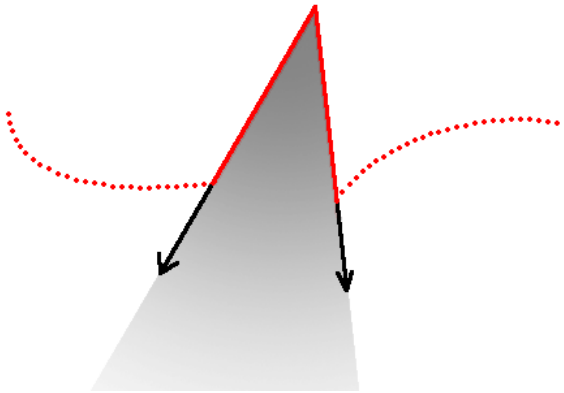
# РЕШЕНИЯ НА ЗАДАЧИТЕ ОТ BOI 2008, ПРЕДЛОЖЕНИ ОТ БЪЛГАРСКИТЕ УЧАСТНИЦИ В БАЛКАНИАДАТА

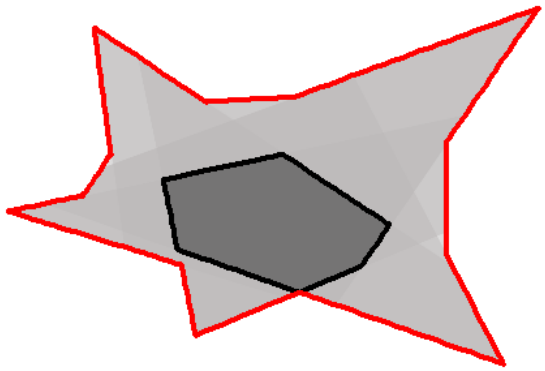
## Задача 1. Наблюдение на езеро

**Автор: Стефан Аврамов**

Нека разгледаме две съседни отсечки от многоъгълника. Ако приемем, че това са два лъча с начало точката между тях, то те ще ограничат областта, в която се намират всички точки, от които се виждат двете отсечки и които евентуално са във вътрешността на многоъгълника:



Ако разгледаме тези области за всеки две съседни отсечки, то тогава множеството от точките, които се търсят в задачата, ще се намира в многоъгълника, който се образува като пресечем всички области (или това е общата им част). Той е изпъкнал и се намира във вътрешността на началния многоъгълник, тъй като е ограничен от неговите отсечки:



Точките във върховете на обединяващия многоъгълник също могат да бъдат отговор на задачата. Освен това те лежат и върху пресечна точка на поне две прави, определени от отсечките на началния многоъгълник. Това означава, че ако проверим пресечните точки между всеки две отсечки (прави), ще проверим и всички точки във върховете на втория многоъгълник (ако той съществува разбира се).

Сега остава само да определим дали от дадена точка се вижда целия многоъгълник. Това става като за всеки две отсечки проверим дали едната скрива част от другата. Сложността на това е N^2, което дава O(N^4) за целия алгоритъм (проверяваме N^2 точки).

С малко повече наблюдение се вижда, че е достатъчно последното да е изпълнено само за всеки две съседни отсечки (проверката също се улеснява поради общия край - отсечките трябва само да са в различни равнини спрямо общата им точка и точката, която проверяваме). По този начин се гарантира, че при обхождането на многоъгълника обратно на часовниковата стрелка, винаги всяка следваща отсечка ще продължава в положителна посока спрямо точката, която проверяваме, и няма начин някоя отсечка да застане пред друга. Така алгоритъма става с O(N^3).

# Задача 2. Перални

**Автор: Антон Анастасов**

Още с прочитане на условието се вижда, че задачата е оптимизационна. Ограниченията предполагат решение с полиномиална сложност, затова можем да забравим използването на какъвто и да е вид пълно изчерпване. Нека разгледаме интервала [u, v]. Нека в u и v има хотели със перални, а всички останали хотели, които се намират в интервала нямат собствени перални. Лесно можем да съобразим, че всички хотели, които са вътре в интервала, биха избрали или u или v като най-близък хотел с пералня. Нека опитаме да решим задачата като я разбием на подзадачи. Приемаме, че сме разпределили оптимално всички хотели до u-тия (като последната пералня е в u-тия хотел) и имаме още k на брой перални да поставим. Ако изберем следващата пералня да се намира в v-тия хотел (v > u) и всеки хотел в интервала [u, v] свържем с по-близкия хотел измежду u и v свеждаме задачата до такава от същия вид, но с по-малък размер. Вече е ясно, че можем да решим задачата като използваме динамично оптимиране.

