Андрій Андрійович Гриненко,
студент Київського національного університету імені Тараса Шевченка

Данило Петрович Мисак,
студент Київського національного університету імені Тараса Шевченка

Олександр Владиславович Рибак,
аспірант Інституту математики НАН України

Олександр Борисович Рудик,
доцент Київського університету імені Бориса Грінченка

Ярослав Олегович Твердохліб,
студент Київського національного університету імені Тараса Шевченка

Олімпіада з інформатики у місті Києві у 2010-2011 навчальному році

Передмова

Стаття містить умови завдань III (міського) етапу олімпіади з основ інформатики й обчислювальної техніки у місті Києві у 2011 році та авторські розв'язання цих завдань. Публікацію адресовано учням класів з поглибленим вивченням математики, учасникам олімпіад з інформатики, студентам математичних спеціальностей, учителям і викладачам вищих навчальних закладів.

Проведенню III етапу у 2011 році у місті Києві передувало проведення II (районного) етапу. *Орієнтовні* завдання для II етапу Всеукраїнської учнівської олімпіади з інформатики у місті Києві у 2010 році готував КУ ім. Б.Грінченка:

1) проаналізувати орієнтований граф як схему перегонів: знайти старт, фініш, пункти, які можна уникнути абр які розбивають план;
2) дешифрувати запис дії додавання;
3) занумерувати клітинки таблиці рухом по спіралі та відновити номер рядка і стовпчиках за номером клітинки.

Перші дві задачі складають зміст завдання № 4 відбірково-тренувальних зборів команди міста Києва.

Задачі 1 і 4 III етапу було підібрано таким чином, щоб їх половинне

розв’язання було посильне навіть початківцям. Задачі 3 (дерево Фенвіка) і 6 (пошук максимального паросполучення) було підібрано з перспективою включення до завдань відбірково-тренувальних зборів.

Найкращі результати такі.

Задачі						I тур	II тур	III етап
1	2	3	4	5	6			
100	80	100	100	100	95	280	215	422

Учасники III етапу олімпіади після проведення кожного туру могли отримати електронні копії умов завдань, ідеї розв'язання та тестові файли. Таким чином, вони мали можливість ґрунтовно підготуватися до апеляції результатів перевірки. Під час апеляції умови завдань, ідеї розв’язання й тести визнано коректними.

1. Умови завдань

1. Код Грея (автор — Олександр Рудик)

Максимальна оцінка: 100 балів

Обмеження на час: 1 сек.

Обмеження на пам’ять: 32 МБ

Вхідний файл: code.in

Вихідний файл: code.out

Програма: code.*

Кодом Грея називають непозиційну систему запису цілих натуральних чисел за допомогою двох символів 0 та 1 таким чином. Нуль кодують послідовністю нулів. При зростанні цілого числа:

- наймолодший 1-й розряд у послідовності символів змінюють у такій послідовності: 0, 1, після чого у наступний 2-й розряд записують 1, а наймолодший розряд змінюють уже у протилежному порядку;
- два наймолодші розряди змінюють у такому порядку: 00, 01, 11, 10, після чого у 3-й розряд записують 1, а два наймолодші розряди змінюють уже у протилежному порядку: 10, 11, 01, 00 ... ;
- *k* наймолодших розрядів змінюють у порядку, визначеному попередніми кро-

ками, після чого у наступний (k + 1)-й розряд записують 1, а молодші розряди змінюють уже у протилежному порядку.

Коди Грея довжини 4 чисел від 0 до 15 включно такі (записано у порядку зростання числа): 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000.

Коди Грея двох послідовних натуральних чисел відрізняються лише в одному розряді. Це використовують для збільшення надійності роботи системи оптичних фотодіодів при встановленні кута повороту дисків — носіїв інформації.

Завдання

Визначте код Грея натурального числа.

Вхідні дані

Вхідний файл містить лише десятковий запис натурального числа n при $n < 10^{18}$.

Вихідні дані

Вихідний файл має містити код Грея числа n мінімальної довжини. Інакше кажучи, цей код має починатися з одиниці й містити j символів, де $2^{j-1} \le n < 2^j$.

Приклади

code.in	code.out
2	11
7	100
13	1011

2. Тетрис (автор — Андрій Гриненко)

Максимальна оцінка: 100 балів

Обмеження на час: 5 сек.

Обмеження на пам’ять: 32 МБ

Вхідний файл: tetris.in

Вихідний файл: tetris.out

Програма: tetris.*

Розглянемо гру тетрис з такими правилами. На кожному кроці одна із зображених на рис 1. фігура падає в яму прямокутної форми ширини 10, якщо дивитися збоку. Довжини сторін усіх квадратів, з яких складаються фігури 1–7, дорівнюють 1.

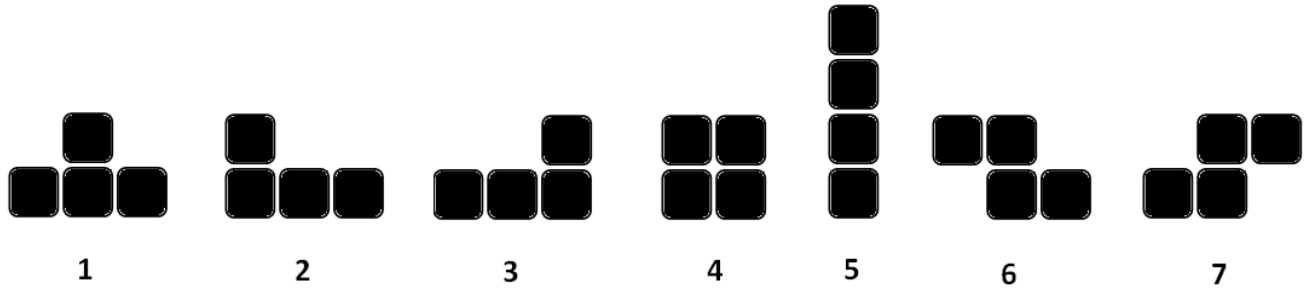

Рис. 1. Фігури для гри тетрис.

Перед початком падіння і лише перед ним фігурку можна повернути на кут 0°, 90°, 180° або 270° за рухом годинникової стрілки.

Рис 2. Зліва направо подано результат обертання фігури 1
на 0°, 90°, 180° або 270° за рухом годинникової стрілки.

Після обертання (у тому числі на 0°) кожну фігуру можна рухати ліворуч, праворуч або вниз на одну клітинку. Таке переміщення дозволено лише, якщо при цьому ця фігура не перетне жодну з фігур, що вже лежать у ямі, і не перетне бокові або нижню стінки ями. Фігура може зупинитись лише тоді, коли хоча б одна з її клітинок своєю нижньою стороною торкається дна ями або фігури, яка вже лежить у ямі.

Завдання

Ваша програма має вказати, як потрібно остаточно розташувати фігури, що падають у яму, щоб зменшити різницю найбільшої висоти, на якій розташовані клітинки фігур у ямі, і кількості повністю заповнених рядків у ямі.

Вхідні дані

Перший рядок вхідного файлу містить одне натуральне число n — кількість фігур, що падатимуть у яму.

Другий рядок містить n чисел: $f_1, f_2, \ldots, f_n$. Тут f_j — тип фігури (див. рис. 1), що впаде у яму на кроці j.

На відміну від решти задач і вперше на ІІІ етапі у м. Києві, *з вмістом усіх вхідних файлів до даної задачі можна ознайомитися під час виконання завдання:*

1. Скопіюйте файл D:\Olimp\Doc\tetris.exe у теку D:\Olimp\Work.
2. Запустіть на виконання tetris.exe у теці D:\Olimp\Work.
3. Введіть пароль «tet2ris» і завершіть розгортання rar-архіву.

Вихідні дані

Вихідний файл має містити n трійок невід'ємних цілих чисел, α_j, x_j і y_j, де:

α_j — градусна міра кута, на який потрібно повернути j-ту у порядку падіння фігуру (0°, 90°, 180° або 270° *за рухом годинникової стрілки*);

x_j — відстань до фігури від лівого краю ями після усіх переміщень;

y_j — відстань до фігури від дна ями після усіх переміщень.

Нагадаємо, що відстанню між двома замкненими обмеженими фігурами називають найменшу відстань між двома точками, що належать до цих фігур.

Потрібно вказати лише кінцеве розташування фігур після усіх переміщень згідно з правилами гри без детального опису того, як отримати таке розташування.

Оцінювання

Учасник отримає відмінну від нуля кількість балів за тест лише при безаварійному вчасному завершенні роботи програми з вихідними даними, що подають кінцеву комбінацію фігур, якої можливо досягнути, дотримуючись правил гри. Ця кількість дорівнює $[N \cdot (0{,}2 + 0{,}8 \cdot (s_{\min} + 1)/(s + 1))]$, де:

$[x]$ — ціла частина x;

N — максимальна кількість балів за даний тест;

$s_{\min}$ — мінімальне досягнення за тест серед усіх учасників;

s — досягнення учасника за тест — $s = h_{\max} - r$, де:

$h_{\max}$ — максимальна висота, на якій розташовано клітинку фігури в ямі;

r — кількість повністю заповнених рядків у ямі.

Приклад

tetris.in	tetris.out

5 7 5 4 2 3	0 0 0 90 2 0 0 7 0 90 6 0 270 8 0

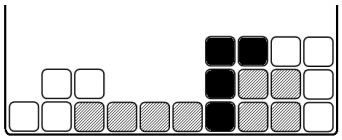

Рис. 3. Остаточне розташування фігур для поданого прикладу вхідних та вихідних даних. За даний тест досягнення учасника становитиме 2 = 3 – 1, бо $h_{\max} = 3$, $r = 1$.

3. Військові навчання (автор — Ярослав Твердохліб)

Максимальна оцінка: 100 балів

Обмеження за часом: 1 сек.

Обмеження за пам'яттю: 64 МБ

Вхідний файл: відсутній

Вихідний файл: відсутній

Програма: military.*

Маленький Петрик в дитинстві дуже любив грати у солдатів. Тепер він виріс і став генералом. Нещодавно йому доручили керувати військовими навчаннями. Під час навчань він із допомогою рації дає солдатам команди, які ті виконують і одразу ж доповідають йому про результат.

