Теоретическое решение задачи B.

Алгоритм решения и доказательства его правильности.

Докажем, что эту задачу, для исходного вектора востребованности *data_in*, можно свести к нахождению максимальной неубывающей подпоследованости у обратного к заданному вектору. Поскольку, для любой пары *(i, j)* – порядковые номера в векторе *data_in*, которые взяты слева на право, в подпоследованости должны выполняться условия того, что *data_in[i] ≥ data_in[j]* и *i > j*, то обернув вектор мы получим подобные соотношения: *data_in[j] ≥ data_in[i]* и *i > j*. Таким образом мы доказали, что эта задача сводима к более привычной.

Сведенная задача хорошо решается динамическим программированием. Построим следующую динамику: пусть *data[i]* (*i* = 0…n) – это число, на которое оканчивается невозрастающая последовательность длины *i*, а если таких чисел несколько, то наибольшее из них. Изначально будем предполагать, что *data[0]* = −∞, а все остальные элементы *data[i]* = ∞.

Заметим два важных свойства этой динамики:

1) *data[i]≥data[i-1]* для всех *i=1...n*;

2) каждый элемент *data_in[i]* обновляет максимум один элемент *data[j]*.

Это означает, что при обработке очередного *data_in[i]* мы можем с помощью двоичного поиска в массиве *data* найти первое число, которое строго больше текущего *data_in[i]*, и обновить его. Так же, для восстановления ответа, в конце каждой итерации мы будем, при необходимости, обновлять переменную *length* – длина наибольшей, на данный момент, подпоследовательности, ведь если значение *length* меньше чем длина текущей подпоследовательности, тогда *length* необходимо обновить.

Для восстановления ответа будем поддерживать заполнение двух массивов: *pos* и *prev*. В *pos[i]* будем хранить индекс элемента, на который заканчивается оптимальная подпоследовательность длины *i*, а в *prev[i]*— позицию предыдущего элемента для *data_in[i]*.

Временная сложность.

Алгоритм состоит из n итераций в каждой из которой мы используем алгоритм двоичного поиска, который, как известно, работает за *O(log n)*.

Итоговая временная сложность – *O(n * log n)*.

Затраты памяти.

Для реализации описанного выше алгоритма, требуется:

1. Исходный массив размером *n*;
2. Массив окончаний последовательностей(*data*) размером *n + 1*:
3. Массив предыдущих позиций элементов(*prev*) размером *n*;
4. Массив индексов элементов, на который заканчивается максимальная подпоследовательность(*pos*) размером *n + 1*;
5. Массив ответов(*answer*) размером *length*(*n ≥ length*).

Так же необходимо константное число вспомогательных переменных (таких как temp, amount, cur, . . .).

Таким образом, итоговые затраты памяти - *O(n + n + 1 + n + n + 1 + length) = = O(4*n + 2 + length) = O(n).*