Анализ на задачите от състезанието по програмиране на **19 ноември 2011 г.**

A. Прости числа 2

Алгоритъм на Gries/Misra за намиране на проси числа 2

B. Най–близките две точки 4

C. Опознай Родината си! 6

Алгоритъм на Дийкстра 6

Представяне на графа чрез списък на съседите му 6

Реализация на алгоритъма на Дийкстра с приоритетна опашка 8

D. MODEX 10

Бързо повдигане в степен по модул на C 11

Бързо повдигане в степен по модул на Java 11

E. Охлюв 12

F. Анаграма 13

G. Ограничени суми 13

H. Редица 13

I. Цифри 14

J. Последния - първи 14

A. Прости числа

(Велислав Николов)

В задачи, подобни на тази се използва подхода ***precomputing***. Идеята е **веднъж, в началото** на програмата да генерираме и запазим всички възможни отговори, за които очакваме да бъдем питани, след което за всяка заявка да извеждаме отговора с константна или логаритмична сложност. За да можем да отговаряме на заявките, предварително трябва да намерим всички прости числа в интервала **[2; 10^8]**. След което за повечето от тях ще прилагаме различни модификации на двоично търсене **върху вече намерените прости числа** (с логаритмична сложност), а за заявката за най-малък прост делител ще отговаряме с константна сложност. Важно е да се спомене, че в процеса на търсенето на простите числа и запазването им в масив те са **сортирани в нарастващ ред**, което ни позволява да използваме двоичното търсене.

Алгоритъм на Gries/Misra за намиране на проси числа

Ще разгледаме сравнително прост алгоритъм за намиране на всички прости числа в интервала **[2;N]**. Най-известния алгоритъм за тези цел е не без известния алгоритъм на Евклид. Той обаче има сложност **O(NloglogN)**. Съществува линеен алгоритъм, който е над два пъти по-бърз, но негов недостатък е че заема памет от порядъка на **N**, което го прави неприложим за **N > 10^8**. Въпреки това той е особено полезен и поради един свой „страничен ефект" – факторизация на всички числа в интервала **[2;N]**, което може да бъде полезно в някои задачи.

Описание на алгоритъма

Нека **lp[]** е масив, инициализиран с нули, в който за всяко **i** в интервала **[2;N]** ще пазим неговия **най-малък прост делител**, а намерените до момента прости числа ще пазим в масива **pr[]**.

За всяко **i** от **2** до **N** разглеждаме следните два случая:

- **lp[i] == 0** – това означава, че **i** е просто, следователно няма други делители, а оттук **lp[i] = i**, след което добавяме **i** в края на **pr[]**.
- **lp[i] ≠ 0** – това означава, че **i** е съставно и неговия най-малък прост делител се явява **lp[i]**.

За всяко просто число p_j **< lp[i]** е вярно че $x_j = i * p_j$, т.е. **lp[**x_j**] = p[i]**. Използвайки това, запълваме всички елементи на **lp[]**.

Следва реализация на описания алгоритъм:

```
for (int i = 2; i <= N; i++)
{
	if (lp[i] == 0)
	{
		lp[i] = i;
		pr.push_back (i);
	}

	for (int j = 0; j < pr.size() && pr[j] <= lp[i] && i * pr[j] <= N; j++)
		lp[i * pr[j]] = pr[j];
}
```

i = **2, primes(2)**

2	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
2	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

i = **3, primes(2, 3)**

2	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
2	3	2	0	2	0	0	3	0	0	0	0	0	0	0	0	0	0	0

i = **4, primes(2, 3)**

2	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
2	3	2	0	2	0	2	3	0	0	0	0	0	0	0	0	0	0	0

i = **5, primes(2, 3, 5)**

2	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
2	3	2	5	2	0	2	3	2	0	0	0	0	3	0	0	0	0	0

i = 6**, primes(2, 3, 5)**

2	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
2	3	2	5	2	0	2	3	2	0	2	0	0	0	0	0	0	0	0