Подзадача можем да дефинираме посредством двуаргумената функция *dp(last_hotel, laundries_left)*, която ни дава минималната сума от разстояния, която може да се получи като знаем, че сме разпределили всички хотели измежду [1, hotel_last], последната пералня е в hotel_last и имаме максимум laundries_left перални на разположение да поставим в интервала [last_hotel, n]. Нека дефинираме още една функция: *assign(u, v)*, която ни дава сумата от разстоянията при разпределянето на всички хотели в интервала [u, v] (като знаем, че в u и v има перални) като всеки от тези хотели се свързва с по-близкия хотел измежду u и v. Рекурентната зависимост за функцията dp изглежда така:

(1) при laundries_left = K - 1, връщаме сумата на разстоянията на хотели [1, last_hotel] до last_hotel
(2) в противен случай, връщаме минимума измежду dp(prev_hotel, laundries_left + 1) + assign(prev_hotel, last_hotel), 1 <= prev_hotel <= last_hotel

Отговора е минимума измежду: dp(hotel, k - 1) + сумата на разстоянията на хотели [hotel, n] до hotel, където 1 <= hotel <= n

Нека изследваме сложността на алгоритъма. Броят на подзадачите е n * k. За всяка подзадача имаме максимум n случая да разгледаме (2), като във всеки случай имаме по едно извикване на функцията *assign*. Най-тривиалната реализация на *assign* има сложност O(n) (за всеки хотел в интервала [u, v] добавяме по-малкото от разстоянието до хотел u или до хотел v). Тогава целия алгоритъм става със сложност брой_подзадачи * брой_случаи_за_подзадача * сложност_на_assign, което е O(n ^ 3 * k). Решение с такава сложност е достатъчно за получаването на максимален брой точки на официалните тестове. Това решение може да бъде подобрено. Една от лесните оптимизации е следната: вместо всеки път да пресмятаме функцията *assign*, можем да пресметнем отначало отговорите за всички възможни нейни извиквания и да ги запазим в двумерен масив. Броят на различните аргументи, които може да получи е точно n * n (1 <= u <= n, 1 <= v <= n). Така към алгоритъма ще добавим O(n ^ 3) (n * n * сложност_на_assign), но като цяло алгоритъма ще стане O(n ^ 3 + n ^ 2 * k * O(1)), защото еднократно извикване на *assign* вече ще има сложност O(1). Така получаваме алгоритъм със сложност

O(n ^ 3 + n ^ 2 * k), който е по-добър от предишния.

Нека разледаме функцията *assign(u, v)*. Всички хотели, които са по-близко до u ще използват пералнята на u, а всички останали - тази на v. Ако можем да намерим ефективно последния хотел z, който би използвал u, можем да разделим тази функция на две части - намиране на сумата на разстоянията от на всички хотели в интервала [u, z] до u, и на тези в интервала [z + 1, v] до v. Нека видим как можем да намерим сумата на разстоянията на хотелите в даден интервал до някой от двата крайни за интервала. Нека дефинираме два масива едномерни масива sum_left[] и sum_right[]. sum_left[i] е равно на сумата на разстоянията на хотелите в интервала [1, i] до хотел 1. sum_right[i] е равно на сумата на разстоянията на хотелите в интервала [i, n] до хотел n. Как можем да намерим сумата от разстоянията на хотелите в интервала [a, b] до хотел b: sum_right[1] - sum_right[a + 2] - (b - a + 1) * (x[N - 1] - x[b]). Оставяме на читателя да докаже защо горният израз дава нужната сума. Сега остава да можем ефективно да намираме горе-споменатото z. Тъй като итерираме prev_hotel от 1 до last_hotel лесно можем да забележим, че z ще се увеличава постепенно. Затова вместо винаги да намираме z наново, можем просто да го ъпдейтваме от предишното z с увеличаване. Това можем да направим с един допълнителен цикъл. На пръв поглед не спечелихме нищо - премахнахме един цикъл и въведохме един нов. Посредством амортизиран анализ може да се докаже, че вътрешния цикъл ще се изпълни сумарно максимум n пъти за всичките n изпълнения на външния цикъл. По този начин решението става със сложност O(n ^ 2 * k).

Новата функция assign изглежда така:
assign(u, v, z)
  return sum_left [z + 1] - sum_left [u] - (z - u + 1) * (x [u] - x [0]) + sum_right [z + 2] - sum_right [v + 2] - (v - z) * (x [N - 1] - x [v]);

ПРИМЕРНИ РЕАЛИЗАЦИИ:

O(n ^ 3 * k) решението:

```
/*
    FROM: Balkan Olympiad 2008, Day 1
    PROB: laundry
    KEYW: dynamic programming
*/
// Complexity: O (n ^ 3 * k)
#include <cstring>
#include <iostream>
#include <algorithm>
using namespace std;
const int MAXN = 256;
const int MAXK = 32;
const int INF = 0x3f3f3f3f;
int N, K;
int x [MAXN];
int best [MAXK][MAXN];
void read ()
{
    scanf ("%d %d", &N, &K);
```

```
    for (int i = 0; i < N; ++i)
        scanf ("%d", &x [i]);
}
int solve ()
{
    if (K >= N)
        return 0;
    memset (best, 0x3f, sizeof (best));

    for (int i = 0; i < N; ++i)
    {
        best [K - 1][i] = 0;
        for (int j = 0; j < i; ++j)
            best [K - 1][i] += x [i] - x [j];
    }

    for (int i = K - 2; i >= 0; --i)
        for (int j = 0; j < N; ++j)
        {
            for (int k = 0; k <= j; ++k)
            {
                int dist = 0;
                for (int l = k; l <= j; ++l)
                    dist += min (abs (x [j] - x [l]), abs (x [k] - x [l]));
                best [i][j] = min (best [i][j], best [i + 1][k] + dist);
            }
        }

    int res = INF;
    for (int i = 0; i < N; ++i)
    {
        int cur = best [0][i];
        for (int j = i; j < N; ++j)
            cur += x [j] - x [i];
        res = min (res, cur);
    }
    return res;
}
int main ()
{
    read ();
    printf ("%d\n", solve ());
    return 0;
}
```

