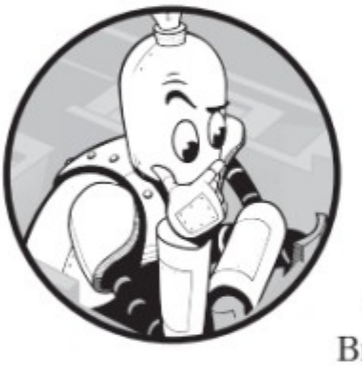
Оглавление

1

Хеш-таблицы



В этой главе мы решим две задачи, опираясь на средства эффективного поиска. В первой из них нужно будет проверить идентичность снежинок. Во второй мы будем определять, какие из слов являются сложносоставными. Вы увидите, что некоторые корректные подходы к решению оказываются недостаточно быстрыми. Для существенного повышения скорости вычислений мы используем структуру данных под названием «хеш-таблица», которую также изучим в деталях.

В завершение главы рассмотрим третью задачу: определим, сколькими способами можно удалить букву из слова, чтобы получить другое слово. В ней будут показаны риски необдуманного использования рассмотренной структуры данных, ведь при освоении чего-то нового всегда хочется сразу начать применять это везде!

Задача 1. Уникальные снежинки

Рассмотрим задачу под номером `cco07p2` с сайта DMOJ.

Условие

Дана коллекция снежинок, нужно определить, содержит ли она идентичные снежинки.

Каждая снежинка описывается шестью целыми числами, каждое из которых характеризует длину одного ее луча. Вот пример:

```
3, 9, 15, 2, 1, 10
```

При этом значения длин лучей могут повторяться, скажем, вот так:

```
8, 4, 8, 9, 2, 8
```

Но как понять, являются ли две снежинки идентичными? Рассмотрим несколько примеров.

Сначала сравним две снежинки:

```
1, 2, 3, 4, 5, 6
```

и

```
1, 2, 3, 4, 5, 6
```

Очевидно, что они идентичны, потому что числа одной совпадают с числами другой, находящимися в соответствующих позициях.

А вот второй пример:

```
1, 2, 3, 4, 5, 6
```

и

```
4, 5, 6, 1, 2, 3
```

Эти снежинки тоже идентичны, так как если начать с `1` во второй снежинке и продвигаться вправо, то сперва идут значения `1, 2, 3`, после чего происходит возврат в начало и последовательность продолжается значениями `4, 5, 6`. Оба этих сегмента вместе формируют снежинку, идентичную первой.

Каждую снежинку можно представить как круг. Тогда два предложенных здесь образца будут идентичными, потому что при правильном выборе начальной точки сопоставления во второй снежинке и следовании по часовой стрелке мы получаем первую снежинку.

Попробуем пример посложнее:

```
1, 2, 3, 4, 5, 6
```

и

```
3, 2, 1, 6, 5, 4
```

На основе только что рассмотренных вариантов можно сделать вывод, что эти снежинки не идентичны. Если начать с `1` во второй снежинке и продвигаться по кругу вправо до возвращения к точке отсчета, то мы получим `1, 6, 5, 4, 3, 2`. Эта форма далека от `1, 2, 3, 4, 5, 6` первой снежинки.

Тем не менее если начать с `1` во второй снежинке и перемещаться не вправо, а влево, тогда мы получим `1, 2, 3, 4, 5, 6`. Перемещение влево от `1` дает `1, 2, 3`, после чего происходит переход к правому краю и дальнейшее продвижение по значениям `4, 5, 6`.

Таким образом, две снежинки считаются идентичными и в случае, если числа совпадают при продвижении влево от точки отсчета.

Обобщая сказанное, можно заключить, что две снежинки идентичны, если они описываются одинаковыми последовательностями чисел либо если сходство обнаруживается при перемещении по этим последовательностям влево или вправо.

Входные данные

Первая строка входных данных является целым числом *n*, описывающим количество сравниваемых снежинок. Значение *n* будет находиться в диапазоне от 1 до 100 000. Каждая из следующих *n* строк характеризует одну снежинку: содержит шесть целых чисел, каждое из которых может иметь значение в диапазоне от 0 до 10 000 000.

