# Задача A. Псевдопростые числа

*Автор задачи - К.Батузов, авторы разбора - К. Батузов, М. Трухина*

Заметим, что псевдопростых чисел, не превосходящих $10^9$, очень мало. Их всего 6: 2, 3, 5, 29, 869, 756 029. Следующее число уже равно 571 580 604 869. Все псевдопростые числа попарно взаимно просты (это очевидно следует из формулы, для их вычисления). Значит, число $X$ можно раскладывать на их произведение жадно: пока $X$ делится на 2, делим его на 2, затем так же делаем для 3, 5, ..., 756 029. Если в конце осталась единица — разложить можно, иначе — нет. Для того, чтобы получить разложение, необходимо запомнить, сколько раз мы поделили $X$ на каждое псевдопростое число.

# Задача B. Сортировка «с конфеткой»

*Автор задачи и разбора - В.Антонов*

Решение задачи на 1 балл подразумевает любую разумную реализацию описанного в условиях алгоритма. Чтобы решить задачу на 2 балла, требовалось придумать эффективный способ проверки наличия пары чисел в наборе, заданном во входных данных. Один из возможных вариантов — завести квадратную матрицу $A$ размером $n$ х $n$, где $A_{ij}$ будет равно 1, если пара “$i$ $j$” есть во входном файле, и 0, если нет. Чтобы завести матрицу размером 5000 x 5000, где элемент матрицы имеет размер в 1 байт, потребуется около 25 мегабайт, что превышает ограничения задачи по памяти. Поэтому, для хранения нулей или единиц будем использовать 1 бит, таким образом, количество требуемой памяти уменьшится почти в 8 раз, что удовлетворят условиям задачи.

Другое из возможных решений использует списки. Для каждого элемента $i$ будем хранить все такие $j$, что пара “$i$ $j$” присутствует во входных данных. Тогда для проверки наличия пары в заданных, потребуется порядка $N$ операций. Всего проверок наличия пар будет порядка $N^2$, тогда для выполнения программы потребуется порядка $N^3$ операций, что не будет укладываться в отведённый интервал времени. Попробуем провести оценку количества операций по-другому. Каждая пара из входного файла будет проверена не более чем $N$ раз. Поэтому, для выполнения программы потребуется порядка $N*M$ операций, что укладывается в ограничения задачи по времени.

## Задача C. Электричка

*Автор задачи - В.Матюхин, автор разбора - Р.Жуйков*

Пусть электричка состоит из n вагонов, тогда в случае нумерации вагонов с головы i-й вагон имеет номер i, а в случае нумерации с хвоста i-й вагон имеет номер $n-i+1$. Отсюда следует, что если во входных данных $i \neq j$, то имеет место нумерация с хвоста и n легко найти из равенства $n-i+1 = j$ (получаем $n = i+j-1$). В случае $i = j$ определить количество вагонов невозможно, так как при нумерации вагонов с головы такая ситуация могла произойти при любом $n \geq i$.

# Задача D. Обувной магазин

*Автор задачи - В.Гуровиц, автор разбора - Р.Жуйков*

Задача решается «жадным» алгоритмом. Предварительно отсортируем пары обуви из магазина по возрастанию размеров. Подойдет любой из известных вам методов сортировки, в том числе сортировка подсчетом. После чего будем просматривать пары ботинок по очереди и пытаться их надеть. Если надеть удается, то оставляем текущие ботинки надетыми и переходим к следующей паре. В противном случае переходим к рассмотрению следующей пары.

