Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №8 по курсу «Дискретный анализ»

Студент: И. П. Моисеенков
Преподаватель: Н. С. Капралов
Группа: М8О-208Б-19

Дата: 04.05.2021
Оценка:
Подпись:

Москва, 2021

Лабораторная работа №8

Разработать жадный алгоритм решения задачи, определяемой своим вариантом. Доказать его корректность, оценить скорость и объём затрачиваемой оперативной памяти.

Реализовать программу на языке C или C++, соответсвующую построенному алгоритму.

Задача: Дана последовательность длины **N** из целых чисел **1, 2, 3**. Необходимо найти минимальное количество обменов элементов последовательности, в результате которых последовательность стала бы отсортированной.

Формат входных данных: Число **N** на первой строке и **N** чисел на второй строке.

Формат результата: Минимальное количество обменов.

1 Описание

Согласно [1], жадный алгоритм - это алгоритм, который на каждом шаге делает локально наилучший выбор в надежде, что итоговое решение будет оптимальным.

Данная задача как раз решается с помощью жадного алгоритма.

Назовём *зоной* i, $i = 1, 2, 3$ подмножество массива, в котором должны находиться только числа i для того, чтобы он был отсортированный. Для определения границ зон при считывании данных нам необходимо подсчитать, какое число сколько раз встречается. Например, если на вход поступает массив $3, 2, 1, 1$, то зона 1 - первые два элемента, зона 2 - третий элемент, зона 3 - последний элемент.

Будем проходить по массиву слева направо. Как только мы обнаружим элемент, который находится не в своей зоне, мы будем сразу же заменять его на нужный элемент, взятый из другой зоны. Таким образом, у нас могут возникнуть следующие ситуации:

- Мы находимся в зоне 1, но обнаружили число 2. Значит нужно найти во второй (а если во второй нет, то в третьей) зоне число 1 и поменять числа местами.
- Мы находимся в зоне 1, но обнаружили число 3. Значит нужно найти в третьей (а если в третьей нет, то во второй) зоне число 1 и поменять числа местами.
- Мы находимся в зоне 2, но обнаружили число 3. Значит нужно найти в третьей зоне число 2 и поменять числа местами.

Других вариантов быть не может. Мы гарантируем, что весь массив до текущей позиции был отсортирован.

Если динамически обновлять позиции, с которыми мы будем обмениваться числами, то сложность алгоритма составляет $O(n)$ по времени и памяти

2 Исходный код

Для определения границ между зонами создадим массив *counts*, в котором будет вестись подсчет количества каждого числа. *counts[i]* - количество чисел $i + 1$ в исходных данных.

Таким образом, можем определить границы зон:

- $[0; counts[0])$ - зона 1
- $[counts[0]; counts[0] + counts[1])$ - зона 2
- $[counts[0] + counts[1]; N)$ - зона 3, N - длина массива.

Для того, чтобы за константу находить числа для обмена, введем три переменные - $pos2\_1, pos3\_1, pos3\_2$. $posi\_j$ - позиция числа j в зоне i. В первом и третьем случае мы всегда берем самое левое подходящее число. Во втором случае - самое правое. Это нужно для того, чтобы все 3 оказались в правой части массива (в зоне 3).

Для обновления значений этих переменных мной была реализована функция *getNextPos()*. Все вызовы этой функции будут суммарно работать за линейное время, так как для каждой из трех переменных мы в худщем случае выполним проход по всему массиву.

В функции *main()* реализован алгоритм, описанный выше.

```
1  #include <iostream>
2  #include <vector>
3
4  long long getNextPos(short num, long long curPos, std::vector<short> &nums, bool
       reverseOrder = false) {
5      if (nums.empty() || curPos < 0 || curPos >= nums.size()) {
6          return 0;
7      }
8      if (!reverseOrder) {
9          while (nums[curPos] != num && curPos + 1 < nums.size()) {
10             ++curPos;
11         }
12     } else {
13         while (nums[curPos] != num && curPos - 1 >= 0) {
14             --curPos;
15         }
16     }
17     return curPos;
18 }
19
20 int main() {
21     std::ios_base::sync_with_stdio(false);
22     std::cin.tie(nullptr);
23     const int MAX_NUM = 3;
```

```
24
25      long long n;
26      std::cin >> n;
27      std::vector<short> nums(n);
28      std::vector<long long> counts(MAX_NUM);
29      for (long long i = 0; i < n; ++i) {
30          std::cin >> nums[i];
31          ++counts[nums[i] - 1];
32      }
33
34      long long swaps = 0;
35
36      // posi_j - the position of number j in a zone i
37      long long pos2_1 = getNextPos(1, counts[0], nums);
38      long long pos3_1 = getNextPos(1, n - 1, nums, true);
39      long long pos3_2 = getNextPos(2, counts[0] + counts[1], nums);
40
41      for (int i = 0; i < n; ++i) {
42          if (i < counts[0] && nums[i] != 1) {
43              // found number 2 or 3 which must be replaced by 1
44              if (nums[i] == 2) {
45                  std::swap(nums[i], nums[pos2_1]);
46                  ++swaps;
47                  pos2_1 = getNextPos(1, pos2_1, nums);
48              } else {
49                  std::swap(nums[i], nums[pos3_1]);
50                  ++swaps;
51                  pos3_1 = getNextPos(1, pos3_1, nums, true);
52              }
53          } else if (counts[0] <= i && i < counts[0] + counts[1] && nums[i] != 2) {
54              // found number 3 which must be replaced by 2
55              // numbers 1 can't be here, they are already placed correctly
56              std::swap(nums[i], nums[pos3_2]);
57              ++swaps;
58              pos3_2 = getNextPos(2, pos3_2, nums);
59          }
60      }
61
62      std::cout << swaps << "\n";
63  }
```