Выходные данные

На выходе должна быть получена одна текстовая строка:

- Если идентичных снежинок не обнаружено, следует вывести:

  ```
  No two snowflakes are alike (нет одинаковых снежинок).
  ```

- Если есть хотя бы две идентичные снежинки, следует вывести:

  ```
  Twin snowflakes found (найдены одинаковые снежинки).
  ```

Время выполнения вычислений ограничено двумя секундами.

Упрощаем задачу

Одна из универсальных стратегий для решения задач по программированию состоит в проработке их упрощенных версий. Давайте немного разомнемся, снизив сложность нашей задачи.

Предположим, что мы работаем не со снежинками, состоящими из множества целых чисел, а с отдельными целыми числами. У нас есть коллекция, и нужно узнать, содержит ли она одинаковые числа. Проверить одинаковость двух целых чисел в Си можно с помощью оператора `==`. Мы будем тестировать все варианты пар чисел, и если найдем одинаковые числа хотя бы в одной паре, то остановимся и выведем:

```
Twin integers found (найдены одинаковые числа).
```

Если мы не обнаружим одинаковых чисел, то вывод будет таким:

```
No two integers are alike (нет одинаковых чисел).
```

Теперь создадим функцию `identify_identical` с двумя вложенными циклами для сравнения пар целых чисел, как показано в листинге 1.1.

Листинг 1.1. Поиск одинаковых целых чисел

```
void identify_identical(int values[], int n) {
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = i+1; j < n; j++) {❶
      if (values[i] == values[j]) {
        printf("Twin integers found.\n");
        return;
      }
    }
  }
  printf("No two integers are alike.\n");
}
```

Мы передаем числа в функцию через массив `values`. Также мы передаем `n`, то есть количество чисел в массиве.

Обратите внимание, что мы начинаем внутренний цикл с `i + 1`, а не с `0` ❶. Если начать с `0`, то `j` будет равняться `i` и мы будем сравнивать элемент с самим собой, получая ложноположительный результат.

Протестируем `identify_identical` с помощью небольшой функции `main`:

```
int main(void) {
  int a[5] = {1, 2, 3, 1, 5};
  identify_identical(a, 5);
  return 0;
}
```

При выполнении кода вывод покажет, что функция определила совпадающую пару единиц. В дальнейшем я ограничусь минимумом тестов, но при этом важно, чтобы вы сами экспериментировали.

Решение основной задачи

Теперь попробуем изменить функцию `identify_identical` для решения задачи со снежинками. В код нужно будет внести два изменения.

1. Нужно работать не с одним, а с шестью целыми числами сразу. Для сравнения хорошо подойдет двумерный массив, в котором каждая строка будет снежинкой с шестью столбцами (по одному для каждого элемента).
2. Как мы выяснили, снежинки могут оказаться одинаковыми в трех случаях. К сожалению, это подразумевает, что для их сравнения уже не получится использовать `==`. Необходимо учесть возможности «перемещения вправо» и «перемещения влево» (не говоря уже о том, что `==` в Си для сравнения массивов не используется). Так что основным изменением имеющегося алгоритма станет правильное сопоставление снежинок.

Для начала напишем две вспомогательные функции: одну для проверки «перемещения вправо» и вторую для проверки «перемещения влево». Каждая из них будет получать параметры двух снежинок и стартовую точку обхода второй снежинки.

Проверка вправо

Заголовок функции для `identical_right`:

```
int identical_right(int snow1[], int snow2[], int start)
```

Чтобы определить, совпадают ли снежинки при «передвижении вправо», можно просканировать `snow1` с индекса `0` и `snow2` с индекса `start`. В случае обнаружения разногласия между сопоставляемыми элементами будет возвращаться `0`, указывая, что снежинки не идентичны. Если же все сопоставляемые элементы совпадут, возвращаться будет `1`. Таким образом, `0` означает `false`, а `1` — `true`.

В листинге 1.2 приводится первый вариант кода для этой функции.

Листинг 1.2. Определение идентичности снежинок перемещением вправо (с ошибкой!)

```
int identical_right(int snow1[], int snow2[],
                    int start) { //ошибка!
  int offset;
  for (offset =0; offset < 6; offset++) {
    if (snow1[offset] != snow2[start + offset]) ❶
      return 0;
  }
  return 1;
}
```

К сожалению, этот код не работает нужным образом. Проблема в выражении `start + offset` ❶. Если `start = 4` и `offset = 3`, тогда `start + offset = 7` и мы обращаемся к элементу `snow2[7]`.