O(n ^ 3 + n ^ 2 * k):

```
/*
    FROM: Balkan Olympiad 2008, Day 1
    PROB: laundry
    KEYW: dynamic programming
*/
// Complexity: O (n ^ 3 + n ^ 2 * k)
#include <iostream>
#include <algorithm>
using namespace std;
const int MAXN = 512;
const int MAXK = 64;
const int INF = 0x3f3f3f3f;
int N, K;
int x [MAXN];
int best [MAXK][MAXN];
int dist [MAXN][MAXN];
void read ()
{
    scanf ("%d %d", &N, &K);
    for (int i = 0; i < N; ++i)
        scanf ("%d", &x [i]);
}
int solve ()
{
    if (K >= N)
        return 0;
    memset (best, 0x3f, sizeof (best));

    for (int i = 0; i < N; ++i)
    {
        best [K - 1][i] = 0;
        for (int j = 0; j < i; ++j)
            best [K - 1][i] += x [i] - x [j];
    }

    for (int i = 0; i < N; ++i)
        for (int j = i; j < N; ++j)
        {
            dist [i][j] = 0;
            for (int k = i; k <= j; ++k)
                dist [i][j] += min (abs (x [k] - x [i]), abs (x [k] - x [j]));
        }

    for (int i = K - 2; i >= 0; --i)
```

```
        for (int j = 0; j < N; ++j)
        {
            for (int k = 0; k <= j; ++k)
                best [i][j] = min (best [i][j], best [i + 1][k] + dist [k][j]);
        }

    int res = INF;
    for (int i = 0; i < N; ++i)
    {
        int cur = best [0][i];
        for (int j = i; j < N; ++j)
            cur += x [j] - x [i];
        res = min (res, cur);
    }
    return res;
}
int main ()
{
    read ();
    printf ("%d\n", solve ());
    return 0;
}
```

O(n ^ 2 * k):

```
/*
    FROM: Balkan Olympiad 2008, Day 1
    PROB: laundry
    KEYW: dynamic programming
*/
// Complexity: O (n ^ 2 * k)
#include <iostream>
#include <algorithm>
using namespace std;
const int MAXN = 1024;
const int MAXK = 128;
const int INF = 0x3f3f3f3f;
int N, K;
int x [MAXN];
int sum_left [MAXN], sum_right [MAXN];
int best [MAXK][MAXN];
void read ()
{
    scanf ("%d %d", &N, &K);
    for (int i = 0; i < N; ++i)
```

```
        scanf ("%d", &x [i]);
}
inline int assign (int u, int v, int z)
{
    return sum_left [z + 1] - sum_left [u] - (z - u + 1) * (x [u] - x [0]) +
        sum_right [z + 2] - sum_right [v + 2] - (v - z) * (x [N - 1] - x [v]);
}
int solve ()
{
    if (K >= N)
        return 0;

    sum_left [0] = 0;
    for (int i = 1; i <= N; ++i)
        sum_left [i] = sum_left [i - 1] + x [i - 1] - x [0];
    sum_right [N + 1] = 0;
    for (int i = N; i >= 1; --i)
        sum_right [i] = sum_right [i + 1] + x [N - 1] - x [i - 1];
    memset (best, 0x3f, sizeof (best));
    for (int i = 0; i < N; ++i)
        best [K - 1][i] = sum_right [1] - sum_right [i + 2] - (i + 1) * (x [N - 1] - x
[i]);

    for (int i = K - 2; i >= 0; --i)
        for (int j = 0; j < N; ++j)
        {
            int z = 0;
            for (int k = 0; k < j; ++k)
            {
                while (z + 1 < j && x [z + 1] - x [k] <= x [j] - x [z + 1]) ++z;
                best [i][j] = min (best [i][j], best [i + 1][k] + assign (k, j, z));
            }
        }

    int res = INF;
    for (int i = 0; i < N; ++i)
        res = min (res, best [0][i] + sum_left [N] - sum_left [i + 1] - (N - 1 - i) *
(x [i] - x [0]));
    return res;
}
int main ()
{
    read ();
    printf ("%d\n", solve ());
    return 0;
}
```

Съществуват и решения, които се базират на оптимизация на вътрешния цикъл на динамично програмиране посредством използване на опашка. Познати са ни две различни реализации със сложности: O(n * k * log(n) + n ^ 2) и O(n * k * log(n) ^ 2). Тяхното обяснение е твърде сложно. Ако все пак имате интерес с радост бихме се опитали да Ви ги обясним. Можете да ни потърсите в Skype: brotherbg (Антон Анастасов) и exod40- (Румен Христов). В бъдеще очаквайте добавяне към този анализ горните две решения.

## Задача 3. Лъжливите рибари

**Решение 1.**
**Автор: Румен Христов**

Задачата на пръв поглед изглежда, че може да се реши с някакъв алчен (greedy) алгоритъм. Най-простият алгоритъм в случая е, когато една риба се среща само при един рибар или в списъка на един рибар има само една риба, тогава установяваме, че този рибар е хванал тази риба и премахваме рибата от списъците на всички рибари. Продължаваме това, докато нито едно от горните две условия не е изпълнено. Но при малко повече разсъждения или проверяване с бавно решение, бихме установили, че този алгоритъм греши при тестове от рода:

5

1 2 5 0

3 4 0

1 4 0

3 4 0

1 2 5 0

В този пример никое от горните две условия не е изпълнено, но сме сигурни, че 3-тия рибар ще вземе първата риба, защото, ако вземе четвъртата, то тогава за 2-рия и 4-тия рибар ще остане само 3-тата риба, т.е. това е невъзможно подреждане.