Навчання проходять на полігоні, який можна вважати квадратом площини із запровадженою декартовою системою координат. Генерал Петро вирішив давати команди лише трьох типів:

1) висадити солдата у точку з координатами (x, y) та надати йому номер s;
2) відкликати солдата з номером s, якщо такий є на полігоні;

3) визначити кількість солдатів, які перебувають на відстані від точки (x, y), що не перевищує d. При цьому відстань між точками (x_1, y_1) та (x_2, y_2) вважають рівною $|x_1 - x_2| + |y_1 - y_2|$.

Завдання

Напишіть програму military.*, яка б реалізувала таке:

1. Процедуру addSoldier, яку мовами Pascal або C++ потрібно оголосити так:

```
procedure addSoldier(x,y,s:longint);
void addSoldier(int x,y,s);
```

При кожному виклику цієї процедури ваша програма має додавати солдата у точку з координатами (x; y) та надавати йому номер s. Гарантується, що у цей час солдата з таким номером не буде на полігоні.

2. Функцію removeSoldier, яку мовами Pascal або C++ потрібно оголосити так:

```
function removeSoldier(s:longint):longint;
int removeSoldier(int s);
```

При кожному виклику цієї функції ваша програма має видалити солдата з номером s з полігона та повернути 1, якщо солдат з таким номером був на полігоні до виклику функції, або повернути 0 без виконання жодних дій, якщо такого солдата не було.

3. Функцію query, яку мовами Pascal або C++ потрібно оголосити так:

```
function query(x,y,d:longint):longint;
int query(int x,y,d);
```

При кожному виклику цієї функції ваша програма має порахувати кількість солдат, що перебувають на відстані від точки (x, y), що не перевищує d, і повернути цю кількість.

Нижче подано приклади реалізації програм мовами C++ і Pascal.

```
/* GCC */
#include <algorithm>
#include <iostream>
using namespace std;

bool was[50001];
void addSoldier(int x,int y,int s)
{
     was[s]=true;
}
int removeSoldier(int s)
{
     int ret=0;
```

```
     if (was[s]) ret=1;
     was[s]=0;
     return ret;
}
int query(int x,int y,int d)
{
     return 1;
}
```

```
{ Free Pascal }
unit military;

interface

     procedure addSoldier(x,y,s:longint);
     function removeSoldier(s:longint):longint;
     function query(x,y,d:longint):longint;

implementation
     var
     was:array[1..50000] of boolean;
     procedure addSoldier(x,y,s:longint);
     begin
          was[s]:=true;
     end;

     function removeSoldier(s:longint):longint;
     var ret:longint;
     begin
          if (was[s]=true) then ret:=1 else
          ret:=0;
          was[s]:=false;
          removeSoldier:=ret;
     end;

     function query(x,y,d:longint):longint;
     begin
          query:=1;
     end;
end.
```

У поданих кодах масив `was` створено для того, щоб нагадати учасникам, яким чином і в якому місці коду потрібно запроваджувати змінні, які потім можна використовувати. Створення *саме такого* масиву не є обов'язковим, тоді як функції і процедуру потрібно назвати саме так, як це вказано в умові.

Обмеження на параметри. Загальна кількість викликів усіх процедур не перевищує 50 000. В усіх викликах процедур аргументи x, y та d не перевищують за модулем 300, $0 \le d$, аргумент s не перевищуватиме 50 000, $0 < s$.

Інтерактивність. Результат компіляції вашого коду буде підключено *як модуль* до програми перевірки, що буде здійснювати взаємодію з системою

тестування й викликати згадану процедуру `addSoldier` та функції `removeSoldier` та `query`. Будь-які дані можуть бути введені або виведені лише через програму grader.*. Цю програму Вам здавати не потрібно. Її *для перевірки* надасть журі. *Код учасника не має містити звертання до будь-яких файлів*.

Оцінювання. Якщо програма відповість неправильно хоча б на один запит, за відповідний тест присуджують 0 балів.

Компіляція. *Під час перевірки буде використано лише такі компілятори*: Free Pascal 2.2.2, Turbo Delphi Explorer 2006, Dev C++ 4.9.9.2 разом з Mingw/GCC 3.4.2 і Visual C++ 2008 Express Edition:

- для того, щоб скомпілювати розв'язання за допомогою Visual Studio, GNU C++ або Turbo Delphi, потрібно створити проект, у який додати файли grader.*, military.* (у випадку C++ треба ще military.h), виділити grader.* і провести компіляцію;
- для того, щоб скомпілювати розв'язання за допомогою Free Pascal, потрібно скомпілювати grader.pas, причому military.pas має лежати у тій самій теці.

При тестуванні компілятори Turbo C і Turbo Pascal не буде використано, бо вони не забезпечують доступ до відповідного обсягу пам'яті. Їх можна використовувати під час виконання завдання. Потім потрібно змінити замовлення компілятора на **`{ Free Pascal }`** або **`/* GCC */`** і переконатися, що у відповідних середовищах програмування програму можна скомпілювати й запустити на виконання.

Приклади реалізації можна отримати таким чином:

1. Скопіюйте файл D:\Olimp\Doc\military.exe у теку D:\Olimp\Work.
2. Запустіть на виконання military.exe у теці D:\Olimp\Work.
3. Введіть пароль «mili2011tary» і завершіть розгортання rar-архіву.

Для перевірки на тесті, відмінному від запропонованого в архіві, потрібно змінити вміст military.in, скомпілювати та запустити grader.*, що міститься у згаданому архіві military.exe.

Результат виконання програми буде записано у файл military.out.

Формат *наданого в архіві* файлу military.in такий. Перший рядок містить число N — кількість запитів. Кожний з наступних N рядків починається з числа 1, 2 або 3 — типу запиту:

- якщо перше число 1, то слідом йдуть три цілих числа *x, y, s* — параметри addSoldier;

- якщо перше число 2, то слідом йде одне ціле число s — параметр removeSoldier;
- якщо перше число 3, то слідом йде три цілих числа x, y та d — параметри query.

Формат наданого в архіві або створеного grader.* файлу military.out такий:

- файл містить N рядків;
- j-ий рядок відповідає запиту j вхідного файлу. Усі рядки, що відповідають запитам першого типу, порожні, а інші містять результати виконання процедур (описано вище).

Наданий варіант програми перевірки не аналізує вхідній файл щодо справдження обмежень умови та відповідності формату. При перевірці обмежень буде дотримано.

4. Парність (автор — Олександр Рудик)

Максимальна оцінка: 100 балів

Обмеження на час: 6 сек.

Обмеження на пам'ять: 32 МБ

Вхідний файл: oddeven.in

Вихідний файл: oddeven.out

Програма: oddeven.*

Нехай n — довільне натуральне число, а послідовність $i_1, i_2, \ldots, i_n$ містить усі натуральні числа від 1 до n включно. Порушенням порядку у такій послідовності називають систему таких двох нерівностей, що справджуються: $j < k$ та $i_j > i_k$. Якщо послідовність зростає, то кількість порушень порядку дорівнює 0. Якщо послідовність спадає, то така кількість дорівнює $n(n-1)/2$. В усіх інших випадках ця кількість розташована між вказаними величинами.

Завдання

Встановіть парність кількості порушень порядку послідовності.

Вхідні дані

У першому рядку вхідного файлу вказано кількість послідовностей m. Кожний з наступних m рядків містить натуральне число n і послідовність різних натуральних чисел від 1 до n включно: $i_1, i_2, \ldots, i_n$ при $2 \le m \le 16$, $2 \le n \le 1\,048\,576$. У 50 % тестів $n \le 4096$.

Вихідні дані

Єдиний рядок вихідного файлу має містити m символів — нулів або одиниць — без пропусків: k-й символ рядка — це залишок від ділення на 2 кількості порушень порядку k-ї послідовності, заданої $(k + 1)$-м рядком вхідного файлу.

Приклад

oddeven.in	oddeven.out
5 3 1 2 3 3 2 3 1 3 1 3 2 4 2 3 4 1 4 3 4 1 2	00110

5. Фарбована сума (автор — Данило Мисак)

Максимальна оцінка: 100 балів

Обмеження на час: 1 сек.

Обмеження на пам'ять: 32 МБ

Вхідний файл: sum.in

Вихідний файл: sum.out

Програма: sum.*

Фарбованою сумою n натуральних чисел («фарбованих доданків») $a_1, a_2, \ldots, a_n$ назвемо натуральне число s, яке задовольняє такі умови:

1. Цифри числа s можна розфарбувати в n різних кольорів так, що коли в ньому залишити цифри лише i-го кольору, ми отримаємо десятковий запис числа a_i, $1 \le i \le n$.
2. s — найбільше з усіх натуральних чисел, що задовольняють умову 1.

Зрозуміло, що для довільного набору фарбованих доданків $a_1, a_2, \ldots, a_n$ існують числа, які задовольняють першу умову. Наприклад, таким буде число, що утворене шляхом приписування всіх доданків один до одного. Чисел, що задовольняють першу умову, скінченна кількість, тому серед них існує найбільше.

Отже, фарбована сума завжди існує та визначена однозначно. Як видно з означення, вона не залежить від порядку доданків.

Завдання

Знайдіть фарбовану суму n заданих натуральних чисел.

Вхідні дані

У першому рядку вхідного файла вказано кількість фарбованих доданків n, $2 \le n \le 10^5$. У другому рядку задано n фарбованих доданків, розділених символами пропуску. Кожен доданок — натуральне число, менше за 10^9. У 50 % тестів до цієї задачі $n \le 1000$, а у 25 % усіх тестів $n \le 3$.

Вихідні дані

Вихідний файл повинен містити єдине натуральне число — фарбовану суму вказаних у вхідному файлі доданків.

Приклади

sum.in	sum.out
3 417 8 53	854317
2 919 700	971900
4 2009 2010 2011 2012	2222012011010090

6. Розклад уроків (автор — Олександр Рибак)

Максимальна оцінка: 100 балів

Обмеження на час: 1 сек.

Обмеження на пам’ять: 32 Мб

Вхідний файл: schedule.in

Вихідний файл: schedule.out

Програма: schedule.*

Уявіть, що Вам потрібно скласти розклад уроків на тиждень. Для кожного класу відомо, які предмети мають викладати у цьому класі та скільки уроків

припадає на кожен предмет. Кожний предмет викладає лише один учитель, тому уроки з одного предмета не можуть бути у двох класах одночасно. Уроки мають суцільну нумерацію: наприклад, якщо у школі щодня буває не більше за 7 уроків, то перший урок вівторка має номер 8, другий урок вівторка — номер 9 і т. д. Уроки з одним і тим самим номером проходять для всіх класів у один і той самий час. У розкладі допускають «вікна» — випадки, коли один або декілька уроків пропускають, а перед цим і після цього уроки проводять.