Этот код не учитывает, что нам нужно перейти к левой стороне `snow2`. Если код может использовать ведущий к ошибке индекс `6` или выше, то индекс нужно сбрасывать вычитанием шестерки. Это позволит продолжить с индекса `0` вместо индекса `6`, индекса `1` вместо `7` и т. д. Попробуем исправить ошибку в листинге 1.3.

Листинг 1.3. Определение идентичности снежинок перемещением вправо

```
int identical_right(int snow1[], int snow2[],
                    int start) {
  int offset, snow2_index;
  for (offset =0; offset < 6; offset++) {
    snow2_index = start + offset;
    if (snow2_index >= 6)
      snow2_index = snow2_index - 6;
        if (snow1[offset] != snow2[snow2_index])
      return 0;
  }
  return 1;
}
```

Такой вариант работает, но его можно улучшить. Одно из возможных изменений — это использование `%`, оператора вычисления остатка от деления (mod). Он вычисляет остаток, то есть `x % y` возвращает остаток от целочисленного деления `x` на `y`. Например, `6 % 3` будет равно нулю, так как остатка от операции деления шести на три не будет. Вычисление `6 % 4` даст `2`, поскольку в остатке при делении шести на четыре будет два.

Оператор mod можно применить здесь для более простой реализации перехода к началу последовательности чисел. Заметьте, что `0 % 6` даст ноль, `1 % 6` даст один, ..., `5 % 6` даст пять. Каждое из этих чисел меньше шести, в связи с чем будет само являться остатком от деления на шесть. Числа от нуля до пяти соответствуют действительным индексам `snow`, поэтому хорошо, что `%` их не меняет. А для индекса `6` операция `6 % 6` даст ноль: шесть на шесть делится без остатка, перенося нас к началу последовательности чисел снежинки.

Обновим функцию `identical_right` с использованием оператора `%`. В листинге 1.4 показан ее доработанный вариант.

Листинг 1.4. Определение идентичности снежинок перемещением вправо с вычислением остатка

```
int identical_right(int snow1[], int snow2[], int start) {
  int offset;
  for (offset =0; offset < 6; offset++) {
```

```
    if (snow1[offset] != snow2[(start + offset) % 6])
      return 0;
  }
  return 1;
}
```

Использовать прием с получением остатка или нет — дело ваше. Он сокращает одну строку кода и является шаблоном, который знаком многим программистам. Тем не менее применить его не всегда возможно, даже для задач, требующих сброса индекса, — например, `identical_left`. Как раз сейчас мы к этому и перейдем.

Проверка влево

Функция `identical_left` очень похожа на `identical_right`, но здесь нам нужно сначала перемещаться влево, а затем переходить к правой стороне. При обходе снежинки вправо мы не должны были использовать индекс `6` и выше. На этот раз нужно избегать обращения к индексу `-1` и ниже.

К сожалению, в данном случае решение с получением остатка не подойдет. В Си `-1 / 6` дает ноль, оставляя остаток `-1`, то есть `-1 % 6` будет `-1`. Нам же нужно, чтобы операция `-1 % 6` давала пять.

В листинге 1.5 приведен код функции `identical_left`.

Листинг 1.5. Определение идентичности снежинок перемещением влево

```
int identical_left(int snow1[], int snow2[], int start) {
  int offset, snow2_index;
  for (offset =0; offset < 6; offset++) {
    snow2_index = start - offset;
    if (snow2_index < 0)
      snow2_index = snow2_index + 6;
    if (snow1[offset] != snow2[snow2_index])
      return 0;
  }
  return 1;
}
```

Обратите внимание на сходство между этой функцией и функцией из листинга 1.3. Все, что мы изменили, — это выполнили вычитание смещения вместо его добавления и изменили граничную проверку с `6` на `-1`.

Объединение функций проверки

Используя вспомогательные функции `identical_right` и `identical_left`, напишем функцию `are_identical`, которая будет сообщать, идентичны две снежинки или нет. В листинге 1.6 приведен код для этой функции. Мы проводим сравнение

снежинок с перемещениями вправо и влево для каждой возможной стартовой точки в `snow2`.