Покажем, что мы получили максимально возможное число ботинок. Представим надетые ботинки последовательностью их размеров, обозначим ее $\{b_i\}$. Пусть $\{a_i\}$ – максимально возможная последовательность ботинок. Рассмотрим первый ботинок, его размер – $a_1$. Так как $b_1$ – минимально возможный размер ботинка, который можно надеть, то $b_1 \leq a_1$. Если $a_1 = b_1$, перейдем к следующему шагу, иначе $b_1 < a_1$. Заметим, что согласно выбору последовательности $\{a_i\}$ $a_2–a_1 \geq 3$. Но тогда $a_2–b_1 \geq 3$, следовательно мы можем заменить $a_1$ на $b_1$ в последовательности a, не испортив ее и перейти к следующему шагу. Заметим, что на каждом шаге для $a_i$ найдется $b_i$, так как $a_i–b_{i-1} \geq 3$. С помощью таких преобразований мы из последовательности a получим последовательность $\{b_i\}$, но мы брали последовательность $\{a_i\}$ таким образом, чтобы ее длина была максимальна, значит, длина $\{b_i\}$ так же максимальна.

# Задача E. Перекраска клеток

*Автор задачи и разбора - А.Шестимеров*

Самый простой способ решения этой задачи –хранить в массиве, соответствующему полю, цвет клетки и

- при покраске менять цвет
- при запросе проходить вверх, вниз, влево и вправо пока не встретится белый элемент

При таком подходе, каждый запрос покраски будет выполнен за 1 операцию, а запрос соседей – N+M, например, если осталась одна неокрашенная клетка в углу. Значит, сложность такого решения получится O(K*(N+M)). При заданных ограничениях такое решение может набрать только 1 балл.

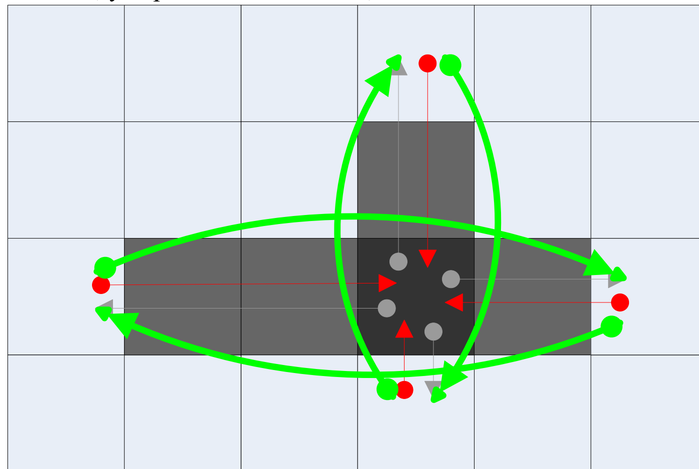
Для решения задачи на 2 балла необходимо уменьшить сложность поиска соседей. Причём достаточно оптимизировать поиск левых и правых соседей (т. к. N<=20). Одна из команд-участниц, реализовала для этого для каждой строки дерево отрезков – структуру, которая позволяет за O(log M) покрасить клетку и за O(log M) – найти число покрашенных клеток в любой строке на любом отрезке. Тогда, можно узнать соседей за $O(\log^2 M)$ (log M – запрос на отрезке и log M двоичный поиск отрезка на котором одна неокрашенная крайняя клетка.

Такой подход позволяет получить 2 балла, но существует более простое и быстрое решение.

Будем хранить для каждой клетки цвет и координаты её ближайших белых соседей:

```
  TElem = record
    color: 0..1;
    l,r,u,d:word; // номер столбца левого и правого соседа и строки верхнего и
нижнего. При этом ссылки у крайних клеток чтобы не выходить за границы
массива можно сделать равным 0.
  end;
```

Тогда за O(1) можно будет получить соседей. А при перекраске клетки надо у левого поменять ссылку на правого соседа, у верхнего на нижнего, и т.д.