Завдання

Подайте приклад потрібного розкладу або вкажіть, що розв'язку немає.

Вхідні дані

Перший рядок вхідного файлу містить у вказаному порядку цілі числа N ($1 \le N \le 50$) та T ($1 \le T \le 50$), де N — кількість класів, а T — максимально можлива кількість уроків протягом тижня. Кожний з наступних N рядків задає перелік предметів, уроки з яких потрібно проводити у відповідному класі (по рядку на клас). У кожному такому рядку записано T невід'ємних цілих чисел:

- кожен предмет позначено деяким *натуральним* числом від 1 до 50 (ця нумерація спільна для всіх рядків);
- номер кожного предмета зустрічається у рядку стільки разів, скільки уроків протягом тижня припадає на цей предмет у відповідному класі;
- якщо рядок містить менше ніж T *натуральних* чисел, він доповнений такою кількістю нулів, щоб загальна кількість чисел у ньому дорівнювала T. Інакше кажучи, кількість нулів — це кількість уроків, які клас пропускає.

Позначимо через $Q(i, j)$ кількість чисел, які дорівнюють j й розташовані у рядку, що задає перелік предметів для i-го класу ($1 \le i \le N$). При $1 \le j \le 50$ справджується така нерівність: $Q(1, j) + Q(2, j) + ... + Q(N, j) \le T$. Інакше кажучи, розклад потрібно скласти за умови, що кожному вчителю призначено провести протягом тижня не більше ніж T уроків.

Вихідні дані

Вихідний файл має містити N рядків по T чисел. Для $k = 1, 2, ..., N$ у k-му рядку вихідного файлу має бути заданий розклад для класу, описаного $(k + 1)$-м рядком вхідного файлу. Кожне число у k-му рядку вихідного файлу має

траплятися стільки разів, скільки воно трапилося у $(k + 1)$-му рядку вхідного файлу. При кожному j $(1 \le j \le T)$ на j-х місцях різних рядків не може бути однакових *додатних* чисел — нулів це не стосується.

Якщо шуканих розкладів багато, подайте будь-який з них. Якщо потрібного розкладу немає, виведіть число −1.

Приклади

schedule.in	schedule.out
2 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7	1 2 3 4 5 6 7 2 3 4 5 6 7 1
3 5 1 1 2 3 2 9 3 3 3 0 1 3 0 0 0	2 1 3 1 2 3 3 0 9 3 1 0 0 3 0

2. Ідеї розв'язання

1. Код Грея

Поданий далі алгоритм знаходження найменшого за довжиною коду Грея випливає безпосередньо з означення, поданого в умові задачі.

1. Зчитуємо величину n з вхідного файлу.
2. Знаходимо мінімальну довжину коду, перебираючи *у порядку зростання* натуральні числа j до справдження нерівності $n < 2^j$.
3. Зменшуємо величину j на 1.
4. Рядковій змінній s надаємо величину порожнього рядка.
5. Повторюємо до справдження рівності $n = 0$ такі кроки:
   - поки справджуються обидві нерівності $n < 2^j$ і $0 < j$, долучимо до змінної s праворуч цифру 0 і зменшуємо величину j на 1 (на початку виконання циклу 5 ці дії не буде зроблено, бо перша нерівність буде хибною);

- якщо $0 < n$, то:
    - змінимо величину n на $2^{j+1} - n - 1$ (таким чином враховано зміну порядку перебору молодших розрядів після написання 1 замість 0 у старшому розряді);
    - долучимо до змінної *s* праворуч цифру 1;
    - якщо $n = 0$, то долучимо до змінної *s* праворуч *j* цифр 0;
    - зменшуємо величину *j* на 1.

6. Записуємо у вихідний файл рядкову змінну *s*.

Альтернативою до цього є:

- обчислення двійкового запису `n xor (n div 2)`, де:
    - `xor` — порозрядна (побітна) операція над двійковими записами:
        - повертає 0 у розряді, якщо числа мають однакові двійкові цифри у цьому розряді;
        - повертає 1 у розряді, якщо числа мають різні двійкові цифри у цьому розряді;
    - `div` — операція обчислення частки при діленні цілих чисел;
- попереднє обчислення кодів Грея для всіх натуральних чисел, що менші за певний степінь двійки. Далі потрібно створити програму, у якій результати розрахунків подано масивом сталих. Однак така альтернатива не дає можливості набрати 100 % балів.

2. Тетрис

Головною особливістю задачі є те, що під час виконання учасникам відомі файли, на яких відбувається тестування. Цю інформацію можна використати для ефективнішого розв'язання.

Враховуючи запропоновану систему оцінювання, щоб отримати не менше ніж 20 % балів достатньо написати розв'язання, що буде видавати коректну відповідь. Якщо всі фігури не обертати й розміщувати на відстані 0 від лівої стінки ями, то достатньо встановити для усіх можливих послідовних комбінацій двох фігур, на яку відстань буде наступна фігура далі від дна ями за попередню. Таке розв'язання набиратиме не менше ніж 20 % балів.

Описане розв'язання можна оптимізувати:

- врахувавши можливість обертання фігур;
- розташувавши кожний тип фігур на певній відстані від лівої стінки ями,

з метою покращити досягнення за тест.

Зауважимо, що в тестах 5 і 6 використано лише деякі типи фігур. Тому ці випадки можна розглянути окремо, істотно покращивши досягнення за ці тести.

Для подальшого покращення результатів потрібно визначати позиції, в яких можна розташувати поточну фігуру. Для цього достатньо зберігати список клітинок у ямі, які вже зайнято фігурами, що впали. Для кожної наступної фігури, починаючи з верхніх рядків, визначати список доступних для неї позицій і певним чином вибирати з них оптимальну для її розміщення. Наприклад, позицію, що розташована найближче до дна ями.

3. Військові навчання

Відстань, про яку йдеться в умові задачі, ще називають *мангеттенською*. Вулиці Мангеттену — району Нью-Йорка — переважно перетинаються під прямим кутом, бо прокладені з півночі на південь або зі сходу на захід. Запроваджена в умові задачі відстань між перехрестями — природна з погляду пішохода чи водія, які не можуть прямувати навпростець через хмарочоси.

Множина точок, що розташовані від точки (x_0, y_0) на відстані, що не перевищує d, є квадратом з вершинами: $(x_0 \pm d, y_0)$ та $(x_0, y_0 \pm d)$. Зазвичай, працювати з фігурами такого вигляду незручно, тому повернемо систему координат на 45° і здійснимо гомотетію (розтягування) з центром у початку координат і коефіцієнтом $2^{½}$. Для цього застосуємо таке лінійне перетворення координат $(x, y) \to (x', y')$:

$$x' = x + y = 2^{½} (x \cos 45° + y \sin 45°);$$
$$y' = – x + y = 2^{½} (– x \sin 45° + y \cos 45°).$$

При цьому:

- пара цілих чисел (x, y) відображається у пару цілих чисел (x', y') *однакової парності*;
- квадрат з вершинами: $(x_0 \pm d, y_0)$ або $(x_0, y_0 \pm d)$ відображається у квадрат з вершинами $(x'_0 \pm d, y'_0 \pm d)$ при $x'_0 = x_0 + y_0$, $y'_0 = – x_0 + y_0$.

Отже, для вирішення задачі нам потрібно реалізувати структуру даних, яка б могла підтримувати такі типи запитів: додати точку з цілими координатами,

видалити точку, порахувати кількість точок у заданому квадраті. Існують декілька варіантів такої структури.

Авторське розв'язання. Двовимірне дерево Фенвіка

Деревом Фенвіка (суматором) називають структуру даних, яка дозволяє для лінійного масиву змінювати значення у його комірці та шукати суму на проміжку 1.. X. Обидві операції виконують за час $O(\log M)$, де M — кількість комірок у масиві. Двовимірний суматор — це узагальнення одновимірного суматора. Ця структура може змінювати значення у комірці двовимірного масиву і шукати суму у прямокутнику з протилежними вершинами (1, 1) і (X, Y). Кожну операцію виконують за час $O(\log^2 M)$. Прочитати детальніше про цю структуру даних можна, наприклад, за такою адресою: http://e-maxx.ru/algo/fenwick_tree.

Для того, щоб знайти суму чисел у довільному прямокутнику з протилежними вершинами (X_1,Y_1), (X_2,Y_2) при $X_1 \le X_2$ і $Y_1 \le Y_2$, можна скористатись формулою включення-виключення. Позначимо через sum(X,Y) суму чисел у прямокутнику з протилежними вершинами (1, 1) та (X, Y). Тоді шукана сума дорівнює $\text{sum}(X_2,Y_2) - \text{sum}(X_1 - 1,Y_2) - \text{sum}(X_2,Y_1 - 1) + \text{sum}(X_1 - 1,Y_1 - 1)$.

Отже, сумарна складність нашого алгоритму $O(N \log^2 R)$, де N – кількість запитів, а R — довжина діапазону координат X або Y. У нашому випадку $R \le 600$.

Альтернативне розв'язання

Очевидним розв'язанням з часовою складністю $O(N^2)$ є таке:

- при кожному висаджуванні солдата запам'ятовуємо його координати та номер за допомогою певного масиву;
- при кожному відкликанні солдата вилучаємо його координати й номер з масиву;
- при кожному запиті щодо кількості солдатів, розташованих від даної точки на відстані, що не перевищує d, аналізуємо елементи масиву і для кожного солдата перевіряємо, на якій відстані від даної точки він розташований.

Перші два типи запитів виконують за сталий час, якщо зберігати:

- у двовимірному масиві `A[x][y]` — кількість солдатів з координатами (x,y);
- у масивах `X[i]` та `Y[i]` — координати x та y солдата з номером i або –1000, якщо такого солдата немає.

«Висадити» солдата *s* у точку (*x*, *y*) можна таким чином:

- `inc(A[x][y]); X[s]:=x; Y[s]:=y;` — мовою Pascal;
- `++A[x][y]; X[s]=x; Y[s]=y;` — мовою C++.

«Відкликати» солдата *s* можна таким чином:

- `if X[s]<>-1000 then begin dec(A[X[s]][Y[s]]); X[s]:=-1000; Y[s]:=-1000 end;` — мовою Pascal;
- `if (X[s]!=-1000) { --A[X[s]][Y[s]]; X[s]=Y[s]=-1000; }` — C++.

