電気電子情報工学実験 II (b)
実践的・競技プログラミング 練習コンテスト（Contest 2-1）
解説

廣田悠輔 *)，横野智紀
*): y-hirota@u-fukui.ac.jp

目次

1 電源タップ（横野智紀）

解説

未使用な差し込み口 1 つを使って，新たに差し込み口を A_i 個増やすという操作は，未使用な差し込み口を $A_i - 1$ 個増やす操作と言い換えることができる．このとき，差し込み口 0 個の際に矛盾することになるが，今回は差し込み口を増やしていくことが最善であり，1 つ以上空いている状態を維持することが常に可能であることから問題にはならない．

A_i の制約は $0 \leq A_i \leq 10^4$ となっている．つまり，新たに差し込み口を増やすことのないタップ $A_i = 0$ も存在する．これについて，使うだけ損になることは明らかである．

以上より，解は次の式で求められる．

$$\sum_{i}^{N} \max(A_i - 1, 0) + 1$$

これをそのままプログラムに実装すれば良い．計算量は $O(N)$ である．

回答例

```
1 #include<stdio.h>
2
3 int main(void) {
```

```
4     int i, n, ans = 1, in;
5     scanf("%d", &n);
6     for(i = 0; i < n; i++) {
7       scanf("%d", &in);
8       in--;
9       if(in > 0) ans += in;
10    }
11    printf("%d\n", ans);
12    return 0;
13  }
```

2　ヒロタ氏の昼食（廣田悠輔）

解説

同じ一品料理を重複して食べることはできないので，候補となる N 種類の一品料理それぞれについて「食べる」,「食べない」のいずれかから選択することになる．すなわち，考えられる選択は 2^N ケースとなる．制約 $1 \leq N \leq 16$ が与えられているので，最大でも $2^{16} = 65536$ ケースだけを考えれば良い．したがって，それらの全ケースについて合計金額と合計カロリーを計算し，最低カロリー条件を満たすものの中から最低金額のものを調べれば制限時間内に回答可能である．

全探索を実装する方法を考える．一品料理の種類数 N が $N = 3$ だとか $N = 4$ のような小さな固定値の場合には，N 重ループを使って全探索をするという手段が考えられるが，この問題では N は入力として与えられその最大値も $N = 16$ と大きいので，この方法は適さない[*1]．

本問題の全探索を実装するもっとも標準的な方法の一つは二分木の深さ優先探索である．一品料理それぞれの「食べる」,「食べない」の選択を枝とする二分木を考え，2^N 個の葉それぞれにおける合計価格と合計カロリーを調べる．これはシンプルな関数の再帰呼び出しを使った二分木の深さ優先探索により実現可能である．具体的な実装は回答例を参照されたい．

別解として，今回のような単純な二分木の葉の全探索であればビット全探索と呼ばれるテクニックを使うことで再帰を使わずに全探索を実現することも可能である．参考までに回答例を示しておく．

回答例（再帰による深さ優先探索）

```
1   #include <stdio.h>
2   #include <limits.h>
3
4   #define MAX_N (20)
5
6   int dfs(int const n, int const * const p, int const * const q, int
        const m,
```

[*1] 1 重ループから 16 重ループまで全パタンの関数を作成する，あるいは 16 重ループ内に分岐を付け加えるといった方法で $1 \leq N \leq 16$ に対応した全探索の実装は不可能ではないが，あまり良い考えではない．

```
7           int const sum_price, int const sum_kcal) {
8   int price_nouse_this, price_use_this;
9
10  if (n == 0) {
11    if (sum_kcal >= m) {
12      return sum_price;
13    } else {
14      return INT_MAX;
15    }
16  }
17
18  price_nouse_this = dfs(n - 1, &p[1], &q[1], m, sum_price, sum_kcal);
19  price_use_this = dfs(n - 1, &p[1], &q[1], m, sum_price + p[0],
        sum_kcal + q[0]);
20
21  return (price_nouse_this < price_use_this) ? price_nouse_this :
        price_use_this;
22 }
23
24 int main(void) {
25  int n, m, p[MAX_N], q[MAX_N];
26  int i;
27  int min_price;
28
29  scanf("%d", &n);
30  for (i = 0; i < n; ++i) {
31    scanf("%d␣%d", &p[i], &q[i]);
32  }
33  scanf("%d", &m);
34
35  min_price = dfs(n, p, q, m, 0, 0);
36
37  if (min_price == INT_MAX) {
38    /* No feasible selection */
39    printf("%d\n", -1);
40  } else {
41    /* Best selection */
42    printf("%d\n", min_price);
43  }
44
```

```
45    return 0;
46  }
```