i = 7**, primes(2, 3, 5, 7)**

2	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
2	3	2	5	2	7	2	3	2	0	2	0	2	3	0	0	0	0	0

i = 8**, primes(2, 3, 5, 7)**

2	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
2	3	2	5	2	7	2	3	2	0	2	0	2	3	2	0	0	0	0

i = 9**, primes(2, 3, 5, 7)**

2	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
2	3	2	5	2	7	2	3	2	0	2	0	2	3	2	0	2	0	2

i = 10**, primes(2, 3, 5, 7)**

2	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
2	3	2	5	2	7	2	3	2	0	2	0	2	3	2	0	2	0	2

```
i = 11, 13, 17, 19 primes(2, 3, 5, 7, 11, 13, 17, 19)
```

2	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
2	3	2	5	2	7	2	3	2	**11**	2	**13**	2	3	2	**17**	2	**19**	2

Връзки - http://e-maxx.ru/algo/prime_sieve_linear

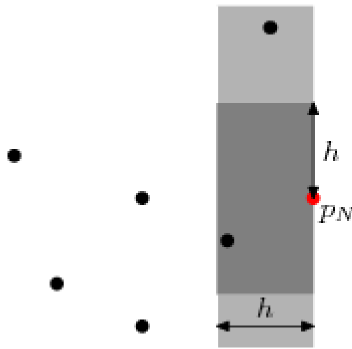
B. Най–близките две точки

(Велислав Николов)

Задачата може лесно да се с **квадратичен** алгоритъм, като изчислим разстоянията между всеки две точки и изведем най-малкото от тях, но подобен алгоритъм няма да се вмести във времевото ограничение, дори и ако приложим всички описани тук оптимизации. В разгледаната задача се търси най-голямото, а не най-малкото разстояние между зададени точки, но описаните оптимизации са валидни като цяло и следва да бъдат използвани с цел ускоряване бързодействието на програмите, които пишете. И все пак има нещо много важно - "Никога не оптимизирайте предварително, без да имате работещ алгоритъм, без да сте си изяснили решението, без да сте мислили за нещо с по-ниска алгоритмична сложност" - това (подобно формулирано) го е казал Knuth.

Задачата е добре известна алгоритмична задача, позната като **"Closest pair problem"** и за решаването ѝ съществуват няколко алгоритъма, решаващи я за **O(NlogN)**. Ще разгледаме един от тях, описан детайлно тук, както и тук.

Да предположим, че до момента сме обработили точките от **1** до **N-1** (предварително сортирани по техните x координати) и сме намерили някакво текущо най-късо разстояние **h**. Разглеждайки точка **N**, търсим друга на разстояние по-малко от **h**. Поддържаме множество от вече разгледаните точки, които са на разстояние **h** от точка **N**, както е показано в светлосивия правоъгълник. Когато преминем на следваща точка или намерим точка на разстояние по-малко от **h**, премахваме кандидатите от множеството. Тъй като тези кандидат точки трябва да са сортирани по y, и в същото време да можем с логаритмична сложност да премахваме и/или добавяме към множеството, то идеална структура за целта е **stl::set**. Търсейки точка на разстояние по-малко от **h** от текущата **N** се оказва, че има смисъл да търсим само измежду точките с **y** координати в тъмно сивия правоъгълник.




```
typedef long long ll;
typedef pair<ll, ll> pairll;
ll x, y;

pairll pnts [MAXN + 100];

double best = MAXDIST;
int compx(pairll a, pairll b) { return a.px < b.px; }

set<pairll> box;

//Sort the point by x
sort(pnts, pnts + n, compx);

//The first point is a candidate
box.insert(pnts[0]);

//When we start the left most candidate is the first one
int left = 0;

//For each point from left to right
for (int i = 1;i < n;++i)
{
	//Shrink the candidates
	while (left < i && pnts[i].px - pnts[left].px > best)
		box.erase(pnts[left++]);

	//Compute the y head and the y tail of the candidates set
	set<pairll>::iterator it = box.lower_bound(
				make_pair(pnts[i].py - best, pnts[i].px - best));

	//We take only the interesting candidates in the y axis
	for (; it != box.end() && pnts[i].py + best >= it->py; it++)
		best = min(best, dist(pnts[i], *it));

	//The current point is now a candidate
	box.insert(pnts[i]);
}

printf("%.5f\n", sqrt(best));
```