Когато видим задача, в която има няколко елемента от едно множество (в случая рибарите) и няколко елемента от друго множество (в слуачая рибите) и че някои елементи от първото множество избират/хващат/убиват (match) по няколко елемента (в случая точно един) елемент от второто множество, то тогава е доста вероятно тази задача да се решава с максимален поток (maximum flow) или по-точно двойкосъчетание (bipartite matching). Проблемът за намиране на максимален поток в двуделен (bipartite) граф е доста известен и може да го намерите в доста книги и статии*.

Как строим двуделния граф? В лявата част на графа стоят рибарите, а в дясната част рибите. Имаме ребро в графа, когато някой рибар заявява, че е хванал някоя риба. Ребрата, които участват в максималното двойкосъчетание (със сигурност ще получим такова двойкосъчетание, защото по условие всеки рибар е хванал точно една риба), са една възможност за всеки рибар. За жалост живота не е толкова лесен и трябва да проверим дали няма повече от една възможна риба за някой рибар. Вземаме всеки рибар и виждаме, че има максимално двойкосъчетание, в което е хванал Pi-тата риба. Конструираме нов граф, същия като предишния без реброто I → Pi, т.е. казваме, че търсим такова двойкосъчетание, където не искаме i-тия рибар да хване същата риба. Имаме два

варианта:

Получваме ново максимално двойкосъчетание, т.е. всеки рибар отново се е съчетал с някоя риба, като I-тия не се е съчетал със същата риба, т.е. имаме повече от един вариант за съчетание на I-тия рибар, в което се получава максимално двойкосъчетание. Това означава, че не сме сигурни коя риба е хванал I-тия рибар.

Не получаваме ново максимално двойкосъчетание, т.е. единствената възможност за съчетание на I-тия рибар е Pi-тата риба.

Сложността на описаният алгориритъм е *N * Сложността за намиране на максимален поток,* заради малките ограничения ( N <= 30 ) може да използваме и най-простите алгоритми за намиране на максимален поток.

* - Може да намерите информация за максимален поток и максмални двойкосъчетания в различни източници, които лесно бихте намерили чрез Google :), но хубаво описание на български на проблема има в “Програмиране =++ алгоритми”, а на английски в Wikipedia.

Примерна реализация, която намира максимален поток чрез алгоритъма на Диниц:

```
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;

const int MAXN = 1 << 7;

struct edge {
	int to;
	int cap;
	int revy;
	int isrev;
	edge () {}
	edge ( int _to , int _cap , int _isrev )
	{ to = _to; cap = _cap; isrev = _isrev;	}
};

int n;
int ans[MAXN];

int v;
vector < edge > a[MAXN];
int source , sink;
```

```
int d[MAXN];
int used[MAXN];

void make_edge ( int q , int w ) {
      int s1 = (int)a[q].size();
      int s2 = (int)a[w].size();

      a[q].push_back ( edge ( w , 1 , 0 ) );
      a[w].push_back ( edge ( q , 0 , 1 ) );

      a[q][s1].revy = s2;
      a[w][s2].revy = s1;
}

void read() {
      int i;
      int x;

      scanf ("%d",&n);

      source = 2 * n + 1;
      sink = 2 * n + 2;
      v = 2 * n + 2;

      for (i=1;i<=n;i++) {
            make_edge ( source , i );
            make_edge ( n + i , sink );
      }

      for (i=1;i<=n;i++)
            while ( scanf ("%d",&x) , x )
                  make_edge ( i , n + x );
}

int bfs() {
      static int q[MAXN];
      int i , j;
      int sz;

      for (i=1;i<=v;i++) {
            used[i] = 0;
```

```
            d[i] = -1;
      }

      q[sz = 0] = sink;
      d[sink] = 0;

      for (i=0;i<=sz;i++)
            for (j=0;j<(int)a[ q[i] ].size();j++)
                  if ( d[ a[ q[i] ][j].to ] == -1 && a[ a[ q[i] ][j].to ][ a[ q[i]
][j].revy ].cap ) {
                        d[ a[ q[i] ][j].to ] = d[ q[i] ] + 1;
                        q[ ++ sz ] = a[ q[i] ][j].to;
                  }

      return d[source] != -1;
}

int dfs ( int node ) {
      if ( node == sink ) return 1;
      used[node] = 1;
      int i;

      for (i=0;i<(int)a[node].size();i++)
            if ( !used[ a[node][i].to ] && d[ a[node][i].to ] + 1 == d[node] &&
a[node][i].cap )
                  if ( dfs ( a[node][i].to ) ) {
                        -- a[node][i].cap;
                        ++ a[ a[node][i].to ][ a[node][i].revy ].cap;
                        return 1;
                  }

      return 0;
}

void fix_edge ( int q , int w , int s , int t ) {
      a[q][w].cap = s;
      a[ a[q][w].to ][ a[q][w].revy ].cap = t;
}

void solve() {
      int i , j , k , d;
```

```
int maxflow;
int fnd = 0;

for (i=1;i<=n;i++) {
	for (k=1;k<=v;k++)
		for (d=0;d<(int)a[k].size();d++)
			if ( !a[k][d].isrev )
				fix_edge ( k , d , 1 , 0 );

	fix_edge ( i , 1 , 0 , 0 );

	maxflow = 0;
	while ( bfs() )
		while ( dfs ( source ) )
			++ maxflow;

	if ( maxflow != n ) {
		ans[i] = a[i][1].to - n;
		continue;
	}

	for (j=2;j<(int)a[i].size();j++)
		if ( !a[i][j].isrev && !a[i][j].cap )
			break;

	ans[i] = a[i][j].to - n;

	fix_edge ( i , 1 , 1 , 0 );
	fix_edge ( source , i - 1 , 1 , 0 );
	fix_edge ( i , j , 0 , 0 );
	fix_edge ( a[i][j].to , 0 , 1 , 0 );
	-- maxflow;

	while ( bfs() )
		while ( dfs ( source ) )
			++ maxflow;

	if ( maxflow == n )
		ans[i] = -1;
}
```

```
	for (i=1;i<=n;i++)
		if ( ans[i] > 0 ) {
			printf ("%d %d\n",i,ans[i]);
			fnd = 1;
		}

	if ( !fnd ) printf ("-1\n");
}

int main() {
	read();
	solve();

	return 0;
}
```