Обидві ці дії виконуються за сталий час.

Для пришвидшення відповіді на третій запит можна зробити таке:

1. Виберемо сталу *K*.
2. Кожного солдата будемо вважати або «старим», або «новим»:
   - у комірках таблиці *A* будемо зберігати кількість «старих» солдатів у точках з координатами, що є індексами комірки;
   - у комірках допоміжного лінійного масиву довжини *K* будемо зберігати інформацію про «нових» солдатів — тих, кого висаджено на полігон останніми.
3. За таблицею *A* побудуємо допоміжну таблицю *B*, у комірці (*X*,*Y*) якої буде стояти сума чисел у прямокутнику з протилежними комірками (1,1) та (*X*,*Y*) таблиці *A*. Це можна зробити згідно з такою формулою:
$$B(X,Y) = A(X,Y) + B(X-1,Y) + B(X,Y-1) - B(X-1,Y-1).$$
4. При запиті першого типу будемо долучати подію «прийшов новий солдат з такими координатами і таким номером» у проміжний масив.
5. При запиті другого типу будемо долучати подію «видалено солдата з такими координатами і таким номером» у проміжний масив. Якщо солдата з таким номером не було на полігоні, не робитимемо нічого.
6. Якщо розмір допоміжного масиву дорівнює *K*, переносимо інформацію з нього у таблицю *A*, перебудуємо таблицю *B* й очистимо допоміжний масив.
7. При кожному запиті третього типу будемо шукати суму чисел у відповідному квадраті у таблиці *A*, застосовуючи формулу включення-виключення для таблиці *B* та аналізуючи елементи допоміжного масиву (будемо збільшувати

чи зменшувати відповідь на 1, якщо висаджують чи відкликають солдата з відповідного квадрата).

Запити перших двох типів, як і раніше, виконують за сталий час, а запит третього типу виконують за час $O(K)$. Додатково через кожні K запитів перших двох типів перебудовуємо таблицю B за час $O(R^2)$, де R, як і раніше, — довжина діапазону координат X або Y. Отримуємо ефективність за часом $O(NK + R^2N/K)$.

Потрібно лише вдало підібрати сталу K. Можна вважати що $R^2 = 360\ 000$. При $K = 600$ маємо: $NK \le 30\ 000\ 000$, $R^2N/K \le 30\ 000\ 000$, $NK + R^2N/K \le 60\ 000\ 000$.

4. Парність

50% балів можна було набрати, перевіряючи справдження нерівності $i_j > i_k$ при $j = 1, 2, \dots, n - 1$ та $k = j + 1, j + 2, \dots, n$. Ефективність такого алгоритму $O(n^2)$.

Для викладу ідеї оптимального розв'язання скористаємося такими міркуваннями.

Означення 1. *Запровадимо такі поняття.*

1. *Підстановкою називають взаємно однозначне відображення скінченої множини у себе. Не обмежуючи загальності міркувань, надалі будемо розглядати лише множини вигляду* $\{1, 2, \dots, n\}$ *— множини всіх перших n натуральних чисел.*
2. *Транспозицією називають підстановку, яка кожному елементу, за виключенням двох, ставить у відповідність цей самий елемент.*
3. *Порушенням порядку підстановки на множині* $\{1, 2, \dots, n\}$, *що числу j ставить у відповідність число* i_j *при* $j = 1, 2, \dots, n$, *називають систему таких нерівностей:* $j < k$ *та* $i_j > i_k$, *що справджуються.*
4. *Підстановку називають парною, якщо кількість транспозицій, у добуток яких можна розкласти дану підстановку, є парною.*
5. *Циклом довжини k називають підстановку, яка певний елемент* j_1 *відображає у деякий елемент* j_2 , j_2 *—* $в$ j_3 , $\dots$, j_{k-1} *—* $в$ j_k , j_k *—* $в$ j_1 *за умови, що всі елементи* $j_1, j_2, j_3, \dots, j_{k-1}, j_k$ *— різні.*

Зауваження 1. *Справджуються такі висловлювання.*

1. *Транспозиція, що «міняє місцями» сусідні натуральні числа, змінює кількість порушень порядку на 1.*

2. *Довільна транспозиція змінює кількість порушень порядку на непарне число, бо є добутком непарної кількості транспозицій, що «міняє місцями» лише сусідні натуральні числа.*
3. *Парність кількості транспозицій у розкладі підстановки не залежить від конкретного подання добутком транспозицій і збігається з парністю кількості порушень порядку.*
4. *Цикл з непарною довжиною є добутком парної кількості транспозицій.*
5. *Цикл з парною довжиною є добутком непарної кількості транспозицій*.
6. *Підстановка є парною тоді й лише тоді, коли її подання добутком циклів без спільних елементів містить парну кількість циклів парної довжини*.

Авторське розв'язання ґрунтується на останньому твердженні (висловлюванні 6 зауваження 1) і передбачає розбиття підстановки у добуток циклів зі встановленням їхніх довжин. Для цього достатньо лише один раз проглянути послідовність $i_1, i_2, \ldots, i_n$. Ефективність такого алгоритму $O(n)$ з досить малим коефіцієнтом пропорційності.

Усі подальші міркування цього розділу стосуються лише технології створення тестів певного вигляду та доведенні їхньої коректності. Тести створено програмою gen.pas, розташованою у теці вхідних і вихідних файлів задачі в архіві, опублікованому за адресою http://kievoi.narod.ru/rar/2011_1.rar. У кожному тесті кількість послідовностей, їхню довжину і структуру вибирають випадковим чином.

Кожна послідовність кожного тесту, крім двох останніх (послідовностей), складається з блоків-підпослідовностей такої структури:

$$k, k+1, \ldots, k_1-1, k_1, k_0, k_0+1, \ldots, k-1 \text{ при } k_0 < k \le k_1. \qquad (1)$$

Такий блок займає у послідовності місця з k_0 по k_1 включно. Маємо:

- кількість порушень порядку в усій послідовності є сумою кількостей порушень порядку в усіх блоках;
- у блоці-підпослідовності (1) кількість порушень порядку дорівнює добутку довжин його максимальних підпослідовностей, що зростають:

$$(k_1 - k + 1)\cdot(k - k_0).$$

Останні два рядки вхідного файлу задають монотонно спадні послідовності. Довжини цих послідовностей відрізняються на 2. Серед послідовних чотирьох

чисел n, $n-1$, $n-2$, $n-3$ два й лише два парні, а з них одне й лише одне число кратне чотирьом. Тому серед двох чисел $n\cdot(n-1)/2$ і $(n-2)\cdot(n-3)/2$ одне й лише одне парне. Отже, еталон кожного з вихідних файлів має одне з двох закінчень: 01 або 10.

Еталони вихідних файлів створено без опрацювання вхідних даних за допомогою авторського розв'язання задачі.

5. Фарбована сума

Якщо один або кілька фарбованих доданків закінчуються нулями, всі такі нулі можна відкинути, знайти фарбовану суму новоутвореного набору чисел, а потім дописати відкинуті нулі в кінець отриманого числа. Так ми одержимо фарбовану суму початкового набору доданків. Тому досить розглянути спрощену задачу — задачу пошуку фарбованої суми чисел, жодне з яких не закінчується нулем. Далі будемо розглядати саме таку задачу. Для її розв'язання скористаємось таким алгоритмом, що працює з набором невід'ємних цілих чисел, менших за 10^9:

1) Допишемо до кожного фарбованого доданка ззаду кілька нулів (тобто, збільшимо в 10, 100, 1000, … разів) так, щоби доданок мав рівно дев'ять цифр. Ототожнимо кожен доданок із рядком, що складається з 9 символів (цифр).
2) Ініціалізуємо змінну *Sum*, в якій після завершення роботи алгоритму буде записано шукане значення, порожньою послідовністю цифр.
3) Візьмемо найбільший у лексикографічному порядку рядок. Йому відповідатиме максимальне число з набору. Якщо таких рядків кілька, візьмемо будь-який із них. Якщо цей рядок складається лише з нулів, то й усі інші рядки такі самі, і алгоритм завершує свою роботу. Відповідь тоді записано в змінній *Sum*.
4) Перший символ взятого рядка дописуємо в кінець послідовності цифр, що зберігається в змінній *Sum*.
5) Змінюємо рядок, який узяли на кроці 3: відкидаємо його перший символ (після цього новим першим символом рядка може виявитися й нуль), а останнім символом дописуємо 0. Повертаємося до кроку 3.

Доведення коректності алгоритму

Спершу зауважимо, що алгоритм ніколи не повертає порожньої послідовності цифр (оскільки, згідно з умовою, всі вхідні числа додатні) і завершує свою роботу після скінченної кількості кроків: на кожній ітерації сумарна кількість нулів, що стоять наприкінці доданків, збільшується, а самі доданки завжди менші за 10^9. Крім того, ми дописуємо до послідовності *Sum* цифри кожного доданка у правильному порядку і допишемо зрештою кожну цифру кожного доданка — поки хоч одну цифру не було дописано на кроці 4 й відкинуто на кроці 5, у наборі знайдеться ненульове число (рядок, що має відмінний від нуля символ). З іншого боку, жодних зайвих цифр ми не дописуємо. Таким чином, алгоритм утворює число, що задовольняє умову 1 з означення фарбованої суми. Назвемо числа, що задовольняють першу умову, «претендентами». Нам залишилось показати, що фарбована сума,— максимальне з чисел-претендентів,— збігається з отриманим числом.

Спершу зауважимо, що будь-яке число-претендент можна утворити згідно зі схемою, поданою в алгоритмі — єдиним винятком є крок 3, на якому тепер дозволяється брати не лише максимальне, а й будь-яке ненульове число з поточного набору. Тоді, взагалі кажучи, різному вибору чисел на кроці 3 відповідатимуть різні претенденти. Вибір на кроці 3 завжди найбільшого з чисел назвемо «канонічним вибором».