Тук може да прочетете друг алгоритъм, за решаване на задачата.

C. Опознай Родината си!

(Велислав Николов)

В тази задача търсим **най-къся път в граф** от зададен стартов връх, минаващ през други два върха.

За намирането на най-къся път в граф **от даден връх до всички останали върхове**, с **положителни цени по ребрата** се използва алгоритъма на Дийкстра. Класическата реализация използваща **матрица на съседство** е квадратична, както по памет, така и по време, спрямо броя на върховете на графа. Тази реализация е обяснена тук, както и на много други места. Всъщност при зададените ограничения (10 000 върха) не би било възможно да използваме представяне, чрез матрица на съседство.

Ние обаче ще разгледаме реализация, използваща **списък на съседите** и приоритетна опашка, което ще доведе до сложност по време**O((V + E) log V)** и **O(V + E)** по памет, където **E** е броят на ребрата, а **V** е броят на върховете. Повече информация за представяне на графи, както и някои основни алгоритми може да се намери тук.

За решаването на задачата трябва да пуснем алгоритъма на Дийкстра веднъж от P1 и веднъж от P2. Ако съществува път от P1 до PB, минаващ през P2, то това е потенциален кандидат за решение на задачата. Трябва обаче да разгледаме и втората възможност, а именно да проверим дали съществува път от P2 до PB, минаващ през P1.

- PA2 -> ... -> PA1 -> ... -> PB
- PA1 -> ... -> PA2 -> ... -> PB

Отговора на задачата е по-малкото от двете разстояния.

Алгоритъм на Дийкстра

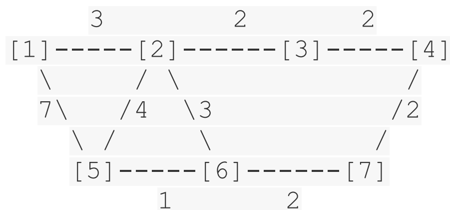
Представяне на графа чрез списък на съседите му

Списъкът на съседство представлява масив (да го кръстим **edges**) от динамични масиви, където **edges[i]** съдържа всички съседни върхове на **i**, заедно с разстоянията до тях. Това представяне се използва много често, защото **е подходящо за разредени графи**, тъй като използва по-малко памет от матрицата на съседство, макар и операциите триене и добавяне на ребро да са по-бавни. Ето как би изглеждала подобна имплементация:

```
#define MAX 100005

#define MP(x, y) make_pair((x), (y))
```

```
typedef pair<int, int> ii;
vector <ii> edges[MAX]; // списък на съседите на i-тия връх
int V, E, from, to, cost;

int main()
{
	scanf("%d %d\n", &V, &E);
	for(int i = 0;i < V;i++)
	{
		scanf("%d %d %d\n", &from, &to, &cost);
		from--; to--;

		edges[from].push_back(MP(to, cost));
		edges[to].push_back(MP(from, cost)); //ако графа е двойно свързан
	}
	return 0;
}
```

Да разгледаме един примерен граф.