Листинг 1.6. Определение идентичности снежинок

```
int are_identical(int snow1[], int snow2[]) {
  int start;
  for (start = 0; start < 6; start++) {
    if (identical_right(snow1, snow2, start)) ❶
      return 1;
    if (identical_left(snow1, snow2, start)) ❷
      return 1;
  }
  return 0;
}
```

Вначале мы сравниваем `snow1` и `snow2`, перемещаясь вправо по `snow2` ❶. В случае их идентичности возвращается `1` (`true`). Далее идет проверка перемещением влево ❷.

Здесь есть смысл приостановиться и протестировать функцию `are_identical` на примере нескольких пар снежинок. Выполните это самостоятельно, прежде чем продолжать.

Решение 1: последовательное сравнение

Когда нам нужно сравнить две снежинки, мы применяем вместо оператора `==` функцию `are_identical`. Теперь их сопоставление стало таким же простым, как сравнение двух целых чисел.

Давайте перепишем функцию `identify_identical` из листинга 1.1 так, чтобы она использовала новую `are_identical` из листинга 1.6. Мы будем сравнивать пары снежинок и получать одно из двух сообщений, в зависимости от результата проверки их идентичности. Код приведен в листинге 1.7.

Листинг 1.7. Поиск идентичных снежинок

```
void identify_identical(int snowflakes[][6], int n) {
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = i+1; j < n; j++) {
      if (are_identical(snowflakes[i], snowflakes[j])) {
        printf("Twin snowflakes found.\n");
        return;
      }
    }
  }
  printf("No two snowflakes are alike.\n");
}
```

Функция `identify_identical` для снежинок почти полностью совпадает с ее версией для целых чисел из листинга 1.1. Все, что мы сделали, — это заменили `==` на функцию, сравнивающую снежинки.

Считывание входных данных

Текущее решение пока не закончено. Нужно еще написать код для считывания снежинок из стандартного потока ввода. Сперва вернемся к описанию задачи в начале главы. Нужно считать строку с целым числом *n*, указывающим общее число снежинок, после чего считать каждую из *n* строк как отдельную снежинку.

В листинге 1.8 приведена функция `main`, которая обрабатывает входные данные и затем вызывает `identify_identical` (листинг 1.7).

Листинг 1.8. Функция `main` для Решения 1

```
#define SIZE 100000

int main(void) {
  static int snowflakes[SIZE][6]; ❶
  int n, i, j;
  scanf("%d", &n);
  for (i = 0; i < n; i++)
    for (j = 0; j < 6; j++)
      scanf("%d", &snowflakes[i][j]);
  identify_identical(snowflakes, n);
  return 0;
}
```

Обратите внимание, что массив `snowflakes` теперь стал статическим ❶. Дело в том, что он огромен, и, если не сделать его статическим, размер необходимого пространства просто превысит объем доступной для данной функции памяти. С помощью ключевого слова `static` массив помещается в отдельную область памяти, где его размер уже не вызовет сложностей. Тем не менее эту возможность нужно использовать осторожно. Стандартные локальные переменные инициализируются при каждом вызове функции, а статические сохраняют значение, полученное от предыдущего вызова функции (см. «Ключевое слово static» на с. 21).

Также заметьте, что созданный массив может содержать до `100000` снежинок ❶. Вас может обеспокоить такая растрата памяти. Что, если на входе окажется всего несколько снежинок? В конкурсных задачах считается нормальным жестко прописывать требования памяти для максимального количества входных данных, так как ваше решение наверняка будут проверять методом стресс-тестирования.

Концовка функции проста. Мы считываем количество снежинок с помощью `scanf` и используем это число для управления количеством итераций цикла `for`. В каждой

итерации перебор происходит шесть раз, по разу для каждого целого числа. Далее вызывается `identify_identical`, которая выводит результат.

Объединив функцию `main` с другими ранее написанными функциями, мы получаем завершенную программу, которую можно отправлять на проверку. Попробуйте... и вы получите ошибку «Time-Limit Exceeded» (превышен лимит времени). Похоже, здесь еще есть над чем поработать.