Нехай число, отримане алгоритмом, менше від справжньої фарбованої суми. Тоді існує хоча б одна така послідовність вибору на кроці 3, що число, отримане з допомогою нею, є справжньою фарбованою сумою й перевищує того претендента, якого вдалось отримати, вибираючи кожного разу максимальне з чисел. З усіх таких послідовностей (якщо вона не одна) виберемо ту, яка відрізняється від канонічного вибору на якомога більш пізній ітерації. Якщо й таких послідовностей є кілька, виберемо довільну з них. Вибрану послідовність назвемо оптимальною. Тепер розглянемо першу ітерацію, на якій вибір, що здійснюється на кроці 3 алгоритму, різний для канонічної та оптимальної послідовностей. «Забудемо» про всі попередні ітерації та будемо вважати, що алгоритм почав свою роботу саме на цій ітерації — вважатимемо, що це нульова ітерація. Число,

яке на нульовій ітерації було найбільшим, і яке вибрала б канонічна послідовність, назвемо канонічним, а число, вибране оптимальною послідовністю,— оптимальним. Зауважимо, що, хай на якій ітерації перебуває алгоритм, ми казатимемо саме про ті канонічне та оптимальне числа, які були вибрані на нульовій ітерації. Згідно з вибором оптимальної послідовності, оптимальне число відрізняється від канонічного. Першу (найлівішу) цифру канонічного числа, що більша від цифри на відповідному місці оптимального числа, назвемо «великою». Зауважимо: ми вважаємо, що кожне з чисел має рівно 9 цифр і його запис може починатися з нулів.

Тепер будемо здійснювати вибір відповідно до оптимальної послідовності. Доведемо методом математичної індукції, що:

а) після кожної ітерації, починаючи з першої, кількість цифр, узятих із оптимального числа, більша за кількість цифр, узятих із канонічного числа;

б) велику цифру канонічного числа взято ще не було.

Таке твердження призводить до суперечності. Наприклад, через те, що після останньої ітерації мають бути взятими всі цифри, включно з великою цифрою канонічного числа. Тому все, що нам залишається зробити, — це довести твердження.

Для першої ітерації твердження справджується, адже з оптимального числа взято одну цифру, а з канонічного — ще жодної.

Нехай після якоїсь ітерації l порушилась справедливість пункту а). Тоді, якщо твердження справджувалось на попередній ітерації, це означає, що кількості цифр, узятих із оптимального та канонічного чисел, після ітерації l однакові. Нехай вони дорівнюють по k. Побудуймо тоді нову послідовність вибору. Вона буде збігатися з оптимальною, лише вибір k перших цифр оптимального та канонічного чисел буде розміняно місцями: в новій послідовності, поки ми не дійшли до ітерації $l + 1$, беремо цифру з оптимального числа тоді й лише тоді, коли оптимальна послідовність бере цифру з канонічного числа; а з канонічного беремо цифру тоді й лише тоді, коли оптимальна послідовність бере цифру з оптимального числа. Згідно з припущенням індукції (пункт б), перші $k - 1$ цифр оптимального та канонічного чисел збігаються, а k-та цифра або збігається, або

більша в канонічного числа. До того ж, *k*-ту цифру оптимальна послідовність вибрала спершу в оптимального числа, а потім в канонічного, а побудована щойно послідовність — навпаки. Звідси випливає, що число-претендент, утворене побудованою щойно послідовністю, не менше (а може навіть і більше) від числа, утвореного оптимальною послідовністю. Це суперечить вибору оптимальної послідовності: якщо нерівність строга, то виходить, що оптимальна послідовність утворила не найбільше можливе число; якщо ж утворені числа однакові, нова послідовність вибору збігається з канонічною принаймні на одну ітерацію довше, ніж оптимальна.

Нехай тепер після якоїсь ітерації *l* порушилась справедливість пункту б), тобто на цій ітерації було взято велику цифру канонічного числа. Згідно з пунктом а) індукційного припущення, з оптимального числа після *l*-ї ітерації взято не менше цифр, ніж із канонічного. Отже, на якійсь попередній ітерації *m* було взято цифру оптимального числа, що стоїть на тому ж місці, що й велика цифра в канонічному. Згідно з означенням великої цифри, ця цифра менша від неї. Побудуємо, як і раніше, нову послідовність вибору. Вона буде збігатися з оптимальною до *m*-ї ітерації включно, за винятком того, що вибір оптимального та канонічного чисел буде розміняно місцями. Після ж *m*-ї ітерації в новій послідовності вибір може здійснюватися як завгодно. Таким чином, перша $m-1$ цифра числа-претендента, утвореного новою послідовністю вибору, збігаються з відповідними цифрами числа, утвореного оптимальною послідовністю. Цифра ж на місці *m* — більша, і, хай які цифри стоять на решті місцях, отримане число-претендент більше від того, що утворене оптимальною послідовністю. Це суперечить вибору оптимальної послідовності.

Реалізація алгоритму

Із фарбованими доданками можна оперувати як із рядками або як із числами. Будемо вважати, що ми працюємо з ними як із числами, враховуючи, що під першою цифрою числа слід тоді розуміти цілу частину при діленні цього числа на 10^8.

Якщо відкинути технічні деталі реалізації, єдиним гнучким і неочевидним моментом залишається пошук максимального числа на кожній ітерації алгоритму.

Простіше за все щоразу заново шукати максимум серед усіх елементів масиву *Numbers*[1], …, *Numbers*[*n*] з поточним набором чисел. Такий підхід (якщо знехтувати часом, що витрачається на бітові операції з числами) вимагає $O(dn^2)$ часу, де d — найбільша кількість цифр, яку може мати фарбований доданок (у нашому випадку $d = 9$).

Значно покращити ситуацію можна, зберігаючи набір чисел згідно зі структурою т. зв. бінарної купи. Як і раніше, в комірках *Numbers*[1], … , *Numbers*[*n*] масиву *Numbers* буде записано всі числа набору. Але порядок між ними підтримуватимемо такий, що в комірці *Numbers*[*k*], $k > 1$, завжди буде записано число, не більше за те, яке записане в комірці *Numbers*[*k* div 2] (тут через div позначено ділення без залишку). Тоді, як видно, в комірці *Numbers*[1] завжди зберігатиметься максимальне число набору, і на його пошук витрачати часу вже не потрібно. Але коли це максимальне число змінюється (після того, як відкидаємо його першу цифру та дописуємо ззаду 0), описану структуру бінарної купи в масиві слід відновити.

Відновлення структури відбувається в такий спосіб. Знайдемо більше з двох значень *Numbers*[2] та *Numbers*[3]. Нехай воно записане в комірці k ($k = 2$ або $k = 3$). Якщо це значення не більше за *Numbers*[1], структуру не було порушено. Інакше розміняємо місцями числа в комірках 1 та k. Тепер розглянемо більше з двох значень *Numbers*[2*k*] та *Numbers*[2*k* + 1]. Нехай воно записане в комірці l ($l = 2k$ або $l = 2k + 1$). Якщо воно не перевищує числа, записаного (тепер уже) в k-й комірці, структуру відновлено. Інакше розміняємо вміст комірок *Numbers*[*k*] та *Numbers*[*l*] і продовжимо, розглядаючи елементи *Numbers*[2*l*] та *Numbers*[2*l* + 1] і т. і. Робитимемо це доти, доки структуру купи не буде відновлено. Це рано чи пізно станеться, адже на якомусь кроці розглядувані значення вийдуть за межі масиву, а елементи *Numbers*[*m*] з індексами $m > n$ можемо, наприклад, вважати нульовими.

Якщо використовувати структуру бінарної купи, на кожному кроці для її підтримування витрачається $O(\log n)$ часу, тому загальна тривалість виконання алгоритму складатиме — без урахування часу на бітові операції — $O(dn \log n)$.

Жадібніші підходи

Хоч описаний алгоритм, безумовно, є жадібним, на думку учасника олімпіади міг спасти один із двох значно більш жадібних підходів:

1) Розглянути всі перші цифри початкового набору чисел, вибрати з них найбільші (наприклад, дев'ятки, якщо вони є) й поставити на початок числа-результату; відкинути всі ці цифри, знов розглянути перші цифри утвореного набору чисел, найбільші з них дописати до результату й відкинути і т. д.
2) Розглянути всі перші цифри початкового набору чисел, вибрати одну довільну найбільшу з них і поставити на початок числа-результату; відкинути цю цифру, знов розглянути перші цифри утвореного набору чисел, одну найбільшу з них дописати до результату й відкинути і т. д.

Ці підходи не завжди дають правильний результат. Наприклад, якщо на вході задано два числа 56 та 57, перший із поданих алгоритмів поверне число 5576, а другий може повернути 5657, тоді як справжньою фарбованою сумою вказаних чисел є 5756.

На ґрунті другого підходу, однак, може базуватися алгоритм рекурсивного перебору всіх можливих варіантів вибору числа з максимальною першою цифрою, що для $n \leq 3$ дає правильну відповідь за прийнятний час.

6. Розклад уроків

В умові зазначено, що з кожного предмету треба провести не більше ніж T уроків. Доведемо, що за цієї умови завжди можна знайти потрібний розклад.

Назвемо клас *насиченим*, якщо у цьому класі треба провести саме T уроків. Предмет назвемо *насиченим*, якщо з цього предмету в усіх класах разом потрібно провести саме T уроків. Інакше кажучи, насичений клас не може пропустити жодного уроку, а насичений предмет на кожному уроці має вивчатися в якомусь класі.

Спочатку знайдемо розклад для першого уроку, задовольнивши такі вимоги:

- кожний насичений предмет вивчається у деякому класі;
- кожний насичений клас має якийсь урок.

Після цього:

- кожний урок, який буде у першому стовпчику розкладу, вилучимо з відповідного рядка умови (якщо число, що позначає внесений до розкладу урок, зустрічається у відповідному рядку умови більше одного разу, ми зменшимо кількість таких чисел у цьому рядку на 1);
- інші уроки будемо намагатися розмістити в решті $(T - 1)$ стовпчиках таблиці розміру $N \times T$, що задає розклад (див. формат вихідного файлу).

При складанні першого стовпчику задіяні всі насичені уроки, після цього з кожного предмету буде потрібно провести не більше ніж $(T - 1)$ урок. Тому в такий же спосіб ми можемо побудувати розклад для другого і наступних уроків. Перед побудовою кожного стовпчика розкладу будемо поновлювати списки насичених класів і предметів. Насиченими будемо вважати ті класи, яким залишилося стільки уроків, скільки є ще незаповнених стовпчиків у розкладі. Аналогічно поновимо перелік насичених предметів. На кожному етапі будемо використовувати, звісно, оновлені списки насичених класів і предметів. Кількість незаповнених стовпчиків таблиці-розкладу після кожного заповнення буде зменшуватися, тому за T кроків ми побудуємо весь розклад.