Процедура покраски клетки (i,j)

```
Procedure Color(i,j:integer);
var
  rj,lj,di,ui:integer;
begin
  a[i,j].color := B;     // красим клетку
  lj := a[i,j].l;
  rj := a[i,j].r;
  ui := a[i,j].u;
  di := a[i,j].d;
  a[i,lj].r:=rj;           // обновляем ссылки на соседей
  a[i,rj].l:=lj;
  a[ui,j].d:=di;
```

```
  a[di,j].u:=ui;
  count:=count+1;
end;
```
Процедура вывода соседей:
```
procedure Print(i,j:integer);
var
  count:integer;
begin
  count:=0;
  if a[i,j].l<>0 then inc(count);
  if a[i,j].r<>0 then inc(count);
  if a[i,j].u<>0 then inc(count);
  if a[i,j].d<>0 then inc(count);
  writeln(count);
  if a[i,j].l<>0 then writeln(i,' ',a[i,j].l);
  if a[i,j].r<>0 then writeln(i,' ',a[i,j].r);
  if a[i,j].u<>0 then writeln(a[i,j].u,' ',j);
  if a[i,j].d<>0 then writeln(a[i,j].d,' ',j);
end;
```

Ещё один способ обновления ссылок – обновлять не при покраске, а при рекурсивном поиске соседей. В этом случае, если ссылка должна обновиться, то она обновится один раз и время всех запросов всё равно останется K*O(1):
```
function GetLeft(i,j:integer):integer;
begin
  if a[i,j].color=0 then
    GetLeft:=j
  else begin
          if (a[i,j].l<>0) then
                  a[i,j].l:=GetLeft(i,a[i,j].l);
          GetLeft:=a[i,j].l;
  end;
end;
```
Это пример рекурсивной функции, для получения левого соседа. Например, для неграничной клетки (i,j) левый сосед будет иметь координаты (i,getleft(i,j-1)). Этот способ очень похож, на тот, что используется в структуре данных для реализации систем непересекающихся множеств.
Итак, из 5 команд успешно сдавших эту задачу три команды предложили одну из реализаций деревьев отрезков, одна - систему непересекающихся множеств, и лишь одна команда реализовала простое обновление ссылок.

# Задача F. Шахматы

*Автор задачи - В.Гуровиц, автор разбора - М.Бабенко*

Данная задача решается стандартным методом "перебор с возвратом" ("backtracking"). А именно, требуется реализовать функцию, которая по переданной ей в параметрах конфигурации поля выясняет, существует ли последовательность ходов (взятий фигур), после которой на доске остается ровно одна фигура. Опишем структуру данной функции. Она крайне проста. В случае, если переданная конфигурация содержит всего одну фигуру, то ответ на вопрос является положительным (последовательность ходов пуста). Иначе же требуется выполнить перебор. А именно, мы перебираем фигуру, которой предстоит сейчас сделать ход-взятие, и ее конечное положение. После чего для образовавшейся новой конфигурации рекурсивно проверяем, можно ли из нее достичь цели. В случае, если рекурсивный вызов вернет положительный ответ, то перебор прекращается, и происходит выход из функции проверки (также с положительным ответом). Если при переборе положительного исхода найдено не было, то функция возвращает отрицательный ответ.

Обсудим теперь некоторые способы оптимизации описанного выше метода. Во-первых, для хранения конфигурации удобно использовать два массива. Первый из них будет одномерным и хранить в себе все фигуры (в произвольном порядке) вместе с их координатами, которые встречаются на доске. Второй массив будет двумерным и задавать отображение из клеток доски в индексы фигур в первом массиве. При таком способе организации можно эффективно осуществить перебор всех фигур, а также выполнить обратную задачу: по заданной координатной паре выяснить, какая фигура ей соответствует.

При переборе текущего хода удобно, фиксировав предварительно фигуру, которой этот ход будет совершен, просмотреть все клетки доски, которые оказываются под боем данной фигуры. Альтернативный подход -- перебор пары фигур -- более трудоемок, поскольку в случае. когда бьющая фигура -- это ферзь, слон или ладья, то требуется дополнительная проверка того, что линия боя свободна от посторонних фигур.

Для того, чтобы исключить из массива удаляемую фигуру можно воспользоваться приемом, при котором мы обмениваем удаляемую фигуру с последней в массиве, а также уменьшаем счетчик числа фигур. Данное преобразование допустимо, поскольку порядок фигур в массиве для нас роли не играет.