Выявление проблемы

Наше первое решение оказалось слишком медленным, и мы получили ошибку «Time-Limit Exceeded». Проблема кроется в двух вложенных циклах `for`, которые сравнивают все снежинки между собой. В результате, когда число n оказывается большим, выполняется огромное число операций сравнения.

Давайте определим, сколько же таких операций выполняет наша программа. Поскольку мы сравниваем каждую пару снежинок, можно переформулировать этот вопрос в отношении именно пар. Например, если даны четыре снежинки 1, 2, 3 и 4, то наша схема выполняет шесть сравнений: снежинок 1 и 2, 1 и 3, 1 и 4, 2 и 3, 2 и 4, а также 3 и 4. Каждая пара формируется выбором одной из n снежинок в качестве первой и одной из оставшихся $n-1$ снежинок в качестве второй.

Для каждого из n решений для первой снежинки есть $n - 1$ решений для второй. В общей сложности это дает $n(n - 1)$ решений. Но формула $n(n - 1)$ удваивает фактическое количество сравнений снежинок, то есть, к примеру, включает и сравнение 1 с 2, и сравнение 2 с 1. Наше же решение сравнивает их только один раз, поэтому мы можем добавить в формулу делитель 2, получив $n(n - 1)/2$ сравнений для n снежинок.

На первый взгляд такое вычисление не выглядит медленным, но давайте подставим в $n(n - 1)/2$ разные значения n. Если взять 10, получится $10(9)/2 = 45$. С выполнением 45 сравнений любой компьютер справится за считанные миллисекунды. Что насчет 100? Теперь получается 4950: тоже никаких проблем. Выходит, что для небольших значений n все работает хорошо, но условие задачи говорит, что количество снежинок может достигать 100 000. Давайте подставим в формулу $n(n - 1)/2$ это значение. Теперь получается 4 999 950 000 операций сравнения. Если выполнить тест для 100 000 снежинок на обычном ноутбуке, то вычисление может занять до четырех минут. А по условиям задачи необходимо уложиться в 2 секунды. В качестве ориентира можно считать, что современный компьютер способен выполнять в секунду около 30 миллионов операций. Это значит, что попытка уложить 4 миллиарда сравнений в 2 секунды, безусловно, обречена на провал.

Если раскрыть скобки в $n(n - 1)/2$, получится $n^2/2 - n/2$. Самая большая степень здесь — это квадрат. Поэтому создатели алгоритма и назвали его $O(n^2)$, или

алгоритмом с *квадратичным временем*. $O(n^2)$ произносится как «О-большое от n в квадрате» и является показателем скорости, с которой количество работы растет по отношению к размеру задачи. Краткая информация об «О-большом» приведена в приложении А.

Нам приходится проводить так много сравнений, потому что идентичные снежинки могут находиться в любых частях массива. Если найти способ собирать похожие снежинки в группы, то можно будет быстрее проводить сравнение. Для группировки похожих снежинок можно попробовать отсортировать массив.

Сортировка снежинок

В Си есть библиотечная функция `qsort`, позволяющая легко упорядочить массив. Главное, что нам потребуется, — это функция сравнения, которая получает ссылки на два сортируемых элемента и возвращает отрицательное целое число, если первый элемент меньше второго, `0`, если они равны, и положительное целое число, если первый больше второго. Можно использовать `are_identical` для определения, являются ли две снежинки одинаковыми, и если да, то возвращать `0`.

Как при этом определить, что одна снежинка больше или меньше другой? Нужно просто принять соответствующее правило. Например, можно установить, что «меньшей» снежинкой считается та, чей первый отличающийся элемент меньше, чем соответствующий элемент другой снежинки. Это реализуется в листинге 1.9.

Листинг 1.9. Функция сравнения для упорядочивания

```
int compare(const void *first, const void *second) {
  int i;
  const int *snowflake1 = first;
  const int *snowflake2 = second;
  if (are_identical(snowflake1, snowflake2))
    return 0;
  for (i = 0; i < 6; i++)
    if (snowflake1[i] < snowflake2[i])
      return -1;
  return 1;
}
```

К сожалению, упорядочивание таким образом не поможет решить проблему. Ниже приведен тестовый пример с четырьмя снежинками, который наверняка будет решен с ошибкой:

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