Залишилось описати, як скласти розклад для першого уроку. Це завдання зводиться до класичної задачі пошуку паросполучення у дводольному графі. Розглянемо граф, у якому одні вершини відповідають класам, а інші — предметам. З'єднаємо ребрами такі пари вершин U, V, що U відповідає деякому класу, а V відповідає предмету, з якого у цьому класі потрібно провести хоча б один урок. *Паросполученням* назвемо таку сукупність ребер графа, у якій жодні два ребра не мають спільних вершин. Це якраз відповідає таким двом умовам:

- ніякий предмет не може вивчатися у двох класах одночасно;
- у кожному класі в один і той самий час може бути урок лише з одного предмету.

Потрібно розподілити предмети таким чином, щоб це відповідало деякому паросполученню, і включити у нього всі насичені вершини, тобто вершини, які відповідають насиченим класам або предметам. Покажемо, як це зробити.

Сукупність вершин, які відповідають класам, назвемо *нижньою долею*, а сукупність вершин, які відповідають предметам, — *верхньою долею*. Спочатку

побудуємо паросполучення, в яке з нижньої долі входять усі насичені вершини і лише вони. Уявімо, що вже є паросполучення, в яке з нижньої долі входять лише насичені вершини, але не всі. Можна вважати, що таке паросполучення існує і спочатку, але воно не містить жодного ребра. Покажемо, як збільшити кількість ребер у цьому паросполученні на 1. Для цього перетворимо граф на орієнтований. Усі ребра, які входять до паросполучення, орієнтуємо «згори вниз» — від предметів до класів, а решту ребер орієнтуємо «знизу вгору» — від класів до предметів. Виберемо довільну насичену вершину U з нижньої долі, яка ще не входить у паросполучення. Знайдемо таку вершину V з верхньої долі, яка також не входить у паросполучення і до якої можна дійти від U, рухаючись ребрами у відповідності до їх орієнтації (доведення існування такої вершини V подано у наступному абзаці). Вершину V та ланцюг ребер, що веде до неї, можна знайти за допомогою відомої процедури *пошуку в ширину*. Нехай U, V_1, U_1, V_2, U_2, ... , V_k, U_k, V — послідовні вершини, які лежать на згаданому ланцюжку ребер. Також можливий випадок, коли $k = 0$, тобто коли з U можна безпосередньо дістатися до V. Згідно з побудовою, вершини V_1, V_2, ... , V_k розташовані у верхній долі, а U_1, U_2, ... , U_k — у нижній. До паросполучення входять k ребер (V_1, U_1), ... , (V_k, U_k). Замінимо їх на ($k + 1$) ребро (U, V_1), (U_1, V_2), ... , (U_k, V), що не належать до нього. У результаті отримаємо паросполучення, яке містить на одне ребро більше і в яке з нижньої долі залучено лише насичені вершини. У випадку $k = 0$ ми просто долучимо до паросполучення ребро (U, V). Повторюючи вказане перетворення, побудуємо паросполучення, якому належать усі насичені вершини першої долі.

Доведемо від супротивного, що потрібна вершина V на кожному кроці знайдеться. Припустимо, що це не так. Нехай Y_1, Y_2, ... , Y_l — усі вершини *верхньої* долі, яких можна досягнути з U, рухаючись орієнтованими ребрами. Згідно з припущенням, усі ці вершини належать до паросполучення. Нехай (Y_1, X_1), (Y_2, X_2), ... , (Y_l, X_l) — ребра паросполучення, які містять згадані вершини. Кожної вершину X_i ($1 \le i \le l$) також можна досягнути з U, рухаючись орієнтованими ребрами. Для цього потрібно лише вийти з відповідної Y_i. Вершини U, X_1, X_2, ... , X_l є насиченими, тобто з кожної з них, без урахування орієнтації, виходить по T ребер. Усі ці ребра мають у якості інших кінців лише вершини Y_1, Y_2, ... , Y_l,

інакше у верхній долі існувала би ще одна вершина, досяжна з U. Тому до вершин Y_1, Y_2, ... , Y_l разом входять, без урахування орієнтації, не менше ніж $(l + 1)\cdot T$ ребер. Але це неможливо, бо до кожної з цих l вершин може входити щонайбільше T ребер. Отримана суперечність свідчить про хибність припущення про відсутність потрібної вершини V.

Тепер, якщо це не вийшло відразу, перебудуємо паросполучення таким чином, щоб і всі насичені вершини *верхньої* долі увійшли до нього. Змінимо орієнтацію всіх ребер. Розглянемо насичену вершину V *верхньої* долі, яка не належить паросполученню. Нам достатньо знайти:

- або вершину U в нижній долі, яка не входить до паросполучення і є досяжною з V згідно з новою орієнтацією ребер;
- або ненасичену вершину W у верхній долі, яка знову-таки досяжна з V.

Існування однієї з таких вершин доводиться точно так само, як до цього ми доводили існування вершини V, досяжної з U. У першому випадку в нас буде ланцюг V, U_1, V_1, U_2, V_2, ... , U_k, V_k, U, в якому ребра (U_1, V_1), ... , (V_k, U_k) можна замінити на (V, U_1), (V_1, U_2), ... , (V_k, U). У другому випадку отримаємо ланцюг V, U_1, V_1, U_2, V_2, ... , U_k, W, в якому ребра (U_1, V_1), (U_2, V_2), ... , (V_k, W) можна замінити на (V, U_1), (V_1, U_2), ..., (V_{k-1}, U_k). У цьому (другому) випадку ми не збільшимо кількість ребер, але долучимо до паросполучення насичену вершину V замість ненасиченої W. Будемо повторювати описане перетворення, поки не долучимо всі насичені вершини *верхньої* долі до пар.

Після вказаних процедур ми отримаємо розподіл предметів для першого уроку. Повторюючи їх, можна побудувати весь розклад.

Оцінимо кількість операцій, яку вимагає вказаний алгоритм. Пошук у ширину потребує $O(e)$ операцій, де e — кількість ребер у графі. Наш граф містить $O(NT)$ ребер. Для побудови паросполучення ми застосовуємо пошук у ширину $O(N)$ разів, бо маємо не більше ніж N насичених класів, а тому й не більше ніж N насичених предметів. Отже, пошук одного паросполучення потребує $O(N^2T)$ операцій. Для багатьох графів справджується оцінка $O(NT)$. Така оцінка досягається завдяки тому, що на більшості пошуків у ширину проглядаються далеко не всі ребра. Тому для одного паросполучення маємо теоретичну оцінку

$O(N^2T)$ і практичну — $O(NT)$. Для побудови розкладу ми шукаємо T паросполучень. Отже, для розв'язання задачі теоретично потрібно $O(N^2T^2)$ операцій, а на практиці — $O(NT^2)$.

Зробимо ряд зауважень щодо можливого *практичного використання* поданого розв'язання задачі.

1. Описаний алгоритм на кожному етапі намагається навантажити лише насичені класи. Тому багато «вікон» виникає саме на початку розкладу. Для того, щоб розклад виглядав більш природно, його можна формувати з кінця.

2. Обмеження, що кожен предмет веде лише один учитель, не є істотним полегшенням задачі. Воно лише робить умову задачі зручнішою для сприйняття. До даного обмеження можна звести випадок, коли предмет ведуть k вчителів, але їх сумарне навантаження не перевищує kT уроків. Для цього треба лише розподілити, які вчителі будуть вести свій предмет у яких класах і для кожного вчителя занумерувати його предмет окремим числом. Розподіляти треба так, щоб кожному вчителеві дісталося не більше T уроків. Схоже перетворення можна зробити, коли один вчитель веде декілька уроків. Тоді всі його уроки потрібно позначити одним числом.

3. Стовпчики побудованої таблиці-розкладу можна переставляти. Тому розв'язання можна пристосувати до складнішої ситуації, коли якийсь учитель не може бути у школі під час проведення деякого уроку. Нехай s – кількість учителів, а k_1, k_2, ... , k_s – кількості уроків, які вони мають провести. Нехай для i-того учителя ($1 \le i \le s$) є t_i номерів уроків, на яких він не може бути. Тоді якщо $k_1t_1 + k_2t_2 + ... + k_st_s < T$, то завжди можна підібрати гарний розклад. Дійсно, будемо переставляти стовпчики розкладу циклічно. Тоді i-тий учитель може «зіпсувати» не більше k_it_i зсувів, отже потрібний варіант залишиться.

Деякі інші застосування паросполучень описані у статтях:

[1] О. Аксенов, О. Рибак, *Застосування однієї графової теореми при розв'язуванні деяких задач*, У світі математики, 8 (2002), № 3, 55-58, доступна за адресою http://mathsociety.kiev.ua/rybaka01.pdf.

[2] О. Рибак, *Ідеальні розбиття підмножин*, У світі математики, 10 (2004), № 2, 29–35, доступна за адресою http://mathsociety.kiev.ua/rybaka02.doc.

