

# Стопка карточек

## Решение

Ну, конечно же, динамическое программирование...

Мы хотим узнать, какое максимальное количество карточек можно собрать, имея заданную сумму денег  $S$ . Вот и попробуем добавлять карточки по одной, двигаясь вдоль стопки сверху вниз, и вычислять при этом нужную нам величину. Сразу становится понятно, что для вычислений понадобится знать и максимальное количество карточек, которые можно собрать, располагая  $q$  талерами, для  $q=1, 2, \dots, S$ . Однако на этом пути мы довольно быстро столкнёмся с серьёзными трудностями. Коротко: каждый раз мы имеем 2 варианта – либо забрать карточку, либо отбросить её; тогда мы очень быстро приходим к тому, что для каждой суммы  $q$  надо хранить дополнительную информацию, чтобы учесть стоимость отброшенных карточек, в общем задача расплзается, и удержать её в руках трудно. Я, тем не менее, советую обдумать и этот вариант, а сам ускользну от его изложения и перейду к другому, двойственному, решению.

Двойственному в том смысле, что мы будем искать не наибольшее количество карточек, которые можно собрать, располагая некоторой суммой денег, а будем вычислять наименьшую сумму денег, необходимую для того, чтобы собрать  $i$  карточек,  $i=1, 2, \dots, N$ . Довольно часто такой приём оказывается эффективным. Вот и сейчас...

Итак, обозначим  $A[k,i]$  наименьшую сумму денег, которая необходима для того, чтобы собрать у себя  $i$  карточек, при условии, что мы просмотрели верхние  $k$  карточек в стопке. Под словом "просмотрели" имеется в виду, что каждую из  $k$  карточек мы либо забрали, либо отбросили, даже если в вычисленную сумму войдёт плата в 1 талер за отбрасывание одной или нескольких последних карточек в стопке.

Вычислить  $A[k,i]$  просто:

$$A[k,i] = \min ( A[k-1,i]+1, A[k-1,i-1] + price[k] ),$$

$k=1, 2, \dots, N, \quad i=1, 2, \dots, k.$

где  $price[k]$  - цена  $k$ -ой карточки. Первая величина соответствует отбрасыванию  $k$ -ой карточки, вторая получается, если мы эту карточку забираем.

Кроме того, положим  $A[0,0]=0$ , и  $A[0,i]=\infty$ ,  $i=1, 2, \dots, N$ .

Остаётся только взять для каждого  $i$  от 1 до  $N$  наименьшее значение ( в прилагаемой программе эта величина хранится в массиве  $minA$ ):

$$minA[i] =$$

Ответ задачи даёт наибольшее  $i$ , для которого  $minA[i] \leq S$ .

Легко видеть, что сложность полученного алгоритма равна  $O(N^2)$ .

Пару слов о памяти. Понятно, что для вычисления всех  $A[k,i]$  нам не понадобится двумерный массив - вполне хватит двух одномерных массивов для хранения  $A[k-1,i]$  и  $A[k,i]$ . А если вычислять  $A[k,i]$  справа налево (т.е. в порядке убывания  $i$ ), то и одного - ведь для вычисления  $A[k,i]$  требуются только величины  $A[k-1,j]$  с  $j < i$ .

Реализация такого решения приведена в файле `pile1.pas`. Решение с таким быстроедействием получало полные баллы.

Итак, метод в очередной раз восторжествовал. Барабаны гремят, трубы гудят, а в душе под эту какофонию расцветают незабудки.

Драматургия подразумевает, что сейчас появится большое «но». Да, решение со сложностью  $O(N^2)$  получало полные баллы, но только учитывая учебный характер RunSite, и то, что тема у нас сейчас именно динамическое программирование. А вот и ожидаемое большое

**НО!** Не динамическим программированием единым ... ☺

Задача интересна тем, что в ней можно придумать очень много разных решений. В том числе и различных по сложности. Из множества возможных решений я изложу только ещё одно. Так что останется ещё много возможностей поиграть с этой задачей...

В файле `pile2.pas` приводится реализация решения со сложностью  $O(N \cdot \log N)$ . И оно совсем несложное. Коротко опишу основные моменты этого решения.

Основная идея совсем простая: вычисляем `MinCost(k)` – минимальную сумму, необходимую для того, чтобы собрать у себя  $k$  карточек. А затем выполняем бинарный поиск, чтобы найти наибольшее количество карточек, которые можно собрать, располагая суммой  $S$  талеров. Всего потребуется  $O(\log N)$  вычислений `MinCost(k)`. А для вычисления `MinCost(k)` достаточно  $O(N)$  операций.

Основной шаг вычисления `MinCost(k)` такой: последовательно для  $i=N, N-1, N-2, \dots, k$  вычисляем

минимальное количество талеров, необходимое для того, чтобы собрать у себя ровно  $k$  карточек, причём последняя взятая карточка находилась в стопке на  $i$ -ом (сверху) месте.

Среди всех полученных значений выбираем наименьшее.

Ясно, что если последняя взятая карточка находилась на  $i$ -ом месте, то мы должны, кроме неё, взять  $(k-1)$  самых дешёвых карточек из тех карточек, которые лежали выше  $i$ -й. Иначе говоря,

Минимальные затраты на то, чтобы забрать ровно  $k$  карточек, причём последняя взятая карточка находится на  $i$ -ом сверху месте =  
= стоимость  $(k-1)$  самых дешёвых карточек среди первых  $(i-1)$  карточек + стоимость  $i$ -й сверху карточки +  $(i-k)$ .

Последнее слагаемое – это стоимость отброшенных карточек.

Итак, начинаем с  $i=N$ . Вычисляем `sbase` – стоимость  $(k-1)$  самых дешёвых карточек. Отмечаем в булевском массиве `b`, что эти карточки уже заняты – их нельзя больше добавлять в множество  $B$ , где  $B$  – это множество  $(k-1)$  самых дешёвых карточек из первых  $i$  карточек. Далее, если  $i$ -я карточка уже занята (т.е. она в данный момент входит в

множество  $B$ ), то удаляем её из множества  $B$  и добавляем в множество  $B$  самую дешёвую из тех карточек, которые вообще могут быть в него добавлены, пересчитывая соответственно величину  $sbase$ , и вычисляем

Минимальные затраты на то, чтобы забрать ровно  $k$  карточек, причём последняя взятая карточка находится на  $i$ -ом сверху месте =  
 $= sbase + \text{стоимость } i\text{-й сверху карточки} + (i-k)$ .

После этого отмечаем, что  $i$ -ю сверху карточку больше нельзя добавлять в множество  $B$  (хотя, возможно, она уже была отмечена), и переходим к следующему  $i$ .

Разумеется, нам необходимо отсортировать стоимости карточек по возрастанию. Это действие выполняется один раз в самом начале и требует времени  $O(N \cdot \log N)$ .

Всё сказанное звучит жутковато, но на самом деле очень несложно – рассмотрите внимательно текст программы.