Ето как би изглеждал един примерен вход, който трябва да прочетем:

```
7 9
5 1 7
6 7 2
4 7 2
5 6 1
5 2 4
4 3 2
1 2 3
3 2 2
2 6 3
```

И представянето му, чрез списък на съседите:

edges[**0**]: (**4**, 7) (**1**, 3)

edges[**1**]: (**4**, 4) (**0**, 3) (**2**, 2) (**5**, 3)

edges[**2**]: (**3**, 2) (**1**, 2)

edges[**3**]: (**6**, 2) (**2**, 2)

edges[**4**]: (**0**, 7) (**5**, 1) (**1**, 4)

edges[**5**]: (**6**, 2) (**4**, 1) (**1**, 3)

edges[**6**]: (**5**, 2) (**3**, 2)

Реализация на алгоритъма на Дийкстра с приоритетна опашка

За реализацията ще са ни нужни следните структури:

```
typedef pair<int, int> ii;

vector <ii> edges[MAX]; // списък на съседите на i-тия връх
vector <int> dist; // най-късото разстояние от стартовия до i-тия връх
vector <int> parent; // предшественика на i-тия връх
vector <int> vis (V, false); // дали и посищаван i-тия връх
```

Същинската част на алгоритъма:

```
void dijkstra(int sv)
{
	// инициализация
	dist.assign(V, MAXINT); // най-късия път до всеки е достатъчно дълъг
	parent.assign(V, -1);   // никой няма предшественици
	vis.assign(V, false);   // никой не е посетен

	int i, from, to, cost;

	dist[sv] = 0; //разстоянието до стартовия връх маркираме с 0
	parent[sv] = -1; // стартовия връх няма предшественик

	// приоритетна опашка, елемента с най-малка стойност се намира на
	// върха й. В нея постепенно ще поставяме всеки достижим до момента
	// връх, както и разстоянието до него. Важно е да се отбележи, че всеки
	// връх се разглежда най-много веднъж в процеса на обхождане!
	priority_queue<ii, vector<ii>, greater<ii> > pq;

	// поставяме в опашката двойката стойности
	// (намереното до момента най-късо разстояние до sv, самото sv)
	pq.push(MP(dist[sv], sv));

	// докато опашката не е празна, т.е. докато има необходени върхове
	while (!pq.empty())
	{
		// взимаме елемента на върха на опашката, той става
		// текущия връх в който се намираме
		from = pq.top().second;
		pq.pop(); // вадим го от опашката

		if (vis[from]) // проверяваме дали не сме го посещавали вече
			continue;

		vis[from] = true; // маркираме го като посетен

		// за всеки съсед на текущия връх
		for(i = 0; i < edges[from].size(); i++)
		{
			to = edges[from][i].first; // поредния му съсед
			cost = edges[from][i].second; // разстоянието до него
```

```
                    // проверяваме дали разстоянието до поредния
                    // съсед на текущия връх може да бъде подобрено
                    if (dist[from] + cost < dist[to])
                    {
                          // актуализираме разстоянието
                          dist[to] = dist[from] + cost;

                          // актуализираме предшественика му
                          parent[to] = from;

                          // поставяме в опашката двойката (намереното
                          // до момента най-късо разстояние до to, самото to)
                          pq.push(MP(dist[to], to));
                    }
              }
       }
}
```

След извикването

```
int sv = 5;
dijkstra(sv - 1);
```

състоянието на масива **dist**, съдържащ най-късите пътища от стартов връх 5 (върховете са индексирани от **0**!) до всички останали върхове е:

0	1	2	3	4	5	6
7	**4**	**6**	**5**	**0**	**1**	**3**

Състоянието на масива **parent**, съдържащ върха от който сме стигнали до **i**:

0	1	2	3	4	5	6
4	**4**	**1**	**6**	**-1**	**4**	**5**

Остава да намерим и конкретните пътища от стартовия връх, а това може да стане със следните помощни функции:

```
void printPath(int s, int j)
{
      if (parent[j] != s)
            printPath(s, parent[j]);
      printf("%d ", j + 1);
}

void printResult(int s)
{
      for (int i = 0; i < V; i++)
      {
            if (dist[i] != MAXINT && i != s)
            {
                  printf("%d -> %d (%d): ", s + 1, i + 1, dist[i]);
```

```
                        printf("%d ", s + 1);
                        printPath(s, i);
                        printf("\n");
                }
        }
}
```

Ето и резултата от извикването на `printResult(sv - 1)`:

5 -> 1 (7): 5 1
5 -> 2 (4): 5 2
5 -> 3 (6): 5 2 3
5 -> 4 (5): 5 6 7 4
5 -> 6 (1): 5 6
5 -> 7 (3): 5 6 7

След като човек е наясно с алгоритъма на Дийкстра лесно може да го използва в конкретната задача.