3. Авторські розв'язання

1. Код Грея

```
{ Turbo Pascal }
{$N+}
var a: array[0..61] of extended;
  j,k: shortint;  s: string;
    o: text;      x: extended;     BEGIN
assign(o,'code.in');  reset(o);
readln(o,x);           close(o);
s:='';
           j:=0;  a[j]:=1;
repeat inc(j);    a[j]:=a[j-1]*2
 until x<a[j];
dec(j);
if x=0 then s:='0' else
REPEAT while (x<a[j]) and (0<j) do begin
                               s:=s+'0';
                               dec(j) end;
       if (0<x) then                begin
                      x:=2*a[j]-x-1;
                      s:=s+'1';
       if (x=0) then
                for k:=1 to j do s:=s+'0';
       dec(j)                          end;
UNTIL x=0;
assign(o,'code.out');  rewrite(o);
writeln(o,s);             close(o)   END.
```

2. Тетрис

```
/* GCC */
#include <algorithm>
#include <cstdio>
#include <cstdlib>
#include <cctype>
#include <cmath>
#include <iostream>
#include <queue>
#include <list>
#include <map>
#include <numeric>
#include <set>
#include <sstream>
#include <string>
#include <vector>
using namespace std;
#define INF 1000000000
typedef pair<int, int> PII;
typedef vector<int> VI;
typedef vector<int> VB;
typedef vector<PII> VPII;
typedef vector<VI> VVI;
typedef vector<VB> VVB;
```

```
// Функція, що виконує поворот фігури на
// 90 градусів за годинниковою стрілкою.
VVB rotate (const VVB& a)
{
    if (!a.size())
        return VVB();
    int n = a.size();
    int m = a[0].size();

    VVB res (m, VB(n));

    for (int i=0; i<n; ++i)
        for (int j=0; j<m; ++j)
            res[m-j-1][i] = a[i][j];

    return res;
}

// Масив доступних фігур
vector <VVB> figures;

// Ініціалізація масива доступних фігур
void initFigures()
{
    //  *
    // ***

    VVB f1(2, VB (3));
    f1[0][0] = f1[0][1] =
    f1[0][2] = f1[1][1] = true;

    // *
    // ***

    VVB f2(2, VB (3));
    f2[0][0] = f2[0][1] =
    f2[0][2] = f2[1][0] = true;

    //   *
    // ***

    VVB f3(2, VB (3));
    f3[0][0] = f3[0][1] =
    f3[0][2] = f3[1][2] = true;

    // **
    // **

    VVB f4(2, VB (2));
    f4[0][0] = f4[0][1] =
    f4[1][0] = f4[1][1] = true;

    // *
    // *
    // *
```

```
    // *

    VVB f5(4, VB (1));
    f5[0][0] = f5[1][0] =
    f5[2][0] = f5[3][0] = true;

    // **
    //  **

    VVB f6(2, VB (3));
    f6[0][1] = f6[0][2] =
    f6[1][0] = f6[1][1] = true;

    //  **
    // **

    VVB f7(2, VB (3));
    f7[0][0] = f7[0][1] =
    f7[1][1] = f7[1][2] = true;

    figures.push_back(f1);
    figures.push_back(f2);
    figures.push_back(f3);
    figures.push_back(f4);
    figures.push_back(f5);
    figures.push_back(f6);
    figures.push_back(f7);
}

// Масив заповнених клітинок у стакані
VVB a;

// Функція, що визначає чи вільна певна
// клітинка стакана
bool getA(int x, int y)
{
    return x < 0 || y < 0 || y > 9 ||
           (x < a.size () && a[x][y]);
}

// Функція, що визначає чи можна поставити
// фігуру на певну позицію у стакані
bool canPut(const VVB& f, int x, int y)
{
    for (int i=0; i<f.size (); ++i)
        for (int j=0; j<f[i].size (); ++j)
            if (f[i][j] && getA(x+i, y+j))
                return false;
    return true;
}

// Функція, що визначає чи може зупинитись
// фігура у певній позиції у стакані
bool canStop(const VVB& f, int x, int y)
{
```

```
    for (int i=0; i<f.size (); ++i)
        for (int j=0; j<f[i].size (); ++j)
            if (f[i][j] &&
                getA(x+i-1, y+j))
                return true;
    return false;
}

// Функція, що визначає список позицій,
// яких може досягти і в яких може
// зупинитись вхідна фігура після
// переміщень
VPII getPositions(const VVB& f)
{
    VPII res;

    VB prev(10, true);

    for (int i=a.size(); i>=0; --i)
    {
        VB can(10, false);
        VB cur(10, false);

        for (int j=0; j<10; ++j)
        {
            can[j] = canPut(f, i, j);
            cur[j] = can[j] && prev[j];
        }

        for (int j=0; j<9; ++j)
            if (cur[j] && can[j+1])
                cur[j+1] = true;

        for (int j=9; j>0; --j)
            if (cur[j] && can[j-1])
                cur[j-1] = true;

        for (int j=0; j<10; ++j)
            if (cur[j] &&
                canStop(f, i, j))
                res.push_back(PII(i, j));

        if (cur == VB(10, false))
            break;

        swap(cur, prev);
    }

    return res;
}

// Функція, що розміщує фігуру на певній
// позиції в стакані
void doPut(const VVB& f, int x, int y)
{
```

```
    while (a.size() < x+f.size())
        a.push_back(VB(10));

    for (int i=0; i<f.size(); ++i)
        for (int j=0; j<f[i].size(); ++j)
            if (f[i][j])
                a[i+x][j+y] = true;
}

int main()
{
     freopen("tetris.in", "r", stdin);
     freopen("tetris.out", "w+", stdout);

    initFigures();

    int n;

    cin >> n;

    while (n--)
    {
        int ff;
        cin >> ff;

        VVB f = figures[ff-1];

        int rx = INF;
        int ry = INF;
        int rr = 0;

        // Вибір найнижчою у стакані
        // позицію куди можна помістити
        // фігуру після поворотів.
        for (int rot=0; rot<4; ++rot)
        {
            VPII pos = getPositions(f);

            for (int i=0; i<pos.size();
                 ++i)
                if (pos[i].first < rx)
                {
                    rx = pos[i].first;
                    ry = pos[i].second;
                    rr = rot;
                }

            f = rotate(f);
        }

        // Розміщення фігури у вибраній
        // позиції
        for (int rot=0; rot<rr; ++rot)
            f = rotate(f);
```

```
        doPut(f, rx, ry);

        // Виведення вибраної позиції
        cout << rr * 90 << " " << ry
             << " " << rx << endl;
    }

     return 0;
}
```

3. Військові навчання

```
/* GCC */
#include <algorithm>
#include <iostream>
using namespace std;

int fen[1201][1201];
pair<int,int> now[50001];
// Додавання до дерева Фенвіка
inline void add(int x,int y,int c)
{
     x+=600; y+=600;
     for (int i=x;i<=1200;i|=i+1)
          for (int j=y;j<=1200;j|=j+1)
               fen[i][j]+=c;
}
// Пошук суми у дереві Фенвіка
inline int sum(int x,int y)
{
     if (x<-600 || y<-600) return 0;
     if (x>600) x=600;
     if (y>600) y=600;
     x+=600; y+=600;
     int ans=0;
     for (int i=x;i>=0;i=(i&(i+1))-1)
          for (int j=y;j>=0;j=(j&(j+1))-1)
               ans+=fen[i][j];
     return ans;
}
// Розділення довільного прямокутника на
// прямокутники з лівим нижнім кутом у
// початку координат (формула вкл-викл)
int sum(int x1,int y1,int x2,int y2)
{
     return sum(x2,y2)-sum(x1-1,y2)
           -sum(x2,y1-1)+sum(x1-1,y1-1);
}
// Додавання солдата
void addSoldier(int x,int y,int s)
{
     now[s]=make_pair(x+y+1e6,x-y+1e6);
     add(x+y,x-y,1);
}
// Видалення солдата
```

```
int removeSoldier(int s)
{
     if (now[s].first==0) return 0;
     add(now[s].first-1000000,
         now[s].second-1000000,-1);
     now[s].first=now[s].second=0;
     return 1;
}
// Відповідь на запит
int query(int x,int y,int d)
{
     return sum(x+y-d,x-y-d,
                 x+y+d,x-y+d);
}
```

4. Парність

```
{ Free Pascal }
const n_=1234567;
var w: array[1..n_] of longint;
    b: array[1..n_] of boolean;
    a: byte;            {залишок mod 2}
    l,           {номер послідовності}
    m,      {кількість послідовностей}
    n,         {довжина послідовності}
    p,                  {довжина циклу}
  j,k: longint; {лічильники елементів}
    s: string;       {рядок-відповідь}
   i,o: text;                    BEGIN
assign(i,'oddeven.in');      reset(i);
assign(o,'oddeven.out');   rewrite(o);
s:='';
readln(i,m);
for l:=1 to m do                 BEGIN
   a:=0;
   read(i,n);
   for j:=1 to n do begin
      read(i,w[j]);
      b[j]:=false     end;
   for j:=1 to n do if not b[j]
                    then         begin
       p:=0;
       k:=j;
       repeat b[k]:=true;
              k:=w[k];
              inc(p);
        until b[k];
       if p mod 2 = 0
       then a:=(a+1) mod 2        end;
   case a of
        0: s:=s+'0';
        1: s:=s+'1'
   end                            END;
writeln(o,s);
close(i);
close(o)                          END.
```

5. Фарбована сума

```
  /* GCC */

  using namespace std;

  #include <stdio.h>
  #include <algorithm>
  #include <functional>

  #define base 10
  // Основа системи числення.

#define max_n 100000
  // Максимальна кількість доданків.

#define max_d 9
  // Максимальна кількість цифр у доданку.

  int n, numbers[max_n];
// Кількість доданків та масив,
// що їх зберігатиме.

  int main()
  {
      freopen("sum.in","r",stdin);
      freopen("sum.out","w",stdout);
      scanf("%d",&n);

    int zeros = 0;
      // Кількість нулів наприкінці доданків.

    for (int i = 0; i < n; i++)
      {
          int current, digits = 0;
          // Поточний зчитаний доданок
          // та кількість цифр у ньому.

        scanf("%d",&current);
          numbers[i] = current;
          while (current % base == 0)
          {
              zeros++;
              digits++;
              current /= base;
          }
          while (current > 0)
          {
              digits++;
              current /= base;
          }
          for ( ; digits < max_d; digits++)
                  numbers[i] *= base;
          // Дописуємо в кінець зчитаного до-
```

```
    // данка необхідну кількість нулів.
}
sort(numbers,numbers+n,greater<int>());

int factor = 1;
for (int i = max_d - 1; i > 0; i--)
    factor *= base;
// Число, на яке слід поділити
// без остачі, щоб отримати
// першу цифру доданка.

while (numbers[0] > 0)
{
    int digit = numbers[0] / factor;
    // Перша цифра найбільшого
    // на даний момент доданка.

    printf("%d",digit);
    numbers[0] =
  base * (numbers[0] - digit * factor);
    // Відкидаємо першу цифру,
    // ззаду дописуємо нуль.

    int current = 0;
    // Поточний елемент купи, який,
    // можливо, слід поміняти
    // з дочірнім.

    bool changed;
    // Чи відбувся на останньому кроці
    // обмін елементів купи.

    do
    {
        changed = false;
        int children[] =
    {current * 2 + 1, current * 2 + 2};
        // Можливі індекси елементів
        // купи для обміну з поточним.

        if (children[0] < n)
        {
            int child = children[0];
            // Індекс потенційного еле-
            // мента купи для обміну.

            if (children[1] < n &&
 numbers[children[1]] > numbers[child])
                child = children[1];
            if (numbers[child] >
                      numbers[current])
            {
                // Обмін елементів
                // купи місцями.
```

```
                    swap(numbers[child],
                         numbers[current]);
                    changed = true;
                    current = child;
                }
            }
        } while (changed);
    }
    for (int i = 0; i < zeros; i++)
        printf("0");
    printf("\n");
    return 0;
}
```