3 Консоль

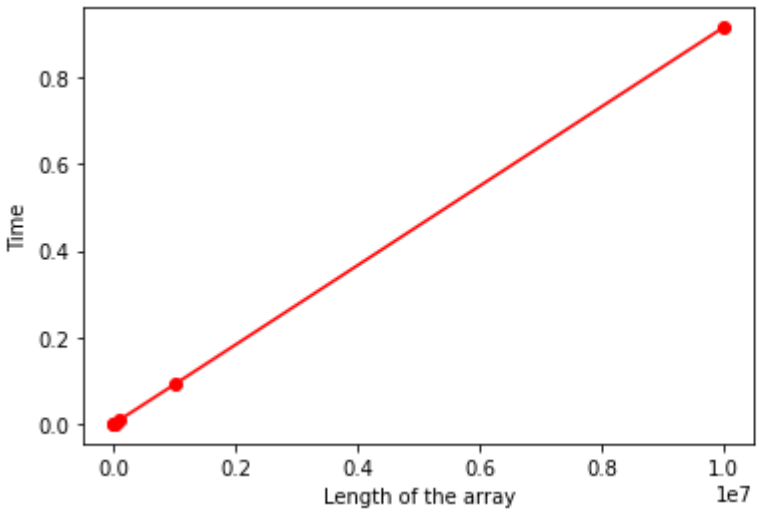
```
mosik@LAPTOP-69S778GL:~/da_lab8$ g++ -pedantic -Wall -std=c++11 -Werror
-Wno-sign-compare -O2 -lm main.cpp
mosik@LAPTOP-69S778GL:~/da_lab8$ ./a.out
3
3 2 1
1
mosik@LAPTOP-69S778GL:~/da_lab8$ ./a.out
4
3 2 1 1
2
mosik@LAPTOP-69S778GL:~/da_lab8$ ./a.out
5
2 3 1 2 1
2
```

4 Тест производительности

Убедимся, что построенный жадный алгоритм действительно имеет линейную сложность. Для этого замерим время работы программы на нескольких тестах: с последовательностями длин 100, 1000, 10000, 100000, 1000000, 10000000.

```
mosik@LAPTOP-69S778GL:~/da_lab8$ ./a.out <test1e2
39
Time: 0.000198 s
mosik@LAPTOP-69S778GL:~/da_lab8$ ./a.out <test1e3
350
Time: 0.0002908 s
mosik@LAPTOP-69S778GL:~/da_lab8$ ./a.out <test1e4
3345
Time: 0.0011082 s
mosik@LAPTOP-69S778GL:~/da_lab8$ ./a.out <test1e5
33434
Time: 0.0094662 s
mosik@LAPTOP-69S778GL:~/da_lab8$ ./a.out <test1e6
333194
Time: 0.0914062 s
mosik@LAPTOP-69S778GL:~/da_lab8$ ./a.out <test1e7
3331149
Time: 0.915165 s
```

Отметим полученные результаты на графике.



По графику видно, что время работы программы возрастает прямо пропорционально объему входных данных, значит сложность программы действительно равна $O(n)$.

5 Выводы

Жадный алгоритм - это довольно простой и действенный метод решения задач на оптимизацию, который может быть полезен там, где не справляется динамическое программирование. Однако не все задачи могут быть решены с использованием жадных алгоритмов (типичный пример - задача о дискретном рюкзаке).

Жадные алгоритмы часто используются на практике. Но так как в реальном мире приходится работать с данными огромного размера, то вычислительных мощностей для точного алгоритма может не хватать. Поэтому применяются приближенные жадные алгоритмы, которые работают гораздо быстрее, но дают приблизительный ответ вместо точного.

Список литературы

[1] *Жадные алгоритмы — Хабр.*
URL: `https://habr.com/ru/post/120343/` (дата обращения: 04.05.2021).