



Идеи за решаване на задачите

Задача 1. Дявол

Тази задача е предложена от Николина Николова,
ФМИ на СУ „Св. Климент Охридски“

Ето пример на „речешка задачка“ – такава, дето се решава отзад-напред. Ако означим парите преди последното преминаване на моста с B_k , то след преминаването остават $2 \cdot B_k - A = 0$. Значи $B_k = A/2$. и това са парите останали след предпоследното преминаване (с номер $K - 1$). По аналогичен начин от $2 \cdot B_{k-1} - A = B_k$ определяме $B_{k-1} = (B_k + A)/2$. Продължаваме в същия дух за всяко предходно преминаване докато не стигнем до първото и определим $B_1 = (B_2 + A)/2$, което е и отговорът на задачката.

Леле! – написах 6 реда (!!!) обяснение на нещо, което и без друго е очевидно ;-)

Кодът на програмата, написан на езика C++

```
#include <iostream>
using namespace std;

int main()
{
    int a, k, b=0; //b - current sum; at the end it is 0
    cin >> a >> k;
    for (int i=0; i<k;i++)
        b=(b + a)/2;
    cout << b << endl;
    return 0;
}
```

Задача 2. Преследване

Тази задача е предложена от Евгений Василев,
катедра Информатика, НПМГ „Акад. Л. Чакалов“ – София

Да разгледаме ситуацията, когато i -я колоездач настига $i+1$ – вия за време t . Тогава изминатите пътища от двамата са съответно $s_i = t \cdot v_i$ и $s_{i+1} = t \cdot v_{i+1}$. Знаем, че $s_i - s_{i+1} = L/N$. Тогава $t \cdot v_i - t \cdot v_{i+1} = L/N$ или $t = L/N / (v_i - v_{i+1})$. Времето t ще е толкова по-малко колкото е по-голямо ($v_i - v_{i+1}$) или казано с други думи победител е този, за който разликата в скоростите с предния е максимална! Остава да намерим тази максимална разлика, да изчислим времето на победителя и после изминатия от него път. Да не забравим, че ако всички колоездачи имат еднаква скорост на движение, никой не би настигнал предния (горките спортисти: вечно ще въртят педалите ☹) – няма победител.

Кодът на програмата, написан на езика C++

```
include <iostream>
#define MAXN 10
using namespace std;

int main () {
    int L, n, V[MAXN+1], i, dV, dVmax=0, winner=0;
    double t, s;
```



```
cin >> L >> n;
for (i=0; i < n; i++) cin >> V[i];
V[n]=V[0];

for (i=0; i<n; i++) if ((dV=V[i]-V[i+1])>dVmax) dVmax=dV, winner=i;

if (dVmax) {
    t = (double)L/n/dVmax;
    s = V[winner]*t;
    cout << winner+1 << " " << t << " " << s << endl;
}
else cout << "NO WINNER\n";
return 0;
}
```

Задача 3. Пиле по императорски

Тази задача е предложена от Кристиан Цъклев,
8 клас, СМГ „Паусий Хилендарски”,
Разширен национален отбор по информатика (младша възраст) - 2010

Прочитаме N и теглата на кубче от отделните продукти. След това четем символ по символ описанието на поредното стълбче с кубчета. Трябва да се съобрази, че Светльо премества кубчета в ред обратен на реда в описанието им: първите символи от описанието, които са различни от p означават кубчета, които не се местят (намират се под най-долното фъстъчено кубче). Дали да се сумира или не теглото на поредното кубче се управлява с помощта на булевата променлива *peanut*. В началото на всяко описание на стълбче *peanut* се установява във false, което означава „непреместваеми кубчета, не сумирай”, а първото срещнато p в описанието я установява в true, което означава „преместваеми кубчета – сумирай”. След като повторим горното за всички стълбчета, извеждаме сумата на теглата на всички преместени кубчета.

Амиии ... това е! Някакви въпроси?

Кодът на програмата, написан на езика C++

```
#include<iostream>
using namespace std;

int n,m;
char cube;
long long gram,food[5];

int main()
{
    cin>>n;
    for(int i=0;i<5;i=i+1)
    {
        cin>>food[i];
    }
    for(int i=0;i<n;i=i+1)
    {
        bool peanut=false;
        cin>>cube;
        while(cube!='a')
        {
```



```
if (cube=='p') {peanut=true; gram=gram+food[4];}  
if (peanut)  
{  
    if (cube=='b') {gram=gram+food[0];}  
    if (cube=='c') {gram=gram+food[1];}  
    if (cube=='C') {gram=gram+food[2];}  
    if (cube=='m') {gram=gram+food[3];}  
}  
cin>>cube;  
}  
}  
cout<<gram<<"\n";  
return 0;  
}
```

Задача 4. Lidl

Тази задача е предложена от Христо Стоянов,
8 клас, СМГ „Паусий Хилендарски”,
Национален отбор по информатика (младша възраст) - 2010

От алгоритмична гледна точка задачата не представлява трудност. Алгоритъмът е даден в условието и трябва единствено да се реализира.