Далее, не обязательно передавать в процедуру проверки всю конфигурацию целиком. Вместо этого можно ограничиться единственной конфигурацией, которую хранить в глобальных переменных программы. Все попытки сделать ход, которые осуществляет процедура проверки, нужно теперь применять именно к этой глобальной конфигурации. При этом, однако нужно позаботиться о том, чтобы процедура имела возможность "откатить" изменения. Для этого ей достаточно запоминать (в локальных переменных) позицию, из которой выполняется ход, и позицию, в которую он выполняется.

В заключение отметим, что указанные оптимизации не является, строго говоря, обязательными, поскольку ограничение по времени работы было выбрано со значительным запасом. Однако, их применение не приводит к существенному усложнению когда, а иногда даже наоборот -- позволяет повысить его читабельность.

# Задача G. Красно-синий граф

*Автор задачи и разбора - Б.Василевский*

Данный набор точек и стрелок можно рассматривать как ориентированный граф. Каждое ребро ведет из вершины с меньшим номеров в вершину с большим, причем оно окрашено либо в красный, либо в синий цвет. Заметим, что если убрать ориентацию на ребрах, то получился полный граф.

Поменяем ориентацию всех красных ребер в нашем графе. Таким образом, они будут вести из вершины с большим номером в вершину с меньшим. Главная идея решения такова: данная раскраска графа однотонна, если и только если полученный граф не будет содержать циклов. Доказательство этого утверждения --- самая сложная часть задачи, это мы сделаем ниже.

Таким образом, для решения задачи достаточно проверить описанный граф на существование циклов. Это просто сделать за $O(N^2)$.

Если исходный граф не является однотонным, то, очевидно, в новом графе найдется цикл. Так что осталось доказать только прямое утверждение.

Итак, главный факт будем доказывать по индукции. При N = 1 он очевиден.

**Замечание.** Рассмотрим подробнее, что значит отсутствие циклов в графе. Несложно показать, что в нем всегда найдется вершина $u_1$, из которой не выходит ни одного ребра. Если удалить эту вершину, то он останется ациклическим, и рассуждение можно повторить снова (полученная на следующем шаге вершина $u_2$ будет иметь в исходном графе только одно исходящее ребро --- в $u_1$). Таким образом, у любого ациклического графа можно так перенумеровать вершины, что ребра будут вести от меньших к большим.

Пусть теперь при N = K утверждение верно, то есть любой однотонный граф ≤ K вершин после изменения ориентации красных ребер на противоположную получится ациклический граф. Докажем его для N = K + 1.

В каком случае к ациклическому графу 1 -> 2 -> 3 -> ... -> K (считаем, что вершины уже перенумерованы описанным образом) можно добавить еще одну вершину (K + 1), чтобы не появилось циклов?
Пусть сначала *в нее* ведут ребра только из вершин 1, 2, ..., J (для какого-то J, может быть и 0). *Из нее* в таком случае ребра ведут во все вершины J + 1, J + 2, …, K. Тогда, очевидно, новую вершину можно вставить на место J + 1 в наш граф с сохранением ацикличности.
Если это не выполняется, то докажем неоднотонность исходного графа. Итак, не найдется такого J, что в (K + 1) входят ребра *только* из первых J вершин. Тогда существуют такие две вершины A и B, такие что A < B, в A входит ребро из (K + 1), из B выходит ребро в (K + 1) (главную роль здесь играет факт о том, что между любыми двумя вершинами есть ребро). Тогда получился цикл, состоящий из трех вершин: (K + 1) -> A -> B. Непосредственно можно проверить, что граф в таком случае получается не однотонный.

Итак, индукции мы получили, что если граф однотонный, то полученный из него не будет содержать циклов.