Въпроси/коментари/препоръки/мнения/награди: skype: exod40-

# Задача 3. Лъжливите рибари

**Решение 2.**
**Автор: Стефан Аврамов**

Нека на всеки рибар и всяка риба съпоставим по един връх и всеки рибар е свързан с всяка риба, за която твърди, че я е хванал. Очевидно е, че ако ако пуснем алгоритъм за максимален поток върху този граф (ребрата са с капацитет 1, рибарите са източници, а рибите консуматори), ще си отговорим лесно на въпроса дали е възможно правилно разпределение между рибари и риби (дали на всеки рибар може да се съпостави по единствена риба и на всяка риба по единствен рибар). Това е така, когато потока във графа е равен на броя рибари (риби).

Кога сме сигурни, че един рибар е хванал определена риба? Когато това е единствения начин всеки рибар да получи риба. Ако един рибар може да бъде свързан с две различни риби и когато и в двата случая можем да свържем и останалите рибари с риби => не можем да сме сигурни коя риба е хванал този рибар. От тук логично следва решението: за всеки рибар пробваме да го свържем с всяка риба (до която има ребро), махаме това ребро, и пускаме поток. Ако потока е максимален => е възможно такова разпределение. Ако това е така само при една риба => сигурни сме, че този рибар е хванал тази риба. Така пускаме N^2 потока, което е достатъчно бързо при тези ограничения. Но задачата е възможно да са реши и с N потока. Ако още отначало намерим едно възможно разпределение, можем просто за всеки рибар да махнем реброто, с което е свързан за своята риба, и да се опитаме да намерим ново разпределение (да пуснем нов поток) - ако потока отново е максимален => не сме сигурни за този рибар. Иначе сме сигурни, че е хванал рибата, която сме намерили с потока, пуснат в началото.

# Задача 4. Субстанции

**Автор: Антон Анастасов**

Ключът към решаването на задачата е в преформулирането ѝ в задача за графи. Нека разгледаме ориентиран претеглен граф, чиито върхове биват различните конфигурации на дадените N характеристики. Два върха в този граф са свързани с ребро, когато съществува субстанция преобразуваща началната конфигурация в крайната, а теглото на реброто е времето, което е необходимо, за да се извърши самото преобразуване. Лесно сведохме задачата до стандартна - намиране на най-кратък път в граф. За решаването ѝ съществуват много алгоритми, различаващи се по сложност. Нека първо видим колко голям граф може да получим. Броят на върховете в графа е 2 ^ N (2 на степен N-та). Това е така, защото за всяка от характеристиките имаме точно 2 възможности - или е налична в текущата конфигурация, или не е. Имайки предвид, че N <= 20, броят на върховете може да достигне до 1048576, което означава, че не бихме могли да използваме матрица на съседство за представянето на графа в паметта. Броят на ребрата също е голям - в най-лошия случай всяка субстанция би ни съответствала на едно ребро, което означава, че от всеки връх биха излизали точно M (M <= 100) ребра, а сумарно това е точно M * (2 ^ N), което е над 100 милиона ребра. Ясно се вижда, че дори използването на "списък на съседите" е невъзможно. Как да се справим с този проблем? Нека се върнем на различните алгоритми за намиране на най-кратък път в граф. Единствено алгоритъмът на Дейкстра би могъл да ни свърши работа за такъв голям граф. Използването на ефективна приоритетна опашка е не само препоръчително, но и наложително в текущата задача. Решаването на проблема с построяването на графа лесно може да се избегне при използването на алгоритъма на Дейкстра. Във всеки един момент от изпълнението му не ни е нужен целия граф, нужни са ни единствено съседите на върха, който обработваме. Така, вместо да построяваме целия граф, ще го построяваме само стъпка по стъпка, и няма да се сблъскаме с проблеми свързани с паметта.