Решението представлява, всъщност, „нареждането” на всеки клиент на желаната опашка. За всяка опашка имаме общото количество продукти, което се е натрупало, и търсим минималното такова, като ако има повече от една опашка с минимален брой продукти взимаме тази с най-малък номер. След това прибавяме продуктите на текущия клиент към нея и продължаваме със следващия. Продуктите, които той закупува, могат да се пресметнат лесно, както се вижда в реализацията.

Кодът на програмата, написан на езика C++

```
#include <cstdio>  
  
int ans[128][1 << 10];  
int n;  
int m;  
int k;  
int payDesk[128];  
  
int main() {  
    scanf("%d %d %d", &n, &k, &m);  
    int products = 0;  
    int min;  
  
    for (int i = 0; i < n; ++i) {  
        products %= k;  
        products++;  
        min = 0;  
        for (int j = 0; j < m; ++j) {  
            if (payDesk[j] < payDesk[min]) {  
                min = j;  
            }  
        }  
        payDesk[min] += products;  
        ans[min][products]++;  
        //printf("%d %d\n", min + 1, products);  
    }  
}
```



```
}  
  
for (int i = 0; i < m; ++i) {  
    printf("%d", ans[i][1]);  
    for (int j = 2; j <= k; ++j) {  
        printf(" %d", ans[i][j]);  
    }  
    printf("\n");  
}  
  
return 0;  
}
```

В подобно решение търсенето на опашката с минимален брой продукти може да се оптимизира. Ускоряването може да се постигне като имаме константен достъп до касата с най-малко продукти. За целта ги държим сортирани в нарастващ ред и когато искаме да прибавим закупените неща от един клиент просто ги прибавяме към първата каса в масива. След това я изместваме назад, докато преди нея има единствено такива с по-малко продукти или с равен, но с по-малък номер. Тъй като ни трябва и номера на касата, което в предната реализация се осигурява от това, че ги държим подредени по номер, тук това не е така и ни трябва допълнителен масив. Съответно в него ще държим номерата на касите, но тези номера ще са подредени в зависимост от това колко продукти има на касата.

Трябва да се направи съображението, че в най-лошия случай няма да има никакво подобрение, защото може да се наложи касата да отиде от първо място, в края, което би направило търсенето отново линейно, но това е само частен случай, като във всички други скоростта би се подобрила.

Кодът на подобрения вариант на програмата, написан на езика C++

```
#include <cstdio>  
  
int ans[128][1 << 10];  
int n;  
int m;  
int k;  
int payDesk[128];  
int payDeskIndex[128];  
  
int main() {  
    scanf("%d %d %d", &n, &k, &m);  
    for (int i = 0; i < m; ++i) {  
        payDeskIndex[i] = i;  
    }  
    int products = 0;  
    int min;  
    int j;  
    int swap;  
  
    for (int i = 0; i < n; ++i) {  
        products %= k;  
        products++;  
        ans[payDeskIndex[0]][products]++;  
        payDesk[payDeskIndex[0]] += products;  
        //printf("%d %d\n", payDeskIndex[0] + 1, products);  
        j = 0;  
        bool more = payDesk[payDeskIndex[j]] > payDesk[payDeskIndex[j+1]];
```



```
more = more || (payDesk[payDeskIndex[j]] == payDesk[payDeskIndex[j+1]]
                && payDeskIndex[j] > payDeskIndex[j+1])
);
while (j < m - 1 && more) {
    swap = payDeskIndex[j];
    payDeskIndex[j] = payDeskIndex[j+1];
    payDeskIndex[j+1] = swap;
    j++;
    more = payDesk[payDeskIndex[j]] > payDesk[payDeskIndex[j+1]];
    more = more || (payDesk[payDeskIndex[j]] == payDesk[payDeskIndex[j+1]]
                    && payDeskIndex[j] > payDeskIndex[j+1])
);
}

for (int i = 0; i < m; ++i){
    printf("%d", ans[i][1]);
    for (int j = 2; j <= k; ++j) {
        printf(" %d", ans[i][j]);
    }
    printf("\n");
}

return 0;
}
```

При увеличаване на ограниченията, може да се наложи допълнително ускоряване на търсенето или оптимизация на използваната паметта.

Съществува структура от данни, която в STL е реализирана в `priority_queue`, с чиято помощ можем да постигнем $\log_2 m$ скорост на промяна на количеството продукти в касата, като запазваме константния достъп до тази, с най-малко продукти.

В горните две решения може да се появи проблемът, че таблицата с отговора е твърде голяма и заема много памет, като голяма част от нея са нули. За решение на тази пречка може да се използва `map`. Това е структура от данни, в която на определен ключ се съпоставя дадена стойност, като не може на един и същи ключ да се съпоставят различни стойности. За ключ ползваме позицията на записа в таблицата. Масива с отговори го заменяме със структурата и така пазим само данните, които са ни нужни, а при отсъствие на запис за дадена клетка – все едно е записана нула.