# Задача H. Лапта

*Автор задачи и разбора - А.Гусаков*

Предположим, что мы для заданного времени time умеем находить точку (если она существует), в которую нужно бросить мяч, чтобы до его подъема соперником у нас было в запасе время, не меньшее time. Этим будет заниматься функция TryTime(time: Real; var x_ans, y_ans:Real):boolean. Функция принимает значение False, если такой точки не существует и True, если существует. В случае существования точки, ее абсцисса и ордината будут помещены в x_ans и y_ans соответственно.

Если мы уже реализовали эту финкцию, то наибольшее время мы сможем найти с помощью бинарного поиска. Будем поддерживать такое свойство: искомое максимальное время лежит на отрезке [l, r].

```
l:=0;
r:=MAX_ANS;
  while (r-l>2*EPS) do begin // EPS=точность, с которой мы ищем время
    m:=(l+r)/2; // m=середине отрезка
    if TryTime(m,x_ans,y_ans) then //Если икомое время не меньше m
      l:=m // то отрезок:=его правая половина
    else    // иначе
      r:=m;// отрезок:=его левая половина
  end;
```

Искомым временем будет являться (l+r)/2, а координатами x_ans и y_ans.

Поскольку длина отрезка после каждой итерации цикла уменьшается в 2 раза, то количество вызовов функции TryTime будет достаточно маленьким.

Осталось самое главное – реализовать TryTime(time, x_ans, y_ans). По сути, требуется найти точку, с неотрицательной ординатой, лежащую не дальше, чем D от начала координат, и не принадлежащую ни одному кругу (x[i], y[i], v[i]*time) или определить, что такой точки не существует. Предположим, что такая точка P существует. Тогда будем непрерывно двигать ее вверх, пока во что-нибудь не “упремся”. Если после того как мы во что-то “уперлись” возможно двигаться, повышая высоту, то сделаем это. В итоге, наша точка либо станет самой верхней, то есть (0, D), либо совпадет с точкой пересечения двух окружностей (иначе можно будет двигаться еще выше). Таким образом, чтобы проверить, существует ли наша точка, будем искать ее во множестве точек, состоящем из точек пересечения окружностей (0, 0, D), (x[1], y[1], v[1]*time), … , (x[n], y[n], v[n]*time) и точки (0, D). Проверить, подходит ли нам точка, совсем несложно: надо просто понять, можно ли в нее кидать мяч и что ни один соперник не добежит до нее быстрее, чем за время time.

Всего точек пересечения может получиться не более 2*N^2, каждую точку проверять мы можем за N операций, поэтому на функцию TryPoint уйдет порядка N^3 оперций, что при данных ограничениях, не так много.

# Задача I. Машинки

*Автор задачи и разбора - А.Лахно*

Среднее время ожидания машинки есть суммарное время ожидания всех машин, деленное на их количество. Поскольку количество машинок $N + M$ от плана действий постового не зависит, то можно минимизировать суммарное время ожидания.

Далее, для решения задачи, воспользуемся методом динамического программирования.

Будем насчитывать таблицу $B[i][j][h]$ ($0 \le i \le N$, $0 \le j \le M$, $1 \le h \le 2$) — минимальная добавка к суммарному времени ожидания, которую можно получить после проезда $i$ машинок с первой дороги, $j$ машинок со второй дороги, при условии, что после этого будет активна дорога с номером $h$. $B[i][j][h]$ включает в себя суммарное время ожидания $i + j$ проехавших машинок плюс время, прошедшее от начала движения до проезда этих машинок, помноженное на $(N + M - (i + j))$ — количество оставшихся машинок.

Пересчет таблицы можно производить по следующим формулам:

$$B[i][j][1] = min\{B[i-1][j][1] + (N + M - (i + j - 1))\times t1[i],\ B[i-1][j][2] + (N + M - (i + j - 1))\times(\mathrm{T} + t1[i])\},$$

$$B[i][j][2] = min\{B[i][j-1][2] + (N + M - (i + j - 1))\times t2[j],\ B[i][j-1][1] + (N + M - (i + j - 1))\times(\mathrm{T} + t2[j])\},$$

где $t1[i]$ и $t2[j]$ — времена проезда перекрестка $i$-й машинкой с первой дороги и $j$-й машинкой со второй дороги соответственно, Т — время, необходимое для “переключения” потока машин с одной дороги на другую.