回答例（ビット全探索）

```
1   #include <stdio.h>
2   #include <limits.h>
3
4   #define MAX_N (20)
5
6   int main(void) {
7     int n, m, p[MAX_N], q[MAX_N];
8     int i;
9     unsigned long bits;
10    int min_price;
11
12    scanf("%d", &n);
13    for (i = 0; i < n; ++i) {
14      scanf("%d␣%d", &p[i], &q[i]);
15    }
16    scanf("%d", &m);
17
18    /* Find the best possible selection. */
19    min_price = INT_MAX;
20    for (bits = 0; bits < (1u << n); ++bits) {
21      int sum_price = 0;
22      int sum_kcal = 0;
23
24      for (i = 0; i < n; ++i) {
25        int const use_this = (bits >> i) % 2;
26        sum_price += use_this * p[i];
27        sum_kcal += use_this * q[i];
28      }
29
30      if (sum_kcal >= m && sum_price < min_price) {
31        min_price = sum_price;
32      }
33    }
34
35    if (min_price == INT_MAX) {
```

```
36      /* No feasible selection */
37      printf("%d\n", -1);
38    } else {
39      /* Best selection */
40      printf("%d\n", min_price);
41    }
42
43    return 0;
44  }
```

3　友達 (の友達)+（横野智紀）

解説

基本的なグラフ探索の問題である．特に注意すべきことは，N が最大で 10^5 であることと，M が最大で 10^5 であることである．グラフ表現には，二次元配列を用いる隣接行列表現と，リスト構造を用いる隣接リスト表現がある．隣接行列表現は実装が容易である一方，頂点数を N として $O(N^2)$ のメモリが必要である他，ある頂点が連結している辺の数を X として，それらの列挙は連結判定が必要になることから $O(N)$ かかってしまう（更に，基本的に多重辺については表現が不可能である）．密なグラフにおいては有効であるが，今回のような疎なグラフにおいては向いていない．実際，隣接行列表現での実装では実行制限時間や実行制限メモリ量を超過することになる．

無駄なくグラフを表現する際には隣接リスト表現が有効である．メモリ量については，頂点数を N，辺の総数を M として $O(N+M)$ のメモリで十分となり，今回の制約において適している．ある頂点が連結している辺の数を X として，それらの列挙は $O(X)$ で可能である．

ある人の頂点 Z に連結している頂点数を数える際には，幅優先探索 (BFS) や深さ優先探索 (DFS) を用いることにより $O(N)$ で探索が可能である．実装例では深さ優先探索を用いて実装している．再帰関数は一見難しく見えるが，こういった深さ優先探索を行う際には少ない実装量で実装することが可能になる．なお，競技プログラミング特有のテクニックとして，再帰関数の引数を丁寧に書くというのは実装の手間が増えて時間がかかってしまうため，グローバル変数を活用することで記述量を減らすことができる．

頂点 Z に連結している頂点数を探索によって求めた後，頂点 Z に直接つながっている頂点数を引き，最後に自分自身（つまり 1）を引くことで解を求めることができる．計算量は探索及び辺の入力がボトルネックとなって $O(N+M)$ である．

この問題においてはリストのデータ構造が必須である．競技プログラミングにおいてリストや連想配列といった高度なデータ構造を要求されることは多々あるため，ライブラリを整備するか，C++ や Java といった多機能な言語を使用することを推奨する．

回答例

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #define N 100005
```

```
4
5  /*リスト一式*/
6  typedef struct listElement{
7    int data;
8    struct listElement *next;
9    struct listElement *prev;
10 } LIEL;
11 typedef struct list{
12   int size;
13   LIEL *front;
14   LIEL *back;
15 } LIST;
16 LIEL *new_listElement(int d){
17   LIEL *ret = (LIEL *)malloc(sizeof(LIEL));
18   ret->data = d;
19   ret->next = NULL;
20   ret->prev = NULL;
21   return ret;
22 }
23 LIST *new_list(void){
24   LIST *ret = (LIST *)malloc(sizeof(LIST));
25   ret->back = NULL;
26   ret->front = NULL;
27   ret->size = 0;
28   return ret;
29 }
30 void push_back(int d, LIST *list){
31   LIEL *new = new_listElement(d);
32   if(list->size == 0){
33     list->front = list->back = new;
34   }else{
35     list->back->next = new;
36     new->prev = list->back;
37     list->back = new;
38   }
39   list->size++;
40 }
41
42 LIST *list[N];
43 int flag[N];
```