6. Розклад уроків

```
{ Turbo Pascal}
PROGRAM Schedule;
const s=50; {Максимально можливий номер
       предмету.}
var e:array[1..50,1..50] of byte; {Елемент
          e[i,j] показує, скільки потрібно
         провести уроків з j-того предмету
                          в i-тому класі}
    sch:array[1..50,1..50] of byte;
                          {Розклад уроків}
    dc:array[1..50] of byte;
      {Степені вершин, що позначають класи}
    ds:array[1..50] of byte;
   {Степені вершин, що позначають предмети}
    pc:array[0..50] of byte; {Елемент pc[i]
       позначає номер предмету,
       співставленого i-у класу у
       паросполученні. Якщо pc[i]=0,
       то i-тому класу не співставлено
       жодного предмету}
    ps:array[0..50] of byte; {Елемент ps[j]
       позначає номер класу, співставленого
       j-тому предмету в паросполученні.
       Якщо ps[j]=0, то j-тому предмету не
       співставлено жодного класу}
    i,j,k: word;       {Індекси для циклів}
    n:word;            {Кількість класів}
    q:word; {Номер стовпчика розкладу,
               який будують у даний момент}
    t:word; {Максимально можлива кількість
       уроків}
    idsch:boolean; {Індикатор, який показує,
       чи існує потрібний розклад - програма
       розрахована і на цей випадок,  хоча в
                          тестах його немає}
{Знаходження предметів,
          що співставляют насиченим класам}
PROCEDURE FindParingForClasses;
  var idfind:array[1..50] of boolean;
       {Елемент idfind[j] показує, чи
```

```
      досягнута вершина, відповідна
      j-у предмету, на даний момент
      пошуку в ширину}
    prv:array[1..50] of byte; {Елемент
      prv[j] показує, з якої вершини ми
      прийшли у процесі пошуку в ширину
      до вершини, відповідної j-тому
      предмету. Маємо на увазі не
      безпосередньо попередня вершина, а
      парна до неї в паросполученні.
      Тобто prv[j] також відповідає
      предмету, а не класу - це дозволяє
      зменшити вдвічі частину програми,
      що описує пошук у ширину}
    v:array[0..50] of byte; {Список
      відповідних предметам вершин,
      досягнутих у процесі пошуку в
      ширину}
    i,j:word;     {Індекси для циклів}
    k,k1,k2:word; {Індекси, що керують
      пошуком у ширину.
      В елементах від v[0] до v[k-1] -
      повністю оброблені вершини, тобто
      вершини, всі сусіди яких уже
      проглянуто.
      В елементах від v[k] до v[k1] -
      вершини, сусіди яких проглядають у
      даний момент.
      В елементи від v[k1+1] до v[k2]
      записують щойно знайдені сусіди
      тих вершин, які знаходяться в
      елементах від v[k] до v[k1].}
    idstop:boolean; {Індикатор завершення
      пошуку в ширину. Набуває значення
      TRUE, коли знайдений предмет, не
      включений у поточне паросполучення}
Begin
  for j:=1 to s do ps[j]:=0;
  for i:=1 to n do
    {Пошук пари для i-того класу,
     якщо він є насиченим: його
     навантаження дорівнює q}
    if (dc[i]=q) then begin
       k:=0;
       k1:=0;
       k2:=0;
       {Створюється віртуальна 0-ова
        вершина, яка є парною до i-того
        класу. Це дозволяє виконати
        перший напівцикл пошуку в ширину
        в межах основного циклу.}
       ps[0]:=i;
       v[0]:=0;
       for j:=1 to s do idfind[j]:=FALSE;
       idstop:=FALSE;
```

```
        {Основний цикл пошуку в ширину.}
        while not(idstop) do begin
          while ((k<=k1)and(not(idstop)))
          do begin
            j:=s;
            while ((j>0)and(not(idstop)))
            do begin
              if ((e[ps[v[k]],j]<>0)and
                  (not(idfind[j]))) then
              begin
                 prv[j]:=v[k];
                 idfind[j]:=TRUE;
                 Inc(k2);
                 v[k2]:=j;
                 if (ps[j]=0) then
                    idstop:=TRUE;
              end;
              if not(idstop) then Dec(j);
            end;
            Inc(k);
          end;
          k1:=k2;
        end;
        {Перебудова паросполучення, яка
         включає у нього i-тий клас та
         j-тий предмет. Для кожного
         предмету вказано парний йому
         клас, а не навпаки.}
        while (j<>0) do begin
          ps[j]:=ps[prv[j]];
          j:=prv[j];
        end;
      end;
  End;

PROCEDURE FindParingForSubjects;
  var idfind:array[1..50] of boolean;
       {Елемент idfind[i] показує, чи
        досягнуто вершину, що відповідає
        i-тому класу, на даний момент
        пошуку в ширину.}
      prv:array[1..50] of byte; {Елемент
        prv[i] показує, з якої вершини ми
        прийшли у процесі пошуку в ширину
        до вершини, відповідної i-тому
        класу. Мається на увазі не
        безпосередньо попередня вершина, а
        парна до неї в паросполученні.
        Тобто prv[i] також відповідає класу
        а не предмету - це дозволяє
        зменшити вдвічі частину програми,
        що описує пошук у ширину.}
      v:array[0..50] of byte; {Список
        відповідних класам вершин,
        досягнутих у процесі пошуку в
```

```
      ширину.}
    i,j:word; {Індекси для різних
      циклів.}
    k,k1,k2:word; {Індекси, що керують
      пошуком у ширину.
      В елементах від v[0] до v[k-1] -
      повністю оброблені вершини, всі
      сусіди яких уже проглянуто.
      В елементах від v[k] до v[k1] -
      вершини, сусіди яких проглядають
      в даний момент.
      В елементи від v[k1+1] до v[k2]
      записують щойно знайдені сусіди
      тих вершин, розташованих у
      елементах від v[k] до v[k1]}
    idstop:boolean; {Індикатор завершення
      пошуку в ширину. Набуває значення
      TRUE, коли знайдено клас, не
      включений у поточне паросполучення,
      або клас, парний до якого предмет
      не є насиченим.}
Begin
  for i:=1 to n do pc[i]:=0;
  for j:=1 to s do pc[ps[j]]:=j;
  {За списком пар для предметів будуємо
   список пар для класів}
  for j:=1 to s do
    {Пошук пари для j-того предмету,
     якщо він є насиченим: його
     навантаження цього предмету рівне q.
     Пошук здійснюємо за умови, що цю
     пару не було знайдено у попередній
     процедурі.}
    if ((ds[j]=q)and(ps[j]=0)) then begin
       k:=0;
       k1:=0;
       k2:=0;
       {Створення віртуальної нульової
        вершини, що у парі з j-тим
        предметом. Це дозволяє виконати
        перший напівцикл пошуку в ширину
        в межах основного циклу}
       pc[0]:=j;
       v[0]:=0;
       for i:=1 to n do idfind[i]:=FALSE;
       idstop:=FALSE;
       {Основний цикл пошуку в ширину}
       while not(idstop) do begin
         while ((k<=k1)and(not(idstop)))
         do begin
           i:=n;
           while ((i>0)and(not(idstop)))
           do begin
             if ((e[i,pc[v[k]]]<>0)and
                 (not(idfind[i]))) then
```

```
              begin
                 prv[i]:=v[k];
                 idfind[i]:=TRUE;
                 Inc(k2);
                 v[k2]:=i;
                 if (pc[i]=0) then
                    idstop:=TRUE;
                 if (pc[i]>0) then
                    if (ds[pc[i]]<q) then
                       idstop:=TRUE;
              end;
              if not(idstop) then Dec(i);
            end;
            Inc(k);
          end;
          k1:=k2;
        end;
        {Перебудова паросполучення, яка
         включає у нього j-тий предмет та
         i-тий клас. Можливе виключення з
         паросполучення предмету, який
         тільки-но був парним до і-того
         класу, якщо цей предмет не був
         насиченим. Для кожного класу
         вказано парний йому предмет, а
         не навпаки}
        while (i<>0) do begin
          pc[i]:=pc[prv[i]];
          i:=prv[i];
        end;
      end;
  End;

BEGIN
  {Читання вхідних даних із одночасним
   перетворенням їх на масив e}
  assign(input,'schedule.in');
  assign(output,'schedule.out');
  reset(input);
  read(n,t);
  for i:=1 to n do begin
    for j:=1 to s do e[i,j]:=0;
    for j:=1 to t do begin
      read(k);
      if (k<>0) then Inc(e[i,k]);
    end;
  end;
  close(input);
  idsch:=TRUE;
  {Визначення степенів вершин,
   що відповідають предметам,
   з одночасною перевіркою
   існування потрібного розкладу}
  for j:=1 to s do begin
    k:=0;
```

```
    for i:=1 to n do k:=k+e[i,j];
    idsch:=idsch and (k<=t);
    ds[j]:=k;
  end;
  rewrite(output);
  {Якщо потрібного розкладу не існує,
   виводимо -1}
  if not(idsch) then writeln(-1);
  {Якщо потрібний розклад існує,
            проводимо його пошук}
  if (idsch) then begin
     {Визначення степенів вершин,
      що відповідають класам}
     for i:=1 to n do begin
       dc[i]:=0;
       for j:=1 to s do
         dc[i]:=dc[i]+e[i,j];
     end;
     {Побудова стовпчиків розкладу з t-того
      до першого у порядку спадання}
     for q:=t downto 1 do begin
       {Пошук паросполучення, за яким
        створюємо розклад на q-тий урок}
       FindParingForClasses;
       FindParingForSubjects;
       {Занесення знайденого паросполучення
        у розклад та його виключення з
        масиву e, в якому зберігаємо
        ребра дводольного графа}
       for i:=1 to n do begin
         sch[i,q]:=pc[i];
         if (pc[i]<>0) then begin
            Dec(dc[i]);
            Dec(ds[pc[i]]);
            Dec(e[i,pc[i]]);
         end;
       end;
     end;
     {Виведення знайденого паросполучення}
     for i:=1 to n do
       for j:=1 to t do begin
         write(sch[i,j]);
         if (j<t) then write(' ');
         if (j=t) then writeln;
       end;
  end;
  close(output);
END.
```