РЕАЛИЗАЦИЯ:
Приемаме, че езика, който ползваме, е C++. Нека разледаме основните въпроси по реализацията на решението. Ще използваме приоритетната опашка, предоставена в STL. На преден план излиза въпроса дали отговора на задачата винаги се побира в тип int, тип long long или е нужна реализацията на "дълги" числа. Лесно можем да докажем, че отговора се побира в long long. Най-краткият път (ако път въобще съществува) не би минавал през един връх повече от веднъж (което се съдържа и в дефиницията на път), което означава, че в най-лошия случай пътя може да мине през точно 1048575 ребра. 1048575 * 30000 (теглата на ребрата са ограничени до 30000) = 31457250000, което се побира в long long, но не и в int. Използването на long long е достатъчно. Въпреки това за официалните тестове дори и тип int е достатъчен, но доказателство, че това е така за всички възможни тестове нямаме. Как бихме могли да номерираме върховете? Можем да се възползваме от двоичното представяне на числата. Нула на съответна позиция би означавала, че дадена характеристика отсъства, а единица - присъства. Как можем да проверим дали дадени характеристики присъстват или отсъстват? На помощ ни идват побитовите операции. Предлагаме на читателя, ако не е добре запознат с тях, да им обърне внимание. Тяхното използване е не само лесно, но и много ефективно. Като добър източник можем да предложим туториала в TopCoder. Всички операции със характеристиките могат лесно да се реализират посредством побитови операции. Предложената реализация има сложност O(2 ^ N * M * log(2 ^ N * M)) в най-лошия случай.

ПРИМЕРНА РЕАЛИЗАЦИЯ:

```
/*
    FROM: Balkan Olympiad 2008, Day 2
```

```
    PROB: thecure
    KEYW: graph theory, bit operations, dijkstra
*/
// Complexity: O (2 ^ n * m * log (2 ^ n * m))
#include <queue>
#include <string>
#include <cstring>
#include <iostream>
#include <algorithm>
using namespace std;
struct state
{
    long long cost;
    int node;

    inline bool operator < (const state &rhs) const
    {
        return cost > rhs.cost;
    }
};
const int MAXN = 20;
const int MAXM = 128;
int N, M;
int cost [MAXM];
int c_on [MAXM], c_off [MAXM];
int r_on [MAXM], r_off [MAXM];
long long best [1 << MAXN];
int parse (string str, char ch)
{
    int mask = 0;
    for (int i = 0; i < (int) str.size (); ++i)
    {
        mask <<= 1;
        if (str [i] == ch)
            mask |= 1;
    }
    return mask;
}
void read ()
{
    cin >> N >> M;
    for (int i = 0; i < M; ++i)
    {
        string s, t;
        cin >> cost [i] >> s >> t;
```

```
        c_on [i] = parse (s, '+');
        c_off [i] = parse (s, '-');
        r_on [i] = parse (t, '+');
        r_off [i] = parse (t, '-');
    }
}
long long solve ()
{
    memset (best, 0x3f, sizeof (best));
    best [(1 << N) - 1] = 0;
    priority_queue <state> pq;
    pq.push ((state) {0, (1 << N) - 1});

    int nxt;
    state t;
    while (!pq.empty ())
    {
        t = pq.top ();
        pq.pop ();
        if (best [t.node] < t.cost)
            continue;
        if (t.node == 0)
            return t.cost;
        for (int j = 0; j < M; ++j)
            if ((t.node & c_on [j]) == c_on [j])
                if ((t.node & c_off [j]) == 0)
                {
                    nxt = t.node | r_on [j];
                    nxt = nxt & ~(r_off [j]);
                    if (best [nxt] > t.cost + cost [j])
                    {
                        best [nxt] = t.cost + cost [j];
                        pq.push ((state) {best [nxt], nxt});
                    }
                }
    }
    return -1;
}
int main ()
{
    read ();
    cout << solve () << endl;
    return 0;
}
```

## Задача 5. Сватбени торти

**Автор: Стефан Аврамов**

Първо трябва да се запитаме кога можем да направим торта от определени блатове (използвайки ги всичките)? Това е винаги възможно, когато блатовете са различни. Сега ще докажем, че минималния брой торти, които можем да направим от дадените блатове, е броя на максималния брой еднакви блатове. Нека блата, който се повтаря максимално пъти е X и се повтаря Y пъти. По-малък брой не е възможен, защото в поне една торта ще има два еднакви блата X. От повече няма смисъл. Ако са повече следователно поне в една торта няма от блата X, а има точно Y торти, в които го има този блат. От тортата без X можем да вземем произволен блат и да го сложим в една от тортите с блат X (сигурни сме, че това е възможно защото всички останали блатове се повтарят Y или по-малко пъти).