Допълнително усложнение в задачата е че теглата на ребрата са зададени в някаква бройна система K ($2 \le K \le 36$). Отговора на всеки тестов пример също трябва да бъде изведен в зададена бройна система. Не е голям проблем сами да си напишем подобни функции, но с цел да си спестим време можем да използваме стандартните **strtol** и **itoa**, по подобен начин:

```
char *buff = new char[50];

//прочитаме разстоянието между върховете като низ
scanf("%d %d %d %s\n", &u, &v, &inpBase, buff);

//конвертираме низа в 10-тична бройна система, указвайки бройната система в която е
записано числото
c = strtol (buff, &buff, inpBase);

//извеждаме резултата в outBase бройна система
printf("%s\n", itoa(solve(), buff, outBase));
```

Задачата (без усложняването с бройните системи) е давана на бронзовата дивизия на USACO през Декември 2010.

Условие (Apple Delivery) – http://tjsct.wikidot.com/usaco-dec10-silver

Анализ - http://ace.delos.com/TESTDATA/DEC10.apple.htm

D. MODEX

(Велислав Николов)

Задачата изисква **бързо** повдигане на степен по модул. На пръв поглед изглежда, че ще са ни нужни големи числа, но при поставените ограничения за числата, тя може да се реши и без такива.

За разлика от стандартното повдигане в степен, изискващо **N** умножения, алгоритъма за бързо повдигане в степен изисква **logN** умножения - http://e-maxx.ru/algo/binary_pow.

Бързо повдигане в степен по модул на C

```
#include <stdio.h>

typedef unsigned long long ull;
ull x, y, n;

ull powMod(ull a, ull n, ull m) //(a ^ n) % m
{
	ull f = 1;
	while(n)
	{
		if(n & 1)
			f = (f * a) % m;
		n >>= 1;
		a = (a * a) % m;
	}
	return f % m;
}

int main()
{
	int t;
	scanf("%d\n", &t);
	while(t--)
	{
		scanf("%llu %llu %llu\n", &x, &y, &n);
		printf("%llu\n", powMod(x, y, n));
	}
	return 0;
}
```

Все пак ако ограниченията за числата надхвърляха стандартните типове в C/C++, то имаме две възможностти:

1. Да си напишем сами клас за работа с дълги числа на C/C++.
2. Да използваме функционалността на класа **BigInteger** или **BigDouble** в Java.

Втората е за предпочитане, тъй като писането на наш собствен клас за работа с големи числа, работещ коректно, ще отнеме повече време от използването на вече написания **BigInteger**, който работи коректно и метода му ***modpow*** имплементира алгоритъма за бързо повдигане в степен по модул.

Бързо повдигане в степен по модул на Java

```
import java.math.BigInteger;
import java.util.Scanner;
```

```
public class Program
{
	public static void main(String[] args)
	{
		Scanner s = new Scanner(System.in);
		int t = s.nextInt();

		while (t-- != 0)
		{
			BigInteger x = BigInteger.valueOf(s.nextInt());
			BigInteger y = BigInteger.valueOf(s.nextInt());
			BigInteger n = BigInteger.valueOf(s.nextInt());
			System.out.println(x.modPow(y, n));
		}
	}
}
```

E. Охлюв

(Велислав Николов)

Итеративното изчисляване на позицията на охлюва ден след ден би било крайно неефективно и би получило **TL**.

