элемент.
- *Двусвязный список*: указатели на следующий и предыдущий. **Кольцевая реализация**: последний элемент указывает на первый, создавая цикл. Это упрощает обход и операции вставки/удаления, так как не нужно отдельно обрабатывать начало и конец списка.