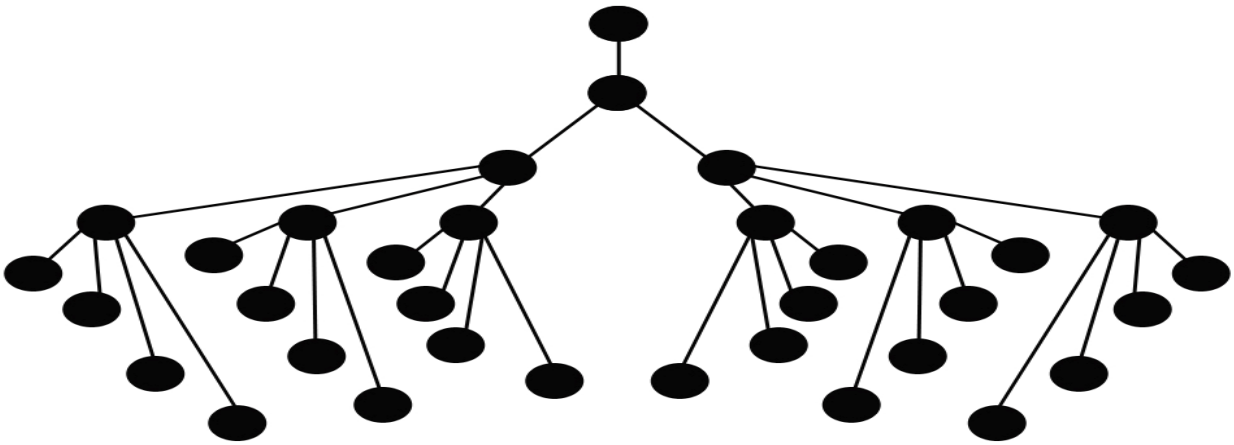
Остава да наредим блатовете, така че да съставим максимално балансирани торти. Просто сортираме блатовете и ги слагаме последователно по тортите (като стигнем до последната торта продалжаваме от първата). Така сме сигурни че в една торта няма да има два еднакви блата и накрая ще ни останат (% Y) торти с (/ + 1) брой блатове, а всички останали торти ще са с един блат по-малко. Ако е възможно сбора от разликите между съседните блатове става нула, а иначе следващото най-добро - единица. Сложността на алгоритъма е O(N*logN) поради сортирането.