```
44
45 int dfs(int now) {
46   int res = 1;
47   LIEL *itr;
48   for(itr = list[now]->front; itr != NULL; itr = itr->next) {
49     if(flag[itr->data] == 1) continue; /* チェック済みなら探索しない */
50     flag[itr->data] = 1; /* チェックを付けてから次の探索へ */
51     res += dfs(itr->data);
52   }
53   return res;
54 }
55
56 int main(void) {
57   int i, a, b;
58   int n, m, z;
59   scanf("%d␣%d␣%d", &n, &m, &z);
60   z--; /* 0-based indexing に直している */
61   for(i = 0; i < n; i++) {
62     list[i] = new_list(); /* リストの初期化 */
63   }
64   for(i = 0; i < n; i++) {
65     flag[i] = 0; /* フラグの初期化 */
66   }
67   for(i = 0; i < m; i++) {
68     scanf("%d␣%d", &a, &b);
69     a--; b--; /* 0-based indexing に直している */
70     push_back(a, list[b]); /* b から a への辺を追加 */
71     push_back(b, list[a]); /* a から b への辺を追加 */
72   }
73
74   flag[z] = 1; /* 始点についてチェック済みにしておく */
75   int ans = dfs(z) - list[z]->size - 1;
76   printf("%d\n", ans);
77
78   return 0;
79 }
```

4　市松模様（横野智紀）

解説

一つのテストケースの中に，クエリ (出力を求められる問題) が最大で 10^5 個あることや，極めて大きい座標の制約であることから，定数時間あるいは制約に対して対数時間で求める解が必要になる．

市松模様の性質をよく観察すると，縦は $2H$，横は $2W$ で同じパターンが繰り返されていることに気づく．基準となるマスを仮に座標 $(0,0)$ とおいたとき，座標 (Y,X) の状態は座標 $(Y\%(2H), X\%(2W))$ と同じである．なお，$\%$ は剰余演算を示す．また，基準であるマスが (R,C) であるときに，座標 (Y,X) の状態を求める際には，基準が座標 $(0,0)$ とおいたときと同様な考え方を適用するために，座標をずらすことが有効である．

よって基準となるマスを座標 (R,C) とおいたとき，座標 (Y,X) の状態は座標 $((Y-R)\%(2H), (X-C)\%(2W))$ と同じである．

ただし，これをそのまま実装すると処理系によっては負の数に対して処理を行った際に適切な値を示さない．特に ANSI C においては負の数への剰余演算は未定義とされている．よって，一度負の数を 0 以上の数へ変換する必要がある．この処理は絶対値に対する切り上げの割り算と掛け算によって実装できる．

座標 $(Y\%(2H), X\%(2W))$ について，各座標が白か黒かについては，

- $Y < H$
- $X < W$

この二つの条件の排他的論理和を用いて判定が可能である．

計算量は各クエリに対して $O(1)$ であり，全体で $O(T)$ である．

こうした問題は手を動かして観察することで，性質を見つけ出すことができる．サンプルケースを図に書くことや，自力でサンプルケースを作ることは競技プログラミングにおいて有効な手法となる．

回答例

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int fc(int p, int m) {
5    if(p < 0) {
6      p += (abs(p)+m-1) / m * m;
7    }
8    p %= m;
9    return p;
10 }
11
12 int main(void) {
13   int h, w, r, c, t, i;
14   int x, y;
15   scanf("%d␣%d␣%d␣%d␣%d", &h, &w, &r, &c, &t);
```

```
16    for(i = 0; i < t; i++) {
17      scanf("%d␣%d", &y, &x);
18      y -= r;
19      x -= c;
20      y = fc(y, 2*h);
21      x = fc(x, 2*w);
22      if((y < h) ^ (x < w)) printf("%s\n", "White");
23      else printf("%s\n", "Black");
24    }
25    return 0;
26  }
```