Величина $B[i-1][j][2] + (N + M - (i + j - 1))\times(\mathrm{T} + t1[i])$ к примеру, означает, что уже проехали $i - 1$ машинка с первой дороги, $j$ машинок со второй, и “активна” вторая дорога. Мы “переключаем” поток на первую дорогу и пропускаем одну машинку. Остальные величины в формулах пересчета трактуются аналогичным образом.

Отдельно насчитываются граничные элементы таблицы:

$B[0][0][1] = 0$,

$B[i][0][1] = sst1[i] + (N + M - i)\times st1[i]$,

$B[0][j][2] = j\times T + sst2[j] + (N + M - i)\times(T + st2[j])$,

где $st1[i] = \sum_{k=1}^{i} t1[k]$, $st2[j] = \sum_{k=1}^{j} t2[k]$, $sst1[i] = \sum_{k=1}^{i} st1[k]$, $sst2[j] = \sum_{k=1}^{j} st2[k]$ (эти величины следует насчитать заранее).

$st1[i]$ означает время, необходимое для проезда первых $i$ машинок с первой дороги. $sst1[i]$ означает суммарное время ожидания первых $i$ машинок с первой дороги при их последовательном проезде, начиная с первого момента времени. Аналогичным образом трактуются $st2[j]$и $sst2[j]$.

Не нарушая корректности дальнейшего заполнения таблицы, можно положить $B[0][0][2] = \infty$, $B[i][0][2] = \infty$, $B[0][j][1] = \infty$.

После заполнения таблицы ответ задачи получается как $min\{B[N][M][1], B[N][M][2]\}/(N + M)$.

Для восстановления плана действий постового, можно в процессе заполнения таблицы запоминать, на каком из двух возможных вариантов в формулах (*), достигается минимум.

Время работы предложенного решения составляет $O(NM)$.

Задача на 1 балл при ограничениях N, M ≤ 10 подразумевает переборное решение. Всевозможные “разумные” планы действий постового представляются последовательностями из 1 и 2 длины $N + M$, в которых ровно $N$ единиц. Число последовательности означает номер дороги, машинку с которой мы хотим в данный момент пропустить.

Таким образом, для решения задачи надо перебрать все такие последовательности и по каждой из них явно насчитать суммарное время ожидания. Время работы этого решения составляет $O(2^{N+M}\times(N + M))$.

## Задача J. Контейнеры

*Автор задачи и разбора - Д.Кириенко*

Для начала заметим, что мы можем работать только с одним ящиком в каждой стопке – самым верхним. Мы можем снимать самый верхний ящик или класть наверх стопки другой ящик. Такая структура данных, в которой работать можно только с одним верхним элементом называются **стеком**. Итак, задачу можно переформулировать следующим образом:
Дано $n$ стеков, заполненных числами от 1 до $n$. Разрешается перекладывать элементы из одного стека в другой. Необходимо разложить все числа по стекам правильным образом (все числа равные $k$ необходимо собрать в стеке с номером $k$) или определить, что задача решения не имеет.

Если $n = 1$, то ничего перекладывать не надо, задача уже решена.

Если $n = 2$, то задача имеет решение только в следующих двух возможных случаях (стеки изображены растущими слева направо, то есть самый верхний элемент в стеке является самым правым):

Стек 1: 1 1 ... 1 2 2 ... 2
Стек 2: 2 2 ... 2

или

Стек 1: 1 1 ... 1
Стек 2: 2 2 ... 2 1 1 ... 1

В первом случае необходимо переложить все ящики вида 2 из стека 1 в стек 2, во втором случае необходимо переложить все ящики вида 1 из стека 2 в стек 1. При этом количество элементов в каждой из групп, изображённых на рисунке (то есть каждой группы, содержащей многоточие) может быть произвольным, в том числе может равняться нулю.