За едно денонощие охлюва изкачва точно **B – A** метра. Цялото му пътуване трае **X** дена и **X** нощи, плюс още един ден за достигане на самия връх. Следователно се интересуваме от най-малкото цяло число **X**, такова че **X * (A - B) + A ≥ V**. Отговора на задачата е **X + 1**. Ето и как намираме X:

X * (A - B) + A ≥ V

X * (A - B) ≥ V - A

X = (V - A) / (A - B)

X = (V - A + A - B - 1) / (A - B)

X = (V - B - 1) / (A - B)

Алтернативно решение

Може да се използва двоично търсене, започвайки от някакво X и проверяваме дали сме стигнали върха:

```
int lo = 1, hi = v;
while( lo < hi )
{
	long long mid = ( lo+hi ) / 2;
```

```
        if(mid * a - (mid - 1) * b >= v )
               hi = mid;
        else
               lo = mid+1;
}

printf( "%d\n", lo );
```

Сложност **O(log v)**.

F. Анаграма

(Велислав Николов)

Задачата се решава много лесно, ако за всеки тестов пример сортираме всеки низ и го поставяме в обект от тип std::set<string>. Това ни гарантира, че всеки низ ще се съдържа само веднъж. Очевидно отговора за всеки тестов пример и броя на елементите в дървото.

G. Ограничени суми

(Емил Келеведжиев)

Използваме масива b[i][j], за да пресмятаме колко са редиците в които участват i числа със сума по-малка от j.

Лесно се съобразяват стойностите на b[i][j] за редици от едно число (т.е. при i=1). Тогава b[1][j] е равно на 1 или на 0, в зависимост от това, дали j е по-малко от p или не е.

Полагаме b[i][0]=1 за всяко i, защото само една редица има нулева сума: 0+0+...+0 с i събираеми.

Предполагаме, че за всички разглеждани стойности на j, вече сме пресметнали b[1][j], b[2][j], ... b[i-1][j]. За да пресметнем b[i][j], разглеждаме редица от i елемента. На последното място в тази редица може да стои всяко число k, което събрано със сумата на числата от предните елементи дава стойност по-малка от j. При всеки такъв избор имаме b[i-1][j-k] възможности. Затова b[i][j] е сумата от броя на редиците с i-1 елемента при тези случаи.

Накрая трябва да съберем всички бройки на редици от n елемента, които имат суми 0, 1, 2, ..., s-1.

H. Редица

(Емил Келеведжиев)

Изполваме масива b[i] за пресмятане на търсения брой редици с дължина i. Непосредствено може да преброим, че b[1]=2, b[2]=4 и b[3]=7.

Предполагаме, че сме пресметнали стойности b[1], b[2], ..., b[i-1].

Разглеждаме множеството на всички редици от описания вид. Ако последният елемент е 0, на мястото на предишните елементи може да стои всяка редици от описания вид, т.е. броят им е b[i-1]. Ако последният елемент е 1, а предпоследният е 0, това значи, че броят на предишните редици е b[i-2]. Ако последният и предпоследния елементи са 1, то предпредпоследният елемент е 0 и това значи, че броят на предипните редици е b[i-3]. Понеже с това се изчерпват всички възможности, броят b[i] е сумата b[i-1]+b[i-2]+b[i-3].

Така намирането на всички b[i] става с последователно итеративно пресмятане.

I. Цифри

(Емил Келеведжиев)

Нека дължината на дадения низ е n. Следва, че броят на низовете от разглеждания вид е n-1. В тези низове се срещат всички цифри на дадения низ точно по n-1 пъти. Следователно отговорът на задачата се получва като пресметнем сумата от цифрите на дадения низ и резултата умножим по n-1.

J. Последния - първи

(Николай Киров)

Задачата е чисто техническа, зададените размерности не изискват използването на специални алгоритми. Има малка трудност при разбирането на задачата и при организирането на входа. За сортировките се използва функцията sort от STL.