АиСД | SET-2 | A4

Потякин Арсений

TODO:

1. Разработайте DaC-алгоритм CINV, временная сложность которого должна соответствовать $O(n \log n)$, для подсчета степени упорядоченности массива путем вычисления количества необходимых перестановок. Описание алгоритма представьте в любом удобном формате. Опишите суть шагов DIVIDE, CONQUER и COMBINE, а также представьте рекуррентное соотношение для $T(n)$ и обоснуйте соответствие требуемой асимптотической верхней границе временной сложности. Проанализируйте, возвращает ли разработанный вами алгоритм CINV минимальное количество необходимых инверсий.

2. Элементы a_i и a_j массива A назовем значительно инвертированными, если $i < j$, но $a_i > 2a_j$. Какие изменения и доработки необходимо внести в алгоритм CINV, разработанный на предыдущем шаге, чтобы в качестве степени упорядоченности велся подсчет количества пар значительно инвертированных элементов? Например, в массиве $A = [1, 3, 4, 2, 5]$ нет значительно переставленных элементов, а в массиве $A = [5, 3, 2, 4, 1]$ всего 4 пары значительно переставленных элементов: $5 \leftrightarrow 1, 5 \leftrightarrow 2, 4 \leftrightarrow 1, 3 \leftrightarrow 1$. Асимптотическая верхняя граница временной сложности измененного алгоритма должна остаться неизменной.

Task 1: (Разработка алгоритма CINV)

Описание: Алгоритм реализован аналогично сортировке слиянием (aka Merge Sort), то есть в основе идеи лежит разделение массива на подмассивы и рекурсивный подсчнт количество инверсий в каждом подмассиве, после чего при каждом слиянии двух подмассивов учесть инверсии, которые возникают в разных подмассивах.

Описание шагов:

DIVIDE: Деление массива на два подмассива (примерно равных частей), до тех пор, пока не дойдем до базового случая, когда подмассив содержит ровно 1 элемент.

CONQUER: Рекурсивно применяется основная логика алгоритма к

подмассивам для подсчета количества инверсий.
COMBINE: Во время слияния двух подмассивов учитываем инверсии, возникающие между двух подмассивов. Проходя по двум подмассивам A_1 и A_2, если элемент из A_1 больше, чем элемент из A_2, то для каждого такого случая инкрементируем количество инверсий.
Рекуррентное соотношение: $T(n) = 2T(\frac{n}{2}) + O(n)$, где $2T(\frac{n}{2})$ - рекурсивное решение для двух подмассивов, $O(n)$ - время, затраченное на подсчет инверсий и слияние двух отсортированных подмассивов. Используя мастер-теорему можно сказать, что $T(n) = O(n \log n)$, т.к. $a = 2, b = 2, k = 1$, т.е. $1 = \log_2 2 \Longrightarrow 1 = 1$, что соответствует виду $O(n^k \cdot f(n) \cdot \log n)$.
Код средствами языка C++:

```
#include <iostream>
#include <vector>

using namespace std;

long long mergeAndCount(vector<int>& arr,
             vector<int>& temp,
             int left,
             int mid,
             int right) {
 int i = left;
 int j = mid + 1;
 int k = left;
 long long invCount = 0;

 while (i <= mid && j <= right) {
  if (arr[i] <= arr[j]) {
   temp[k++] = arr[i++];
  } else {
   temp[k++] = arr[j++];
   invCount += (mid - i + 1);
  }
 }

 while (i <= mid)
 temp[k++] = arr[i++];
```

```
 while (j <= right)
 temp[k++] = arr[j++];

 for (i = left; i <= right; i++)
 arr[i] = temp[i];

 return invCount;
}

long long countAndMerge(vector<int>& arr,
             vector<int>& temp,
             int left,
             int right) {
 long long invCount = 0;
 if (left < right) {
  int mid = (left + right) / 2;

  invCount += countAndMerge(arr, temp, left, mid);
  invCount += countAndMerge(arr, temp, mid + 1, right);

  invCount += mergeAndCount(arr, temp, left, mid, right);
 }
 return invCount;
}

long long CINV(vector<int>& arr) {
 vector<int> temp(arr.size());
 return countAndMerge(arr, temp, 0, arr.size() - 1);
}

int main() {
 vector<int> arr1 = {1, 3, 4, 2, 5};
 vector<int> arr2 = {5, 4, 3, 2, 1};
 vector<int> arr3 = {1, 2, 3, 4, 5};
 vector<int> arr4 = {10, 20, 30, 25, 15};
 cout << "Количество инверсий: " << CINV(arr1) << '\n';
 cout << "Количество инверсий: " << CINV(arr2) << '\n';
 cout << "Количество инверсий: " << CINV(arr3) << '\n';
 cout << "Количество инверсий: " << CINV(arr4) << '\n';
```

```
 return 0;
}
```

В конце проведены тесты со следующим выводом:

```
Количество инверсий: 2
Количество инверсий: 10
Количество инверсий: 0
Количество инверсий: 4
```

Как видно из тестов, алгоритм возвращает минимальное число инверсий.

Task 2: (Доработка алгоритма)
DIVIDE: Этот шаг остается неизменным.
CONQUER: Этот шаг тоже остается неизменным.
COMBINE: Заменим сравнение $a_i > a_j$ на $a_i > 2a_j$, а также во время слияния двух подмассивов ищем количество элементов в правом массиве, которые меньше чем половина текущего элемента в левом подмассиве.
Код средствами языка C++:

```
#include <iostream>
#include <vector>

using namespace std;

long long mergeAndCount(vector<int>& arr,
vector<int>& temp,
int left,
int mid,
int right) {
 int i = left;
 int j = mid + 1;
 int k = left;
 long long invCount = 0;
```

```
 for (int i = left, j = mid + 1; i <= mid; i++) {
  while (j <= right && arr[i] > 2 * arr[j]) {
   j++;
  }
  invCount += (j - (mid + 1));
 }

 i = left;
 j = mid + 1;
 while (i <= mid && j <= right) {
  if (arr[i] <= arr[j]) {
   temp[k++] = arr[i++];
  } else {
   temp[k++] = arr[j++];
  }
 }

 while (i <= mid)
 temp[k++] = arr[i++];

 while (j <= right)
 temp[k++] = arr[j++];

 for (i = left; i <= right; i++)
 arr[i] = temp[i];

 return invCount;
}

long long countAndMerge(vector<int>& arr,
vector<int>& temp,
int left,
int right) {
 long long invCount = 0;
 if (left < right) {
  int mid = (left + right) / 2;

  invCount += countAndMerge(arr, temp, left, mid);
  invCount += countAndMerge(arr, temp, mid + 1, right);
```

```
    invCount += mergeAndCount(arr, temp, left, mid, right);
   }
   return invCount;
  }

  long long CINV(vector<int>& arr) {
   vector<int> temp(arr.size());
   return countAndMerge(arr, temp, 0, arr.size() - 1);
  }

  int main() {
   vector<int> arr = {5, 3, 2, 4, 1};
   cout << "Количество значительно инвертированных пар: ";
   cout << CINV(arr) << '\n';

   return 0;
  }
```

Изменения: в функции mergeAndCount добавлен новый цикл (не влияющий на временную сложность алгоритма)
Таким образом, измененный функционал никак не отразился на временной сложности алгоритма.