Во всех остальных случаях при $n = 2$ решения нет.

Если же $n > 2$ то решение есть всегда. Для этого достаточно привести какой-нибудь конструктивный алгоритм решающий поставленную задачу.

Рассмотрим один из возможных алгоритмов на следующем примере для $n = 5$:

Стек 1: 1 5 2
Стек 2: 2 1 4
Стек 3: 3 5 4
Стек 4: 5 2 1
Стек 5: 3 5 2

Сначала соберём все числа в стек 1: переложим все числа из стека 2 в стек 1, затем из стека 3 в стек 1 и т. д. Получим такую картинку:

Стек 1: 1 5 2 4 1 2 4 5 3 1 2 5 2 5 3
Стек 2:
Стек 3:
Стек 4:
Стек 5:

Теперь будем извлекать элементы по одному из стека 1 и класть каждый элемент в стек, соответствующий ему номеру. При этом если из стека был извлечён элемент номер 1 мы будем класть его в стек номер 2. После окончания этой процедуры стек номер 1 будет пустым, в стеке номер 2 будут находиться элементы, равные 1 или 2, а в стеках номер 3..$n$ будут находится элементы, равные номеру стека:

Стек 1:
Стек 2: 2 2 1 2 1 2 1
Стек 3: 3 3
Стек 4: 4 4
Стек 5: 5 5 5 5

Теперь будем извлекать элементы из стека 2, элементы, равные 1 будем класть в стек 1, а элементы, равные 2 – в стек 3:

Стек 1: 1 1 1
Стек 2:
Стек 3: 3 3 2 2 2 2
Стек 4: 4 4
Стек 5: 5 5 5 5

Для полного решения задачи осталось переложить все верхние элементы, равные 2 из стека 3 в стек 2:

Стек 1: 1 1 1
Стек 2: 2 2 2 2
Стек 3: 3 3
Стек 4: 4 4
Стек 5: 5 5 5 5

Если суммарное количество элементов во всех стеках равно $N$, то наше решение выполняет не более $4N$ операций. Поскольку по условиям задачи $N \le 500^2$, то общее количество перекладываний в решении не превосходит $10^6$, то есть укладывается во временное ограничение.

Теперь оценим объем памяти, который необходим для решения задачи. В варианте задачи «на 1 балл» $n \le 10$ и $N \le 100$, то есть для представления $n$ стеков, в каждом из которых будет не более $N$ элементов можно использовать двумерный массив размером $10 \times 100$ элементов (потому что в каждом из стеков может оказаться до $N$ элементов, если все ящики на складе оказались одного вида). В варианте задачи «на 2 балла» такое решение невозможно, поскольку $n$ достигать значения 500, а $N$ – $500^2$, следовательно, для представления стеков в виде статического двумерного массива может понадобиться $500^3$ = 125 000 000 ячеек памяти, что невозможно ввиду ограничения в 64 МБ на программу. Поэтому для решения необходимо сделать хитрую реализацию $n$ стеков, которой будет необходим объем памяти, пропорциональный $N$, а не $n \times N$. Например, можно использовать для каждого стека динамический массив, увеличивая его размер только при необходимости добавить в стек новые элементы или хранить все элементы в одном массиве используя ссылочную реализацию (все элементы всех стеков хранятся в одном массиве, при этом элемент является структурой данных из двух полей: значение элемента и номер элемента, который в этом стеке располагается непосредственно под ним. Для каждого же стека мы храним индекс элемента массива, в котором записан самый верхний элемент в нашем стеке. По сути, $n$ стеков хранятся в одном массиве в виде $n$ односвязных списков). Также можно использовать шаблон stack из стандартной библиотеки STL языка C++, поскольку размер его также меняется динамически.

Ещё отметим, что тесты на 1 балл не содержали случаев $n = 2$.