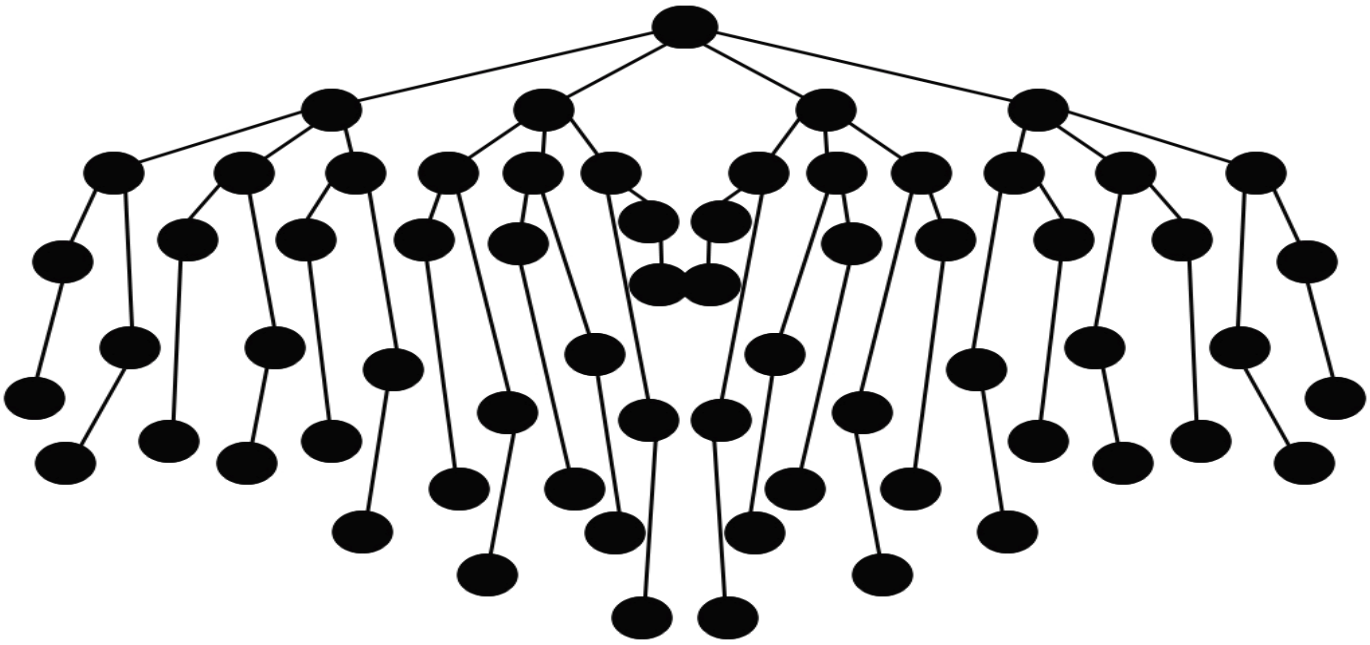
## Задача 6. Рибари

**Автор: Иван Георгиев**

Задачата предоставя възможност за реализация на различни решения, базирани на някои свойства на проблема. Тук ще представим пълно изчерпване с оптимизации.
Очевидно задачата ни е да проверим дали могат да се намерят разбивания на числата на множители от 1 до 100 такива, че нито един множител да не се среща в повече от 1 число. За целта можем просто да генерираме всички възможни разбивания на въведените числа и да започнем да се опитваме да ги комбинираме. Това можем да направим например по следния начин: използваме рекурсивна функция f с един параметър index, която се опитва да постави числото с номер index от въведените. Граничният случай е когато index сочи към елемента след последния – това означава, че успешно сме поставили всички числа и можем да терминираме рекурсията с положителен отговор. В противен случай при извикване функцията обхожда разбиванията на текущото числото и за всяко разбиване проверява дали някой от множителите вече е използван – ако това е вярно, преминаваме на следващото разбиване. В противен случай отбелязваме всички множители от текущото разбиване за използвани и извикваме f(index + 1) (при връщането, разбира се, отбелязваме всички множители като неизползвани, за да знаем, че вече могат да се използват отново). Ако рекурсията завърши без съобщение за успех, разбиване с търсените свойства не съществува.
Дори в този си вид този алгоритъм работи вярно и за време за всички тестови примери, тъй като ограниченията за входни данни са сравнително ниски, а тези за време – доста високи(time limit-a е 10 секунди). Видимо забавяне се забелязва на деветия тест. Затова предлагаме следната оптимизация, която намалява неколкократно времето за работа и е приложима при по-големи ограничения, а също така може да се използва и в други задачи с пълно изчерпване, тъй като не се базира на някакво специфично свойство на задачата, а по-скоро на свойство на дървото на рекурсията.
В нашата рекурсивна функция на всеки ред имаме различен брой излизащи ребра от текущия връх – броят им е равен на броя разбивания на числото, което разглеждаме. Нека означим този брой с $a_1$, …, $a_N$ съответно за първото, ..., N-тото число. Броят на листата на дървото е равен на $\prod a_i$, $1 <= i <= N$ (всъщност това се постига само в най-лошия случай – при нашата реализация това е случая, в който

всички числа до N-1-то могат да се комбинират с всяко от групиранията, а за последното няма начин да се добави). Броят на всички върхове на дървото обаче е доста променлив – той зависи от реда, в който се въвеждат числата и съответно реда, в който се разклонява дървото. Малко по-математически, броят им е равен на 1 + a1 + a1*a2 + a1*a2*a3 + … + a1*a2*…*aN. Очевидно, ако a1 е много голямо, например най-голямото число от всички, всички събираеми без първото ще са умножени по много голямо число, което ще увеличи броят на върховете в дървото. С аналогични разсъждения за a2, …, aN се достига до извода, че тази сума е минимална, когато числата са сортирани в нарастващ ред, и максимална, когато са сортирани в намаляващ. Следователно бихме спечелили много време, ако при въвеждането на числата ги сортираме в нарастващ ред по брой разбивания.
Следват илюстрации на дърво на рекурсия за числа с 1, 2, 3, 4 разбивания съответно в този и обратния ред за по-реална представа за разликата в големината на дърветата.





РЕАЛИЗАЦИЯ:
В началото на конкретната реализация се използва динамично оптимиране, аналогично на това при задачата за раницата – просто е за произведение. Данните от него се използват по-късно за сортировката и за по-бързо намиране на разбиванията на числата.
Рекурсивната функция gen() се използва за генериране на разбиванията на числата. idx показва индекса на числото, чийто разбивания генерираме в момента, num показва стойността на числото, което остава да разбием, а last – последното число, което сме поставили до момента в разбиването, благодарение на което можем да генерираме само намаляващи редици(може и само нарастващи, просто трябва да има някакъв критерий, който да осигурява уникалността им – няма смисъл да проверяваме едно и също разбиване по няколко пъти). Граничният случай е при num = 1 – няма какво повече да разбиваме, затова просто добавяме текущата редица към разбиванията на числото с индекс idx. В противен случай извикваме рекурсията за всички делители на num, които са по-малки от last.
Рекурсивната функция rec() върши точно описаното в анализа.

```
#include <cstdio>
#include <algorithm>
#include <vector>

int N, M;
int dp[5010];
int from[5010][25];
int top[5010];
int num[10];
std :: vector<int> w[10][150], cur;
int count[10];

struct cmp
{
    inline bool operator() (const int &a, const int &b) const
    {
        return dp[a] < dp[b];
    }
};

void gen(int idx, int num, int last)
{
    if(num == 1)    { w[idx][ count[idx] ++ ] = cur;    return;    }
    for(int i = 0; i < top[num]; i ++)
        if( from[num][i] < last )
        {
            cur.push_back(from[num][i]);
            gen(idx, num / from[num][i], from[num][i]);
            cur.pop_back();
        }
}
```

```
bool used[105];
bool fl;
void rec(int idx)
{
    if(idx == N)    fl = 1;
    if(fl)            return;
    for(int i = 0; i < count[idx]; i ++)
    {
        int to = w[idx][i].size();
        for(int j = 0; j < to; j ++)
            if( used[ w[idx][i][j] ] == 0 )    used[ w[idx][i][j] ] = 1;
            else                               to = j;
        if(to == w[idx][i].size())    rec(idx + 1);
        for(int j = 0; j < to; j ++)        used[ w[idx][i][j] ] = 0;
    }
}

int main()
{
    dp[1] = 1;
    for(int i = 2; i <= 100; i ++)
        for(int j = 5000; j > 0; j --)
            if( i * j <= 5000 && dp[j] )
                dp[i * j] += dp[j], from[i * j][ top[i * j] ++ ] = i;
    scanf("%d%d", &N, &M);
    for(int i = 0; i < M; i ++)
    {
        for(int j = 0; j < N; j ++)
            scanf("%d", &num[j]);
        std :: sort(num, num + N);
        memset(count, 0, sizeof(count));
        for(int j = 0; j < N; j ++)    gen(j, num[j], num[j] + 1);
        fl = 0;
        rec(0);
        printf("%s\n", fl ? "YES" : "NO");
    }
    return 0;
}
```